

303 – ExpressJS & TypeScript

node.js

WIK-NJS303

Durée estimée : 4h30 (hors TP final)
Intervenant : Jeremy Trufier <jeremy@wikodit.fr>



WIK-NJS

Programme nodeJS

- 301 – Introduction
- 302 – Scripting et CLI**
- 303 – Express.js
- 304 – MVC Frameworks
- (305 – Tests unitaires)

1XX – 1er année (pas de notion d'algorithmie)
2XX – 2e année (notions d'algorithmie succinctes)
3XX – 3e année (rappels et pratique, niveau moyen d'algorithmie)
4XX – 4e année (concepts avancés, niveau avancé d'algorithmie)
5XX – 5e année (approfondissement experts)

Introduction

ExpressJS

- Serveur Web
- Minimaliste
- Performant
- Basé sur un système de middleware

ExpressJS

<https://codeshare.io/GL3jgK>

Avec express.js

Sans express.js

```
const http = require('http')
const PORT = process.env.PORT || 8080

http.createServer((req, res) => {
  if (req.url === '/hello-world') {
    return res.end('Bonjour à tous')
  }

  if (m = req.url.match(/^VusersV(.+?)/gi)) {
    return res.end('User to load is: ' + m[0])
  }

  res.writeHead(404)
  res.end('Not found!')
}).listen(PORT, () => {
  console.log('Serveur sur port ', PORT)
})
```

```
const express = require('express')
const app = express()
const PORT = process.env.PORT || 8080

app.get('/hello-world', (req, res) => {
  res.send('Bonjour à tous')
})

app.get('/users/:userId', (req, res) => {
  res.send('User to load is: ' + req.params.userId)
})

app.use((req, res) => {
  res.send(404, 'Not Found')
})

app.listen(PORT, () => {
  console.log('Serveur sur port : ', PORT)
})
```

TypeScript

- Apporte le typage statique au JavaScript ES6
- Trans-compileur (transpiler)
- Améliorer et sécuriser le code JavaScript
- Développé par Microsoft et par le créateur du C#
- Permet du CodeIntel puissant pour améliorer la productivité des développeurs

TypeScript

Sans TypeScript

```
function add(x, y) {  
  return x + y  
}  
  
const a = 4  
const b = 5  
const c = 'toto'  
  
console.log(add(a, b))  
console.log(add(a, c))
```

9

4toto

Avec TypeScript

```
function add(x: number, y: number): number {  
  return x + y  
}  
  
const a = 4  
const b = 5  
const c = 'toto'  
  
console.log(add(a, b))  
console.log(add(a, c))
```

TSError: x Unable to compile TypeScript
compare.ts (10,20): Argument of type "'toto'" is not
assignable to parameter of type 'number'.

TypeScript nous permet de détecter les problème dès la compilation, et même
dès l'écriture du code selon les IDE

Express.JS

Le routing

- Chaînage
- Routing par verbe http
- Route path
 - Simple
 - Regexp
 - Named parameters

```
app.all('*', (req, res, next) => {
  console.log('-> ALL *')
  next()
})

app.get('/', (req, res, next) => {
  console.log('-> GET /')
  res.send('GET /')
})

app.post('/', (req, res, next) => {
  console.log('-> POST /')
  res.send('POST /')
})

app.get('/users', (req, res, next) => {
  console.log('-> GET /users (liste d\'utilisateurs)')
  res.send('GET /users')
})

app.get('/users/:userId', (req, res, next) => {
  console.log('-> GET /users/:userId (userId : ' + req.params.userId + ')')
  res.send('GET /users/:userId')
})

app.get(/^\/images\/.*\.(png, jpg, gif)$/, (req, res, next) => {
  console.log('-> GET /images/* (une image png, jpg, git)')
  res.send('GET /images/*')
})
```

Le routing (2)

```
function a(req, res, next){
  console.log('a')
  next()
}

function b(req, res, next){
  console.log('b')
  setTimeout(() => { // On passe au next() seulement après 5 secondes
    next()
  }, 5000)
}

app.get('/test', [a, b], (req, res, next) => {
  console.log('c')
  next()
}, (req, res, next) => {
  console.log('d')
  res.send('On est passé par a, b, c, d, avec 5 secondes d'attentes avant c')
})
```

L'objet Request

- Contient les informations de la requête

Certain middlewares permettent de rajouter des informations sur l'objet request.

De plus, il est possible de rajouter des attributs sur l'objet `req` pour passer des infos entre les différentes méthodes du routing.

```
app.get('/:param1', (req, res) => {  
  console.log(  
    'Prot: ', req.protocol,  
    ', Url: ', req.url,  
    ', Method: ', req.method,  
    ', Param1: ', req.params.param1,  
    ', Query: ', req.query,  
    ', Header Content-Type: ', req.get('Content-Type')  
  )  
  
  res.status(200).end()  
})
```

L'objet Response

- Permet d'envoyer des données au client

```
// Réponse simple  
res.write('Bonjour !')  
res.end()
```

```
// Set le statut, et simple retour  
res.status(404)  
res.end('Not Found')  
// OU  
res.status(404).end('Not Found')
```

```
// Une simple redirection  
res.redirect(301, 'http://google.fr')
```

```
// Un retour différent selon le format demandé  
res.format({  
  html: () => { res.send('<strong>Bonjour !</strong>') },  
  json: () => { res.send({ message: 'Bonjour !' }) }  
})
```

```
// On set/get un header de la réponse. Attention une fois des données envoyé au client, les  
// headers ne peuvent plus être modifiés, cela peut résulter en une erreur.  
res.set('ETag', '17b3cc1a-8eeb-11e6-ae22-56b6b6499611')  
console.log(res.get('ETag'))
```

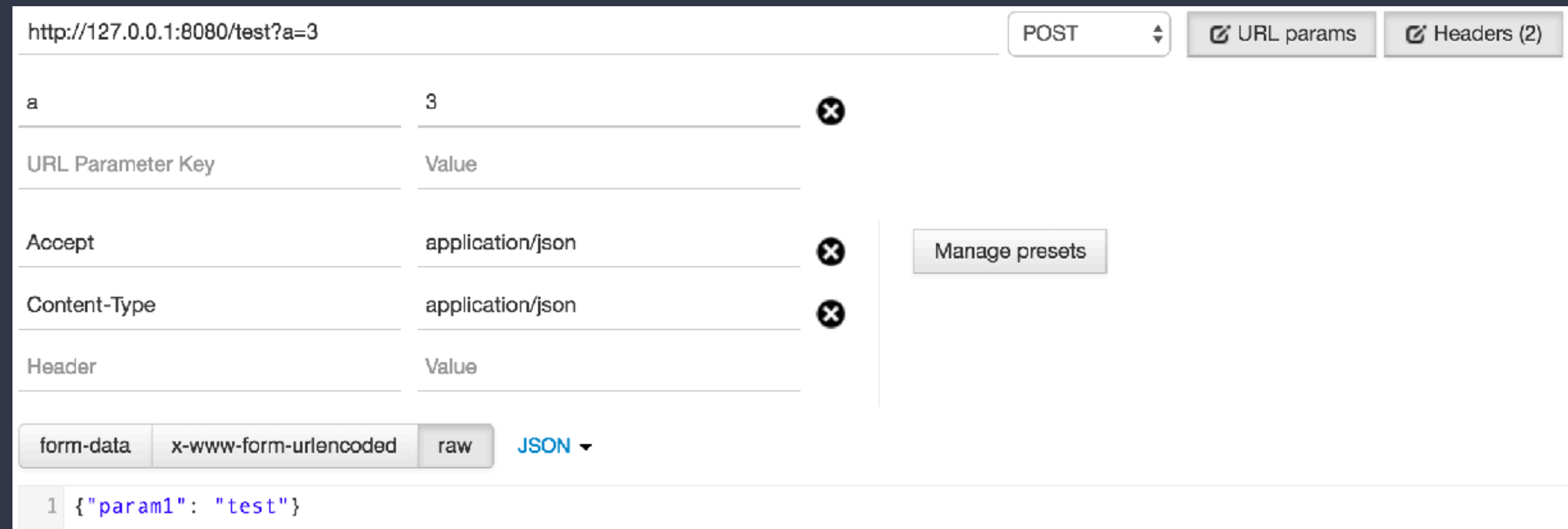
Comment tester ?

Dans le terminal avec CURL

```
curl -i -H 'Accept: text/html' "http://localhost:8080/test?a=3"
```

```
curl -X POST -H 'Content-Type: application/json' -H 'Accept: text/html' -d '{"param1": "test"}' "http://localhost:8080/test?a=3"
```

OU Avec Postman
(dans chrome)



ExpressJS: Les middlewares

C'est quoi ?

- `.get`, `.post`, ... sont des middlewares
- Les plugins expressJS sont des middlewares
- Interception de requêtes avant ou après la logique
- La requête client passe par un ou plusieurs middleware
- On passe au middleware suivant avec ``next()``
- Un middleware peut terminer la réponse avec un ``res.end`` ou ``res.send``

Exemple

```
var logger = function(req, res, next) {  
  next()  
  console.log(`REQUEST: ${req.method} ${req.url}`)  
}  
  
// Middleware qui log les requêtes  
app.use(logger)  
  
// Notre routing  
app.get('/', (req, res, next) => {  
  res.send('Page d'accueil')  
})  
  
// autre routes  
// ...  
  
// Dernier middleware  
app.use((req, res) => {  
  res.status(404)  
  res.end('Not Found')  
})
```


Static middleware

```
app.use(express.static('assets'))
```

njs-303/assets/img/image.jpg sera accessible depuis <http://localhost:8080/img/image.jpg>

```
app.use('/styles', express.static('assets/css'))
```

njs-303/assets/css/app.css sera accessible depuis <http://localhost:8080/styles/app.css>

Body middleware

Attention, c'est un module NPM à installer !!

Pour parser les Content-Type: application/json =>
Pour parser les formulaire HTML =>

Nouvelle propriété req.body =>

```
const bodyParser = require('body-parser')

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

app.post('/test', (req, res, next) => {
  console.log(req.body)
  res.send('On a parsé le body !')
})
```

Error middleware

Middleware à 4 paramètres permet de catcher les next avec un argument

Exemple gestion d'erreur + REST: <https://git.io/vxZKc>

En faisant un next(xxxx)

Ce middleware n'est jamais appelé

Mais le middleware d'erreur à 4 arguments est appelé avec xxxx en premier paramètre

```
app.get('/test', (req, res, next) => {  
  if (false) {  
    return next(new Error("An error occurred"))  
  }  
  
  res.end('never here')  
})  
  
app.use((req, res, next) => {  
  console.log('This console.log is never written')  
})  
  
app.use((err, req, res, next) => {  
  console.log('Here is the error: ', err)  
  
  res.send(500, 'Server Error')  
})
```

ExpressJS: Les templates

Les vues

- Nécessité de séparer les vues de la logique
- Différents moteurs de templates
 - EJS
 - Pug (anciennement Jade)
 - Handlebars
 - ... plusieurs dizaines (centaines?) d'autres !

Pug

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

index.js

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

app.get('/', () => {
  res.render('main', {
    title: 'Bonjour !',
    name: 'Toto',
    content: 'Ma première page'
  })
})
```

views/main.pug

```
html
  head
    title= title
  body
    h1 Bonjour #{name}
    p#main-paragraphe.center= content
```

EJS

```
$ npm install ejs --save
```

index.js

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

app.get('/', () => {
  res.render('main', {
    title: 'Bonjour !',
    name: 'Toto',
    content: 'Ma première page'
  })
})
```

views/main.ejs

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
</head>
<body class="container">
  <h1>Bonjour <%= name %></h1>
  <p id="main-paragraphe" class="center">
    <%= content %>
  </p>
</body>
</html>
```

Handlebars

```
$ npm install hbs --save
```

index.js

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

app.get('/', () => {
  res.render('main', {
    title: 'Bonjour !',
    name: 'Toto',
    content: 'Ma première page'
  })
})
```

views/main.ejs

```
<!DOCTYPE html>
<html>
<head>
  <title>{{title}}</title>
</head>
<body class="container">
  <h1>Bonjour {{name}}</h1>
  <p id="main-paragraphe" class="center">
    {{content}}
  </p>
</body>
</html>
```


TP Todo List

TD1 : Todo List

1. Ouvrir une base de donnée SQLite avec Sequelize et créer une table :
 - todos (message, completion, createdAt, updatedAt)
2. Créer les routes
 - ALL / *=> Redirection vers /todos*
 - POST /todos *=> Ajouter une todo*
 - GET /todos/:todold *=> Récupérer une todo*
 - GET /todos?limit=20&offset=0 *=> Lister tous les todos avec Pagination*
 - DELETE /todos/:todold *=> Supprimer une todo*
 - PATCH /todos/:todold *=> Éditer une todo (la passer en done par exemple)*
 - autre *=> 501 Not Implemented ou 404 Not Found*
3. Implémenter chaque route dans l'ordre ci-dessus, vérifier le bon fonctionnement à chaque fois. Les routes doivent répondre au format JSON.
4. Bonus : Ajouter des filtres sur "/todos" pour rechercher les todos en fonction de leur état : "?completion=done"

TD2 : Rajout de vues

1. Ajouter la gestion du multi-format de réponse
2. Ajouter "GET /todos/add" et "GET /todos/:todold/edit"
3. Chaque appel peut désormais retourner soit une page HTML dans le navigateur, avec des formulaires, tableaux, etc... Soit une structure JSON si le header 'Accept' est à 'application/json'

Des exemples d'organisation pour la resource Todo

- **GET** /todos => **views/todos/index.pug**
- **GET** /todos/add => **views/todos/edit.pug**
- **GET** /todos/:todold => **views/todos/show.pug**
- **GET** /todos/:todold/edit => **views/todos/edit.pug**
- **POST** /todos => **HTML** : redirection /todos ; **JSON** : status succès
- **PUT/PATCH** /todos/:todold => **HTML** : redirection /todos ; **JSON** : status succès
- **DELETE** /todos/:todold => **HTML** : redirection /todos ; **JSON** : status succès

TD3 : Rajout de la gestion users

1. Reprendre le TD3
2. Ajouter un model `users`
3. Créer les routes REST correspondantes et les formulaires correspondants
4. Rajouter un champ mot de passe aux utilisateurs
5. Utiliser le module `bcrypt` pour hasher le mot de passe
6. Ajouter le middleware `expressjs/session`
7. Ajoutez un formulaire de login, qui enregistre le userId en session (le user est redirigé vers ce formulaire s'il n'est pas connecté (middleware))
8. Chaque utilisateur ne peut voir que ces propres Todos, chaque utilisateur doit pouvoir cocher une Todo, qui passe alors en fin de liste
9. Grand Bonus : Team
 - Un utilisateur peut appartenir à une seule Team (ou aucune)
 - Un utilisateur appartenant à une team a une option "Voir mes todos / Voir les todos de mon équipe"
 - Les todos peuvent être assignées à un utilisateur de la même team
 - Les todos peuvent être terminées par n'importe quel utilisateur

[X] Sortir les poubelles (par [MaChérie](#), pour [moi](#), complété le 11/10 à 7h30)

TypeScript

Conseils

- Utiliser Visual Studio Code (pour bénéficier de toute la puissance de TypeScript)
- Puis les extensions suivantes
 - **Document This** => Permet en un raccourci clavier d'ajouter le JSDoc de fonctions, classes, ...
 - **TSLint** => Permet de détecter de possibles problèmes lors de l'écriture de code, et permet de définir des règles d'écriture du code
- Tout le reste pour TypeScript est fourni par Visual Studio Code

Il est aussi nécessaire d'installer TypeScript sur votre système : **npm install -g typescript**

TSNode permet de lancer des applications TS comme avec node : **npm install -g ts-node**

TS-TD1

Mettez en place cette arborescence et installez les pre-requis.

Les modules commençant par @types permettent d'ajouter les informations de type aux modules qui n'en possèdent pas.

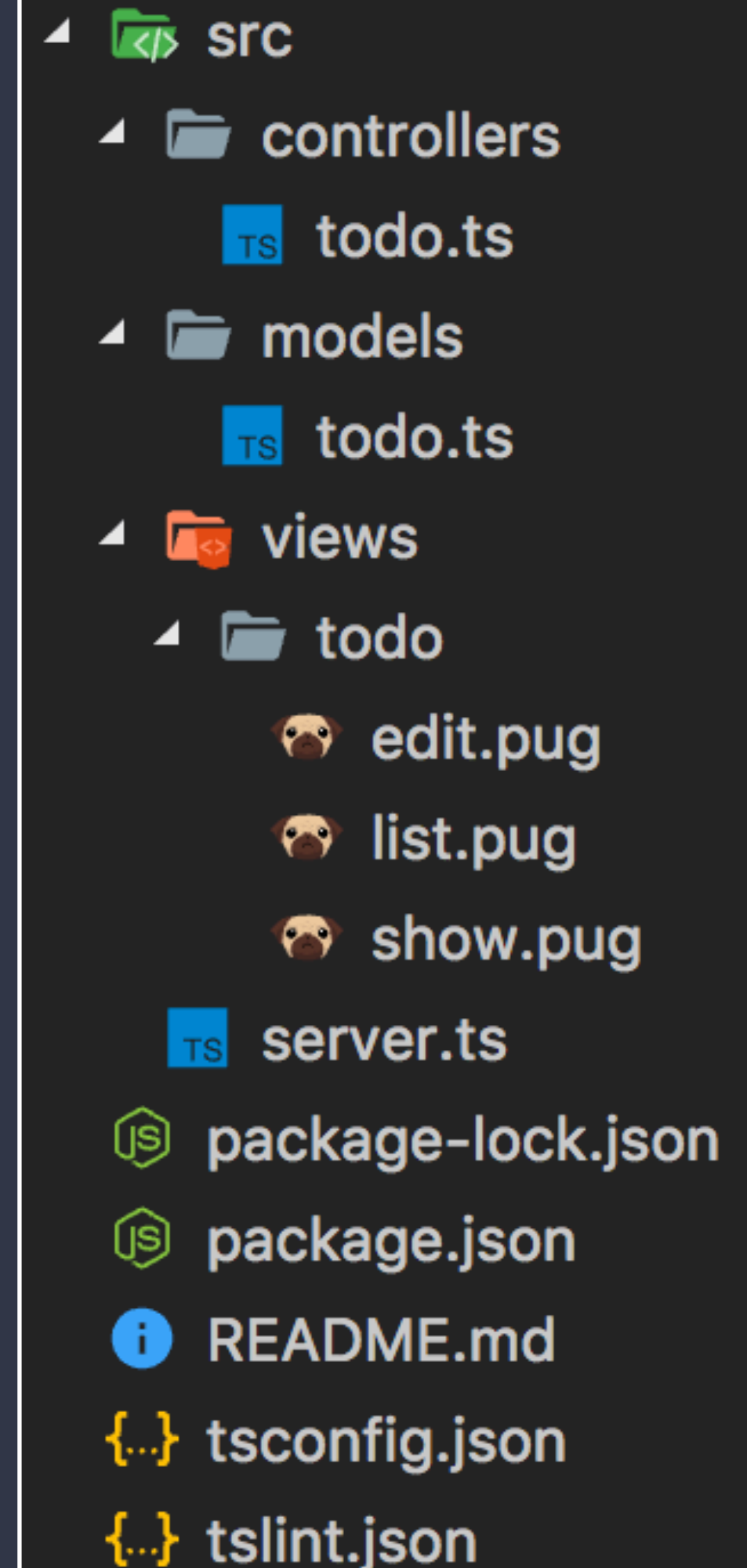
=> Ils contiennent les définitions des modules

Nodemon nous permet de relancer automatiquement le serveur quand on modifie des fichiers

Prérequis

```
npm install --save express lodash sequelize sequelize-typescript reflect-metadata sqlite3
npm install --save-dev typescript nodemon @types/lodash @types/express @types/sequelize tslint-config-airbnb ts-node
```

Arbo cible



```
src
├── controllers
│   └── todo.ts
├── models
│   └── todo.ts
├── views
│   ├── todo
│   │   ├── edit.pug
│   │   ├── list.pug
│   │   └── show.pug
│   └── server.ts
├── package-lock.json
├── package.json
├── README.md
├── tsconfig.json
└── tslint.json
```


TS-TD2

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es6",
    "noImplicitAny": true,
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "allowSyntheticDefaultImports": true,
    "baseUrl": "./src",
    "paths": {
      "*": [
        "node_modules/*"
      ]
    },
    "include": [
      "src/**/*"
    ]
  }
}
```

package.json

```
...
"scripts": {
  "dev": "nodemon --watch src/**/*.ts --exec ts-node src/boot.ts"
},
...
```

Les scripts peuvent être lancés avec **npm run <script name>**

Ici Nodemon permettra de relancer notre serveur dès qu'un fichier change

tslint.json

```
{
  "extends": "tslint-config-airbnb",
  "rules": {
    "semicolon": [ true, "never" ],
    "import-name": false,
    "no-increment-decrement": false
  }
}
```

TSLint vérifiera notre code et façon de coder

TS-TD3

src/boot.ts

```
import * as express from 'express'

const app = express()

app.set('port', process.env.PORT || 3000)

app.use((req: express.Request, res: express.Response, next: express.NextFunction) => {
  const a = { op1: 3, op2: 4 }
  const r = a.op1 + a.op2
  console.log(r)
  res.status(404).send('Not Found')
})

app.listen(app.get('port'), () => {
  console.log(`Started on port ${app.get('port')}`)
})
```

Aucune différence par rapport à du JavaScript pour le moment,
excepté le support de la nouvelle syntaxe ES6 pour les imports/exports

Pour ceux qui ont IntelliSense, en tapant le code, la documentation et les possibilités s'affichent.

Débugger

- Très important de savoir utiliser un debugger
- Les ``console.log`` sont vite limités
- Chrome possède un debugger pour le JavaScript
- La plupart des IDE aussi
- Possibilité de mettre des points d'arrêt et de voir l'état à un instant T du programme : les fonctions qui ont mené jusqu'à cet endroit, les variables en cours et leur valeur
- Possibilité de dérouler le code Pas-à-Pas jusqu'à l'endroit qui pose problème

Débugger

server.ts — njs-402-ts [Non prise en charge]

DÉBOGUEUR ▶ Launch Proc ⚙️ ▶

VARIABLES

Local

- this: undefined
- req: IncomingMessage {
 _readableSta...
 _consuming: false
 _dumped: false
 _events: Object {}
 _eventsCount: 0
 _maxListeners: undefined
 _parsedUrl: Url {protocol: null...
 _readableState: ReadableState {o...
 baseUrl: ""
 client: Socket {connecting: fals...
 complete: false
 connection: Socket {connecting: ...
 destroyed: false

ESPION

req.url: "/"

PILE DES APPELS EN PAUSE SUR UN POINT..

app.use	server.ts	8:3
handle	layer.js	95:5
trim_prefix	index.js	317:13
(anonymous function)	index.js	
process_params	index.js	335:12
next	index.js	275:10

```
1 import * as express from 'express'
2
3 const app = express()
4
5 app.set('port', process.env.PORT || 3000)
6
7 app.use((req, res) => {
8   res.status(404).send('Not Found')
9 })
10
11 app.listen(app.get('port'), () => {
12   console.log(`Started on port ${app.get('port')}`)
13 })
14
```

1 2 3 4 5 6 7

Point d'arrêt + Appel actuel en pause

Variables dans le scope Local, ou dans les Closures supérieures ou en Global

Permet d'espionner une expression

Permet de savoir où on se trouve dans la pile d'exécution. Et de pouvoir remonter voir les différents appels de fonctions qui ont mené à l'endroit où on se trouve

1: curl ↕ + 🗑️ ⬆️ ✕

(Jeremyt@MacBook-Pro-2:5004) ~/Development/Ingesup/courses/

(%) curl 127.0.0.1:3000

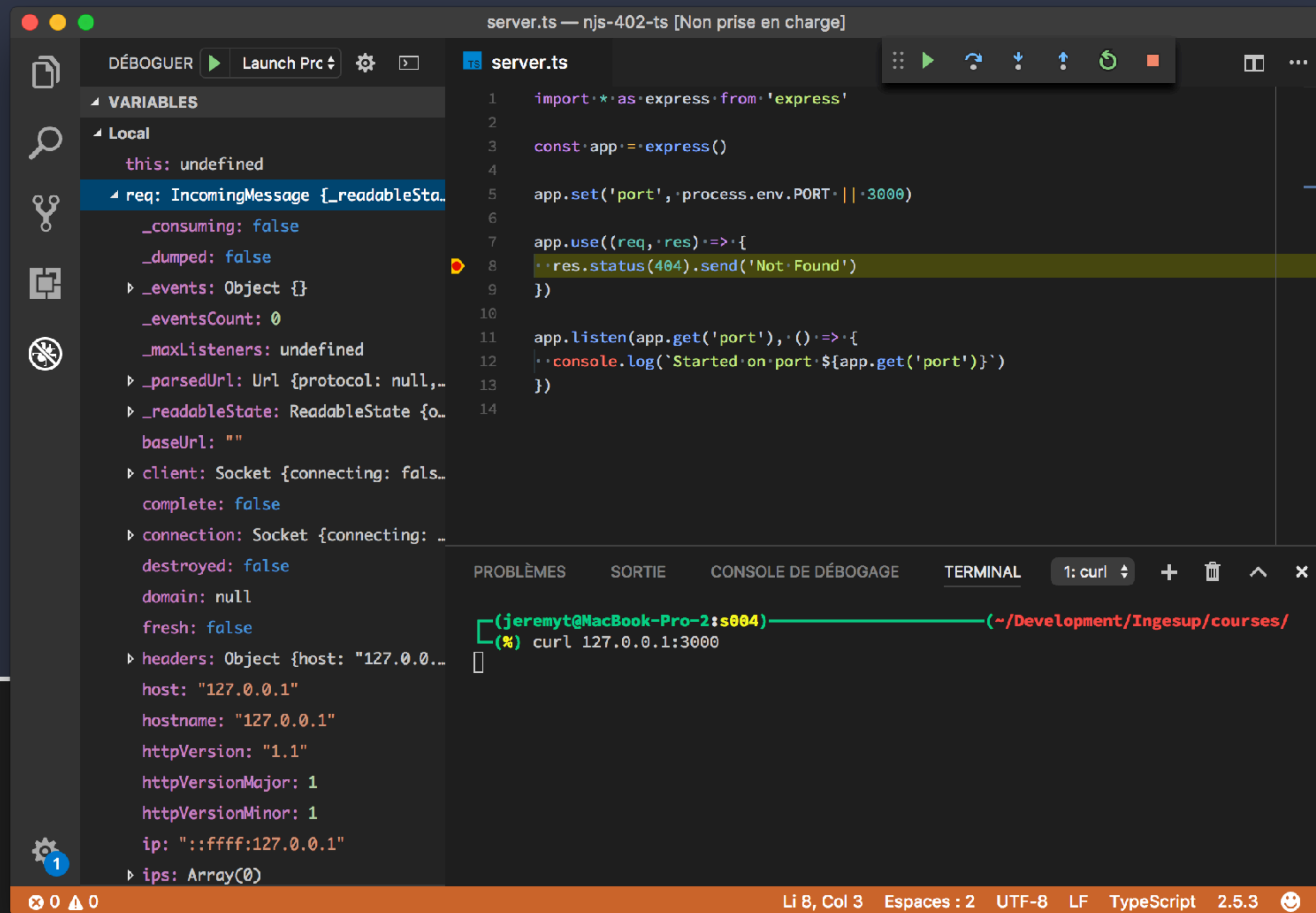
Li 8, Col 3 Espaces : 2 UTF-8 LF TypeScript 2.5.3 😊

1. Continuer
2. Sauter le prochain appel
3. Rentrer dans le prochain ap
4. Sortir de l'appel en cours
5. Relancer le programme
6. Stopper le programme

TS-TD4

`.vscode/launch.json`

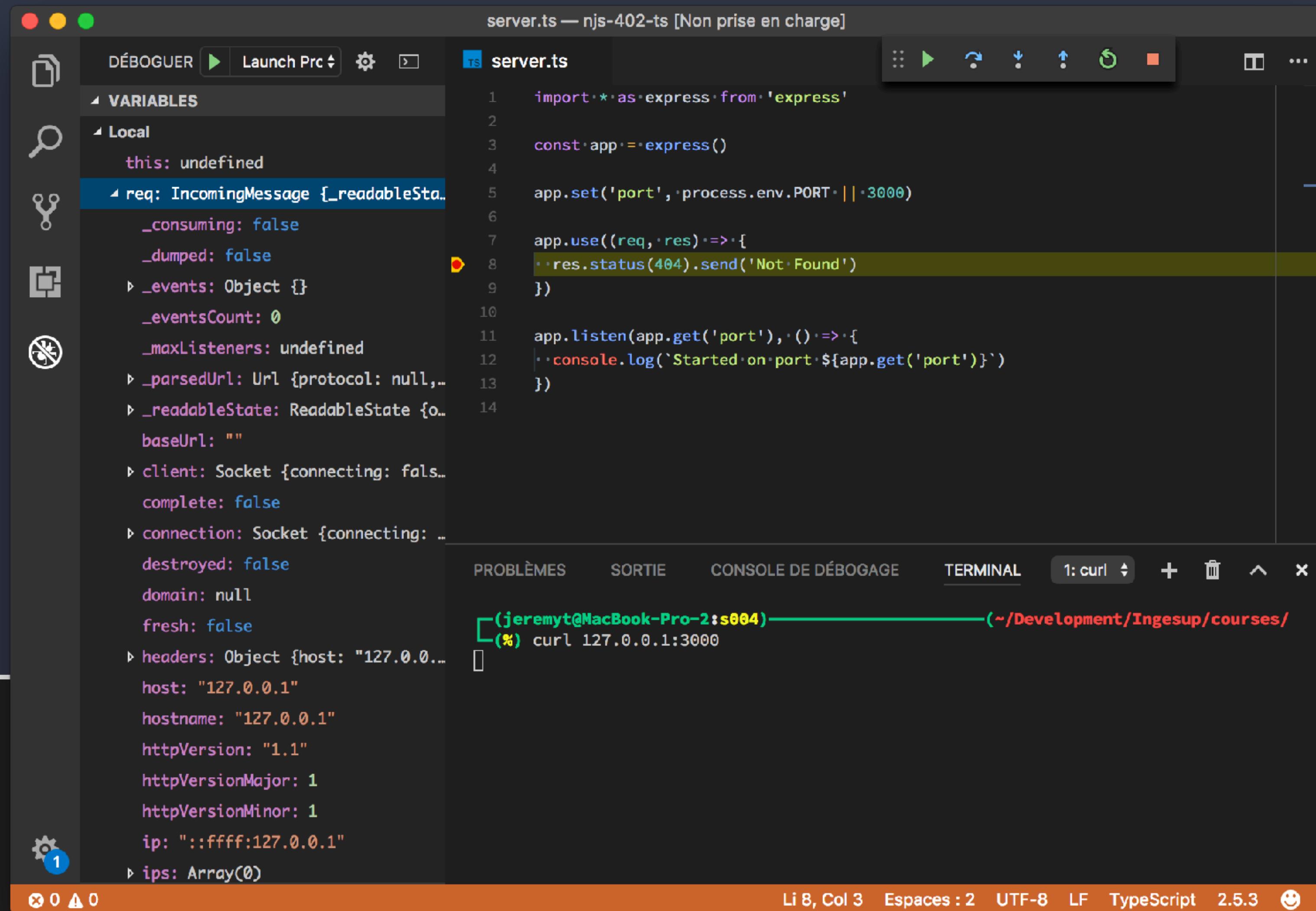
```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "args": ["${workspaceFolder}/src/server.ts"],
      "runtimeArgs": ["--no-lazy", "-r", "ts-node/register"],
      "sourceMaps": true,
      "cwd": "${workspaceRoot}",
      "protocol": "inspector"
    }
  ]
}
```



TS-TD4

`.vscode/launch.json`

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "args": ["${workspaceFolder}/src/server.ts"],
      "runtimeArgs": ["--no-lazy", "-r", "ts-node/register"],
      "sourceMaps": true,
      "cwd": "${workspaceRoot}",
      "protocol": "inspector"
    }
  ]
}
```



Pour la suite

Installez les modules suivant

```
npm install --save body-parser cookie-parser morgan method-override
```

Et leurs définitions typescript respectives

```
npm install --save @types/body-parser @types/cookie-parser @types/morgan @types/method-override
```

cookie-parser => Middleware express qui ajoute le support de cookies

body-parser => Middleware express permettant de récupérer le body d'une requête depuis différents formats

morgan => Middleware express qui permet de logger facilement les requêtes dans la console
errorhandler

<https://github.com/Tronix117/wik-njs-402>

Félicitations !!

Cours WIK-NJS-302 burned :)