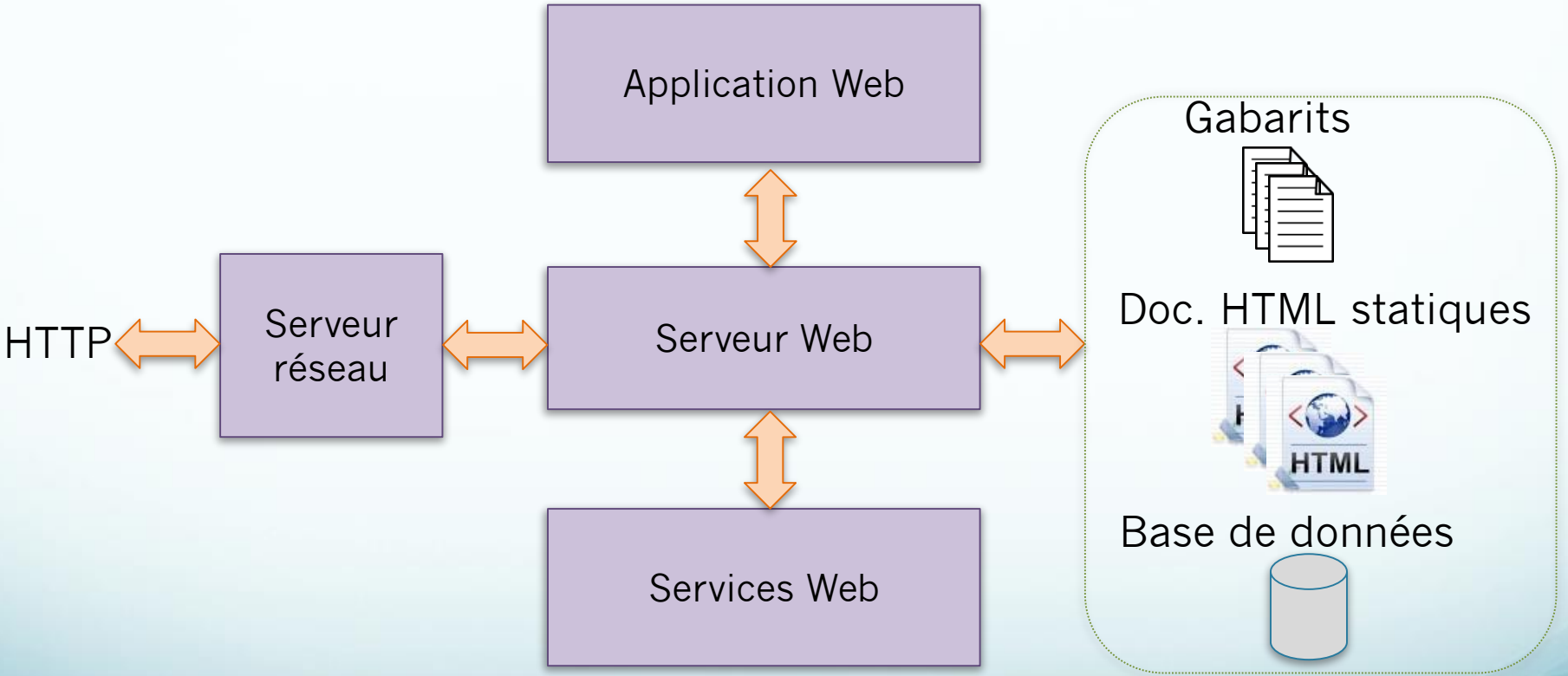
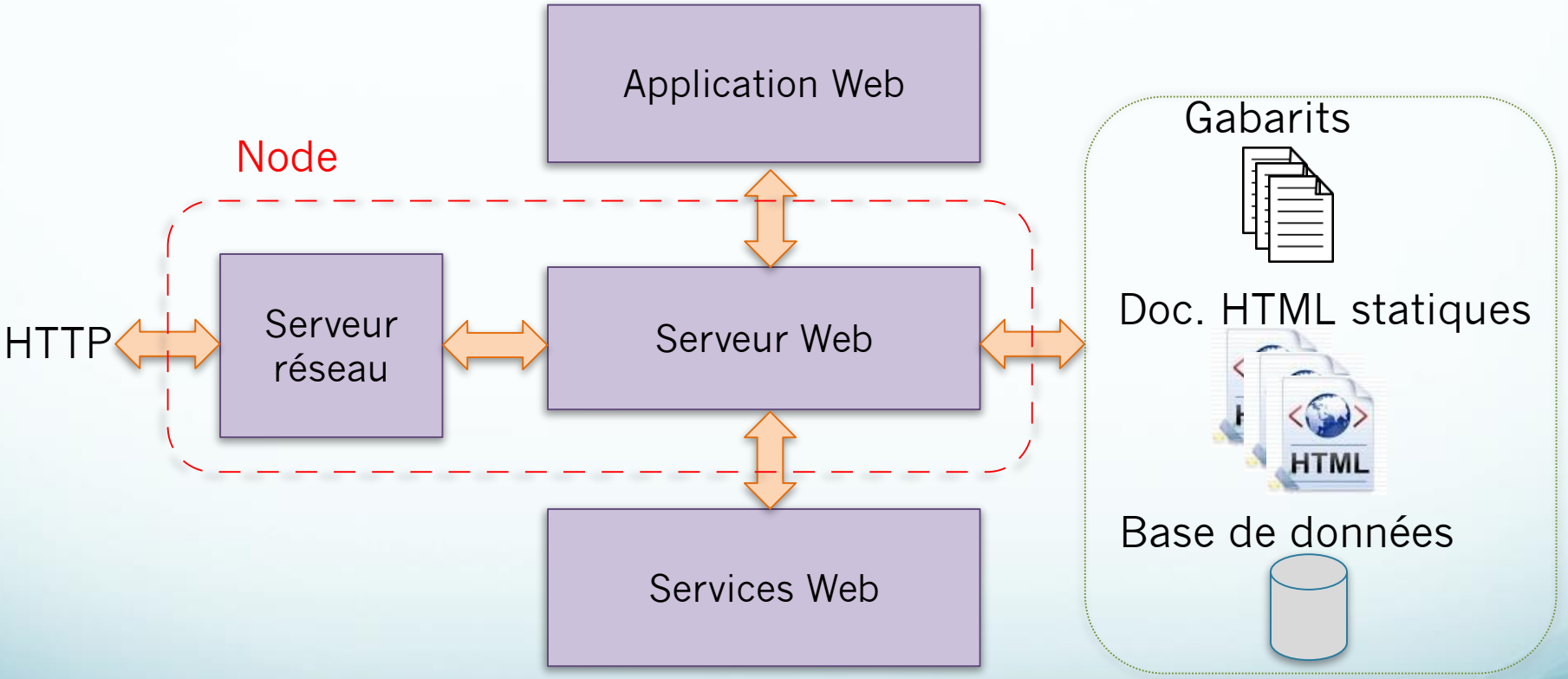
Serveur

Michel Gagnon
Konstantinos Lambrou-Latreille
Nikolay Radoev
École Polytechnique de Montréal

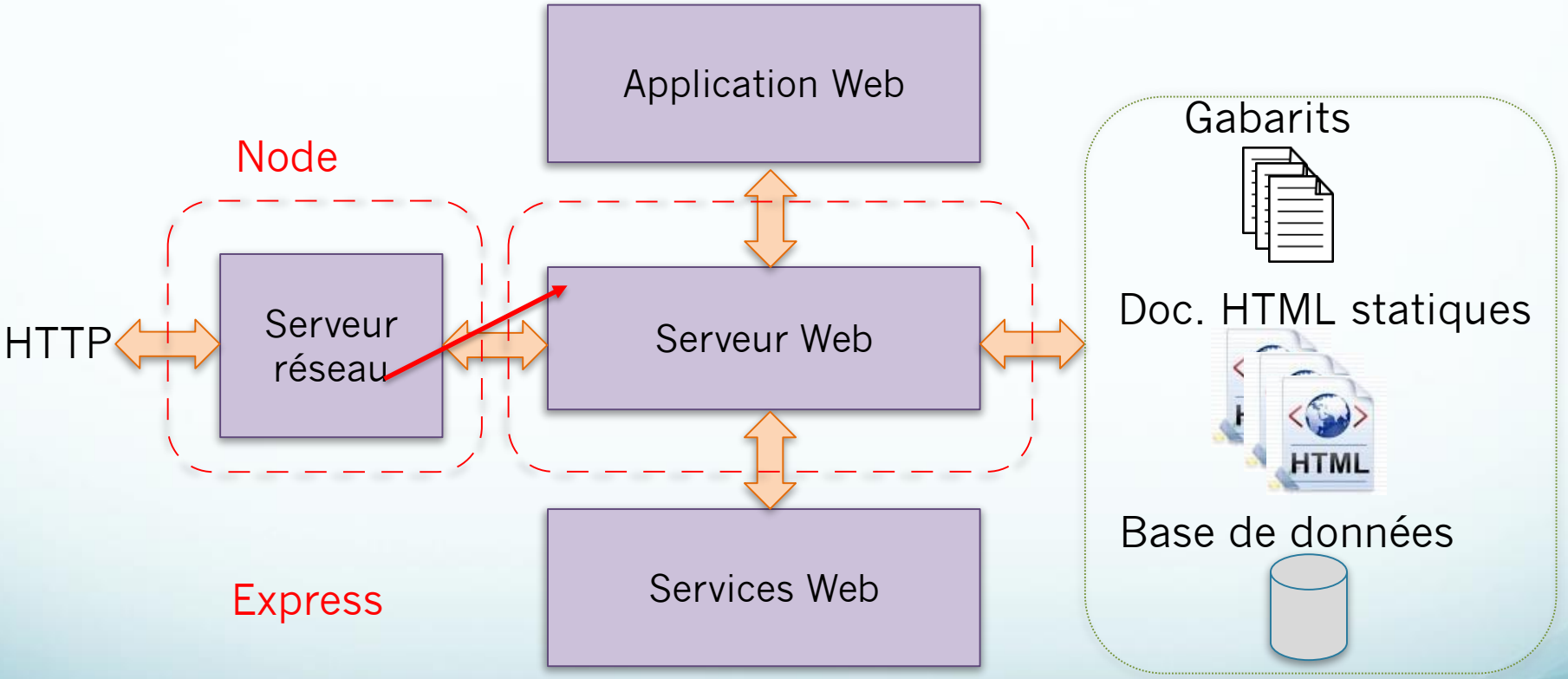
Architecture d'un serveur



Architecture d'un serveur



Architecture d'un serveur



Node

- Distinction floue entre serveur web et serveur statique
  - Serveur statique : ne fait que livrer des fichiers statiques (HTML,CSS, JS, etc)
  - Serveur web : contient de la logique de traitement (*business logic*)
- Programmation par événements
  - Similaire à la boucle d'événement d'un navigateur
  - Se base sur l'engin de JS « V8 »
- Le code JS s'exécute sur un seul fil d'exécution et tout le traitement asynchrone est fait en C++

Node – Pas une solution parfaite

- On doit toujours penser en terme de programmation asynchrone
  - Faut savoir quelle partie du traitement dépend de la fin de quelle action afin d'éviter des problèmes de synchronisation

- *Callback Hell :* code difficile à lire
  - Node se base sur les fonctions de rappel (*callback*) qui peuvent présenter une imbrication trop profonde (*callback hell*)

- JavaScript (??)
  - Node a été créé en 2009 par Ryan Dahl. La réputation de JS à l'époque a été une source de mauvaise réputation pour Node pendant longtemps. Ceci semble avoir changé depuis quelques années.

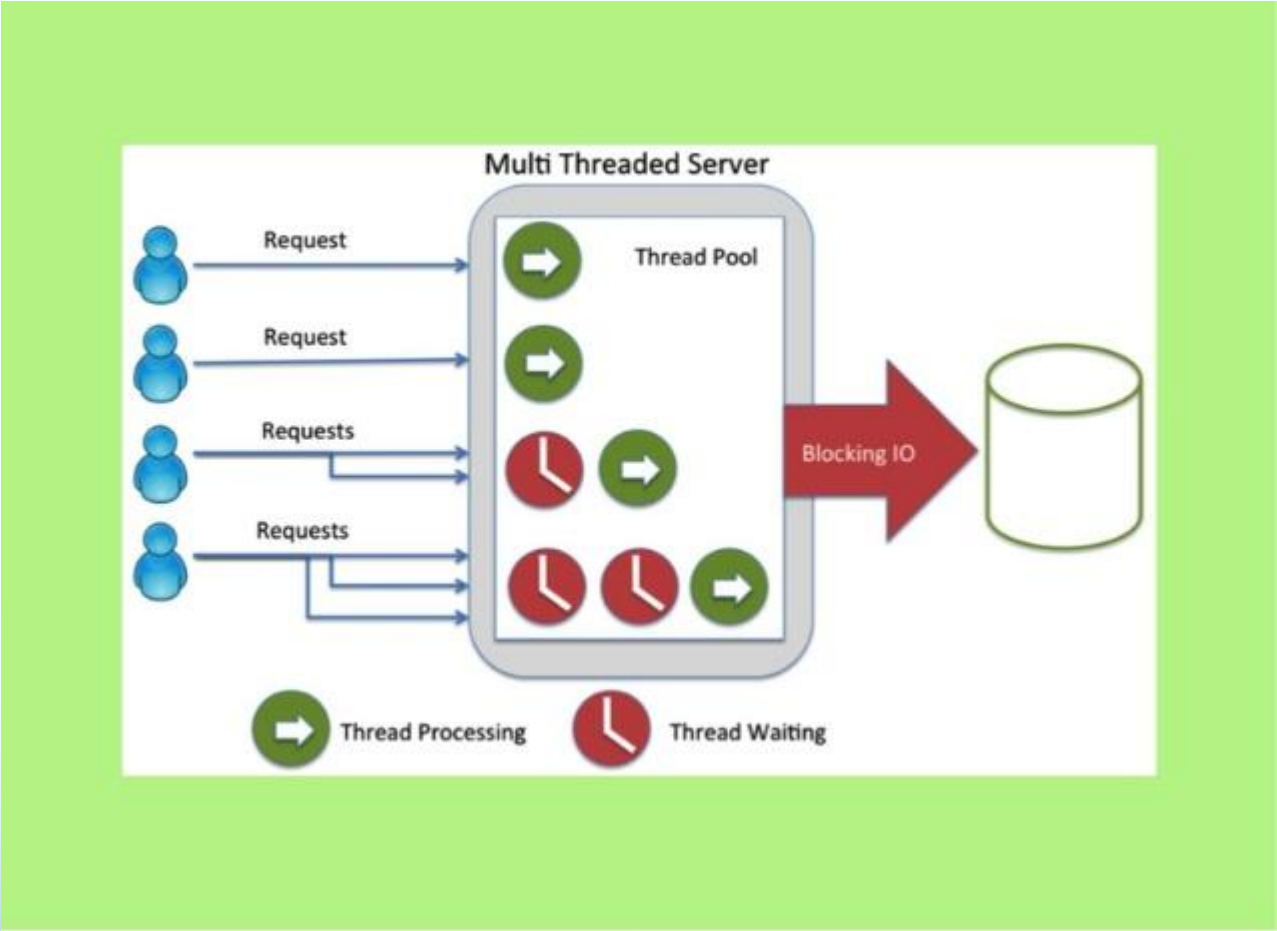
Callback Hell - Exemple

```
const makeBurger = callback => {
    getBeef(beef => {
      cookBeef(beef, cookedBeef {
        getBuns(buns => {
          putBeefBetweenBuns(buns, beef, burger => {
            callback(burger)
          })
        })
      })
    })
}
```

Node – C'est quoi?

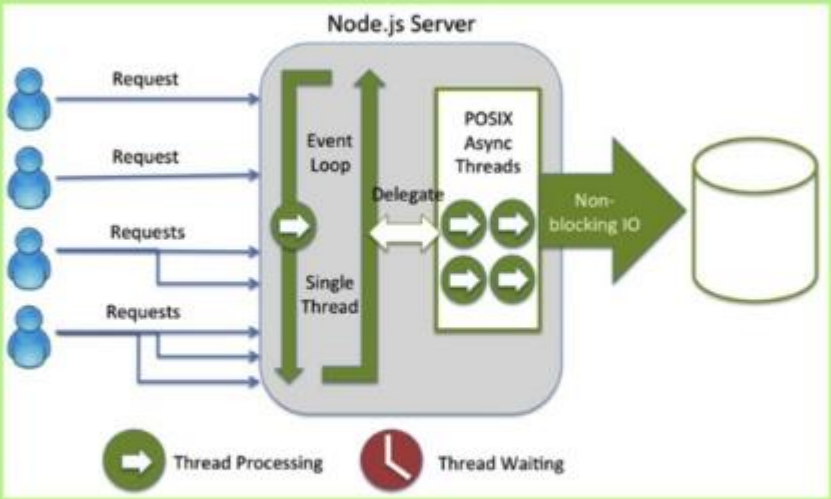
- V8 : L'engin de JavaScript (C++)
  - Le même engin que Chromium/Chrome et ses dérivés
- Une librairie standard (Javascript)
  - Un seul fil d'exécution avec une boucle d'événements
- LibEio : Async I/O (C++)
  - Gestion asynchrone des entrés/sorties (*I/O*)
- LibUv : Abstraction sur LibEio, Libev (C++)
  - Permet d'unifier les différents appels spécifiques à chaque système sous une interface commune
  - Possède un ensemble de fils d'exécution (*thread pool*) qui s'occupe à rouler les taches asynchrones

Multithreading vs Node



https://fr.slideshare.net/jguerrero99/javascript-and-nodejs-fundamentals

Multithreading vs Node



https://fr.slideshare.net/jguerrero99/javascript-and-nodejs-fundamentals

Programmation asynchrone

- La nature asynchrone et les *callback* peuvent rendre le code difficile à lire et maintenir

- Voici un exemple utilisé pour le reste des notes :
  - On (1) lit le fichier **file1.txt** qui contient le nom d'un autre fichier (**file2.txt**) dans lequel on (2) écrit la chaîne "**Hello World**" et finalement on veut (3) lire le fichier **file2.txt** et afficher son contenu dans la console.
    - Les lecture/écriture de fichiers sont asynchrones. Il faut utiliser des fonctions de rappel
    - Il faut faire les 3 étapes séquentiellement.

Programmation asynchrone

```
const fs = require('fs’);

function readMultipleFiles() {
    fs.readFile("file1.txt", (err, data) => {
        const newFileName = data.toString();
        fs.writeFile(newFileName,"Hello World",(err) => {
            fs.readFile(newFileName,(err, dataFile2) => {
                console.log(dataFile2.toString());
            });
        });
    });
}
```

Le module **fs** (file system) permet de manipuler des fichiers

Promesse (*Promise*)

- Une solution possible au code sapin est l'utilisation de promesses (*promise*) ajoutés dans ES2015
- Une promesse est un objet qui retourne (ou rejette) éventuellement le résultat d'une fonction asynchrone
- Promise prend en paramètre une fonction qui prend 2 autres fonctions (resolve,reject) comme paramètres
  - resolve et reject se comportenet comme un return : dès qu'une des 2 fonctions est rencontrée, la promesse est terminée (soit résolue ou rejetée en fonction de la fonction appelée)

Promesse - Exemple

Promesse résolue :

```
const promise =
    new Promise((resolve, reject) => {
        resolve("good");
    });

console.log(promise);  // Promise { 'good' }
```

Promesse rejetée :

```
const promise =
    new Promise((resolve, reject) => {
        reject("bad");
    });

console.log(promise); // Promise { <rejected> 'bad' }
```

Promesse (*Promise*)

- On peut enchaîner des promise avec la fonction *then*
  - Il faut faire un return pour pouvoir continuer la chaîne.
  - On peut rien retourner aussi : le prochain *then* aura une fonction sans paramètres.

- On peut traiter un *reject* avec la fonction *catch*
  - Normalement reservé pour le traitement des erreurs

- Les fonctions *then* et *reject* prennent d'autres fonctions en paramètre
  - Très souvent, on utilise des *arrow functions* comme paramètres
  - Les paramètres de ces fonctions sont la valeur de *resolve* ou *reject* d'une Promesse
  - Le retour de cette fonction est le paramètre d'entrée pour le prochain *then* ou *catch* de la chaîne

Promesse – chaîne de traitement

```
const promiseChain =
    new Promise((resolve, reject) => {
        resolve("start of chain");
    });

promiseChain
    .then(value => { console.log(value); return 1; })
    .then(value => { console.log(value); return 2; })
    .then(value => { console.log(value); return 3; })
    .then(value => { console.log(value); return 4; })
    .catch(err =>  { console.log(err); });
```

La console affichera 1,2,3 (1 chiffre par ligne)
Pourquoi pas 4 ?

Promesse dans une fonction asynchrone

Lecture d'un fichier avec Promise

```
const fs = require('fs');

function readFilePromise(name) {
    return
    new Promise((resolve, reject) => {
        fs.readFile(name,(err, data) => {
            if (err) {
                reject(err);
                return;
            }
            resolve(data.toString());
        });
    });
}
```

Écriture d'un fichier avec Promise

```
const fs = require('fs');

function writeFilePromise(fileName, content) {
    return
    new Promise((resolve, reject) => {
        fs.writeFile(
          fileName, content, (err) => {
            if (err) {
                reject(err);
                return;
            }
            resolve();
        });
    });
}
```

Promesse dans une fonction asynchrone

```
const textToWrite = 'Hello World';
const fileName = 'file1.txt';

// Lecture multiple avec des Promise
readFilePromise(fileName)
  .then(data => {
    writeFilePromise(data, textToWrite)
      .then(() => {
        readFilePromise(data)
          .then(data =>
            console.log(`Contenu de ${fileName} : ${data}`));
      });
   });
```

Le problème de code de sapin existe encore, même si le code est plus facilie à lire
Solution : ???

Promesse dans une fonction asynchrone

```
const textToWrite = 'Hello World';
const fileName = 'file1.txt';

readFilePromise(fileName)
  .then(file2Name => {
    writeFilePromise(file2Name, textToWrite);
    return file2Name; })
  .then(file2Name => { return readFilePromise(file2Name); })
  .then(data =>console.log(`Contenu de ${fileName} : ${data}`))
```

Le problème de code de sapin existe encore, même si le code est plus facilie à lire

Solution : utiliser les chaînes de promesses
Note : chaque *then* peut être écrit sur 1 seule ligne dans un éditeur de texte normal

Async/Await

- Syntaxe plus récente (depuis ES2017) qui permet de simplifier l'utilisation des promesses
  - Async/Await utilise toujours des *promise* en arrière

- Une fonction est déclarée asynchrone avec le mot clé *async* au début de la fonction
  - Cette fonction retourne alors une *promise* implicite
  - Les fonctions anonymes et fléchées peuvent aussi être *async*

- Le mot clé *await* permet d'attendre la réponse d'une fonction asynchrone et mettre le code "en pause" sur une ligne spécifique.
  - *await* est utilisable seulement dans une fonction *async*
  - Permet de remplacer le *then* des Promesses

Async/Await – Exemple simple

```
function a() {
    return Promise.resolve("a");
}
async function b() {
    return "b";
}
console.log(a()); // Promise { 'a' }
console.log(b()); // Promise { 'b' }
```

Async/Await – Lire un fichier

```
const fs = require('fs');
const util = require('util');
const file1Name = 'file1.txt';

// util.promisify transforme une fonction avec callback en
// fonction qui retourne une promise
const read = util.promisify(fs.readFile);
const run = async () => {
    const data = await read(file1Name);
    console.log(`Contenu de ${file1Name} : ${data.toString()}`);
};

run();
```

Async/Await – Séquence asynchrone

Voici l'exemple de la chaîne de promesses avec async/await

```
const fs = require('fs');
const util = require('util');
const file1Name = 'file1.txt';

const read = util.promisify(fs.readFile);
const write = util.promisify(fs.writeFile);

const runMultiple = async () => {
    const textToWrite = "Hello World";
    const data = await read(file1Name);
    await write(data.toString(), textToWrite);
    const value = await read(data.toString());
    console.log(`Contenu du 2e fichier : ${value.toString()}`);
}
runMultiple();
```

Node en tant que serveur HTTP

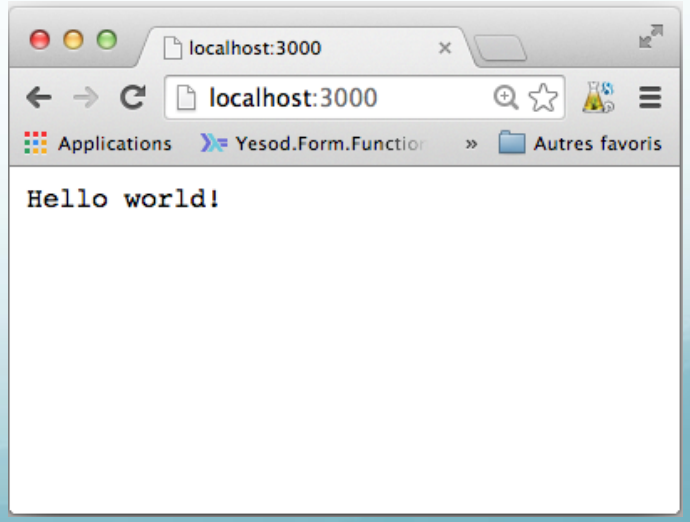
Node sans Express – exemple simple

```
let http = require('http');

let serveur =
   http.createServer((req, res)=>{
     res.writeHead(200, { 'Content-Type': 'text/plain' });
     res.end('Hello world!');
   })

serveur.listen(3000);

console.log('Serveur démarré sur localhost:3000');
```



Node sans Express – exemple simple

Les fonctionnalités de Node sont regroupées dans des modules (chaque module est un objet)

```
let http = require('http');

let serveur =
   http.createServer((req,res)=>{
     res.writeHead(200, { 'Content-Type': 'text/plain' });
     res.end('Hello world!’);
   })

serveur.listen(3000);

console.log('Serveur démarré sur localhost:3000 ’);
```

Node sans Express – exemple simple

```
let http = require('http');

let serveur =
   http.createServer((req,res)=>{
     res.writeHead(200, { 'Content-Type': 'text/plain' });
     res.end('Hello world!’);
   })

serveur.listen(3000);

console.log('Serveur démarré sur localhost:3000 ’);
```

On crée un serveur.
On lui passe une fonction qui sera exécutée pour chaque requête reçue.

Node sans Express – exemple simple

```
let http = require('http');

let serveur =
   http.createServer((req, res)=>{
     res.writeHead(200, { 'Content-Type': 'text/plain' });
     res.end('Hello world!');
   })

serveur.listen

console.log('Serveu
```

Le paramètre res est un objet de type http.ServerResponse.
Autres propriétés: res.statusCode, res.statusMessage, etc.
Autres méthodes : res.getHeader(name), res.setHeader(name, value), etc.

Node sans Express – exemple simple

```
let http = require('http');

let serveur =
   http.createServer((req,res)=>{
     res.writeHead(200, { 'Content-Type': 'text/plain' });
     res.end('Hello world!');
   })

serveur.listen(3000);

console.log('Serveur démarré sur loc
```

Cette méthode est appelée pour spécifier le code de la réponse et ses en-têtes.

Node sans Express – exemple simple

```
let http = require('http');

let serveur =
   http.createServer((req,res)=>{
     res.writeHead(200, { 'Content-Type': 'text/plain' });
     res.end('Hello world!');
   })

serveur.listen(3000);

console.log('Serveur démarré sur loc
```

Cette méthode est appelée pour envoyer la requête, tout en spécifiant le contenu qui doit y être intégré.

Node sans Express – exemple simple

```
let http = require('http');

let serveur =
   http.createServer(function(req,res){
     res.writeHead(200, { 'Content-Type': 'text/plain' });
     res.end('Hello world!’);
   })

serveur.listen(3000);

console.log('Serveur démarré sur localhost:3000 ’);
```

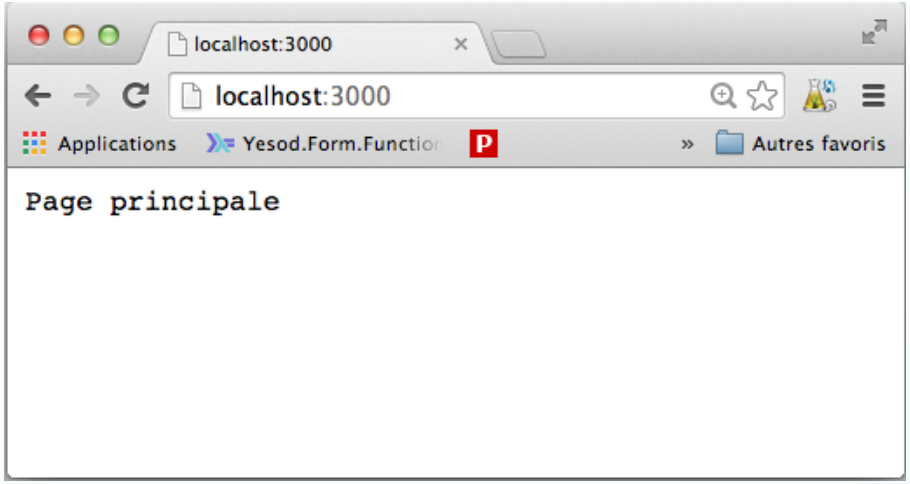
Démarrage du serveur

Node sans Express – exemple avec routage

```
let http = require('http');
let url = require('url');

http.createServer(function(req,res){
  // On décompose la requête
  let r = url.parse(req.url,true);
  let path = r.pathname;
  let query = r.query;
  switch(path) {
  case '/':
    res.writeHead(200, {'Content-Type': 'text/plain;' });
    res.end('Page principale');
    break;
  case '/message':
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Bonjour '+ query.name);
    break;
  default:
    res.writeHead(404, { 'Content-Type': 'text/plain; charset=utf-8' });
    res.end('Désolé. Cette page n\'existe pas');
    break;
  }
}).listen(3000);
```

Node sans Express – exemple avec routage



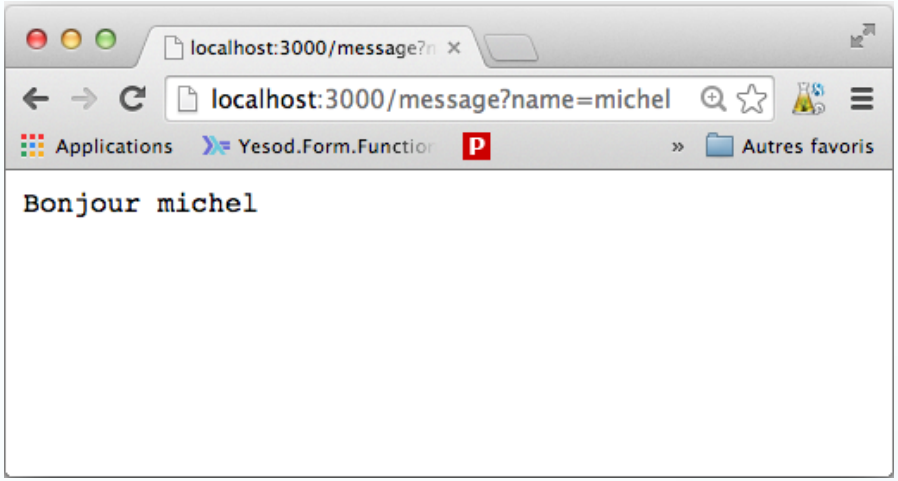

```
let http = require('http');
let url = require('url');

http.createServer(function(req,res){
  // On décompose la requête
  let r = url.parse(req.url,true);
  let path = r.pathname;
  let query = r.query;
  switch(path) {
  case '/':
    res.writeHead(200, {'Content-Type': 'text/plain;' });
    res.end('Page principale');
    break;
  case '/message':
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Bonjour '+ query.name);
    break;
  default:
    res.writeHead(404, { 'Content-Type': 'text/plain; charset=utf-8' });
    res.end('Désolé. Cette page n\'existe pas');
    break;
  }
}).listen(3000);
```

Node sans Express – exemple avec routage

```
let http = require('http');
let url = require('url');

http.createServer(function(req,res){
  // On décompose la requête
  let r = url.parse(req.url,true);
  let path = r.pathname;
  let query = r.query;
  switch(path) {
  case '/':
    res.writeHead(200, {'Content-Type': 'text/plain;' });
    res.end('Page principale');
    break;
  case '/message':
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Bonjour '+ query.name);
    break;
  default:
    res.writeHead(404, { 'Content-Type': 'text/plain; charset=utf-8' });
    res.end('Désolé. Cette page n\'existe pas');
    break;
  }
}).listen(3000);
```



Node sans Express – exemple avec routage

```
let http = require('http');
let url = require('url');

http.createServer(function(req,res){
  // On décompose la requête
  let r = url.parse(req.url,true);
  let path = r.pathname;
  let query = r.query;
  switch(path) {
  case '/':
    res.writeHead(200, {'Content-Type': 'text/plain;' });
    res.end('Page principale');
    break;
  case '/message':
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Bonjour '+ query.name);
    break;
  default:
    res.writeHead(404, { 'Content-Type': 'text/plain; charset=utf-8' });
    res.end('Désolé. Cette page n\'existe pas');
    break;
  }
}).listen(3000);
```

Node sans Express – exemple avec routage

```
let http = require('http');
let url = require('url');

http.createServer(function(req,res){
  // On décompose la requête
  let r = url.parse(req.url,true);
  let path = r.pathname;
  let query = r.query;
  switch(path) {
  case '/':
    res.writeHead(200, {'Content-Type': 'text/plain;' });
    res.end('Page principale');
    break;
  case '/message':
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Bonjour '+ query.name);
    break;
  default:
    res.writeHead(404, { 'Content-Type': 'text/plain; charset=utf-8' });
    res.end('Désolé. Cette page n\'existe pas');
    break;
  }
}).listen(3000);
```

L'URL de la requête peut être analysé, ce qui résultera en un objet.

Node sans Express – exemple avec routage

```
let http = require('http');
let url = require('url');

http.createServer(function(req,res){
  // On décompose la requête
  let r = url.parse(req.url,true);
  let path = r.pathname;
  let query = r.query;
  switch(path) {
  case '/':
    res.writeHead(200, {'Content-Type': 'tex ain;' });
    res.end('Page principale');
    break;
  case '/message':
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Bonjour '+ query.name);
    break;
  default:
    res.writeHead(404, { 'Content-Type': 'text/plain; charset=utf-8' });
    res.end('Désolé. Cette page n\'existe pas');
    break;
  }
}).listen(3000);
```

Les paramètres qui sont envoyés dans la requête sont regroupés dans l'objet query.

Contenu statique C'est quoi?

- Contenu qui ne change pas selon l'état de votre application serveur

- Par exemple ...

  - Les feuilles de styles CSS
  - Le code Javascript pour le côté client
  - Les images
  - Les polices
  - Pages HTML statique
  - Etc.

Node sans Express – Page statique

Nous définissons une fonction qui récupère le document désiré

```
function serveStaticFile(res, path, contentType, responseCode) {
    if(!responseCode) responseCode = 200

    fs.readFile(__dirname + path, function(err, data) {
       if (err) {
          res.writeHead(500, { 'Content-Type': 'text/plain' })
          res.end('500 - Erreur sur le serveur')
       } else {
          res.writeHead(responseCode, {'Content-Type': contentType})
          res.end(data)
       }
    });
}
```

Node sans Express – Page statique

Nous définissons une fonction qui récupère le document désiré

```
function serveStaticFile(res, path, contentType, responseCode) {
    if(!responseCode) responseCode = 200;

    fs.readFile(__dirname + path, function(err,data) {
       if (err) {
          res.writeHead(500, { 'Content-Type': 'text/plain' });
          res.end('500 - Erreur sur le serveur');
       } else {
          res.writeHead(responseCode,{'Content-Type': contentType});
          res.end(data);
       }
    });
}
```

Méthode asynchrone pour lire un fichier. La fonction passée en argument sera exécutée une fois que le contenu du fichier sera chargé.

Node sans Express – Page statique

Nous définissons une fonction qui récupère le document désiré

```
function serveStaticFile(res, path, contentType, responseCode) {
    if(!responseCode) responseCode = 200;

    fs.readFile(__dirname + path, function(err,data) {
       if (err) {
          res.writeHead(50    { 'Content-Type': 'text/plain' });
          res.end('500 - Err      r le serveur');
       } else {
          res.writeHead(      e': contentType});
          res.end(data);
       }
    });
}
```

Correspond au répertoire dans lequel se trouve le script actuellement exécuté

Node sans Express – Page statique

Notre code pour le serveur sera maintenant celui-ci:

```
http.createServer(function(req,res){
    let r = url.parse(req.url,true);
    let path = r.pathname;
    let query = r.query;
    switch(path) {
      case '/':
        serveStaticFile(res, '/public/home.html', 'text/html');
        break;
      case '/about':
        serveStaticFile(res, '/public/about.html', 'text/html');
        break;
      case '/img/logo.jpg':
        serveStaticFile(res, '/public/img/logo.jpeg','image/jpeg');
        break;
      default:
        serveStaticFile(res, '/public/404.html', 'text/html', 404);
        break;
    }
}).listen(3000);
```

Express

- Offre un outil d'« échafaudage » (*scafolding*)
- Ajoute une couche sur Node pour simplifier notre tâche de développeur pour des serveurs web, surtout avec HTTP

- Basé sur une pile de middlewares
  - Un middleware est une fonction prenant trois arguments: la requête (**req**), la réponse (**res**), et un objet (**next**) représentant le prochaine middleware à être exécuté dans la pile
  - On peut aussi définir un middleware qui, en plus des trois arguments cités précédemment prend comme premier argument un objet (**err**), qui sera défini si on a une situation d'erreur
  - Les routeurs sont des cas particuliers de middlewares

Express middleware Exemple

```
let express = require('express')
let http = require('http')
let logger = require('morgan');

let app = express()

// On inclut nos middlware ici
app.use(logger('dev'))
app.use(monPremierMiddleware)
// ...

function monPremierMiddleware(req, res, next) {
  // Faire ce que tu veux avec la requête et la réponse.
  // Une fois terminé, appelle next() pour passer au prochain
  // middleware
  next()
}

http.createServer(app).listen(3000)
```

Pile de middlewares

- Chaque middleware peut, à la fin de son traitement, demander qu'on passe au suivant en appelant **next()**
  - Les middlewares sont exécutés dans l'ordre de déclaration dans le code. L'ordre est donc important.
- Un appel à **res.end()**, **res.send()** ou **res.render()** renvoie la réponse au client
  - **send()** envoie directement au client le contenu qui lui est passé en paramètre
  - **render()** répond en utilisant les paramètres suivants:
    - le gabarit qui doit être utilisé
    - un objet qui pourra contenir des informations qui seront extraites par le gabarit

Pile de middlewares Exemple

```
let express = require('express')
let http = require('http')
let logger = require('morgan');

let app = express()

// On inclut nos middlware ici
app.use(logger('dev'))
app.use(monPremierMiddleware)
app.use((req, res, next) => res.envoyerAllo())

function monPremierMiddleware(req, res, next) {
  // Faire ce que tu veux avec la requête et la réponse.
  // Une fois terminé, appelle next() pour passer au prochain
  // middleware
  res.envoyerAllo = () => res.send('Allo tout le monde')
  next()
}

http.createServer(app).listen(3000)
```

Routage

```
let express = require('express')

let app = express()

app.use((req, res, next) => {
  res.header('Content-Type', 'text/plain')
})

app.get('/', (req, res, next) => {
  res.send('Bienvenue à la page
d\'accueil')
})

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

```
app.use((req, res, next) => {
  res.status(404)
     .send('Page d\'erreur 404!')
})

app.use((err, req, res, next) => {
  console.log(err.stack)
  res.status(500)
     .send('Qqc a brisée!')
})

app.listen(3000)
```

Routage

- Est indiquée par un chemin. Ex: **/home/users**
- On peut utiliser les symboles spéciaux ?, + et *:
  - **/home/*** (n'importe quoi peut suivre **/home/**)
  - **/home/*/users** (n'importe quoi entre les deux)
  - **/home/b?elle** (indique que le **b** est facultatif)
  - **/home/(non)?** (**non** est facultatif)
  - **/users/noo+n** (le 2e **o** peut apparaître 1 ou plusieurs fois)
- On peut utiliser une forme **:id** pourra être reprise dans le code (ce sera un attribut de **req.params**)
  - **/home/user/:id**
  - **Attention : le « : » identifie le paramètre, mais ne doit pas faire partie de la route en tant que tel**
- On peut utiliser une expression régulière:
  - **/(\/data)|(\/users)\/about/**
- On peut utiliser un tableau:
  - **['/info', '/about*', /\/hip|\/hop/]**

Routage

- Il est parfois plus intéressant de regrouper un ensemble de routes qui traitent du même domaine
- Séparer les différentes routes dans des fichiers différents allège le code et permet de mieux diviser la logique interne du serveur
- Express offre une classe Router qui permet de faire ça

Routage

- Il faut instancier le middleware *Router* et l'ajouter à l'application Express avec la méthode générale *use.*
- On peut définir un préfixe pour toutes les requêtes traités par un *Router* spécifique

```
// Fichier app.js
let express = require('express')
let routes = require('./routes')

let app = express()

app.use('/monRouteur', routes)

app.listen(3000)
```

```
// Fichier routes.js
let express = require('express');
let router = express.Router();

/* GET home page. */
router.get('/', (req, res, next) => {
  res.send('Page accueil')
});

router.get('/about', (req, res, next) => {
  res.send('Page à propos')
});

module.exports = router;
```

Corps d'une méthode HTTP

- Par défaut, Express ne sait pas quoi faire avec le corps d'une requête HTTP.
- Il faut utiliser les middlewares **json** et **urlencoded**
  - Beaucoup d'exemples utilisent **bodyParser** qui est maintenant *deprecated* en faveur de la version native dans Express
- En fait, il s'agit de plusieurs middlewares, parmi lesquels ont doit choisir le parseur qui nous intéresse:
  - **app.use(express.urlencoded({extended:true}))**
  - **app.use(express.json())**
- Ce middleware ajoute un attribut **body** à la requête **req**, qui contiendra toutes les paires attribut/valeur envoyées par la soumission de la requête (que ce soit par le protocole standard HTML ou AJAX)

Formulaires

```
// app.js
const express = require('express')

const app = express()

app.use(express.urlencoded({extended: true}))
app.use(express.json())

const commentaires = []

app.post('/commentaires', (req, res) => {
  const auteur = req.body.auteur
  const commentaire = req.body.commentaire
  commentaires.push({auteur, commentaire})
  res.redirect('/commentaires')
})

app.get('/commentaires', (req, res) => {
  res.json(commentaires)
})

app.listen(3000)
```

```
// index.html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Formulaire</title>
  </head>
  <body>
    <form
action="http://localhost:3000/commentaire
s" method="POST">
      <label for="auteur">
        Auteur:
      </label>
      <input type="text" name="auteur">
      <label for="commentaire">
        Commentaire:
      </label>
      <input type="text"
             name="commentaire">
      <input type="submit"
             value="Envoyer">
    </form>
  </body>
</html>
```

Contenu statique

- Créez un dossier qui va contenir le contenu statique
  - *public/* ou *static/* (nomenclature commune)
- Utiliser le middleware *static* de Express

```
// Fichier app.js
let express = require('express')

let app = express()

...

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

...

app.listen(3000)
```

Rendu côté serveur

- Il est possible de générer des pages HTML dynamiques du côté serveur (*server side rendering)*
  - Pratique si la machine du client est moins puissante que le serveur. Cout supplémentaire pour le serveur cependant.
  - Améliorer le SEO (*Search Engine Optimization)* puisqu'on reçoit les pages HTML complètes lors d'une requête

- Approche habituelle: basée sur un squelette HTML
  - L'information dynamique est insérée dans un gabarit minimaliste de HTML et une page complète est produite et renvoyé au client

- Ce que permet normalement un modèle de gabarit:
  - Interpolation
  - Énoncés conditionnels
  - Itérations
  - Réutilisation de bouts de HTML communs (ex : nav, footer, etc.)

Rendu côté serveur

- Express supporte plusieurs modèle de gabarit
  - Handlebars
  - ReactJS (plus commun du côté client)
  - EJS (Embedded JavaScript)
  - Pug (assez populaire, mais avec une syntaxe très différente du HTML)

- Nous allons montrer EJS

```
let express = require('express')
let app = express()

// On indique où vont se retrouver les fichiers EJS
app.set('views', path.join(__dirname, 'views'))

// On spécifie le modèle de gabarit utilisé
app.set('view engine', ejs')
```

Rendu côté serveur - EJS

- EJS (Embedded JavaScript) permet d'inclure du JS directement dans un gabarit HTML.
  - Le ficher ejs est transpilé en HTML avec le code JS exécuté pendant la transpilation

- EJS a une syntaxe très basique, utilisant des balises comme en HTML
  - <% indique qu'on insère du code JS
  - <%= interpolation de la valeur dans la balise directement dans l'HTML (avec encodage)
  - <%- interpolation de la valeur dans la balise directement dans l'HTML (sans encodage)

Rendu côté serveur

- À partir d'une route, on utilise res.render(...) pour convertir un fichier EJS en HTML et l'envoyer au client.

```
let express = require('express')
let app = express()

// Détails sur le modèle de gabarit
app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')

app.get("/list", function (req, res) {
  const items = ["node.js", "ejs", "expressjs", "reactjs", "nextjs"];
  res.render("pages/list", {
    list: items,
  });
});
```

Modèle de données à envoyer dans la vue

Nom du fichier qui se trouve dans le dossier *views*

EJS – Exemple simple

```
<%- include('../template/head')-%>

<body>
    <div>
        <%- include('../template/nav')-%>
        <main>
            <h1>Exemple avec liste</h1>
            <ul>
                <%list.forEach((entry)=> {%>
                <li><%=entry%></li>
                <%});%>
            </ul>
        </main>
        <%- include('../template/footer')-%>
    </div>
</body>
```

EJS – Exemple simple

```
<%- include('../template/head')-%>

<body>
    <div>
        <%- include('../template/nav')-%>
        <main>
            <h1>Exemple avec liste</h1>
            <ul>
                <%list.forEach((entry)=> {%>
                <li><%=entry%></li>
                <%});%>
            </ul>
        </main>
        <%- include('../template/footer')-%>
    </div>
</body>
```

Le modèle passé en paramètre dans **res.render(...)** est utilisable comme un tableau normal

<%=entry%> permet de mettre la valeur de chaque élément du tableau comme la valeur de la balise

EJS
Combinaison de vues

- On utilise **include** pour réutiliser un autre fichier .ejs
  - Ex : **<%- include('../template/nav')-%>**

- La réutilisation est basée sur le concept d'interpolation:
  - Le contenu fichier pointé par le 1er paramètre d'*include()* est inséré à l'endroit de la balise d'EJS
  - On peut passer des paramètres au fichier avec un objet comme 2e paramètre de la fonction *include()*

- Si la vue incluse est quelque chose de très réutilisable (ex : nav, footer, head, etc), on l'appelle parfois gabarit (*template*)

Express - cookies

- Avec Express, on peut créer des cookies normaux et des cookies signés
- Cookie signé:
  - On lui ajoute une signature, qui est un encodage de son contenu utilisant une clé secrète
  - Lorsque le cookie est envoyé, on décode cette signature et on vérifie si le résultat obtenu correspond au contenu envoyé
  - En gros, on s'assure que le client n'a pas modifier le cookie
- Pour créer des cookies, il faut utiliser le middleware **cookie-parser**
- On crée un cookie en appelant la méthode **cookie()** de l'objet **res** (on met {signed:true} comme 2e argument si on veut qu'il soit signé)
- On extrait les cookies par le biais de l'attribut **res.cookies** ou **res.signedCookies**

Cookies - Exemple

```
app.use(require('cookie-parser')('mon secret'));

app.get('/', function(req, res) {
    res.cookie('contenuNonSigne', 'John Lewis');
    res.cookie('contenuSigne', 'John Coltrane', { signed: true });
    res.render('main', {'body’: Blue Train'});
});

app.get('/about', function(req, res) {
    res.render('about',
               {'contenuCookieNonSigne':req.cookies.contenuNonSigne,
                'contenuCookieSigne':req.signedCookies.contenuSigne});
});
```

Express - Échafaudage

- Possibilité de générer un squelette d'un projet Express
  - On retrouve toutes les fonctionnalités introduites dans ce cours telles que les routes, les middlewares les plus communs, le contenu statique, le déploiement, etc.
- Pour le générer, installer la librairie **express-generator**

  $ *npm install -g express-generator*
- Lancez les commandes suivantes (sous Linux):

  $ express nomDeMonAppExpress
  $ cd nomDeMonAppExpress
  $ npm install
  $ npm

Express - Échafaudage

- Structure de projet

  - *public/* - Contient le contenu statique
  - *routes/* - Toutes les définitions de vos routes
  - *views/* - Les vues (par défaut utilise Pug, mais peut être configuré pour EJS ou autre)
  - *app.js* – Initialisation de Express et des middlewares
  - *bin/www* - Configuration du déploiement
  - *package.json* – Dépendance du projet + scripts
  - *node_modules* - Dossier qui contient les modules de votre projet générés avec *npm install*.
    Ne pas mettre sous gestionnaire de versions : peut être potentiellement très gros