eXpress JS

Un micro-framework pour NodeJS

Table des matières

- Introduction
- Routing
- Middleware
- Templating
- Les fichiers static
- Les formulaires
- Base de données
- Sessions et Cookies
- Rest Api
- Jwt Authentication

Introduction

eXpress JS

Introduction

ExpressJS est un framework Web qui vous fournit une API simple pour créer des sites Web, des applications Web...

Il vient se greffer sur NodeJS pour nous aider à

- Créer des applications webs SPA, multipage, mobile first,…
- Configurer des « middleware » servant à gérer les requêtes http.
- Configurer les tables de routage pour effectuer les différentes actions demandées via http
- Gérer les cookies
- Manipuler/gérer des templates dynamiques
- …

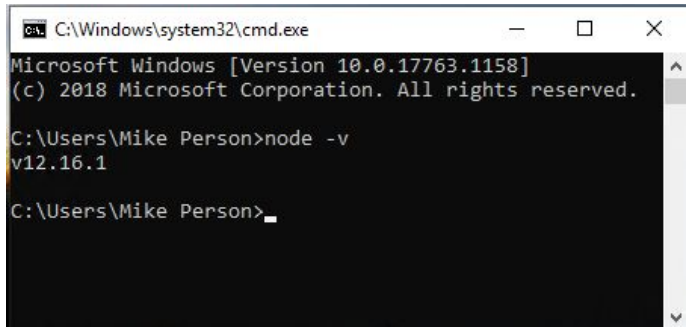
Installation

eXpress JS

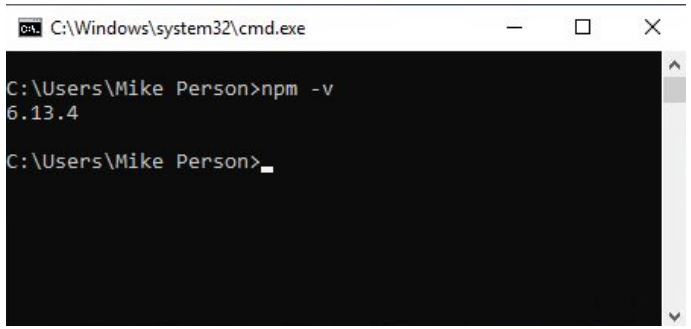
Installation

La première chose à vérifier dans notre environnement c'est la présence de Node.

(la version peu différer)

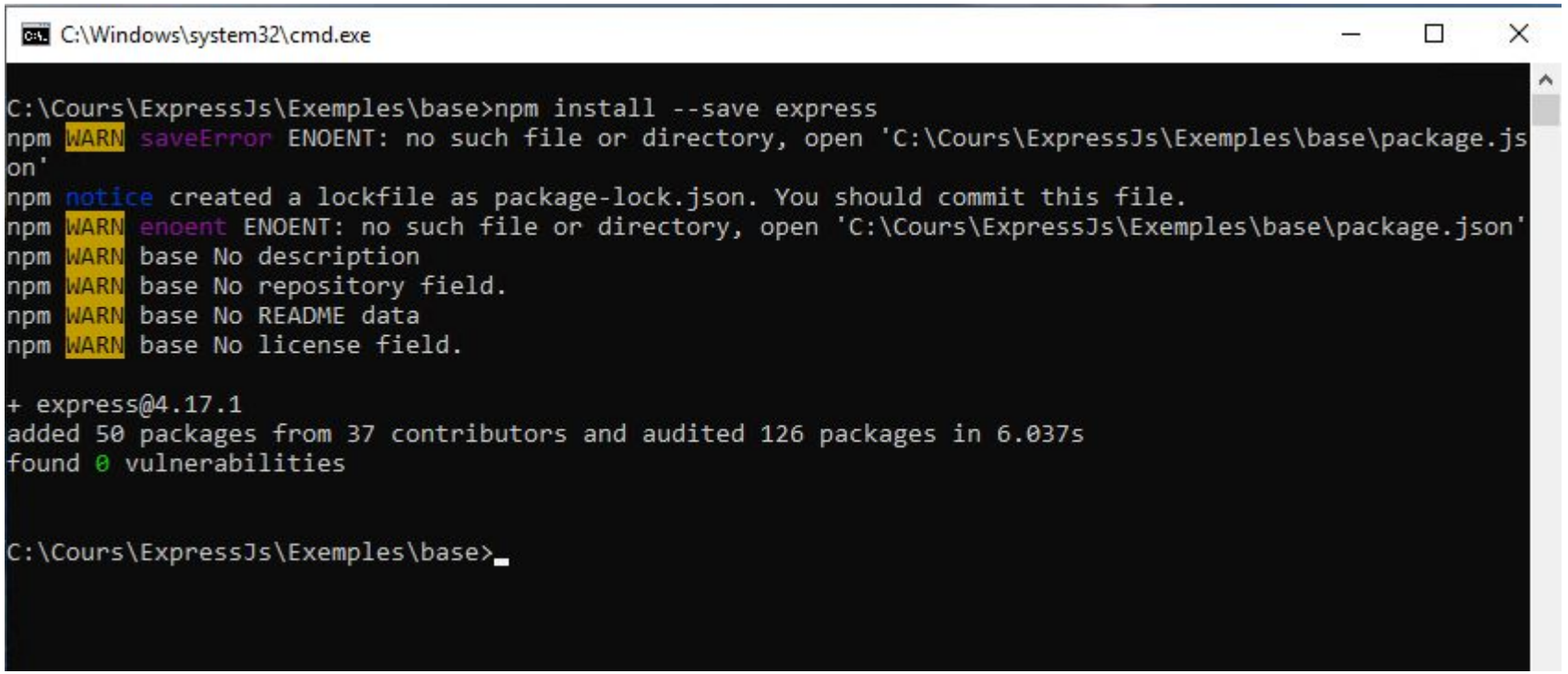


Et celle de npm



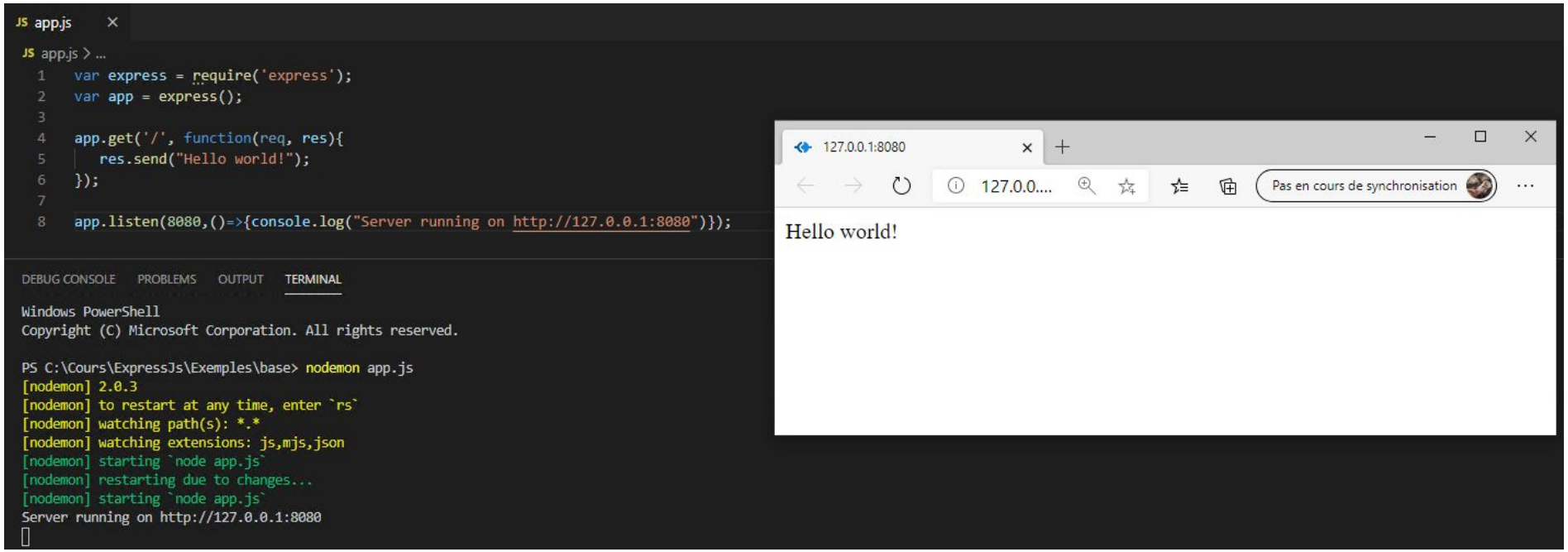
Installation

Il ne nous reste plus qu'à installer eXpress JS via npm



Installation

Pour utiliser le Framework eXpressJS, nous devons « simplement » l'inclure dans nos scripts Node comme un module classique.



Routing

eXpress JS

Routing

Les frameworks Web fournissent des ressources telles que des pages HTML, des scripts, des images, etc. sur différentes routes.

ExpressJS nous facilite la gestion de ces routes via la fonction *app.method(path,handler)*

- Method : l'un des verbes http : GET,POST,PATCH,PUT,DELETE,…
- Path : Chemin de l'url qui doit être capturée
- handler : adresse de la fonction qui doit être exécutée

Remarque :

Expresse propose également la méthode *all* qui permet de réagir à une requête url de la même façon indépendamment du verb utilisé

Routing

NodeJS

```
var http = require('http');
var routes = {

    '/': function index(request, response)
        {
            response.writeHead(200);
            response.end("Hello from test");
        },
    '/test': function test(request,response)
        {
            if (request.method !== 'POST')
            {
                //...Traitement du formulaire
                response.writeHead(200,{
                    'Content-Type': 'text/html'
                });
                response.end("Formulaire ok");
            }
            else
            {
                response.writeHead(200,{
                    'Content-Type': 'text/html'
                });
                response.end("<form>...</form>");
            }
        }
}

var server = http.createServer(function(req,res){

        if(req.url in routes)
        {
            return routes[req.url](req,res);
        }

        res.writeHead(404);
        res.end(http.STATUS_CODES[404]);
});

server.listen(8081,()=>{console.log("Server running on http://127.0.0.1:8081")});
```

NodeJS + Express

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

app.get('/',function(req,res)
{
    res.writeHead(200);
    res.end("Hello from test");
});


app.post('/test', function(req, res){
    //Traitement du formulaire
    res.writeHead(200,{
        'Content-Type': 'text/html'
    });
    res.end("<form>...</form>");
 });
 app.get('/test', function(req, res)
{
    //...Traitement du formulaire
    res.writeHead(200,{
        'Content-Type': 'text/html'
    });
    res.end("Formulaire ok");
});

app.listen(8080,()=>{console.log("Server running on http://127.0.0.1:8080")});
```

Routing

Path

Express utilise le package *path-to-regexp (*https://www.npmjs.com/package/path-to-regexp) pour effectuer le match entre l'url et le path défini au niveau de notre route.

Nous pouvons donc définir une route avec un path simple /, /about, /home …..

Mais également utiliser des masques pour router :

- '/contact?about' : capture les deux url /contact ou /about
- '/ab+cd' : capture les routes /abcd, /abbcd,/abbbcd,….
- '/ab*cd' : capture les routes /abcd, /abxcd,/ab123cd, /abZorrocd….
- '/ab(cd)?e' : capture les routes /abe, /abcde

Routing

Construction des routes

Nous avons la possibilité de facilement construire nos routes pour permettre des segments statiques et dynamique

Exemple :

‘/:?’ permet d’avoir une route http://domain/{?} où ? est un paramètre passé à notre route

```
router.get('/:id', function(req, res) {
  console.dir(req.params);
  res.send('The id you specified is ' + req.params.id);

});
```

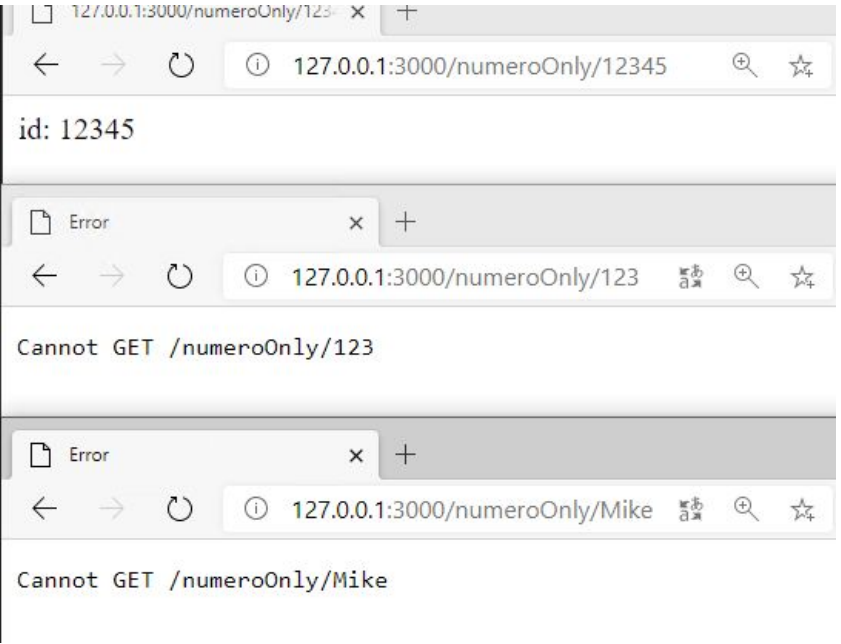
```
router.get('/Hello/:nom/:prenom', function(req, res) {
  res.send('Bonjour   ' + req.params.prenom + '   ' + req.params.nom);
});
```

Routing

Si nous désirons limiter les valeurs des paramètres, nous pourrons utiliser les « expression » correspondantes :

Exemple :

```
const express = require('express');
const router = express.Router();

router.get('/numeroOnly/:id([0-9]{5})', function(req, res){
  res.send('id: ' + req.params.id);
});

  module.exports = router;
```



Routing

Routers (https://expressjs.com/fr/4x/api.html#router**)**

Même si c'est « facile » de mettre en place les routes de la manière dont on vient de voir, maintenir ces routes au sein du fichier app.js peut s'avérer fastidieux.

Express nous propose d'utiliser un package appelé *Router*. (https://expressjs.com/fr/guide/routing.html)

Grâce à ce package, nous pouvons gérer les routes dans un module séparé ce qui facilitera notre maintenance mais également utiliser un middleware pour exécuter du code avant le traitement de la requête.

Son utilisation est simple :

1. Déclarer notre variable

```
const router = express.Router();
```

2. Utiliser les méthodes du router pour enregistrer nos routes

```
router.get('/', function(req, res) { …});
router.get('/home', function(req, res) { …});
…
```

Routing

/config/router.js

ExpressJs > Exemples > RouterSample > config > routes.js > ...

```
1   const express = require('express');
2   const router = express.Router();
3
4   // Home page
5     router.get('/', function(req, res) {
6         console.log("HOME");
7       res.send('Welcome to the home page');
8     });
9     // About Page
10    router.get('/about', function(req, res) {
11      res.send('About Me');
12    });
13
14    module.exports = router;
```

/app.js

ExpressJs > Exemples > RouterSample > app.js > ...

```
1   const express = require("express");
2   const app = express();
3   const PORT = process.env.PORT = 3000;
4
5   const router = require("./config/routes");
6
7
8   app.use('/',router);
9
10  app.listen(PORT,function(){
11    console.log('Server is running at http://127.0.0.1:'+PORT);
12  });
```

Routing

Chainage des routes

Il est possible de définir plusieurs fonctions qui devront être appelée suivant la route (pour par exemple effectuer un pré-traitement).

Pour cela, nous utiliserons un troisième paramètre des fonctions get,post,… qui permet au système de transmettre l'adresse de la fonction suivante.

!!!Attention!!!

- L'ordre d'écriture des routes dans le fichier induit l'ordre d'exécution
- Pas de possibilité de renvoyer deux fois une réponse

```
router.get('/', function(req, res,next) {
    console.log("HOME");
    next();
  res.send('Welcome to the home page ' + message);

});
router.get('/', function(req, res) {
  console.log("HOME 2 ");
  //un traitement ...
  message ="HOME 2 ";
});
```

```
router.get('/', function(req, res,next) {
    console.log("HOME");
    next();
  res.send('Welcome to the home page ');

});
router.get('/', function(req, res) {
  console.log("HOME 2 ");
  //un traitement ...
  res.send( message);
});
```

Exception

Routing

Les méthodes de réponses

- res.send([body])

  Envoie simplement une réponse http.

  Le body peut être :

  - Un buffer (pour un download de fichier par exemple)

```
//buffer
router.get('/buffer', function(req, res) {

  var wb = xlsx.utils.book_new();
  var table = [['a', 'b', 'c'], ['1', '2', '3']]
  var ws = xlsx.utils.aoa_to_sheet(table);
  xlsx.utils.book_append_sheet(wb, ws, 'test');

  // write options
  const wopts = { bookType: 'xlsx', bookSST: false, type: 'base64' };
  const buffer = xlsx.write(wb, wopts);

  res.writeHead(200, {'Content-Type':'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet;base64'
  ,"X-Content-Type-Options" :"nosniff",
  "Content-Disposition": "attachment; filename*=UTF-8''fichier.xlsx"});
  res.end(new Buffer(buffer, 'base64'));

});
```

Routing

- Un string

```
router.get('/string', function(req, res) {


  res.writeHead(200, {'Content-Type':'text/html' });
  res.end("<h1>Hello</h1>");

});
```

- Un json

```
router.get('/json', function(req, res) {

   let u = new users("Jhon","Smith");
   res.writeHead(200, {'Content-Type':'text/application/json' });
   res.end(JSON.stringify(u),"utf-8",()=>{console.log("Json envoyé")});

});
```

Routing

La méthode send est également applicable pour renvoyer un statut (également possible via la méthode sendStatus)

```
router.get('/404', function(req, res) {

  res.status(404).send('Sorry, we cannot find that!');

});

router.get('/204', function(req, res) {

 res.status(204).send('no content');

});
```

```
router.get('/404bis', function(req, res) {

 res.sendStatus(404);

});

router.get('/204bis', function(req, res) {

 res.sendStatus(204);

});
```

Routing

- res.sendFile() (à partir de la version 4,8 d'express)

  Envoie un fichier dans un flux.

```
router.get('/file/:name', function (req, res, next) {
  var options = {
    root: path.join(__dirname, '/../public')
  }

  var fileName = req.params.name
  res.sendFile(fileName, options, function (err) {
    if (err) {
      next(err)
    } else {
      console.log('Sent:', fileName)
    }
  })
})
```

Routing

Différentes options sont possibles pour la fonction sendFile

option	Description	Défaut	Version minimum
maxAge	Définit le temps de mise en cache du fichier	0	
root	Dossier root pour le chemin vers le fichier		
lastModified	Définit la dernière modification du fichier dans le header http	Enabled	4.9.0+
headers	Contient le HTTP headers envoyé avec le fichier		
acceptRanges	Active ou désactive le possibilité d'executer des ranges request (partial request)	true	4.14+
cacheControl	Active ou désactive le header cache-control	true	4.14+

Si nous préférons inviter au téléchargement un fichier, il vaut mieux utiliser la fonction *download*

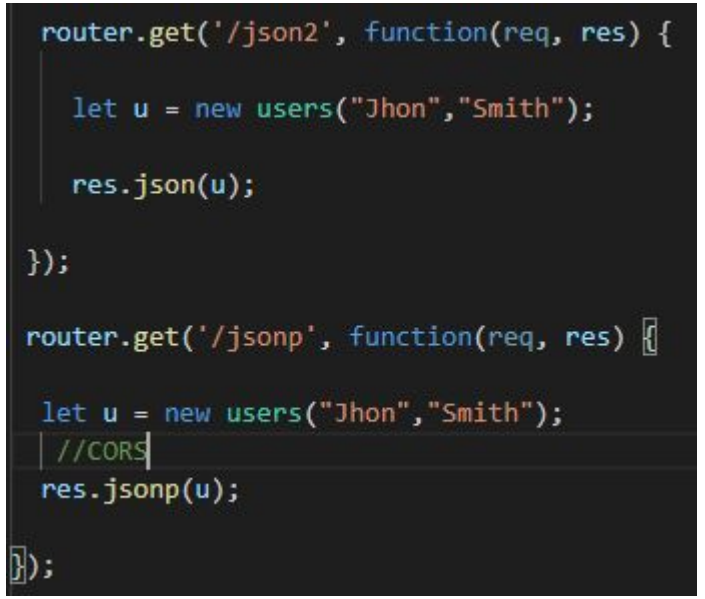
```
router.get('/LaughingSeal', function (req, res) {

  res.download("./public/th.jpg",  'LaughinSeal.jpg', function (err) {
    if (err) {
      console.log(err);
    } else {
      console.log("Fichier téléchargé");
    }
  });
})
```

Routing

- res.json() & res.jsonp

  Envoi le paramètre transmis à la fonction en utilisant Json.Stringify() et en renseignant correctement le Content-Type l'en-tête http.

  La version jsonp est identique mais en activant les options jsonp (utilisation de l'élément script plutôt que l'XmlHttpRequest pour transférer les datas permet de mettre en place facilement le CORS)



Routing

- res.redirect(path)

  Permet d'effectuer une redirection vers une autre chemin en spécifiant un code de statut (par défaut: 302 – redirection non permanente)

  Le path peut-être un chemin relatif ou une url complète.

```
router.get('/google', function(req, res) {
  res.redirect("http://www.google.be");
});
router.get('/reJsonp', function(req, res) {
  res.redirect("/jsonp");
});
router.get('/permanent', function(req, res) {
  res.redirect(302,"/jsonp");
});
```

Middleware

eXpress JS

Middleware

Les middleware sont des fonctions qui on accès à la request,response et au next.

Elles permettent donc de modifier, capturer, logger, … ces différents objets

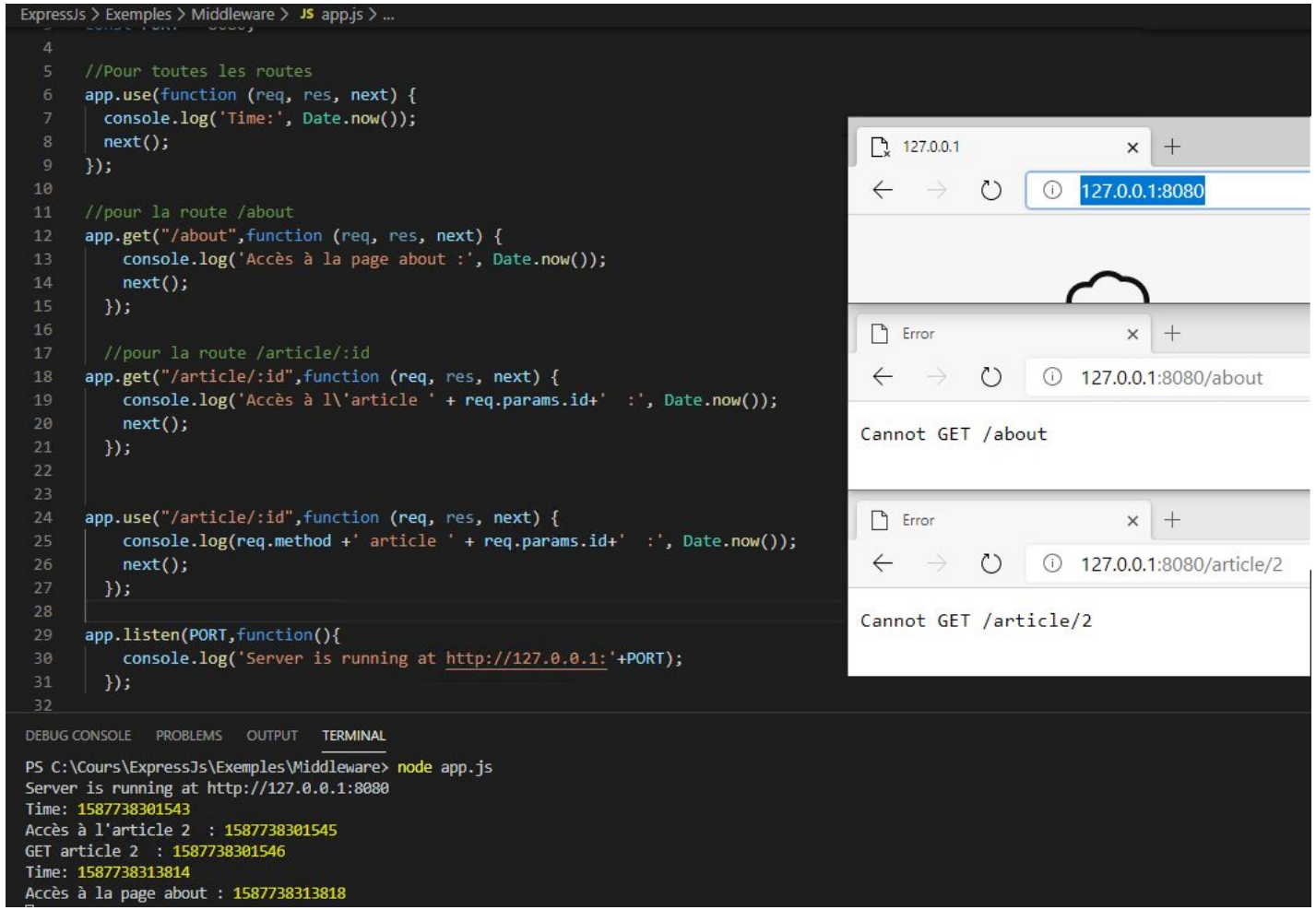
Il y a plusieurs type de Middleware supportés par eXpress :

- Application-level Middleware
- Router-level Middleware
- Error-Handling Middleware
- Built-in Middleware
- Third-Party Middleware

Middleware

Application-Level

Permet de capturer toutes les request au niveau applicatif, quel que soit la route demandée ou en ciblant précisément la route .



Middleware

Router-Level

Même comportement que l'application level MAIS il est lié à une instance de *Router*

ExpressJs > Exemples > Middleware > JS appRouter.js > app.listen() callback

```
1  const express = require("express");
2  const app = express();
3  const router = express.Router();
4  const PORT = 8080;
5   // utiliser router pour l'app
6  app.use('/', router);
7
8  //Pour toutes les routes
9  router.use(function (req, res, next) {
10   console.log('Time:', Date.now());
11   next();
12 });
13
14 //pour la route /about
15 router.get("/about",function (req, res, next) {
16       console.log('Accès à la page about :', Date.now());
17       next();
18   });
19
20   //pour la route /article/:id
21   router.get("/article/:id",function (req, res, next) {
22       console.log('Accès à l\'article ' + req.params.id+'  :', Date.now());
23       next();
24   });
25
26
27   router.use("/article/:id",function (req, res, next) {
28       console.log(req.method +' article ' + req.params.id+'  :', Date.now());
29       next();
30   });
31
32 app.listen(PORT,function(){
33       console.log('Server is running at http://127.0.0.1:'+PORT);
34   });
35
36
```

DEBUG CONSOLE PROBLEMS OUTPUT TERMINAL

```
PS C:\Cours\ExpressJs\Exemples\Middleware> node .\appRouter.js
Server is running at http://127.0.0.1:8080
Time: 1587738611101
Accès à la page about : 1587738611104
Time: 1587738624552
Accès à l'article 1  : 1587738624553
GET article 1  : 1587738624553
```

Middleware

Error-handling Level

Contrairement aux exemples précédent, la fonction qui devra capturer les erreurs aura TOUJOURS 4 arguments

(error, request, response,next)

Remarque :

La fonction doit TOUJOURS être la dernière dans le stack d'appel de express. (après les routes)

Sinon elle ne sera pas appelée

```
const express = require("express");
const fs = require("fs");
const app = express();
const PORT = 8080;

//Pour toutes les routes

app.get("/", function(req,res)
{

   throw new Error('BROKEN')

})

app.use(function (err, req, res, next)
{
   //LOg dans un fichier
   fs.appendFileSync('./logs/error.txt', Date.now() +": " + err +"\r\n", function(fserr) {
      if (fserr) {
         return console.error(fserr);
      }});

   if (res.headersSent) {
      //bonne pratique : en cas de double écriture dans le header ==>on laisse express se charger de l'erreur
     return next(err)
   }
   res.status(500);
   res.render('error', { error: err })
 });


app.listen(PORT,function(){
   console.log('Server is running at http://127.0.0.1:'+PORT);
 });
```

Middleware

Middlewares intégrés

eXpress JS intègre des middleware permettant de travailler avec les cookies, de parser l'html,…

Vous trouverez la liste de ceux-ci ici : https://github.com/senchalabs/connect#middleware

Middlewares Tiers

Il y a également une quantité intéressantes de middleware tiers nous facilitant la vie.

Vous trouverez une liste (non-exhaustive) de ceux-ci à cette adresse : https://expressjs.com/en/resources/middleware.html

Templating

eXpress JS

Templating

La mise en page Html passe ,comme pour beaucoup d'autres Framework, par un moteur de Template.

Le moteur de Template associé traditionnellement avec eXpress s'appelle *Pug (anciennement appelé Jade)*.

Pug est un moteur de templates implémenté en JavaScript qui permet de générer dynamiquement du HTML, à l'instar de Thymeleaf en Java et Twig en PHP.

Sa syntaxe est inspirée de Haml (http://haml.info/) et est donc minimaliste et basée sur les indentations.

Pour utiliser *Pug* avec eXpress, nous devons l'installer :

```
npm install --save pug
```

Nous devons ensuite spécifier à eXpress qu'il doit utiliser *Pug* comme moteur de template.

Pas besoin d'utiliser un *require* mais simplement effectuer un *set* au niveau de l'app pour le moteur et définir le dossier contenant les templates

```
app.set('view engine', 'pug');
app.set('views','./views');
```

Templating

Premier test avec Pug

Pour tester notre moteur, nous pouvons maintenant définir un fichier *Home.pug* dans le dossier views et utiliser la méthode *render* au niveau de notre route.

ExpressJs > Exemples > TemplatingPug > views > home.pug

```
doctype html
html
    head
        title = "Hello From Pug"
    body
        p Hello From Pug!
```

ExpressJs > Exemples > TemplatingPug > config > routes.js > ...

```
const express = require('express');
const router = express.Router();

// Home page
  router.get('/', function(req, res,next) {
    res.render('home');
  });

  module.exports = router;
```

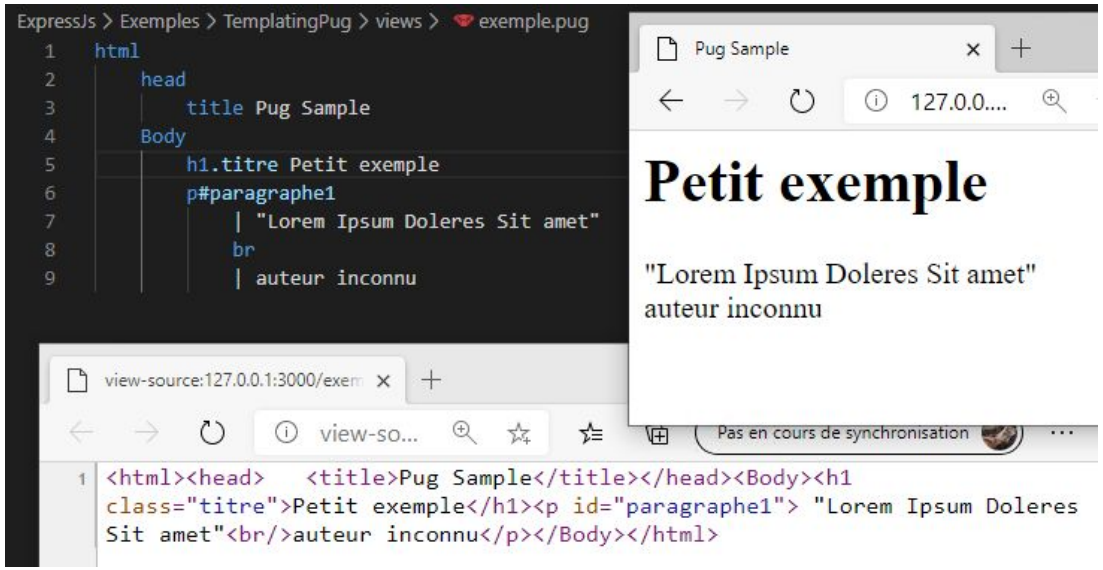
Templating

La syntaxe Pug

Le nom des balises dans Pug est représenté par des sélecteurs inspirés de la syntaxe du CSS.

L'utilisation de balises n'est plus nécessaire, grâce à un système d'indentations.

Les classes et les id sont définis par des raccourcis, respectivement "." et "#".



Templating

La documentation de Pug (https://pugjs.org/language/attributes.html) étant très complète et structurée, nous n'allons pas parcourir toutes la syntaxes mais nous intéresser à certains concepts.

Includes

Pug permet d'intégrer le contenu d'un fichier dans l'autre via le mot clé *include*

include ***relativePath***

Remarque :

- Si le chemin est absolu /… , il sera résolu à partir du chemin renseigné dans notre fichier principal (app.set('views','./views'))
- Si nous ne renseignons pas l'extensions du fichier, l'extension *pug* est ajouté automatiquement

Templating

ExpressJs > Exemples > TemplatingPug > views > header.pug

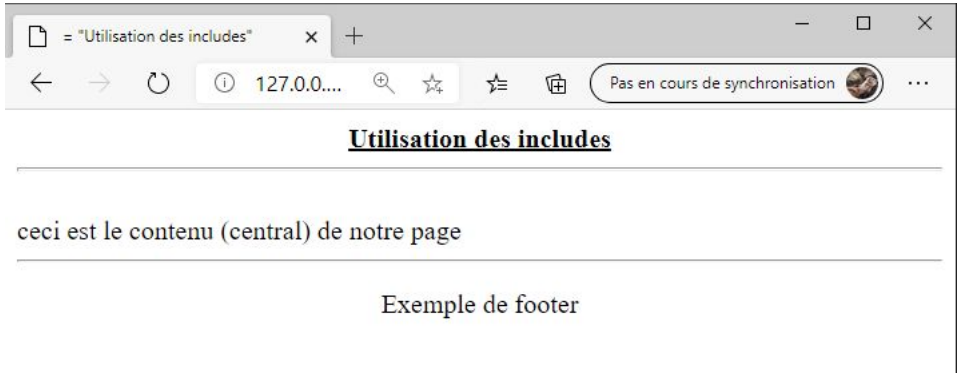
```
doctype html
html
  head
    title = "Utilisation des includes"
  body
    center
      b
        u
          |Utilisation des includes
    hr
    br
```

ExpressJs > Exemples > TemplatingPug > views > includeSample.pug

```
include header
.content
  |ceci est le contenu (central) de notre page
include partials/footer
```

ExpressJs > Exemples > TemplatingPug > views > partials > footer.pug

```
hr
footer
  center
    p
      | Exemple de footer
```



Templating

Include peut être utilisé également pour inclure des fichiers ***non-pug***.

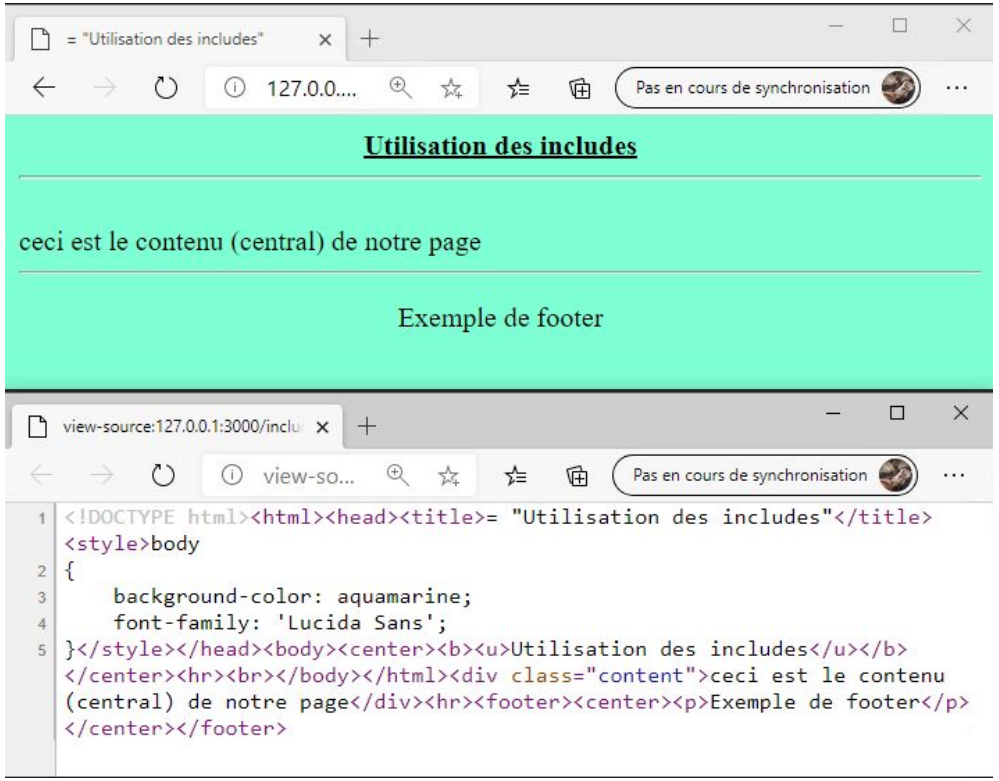
Par exemple, une feuille de style :

header.pug

```
doctype html
html
  head
    title = "Utilisation des includes"
    style
      include ../css/styles.css
  body
    center
      b
        u
          |Utilisation des includes
    hr
    br
```

styles.css

```
body
{
    background-color: aquamarine;
    font-family: 'Lucida Sans';
}
```



Templating

Héritage

Pug va plus loin en proposant un héritage entre les templates (un peu comme Twig en Php).

Grâce à ce principe, nous pouvons définir des ***block*** à l'intérieur de nos templates afin de permettre lors de **l'*extend*** de celui-ci la redéfinition/remplacement du contenu de ces block.

La syntaxe est plutôt simple : *block nom_du_block*

ExpressJs > Exemples > PugHeritage > views > home.pug

```
1  extends layout.pug
2
3  block Contenu
4    h1= title
5    - var pets = ['cat', 'dog']
6    each petName in pets
7      include pet.pug
8
9
```

ExpressJs > Exemples > PugHeritage > views > layout.pug

```
72                    Browse Gallery
73
74
75
76          .container
77            block Contenu
78
79
80
81          footer container
```

Quand à l'héritage : extends nom_fichier_pug

Templating

Un block peut redéfinir des block,etc… ce qui permet de l'héritage sur plusieurs niveaux.

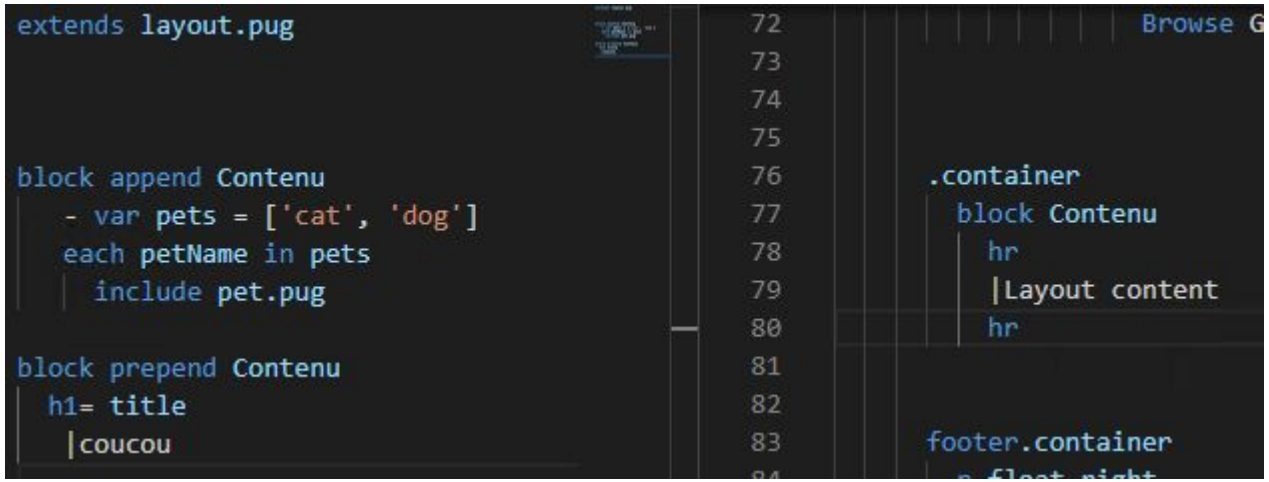
Par défaut, *Pug* remplace el contenu du block par le contenu défini dans l'enfant MAIS il est possible d'utiliser deux directives :

- prepend

  permet d'ajouter du contenu en début du block

- append

  permet d'ajouter du contenu après le block

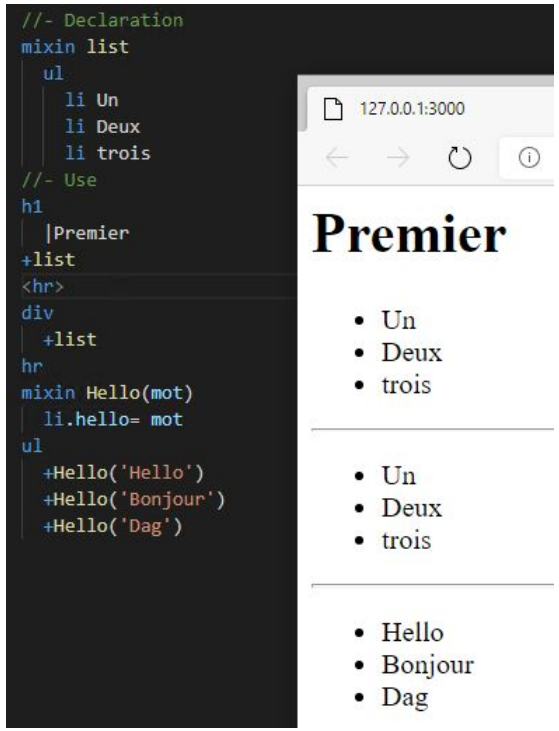


Templating

Mixin

Pug nous permet également de créer des block réutilisables.

Cela se fait via le mot clé *mixin* qui peut prendre des paramètres afin de permettre d'écrire des composant le plus générique possible



Templating

Il est également possible de définir des *Mixin* pour les block et les attributes

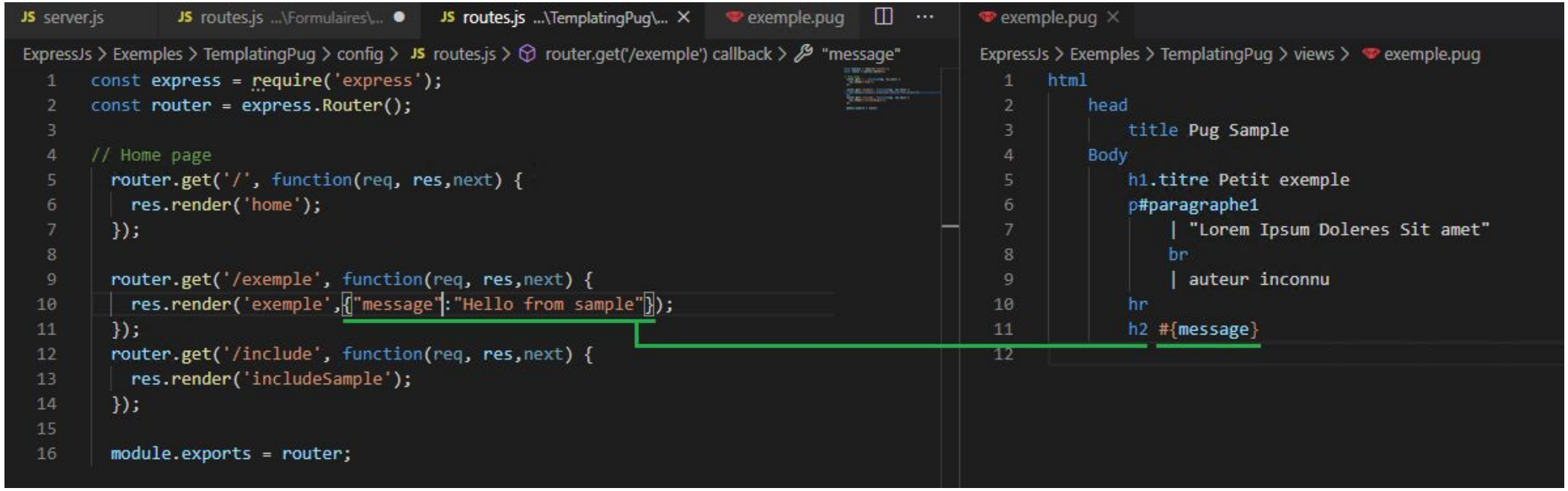
```
mixin article(title)
  .article
    .article-wrapper
      h1= title
      if block
        block
      else
        p No content provided

+article('Hello world')

+article('Hello world')
  p This is my
  p Amazing article


mixin link(href, name)
  //- attributes == {class: "btn"}
  a(class!=attributes.class href=href)= name

+link('/foo', 'foo')(class="btn")
```

Templating

Transmettre des données au template

Lorsque nous désirons transmettre des données (un Model) au template, nous utiliserons le deuxième paramètre de la fonction render.

Et dans notre template, nous utiliserons la syntaxe : *#{variable|objet}*



Les fichiers static

eXpress JS

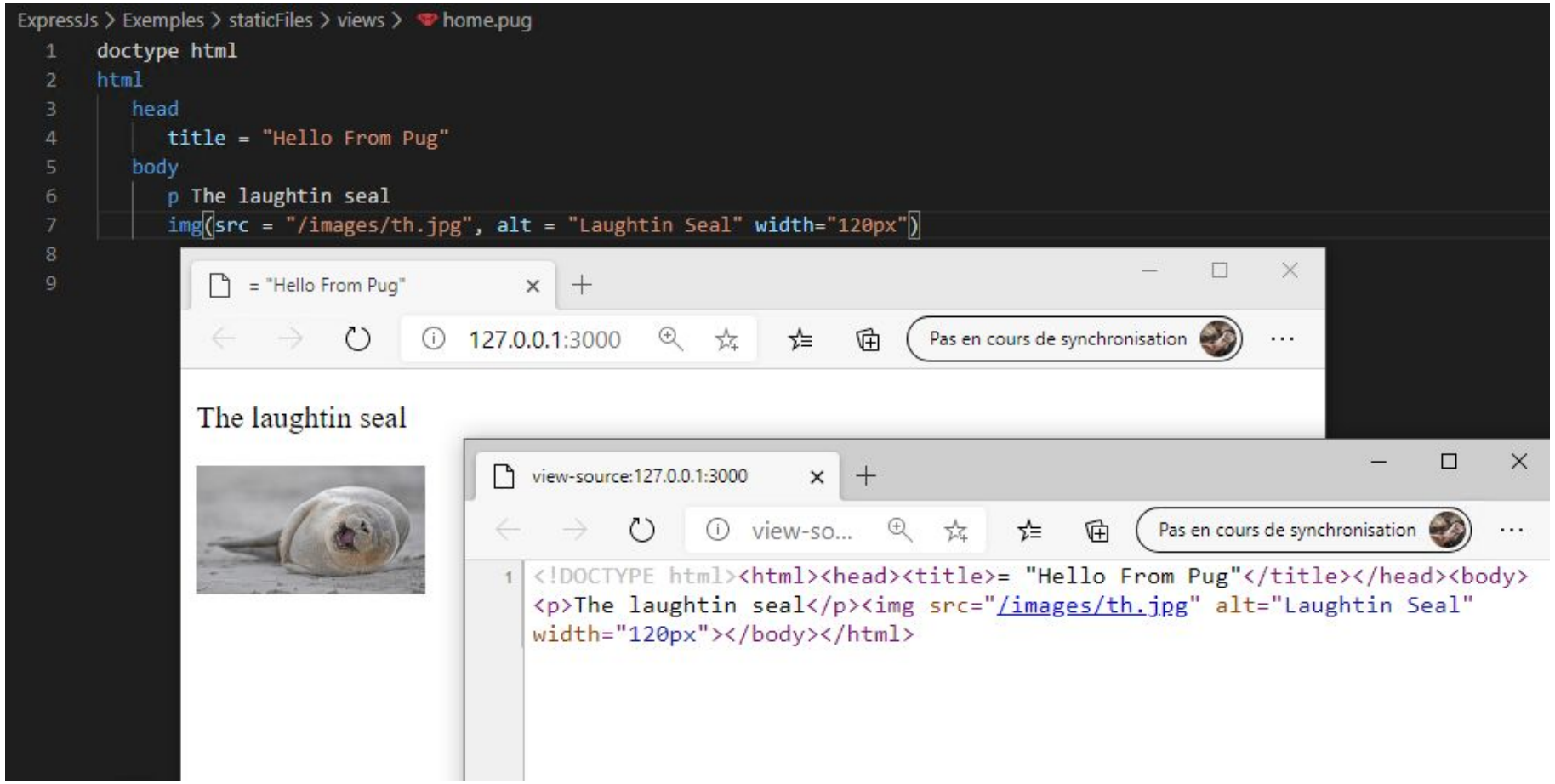
Les fichiers static

Pour permettre l'utilisation de fichiers statiques (css, images, …), nous devons configurer le middleware d'application avec la méthode static de eXpress.

ExpressJs > Exemples > staticFiles > JS server.js > ...

```
1   const express = require("express");
2   const app = express();
3   const PORT = process.env.PORT = 3000;
4
5   const router = require("./config/routes");
6
7   app.use(express.static("public"));
8   app.use('/',router);
9
10  app.set('view engine', 'pug');
11  app.set('views','./views');
12
13  app.listen(PORT,function(){
14    console.log('Server is running at http://127.0.0.1:'+PORT);
15  });
```

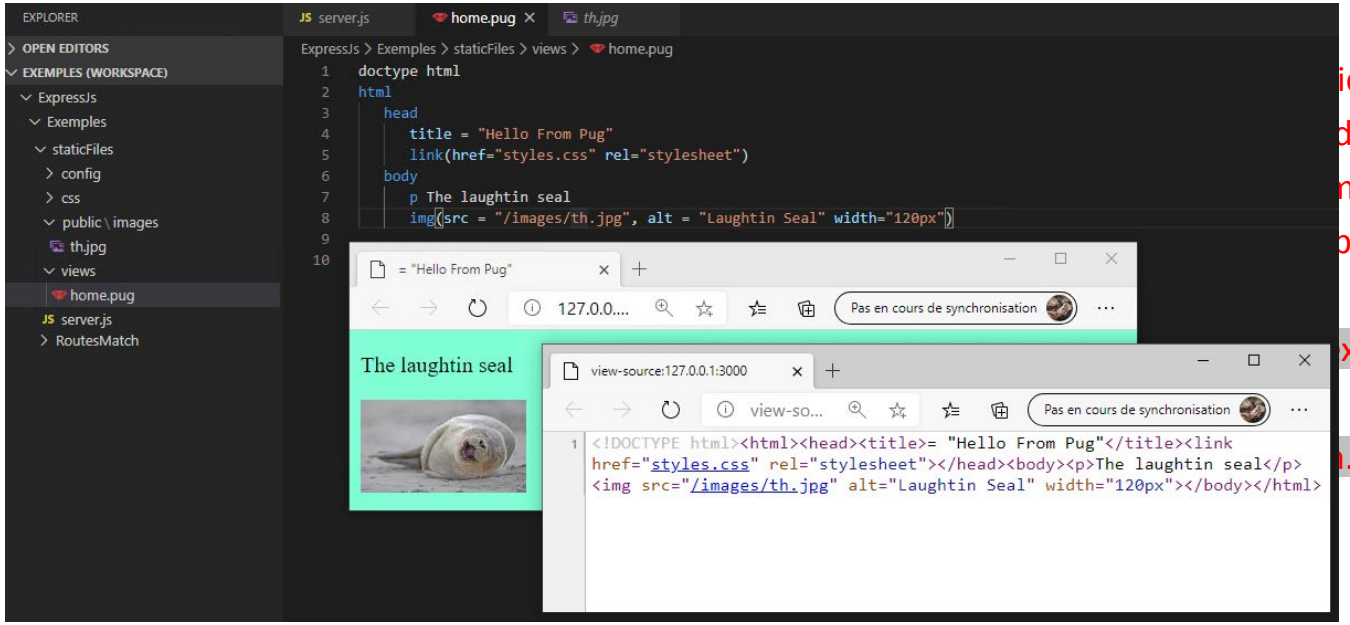
Les fichiers static



Les fichiers static

Au besoin, nous pouvons également définir plusieurs dossiers contenant les fichiers statiques qui doivent être gérés par eXpress

```
app.use(express.static('public'));
app.use(express.static('images'));
app.use(express.static(css'));
...
```



ichier static se fait côté express ce
dans le code source, nous n'ayons
mplet de la ressource,
pendant définir un préfix virtuel pour

xpress.static('public'))

.jpg dans l'html

Les formulaires

eXpress JS

Les formulaires

Toutes les applications web ont besoin de formulaire : contact, commande, forum, ….

eXpress Js nous propose d'utiliser deux dépendances : *body-parser(parser le Json et les données url-encoded)* et *multer(parser les données multipart/form)*

Avant de commencer nous aurons donc la commande suivante a effectuer : npm install --save body-parser multer

Et ensuite, nous configurons les middlewares comme suit :

```
const bodyParser=require("body-parser");
const multer = require("multer");
const upload = multer();
```

```
const router = require("./config/routes");


// pour parser les données de type mime application/json
app.use(bodyParser.json());

// pour parser les données de type mime application/xwww-form-urlencoded
app.use(bodyParser.urlencoded({ extended: true }));


// pour parser les données de type mime multipart/form-data
app.use(upload.array());

//Définition dossier pour le contenu static
app.use(express.static('public'));

//Utilisation du Middleware router
app.use('/',router);

//Définition de Pug comme moteur de vue
app.set('view engine', 'pug');
//Définition du dossier views pour contenir mes templates
app.set('views','./views');
```

Les formulaires

Body-parser (https://github.com/expressjs/body-parser**)**

Ce package permet d'intercepter le body du Handler http avant le traitement par notre route.

En définissant au niveau application `app.use(bodyParser.json());` toutes les routes seront gérées après que ce package ait manipulé et parser le body contenant du json.

Il est également possible de ne cibler que certaines routes via la syntaxe suivante :

```
// create application/json parser
var jsonParser = bodyParser.json()

// create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

// POST /login gets urlencoded bodies
app.post('/login', urlencodedParser, function (req, res) {
  res.send('Bienvenue, ' + req.body.username)
})

// POST /api/users gets JSON bodies
app.post('/api/users', jsonParser, function (req, res) {
  // create user in req.body
})
```

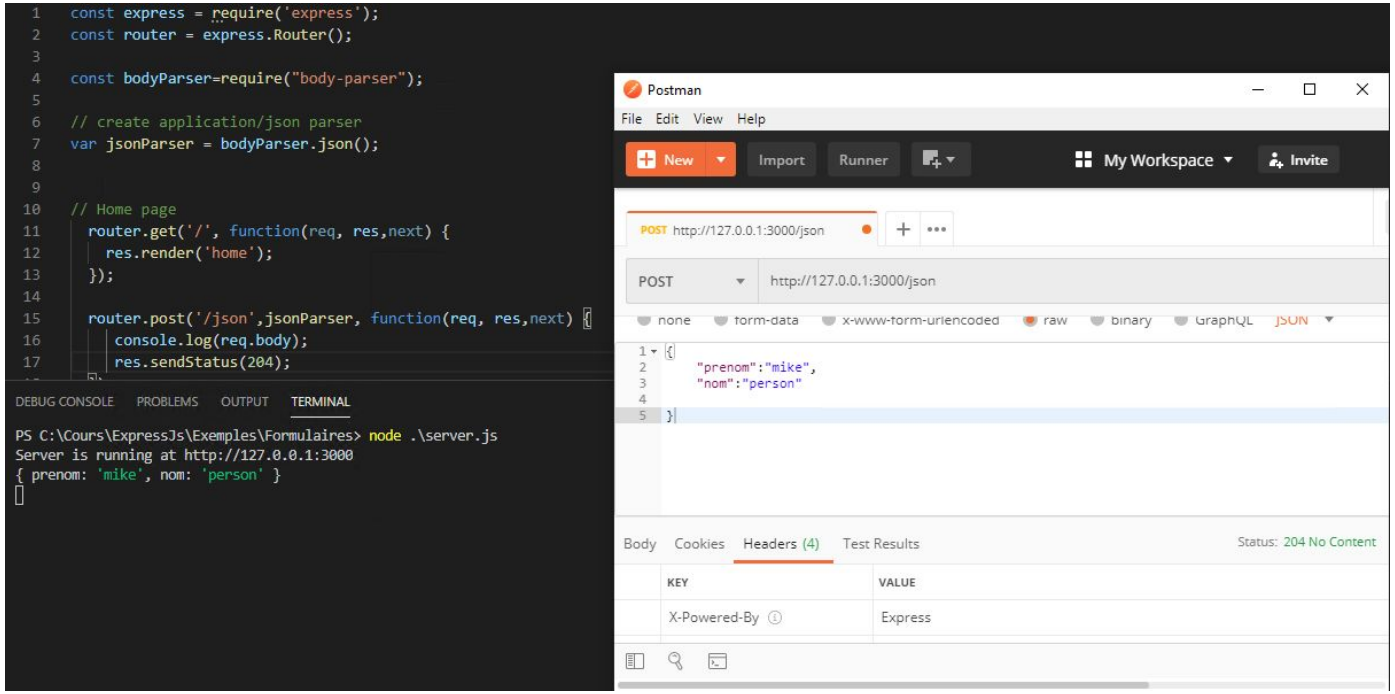
Les formulaires

Les différents parser

1. **bodyParser.json([options])**

   Renvoie un middleware qui analyse uniquement le format json et ne regarde que les requêtes dont l'en-tête *Content-Type* correspond à l'option définie. Ce parser accepte tout encodage Unicode du corps et prend en charge les encodages gzip.

   req.body contiendra les données parsées.



Les formulaires

Option	Description
inflate	Si *True*, le contenu compressé sera décompressé si *False,* le contenu compressé sera rejeté
limit	Limite la taille du body exprimé en kilobyte. Par défaut : 100kb
reviver	Permet de définir la fonction utilisée pour transformer le json une fois parsé. (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse#Example.3A_Using_the_reviver_parameter)
strict	Si T*rue*, le parser n'accepte que les array et object. Si *False,*le parser accepte tout contenu pouvant être utilisé par JSON.parse
type	Définit le type de média accepté. Ça peut être une fonction ou un type tel "application/json", "*.*", …. Par défaut : "application/json"

Les formulaires

2. **bodyParser.raw([options])**

   Renvoie un middleware qui analyse uniquement le format en tant que *Buffer* et ne regarde que les requêtes dont l'en-tête *Content-Type* correspond à l'option définie. Ce parser accepte tout encodage Unicode du corps et prend en charge les encodages gzip.

   req.body contiendra un buffer avec les données transmises.

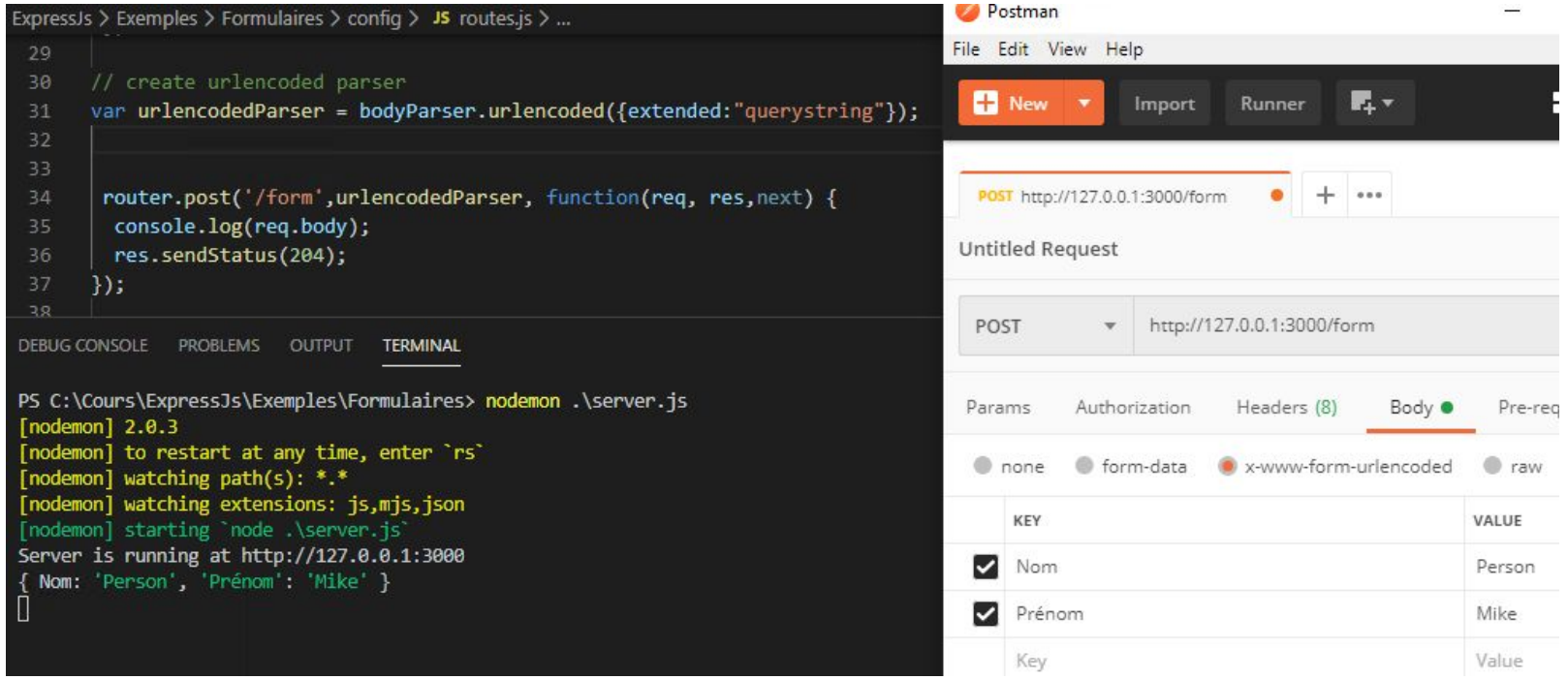
Option	Description
inflate	Si *True*, le contenu compressé sera décompressé si *False,* le contenu compressé sera rejeté
limit	Limite la taille du body exprimé en kilobyte. Par défaut : 100kb
type	Définit le type de média accepté. Ça peut être une fonction ou un type tel "application/octet-stream", "application/*", …. Par défaut : "application/octet-stream"

Les formulaires

3. **bodyParser.urlencoded([options])**

   Renvoie un middleware qui analyse uniquement le format en *urlencoded* et ne regarde que les requêtes dont l'en-tête *Content-Type* correspond à l'option définie. Ce parser accepte uniquement le codage UTF8 du body et prend en charge les compressions/décompression gzip.

   req.body contiendra un object Key/value contenant les données du body



Les formulaires

Option	Description
extended	Vous permet de choisir entre *querystring* ou qs (https://www.npmjs.com/package/qs#readme) pour parser le body. !!!Pas de défaut, vous devez définir une valeur!!!
inflate	Si *True*, le contenu compressé sera décompressé si *False,* le contenu compressé sera rejeté
limit	Limite la taille du body exprimé en kilobyte. Par défaut : 100kb
parameterLimit	Définit la maximum de paramètre pouvant être passé. Par défaut : 1000
strict	Si T*rue*, le parser n'accepte que les array et object. Si *False,*le parser accepte tout contenu pouvant être utilisé par JSON.parse
type	Définit le type de média accepté. Ça peut être une fonction ou un type tel "application/x-www-form-urlencoded", "*/x-www-form-urlencoded" Par défaut : "application/x-www-form-urlencoded"

Les formulaires

***multer(*https://www.npmjs.com/package/multer)**

Le but de ce package est de gérer les formulaires envoyés uniquement en *multipart/form-data.*

Il est généralement utilisé pour l'upload de fichiers.

Après avoir importé multer, nous devons créer une instance de multer. `const upload = multer();`

Des options sont disponibles lors de la création :

dest or storage	Dossier pour stocker les fichiers (exemple : multer({des:'/public/upload/'})
fileFilter	Permet de filter les extensions des fichiers autorisés Exemple : `const upload = multer({ storage: storage, fileFilter: function (req, file, callback) { var ext = path.extname(file.originalname); if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg') { return callback(new Error('Only images are allowed')) } callback(null, true) } })`
limits	Taille limite du fichier
preservePath	Garde le chemin complet du fichier plutôt que simplement le nom de base

Les formulaires

Remarques :

- Le dossier de destination doit exister
- Multer ne rajoute pas l'extension au fichier, c'est à nous de le faire
- Les fonction doivent être déclarées avant leur utilisation dans le signature de multer

Pour plus d'infos :

http://expressjs.com/en/resources/middleware/multer.html

```
var storage = multer.diskStorage({
    destination: function(req, file, cb) {
        cb(null, './public');
    },
    filename: function (req, file, cb) {
        cb(null , file.originalname);
    }
})
var checkFile =function (req, file, callback)
  {
    var ext = path.extname(file.originalname);
    if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg')
        {
            return callback(new Error('Only images are allowed')) }
    callback(null, true) }

var upload2 =  multer({ storage:storage, fileFilter:checkFile, });

router.get("/upload2", function(req,res)
{
    res.render("frmUpload2");
});

router.post("/upload2",upload2.single("userFile"), function(req,res)
{
    console.log("Fichier uploadé");
```

Base de données

eXpress JS

Base de données

La grand majorité des bases de données sont utilisables avec eXpress, cependant, c'est la base de données NoSql MongoDB qui a la préférence des développeurs.

Pour communiquer avec MongoDB, nous avons besoin d'un client : Mongoose.

npm install --save mongoose

Mongoose est un outils de modélisation d'objets conçu pour fonctionner dans un environnement asynchrone.(ODM)

```
//Importation de express
const express = require("express");

//Importation du module mongoose
const mongoose = require('mongoose');


const app = express();
const PORT = process.env.PORT = 3000;
const router = require("./config/routes");


//Définition de la connexion Db par défaut
var mongoDB = 'mongodb://127.0.0.1/ExpressSample';
mongoose.connect(mongoDB, { useNewUrlParser: true , useUnifiedTopology: true });

//récupération de la connexion
var db = mongoose.connection;

//Gestion des erreurs éventuelles
db.on('error', console.error.bind(console, 'MongoDB connection error:'));

//Définition dossier pour le contenu static
app.use(express.static('public'));

//Utilisation du Middleware router
app.use('/',router);

//Définition de Pug comme moteur de vue
app.set('view engine', 'pug');
//Définition du dossier views pour contenir mes templates
app.set('views','./views');

app.listen(PORT,function(){
  console.log('Server is running at http://127.0.0.1:'+PORT);
});
```

Base de données

Définition des models et sauvegarde en DB

Nous pouvons ensuite créer des modèles à partir d'un schéma.

Ce modèle par la suite sera utilisé par mongoose pour enregistrer le document dans MongoDB.

```
//Importation du module mongoose
const mongoose = require('mongoose');

var personSchema = mongoose.Schema({
    name: String,
    age: Number,
    nationality: String
}, { collection: 'Person' });

module.exports = personSchema
```

Schéma vers Model

Sauvegarde dans la DB

```
router.post("/person",urlencodedParser,function(req,res)
{
  console.log("Posted");
  var Person = mongoose.model("Person", personSchema);
  var personInfo = req.body; //Get the parsed information

  if(!personInfo.name || !personInfo.age || !personInfo.nationality){
     res.render('show_message', {
        message: "Vous devez completer le formulaire", type: "error"});
  } else {
     var newPerson = new Person({
        name: personInfo.name,
        age: personInfo.age,
        nationality: personInfo.nationality
     });

     newPerson.save(function(err, Person){
        if(err)
           res.render('show_message', {message: "Erreur DB", type: "error"});
        else
           res.render('show_message', {
              message: "La personne a été ajouté à la DB", type: "success", person: personInfo});
     });
  }
})
```

Base de données

Récupérer les données

Mongoose nous fournit beaucoup de fonctions pour retrouver des documents que vous pouvez retrouver sur https://mongoosejs.com/docs,

En voici 3 qui peuvent déjà nous permettre de gérer la récupérations des données pour une application web.

- Model.find

  Cette fonction permet via les conditions renseignées de retrouver un document sur MongoDB.

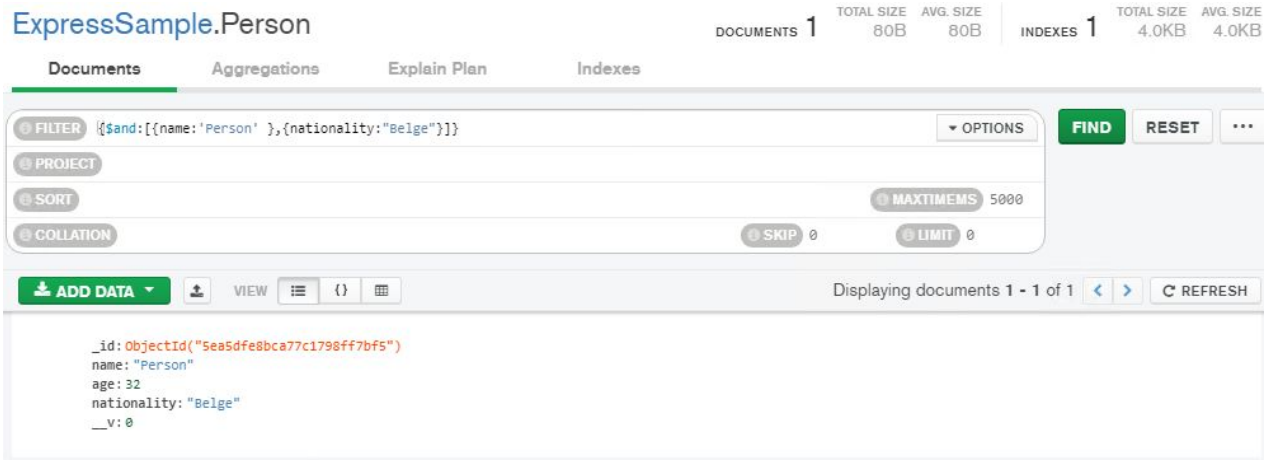
  - Conditions

    Nous utiliserons les même opérators utilisé dans MongoDB

    Exemple :

    Je cherche une personne dont le
    nom est Person et la nationalité est Belge.

    Via MongoDB Compass



Base de données

Via Model.find sous express

Code-behind

```
router.get("/getPerson", async function(req,res)
{

 var Person = mongoose.model("Person", personSchema);
 await Person.find({$and:[{name:'Person' },{nationality:"Belge"}]}).exec(function(err,Model)
  {
    if(err === null)
    {
      res.render("personDetail",{Model :Model});
    }

  });


})
```

Template

ExpressJs > Exemples > Database > views > personDetail.pug

```
1  html
2  head
3     title Person
4     body
5           h1
6               |Résultats de la recherche
7           hr
8           p
9               each  M in Model
10                  |Name : #{M.name}
11                  br
12                  |Age : #{M.age}
13                  br
14                  |Nationalité : #{M.nationality}
15              else
16                  b
17                      |Pas de résultats
```

Base de données

- Model.findOne

  Model.find nous renvoie une collection reprenant les documents récupérés.

  Mais si nous ne voulons en récupérer qu'1!, nous utiliserons plutôt findOne avec les même arguments.

  Cette fonction renverra le document le plus pertinent par rapport au filtres transmis

- Model.findById

  MongoDB génère un _id pour chaque document et il est donc possible de rechercher un document via cet _id.

```
router.get("/Person/:id", async function(req,res)
{

 var Person = mongoose.model("Person", personSchema);
 await Person.findById(req.params.id).exec(function(err,Model)
  {
    if(err === null)
    {
      console.log(Model);
      res.render("personDetails",{Model :Model});
    }

  });


})
```

Base de données

Mettre à jour

Model.updateOne

Cette méthode prend en paramètre une condition et met à jour 1! seul document qui matche (le plus pertinent)

```
router.get("/changeNationnality/:name", async function(req,res)
{
  console.log(req.params.name);
  var Person = mongoose.model("Person", personSchema);
  await Person.updateOne({name:"Person"},{nationality:"American"}, function(err,response)
  {
    res.render("show_message", {message:"Personne mise à jour"});
  });

})
```

Model.deleteOne

Cette méthode prend en paramètre une condition et supprime 1! seul document qui matche (le plus pertinent)

```
router.delete("/Person/:id", async function(req,res)
{
 var Person = mongoose.model("Person", personSchema);
 await Person.deleteOne({"_id":req.params.id},function(err,response)
  {
    if(err === null)
    {
      res.render("show_message", {message:"Personne supprimée"});
    }
  });
})
```

Sessions et Cookies

eXpress JS

Cookies & Sessions

Cookies

Les cookies sont de petits fichiers / données simples qui sont envoyés au client avec une demande du serveur et stockés côté client.

Chaque fois que l'utilisateur charge à nouveau le site Web, ce cookie est envoyé avec la request.

Les cookies sont nottament utilisés pour

- La gestion de session Personnalisation (systèmes de recommandation)
- Le Tracking des utilisateurs

Le package npm qui est utilisé avec express est : cookie-parser

```
npm i cookie-parser
```

Après l'installation, nous devons simplement importer le module

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
```

Cookies & Sessions

Définition et suppression d'un cookie

Pour définir un nouveau cookie, nous utilisons simplement la méthode cookie de l'objet response via la syntaxe suivante :

```
res.cookie(name_of_cookie, value_of_cookie);
```

Pour supprimer un cookie :

```
clearCookie('cookie_name');
```

ExpressJs > Exemples > cookies > config > JS routes.js > [@] <unknown>

```
1   const express = require('express');
2   const cookie = require("cookie-parser");
3   const router = express.Router();
4
5   router.use(cookie());
6
7   router.get("/cookie", function(req,res)
8   {
9       res.cookie("AppCookie" , 'MikeSite').send('Cookie is set');
10  });
11
12  router.get('/', function(req, res) {
13      console.log("Cookies :  ", req.cookies);
14    });
15
16  module.exports = router;
```

Cookies & Sessions

Secrets et options

Lors de l'instanciation du cookie-parser, nous pouvons passer des paramètres.

-Secret

Permet de spécifier un string qui signera le cookie afin d'éviter toute manipulation hors-site de celui-ci

Nous pouvons ensuite définir les options du cookies lors de sa création (voir les options sur https://www.npmjs.com/package/cookie)

ExpressJs > Exemples > cookies > config > routes.js > router.get("/cookie") callback

```
1   const express = require('express');
2   const cookie = require("cookie-parser");
3   const router = express.Router();
4
5   router.use(cookie("4526AD2514AE7854F236C254E785"));
6
7   router.get("/cookie", function(req,res)
8   {
9       res.cookie("TrackMe","Big Brother is watching you",{signed:true ,sameSite:"strict",
10                                                          expires: new Date(Date.now() + 900000),
11                                                          httpOnly: true})
12                                                          .send('Cookie is set');
13  });
14
15  router.get('/', function(req, res) {
16      console.log("Cookies :  ", req.cookies);
17    });
18
19  module.exports = router;
```

Cookie & Sessions

Les sessions

Comme le protocole htpp est « stateless » (il nous oublie après chaque réponse), nous devons trouver un moyen pour garder un contact avec notre internaute : les sessions

Comme nous pouvons désormais utiliser les cookies, nous allons utiliser ce principe pour mettre en place les sessions.

Nous utiliserons le package :Express-session

npm i express-session

Et nous pourrons l'inclure dans notre app via

const session = require ('express-session')

Et configurer le middleware via

app.use(session({'secret':'….'}));

```
1  const express = require("express");
2  const app = express();
3  const session = require("express-session");
4  // npm install dotenv Permet d'utiliser un fichier .env qui
5  //stockera notre config pour la clé session, le port, l'environement,...
6  const dotenv = require('dotenv');
7  dotenv.config();
8  const PORT = process.env.PORT;
9
10 const router = require("./config/routes");
11
12 app.use(express.static("public"));
13 app.use(express.static("css"));
14 app.use(session
15   ({
16       resave :true,
17       saveUninitialized:true,
18       secret:process.env.SESSION_KEY,
19
20   }));
21 app.use( / ,router);
22
23 app.set('view engine', 'pug');
24 app.set('views','./views');
25
26 app.listen(PORT,function(){
27   console.log('Server is running at http://127.0.0.1:'+PORT);
```

Cookies & Sessions

Remarque :

Depuis la version 1.5.0, le middleware cookie-parser n'est plus requis

Les options

Option	Description
cookie	Définit le cookie pour la session. Par défaut celui-ci est créé avec les options suivante : *{ path: '/', httpOnly: true, secure: false, maxAge: null }*
resave	Force l'enregistrement de la session même si celle-ci n'a pas été modifiée
saveUninitialized	Force l'enregistrement de la session nouvellement créée mais non encore modifiée. Mettre à false est interessant pour limiter l'utilisation du storage, pour les logins ou pour respecter la loi GDPR qui demande une acceptation avant de sauver des infos dans les cookies.
Secret (obligatoire)	Le token utilisé pour signer le cookie associé à la session
Store	Par défaut 'Memory store'. Mais il est également possible de stocker la session en Db (https://gabrieleromanato.name/creating-and-managing-sessions-in-expressjs)

Cookies & Sessions

Utilisation des sessions

L'objet request contient une propriété session qui nous permet de manipuler les sessions. La session est stockée sous le format json.

Nous pouvons détruire la session via la méthode destroy();

Plus d'infos :
https://www.npmjs.com/package/express-session

ExpressJs > Exemples > Sessions > views > home.pug

```
doctype
html
    body
        h1

            - if(session.User)
                | Session existante
            - else
                | Pas de sessions
```

ExpressJs > Exemples > Sessions > config > routes.js > ...

```
const express = require('express');
const router = express.Router();


router.get("/session", function(req,res)
{
    var infos = {
      "login": "Mike",
      "Role" : "Admin"
    };
    req.session.User = infos;
    res.render("session");
});

router.get('/', function(req, res) {
    console.log(req.session);
    res.render("home");
  });

  router.get('/destroy', function(req, res) {
    req.session.destroy();
    res.render("detruit");
  });

module.exports = router;
```

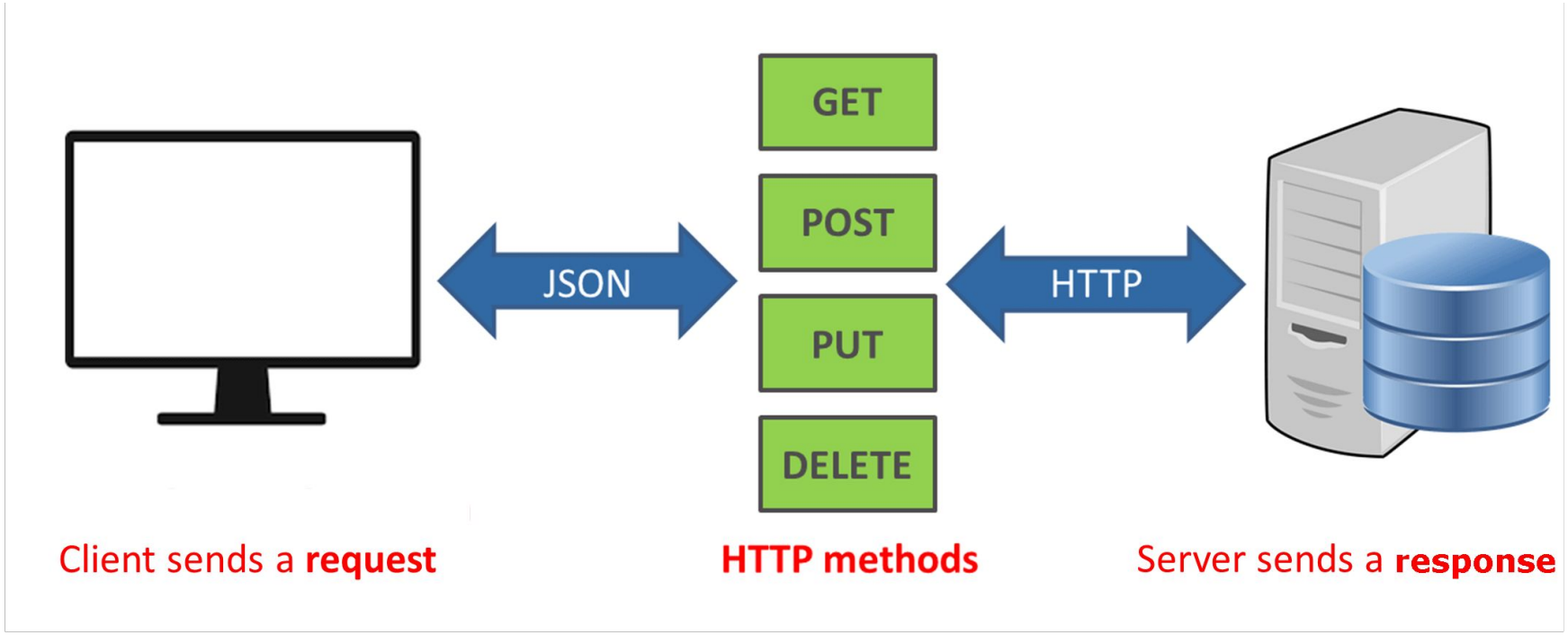
REST API

eXpress JS

REST API

RepreSentational State Transfert API

Une API compatible REST, ou « RESTful », est une interface de programmation d'application qui fait appel à des requêtes HTTP pour obtenir (GET), placer (PUT), publier (POST) et supprimer (DELETE) des données.



REST API

Contraintes architecturales de REST

1. Client-serveur

   Les applications REST ont un serveur qui gère les données et l'état des applications. Le serveur communique avec un client qui gère les interactions utilisateur. (indépendance complète)

2. Stateless

   Les serveurs ne conservent aucun état client. Les clients gèrent l'état de leur application. Leurs demandes aux serveurs contiennent toutes les informations nécessaires à leur traitement.

3. Cacheable

   Les serveurs doivent marquer leurs réponses comme pouvant être mis en cache ou non. Ainsi, les infrastructures et les clients peuvent les mettre en cache lorsque cela est possible pour améliorer les performances

4. Interface uniforme

   Les services REST fournissent des données en tant que ressources, avec un espace de noms cohérent.

5. Système en couches

   les composants du système ne peuvent pas «voir» au-delà de leur couche. Ainsi, vous pouvez facilement ajouter des équilibreurs de charge et des proxys pour améliorer la sécurité ou les performances.

REST API

Les Https verbs et leurs rôles

- POST : Permet de créer une nouvelle resource
- PUT : remplace une ressource existante (mets à jours la totalité des propriétés)
- PATCH : mise à jour partielle d'une resource existante
- DELETE: Suppression d'une ressource
- GET: Récupération d'une ressource

La structure des Urls

Les urls doivent être *Human-readable.* Elles doivent refléter le model de domain et être concise et précise.

Exemple :

http://www.boutique.com/details?id=22&categorie=12&promo=1 Incorrecte

http://www.boutique.com/Promotions/Chaussures/22 Correcte

REST API

Les modules utiles

1. Body-parser (https://github.com/expressjs/body-parser)

   Permet de transformer le body de la requête http en objet javascript

2. Cors (https://github.com/expressjs/cors)

   Permet de configurer notre Rest Api pour accepter les demandes d'origines différentes

3. Helmet(https://github.com/helmetjs/helmet)

   Permet de sécuriser notre API

4. Morgan(https://github.com/expressjs/morgan)

   Permet de mettre en place les logs

REST API

Let's go

Nous allons étape par étape créer un REST API simple permettant de manipuler des films

A. Installation de express et des modules complémentaires

npm install express body-parser helmet morgan cors

```
{
  "name": "rest_api_movies",
  "version": "1.0.0",
  "description": "Mini exemple de rest api avec express",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "cors": "^2.8.5",
    "express": "^4.17.1",
    "helmet": "^3.22.0",
    "morgan": "^1.10.0"
  }
}
```

REST API

Rest Api

Présentation d’une implémentation sous express

Exemples\RestApi

Jwt Authentication

eXpress JS

Jwt Authentication

La stratégie d'authentification traditionnelle utilise une session et un cookie.

JWT fournit une solution stateless pour l'authentification.

Avantages de JWT

- Fournit une solution d'authentification stateless.
- Très populaire et utilisé par de nombreux fournisseurs de services OAuth comme Google, Facebook.
- Il est très facile de vérifier le jeton jwt.
- Plus fiable que les cookies et les sessions.
- L'authentification peut être externalisée ou un service d'authentification peut être utilisé.
- Avoir de nombreuses applications autres que l'authentification, comme celles utilisées pour les revendications

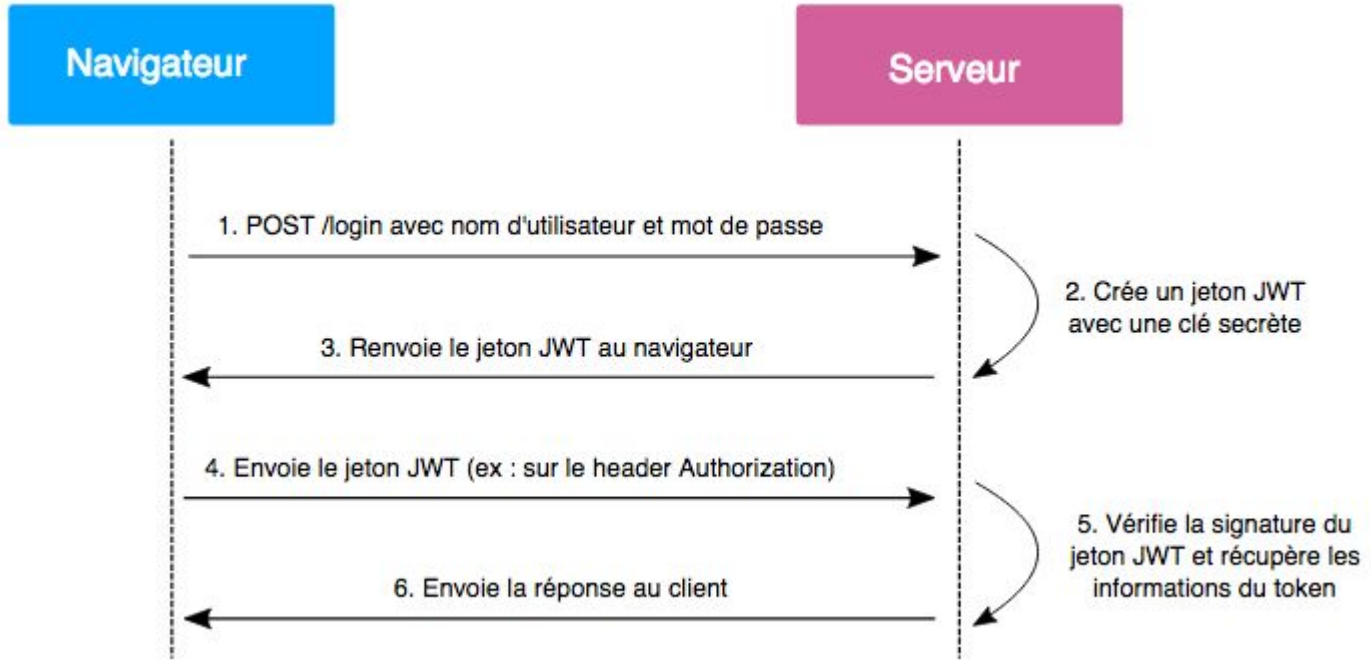
Jwt Authentication

Fonctionnement

Les JWT sont des jetons générés par un serveur lors de l'authentification d'un utilisateur sur une application Web, et qui sont ensuite transmis au client.

Ils seront renvoyés avec chaque requête HTTP au serveur, ce qui lui permettra d'identifier l'utilisateur.

Pour ce faire, les informations contenues dans le jeton sont signées à l'aide d'une clé privée détenue par le serveur. Quand il recevra à nouveau le jeton, le serveur n'aura qu'à comparer la signature envoyée par le client et celle qu'il aura générée avec sa propre clé privée et à comparer les résultats. Si les signatures sont identiques, le jeton est valide.



JWT Authentication

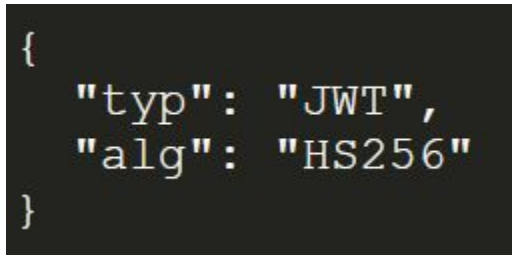
Structure d'un Jeton

Il y a trois parties séparées par un point, chaque partie est créée différemment. Ces trois parties sont :

1. Header (ou en-tête)

   Le header (ou en-tête) est un document au format JSON, qui est encodé en base 64 et qui contient deux parties

   - Le type de token (ici jwt)
   - L'algorithme de hashage utilisé



JWT Authentication

2. Payload (ou contenu)

   Document au format JSON, qui est encodé en base 64 et qui contient les informations (claims) à échanger entre le client et le serveur.

   En règle générale, on fait transiter des informations sur l'identité de l'utilisateur, mais il ne doit absolument pas contenir de données sensibles.

   3 types de claims:

- **Les claims réservés :** Il s'agit de nom réservé et ne pouvant pas être utilisés par le développeur. Ils contiennent des informations concernant le token lui-même.
  - iss : L'origine du token
  - sub : Le sujet du token
  - exp : Définie l'expiration du token
  - iat : Date de création du token
  - …
- **Les claims publics :** Il s'agit de noms personnalisés que l'on créés et qui sont propres à nos besoins.
- **Les claims privés :** Il s'agit de noms à usage privé pour répondre à des besoins spécifiques à vos applications. Ils ne doivent pas entrer en conflit avec les autres types de claims.

JWT Authentication

3. Signature

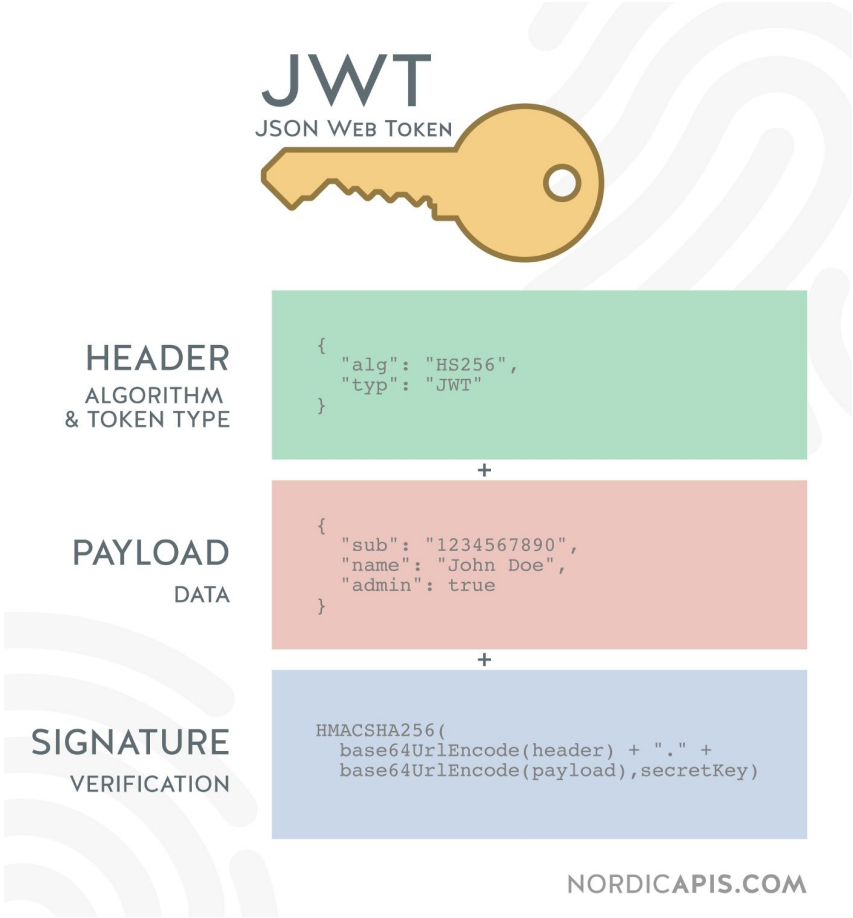
   La signature est composée d'un hash des éléments suivant :

   - Header
   - Payload
   - Secret : Le secret est une signature détenue par le serveur

```
var encodedString = base64UrlEncode(header) + "." + base64UrlEncode(payload);
HMACSHA256(encodedString, 'secret');
```

Pour vérifier un token jwt, vous pouvez utiliser le site : https://jwt.io/

JWT Authentication



JWT Authentication

Mise en place sous express

La mise en place sous express est plutôt simple.

Nous aurons besoin de la librairie *jsonwebtoken*

npm install jsonwebtoken

Ensuite, nous pourrons créer le Token en spécifiant les claims et le *secret* pour la signature.

Nous pouvons ensuite le renvoyer au client

```
// Generate an access token
const accessToken = jwt.sign({ username: user.username,  role: user.role }, accessTokenSecret);

res.json({
    accessToken
});
```

JWT Authentication

Pour vérifier le Token, nous utilisons toujours la même librairie

```
const jwt = require('jsonwebtoken');  > Importation de la librairie

const accessTokenSecret = 'a65f5874c6e954d52c55f';      > Définition d'un string pour le secret qui sera utilisé pour le Hash du Token
const authenticateJWT = (req, res, next) => {
    const authHeader = req.headers.authorization;      > Récupération du champs authorization du header Http

    if (authHeader) {
        const token = authHeader.split(' ')[1];  >on split sur l'espace car le header sera du format Bearer xxxxxxxxxxtoken

        jwt.verify(token, accessTokenSecret, (err, user) => {  > vérification du token grâce au secret
            if (err) {
                return res.sendStatus(403);          > si le token est invalide , on transmet un Http Status code 403
            }

            req.user = user; > On récupère l'utilisateur dans la requête http
            next();          > On continue le flux des routes
        });
    } else {
        res.sendStatus(401);
    }
};

module.exports=authenticateJWT;
```