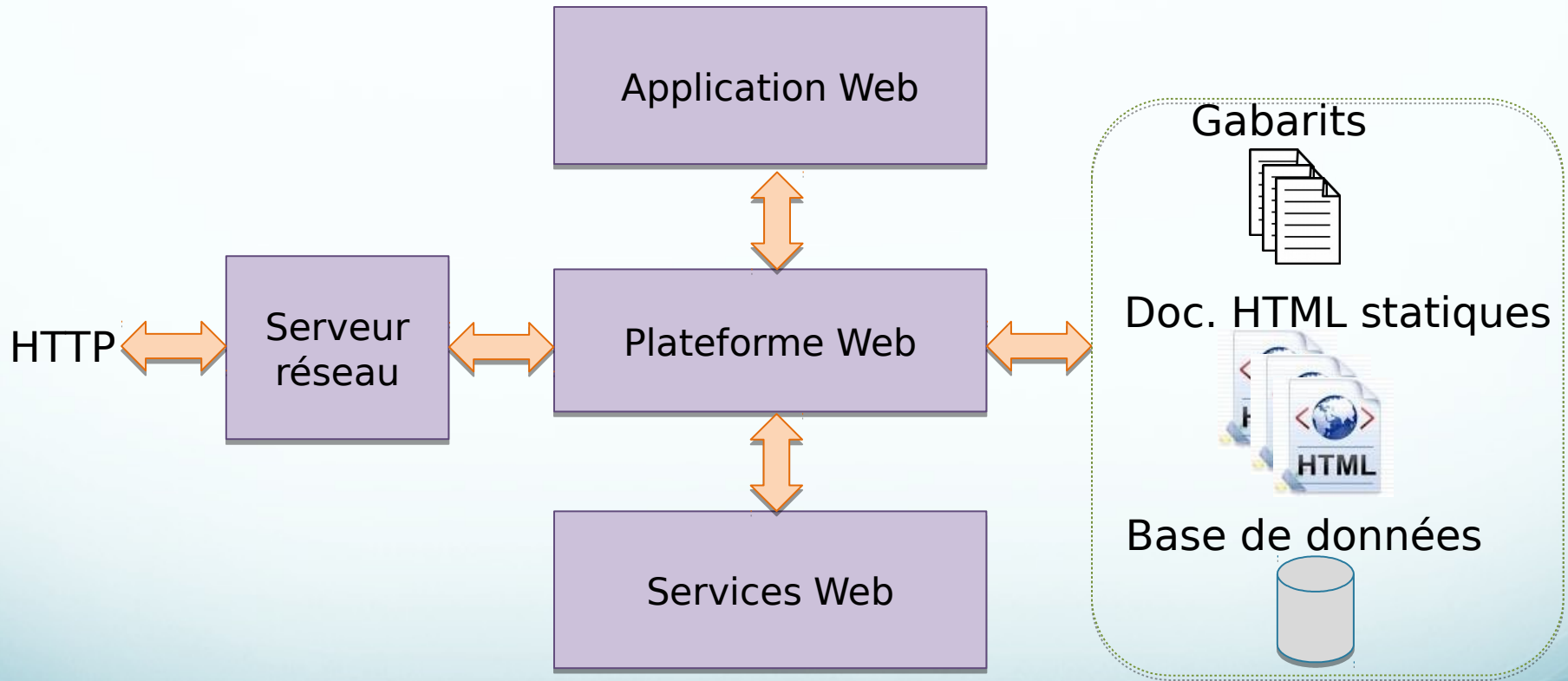


# Serveur

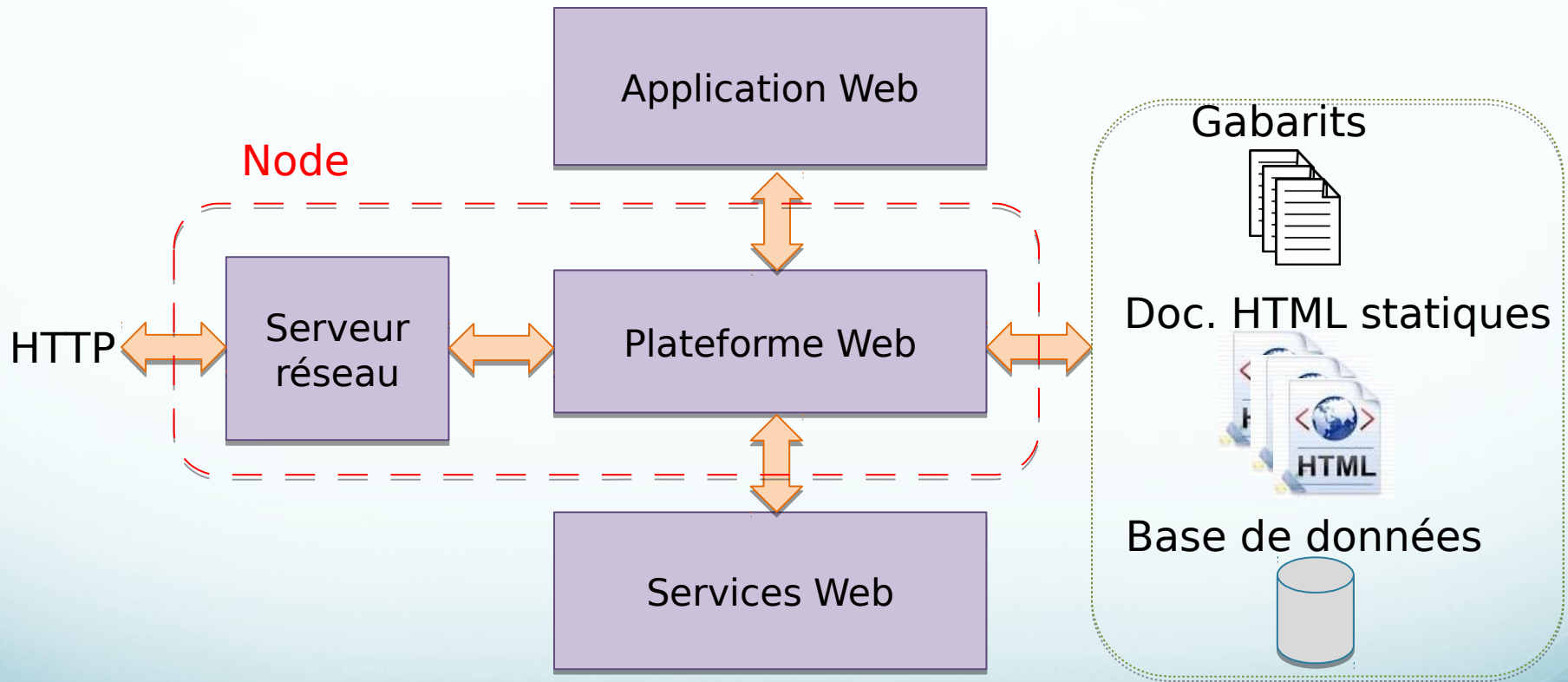
Michel Gagnon  
Konstantinos Lambrou-Latreille  
École polytechnique de Montréal



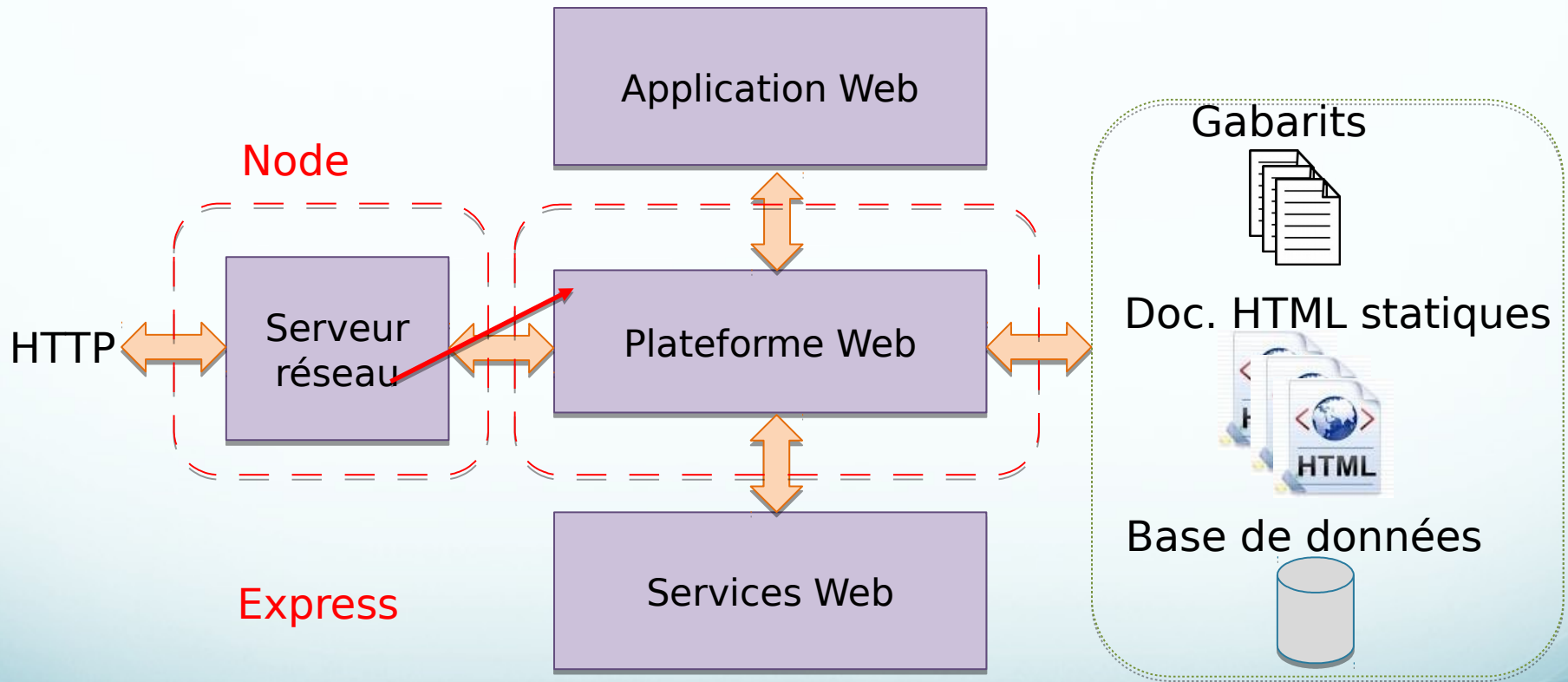
# Architecture d'un serveur



# Architecture d'un serveur



# Architecture d'un serveur



# Node

- Pas vraiment de distinction entre serveur web et application web
- Programmation par événements
- S'exécute sur un seul thread
- On programme en Javascript, compilé à la volée (just-in-time)
- Indépendant du système d'exploitation utilisé
- Routage pour faire le lien entre une requête et le module qui produit le contenu désiré
- Gros écosystème (npm)

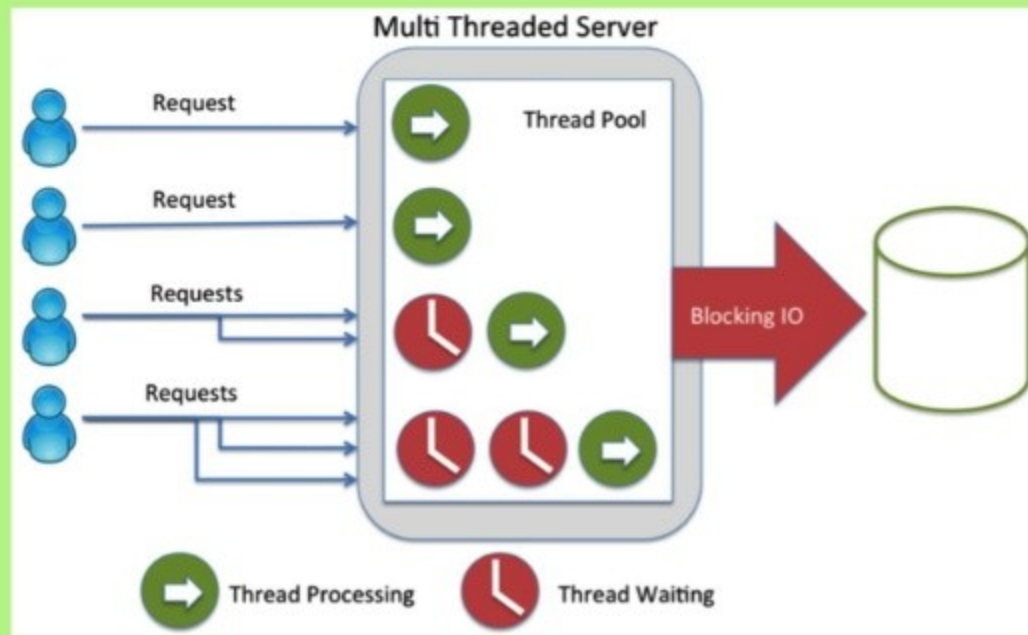
# Node – Problèmes rencontrés

- On doit toujours penser en terme de programmation asynchrone
- Cadriciels et libraries pas toutes matures
- Callback hell
- Javascript

# Node – C'est quoi?

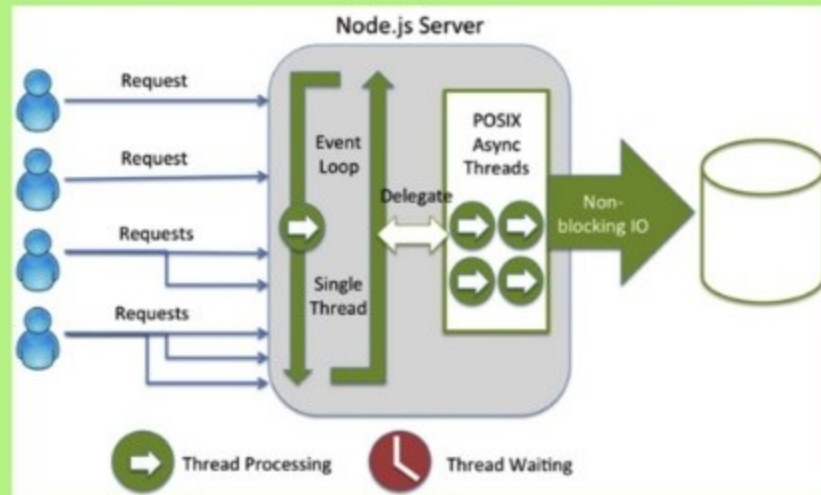
- V8 : L'engin (C++)
- Libev : Boucle des événements (C++)
- LibEio : Async I/O (C++)
- LibUv : Abstraction sur LibEio, Libev (C++)
- Une librairie standard (Javascript)

# Multithreading vs Node





# Multithreading vs Node



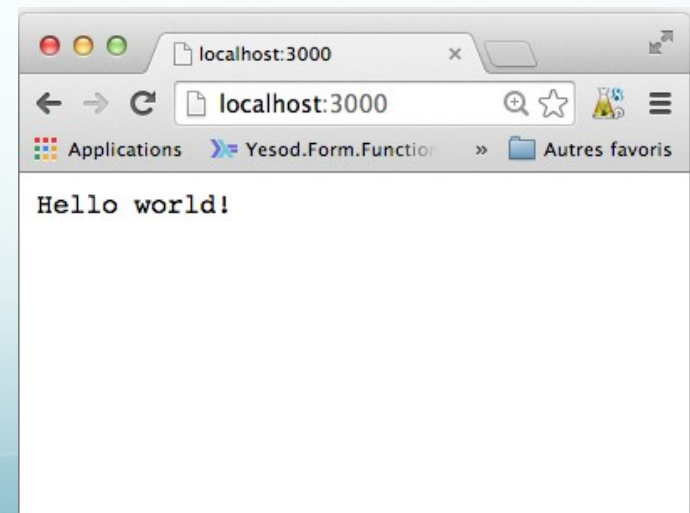
# Node sans Express – exemple simple

```
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');
```



# 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 ');
```

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

# 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 ');
```

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

serveur.listen(3000);
console.log('Serveur démarré');
```

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

serveur.listen(3000);

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

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

serveur.listen(3000);

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

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

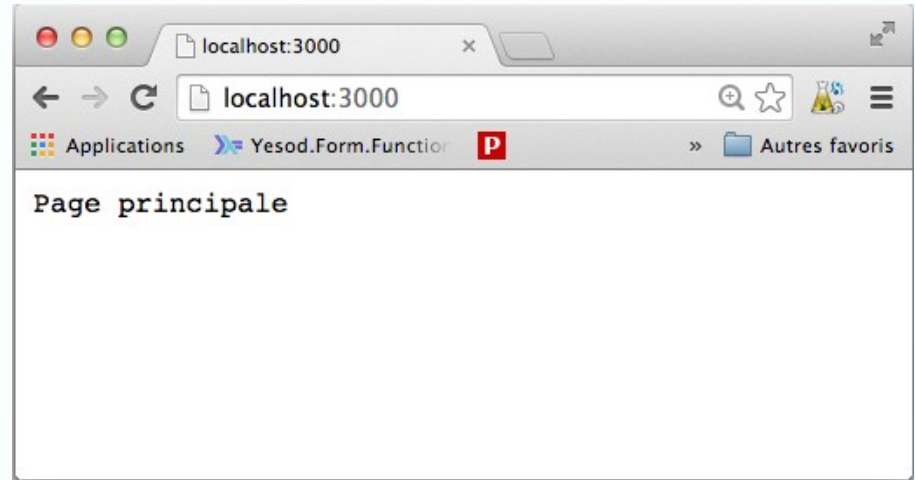
```
let http = require('http');
let url = require('url');

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');

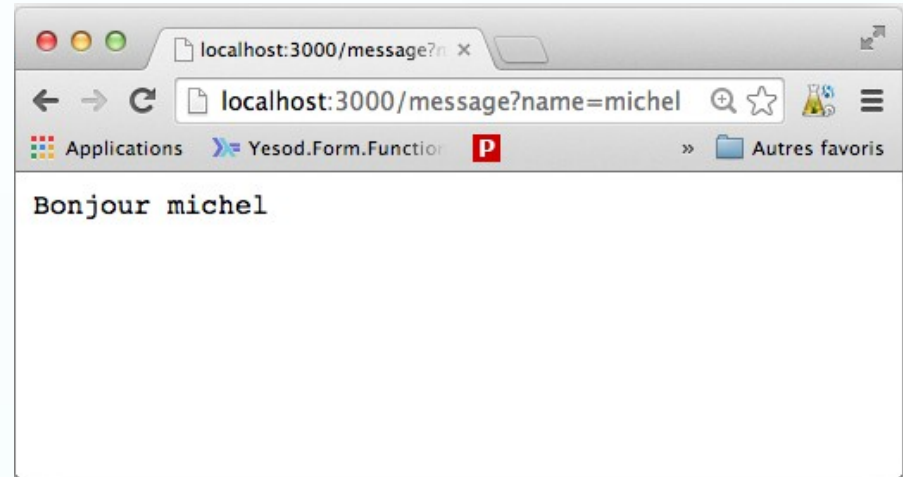
http.createServer(function(req, res){
  // On décompose la requête
  let r = url.parse(req.url, true);
  let path = r.pathname;
  let query = r.query;
  switch(path) {
    case '/':
      res.writeHead(200, { 'Content-Type': 'text/plain;' });
      res.end('Page principale');
      break;
    case '/message':
      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');

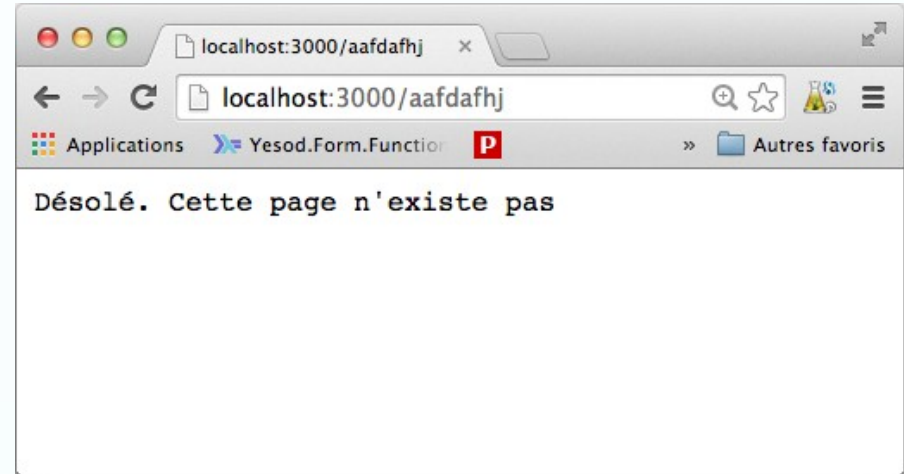
http.createServer(function(req, res){
  // On décompose la requête
  let r = url.parse(req.url, true);
  let path = r.pathname;
  let query = r.query;
  switch(path) {
    case '/':
      res.writeHead(200, { 'Content-Type': 'text/plain;' });
      res.end('Page principale');
      break;
    case '/message':
      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 parsé, 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': '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);
```

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 ...

# 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

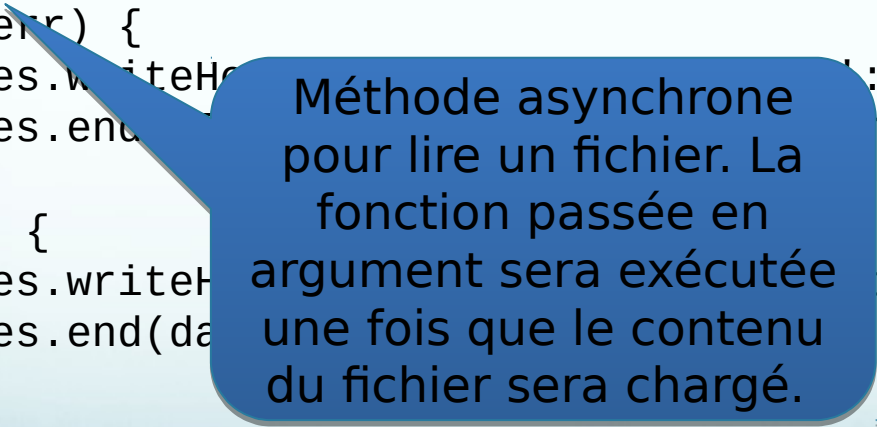
Nous définissons une fonction qui récupère le document désiré

```
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('Erreur 500');  
    }  
    else {  
      res.writeHead(responseCode, {'Content-Type': contentType});  
      res.end(data);  
    }  
  });  
}
```

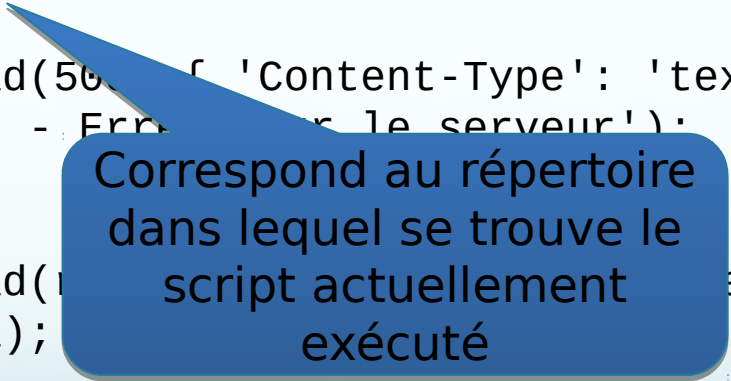


Méthode asynchrone pour lire un fichier. La fonction passée en argument sera exécutée une fois que le contenu du fichier sera chargé.

# 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(500, { 'Content-Type': 'text/plain' });  
      res.end('500 - Erreur sur le serveur');  
    }  
    else {  
      res.writeHead(responseCode, { 'Content-Type': 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 »
- Ajoute une couche sur Node pour simplifier notre tâche de développeur
- 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 middleware 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()**
- Un appel à **res.end()**, **res.send()** ou **res.render()** termine la cascade d'exécution de middlewares
  - **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
    - Nous le verrons plus tard avec Pug

# Pile de middlewares

## Exemple

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

let app = express()

// On inclut nos middleware 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: **/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**
- On peut utiliser une expression régulière:
  - **/(\data)|(\users)\about/**
- On peut utiliser un tableau:
  - **['/info', '/about\*', '/hip|/hop/']**

# Routage

- Comment définir les routes dans des fichiers différents?
- Il faut instancier le middleware *Router*

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

let app = express()

app.use('/', routes)

app.listen(3000)
```

```
// Fichier routes.js
var express = require('express');
var 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;
```

# Formulaires

- Par défaut, Express ne sait quoi faire avec le corps d'une requête HTTP.
- Il faut utiliser le middleware **body-parser**
- En fait, il s'agit de plusieurs middlewares, parmi lesquels on doit choisir le parseur qui nous intéresse:
  - `app.use(bodyParser.urlencoded({extended:true}))`
  - `app.use(bodyParser.json())`
- Ce middleware ajoute un attribut **body** à la requête **req**, qui contiendra toutes les paires attribut/valeur envoyées par la soumission du formulaire (que ce soit par le protocole standard HTML ou Ajax)

# Formulaires

```
// app.js
const express = require('express')
const bodyParser = require('body-parser')

const app = express()

app.use(bodyParser.urlencoded(
  { extended: true }))
app.use(bodyParser.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/
commentaires" 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/*
- 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)
```

# La Vue dans MVC

- La vue est un gabarit HTML:
  - Approche habituelle: basée sur un squelette HTML
  - Pug utilise une autre approche plus compacte et moins verbeuse
- Ce que permet normalement un modèle de gabarit:
  - interpolation
  - énoncés conditionnels
  - Itérations
  - combinaisons de plusieurs sources

# La Vue dans MVC

- Express supporte plusieurs modèle de gabarit
  - Haml
  - CoffeeKup
  - EJS
  - Pug (nouvelle version de Jade)
- Nous allons montrer Pug

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

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

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



# La Vue dans MVC

- À partir d'une route, on utilise `res.render(...)` pour convertir Pug 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', 'pug')
```

```
app.get('/', (req, res, next) => {  
  res.render('index', { title: 'Les amateurs de jazz' })  
})
```

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

Modèle de données  
à envoyer dans la  
vue

# Pug – Exemple simple

```
// layout.pug
doctype html
html
  head
    block hd
    title Les amateurs de jazz
  body
    block content
```

```
// index.pug
extends layout

block content
  h1= title
  p Ce site vous fera voir...
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Les amateurs de jazz</title>
  </head>
  <body>
    <h1>Les amateurs de jazz</h1>
    <p>
      Ce site vous fera voir le jazz comme
      vous ne l'avez jamais vu.
    </p>
  </body>
</html>
```

# Pug

## Combinaison de vues

- On utilise **extends** pour réutiliser une autre vue
- La réutilisation est basée sur le concept de bloc:
  - La vue qu'on étend contient des blocs à des endroits définis
  - On a alors quatre possibilités:
    - On redéfinit un bloc, qui sera alors remplacé par le nouveau code Pug qu'on spécifie
    - On ajoute du code Pug au début du bloc (**prepend**)
    - On ajoute du code Pug à la fin du bloc (**append**)
    - On laisse le bloc tel quel

# Pug

- Pour connaître toutes les fonctionnalités, allez voir le lien suivant :

<https://pugjs.org>

# 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 2<sup>e</sup> argument si on veut qu'il soit signé)
- On extrait les cookies par le biais de l'attribut **res.cookies** ou **res.signedCookies**

# Cookies - Example

```
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': 'Do bi dou bi dou wap!'});
});

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

# Express - Sessions

- Implémenté avec les cookies
  - L'id de la session est sauvegardé dans un cookie
  - Les données sont sauvegardées dans le serveur
- Il faut charger le middleware **express-session**
- Un attribut **session** est ajouté à l'objet **req**
- Initialement, cet attribut est associé à un objet vide
- Il suffit alors d'ajouter des infos à cet objet
- On peut accéder à cet objet dans n'importe quel middleware
- Par défaut, la session est gardée en mémoire vive
  - Autre possibilités: fichier, cache, BD, etc.

# Session - Exemple

```
app.get('/', function(req, res) {  
  req.session.message = 'Bonjour, comment allez-vous';  
  res.render('main', {'body': 'Do bi dou bi dou wap!'});  
});
```

```
app.get('/about', function(req, res) {  
  res.render('about', {'message': req.session.message});  
});
```



# Express - Échafaudage

- Possibilité de générer un squelette de votre projet Express
- On retrouve toutes les fonctionnalités introduites dans ce cours telles que les routes, les middlewares les plus communs, les vues Pug, le contenu statique, le déploiement, etc.
- Pour le générer, installer la librairie **express-generator**  
`$ npm install -g express-generator`
- Lancez les commandes suivantes (sous GNU/Linux):  
`$ express nomDeMonAppExpress`  
`$ cd nomDeMonAppExpress`  
`$ npm install`  
`$ npm start` ou `nodemon` (à installer globalement)

# Express - Échafaudage

- Structure de projet
    - *public/* - Contient le contenu statique
    - *routes/* - Toutes les définitions de vos routes
    - *views/* - Les vues Pug
    - *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**