



**formation.tech**

# Formation Node.js

Romain Bohdanowicz

Twitter : @bioub - <https://github.com/bioub>  
<http://formation.tech/>



**formation.tech**

# Node.js

# Node.js - Introduction



- Crée 2009 par Ryan Dahl
  - A l'origine, Ryan Dahl voulait simplifier la création d'une barre d'upload.
- Projet de la OpenJS Foundation
- Un programme en ligne de commande combinant :
  - le moteur JavaScript V8 de Chrome
  - une boucle d'événement
  - une gestion bas niveau des entrées/sorties
  - des libs externes (openssl, zlib...)
- Documentation  
<https://nodejs.org>

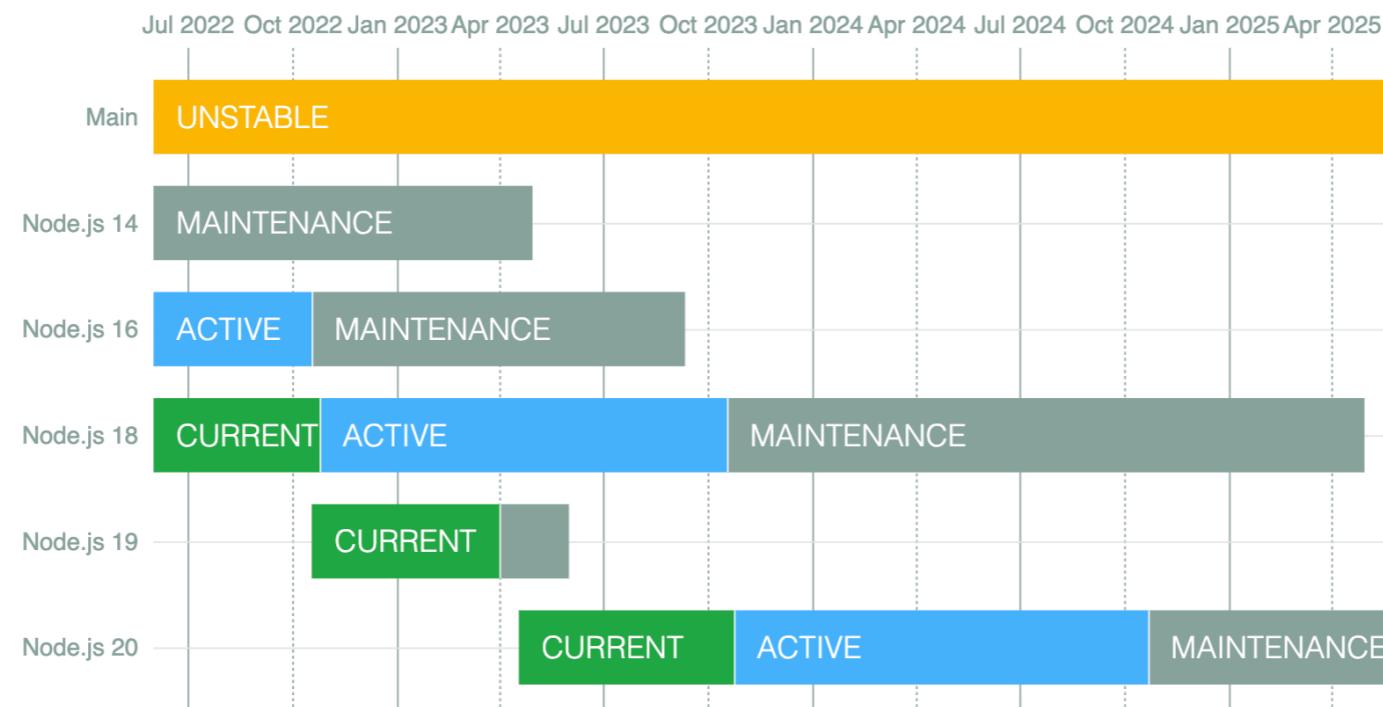
# Node.js - LTS



## Release schedule

Release	Status	Codename	Initial Release	Active LTS Start	Maintenance Start	End-of-life
14.x	Maintenance	Fermium	2020-04-21	2020-10-27	2021-10-19	2023-04-30
16.x	Active LTS	Gallium	2021-04-20	2021-10-26	2022-10-18	2023-09-11
18.x	Current		2022-04-19	2022-10-25	2023-10-18	2025-04-30
19.x	Pending		2022-10-18	-	2023-04-01	2023-06-01
20.x	Pending		2023-04-18	2023-10-24	2024-10-22	2026-04-30

Dates are subject to change.



# Node.js - Installation



- Recommandé : fnm
- Permet d'avoir plusieurs version de Node.js en même temps sur son poste (un projet donné doit fixer une version de Node.js)
- Alternative moderne à nvm, développé en Rust donc plus rapide
- Simplifie les installation et mise à jour de Node.js
- Permet de switcher d'une version de Node.js à une autre en une commande
- Switch automatiquement sur la commande `cd` si configuré, via des fichiers `.node-version` ou `.nvmrc`



# Node.js - Helloworld

- › Lancement du programme  
node FILE\_PATH[.js]

```
/* Un simple helloworld */

/** @function helloworld */
function helloworld() {
  'use strict'; // bonne pratique
  console.log('Helloworld');
}

setInterval(helloworld, 1000);
```

The screenshot shows a terminal window titled "LearningJS — node — 78x16". The command entered is "node Node.js/Slides/helloworld.js". The output consists of nine lines of the word "Helloworld", indicating a 1-second interval between log statements.

```
MacBook-Pro-de-Romain:LearningJS roman$ node Node.js/Slides/helloworld.js
Helloworld
Helloworld
Helloworld
Helloworld
Helloworld
Helloworld
Helloworld
Helloworld
Helloworld
```



# Node.js - Débogage

- En ligne de commande  
`node inspect FILE_PATH[.js]`
- Avec les DevTools de Chrome  
`node --inspect-brk FILE_PATH[.js]`  
Puis sous Chrome aller à l'URL <chrome://inspect>
- Avec les IDEs Webstorm ou Visual Studio Code



# Node.js - 2 utilisations

- Pour les programmes en ligne de commande, ex :
  - build: webpack / grunt / gulp...
  - linters: eslint / tslint / stylelint...
  - tests: mocha / jest / karma
  - serveurs web de dev: http-server / live-server...
  - ...
- Pour les programmes serveur
  - Applications traditionnelles, HTML rendu côté serveur
  - API REST
  - Serveur WebSocket
  - ...

# Node.js - Pourquoi JavaScript côté serveur ?



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

http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, world !');
}).listen(8080);
```

- Avantage du JavaScript côté serveur
  - Performance : le côté non-bloquant de JavaScript le rend particulièrement performant, plus besoin de gérer les problèmes inter-thread ou inter-processus
  - Réutilisation : une bibliothèque ou un composant peut être utilisé sur le client comme sur le serveur
  - Apprentissage : vous connaissez déjà JavaScript
  - Ecosystème : le nombre de bibliothèques open-source (langage le plus populaire sur GitHub)

# Node.js - Event Loop



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

http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, world !');
}).listen(8080);
```

- Node.js implémente côté C++ une boucle d'événement et une gestion non-bloquante des entrées/sorties
- Ici lorsque qu'un requête HTTP arrive sur le port 8080 la fonction de rappel passée en argument de createServer est appelée
- Il faut quelques millisecondes pour exécuter ce callback, le reste du temps JavaScript est libre de faire autre chose (exécuter des requêtes concurrentes, se connecter à une base de données, écrire dans un fichier...)

# Node.js - Documentation



- ▶ Livre  
<https://nodejs.org/en/learn/>
- ▶ API  
<https://nodejs.org/api/>
- ▶ 4 niveaux de "stabilité"

Stability: 0 - Deprecated. The feature may emit warnings. Backward compatibility is not guaranteed.

Stability: 1 - Experimental. The feature is not subject to [semantic versioning](#) rules. Non-backward compatible changes or removal may occur in any future release. Use of the feature is not recommended in production environments.

Experimental features are subdivided into stages:

- 1.0 - Early development. Experimental features at this stage are unfinished and subject to substantial change.
- 1.1 - Active development. Experimental features at this stage are nearing minimum viability.
- 1.2 - Release candidate. Experimental features at this stage are hopefully ready to become stable. No further breaking changes are anticipated but may still occur in response to user feedback. We encourage user testing and feedback so that we can know that this feature is ready to be marked as stable.

Stability: 2 - Stable. Compatibility with the npm ecosystem is a high priority.

Stability: 3 - Legacy. Although this feature is unlikely to be removed and is still covered by semantic versioning guarantees, it is no longer actively maintained, and other alternatives are available.

# Node.js - Chapitres de la Doc API



- Assertion Testing
- Async Hooks
- Buffer
- C++ Addons
- C/C++ Addons - N-API
- Child Processes
- Cluster
- Command Line Options
- Console
- Crypto
- Debugger
- Deprecated APIs
- DNS
- Domain
- ECMAScript Modules
- Errors
- Events
- File System
- Globals
- HTTP
- HTTP/2
- HTTPS
- Inspector
- Internationalization
- Modules
- Net
- OS
- Path
- Performance Hooks
- Process
- Punycode
- Query Strings
- Readline
- REPL
- Stream
- String Decoder
- Timers
- TLS/SSL
- Trace Events
- TTY
- UDP/Datagram
- URL
- Utilities
- V8
- VM
- Worker Threads
- ZLIB

# Node.js - API



- Ne sont pas (ou pas complètement) des APIs
  - C++ Addons et C/C++ Addons - N-API  
Permet de compiler du code C ou C++ et de le "binder" (le rendre accessible) au JavaScript de Node.js
  - Command Line Options  
Arguments du programme Node
  - Debugger  
Un debugger en ligne de commande
  - Deprecated APIs  
La liste des APIs dépréciés
  - ECMAScript Modules  
L'implémentation des Modules ECMAScript (ESM ou ES6 Modules)
  - Internationalization  
Comment compiler Node.js pour améliorer le support i18n
  - REPL  
Permet de lancer Node en mode interactif et de saisir du code dans la console

# Node.js - API



- Les APIs dépréciés (seront supprimé à l'avenir)
  - Domain
  - Punycode
  - <https://nodejs.org/dist/latest/docs/api/deprecations.html>
- Les APIs expérimentaux (leur implémentation peut évoluer)
  - Async Hooks
  - Inspector (jusqu'à Node 12)
  - Trace Events



# Node.js - Console

- Le module console (global) permet de logger dans la console et de réaliser des benchmarks

```
console.time('100-elements');
for (var i = 0; i < 100; i++) {
  console.log(i);
}
console.timeEnd('100-elements');
// 100-elements: 8ms
```

# Node.js - Timers



- Le module Timers (global) contient les fonctions pour différer l'exécution de callbacks.

```
setTimeout(function() {
  console.log('1 fois dans 3 secondes');
}, 3000);

var intervalId = setInterval(function() {
  console.log('toutes les 2 secondes');
}, 2000);

setTimeout(function() {
  console.log('Bye bye');
  clearInterval(intervalId);
}, 15000);
```

# Node.js - File System



- Le module File System (`require('fs')`) permet les accès aux disques, fichiers, dossiers, droits, etc...

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

try {
  var stats = fs.statSync('./dist');
}
catch(e) {
  fs.mkdirSync('./dist');
}

var fichiers = fs.readdirSync('./src');

for (var i=0; i<fichiers.length; i++) {
  var fichier = fichiers[i];
  var src = './src/' + fichier;
  var dest = './dist/' + fichier;

  var contenu = fs.readFileSync(src);
  fs.writeFileSync(dest, contenu);
}
```



# Node.js - Net

- Le module net (require(net)) permet les accès réseau

```
var net = require('net');
var server = net.createServer(function(c) { //'connection' listener
  console.log('client connected');
  c.on('end', function() {
    console.log('client disconnected');
  });
  c.write('hello\r\n');
  c.pipe(c);
});
server.listen(8124, function() { //'listening' listener
  console.log('server bound');
});
```

# Node.js - Serveur Net



- › Un chat serveur avec Net

```
var net = require('net');

var clients = {}, cpt = 0;

var server = net.createServer(function(c) { // 'connection' listener
    var me = 'c' + (++cpt);
    console.log('client connected');

    clients[me] = c;
    c.on('end', function() {
        //clients[me].end();
        delete clients[me];
    });
    c.write('Bienvenue sur le Chat !!! (telnet : taper exit pour quitter)\r\n');

    c.on('data', function(chunk) {
        for (var cid in clients) {
            if (clients.hasOwnProperty(cid)) {
                if (chunk.toString().indexOf('exit') === 0) {
                    clients[me].end();
                    delete clients[me];
                    break;
                }

                if (cid !== me) {
                    clients[cid].write(chunk.toString());
                }
            }
        }
    })
};

server.listen(8124, function() { // 'listening' listener
    console.log('server bound');
});
```



# Node.js - Client Net

- › Un chat client avec Net

```
var net = require('net');
var readline = require('readline');

var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question("Quel est ton pseudo ? ", function(pseudo) {
  console.log("Bienvenue sur le chat", pseudo);

  var client = net.connect({port: 8124}, function() { //connect' listener
    console.log('(connecté au serveur)');
    process.stdin.on('readable', function() {
      var chunk = process.stdin.read();
      if (chunk !== null) {
        var msg = chunk.toString();
        msg = msg.substr(0, msg.length - 1); // on retire le \n
        client.write(pseudo + ': ' + msg);
      }
    });
    client.on('data', function(data) {
      console.log(data.toString());
      //client.end();
    });
    client.on('end', function() {
      console.log('disconnected from server');
    });
  });
  rl.close();
});
```

# Node.js - HTTP



- CreateServer
  - Contrairement à d'autres technologies, l'implémentation du serveur HTTP se fait dans l'application.
- Callback
  - Une fonction de rappel est associée au serveur. Elle sera appelée à chaque requête HTTP.
- Objets Request et Response
  - Node.js abstrait la requête (IncomingMessage) et la réponse (ServerResponse), le callback doit créer une réponse valide avant la fin de son exécution.

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

http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, world !');
}).listen(8080);
```

# Node.js - HTTPS



- **HTTPS**

Le serveur HTTPS démarre avec une clé privée et un certificat. Les serveurs HTTP et HTTPS cohabitent dans la même application.

```
var https = require('https');
var http = require('http');
var fs = require('fs');

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

var options = {
    key: fs.readFileSync('./key.pem', 'utf8'),
    cert: fs.readFileSync('./server.crt', 'utf8')
};

http.createServer(serveur).listen(80);
https.createServer(options, serveur).listen(443);
```

# Node.js - Gestion des routes



- Pages vs Serveur

Node.js possède une implémentation HTTP très bas niveau. Quand en Java ou en PHP un fichier équivaut à une URL, Node.js lui est le serveur dans son ensemble.

- Routes

La gestion des URL se fait donc en internes, l'association entre un chemin d'accès (Méthode HTTP + URL) et du code s'appelle une route.

- Router

L'ensemble des routes est configurée dans un composant que l'ont appelle un router.

# Node.js - Gestion des routes



- Implémentation d'un router basique

Dans l'exemple ci-dessous, on implémente un router à l'aide d'un simple switch.

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');

  switch (req.url) {
    case '/':
      res.end('Hello World');
      break;
    case '/toto':
      res.end('Hello Toto');
      break;
    default:
      res.statusCode = 404;
      res.end('404 Not Found');
      break;
  }
});

server.listen(8080, () => {
  console.log('Server started http://localhost:8080');
});
```



**formation.tech**

# Express

# Express - Framework Web



- Définition d'un framework web :  
Ensemble de composants logiciels permettant d'architecturer un projet logiciel.
- Différences par rapport à une bibliothèque :  
Le framework ne se destine pas à une tache précise (ensemble de bibliothèques)  
Le framework instaure un cadre de travail (squelettes d'application, documentation sur l'architecture...)



# Express - Frameworks web connus

- Java  
Struts (2000), Spring (2003), GWT (2006), Play (2007)...
- Ruby  
Ruby on Rails (2005), Sinatra (2007)...
- Python  
Django (2005)...
- PHP  
Symfony, Zend Framework, CakePHP, CodeIgniter...

# Express - Frameworks web en JS



- Clients
  - AngularJS (2010), Ember.js (2011), Angular, React/Next.js, Vue/Nuxt, Svelte+SvelteKit...
- Server
  - Express (2009), Hapi (2012)
- Fullstack (Client + Server)
  - Meteor (2012), Sails.js (2012)...



# Express - Introduction

- Express
  - Framework pour Node.js le plus populaire, créé en 2009, aujourd'hui en version 4.
  - Permet d'architecturer plus facilement le serveur web.
  - Très souvent utilisé pour construire des APIs REST.
- Avantages sur le module HTTP de Node.js
  - Gestion des URLs et des méthodes HTTP
  - Approche MVC
  - Utilisation de middlewares qui permettent d'étendre le code
  - De nombreux middleware open-source existent
  - Construit comme une surcouche de HTTP, les objets Request et Response sont simplement étendus
- Installation
  - npm install express --save



# Express - Helloworld

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

const app = express();

app.get('/', (req, res) => {
  res.send('Hello world');
});

app.listen(8080);

console.log("URL http://localhost:8080/");
```

The screenshot shows a web browser window with the URL `localhost:8080` in the address bar, displaying the text "Hello world". Below the browser is the Chrome DevTools Network tab. A request to `localhost` is listed, showing the following details:

- Request URL: `http://localhost:8080/`
- Request Method: GET
- Status Code: 200 OK
- Response Headers:
  - Connection: keep-alive
  - Content-Length: 11
  - Content-Type: text/html; charset=utf-8
  - Date: Fri, 23 Oct 2015 16:31:47 GMT
  - ETag: W/"b-8bd69e52"
  - X-Powered-By: Express

# Express - MVC

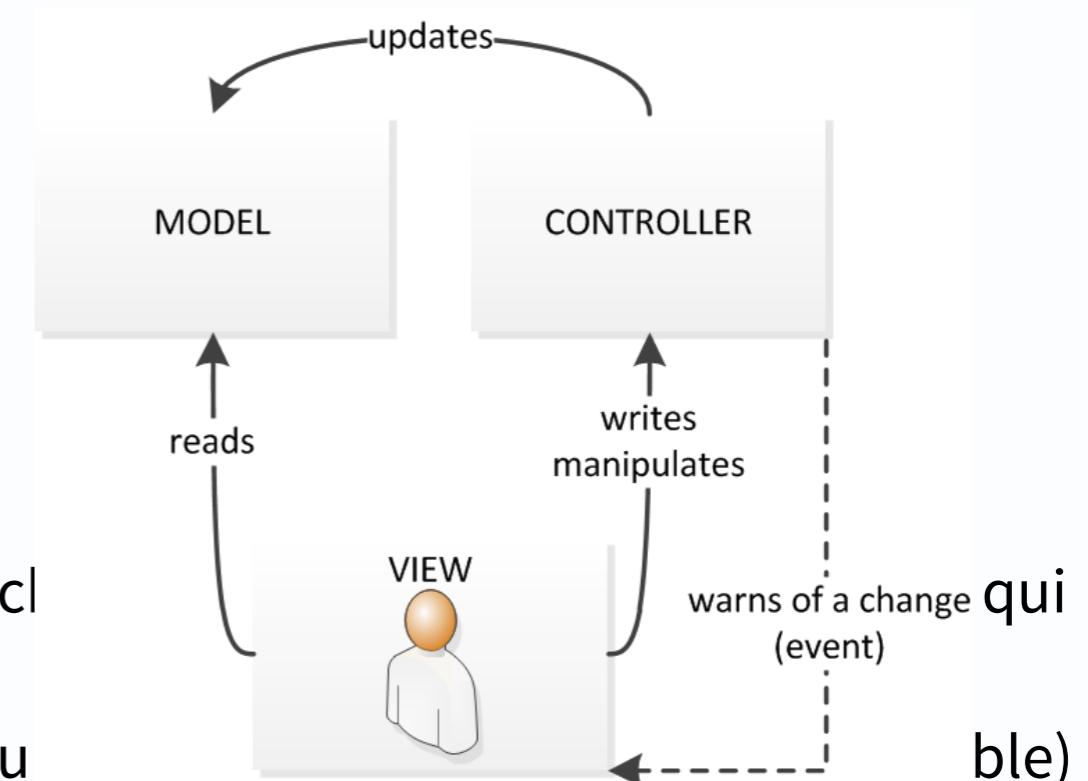


- Définition  
L'architecture MVC est un Design Pattern apparu en Smalltalk et très répandu dans les frameworks web
- Objectif  
L'objectif est de séparer les responsabilités de 3 types de composants : le Modèle (Model), la Vue (View), le Contrôleur (Controller)
- Documentation :  
<http://martinfowler.com/eaaCatalog/modelViewController.html>  
<http://martinfowler.com/eaaDev/uiArchs.html>  
<http://fr.wikipedia.org/wiki/Modèle-vue-contrôleur>

# Express - MVC



- Modèle  
Données, accès aux données, validation
- Vue  
Rendu. Se limiter à :
  - affichage de variable
  - bloc conditionnels if .. else if .. else (ex : affich dépend d'une authentification)
  - boucles foreach (uniquement foreach, ce qu
  - appel à des fonctions de filtre, de formatage, de rendu (parfois appelées aides de vues)
- Contrôleur  
Analyse de la requête, interrogation du Modèle, transmission des données à la vue, gestion des erreurs, des redirections...



# Express - Middleware



- Définition  
Un middleware est une fonction qui va s'exécuter en amont ou en aval d'une requête dans Express pour l'étendre simplement.
- Exemple  
Logs de requêtes, authentification, gestion des requêtes Cross-Domaines, support d'un moteur de templates...
- Connect  
Historiquement Express utilisait le module npm connect pour la mise en place de middleware. A partir d'Express 4, les développeurs d'Express ont développé leur propre système de middleware tout en gardant la compatibilité avec Connect.

# Express - Express Generator



- › Introduction

Express fournit un générateur de squelette d'application pour démarrer rapidement ses projets web (plutôt adapté aux rendus HTML)

- › Installation

```
npm install -g express-generator
```

- › Création du squelette d'application

```
express Helloworld
```

- › Installation des dépendances

```
cd Helloworld && npm install
```

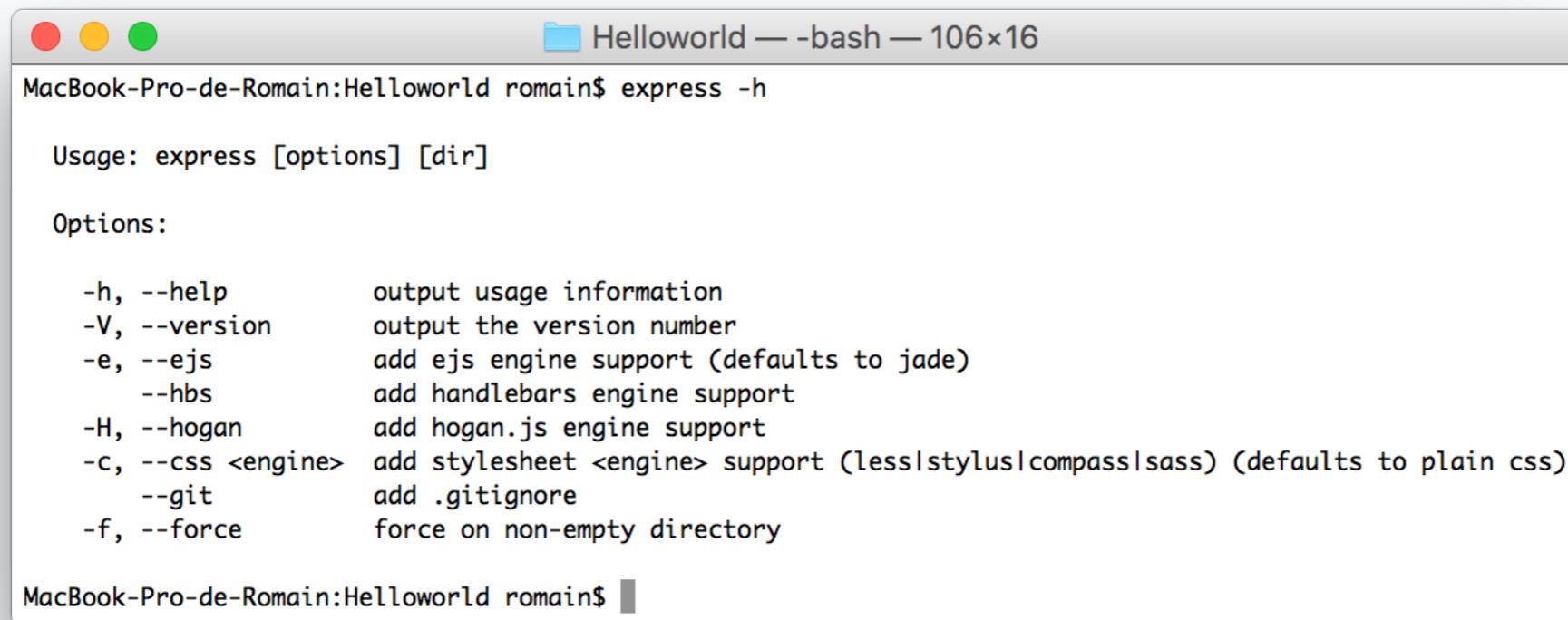
- › Lancement de l'application

```
DEBUG=HelloworldEJS:* npm start
```

# Express - Express Generator



- › Autres options d'installation du squelette



A screenshot of a macOS terminal window titled "Helloworld — bash — 106x16". The window shows the output of the command "express -h". The output includes usage information, options for template engines (ejs, hbs, hogan), and CSS preprocessors (less, stylus, compass, sass). The terminal window has a standard OS X look with red, yellow, and green close buttons at the top left.

```
MacBook-Pro-de-Romain:Helloworld romain$ express -h

Usage: express [options] [dir]

Options:

-h, --help      output usage information
-V, --version   output the version number
-e, --ejs       add ejs engine support (defaults to jade)
--hbs          add handlebars engine support
-H, --hogan     add hogan.js engine support
-c, --css <engine> add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)
--git          add .gitignore
-f, --force     force on non-empty directory

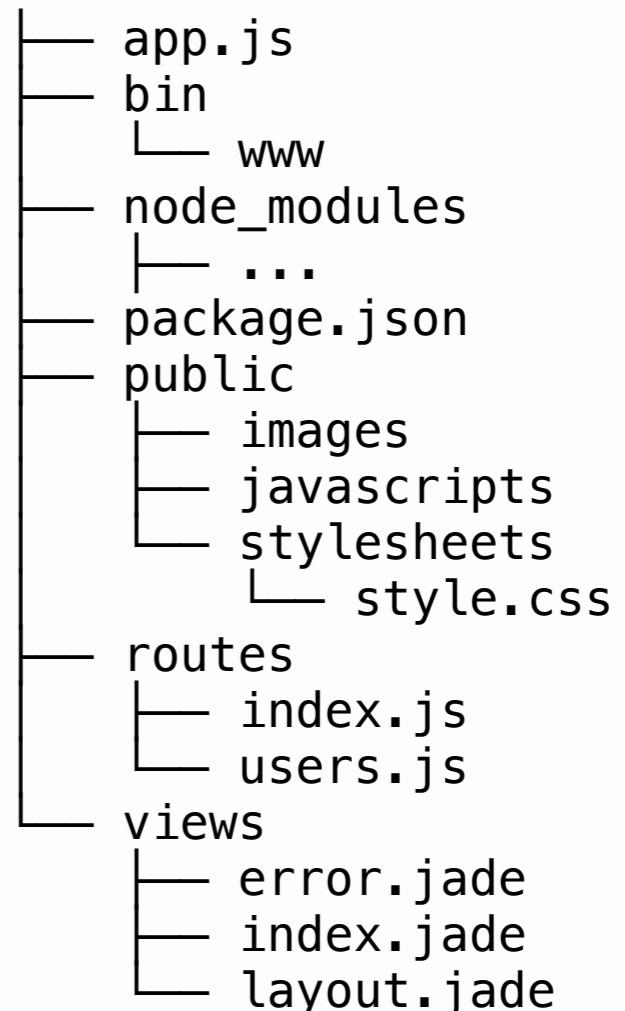
MacBook-Pro-de-Romain:Helloworld romain$
```

- › Choix du moteur de templates  
Jade, EJS, Handlebars, Hogan.js
- › Choix d'un préprocesseur CSS  
CSS, Less, Stylus, Compass, Sass

# Express - Express Generator



- `app.js`  
Configuration de l'application, objet principal
- `bin/www`  
Démarrage du serveur
- `package.json`  
Dépendances npm
- `public`  
Fichiers statiques (images, scripts client, css, pdf...)
- `routes`  
Contrôleurs, configuration des URLs
- `views`  
Fichiers de rendus (ici au format Jade)



# Express - Express Generator



- bin/www
  - Dépendances
  - Définition du port (variable d'env PORT ou 3000)
  - Création du serveur
  - Démarrage du serveur
  - Listeners sur erreurs et démarrage

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var app = require('../app');
var debug = require('debug')('Helloworld:server');
var http = require('http');

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
 * Create HTTP server.
 */

var server = http.createServer(app);

/**
 * Listen on provided port, on all network interfaces.
 */

server.listen(port);
server.on('error', onError);
server.on('listening', onListening);

// ...
```

# Express - Express Generator



- app.js

- Dépendances
- Chargement des routes
- Création de l'objet app
- Définition du moteur de templates
- Définition des middlewares à appeler avant le contrôleur
- Définition des contrôleurs sur un préfixe d'URL
- Middleware pour les erreurs 404
- Middleware pour afficher les erreurs (avec env de dev et de prod)

```
var express = require('express');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
app.use('/users', users);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handlers
// ...
```

# Express - Express Generator



- routes/index.js
  - Dépendances
  - Association d'un contrôleur à la l'URL GET /
  - Appel de la vues en transmettant la variable title (contenu 'Express')

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

/* GET home page. */
router.get('/', function(req, res, next)
{
    res.render('index', { title:
    'Express' });
});

module.exports = router;
```

# Express - Express Generator



- › views/index.jade

- Jade : Syntaxe très concise, l'indentation fait l'imbrication des balises, parenthèses pour les attributs
- Héritage du layout
- Remplacement du block content du layout par celui de la vue (inspiré de Django Template Engine, Twig...)
- Création de la balise h1 ayant comme contenu la variable title
- Création de la balise p qui concatène Welcome to et la variable title

```
// views/index.jade
extends layout

block content
  h1= title
  p Welcome to #{title}
```

```
// views/layout.jade
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

# Express - Express Generator



- views/index.ejs
  - Syntaxe plus simple que Jade proche de PHP, ASP, JSP
  - <%= title %> : écriture de la variable title

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

# Express - Routeur



- Réponse à toutes les méthodes HTTP

```
router.all('/api/*', requireAuthentication);
```

- Réponse aux requêtes sur certaines méthodes HTTP

Méthodes HTTP : get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search, connect

```
router.get('/', function(req, res){  
    res.send('hello world');  
});
```

- Avec une RegExp

```
router.get(/^\/commits\/((\w+)(?:\.\.(\w+))?)$/, function(req, res){  
    var from = req.params[0];  
    var to = req.params[1] || 'HEAD';  
    res.send('commit range ' + from + '...' + to);  
});
```



# Express - Routeur

- Route avec paramètres nommés

```
router.get('/:id', function(req, res, next) {  
  var id = req.params.id;  
  
  if (!model[id-1]) {  
    return next();  
  }  
  
  res.json({  
    data: model[id-1]  
  });  
});
```

- Ne pas confondre avec la query string  
Ex : /contacts?page=1&limit=100

```
// GET /search?q=tobi+ferret  
req.query.q  
// => "tobi ferret"
```

# Express - Middleware



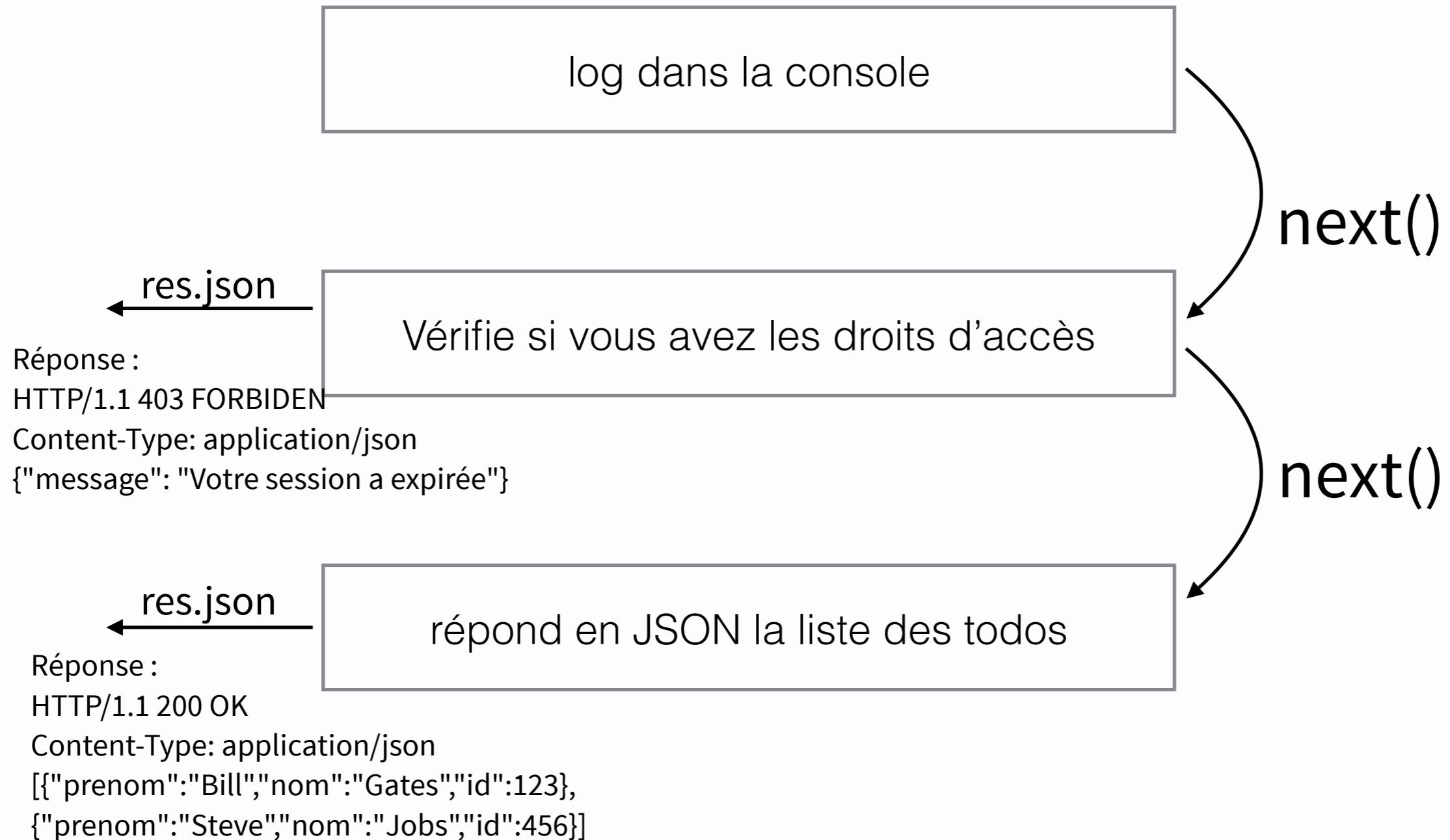
- Middleware  
Fonction qui s'exécute en amont ou en aval d'un contrôleur
- Exemple  
Ajoute les entêtes à la réponse HTTP permettant d'autoriser les requêtes Cross-Domain

```
router.use(function(req, res, next) {  
    res.setHeader('Access-Control-Allow-Origin', '*');  
    res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With');  
    next();  
});
```



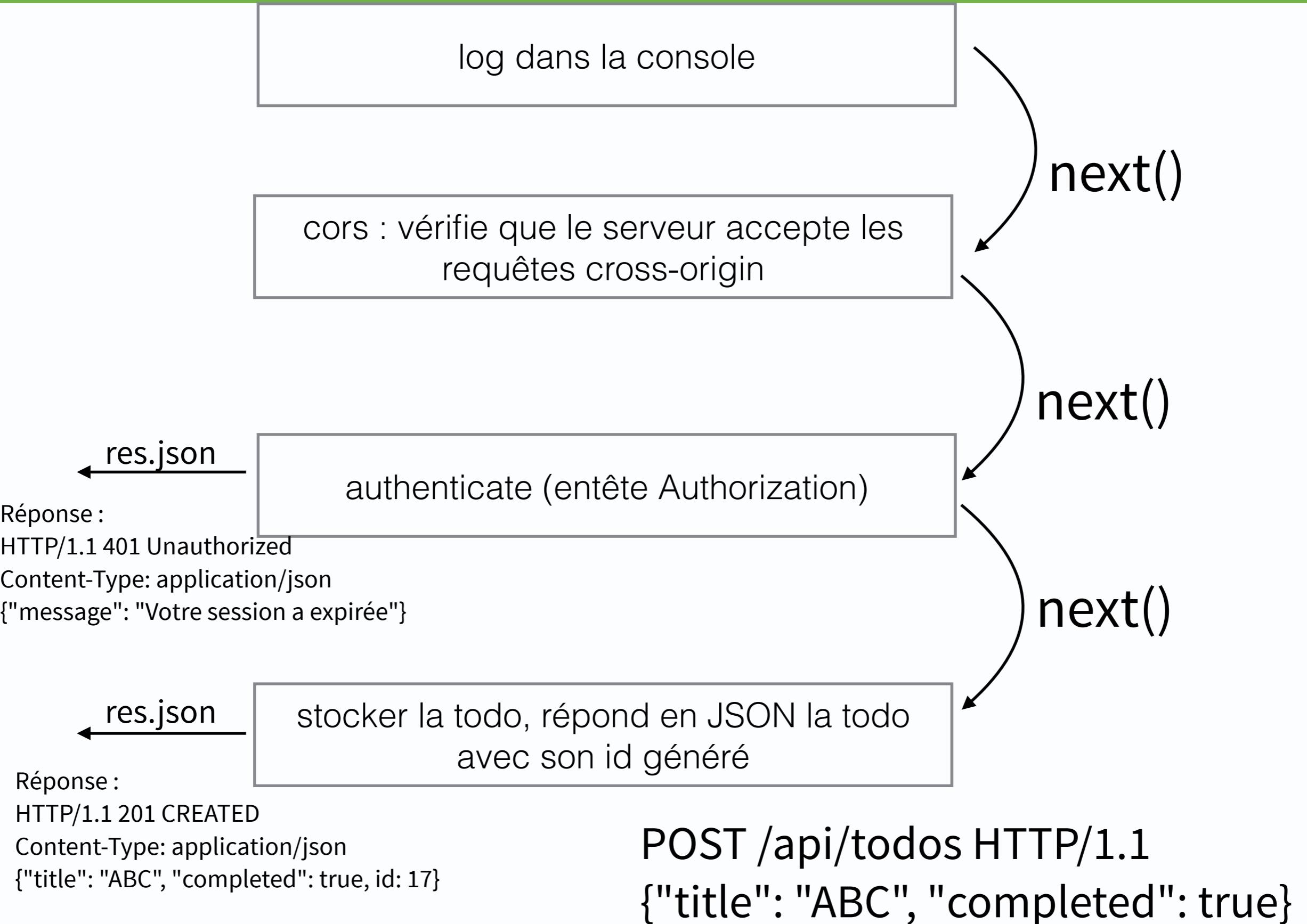
# Express - Middleware

GET /api/todos HTTP/1.1



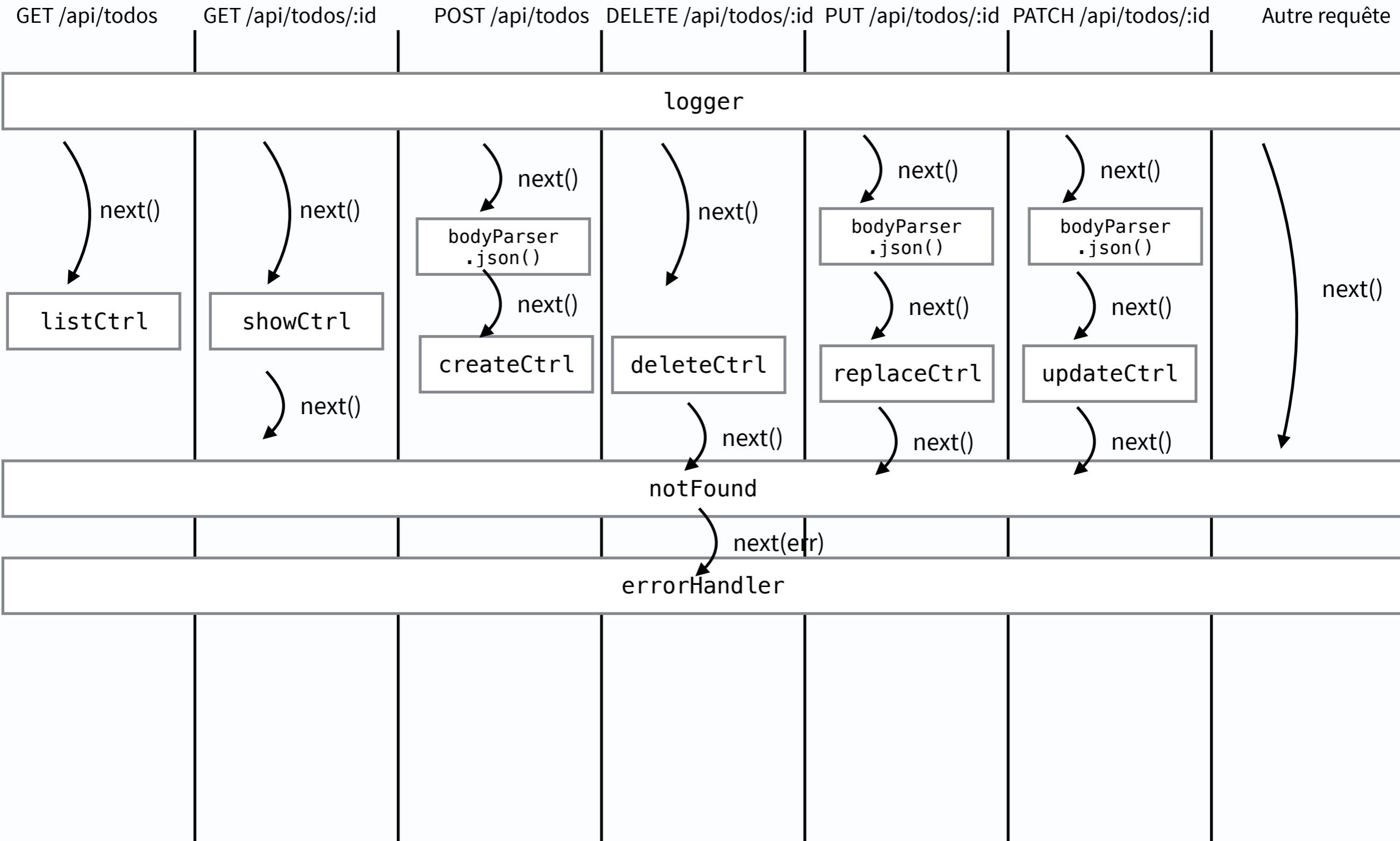


# Express - Middleware





# Express - Middleware



# Express - Middleware



- 3 middlewares
  1. Router qui contient toutes les URLs préfixées par /users
  2. Middleware appelé si l'un des contrôleurs ou middlewares précédents appelle next(), permet de gérer simplement les erreurs 404
  3. Middleware appelé si l'un des contrôleurs ou middlewares précédents appelle next(err) ou les erreurs non-interceptées

```
app.use('/users', users);

app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: err
  });
});
```

# Express - Middleware



- Request  
L'objet Request hérite de IncomingMessage du module HTTP
- Middleware body-parser  
Le middleware body-parser ajouter la propriété body à l'objet request avec le contenu du corps de requête parsé

```
var express = require('express');
var bodyParser = require('body-parser');

var app = express();
app.use(bodyParser.urlencoded({ extended: false }));

app.get('/', function(req, res) {
    var html = '<form method="post">';
    html += '  <p>Prénom : <input name="prenom"></p>';
    html += '  <p>Nom : <input name="nom"></p>';
    html += '  <p><button type="submit">Valider</button></p>';
    html += '</form>';

    res.send(html);
}

app.post('/', function(req, res) {
    // Prénom : Romain, nom : Bohdanowicz
    res.send(`Prénom : ${req.body.prenom}, nom : ${req.body.nom}`);
})

app.listen(3000);
```

# Express - Middleware



- Middleware multer

Le middleware multer ajouter la propriété file ou files (upload multiple) à l'objet request et contient des informations sur le fichier uploadé.

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

var app = express();
var upload = multer({ dest: 'uploads/' });

app.get('/', function(req, res) {
  var html = '<form method="post" enctype="multipart/form-data">';
  html += '  <p>Fichier : <input type="file" name="fichier"></p>';
  html += '  <p><button type="submit">Valider</button></p>';
  html += '</form>';

  res.send(html);
});

app.post('/', upload.single('fichier'), function(req, res){
  console.log(req.file);
  res.status(204).end();
});

app.listen(3000);
```

```
{ filename: 'fichier',
  originalname: '2010_Q3.pdf',
  encoding: '7bit',
  mimetype: 'application/pdf',
  destination: 'uploads/',
  filename: '799e08c05ef96ac6ec6ac5b714941161',
  path: 'uploads/799e08c05ef96ac6ec6ac5b714941161',
  size: 80108 }
```



# Express - JSON

- **JSON**

L'objet Response contient une méthode json qui sérialise un objet et renvoie les bons entêtes HTTP. Associés au méthodes de l'objet Request et aux middleware body-parser et cors, Express est le framework idéal pour la mise en place d'un API REST qui communique en JSON.

```
var app = express();
app.use(cors({ allowedHeaders: 'X-Requested-With' }));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

app.use('/api/v1/contacts', apiContacts);

app.listen(80);
```



# Express - API REST

```
var express = require('express');
var contacts = require('../model/contacts').slice(0);

var api = express.Router();

api.get('/', function(req, res) {
    res.json({contacts});
});

api.get('/:id', function(req, res, next) {
    var id = parseInt(req.params.id);
    var contact = contacts.find(elt => elt.id === id);

    if (!contact) return next();

    res.json({contact});
});

api.post('/', function(req, res, next) {
    var contact = req.body;

    contact.id = contacts[contacts.length-1].id + 1;
    contacts.push(contact);

    res.status(201);
    res.json(contact);
});

module.exports = api;
```

# Express - Designer un API REST



- Guide inspiré de Google, Facebook, Twitter, GitHub...  
<http://blog.octo.com/designer-une-api-rest/>
- Guide inspiré par Heroku  
<https://github.com/interagent/http-api-design>
- Richardson Maturity Model  
<https://martinfowler.com/articles/richardsonMaturityModel.html>



**formation.tech**

# NestJS

# NestJS - Introduction



- Framework web haut niveau pour Node.js
- Plutôt orienté création d'API REST, comparable à API Platform (PHP/Symfony) ou Spring REST
- Fonctionne en JavaScript ou en TypeScript
- Utilise en interne Express ou Fastify et est compatible avec leurs systèmes de plugins/middlewares
- Par rapport aux frameworks sous-jacents NestJS propose une architecture de fichiers, des conventions de code
- Son architecture (Modules, Services, Guards, Interceptors...) est fortement inspiré par Angular
- Crée en 2017 par Kamil Mysliwiec
- Licence MIT
- Doc : <https://docs.nestjs.com/>
- Doc FR : <https://docs.nest-js.fr/>



# NestJS - Installation

- › Installer le programme Nest en ligne de commande  
    `npm i -g @nestjs/cli`
- › Créer un projet  
    `nest new [project-name]`

The screenshot shows a terminal window on a Mac OS X system. The title bar reads "HostAndShare nest new api". The main area of the terminal displays the output of the "nest new" command, which creates a new NestJS project named "api". The terminal shows the creation of various files and folders, including ".eslintrc.js", ".prettierrc", "README.md", "nest-cli.json", "package.json", "tsconfig.build.json", "tsconfig.json", "src/app.controller.spec.ts", "src/app.controller.ts", "src/app.module.ts", "src/app.service.ts", "src/main.ts", "test/app.e2e-spec.ts", and "test/jest-e2e.json". It also asks the user about the package manager to use (npm) and provides instructions to start the project.

```
[→ HostAndShare nest new api
⚡ We will scaffold your app in a few seconds..

CREATE api/.eslintrc.js (599 bytes)
CREATE api/.prettierrc (51 bytes)
CREATE api/README.md (3370 bytes)
CREATE api/nest-cli.json (64 bytes)
CREATE api/package.json (1891 bytes)
CREATE api/tsconfig.build.json (97 bytes)
CREATE api/tsconfig.json (336 bytes)
CREATE api/src/app.controller.spec.ts (617 bytes)
CREATE api/src/app.controller.ts (274 bytes)
CREATE api/src/app.module.ts (249 bytes)
CREATE api/src/app.service.ts (142 bytes)
CREATE api/src/main.ts (208 bytes)
CREATE api/test/app.e2e-spec.ts (630 bytes)
CREATE api/test/jest-e2e.json (183 bytes)

? Which package manager would you ❤️ to use? npm
✓ Installation in progress... 🎈

🎈 Successfully created project api
👉 Get started with the following commands:

$ cd api
$ npm run start
```

# NestJS - Arborescence



```
├── README.md  
├── nest-cli.json  
└── node_modules  
    └── ...  
├── package-lock.json  
└── package.json  
└── src  
    ├── app.controller.spec.ts  
    ├── app.controller.ts  
    ├── app.module.ts  
    ├── app.service.ts  
    └── main.ts  
└── test  
    ├── app.e2e-spec.ts  
    └── jest-e2e.json  
└── tsconfig.build.json  
└── tsconfig.json
```

- nest-cli.json : configuration des générateurs de code (Angular Schematics)
- src : application
- test : tests "End-2-End" / Fonctionnels (Jest)

# NestJS - main.ts



```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

- main.ts est le point d'entrée de l'application
- Contient une fonction bootstrap asynchrone qui crée l'application à partir du module racine et la lance sur le port 3000
- Platforms
  - Par défaut NestJS utilise le framework Express (@nestjs/platform-express)
  - Pour utiliser Fastify :  
<https://docs.nestjs.com/techniques/performance>

# NestJS - Option de create



- cors : Options CORS du package CORS d'express (par défaut : false)
- bodyParser : Indique si le parseur de corps sous-jacent de la plateforme doit être utilisé. (par défaut : true)
- httpsOptions : Ensemble d'options configurables pour HTTPS. (par défaut : undefined)
- rawBody : Indique si le corps brut de la requête doit être enregistré sur la requête. Utilisez req.rawBody. (par défaut : false)
- forceCloseConnections : Force la fermeture des connexions HTTP ouvertes. Utile si le redémarrage de votre application est bloqué à cause des connexions persistantes dans l'adaptateur HTTP. (par défaut : false)
- logger : Spécifie le logger à utiliser. Passez false pour désactiver la journalisation. (par défaut : true)
- abortOnError : Indique s'il faut arrêter le processus en cas d'erreur. Par défaut, le processus est arrêté. Passez false pour annuler le comportement par défaut. Si false est passé, Nest ne fermera pas l'application et relancera l'exception. (par défaut : true)
- bufferLogs : Si activé, les journaux seront mis en mémoire tampon jusqu'à ce que la méthode "Logger#flush" soit appelée. (par défaut : false)
- autoFlushLogs : Si activé, les journaux seront automatiquement vidés et la mémoire tampon détachée lorsque l'initialisation de l'application soit réussie ou échouée. (par défaut : true)
- preview : Indique si l'application doit s'exécuter en mode prévisualisation. En mode prévisualisation, les fournisseurs/contrôleurs ne sont pas instanciés ni résolus. (par défaut : false)
- snapshot : Indique si un instantané du graphique sérialisé doit être généré. (par défaut : false)
- moduleIdGeneratorAlgorithm : Détermine l'algorithme utilisé pour générer les identifiants de module. Lorsqu'il est défini sur deep-hash, l'identifiant du module est généré en fonction de la définition sérialisée du module. Lorsqu'il est défini sur reference, chaque module obtient un identifiant unique basé sur sa référence. (par défaut : 'reference')

# NestJS - Entêtes par défaut de la réponse HTTP



- Entêtes par défaut de la réponse HTTP

Connection:	keep-alive
Content-Length:	12
Content-Type:	text/html; charset=utf-8
Date:	Fri, 24 Jan 2025 10:42:05 GMT
Etag:	W/"c-Lve95gjOVATpfV8EL5X4nxwjKHE"
Keep-Alive:	timeout=5
X-Powered-By:	Express

# NestJS - Modules



```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

- Comme Angular, NestJS fourni un système de module qui permet de :
  - déclarer des contrôleurs
  - fournir des services
  - importer d'autres modules
  - exporter des providers pour les rendre leurs services accessibles depuis d'autres modules

# NestJS - Contrôleur



```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

- Le contrôleur permet de définir les ressources de l'application (URL)
- Architecture MVC : Model Vue Controller
- Sa responsabilité est de s'occuper des problématiques HTTP appeler un service (Model) et de passer les données éventuellement à un template (Vue)
- On essaye de regrouper les URLs d'une même fonctionnalité dans un seul contrôleur
- On paramètre le contrôleur, notamment les routes grâce à des décorateurs (@Controller, @Get...)

# NestJS - Service



```
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

- Un service est une classe qui rend un service à l'application (accéder à la base de données, appeler un API REST, envoyer un email, etc...)
- Dans NestJS, les services peuvent être injectés via un Conteneur d'Injection de Dépendances (DIC), c'est à dire qu'on aura pas à savoir les instancier pour pouvoir les utiliser (Inversion of Control Principle)

# NestJS - Démarrage du serveur



```
{  
  "scripts": {  
    "prebuild": "rimraf dist",  
    "build": "nest build",  
    "start": "nest start",  
    "start:dev": "nest start --watch",  
    "start:debug": "nest start --debug --watch",  
    "start:prod": "node dist/main"  
  }  
}
```

- › Pour démarrer le serveur en développement (redémarrage automatique)

npm run start:dev

- › Pour builder

npm run build

- › Démarrer en prod avec

npm run start:prod

# NestJS - Générateurs de code



```
$ nest
Usage: nest <command> [options]
```

## Options:

- |                            |                             |
|----------------------------|-----------------------------|
| <code>-v, --version</code> | Output the current version. |
| <code>-h, --help</code>    | Output usage information.   |

## Commands:

new [options] [name]	Generate Nest application.
build [options] [apps...]	Build Nest application.
start [options] [app]	Run Nest application.
infoli	Display Nest project details.
add [options] <library>	Adds support for an external library to your project.
generateIg [options] <schematic> [name] [path]	Generate a Nest element.

Schematics available on [@nestjs/schematics](#) collection:

name	alias	description
application	application	Generate a new application workspace
class	cl	Generate a new class
configuration	config	Generate a CLI configuration file
controller	co	Generate a controller declaration
decorator	d	Generate a custom decorator
filter	f	Generate a filter declaration
gateway	ga	Generate a gateway declaration
guard	gu	Generate a guard declaration
interceptor	itc	Generate an interceptor declaration
interface	itf	Generate an interface
library	lib	Generate a new library within a monorepo
middleware	mi	Generate a middleware declaration
module	mo	Generate a module declaration
pipe	pi	Generate a pipe declaration
provider	pr	Generate a provider declaration
resolver	r	Generate a GraphQL resolver declaration
resource	res	Generate a new CRUD resource
service	s	Generate a service declaration
sub-app	app	Generate a new application within a monorepo

# NestJS - Pipes



- TODO

```
import { PipeTransform, Injectable, BadRequestException } from '@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform {
  transform(value: any) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed');
    }
    return val;
  }
}
```



**formation.tech**

# Contrôleurs

# Contrôleurs - Introduction



- Les contrôleurs : points d'entrée pour gérer les requêtes HTTP.
- Les contrôleurs gèrent les requêtes entrantes et renvoient des réponses au client.
- Définis par des décorateurs spécifiques (@Controller).
- Théoriquement leur responsabilité se limite à la requête et réponse HTTP et de chef d'orchestre (ils savent quels services appeler pour récupérer les données)

```
import { Controller, Get } from '@nestjs/common';

@Controller('example')
export class ExampleController {
    @Get()
    findAll(): string {
        return 'This action returns all items';
    }
}
```

# Contrôleurs - Routes



- Le décorateur `@Controller` reçoit en paramètre un préfixe d'URL
- Chaque méthode HTTP possède son propre décorateur `@Get`, `@Post`, `@Put`, `@Delete`, `@Patch` ou `@All` si on veut matcher toutes les méthodes HTTP

```
import { Controller, Get } from '@nestjs/common';

@Controller('example')
export class ExampleController {
  @Get()
  findAll(): string {
    return 'This action returns all items';
  }
}
```

- Par défaut, les retours `object` et `array` sont convertis en JSON, les retours `string`, `number`, `boolean` sont retournés tel quel (on peut utiliser l'objet `Response` d'Express si on souhaite un autre comportement)



# Contrôleurs - Async

- Les méthodes async (qui retourne donc une promesse) sont parfaitement supportées

```
@Get()  
async findAll(): Promise<any[]> {  
    return [];  
}
```

- Ainsi que les Observable RxJS (dans le style Angular)

```
@Get()  
async findAll(): Observable<any[]> {  
    return of([]);  
}
```



# Contrôleurs - Paramètres

- Les paramètres se définissent comme dans Express avec :nomDuParam et se récupère avec le décorateur @Param

```
@Get(':id')
findOne(@Param('id') id: string) {
}
```

- On peut définir un Wildcard avec \*

```
@Get('abcd/*')
findAll() {
    return 'This route uses a wildcard';
}
```



# Contrôleurs - Décorateur de paramètres

- Les décorateurs de paramètres

Décorateur	Concept dans Express
@Request(), @Req()	req
@Response(), @Res()*	res
@Next()	next
@Session()	req.session
@Param(key?: string)	req.params / req.params[key]
@Body(key?: string)	req.body / req.body[key]
@Query(key?: string)	req.query / req.query[key]
@Headers(name?: string)	req.headers / req.headers[name]
@Ip()	req.ip
@HostParam()	req.hosts

- Exemple

```
@Put(':id')
async update(@Param('id') id: string, @Body() post: Partial<BlogPost>) {
}
```



# Contrôleurs - Body

- Par défaut bodyParser est déjà activé
- On peut le désactiver avec bodyParser: false lors de l'appel à la NestFactory.create
- On peut également activer rawBody: true dans NestFactory.create pour récupérer req.rawBody
- Par défaut, json et urlencoded sont supporté, on peut enregistrer autre chose avec :  
app.useBodyParser('text');
- On peut modifier la limite du body reçu (par défaut 100kb) :  
app.useBodyParser('json', { limit: '10mb' });

```
@Put(':id')
async update(@Param('id') id: string, @Body() post: Partial<BlogPost>) {
  return post.title;
}
```



# Contrôleurs - Status code

- Les status codes se définissent via le décorateur @HttpCode

```
@Post()  
@HttpCode(204)  
create() {  
    return 'This action adds a new cat';  
}
```

- On peut également les générer via l'objet Response d'Express ou bien en lançant l'un des exceptions suivantes :

- BadRequestException
- UnauthorizedException
- NotFoundException
- ForbiddenException
- NotAcceptableException
- RequestTimeoutException
- ConflictException
- GoneException
- HttpVersionNotSupportedException
- PayloadTooLargeException
- UnsupportedMediaTypeException
- UnprocessableEntityException
- InternalServerErrorException
- NotImplementedException
- ImATeapotException
- MethodNotAllowedException
- BadGatewayException
- ServiceUnavailableException
- GatewayTimeoutException
- PreconditionFailedException



# Contrôleurs - Versionning

- Lorsque l'API est public, il sera nécessaire de gérer des versions afin de ne pas casser les clients existant
- Cela se passe en général au niveau de l'URL (mais NestJS supporte également les entêtes HTTP, Query String...)

```
const app = await NestFactory.create(AppModule);
// or "app.enableVersioning()"
app.enableVersioning({
  type: VersioningType.URI,
});
await app.listen(process.env.PORT ?? 3000);
```

- Les URL seront de la forme  
<https://example.com/v1/route>  
<https://example.com/v2/route>
-



# Contrôleurs - Versionning

- Versionner suppose avoir plusieurs version en ligne ne même temps
- Cela peut se faire au niveau du contrôleur :

```
import { Controller } from '@nestjs/common';

@Controller({path: 'example', version: '1'})
export class ExampleV1Controller {}
```

- Pour être dispo dans plusieurs versions :

```
import { Controller } from '@nestjs/common';

@Controller({path: 'example', version: ['1','2']})
export class ExampleV1AndV2Controller {}
```



# Contrôleurs - Versionning

- › Ou au niveau de la route :

```
import { Controller, Get, Version } from '@nestjs/common';

@Controller()
export class CatsController {
    @Version('1')
    @Get('cats')
    findAllV1(): string {
        return 'This action returns all cats for version 1';
    }

    @Version('2')
    @Get('cats')
    findAllV2(): string {
        return 'This action returns all cats for version 2';
    }
}
```



# Contrôleurs - Versionning

- On peut également indiquer que le code fonctionne pour toutes les versions :

```
import { Controller, Get, VERSION_NEUTRAL } from '@nestjs/common';

@Controller({
  version: VERSION_NEUTRAL,
})
export class CatsController {
  @Get('cats')
  findAll(): string {
    return 'This action returns all cats regardless of version';
  }
}
```

- Ou bien spécifier une version par défaut et ne plus rien faire au niveau d'un contrôleur / route

```
app.enableVersioning({
  // ...
  defaultVersion: '1'
  // or
  defaultVersion: ['1', '2']
  // or
  defaultVersion: VERSION_NEUTRAL
});
```



formation.tech

# Injection de dépendance

# Injection de Dépendance - Principes SOLID



- SOLID signifie :
  - **S**ingle responsibility principle
  - **O**pen–closed principle
  - **L**iskov substitution principle
  - **I**nterface segregation principle
  - **D**evelopment dependency inversion principle

# Injection de Dépendance - Principes SOLID



- Single responsibility principle

Chaque module, classe ou fonction d'un programme informatique doit être responsable d'une seule partie de la fonctionnalité de ce programme et doit encapsuler cette partie.

# Injection de Dépendance - Principes SOLID



- Open-closed principle

Les entités logicielles (classes, modules, fonctions, etc.) devraient être ouvertes à l'extension, mais fermées à la modification.

# Injection de Dépendance - Principes SOLID



- › Liskov substitution principle

Si  $S$  est un sous-type de  $T$ , les objets de type  $T$  d'un programme peuvent être remplacés par des objets de type  $S$  sans altérer aucune des propriétés souhaitables de ce programme.

# Injection de Dépendance - Principes SOLID



- Interface segregation principle

Plusieurs interfaces spécifiques au client valent mieux qu'une seule interface générale

# Injection de Dépendance - Principes SOLID



- › Dependency inversion principle

Dépendre d'abstractions (interfaces, classes abstraites...) et non de concrétisations (classes)

# Injection de Dépendance - Mauvais Exemple



```
export class CoffeeCup {  
  constructor(  
    private coffeeType: "arabica" | "robusta",  
    private cupCapacity: number  
  ) {}  
}
```

```
import { CoffeeCup } from "./CoffeeCup";  
  
const coffeeCup = new CoffeeCup("arabica", 20);
```



- ✗ Single Responsibility : le café n'est pas séparé de la tasse
- ✗ Open-closed principle : pour créer une tasse de thé, nous devrions modifier le code
- ✗ Dependency inversion principle : CoffeeCup n'a pas de dépendance

# Injection de Dépendance - Mauvais Exemple



```
import fs from "fs/promises";

export class FileLogger {
  constructor(private filePath: string) {}

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await fs.appendFile(this.filePath, formatted);
  }
}
```

```
import { FileLogger } from "./FileLogger";

const logger = new FileLogger("app.log");
await logger.log('Message');
```

- ✗ Single Responsibility : l'écriture de fichiers n'est pas séparée du logging
- ✗ Open-closed principle : pour loguer dans le terminal, il faudrait modifier le code
- ✗ Dependency inversion principle : FileLogger n'a pas de dépendance

# Injection de Dépendance - Mauvais Exemple



```
export class Coffee {  
  constructor(  
    private type: "arabica" | "robusta",  
  ) {}  
}
```

```
import { Coffee } from "./Coffee";  
  
export class Cup {  
  private coffee: Coffee;  
  constructor(  
    private capacity: number  
  ) {  
    this.coffee = new Coffee('arabica');  
  }  
}
```

```
import { Cup } from "./Cup";  
  
const coffeeCup = new Cup(10);
```



- ✓ Single Responsibility : 1 classe pour le café, 1 classe pour la tasse
- ✗ Open-closed principle : pour créer une tasse de thé, il faudrait modifier le code
- ✗ Dependency inversion principle : Coffee est une dépendance codée en dur, il ne peut pas être remplacé dynamiquement

# Injection de Dépendance - Mauvais Exemple



```
import fs from "fs/promises";

export class FileWriter {
  constructor(private filePath: string) {}

  async write(msg: string) {
    await fs.appendFile(this.filePath, msg);
  }
}
```

```
import { FileWriter } from "./FileWriter";

export class Logger {
  private fileWriter: FileWriter;

  constructor(filePath: string) {
    this.fileWriter = new FileWriter(filePath);
  }

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await this.fileWriter.write(formatted);
  }
}
```

```
import { Logger } from "./Logger";

const logger = new Logger("app.log");
await logger.log('Message');
```

# Injection de Dépendance - Mauvais Exemple



- ✓ Single Responsibility : 1 classe pour l'écriture des fichiers, 1 classe pour le logging
- ✗ Open-closed principle : pour loguer dans le terminal, il faudrait modifier le code
- ✗ Dependency inversion principle : Coffee est une dépendance codée en dur, elle ne peut pas être remplacée dynamiquement

# Injection de Dépendance - Mauvais Exemple



```
export class Coffee {  
  constructor(  
    private type: "arabica" | "robusta",  
  ) {}  
}
```

```
import { Coffee } from "./Coffee";  
  
export class Cup {  
  constructor(  
    private capacity: number,  
    private coffee: Coffee  
  ) {}  
}
```

```
import { Coffee } from "./Coffee";  
import { Cup } from "./Cup";  
  
const coffee = new Coffee('arabica');  
const coffeeCup = new Cup(10, coffee);
```



- ✓ Single Responsibility : 1 classe pour le café, 1 classe pour la tasse
- ✓ Open-closed principle : nous pourrions passer une spécialisation de Coffee pour modifier le code (mais cela pourrait briser le principe de substitution de Liskov)
- ✗ Dependency inversion principle : Coffee est une dépendance codée en dur, elle ne peut pas être remplacée dynamiquement

# Injection de Dépendance - Mauvais Exemple



```
import fs from "fs/promises";

export class FileWriter {
    constructor(private filePath: string) {}

    async write(msg: string) {
        await fs.appendFile(this.filePath, msg);
    }
}
```

```
import { FileWriter } from "./FileWriter";

export class Logger {
    constructor(private fileWriter: FileWriter) {}

    async log(msg: string) {
        const formatted = `${new Date().toISOString()} - ${msg}\n`;
        await this.fileWriter.write(formatted);
    }
}
```

```
import { FileWriter } from "./FileWriter";
import { Logger } from "./Logger";

const writer = new FileWriter("app.log");
const logger = new Logger(writer);
await logger.log('Message');
```

# Injection de Dépendance - Mauvais Exemple



- ✓ Single Responsibility : 1 classe pour l'écriture de fichiers, 1 classe pour le logging
- ✓ Open-closed principle : nous pourrions passer une spécialisation de FileWriter pour modifier le code (mais cela pourrait briser le principe de substitution de Liskov)
- ✗ Dependency inversion principle : Coffee est une dépendance codée en dur, elle ne peut pas être remplacée dynamiquement

# Injection de Dépendance - Bon Exemple



```
export interface DrinkInterface {}  
  
import { DrinkInterface } from "./DrinkInterface";  
  
export class Coffee implements DrinkInterface {  
  constructor(private type: "arabica" | "robusta") {}  
}  
  
import { DrinkInterface } from "./DrinkInterface";  
  
export class Cup {  
  constructor(private capacity: number, private drink: DrinkInterface)  
}  
  
import { Coffee } from "./Coffee";  
import { Cup } from "./Cup";  
  
const coffee = new Coffee("arabica");  
const coffeeCup = new Cup(10, coffee);
```



- ✓ Single Responsibility : 1 classe pour le café, 1 classe pour la tasse
- ✓ Open-closed principle : on peut passer une implémentation de DrinkInterface pour modifier le code
- ✓ Dependency inversion principle : la tasse dépend d'une abstraction, nous pourrions facilement créer une tasse de thé



# Injection de Dépendance - Bon Exemple

```
export interface WriterInterface {
  write(msg: string): Promise<void>;
}

import fs from "fs/promises";
import { WriterInterface } from "./WriterInterface";

export class FileWriter implements WriterInterface {
  constructor(private filePath: string) {}

  async write(msg: string) {
    await fs.appendFile(this.filePath, msg);
  }
}

import { WriterInterface } from "./WriterInterface";

export class Logger {
  constructor(private writer: WriterInterface) {}

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await this.writer.write(formatted);
  }
}

import { FileWriter } from "./FileWriter";
import { Logger } from "./Logger";

const writer = new FileWriter("app.log");
const logger = new Logger(writer);
await logger.log('Message');
```

# Injection de Dépendance - Bon Exemple

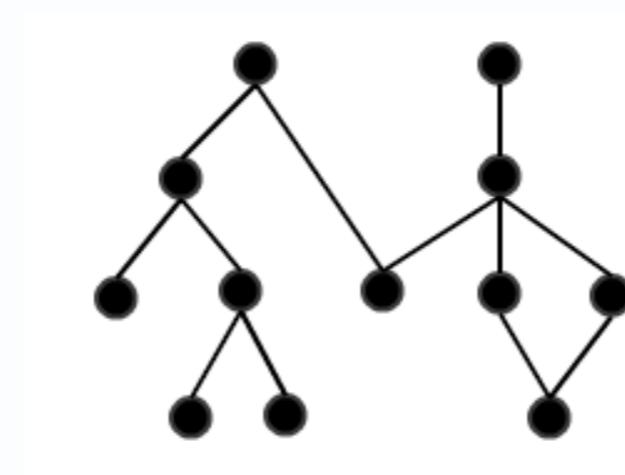


- ✓ Single Responsibility : 1 classe pour l'écriture des fichiers, 1 classe pour le logging
- ✓ Open-closed principle : on peut passer une implémentation de WriterInterface pour modifier le code
- ✓ Dependency inversion principle : Logger dépend d'une abstraction, nous pourrions facilement créer un logger qui loguerait ailleurs que dans un fichier



# Injection de Dépendance - DIC

- A force de respecter ces bons principes de programmation, on peut arriver à des graphes d'injection de dépendance complexes



- Exemple :  
4 instances pour récupérer le client HTTP dans Angular

```
const xhrFactory = {
  build: () => new XMLHttpRequest()
}
const httpHandler = new HttpXhrBackend(xhrFactory)
const httpClient = new HttpClient(httpHandler);
```

# Injection de Dépendance - DIC



- Pour simplifie on va utiliser une bibliothèque pour gérer nos dépendances
- On appelle cela un Conteneur d'Injection de Dépendance (DIC)
- Il faut voir ça comme un système clé / valeur, où :
  - la clé permet de paramétrier et retrouver notre dépendance
  - la valeur permet d'indiquer comment le service doit être instancié (au chargement de l'application, à la demande, en fonction d'une config/d'un environnement...)



**formation.tech**

# Services

# Services - Introduction



- Un service est une classe qui encapsule la logique métier.
- Fournit des fonctionnalités réutilisables dans les contrôleurs ou d'autres services.
- Facilitent la séparation des responsabilités dans une application.
- Réduction de la duplication de code.
- Facilite les tests unitaires grâce à l'injection de dépendances.
-



# Services - Utilisation

- Pour déclarer un service on utilise le décorateur Injectable

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class ExampleService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

- Le service doit également être enregistré (ou importé) au niveau d'un module pour être disponible au niveau d'un contrôleur de ce module.

```
@Module({
  providers: [ExampleService],
  controller: [ExampleController],
})
export class ExampleModule {}
```



# Services - Utilisation

- Le service sera injecté au niveau du contrôleur en déclarant un constructeur

```
import { Controller, Get } from '@nestjs/common';
import { ExampleService } from './example.service';

@Controller()
export class ExampleController {
  constructor(private readonly exampleService: ExampleService) {}

  @Get()
  getHello(): string {
    return this.exampleService.getHello();
  }
}
```



# Services - Utilisation

- Le service peut lui même dépendre d'un ou plusieurs autres services (qui devront également être déclarés ou importés par notre Module)

```
@Injectable()
export class AnotherService {
  constructor(private readonly exampleService: ExampleService) {}

  getMessage(): string {
    return this.exampleService.getHello();
  }
}
```



# Services - Configuration

- La config par défaut d'un service :

```
@Module({
  controllers: [ExampleController],
  providers: [ExampleService]
})
export class ExampleModule {}
```

- Est en réalité la version courte de :

```
@Module({
  controllers: [ExampleController],
  providers: [
    {
      provide: ExampleService,
      useClass: ExampleService
    }
  ]
})
export class ExampleModule {}
```

- Où :

- `provide` permet de configurer la clé
- `useClass` (ou autre) permet d'indiquer à NestJS comment créer l'objet

# Services - Configuration



- Comment configurer l'objet
  - useClass : le cas le plus courant, on est en mode "auto", soit le service n'a pas de dépendance, soit ses dépendances sont des instances de classes qui sont elles-mêmes configurées dans les providers (ou importées)
  - useValue : on donne directement l'objet qui sera donc instancié au démarrage de l'application et non pas à la demande. Utile pour des objets très globaux (config) ou bien dans le test où l'on cherche souvent à remplacer le service par un double

```
@Module({
  controllers: [ExampleController],
  providers: [
    {
      provide: ExampleService,
      useValue: {
        getHello: () => 'Hello World!',
      }
    },
  ],
})
export class ExampleModule {}
```



# Services - Configuration

- `useFactory` : on donne une fonction, qui sera appelé uniquement si on demande l'objet, utile quand la création est conditionnelle ou bien que les dépendances ne sont pas des services (string...)

```
@Module({
  controllers: [ExampleController],
  providers: [
    {
      provide: ExampleService,
      useValue: () => {
        return new ExampleService('une chaîne de caractères');
      },
    },
  ],
})
export class ExampleModule {}
```

- avec une factory on peut également récupérer un autre service avec `inject`

```
@Module({
  controllers: [ExampleController],
  providers: [
    {
      provide: ExampleService,
      useFactory: (config: ConfigService) => {
        return new ExampleService(config.get('key'));
      },
      inject: [ConfigService],
    },
  ],
})
export class ExampleModule {}
```



# Services - Configuration

- La factory peut être une fonction async

```
{  
  provide: 'ASYNC_CONNECTION',  
  useFactory: async () => {  
    const connection = await createConnection(options);  
    return connection;  
  },  
}
```



# Services - Configuration

- `useExisting` : permet de réaffecter la config associée à une autre clé (alias)

```
@Module({
  controllers: [ExampleController],
  providers: [
    {
      provide: ExampleService,
      useExisting: AnotherService,
    },
  ],
})
export class ExampleModule {}
```



# Services - Configuration

- ▶ Scopes

Le scope permet de configurer la mise en cache du service

```
import { Module, Scope } from '@nestjs/common';
import { ExampleController } from './example.controller';
import { ExampleService } from './example.service';

@Module({
  controllers: [ExampleController],
  providers: [
    {
      provide: ExampleService,
      useClass: ExampleService,
      scope: Scope.DEFAULT,
    },
  ],
})
export class ExampleModule {}
```

- ▶ 3 valeurs :

- Scope.DEFAULT : sera mis en cache pendant tout le durée de vie du serveur (singleton)
- Scope.TRANSIENT : n'est pas mis en cache
- Scope.REQUEST : est mis en cache le temps de la requête HTTP

- ▶ Les scopes peuvent également s'appliquer à un Controller

```
@Controller({ path: 'posts', scope: Scope.REQUEST })
export class PostController {
```

# Services - Configuration



- Généralement la clé est une classe, mais cela peut également être autre chose, souvent :
  - une chaîne de caractère
  - un Symbol JavaScript (si risque de conflit sur la chaîne de caractère)
- La clé doit exister une fois le code TS transformé en JS (ne pas utiliser une interface par exemple)
- Dans ce cas on doit utiliser le décorateur `@Inject` pour retrouver son objet

```
import { connection } from './connection';

@Module({
  providers: [
    {
      provide: 'CONNECTION',
      useValue: connection,
    },
  ],
})
export class AppModule {}      @Injectable()
                            export class CatsRepository {
                                constructor(@Inject('CONNECTION') connection: Connection) {}
                            }
```



# Services - Configuration

- Dans de rare cas, la dépendance peut être circulaire, exemple : ExampleService → AnotherService → ExampleService
- C'est en général un problème de conception (sauf dans de rares cas)
- Pour éviter une boucle infinie on utilise la fonction forwardRef

```
@Injectable()
export class CommonService {
  constructor(
    @Inject(forwardRef(() => CatsService))
    private catsService: CatsService,
  ) {}
}
```



# Services - Configuration

- Parfois, certaines dépendances ne sont pas indispensables. Par exemple, une classe peut dépendre d'un objet de configuration, mais utiliser des valeurs par défaut si aucun n'est fourni.
- Dans ce cas, la dépendance est optionnelle, on n'aura pas d'erreur si la config du service n'existe pas

```
import { Injectable, Optional, Inject } from '@nestjs/common';

@Injectable()
export class HttpService<T> {
  constructor(@Optional() @Inject('HTTP_OPTIONS') private httpClient: T) {}
}
```



**formation.tech**

# Modules

# Modules - Introduction



- › Les modules sont les blocs de construction principaux dans NestJS.
- › Ils permettent d'organiser et de structurer l'application en unités logiques.
- › Par défaut, NestJS dispose d'un module racine : AppModule.

# Modules - Providers



- Lorsque qu'on configure un service via un provider dans un module il devient accessible dans les contrôleurs du même module
- Par contre un autre module de l'application n'y aura pas accès
- Pour les rendre accessible ailleurs, il faut exporter le provider, puis importer le module qui le déclare en évitant si possible les dépendances cycliques :  
 $A \rightarrow B \rightarrow A$  ou  $A \rightarrow B \rightarrow C \rightarrow B$

```
@Module({
  imports: [],
  providers: [BookService],
  controllers: [BookController],
  exports: [BookService], // Exporté donc utilisables dans d'autres modules
})
export class BookModule {}
```

```
@Module({
  controllers: [AudioBookController], // BookService pourra être utilisé ici
  providers: [AudioBookService],
  imports: [BookModule, StoryModule],
})
export class AudioBookModule {}
```

# Modules - Modules Dynamiques



- Il est également possible de faire appel à un module "dynamique", c'est à dire un module configurable à son enregistrement
- Dans ce cas NestJS recommande l'utilisation des noms de méthodes :
  - register / registerAsync : si la config est spécifique au module importateur et pourra être rappelée dans des sous-modules
  - forRoot / forRootAsync : à pour vocation d'être appelée une seule fois au niveau du module racine  
exemple TypeOrmModule.forRoot()
  - forFeature / forFeatureAsync : on a déjà utilisé forRoot sur un module racine, mais on souhaite personnaliser

```
@Module({})
export class DynamicModule {
  static forRoot(options: ConfigOptions): DynamicModule {
    return {
      module: DynamicModule,
      providers: [{ provide: CONFIG_OPTIONS, useValue: options }],
    };
  }
}
```



**formation.tech**

# Configuration

# Configuration - Introduction



- Qu'est-ce que @nestjs/config ?
  - Un module intégré pour gérer la configuration dans une application NestJS.
  - Permet de charger, valider et organiser des variables d'environnement.
- Pourquoi l'utiliser ?
  - Centralise la gestion des configurations.
  - Facilite l'utilisation des fichiers .env.
  - Fournit des options de validation via des schémas (ex : Joi).
  - Gère les environnements dev, prod, staging...



# Configuration - Mise en place

- Installation :  
  `npm install @nestjs/config`
- Enregistrement du module

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';

@Module({
  imports: [ConfigModule.forRoot()],
})
export class AppModule {}
```

- Fichier .env

PORT=3000

DATABASE\_URL=postgres://user:pass@localhost/db



# Configuration - Mise en place

- › Lecture des variables d'environnements

```
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';

@Injectable()
export class AppService {
  constructor(private configService: ConfigService) {}

  getPort(): string {
    return this.configService.get<string>('PORT');
  }
}
```



# Configuration - Validation

- Validation avec joi  
npm install joi

```
import * as Joi from 'joi';

ConfigModule.forRoot({
  validationSchema: Joi.object({
    PORT: Joi.number().default(3000),
    DATABASE_URL: Joi.string().required(),
  }),
});
```



# Configuration - Validation

- Fichier env multiples

```
ConfigModule.forRoot({
  envFilePath: `^.env.stage.${process.env.STAGE}`, '.env'],
  validationSchema: configValidationSchema,
}),
```



**formation.tech**

# TypeORM



# TypeORM - Introduction

- TypeORM est un ORM (Object-Relational Mapper) pour TypeScript et JavaScript, qui permet de faciliter l'interaction avec les bases de données en manipulant des objets.
- Supporte les bases de données relationnelles : MySQL / MariaDB / Postgres / CockroachDB / SQLite / Microsoft SQL Server / Oracle / SAP Hana / sql.js
- Ne sera pas adapté si l'objectif n'est pas de manipuler ses objects en lecture / écriture (statistiques...)
- Permet de décrire les relations
- Facilite les migrations



# TypeORM - Configuration

- › Installation pour SQLite  
npm install --save @nestjs/typeorm typeorm
- › Configuration avec NestJS (installer également sqlite3 avec npm)

```
@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'sqlite',
      database: 'data/db.sqlite',
      entities: [Product, Category, Order, User],
      // entities: [__dirname + '/**/*.entity{.ts,.js}'],
      synchronize: true,
    }),
    // ...
  ],
  controllers: [],
  providers: []
})
export class AppModule {}
```



# TypeORM - Configuration

- › Pour MySQL (installer également mysql ou mysql2 avec npm)

```
TypeOrmModule.forRoot({
  type: 'mysql',
  host: 'localhost', // ou l'IP du serveur MySQL
  port: 3306, // port par défaut pour MySQL
  username: 'root', // ton utilisateur MySQL
  password: 'motdepasse', // ton mot de passe MySQL
  database: 'nom_de_la_base', // le nom de la base de données
  entities: [User, Post], // exemple d'entités
  synchronize: false, // ne pas utiliser en production (la gestion des migrations est préférable)
  logging: true, // pour afficher les logs SQL dans la console
});
```

- › Pour PostgreSQL (installer également pg avec npm)

```
TypeOrmModule.forRoot({
  type: 'postgres',
  host: 'localhost', // ou l'IP du serveur PostgreSQL
  port: 5432, // port par défaut pour PostgreSQL
  username: 'postgres', // ton utilisateur PostgreSQL
  password: 'motdepasse', // ton mot de passe PostgreSQL
  database: 'nom_de_la_base', // le nom de la base de données
  entities: [User, Post], // exemple d'entités
  synchronize: false, // ne pas utiliser en production (la gestion des migrations est préférable)
  logging: true, // pour afficher les logs SQL dans la console
});
```



# TypeORM - CLI

- Le CLI permet de générer les tables à partir du mapping ou bien de générer des migrations (prod)

```
npx typeorm
Usage: typeorm <command> [options]
```

## Commands:

typeorm schema:sync	Synchronizes your entities with database schema. It runs schema update queries on all connections you have. To run update queries on a concrete connection use -c option.
typeorm schema:log	Shows sql to be executed by schema:sync command. It shows sql log only for your default dataSource. To run update queries on a concrete connection use -c option.
typeorm schema:drop	Drops all tables in the database on your default dataSource. To drop table of a concrete connection's database use -c option.
typeorm query [query]	Executes given SQL query on a default dataSource. Specify connection name to run query on a specific dataSource.
typeorm entity:create <path>	Generates a new entity.
typeorm subscriber:create <path>	Generates a new subscriber.
typeorm migration:create <path>	Creates a new migration file.
typeorm migration:generate <path>	Generates a new migration file with sql needs to be executed to update schema.
typeorm migration:run	Runs all pending migrations.
typeorm migration:show	Show all migrations and whether they have been run or not
typeorm migration:revert	Reverts last executed migration.
typeorm version	Prints TypeORM version this project uses.
typeorm cache:clear	Clears all data stored in query runner cache.
typeorm init	Generates initial TypeORM project structure. If name specified then creates files inside directory called as name. If its not specified then creates files inside current directory.

## Options:

-h, --help	Show help	[boolean]
-v, --version	Show version number	[boolean]



# TypeORM - Mapping basique

- Exemple de mapping simple :

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()
export class Post {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    title: string;

    @Column('text')
    content: string;

    @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })
    createdAt: Date;

    @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP', onUpdate: 'CURRENT_TIMESTAMP' })
    updatedAt: Date;
}
```



# TypeORM - Décorateur @Entity()

- Le décorateur `@Entity()` est utilisé pour marquer une classe comme étant une entité persistée dans une base de données. Cela signifie que TypeORM va créer une table dans la base de données et associer les instances de cette classe à des lignes de cette table.
- Syntaxe de base

```
@Entity()  
export class User {  
    @PrimaryGeneratedColumn()  
    id: number;  
  
    @Column()  
    name: string;  
}
```



# TypeORM - Décorateur @Entity()

- L'option name permet de spécifier le nom de la table dans la base de données. Si cette option est omise, le nom de la table sera dérivé automatiquement du nom de la classe (en minuscules, avec un \_ comme séparateur si la classe utilise le camelCase).
- Exemple :

```
@Entity("users") // ou @Entity({ name: 'users' })
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;
}
```

- Dans cet exemple, la table générée sera appelée "users" au lieu du nom par défaut dérivé de "User".



# TypeORM - Décorateur @Entity()

- Autres options utiles :
  - schema : choix de la base de données si multiples
  - indexes : définition des index
  - uniqueConstraints : définition des index uniques

```
@Entity({
  name: "customers",
  schema: "sales",
  indexes: [
    { name: "IDX_CUSTOMER_EMAIL", columns: ["email"] },
  ],
})
export class Customer {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  email: string;

  @Column()
  phoneNumber: string;
}
```



# TypeORM - Décorateur @Column()

- Le décorateur @Column est utilisé pour déclarer les colonnes d'une entité et spécifier les propriétés de la base de données associées.
- Options :
  - type : Type de la colonne (par ex. varchar, int)
  - length : Longueur maximale pour les colonnes de type string
  - nullable : Permet de définir une colonne qui peut être NULL
  - unique : Rendre les valeurs de la colonne uniques
  - default : Valeur par défaut pour la colonne
  - primary : Déclare une colonne comme clé primaire
  - select : Inclure ou exclure la colonne dans les sélections de requêtes



# TypeORM - Décorateur @Column()

- Exemple

```
@Entity()  
export class User {  
    @PrimaryGeneratedColumn()  
    id: number;  
  
    @Column({ type: 'varchar', length: 100, unique: true })  
    username: string;  
  
    @Column({ type: 'varchar', nullable: true })  
    middleName: string;  
  
    @Column({ type: 'boolean', default: true })  
    isActive: boolean;  
}
```



# TypeORM - Décorateur @Column()

- Les types de mapping sont déterminés automatiquement à partir du type TypeScript mais il est possible de les modifier si on veut des variantes :
- string : Chaîne de caractères, utilisée pour les colonnes de type texte (VARCHAR, TEXT).
- number : Valeur numérique (généralement correspond à INTEGER, INT, DECIMAL, FLOAT, etc.).
- boolean : Valeur booléenne (TRUE ou FALSE).
- date : Date sans heure (correspond à DATE dans la base de données).
- date avec `@Column('timestamp')` : Date et heure (correspond à TIMESTAMP dans la base de données).
- json : Stocke des objets JSON ou des tableaux (correspond à JSON ou JSONB).
- uuid : Utilisé pour les identifiants universels uniques (UUID).
- binary : Stocke des données binaires (par exemple des fichiers ou des images).
- decimal : Pour les valeurs numériques avec des décimales (utilisé pour des valeurs précises comme de l'argent).
- text : Texte long (équivalent à TEXT en base de données, utilisé pour de grandes chaînes de caractères).
- simple-array : Un tableau simple (chaînes ou nombres), stocké sous forme de texte séparé par des virgules.
- simple-json : Un objet JSON simple stocké sous forme de chaîne (converti en JSON lors de la lecture et l'écriture).
- Voir la liste complète (certains types dépendent de la base) :  
<https://orkhan.gitbook.io/typeorm/docs/entities#column-types>



# TypeORM - @PrimaryGeneratedColumn

- Permet de définir la clé primaire auto générée (sinon utiliser PrimaryColumn)

```
@Entity()  
export class User {  
  @PrimaryGeneratedColumn("uuid")  
  id: string  
}
```

- Il existe quatre stratégies de génération :
  - increment : utilise AUTO\_INCREMENT / SERIAL / SEQUENCE (selon le type de base de données) pour générer un numéro incrémental. (default)
  - identity : uniquement pour PostgreSQL 10+. Les versions de PostgreSQL supérieures à 10 supportent la colonne IDENTITY conforme à la norme SQL. Lorsqu'on marque la stratégie de génération comme "identity", la colonne sera produite en utilisant GENERATED [ALWAYS|BY DEFAULT] AS IDENTITY.
  - uuid : génère une chaîne uuid unique.
  - rowid : uniquement pour CockroachDB, cf doc



# TypeORM - Colonne spéciales

- `@CreateDateColumn` : Enregistre automatiquement la date d'insertion de l'entité.
- `@UpdateDateColumn` : Enregistre automatiquement la date de mise à jour de l'entité.
- `@DeleteDateColumn` : Enregistre automatiquement la date de suppression de l'entité en cas de soft-delete.
- `@VersionColumn` : Gère automatiquement un numéro de version de l'entité à chaque mise à jour.
- `@VirtualColumn` : Permet de générer une valeur à partir d'une requête liée à l'entité  
Exemple :

```
@VirtualColumn({ query: (alias) => `SELECT COUNT("name") FROM
"employees" WHERE "companyName" = ${alias}.name` })
companyName: string;
```



# TypeORM - Accès aux données

- L'accès aux données se fait une classe Repository

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
  ) {}

  findAll(): Promise<User[]> {
    return this.userRepository.find();
  }

  create(user: Partial<User>): Promise<User> {
    return this.userRepository.save(user);
  }
}
```



# TypeORM - Repository

- Voici les principales méthodes de la classe Repository :
- `find`  
Récupère plusieurs enregistrements selon des options ou critères.
- `findOne`  
Trouve un enregistrement unique par ID ou selon des critères spécifiques.
- `findBy`  
Récupère tous les enregistrements correspondant à un ensemble de critères.
- `findOneBy`  
Trouve un enregistrement unique selon des critères spécifiques.
- `findAndCount`  
Récupère une liste d'enregistrements et leur nombre total.



# TypeORM - Repository

- Options des requêtes find, findOne...
- where

Définit les conditions pour filtrer les enregistrements (peut être un objet, une fonction ou une chaîne).
- relations

Charge les relations associées à l'entité (par exemple, relations OneToMany, ManyToOne).
- select

Spécifie les colonnes à inclure dans les résultats.
- order

Trie les résultats par une ou plusieurs colonnes, avec ASC ou DESC.
- skip

Définit le nombre d'enregistrements à ignorer (utile pour la pagination).
- take

Définit le nombre maximum d'enregistrements à récupérer (limitation des résultats).
- loadRelationIds

Charge uniquement les IDs des relations au lieu de leurs objets complets.
- withDeleted

Inclut les enregistrements marqués comme supprimés (soft delete) dans les résultats.



# TypeORM - Repository

- Exemple

```
const users = await userRepository.find({
  where: {
    city: 'Paris', // Filter users by city
  },
  relations: ['orders'], // Include relations with orders
  select: ['id', 'name', 'city'], // Select only the columns id, name, and city
  order: {
    name: 'ASC', // Sort by name in ascending order
  },
  skip: 10, // Skip the first 10 results
  take: 5, // Limit to 5 results
});
```



# TypeORM - Repository

- `save`  
Crée ou met à jour un enregistrement (fusionne avec les données existantes).
- `insert`  
Insère de nouveaux enregistrements directement sans vérifier s'ils existent déjà.
- `update`  
Met à jour des enregistrements existants en fonction de critères.
- `delete`  
Supprime des enregistrements correspondant à des critères.
- `softRemove`  
Marque des enregistrements comme supprimés sans les supprimer physiquement.
- `restore`  
Restaure des enregistrements précédemment marqués comme supprimés.
- `count`  
Compte le nombre d'enregistrements correspondant à des critères.
- `exist`  
Vérifie si un ou plusieurs enregistrements existent selon des critères. (disponible dans les versions récentes)



# TypeORM - Soft Delete

- ▶ Pour implémenter le soft delete, on utilise le décorateur DeleteDateColumn

```
@DeleteDateColumn()  
deletedAt: Date;
```

- ▶ Pour supprimer

```
const user = await userRepository.findOne({ where: { id: 1 }, withDeleted: true });  
await userRepository.softRemove(user);
```

- ▶ Pour restaurer

```
await userRepository.restore(userId);
```

- ▶ Pour récupérer les enregistrements en incluant les soft deletions :

```
await userRepository.find({  
    relations: ['orders'], // Charger les commandes des utilisateurs  
    withDeleted: true, // Inclure les utilisateurs supprimés  
});
```



# TypeORM - Associations

- Une association (ou relation) connecte deux entités dans une base de données.
- Types principaux d'associations dans TypeORM :
  - One-to-One (1:1)
  - One-to-Many / Many-to-One (1:N / N:1)
  - Many-to-Many (N:N)
- Utilisées pour modéliser les dépendances et interactions entre tables.



# TypeORM - Associations

- OneToOne
  - `@OneToOne()` définit la relation.
  - `@JoinColumn()` indique quelle colonne sera utilisée pour stocker la clé étrangère (obligatoire avec OneToOne)

```
import { Entity, PrimaryGeneratedColumn, Column, OneToOne, JoinColumn } from 'typeorm';

@Entity()
export class Profile {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    bio: string;
}

@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    name: string;

    @OneToOne(() => Profile)
    @JoinColumn()
    profile: Profile;
}
```

# TypeORM - Associations



- OneToMany / ManyToOne
  - OneToMany est optionnel
  - JoinColumn également

```
import { Entity, PrimaryGeneratedColumn, Column, OneToMany, ManyToOne } from 'typeorm';

@Entity()
export class Post {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    title: string;

    @ManyToOne(() => User, (user) => user.posts)
    user: User;
}

@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    name: string;

    @OneToMany(() => Post, (post) => post.user)
    posts: Post[];
}
```



# TypeORM - Associations

- ManyToMany
  - JoinTable est obligatoire (nommera la table de liens dans cet exemple product\_categories\_category si on ne lui passe pas de nom en param)

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToMany, JoinTable } from 'typeorm';

@Entity()
export class Category {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    name: string;

    @ManyToMany(() => Product, (product) => product.categories)
    products: Product[];
}

@Entity()
export class Product {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    title: string;

    @ManyToMany(() => Category, (category) => category.products)
    @JoinTable()
    categories: Category[];
}
```



# TypeORM - Associations

- Eager

Si on passe l'option eager: true, un appel à une méthode find\* fera systématiquement la jointure sans qu'on le demande

```
@OneToMany(() => Post, (post) => post.user, { eager: true })
posts: Post[];
```

- Cascade

Avec l'option cascade on indique que les update ou delete se propage automatiquement

<https://orkhan.gitbook.io/typeorm/docs/relations#cascade-options>



# TypeORM - Transactions

- Une transaction est une séquence d'opérations qui doit être exécutée dans son intégralité ou pas du tout (principe ACID).
- Objectif : garantir la cohérence des données, même en cas d'échec partiel.
- TypeORM supporte les transactions pour les bases de données relationnelles.
- 2 styles : EntityManager (plus simple) ou QueryRunner (plus fin)
- EntityManager :

```
@Injectable()
export class UserService {
    constructor(private readonly dataSource: DataSource) {}

    async createUserWithProfile() {
        return this.dataSource.transaction(async (manager) => {
            const user = manager.create(User, { name: 'John' });
            await manager.save(user);

            const profile = manager.create(Profile, { userId: user.id, bio: 'Bio example' });
            await manager.save(profile);
        });
    }
}
```



# TypeORM - Transactions

- QueryRunner :

```
import { Injectable } from '@nestjs/common';
import { DataSource } from 'typeorm';

@Injectable()
export class UserService {
    constructor(private readonly dataSource: DataSource) {}

    async createUserWithQueryRunner() {
        const queryRunner = this.dataSource.createQueryRunner();
        await queryRunner.connect();
        await queryRunner.startTransaction();

        try {
            const user = await queryRunner.manager.save(User, { name: 'John' });
            const profile = await queryRunner.manager.save(Profile, { userId: user.id, bio: 'Bio example' });

            await queryRunner.commitTransaction();
        } catch (error) {
            await queryRunner.rollbackTransaction();
        } finally {
            await queryRunner.release();
        }
    }
}
```



# TypeORM - Index

- QueryRunner :

```
@Index()
@Column()
firstName: string

@Entity()
@Index(["firstName", "lastName"])
@Index(["firstName", "middleName", "lastName"], { unique: true })
export class User {
}
    @Index({ unique: true })
    @Column()
    firstName: string
```



# TypeORM - Entity Listener

- Les Entity Listener permettent d'exécuter du code automatiquement avant ou après certaines opérations (insert, update...)
- Les décorateurs suivants sont disponibles : @AfterLoad, @BeforeInsert, @AfterInsert, @BeforeUpdate, @AfterUpdate, @BeforeRemove, @AfterRemove, @BeforeSoftRemove, @AfterSoftRemove, @BeforeRecover, @AfterRecover

```
@Entity()
export class Post {
    @BeforeUpdate()
    updateDates() {
        this.updatedDate = new Date();
    }
}
```



# TypeORM - Event Subscriber

- Comme les Entity Listener, les Event Subscriber seront appelés automatiquement.
- Ils peuvent cependant faire des requêtes SQL que ne peuvent pas les Entity Listener

```
@EventSubscriber()
export class PostSubscriber implements EntitySubscriberInterface<Post> {
    /**
     * Indicates that this subscriber only listen to Post events.
     */
    listenTo() {
        return Post
    }

    /**
     * Called before post insertion.
     */
    beforeInsert(event: InsertEvent<Post>) {
        console.log(`BEFORE POST INSERTED: `, event.entity)
    }
}
```

- On les enregistre avec la config du module :

```
@Module({
    imports: [TypeOrmModule.forFeature([Post, Comment], {subscribers: [PostSubscriber]})],
    controllers: [PostController],
})
export class BlogModule {}
```



**formation.tech**

# Validation



# Validation - Introduction

- NestJS propose l'intégration de class-validator pour la validation des DTO
- Permet valider les données de manière déclarative.
- Simplifie la validation des données.
- Réduction du code "boilerplate".
- Gestion centralisée des erreurs.
- Installation :  
`npm install class-validator class-transformer`



# Validation - Utilisation

- Exemple de DTO avec décorateurs de validation

```
export class CreateUserDto {  
    @IsString()  
    name: string;  
  
    @IsEmail()  
    email: string;  
  
    @IsInt()  
    @Min(18)  
    age: number;  
}
```

- Configuration de NestJS avec ValidationPipe

```
async function bootstrap() {  
    const app = await NestFactory.create(AppModule);  
    app.useGlobalPipes(new ValidationPipe({  
        whitelist: true,  
        forbidNonWhitelisted: true,  
        transform: true,  
    }));  
    await app.listen(process.env.PORT ?? 3000);  
}
```



# Validation - Utilisation

- Le contrôleur est inchangé

```
@Post('create')
async createUser(@Body() createUserDto: CreateUserDto) {
    return this.userService.create(createUserDto);
}
```

- Sauf si on souhaite passer un tableau :

```
createUsers(@Body(new ParseArrayPipe({ items: CreateUserDto }))) {
```

```
}
```

- La gestion des erreurs est automatique avec ValidationPipe, exemple :

```
{
  "statusCode": 400,
  "message": ["name must be a string", "age must be an integer"],
  "error": "Bad Request"
}
```



# Validation - Décorateurs liés aux types

- `@IsString()`  
Vérifie si la valeur est une chaîne de caractères.
- `@IsInt()`  
Vérifie si la valeur est un entier.
- `@IsBoolean()`  
Vérifie si la valeur est un booléen (true ou false).
- `@IsNumber()`  
Vérifie si la valeur est un nombre (peut inclure des décimales).
- `@IsDate()`  
Valide si la valeur est une date valide.
- `@IsArray()`  
Vérifie si la valeur est un tableau.



# Validation - Décorateurs liés aux types

- `@IsString()`  
Vérifie si la valeur est une chaîne de caractères.
- `@IsInt()`  
Vérifie si la valeur est un entier.
- `@IsBoolean()`  
Vérifie si la valeur est un booléen (true ou false).
- `@IsNumber()`  
Vérifie si la valeur est un nombre (peut inclure des décimales).
- `@IsDate()`  
Valide si la valeur est une date valide.
- `@IsArray()`  
Vérifie si la valeur est un tableau.



# Validation - Décorateurs de validation de format

- `@IsEmail()`  
Vérifie si la valeur est une adresse email valide.
- `@IsUrl()`  
Valide si la valeur est une URL valide.
- `@IsUUID(version?: '3' | '4' | '5')`  
Vérifie si la valeur est un UUID (par défaut version 4).
- `@Matches(regex: RegExp)`  
Vérifie si la valeur correspond à une expression régulière donnée.



# Validation - Décorateurs de validation de contrainte

- `@Min(value: number)`  
Vérifie si la valeur est supérieure ou égale à une limite minimale.
- `@Max(value: number)`  
Vérifie si la valeur est inférieure ou égale à une limite maximale.
- `@Length(min: number, max?: number)`  
Valide la longueur d'une chaîne de caractères (entre min et max).
- `@MinLength(min: number)`  
Vérifie que la chaîne a au moins min caractères.
- `@MaxLength(max: number)`  
Vérifie que la chaîne n'excède pas max caractères.



# Validation - Autres décorateurs courants

- `@IsOptional()`  
Rend la validation facultative pour ce champ si la valeur est null ou undefined.
- `@ValidateNested()`  
Valide les propriétés d'un objet imbriqué.
- `@IsNotEmpty()`  
Vérifie si la valeur n'est ni vide ni nulle.
- `@IsDefined()`  
Vérifie si la valeur est définie (non undefined).
- `@ValidateIf(condition: (object: any, value: any) => boolean)`  
Valide uniquement si une condition personnalisée est remplie.
- Voir la doc pour la liste complète :  
<https://github.com/typestack/class-validator?tab=readme-ov-file#validation-decorators>



# Validation - Personnaliser les messages

- Pour personnaliser les messages on passe une option au décorateur

```
@IsEmail({}, { message: 'L\'email doit être valide' })
email: string;
```



# Validation - Validation conditionnelle

- La validation peut dépendre d'une autre valeur de l'objet

```
@ValidateIf(o => o.isActive)  
@IsString()  
activeReason: string;
```

- Ou être lié à des variantes avec les groupes (create/update...)

```
@IsString({ groups: ['create'] })  
@IsOptional({ groups: ['update'] })  
name: string;
```

- Il faudra alors spécifier pour quel groupe on applique la validation

```
@Post()  
@UsePipes(new ValidationPipe({ groups: ["create"] }))  
async create(@Body() createUserDto: CreateUserDto): Promise<User> {  
  const result = await this.userService.create(createUserDto);  
  return result;  
}
```



# Validation - Validation imbriquée

- Par défaut la validation ne se fait que pour l'objet racine
- Pour valider les objets imbriqués on utilise ValidateNested :

```
import { Type } from 'class-transformer';

export class Address {
    @IsString()
    street: string;

    @IsString()
    city: string;
}

export class User {
    @ValidateNested()
    @Type(() => Address)
    address: Address;
}
```



# Validation - Options de ValidationPipe

- **transform**  
Convertit les données entrantes en instances des DTOs spécifiés. (default: false)
- **whitelist**  
Supprime automatiquement les propriétés non définies dans le DTO. (default: false)
- **forbidNonWhitelisted**  
Rejette les requêtes contenant des propriétés non autorisées (non définies dans le DTO). (default: false)
- **forbidUnknownValues**  
Bloque les valeurs qui ne correspondent pas à un DTO ou une structure validée. (default: true)
- **disableErrorMessages**  
Désactive l'affichage des messages d'erreur détaillés (idéal pour la production). (default: false)
- **validationError**  
Configure la structure des erreurs retournées (par défaut, exclut target et value). (default: { target: false, value: false })
- **transformOptions**  
Définit les options pour la transformation des données (via class-transformer). (default: {})
- **stopAtFirstError**  
Arrête la validation dès qu'une première erreur est rencontrée. (default: false)
- **exceptionFactory**  
Permet de personnaliser les erreurs de validation retournées. (default: crée une exception standard basée sur les erreurs)



# Validation - Créer son propre décorateur

- Un décorateur est une fonction qui retourne une fonction

```
import { registerDecorator, ValidationOptions, ValidationArguments } from 'class-validator';

export function IsFrenchPhoneNumber(validationOptions?: ValidationOptions) {
    return function (object: Object, propertyName: string) {
        registerDecorator({
            name: 'isFrenchPhoneNumber',
            target: object.constructor,
            propertyName: propertyName,
            options: validationOptions,
            validator: {
                validate(value: any) {
                    return /^+33[1-9]\d{8}$.test(value);
                },
            },
        });
    };
}
```



# Validation - Utilitaires

- @nestjs/mapped-types offre des types utilitaires pour dériver sa validation d'autres validations (privilégier @nestjs/swagger si Swagger ou @nestjs/graphql si GraphQL) :
- Pour rendre toutes les propriétés optionnelles

```
export class UpdateCatDto extends PartialType(CreateCatDto) {}
```

- Pour ne garder que certaines propriétés

```
export class UpdateCatAgeDto extends PickType(CreateCatDto, ['age'] as const) {}
```

- Pour retirer certaines propriétés

```
export class UpdateCatDto extends OmitType(CreateCatDto, ['name'] as const) {}
```

- Pour ne garder que les propriétés communes

```
export class UpdateCatDto extends IntersectionType(  
  CreateCatDto,  
  AdditionalCatInfo,  
) {}
```



**formation.tech**

# NoSQL / Mongoose

# NoSQL - Introduction



- NoSQL
  - Not Only SQL, le nom qu'on donne au mouvement depuis quelques années de ne pas tout stocker sous la forme de base de données relationnelles (MySQL, SQLite, PostgreSQL, Oracle, SQL Server...).  
A l'origine en 2009, le nom donné à un meetup à San Francisco.  
Parfois appelé « NoRel » (« not only relational ») pour ne pas prêter à confusion.
- Intérêts
  - Performance, scalabilité, haute-disponibilité
- Catégories
  - Clé / valeur (Redis / Memcached...)
  - Orienté Colonne (HBase / Cassandra...)
  - Orienté Document (MongoDB / CouchDB...)
  - Orienté Graphe (Neo4j)
  - ...

# NoSQL - Clé / valeur

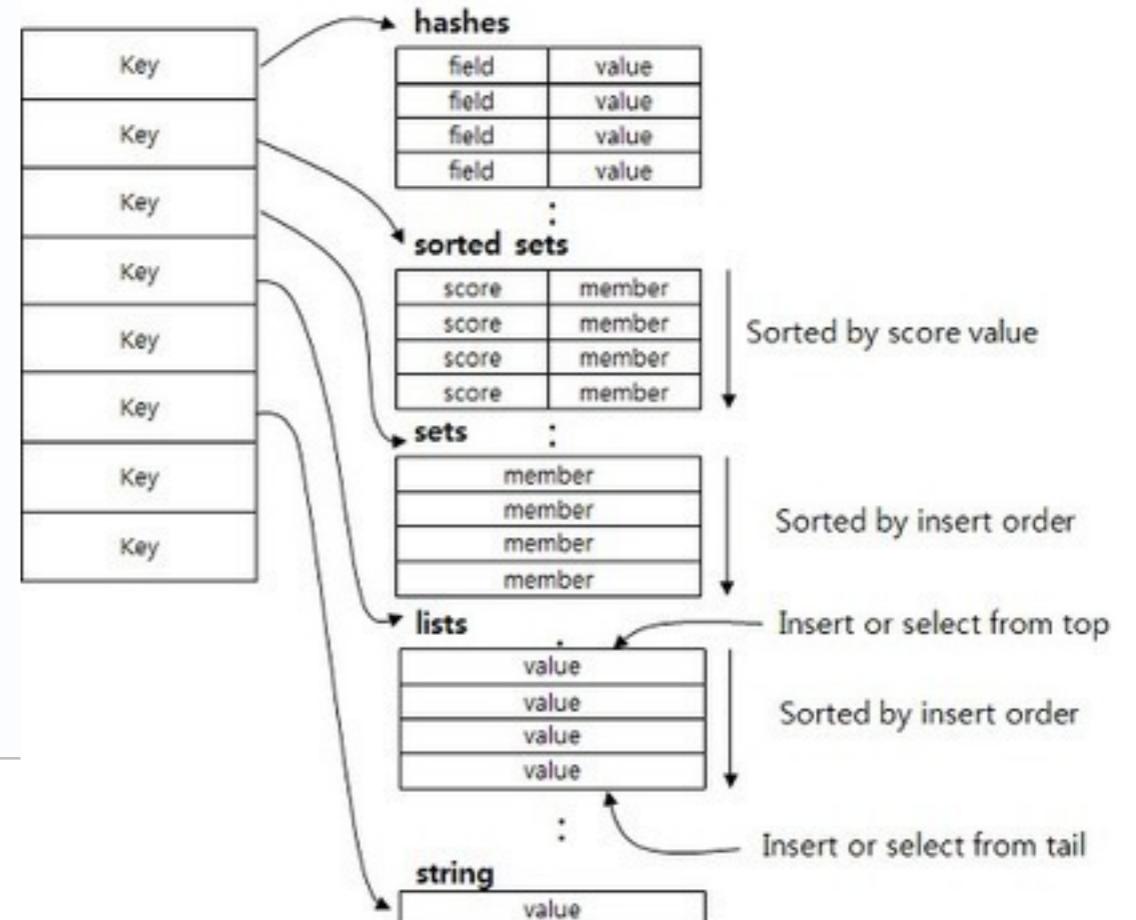


- Exemple Redis

```
const redis = require("redis");
const client = redis.createClient();

client.on("error", function(error) {
  console.error(error);
});

client.set("key", "value", redis.print);
client.get("key", redis.print);
```





# NoSQL - Orienté Colonne

- Exemple Cassandra

```
const cassandra = require('cassandra-driver');

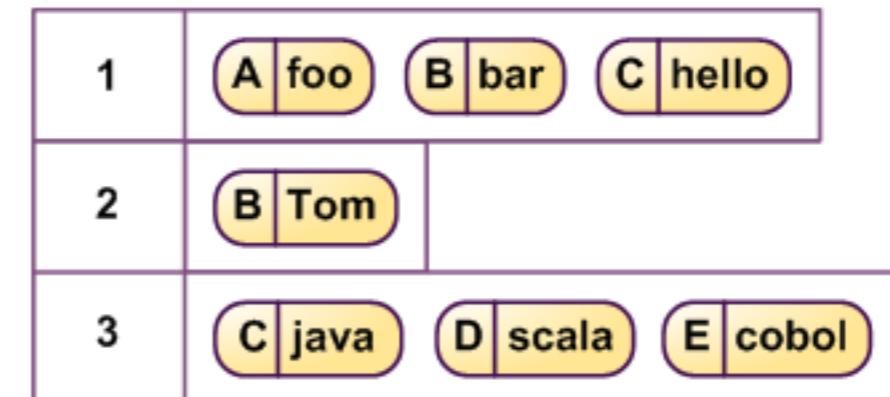
const client = new cassandra.Client({
  contactPoints: ['h1', 'h2'],
  localDataCenter: 'datacenter1',
  keyspace: 'ks1'
});

const query = 'SELECT name, email FROM users WHERE key = ?';

client.execute(query, [ 'someone' ])
  .then(result => console.log('User with email %s', result.rows[0].email));
```

	A	B	C	D	E
1	foo	bar	hello		
2		Tom			
3			java	scala	cobol

Organisation d'une table dans une BDD relationnelle



Organisation d'une table dans une BDD orientée colonnes



# NoSQL - Orienté Document

- Exemple CouchDB

```
const cradle = require('cradle');
const db = new(cradle.Connection)().database('starwars');

db.get('vader', function (err, doc) {
  doc.name; // 'Darth Vader'
  assert.equal(doc.force, 'dark');
});

db.save('skywalker', {
  force: 'light',
  name: 'Luke Skywalker'
}, function (err, res) {
  if (err) {
    // Handle error
  } else {
    // Handle success
  }
});
```

db.users.insert ( ← collection  
{  
 name: "sue", ← field: value  
 age: 26, ← field: value  
 status: "A" ← field: value  
}  
)

document

# NoSQL - Orienté Graphe



- Exemple neo4j

```
const neo4j = require('neo4j-driver')

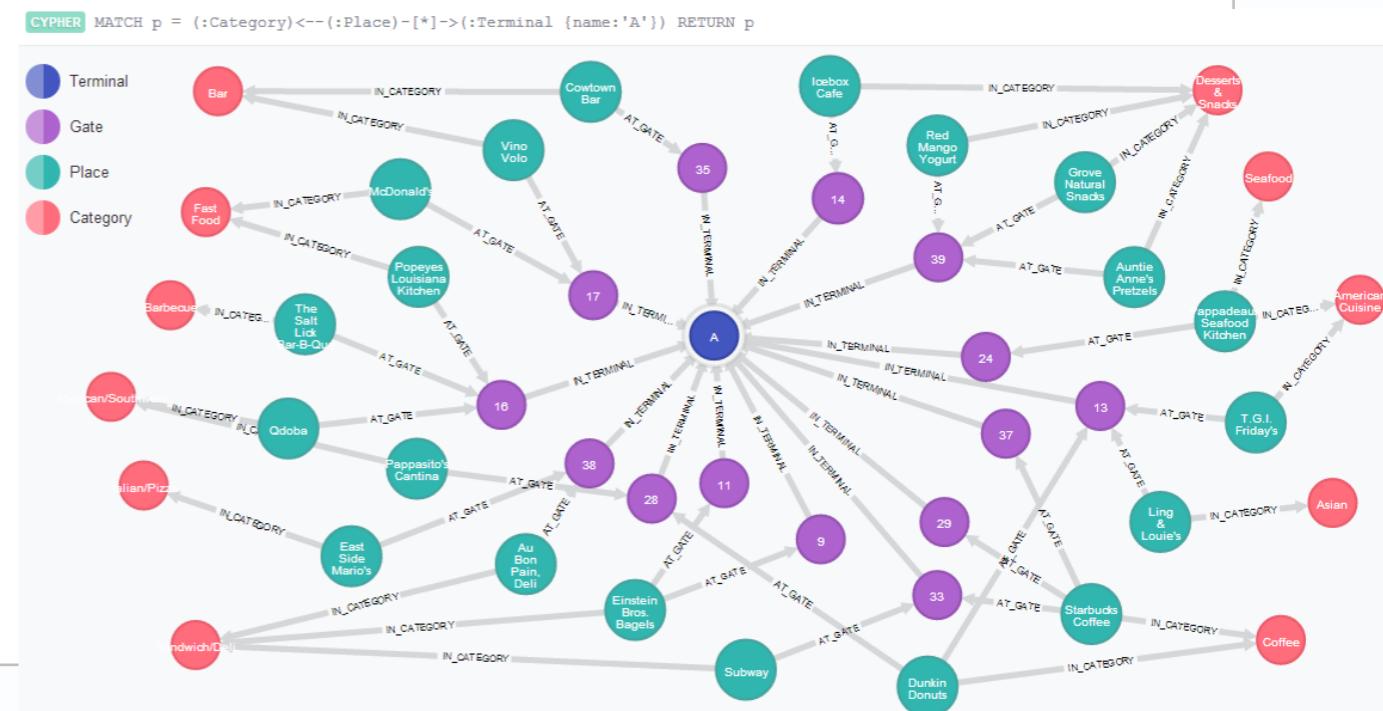
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password))
const session = driver.session()
const personName = 'Alice'

try {
  const result = await session.run(
    'CREATE (a:Person {name: $name}) RETURN a',
    { name: personName }
  )

  const singleRecord = result.records[0]
  const node = singleRecord.get(0)

  console.log(node.properties.name)
} finally {
  await session.close()
}

// on application exit:
await driver.close()
```



# NoSQL - MongoDB



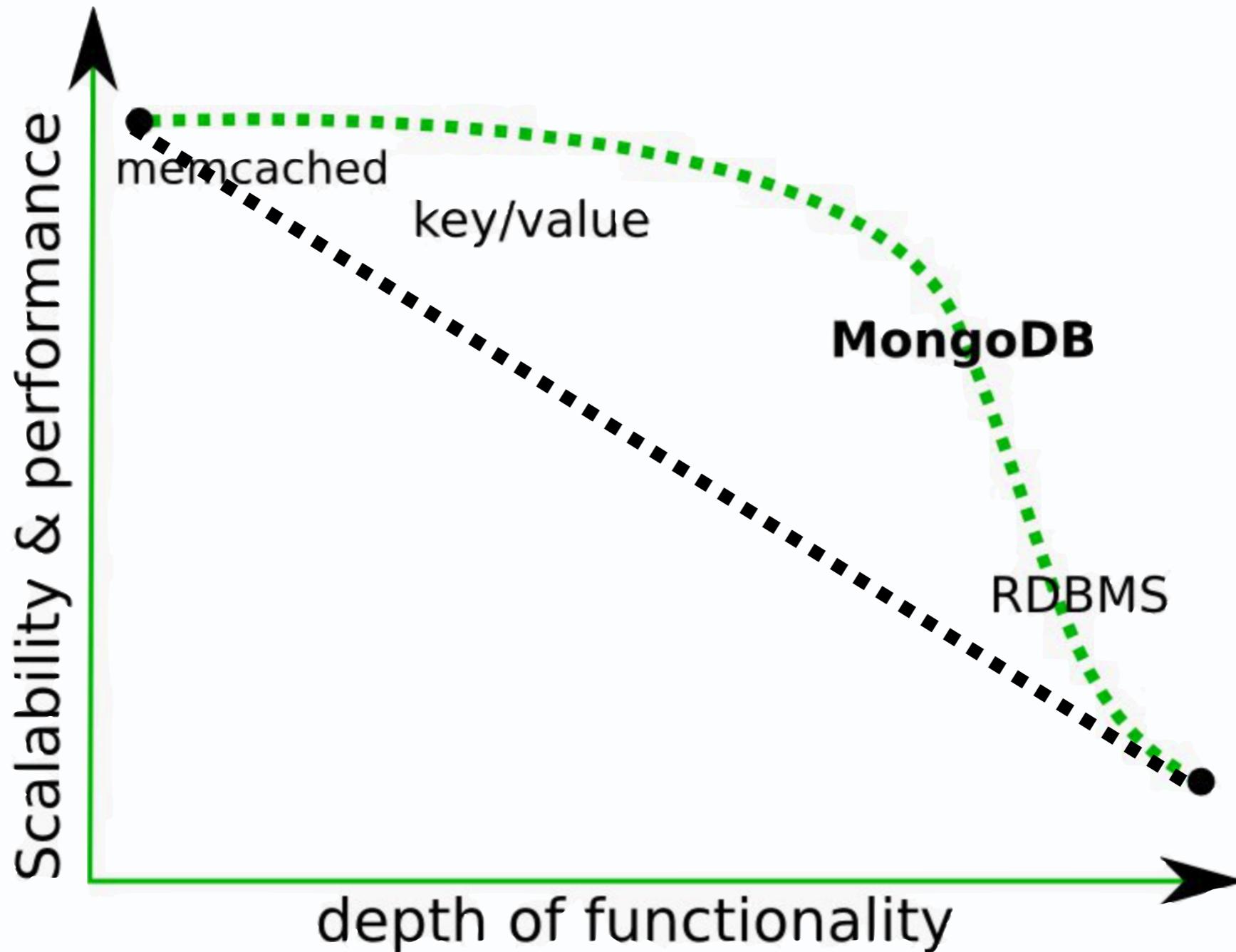
- MongoDB  
Base de données écrite en C++, inclus le moteur JS SpiderMonkey de Mozilla ou Node.js (donc V8)
- Document  
MongoDB permet de manipuler des objets structurés au format BSON (JSON binaire). Les données prennent la forme de documents enregistrés eux-mêmes dans des collections.
- Accès aux données  
L'accès aux données se fait via un protocole réseau, les requêtes sont décrites sous forme d'objet JavaScript
- Absence de Schéma  
Contrairement à un SGBDR, les documents stockés dans une collection peuvent avoir des formats complètement différents. Les données peuvent également être imbriquées.

# NoSQL - MongoDB



- Installation
  - Windows  
<https://www.mongodb.com/>  
<https://www.mongodb.org/dl/win32>
  - Mac  
<https://www.mongodb.com/>  
brew install mongo

# NoSQL - MongoDB



# NoSQL - MongoDB



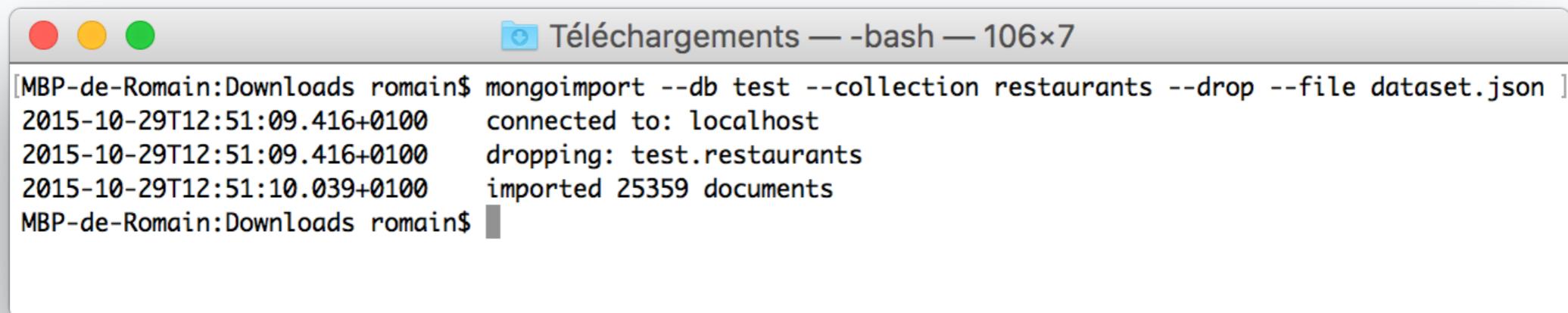
- Parallèle avec un SGBDR

MongoDB	SGBDR
Base de données	Base de données
Collection	Table
Document	Enregistrement
Pas de schéma	Schéma
API JavaScript (Objet JS)	SQL



# NoSQL - MongoDB

- › Jeu de données d'exemple fourni par Mongo  
<https://raw.githubusercontent.com/OpenKitten/Mongo-Assets/master/primer-dataset.json>
- › Importer un jeu de données (via Database Tools)  
<https://www.mongodb.com/try/download/database-tools>)  
mongoimport --collection restaurants --file ~/downloads/primer-dataset.json



```
[MBP-de-Romain:Downloads roman$ mongoimport --db test --collection restaurants --drop --file dataset.json ]
2015-10-29T12:51:09.416+0100      connected to: localhost
2015-10-29T12:51:09.416+0100      dropping: test.restaurants
2015-10-29T12:51:10.039+0100      imported 25359 documents
MBP-de-Romain:Downloads roman$
```

# NoSQL - MongoDB



- **MongoShell**

Mongo livre un programme client en ligne de commande pour accéder à la base.

```
[MBP-de-Romain:~ romain$ mongo
MongoDB shell version: 3.0.7
connecting to: test
[> use address_book
switched to db address_book
[> db.contact.find()
{ "_id" : ObjectId("562d4e878561c01ec2e43cfb"), "prenom" : "Steve", "nom" : "Jobs" }
{ "_id" : ObjectId("562d4e918561c01ec2e43cfc"), "prenom" : "Bill", "nom" : "Gates" }
{ "_id" : ObjectId("562d4eab8561c01ec2e43cfb"), "prenom" : "Mark", "nom" : "Zuckerberg" }
[> db.contact.insert({prenom: 'Steve', nom: 'Ballmer'})
WriteResult({ "nInserted" : 1 })
[> db.contact.find({prenom: 'Steve'})
{ "_id" : ObjectId("562d4e878561c01ec2e43cfb"), "prenom" : "Steve", "nom" : "Jobs" }
{ "_id" : ObjectId("562d4ee1321ac0f47f03ce9d"), "prenom" : "Steve", "nom" : "Ballmer" }
> ]]
```

# NoSQL - MongoDB



- Principales Commandes MongoShell

Shell Helpers	JavaScript Equivalents
show dbs, show databases	db.adminCommand('listDatabases')
use <db>	db = db.getSiblingDB('<db>')
show collections	db.getCollectionNames()
show users	db.getUsers()
show roles	db.getRoles({showBuiltInRoles: true})
show log <logname>	db.adminCommand({ 'getLog' : '<logname>' })
show logs	db.adminCommand({ 'getLog' : '*' })
it	cursor = db.collection.find() if ( cursor.hasNext() ){ cursor.next(); }

# NoSQL - MongoDB



- MongoClient
  - API officiel fourni MongoDB pour accéder aux données sous Node.js
- Installation
  - npm install mongodb --save
- Insertion

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost:27017/addressbook';

MongoClient.connect(url, function(err, db) {
  if (err) {
    console.log('Erreur : ' + err);
    return;
  }
  var cursor = db.collection('contacts').insert({prenom: 'Romain', nom: 'Bohdanowicz'},
function(err, result) {
  if (err) {
    console.log('Erreur : ' + err);
    return;
  }

  console.log('Le contact a bien été inséré');
});
});
```

# NoSQL - MongoDB



## ▶ Modification

```
var cursor = db.collection('contacts').update({nom: 'Bohdanowicz'}, {prenom: 'ROMAIN', nom: 'BOHDANOWICZ'}, {upsert:true}, function(err, result) {
  if (err) {
    console.log('Erreur : ' + err);
    return;
  }

  console.log('Le contact a bien été mis à jour');
});
```

## ▶ Suppression

```
var cursor = db.collection('contacts').removeOne({nom: 'BOHDANOWICZ'}, function(err, result) {
  if (err) {
    console.log('Erreur : ' + err);
    return;
  }

  console.log('Le contact a bien été supprimé');
});
```

# NoSQL - MongoDB



- Recherche

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost:27017/addressbook';

MongoClient.connect(url, function(err, db) {
  if (err) {
    console.log('Erreur : ' + err);
    return;
  }
  var cursor = db.collection('contacts').find();
  cursor.toArray(function(err, contacts) {
    console.log(contacts);
    db.close();
  });
});
```

# NoSQL - MongoDB



- Recherche multi-critères

Exemple : Restaurants de Brooklyn, ET dont la cuisine est française OU italienne ET dont l'une des notes est supérieur à 40

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost:27017/test';
MongoClient.connect(url, function(err, db) {
  if (err) {
    console.log('Erreur : ' + err);
    return;
  }
  var cursor = db.collection('restaurants').find({
    borough: 'Brooklyn',
    $or: [
      { "cuisine": "Italian" },
      { "cuisine": "French" }
    ],
    'grades.score': { $gt: 40 }
  });
  cursor.toArray(function(err, restaurants) {
    restaurants.forEach(function(r) {
      console.log(`Nom : ${r.name}, cuisine : ${r.cuisine}, adresse : ${r.address.building} ${r.address.street}`);
    });
    db.close();
  });
});
```

```
MBP-de-Romain:MongoClient romain$ node multicriteres.js
Nom : Doc Wine Bar, cuisine : Italian, adresse : 83 North 7 Street
Nom : Le Gamin, cuisine : French, adresse : 556 Vanderbilt Avenue
Nom : Peperoncino, cuisine : Italian, adresse : 72 5 Avenue
Nom : Patrizia'S, cuisine : Italian, adresse : 35 Broadway
Nom : Tutta Pasta, cuisine : Italian, adresse : 160 7 Avenue
Nom : Anella, cuisine : Italian, adresse : 222 Franklin Street
Nom : Joe'S Pizza, cuisine : Italian, adresse : 349 5 Avenue
MBP-de-Romain:MongoClient romain$
```

# NoSQL - MongoDB



- Mongoose
  - ODM : Object Document Mapping, permet de communiquer avec Mongo avec des objets Entités
- Installation
  - `npm install mongoose --save`
- Schema
  - Mongo permet l'absence de schéma, ce qui est peu recommandable dans une utilisation sous la forme d'entité. Mongoose réintroduit ce concept.

# NoSQL - MongoDB



- Création d'un Schéma

```
var mongoose = require('mongoose');

var contactSchema = mongoose.Schema({
  firstName: String,
  lastName: String,
})

var Contact = mongoose.model('contact', contactSchema);
```

```
mongoose.connect('mongodb://localhost/addressbook');
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function (callback) {
  var contacts = Contact.find(function (err, contacts) {
    if (err) return console.error(err);
    reply({data: contacts});
  });
});
```



**formation.tech**

# Middlewares

# Middlewares - Introduction



- Qu'est-ce qu'un middleware ?
  - Une fonction exécutée avant qu'une requête atteigne le gestionnaire (handler).
  - Utilisé pour transformer, analyser ou valider les requêtes.
- Exemples d'utilisation :
  - Journalisation (logging)
  - Modification de la requête ou réponse (ex: en-têtes)

# Middlewares - Introduction



- Middleware dans NestJS :
  - Implémenté comme une classe, une fonction ou une méthode.
  - Compatible avec le système modulaire de NestJS.
- Deux types principaux :
  - Middleware global : appliqué à toutes les routes.
  - Middleware par module ou par route.
- Pipeline d'exécution :
  - Requête → Middleware → Guard → Interceptor → Pipe → Contrôleur.



# Middlewares - Exemple

- Exemple
- ```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log('Request...');
    next();
  }
}
```



# Middlewares - Enregistrement

- Enregistrement

```
import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';
import { LoggerMiddleware } from './logger.middleware';

@Module({
  imports: [],
  controllers: [],
  providers: [],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware) // Ajouter le middleware
      .forRoutes('/example/*'); // Cible certaines routes
  }
}
```

- Globalement

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { LoggerMiddleware } from './logger.middleware';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.use(LoggerMiddleware); // Middleware global
  await app.listen(3000);
}
bootstrap();
```



# Middlewares - Enregistrement

- Enregistrement

```
import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';
import { LoggerMiddleware } from './logger.middleware';

@Module({
  imports: [],
  controllers: [],
  providers: [],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware) // Ajouter le middleware
      .forRoutes('/example/*'); // Cible certaines routes
  }
}
```

- Globalement

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { LoggerMiddleware } from './logger.middleware';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.use(LoggerMiddleware); // Middleware global
  await app.listen(3000);
}
bootstrap();
```



**formation.tech**

# Exception Filters

# Exception Filters - Introduction



- Qu'est-ce qu'un Exception Filter ?
  - Un mécanisme pour intercepter et gérer les exceptions dans une application NestJS.
  - Permet de personnaliser la manière dont les erreurs sont retournées aux clients.
- Pourquoi les utiliser ?
  - Centraliser la gestion des erreurs.
  - Fournir des réponses uniformes.
  - Sécuriser les messages d'erreur.

# Exception Filters - Introduction



- Exceptions prises en charge nativement :
  - NestJS utilise des exceptions HTTP standard (ex : `HttpException`).
- Structure de base des réponses :
  - Code d'état HTTP (ex : 404, 500).
  - Message d'erreur.
- Pipeline d'erreur :
  - Requête → Guard → Interceptor → Handler → Exception Filter.



# Exception Filters - Exemple

- Exemple
- ```
import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from '@nestjs/common';
import { Request, Response } from 'express';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
    const status = exception.getStatus();

    response.status(status).json({
      statusCode: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      message: exception.message,
    });
  }
}
```



# Exception Filters - Enregistrement

- › Au niveau du contrôleur

```
import { Controller, Get, UseFilters } from '@nestjs/common';
import { HttpExceptionFilter } from './http-exception.filter';

@Controller('example')
@UseFilters(HttpExceptionFilter)
export class ExampleController {
  @Get()
  throwError() {
    throw new HttpException('Forbidden', 403);
  }
}
```

- › Au niveau de l'app

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { HttpExceptionFilter } from './http-exception.filter';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new HttpExceptionFilter());
  await app.listen(3000);
}
bootstrap();
```



**formation.tech**

# Pipes

# Pipes - Introduction



- Qu'est-ce qu'un Pipe ?
  - Une classe utilisée pour transformer ou valider les données d'une requête avant qu'elles n'atteignent le gestionnaire (handler).
  - Implémente l'interface PipeTransform.
- Rôle principal :
  - Validation : Vérifier que les données sont conformes.
  - Transformation : Convertir les données dans un format attendu.
- Exemples d'utilisation :
  - Valider les paramètres d'une URL.
  - Transformer des données en objets DTO.



# Pipes - Types de pipes

- Pipes intégrés :
  - ValidationPipe : Valide les données contre un schéma DTO.
  - ParseIntPipe : Convertit un paramètre en entier.
  - ParseBoolPipe : Convertit un paramètre en booléen.
  - DefaultValuePipe : Fournit une valeur par défaut.
- Pipes personnalisés :
  - Crées pour des besoins spécifiques non couverts par les pipes intégrés.



# Pipes - Enregistrement

- › Au niveau du param ou body

```
import { Controller, Post, Body } from '@nestjs/common';
import { IsString, IsInt } from 'class-validator';

class CreateUserDto {
  @IsString()
  name: string;

  @IsInt()
  age: number;
}

@Controller('users')
export class UsersController {
  @Post()
  create(@Body(new ValidationPipe()) createUserDto: CreateUserDto) {
    return createUserDto;
  }
}
```



# Pipes - Enregistrement

- Globalement

```
import { ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({ whitelist: true }));
  await app.listen(3000);
}
bootstrap();
```

# Pipes - Custom Pipe



- › Création

```
import { PipeTransform, Injectable, BadRequestException } from '@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform {
  transform(value: string) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed: not a number');
    }
    return val;
  }
}
```

- › Utilisation

```
import { Controller, Get, Param } from '@nestjs/common';
import { ParseIntPipe } from './parse-int.pipe';

@Controller('items')
export class ItemsController {
  @Get(':id')
  findOne(@Param('id', new ParseIntPipe()) id: number) {
    return { id };
  }
}
```



**formation.tech**

# Guards



# Guards - Introduction

- Qu'est-ce qu'un Guard ?
  - Une classe qui détermine si une requête sera traitée par le route handler.
  - Utilisé pour la logique d'autorisation.
- Pipeline d'exécution :
  - Requête → Middleware → Guard → Interceptor → Pipe → Handler.
- Exemples d'utilisation :
  - Vérifier si un utilisateur est authentifié.
  - Contrôler les permissions d'accès à une ressource.



# Guards - Exemple

- Exemple

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();
    return request.headers.authorization === 'valid-token';
  }
}
```

- CanActivate doit retourner un booléen, Promise<Boolean> ou Observable<boolean>



# Guards - Enregistrement

- › Au niveau d'une route (ou d'un contrôleur)

```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { AuthGuard } from './auth.guard';

@Controller('protected')
export class ProtectedController {
  @Get()
  @UseGuards(AuthGuard)
  findAll() {
    return { message: 'Authorized access' };
  }
}
```

- › Globalement

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { AuthGuard } from './auth.guard';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalGuards(new AuthGuard());
  await app.listen(3000);
}
bootstrap();
```



**formation.tech**

# Interceptors

# Interceptors - Introduction



- Qu'est-ce qu'un Interceptor ?
  - Une classe qui va appeler le route handle et peut exécuter du code avant et après
  - Peut modifier la réponse
- Exemples d'utilisation :
  - Mise en cache
  - Benchmark
  - Réponse HATEOAS
  - Transformation des exceptions en code HTTP



# Interceptors - Exemples

- Logging

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    console.log('Avant le traitement de la requête');
    const now = Date.now();
    return next.handle().pipe(
      tap(() => console.log(`Après le traitement: ${Date.now() - now}ms`))
    );
  }
}
```



# Interceptors - Exemples

- Transformer la réponse

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable()
export class TransformInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next.handle().pipe(
      map(data => ({
        status: 'success',
        data,
      }))
    );
  }
}
```



# Interceptors - Exemples

- Transformer les erreurs

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler, BadGatewayException }  
from '@nestjs/common';  
import { Observable, throwError } from 'rxjs';  
import { catchError } from 'rxjs/operators';  
  
@Injectable()  
export class ErrorsInterceptor implements NestInterceptor {  
    intercept(context: ExecutionContext, next: CallHandler): Observable<any> {  
        return next.handle().pipe(  
            catchError(err => {  
                console.error('Erreur capturée:', err);  
                return throwError(() => new BadGatewayException('Une erreur est survenue'));  
            })  
        );  
    }  
}
```



# Interceptors - Enregistrement

- › Au niveau d'une route

```
import { Controller, Get, UseInterceptors } from '@nestjs/common';
import { LoggingInterceptor } from './logging.interceptor';

@Controller('example')
@UseInterceptors(LoggingInterceptor)
export class ExampleController {
  @Get()
  findAll() {
    return { message: 'Données retournées' };
  }
}
```

- › Globalement

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { LoggingInterceptor } from './logging.interceptor';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalInterceptors(new LoggingInterceptor());
  await app.listen(3000);
}
bootstrap();
```



**formation.tech**

# Swagger

# Swagger - Introduction



- Swagger (OpenAPI) : Un outil pour documenter et tester les APIs REST.
- Pourquoi utiliser Swagger avec NestJS ?
  - Documentation interactive pour vos APIs.
  - Simplifie les tests et la validation des endpoints.
  - Améliore la collaboration entre développeurs.
- Installation  
`npm install @nestjs/swagger`



# Swagger - Configuration

```
import { NestFactory } from '@nestjs/core';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  const config = new DocumentBuilder()
    .setTitle('Cats example')
    .setDescription('The cats API description')
    .setVersion('1.0')
    .addTag('cats')
    .build();
  const documentFactory = () => SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, documentFactory);

  await app.listen(process.env.PORT ?? 3000);
}
bootstrap();
```

- Dans cet exemple la documentation devient disponible sur <http://localhost:3000/api>



# Swagger - UI

swagger

## Cats example 1.0

The cats API description

Schemes

HTTP ▾

---

**cats** ▾

---

**default** ▾

POST /cats

Parameters Try it out

Name	Description
------	-------------



# Swagger - Description

- SwaggerModule va automatiquement documenter vos routes et les décorateurs @Body, @Param et @Query
- Pour rendre les propriétés visibles sur nos DTOs on utilise @ApiProperty (possible de passer des options) :

```
import { ApiProperty } from '@nestjs/swagger';

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

- Si on documente un tableau :

```
@ApiProperty({ type: [String] })
names: string[];
```



# Swagger - Description

- On peut enrichir la documentation des routes avec : @ApiTags, @ApiOperation, @ApiBody, @ApiParam @ApiResponse

```
@ApiBearerAuth('JWT-auth')
@ApiTags('Book')
@UseGuards(RolesGuard)
@Roles(RoleEnum.User)
@Controller('book')
export class BookController {

    @ApiOperation({ summary: 'Print Book' })
    @ApiBody({ type: PrintOrderDto })
    @ApiParam({ name: 'bookId', type: 'string' })
    @ApiResponse({ status: 200, description: 'OK' })
    @Post(':bookId/print-order')
    async createPdf(@Param('bookId') bookId: string, @Body() printOrderDto: PrintOrderDto, @Req() req: Request) {
        const user = req.user;

        // Regenerate the access token from the refresh token so it will be valid for the next hour
        const { accessToken } = await this.authService.refreshAccessToken(printOrderDto.refreshToken);

        await this.bookService.handlePrintOrder(user, bookId, printOrderDto, accessToken);
    }
}
```



**formation.tech**

# Sécurité

# Sécurité - Introduction



- › Dans ce chapitre nous allons voir comment NestJS nous aide à mettre en place
  - L'authentification
  - Les autorisations
  - Des gardes fous contre les principales failles de sécurités

# Sécurité - Authentification



- Le cas le plus courant est une authentification avec des token JWT
- Les tokens JWT contiennent une signature qui évite de les stocker dans la DB, il faut à la place garantir la confidentialité d'une clé secrète (avec Nest/Config par exemple)
- Debugger sur <https://jwt.io/>

## Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

## Decoded

EDIT THE PAYLOAD AND SECRET

### HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

### PAYOUT: DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

### VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + " " +
```



# Sécurité - Exemple

```
JwtModule.register({
  global: true,
  secret: 'THIS KEY IS VERY SECRET AND SHOULD BE SET IN ENVIRONMENT VARIABLE',
  signOptions: { expiresIn: '60s' },
}),

import { Injectable, UnauthorizedException } from '@nestjs/common';
import { UsersService } from '../users/users.service';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthService {
  constructor(
    private usersService: UsersService,
    private jwtService: JwtService
  ) {}

  async signIn(
    username: string,
    pass: string,
  ): Promise<{ access_token: string }> {
    const user = await this.usersService.findOne(username);
    if (user?.password !== pass) {
      throw new UnauthorizedException();
    }
    const payload = { sub: user.userId, username: user.username };
    return {
      access_token: await this.jwtService.signAsync(payload),
    };
  }
}
```



# Sécurité - Exemple

```
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);
    if (!token) {
      throw new UnauthorizedException();
    }
    try {
      const payload = await this.jwtService.verifyAsync(
        token,
        {
          secret: jwtConstants.secret
        }
      );
      // 💡 We're assigning the payload to the request object here
      // so that we can access it in our route handlers
      request['user'] = payload;
    } catch {
      throw new UnauthorizedException();
    }
    return true;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : undefined;
  }
}
```



# Sécurité - Exemple

- › On protège ensuite nos routes avec le Guard

```
@UseGuards(AuthGuard)
@Get('profile')
getProfile(@Request() req) {
  return req.user;
}
```

- › On peut également faire l'inverse, sécuriser globalement toutes les routes, et en rendre certaines publiques :

<https://docs.nestjs.com/security/authentication#enable-authentication-globally>

# Sécurité - Autorisations



- Pour gérer les autorisation avec des roles, on peut se créer un décorateur custom

```
import { SetMetadata } from '@nestjs/common';
import { Role } from '../enums/role.enum';

export const ROLES_KEY = 'roles';
export const Roles = (...roles: Role[]) => SetMetadata(ROLES_KEY, roles);

@Post()
@Roles(Role.Admin)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

- Avec un guard :

```
canActivate(context: ExecutionContext): boolean {
  const requiredRoles = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
    context.getHandler(),
    context.getClass(),
  ]);
  if (!requiredRoles) {
    return true;
  }
  const { user } = context.switchToHttp().getRequest();
  return requiredRoles.some((role) => user.roles?.includes(role));
}
```

# Sécurité - Passeport



- Passport : Middleware d'authentification pour Node.js, avec plus de 500 stratégies disponibles (JWT, OAuth, etc.).
- But : Intégrer facilement des mécanismes d'authentification dans NestJS.
- Sécurité : Gère les authentifications complexes de manière standardisée.
- Simplicité : Basé sur des stratégies prêtes à l'emploi (local, JWT, OAuth).
- Extensibilité : Compatible avec les modules NestJS et facile à personnaliser.
- Exemple : Implémentation de JWT avec Passport.  
`npm install --save @nestjs/passport passport passport-local passport-jwt`
- Installation :



# Sécurité - Exemple

```
import { Strategy } from 'passport-local';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
    constructor(private authService: AuthService) {
        super(); // Par défaut, utilise "username" et "password"
    }

    async validate(username: string, password: string): Promise<any> {
        const user = await this.authService.validateUser(username, password);
        if (!user) {
            throw new UnauthorizedException();
        }
        return user; // Attache l'utilisateur à la requête
    }
}
```

# Sécurité - Failles



- Injection SQL

En passant par des ORMs comme TypeORM ou Prisma, nos requêtes seront préparées

```
const user = await this.userRepository.findOne({ where: { email } });
```

- CSRF

Pour se prémunir des Cross Site Request Forgeries (CSRF) NestJS passe par les middlewares Express, par exemple :

```
import { doubleCsrf } from 'csrf-csrf';
// ...
// somewhere in your initialization file
const {
  invalidCsrfTokenError, // This is provided purely for convenience if you plan on creating your own middleware.
  generateToken, // Use this in your routes to generate and provide a CSRF hash, along with a token cookie and token.
  validateRequest, // Also a convenience if you plan on making your own middleware.
  doubleCsrfProtection, // This is the default CSRF protection middleware.
} = doubleCsrf(doubleCsrfOptions);
app.use(doubleCsrfProtection);
```

# Sécurité - Failles



- CORS

CORS est intégré à NestJS mais désactivé par défaut (peut s'activer avec les options de `NestFactory.create` également)

```
const app = await NestFactory.create(AppModule);
app.enableCors();
await app.listen(process.env.PORT ?? 3000);
```

- Helmet

Helmet est un middleware pour Node.js/Express qui sécurise les applications web en configurant des en-têtes HTTP pour protéger contre des vulnérabilités comme les injections, le clickjacking et le vol de données sensibles.

```
import helmet from 'helmet';
// somewhere in your initialization file
app.use(helmet());
```



**formation.tech**

# Tests Automatisés



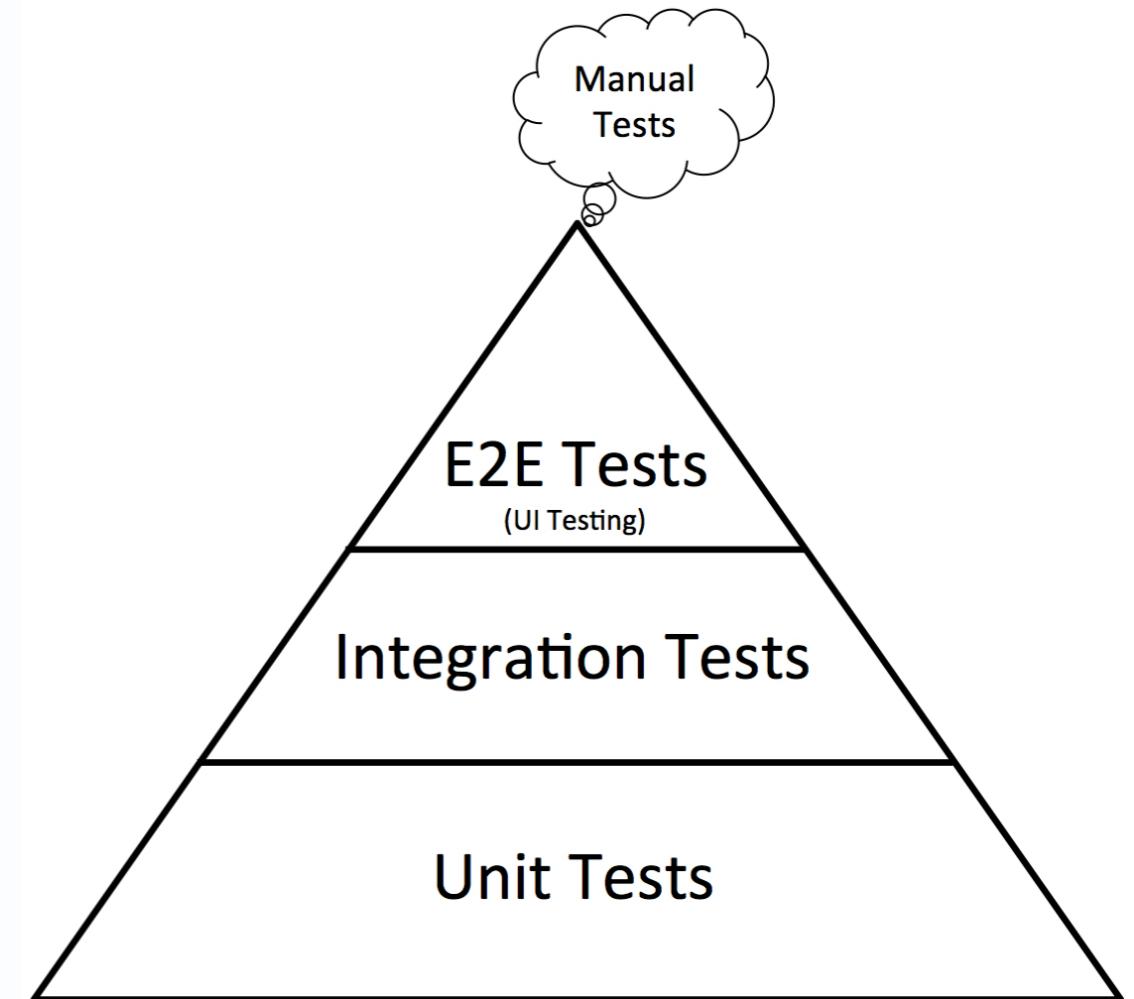
# Tests automatisés - Introduction

- Vérification manuelle
  - Ecrire une recette de tests et demander à une personne de la rejouer à des étapes clés (nouvelle version)
  - Ecrire le test sous la forme de code, et vérifier visuellement que les résultats attendus soit les bons
- Tests automatisés
  - Le test est codé, la vérification se fait dans un rapport
- Historique
  - sUnit en 1994 (SmallTalk), JUnit en 1997 (Java)
  - Les frameworks s'inspirant de jUnit sont catégorisés xUnit (PHPUnit, CUnit...)



# Tests automatisés - Pyramide des Tests

- Types de tests
  - **Unitaire** : tests des méthodes d'une classe
  - **Intégration** : teste l'intégration entre plusieurs classes
  - **Fonctionnels** : teste l'application du point de vue du client (HTTP dans le cas du web)
  - **End-to-End (E2E)** : teste l'application dans le client (y compris JavaScript, CSS...)





# Tests automatisés - Karma



- Lanceur de test  
Permet de lancer vos tests simultanément dans Chrome, Firefox, Internet Explorer...
- Installation  
npm install -g karma-cli  
npm install karma —save-dev
- Configuration du projet  
karma init
- Lancement des tests  
karma start

```
Air-de-Romain:Jasmine romain$ karma init

Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no

Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
> Safari
>

What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
> |
```

```
Air-de-Romain:Jasmine romain$ karma start
02 09 2015 21:30:11.510:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
02 09 2015 21:30:11.518:INFO [launcher]: Starting browser Chrome
02 09 2015 21:30:11.526:INFO [launcher]: Starting browser Safari
02 09 2015 21:30:12.723:INFO [Safari 8.0.7 (Mac OS X 10.10.4)]: Connected on socket HE38s1HTBKXL5t5yAAAA with id 54715269
Safari 8.0.7 (Mac OS X 10.10.4): Executed 1 of 1 SUCCESS (0.038 secs / 0.003 secs)
Safari 8.0.7 (Mac OS X 10.10.4): Executed 1 of 1 SUCCESS (0.038 secs / 0.003 secs)
Chrome 45.0.2454 (Mac OS X 10.10.4): Executed 1 of 1 SUCCESS (0.04 secs / 0.008 secs)
TOTAL: 2 SUCCESS
```



# Tests automatisés - QUnit

- Crée en 2008 par les développeurs de jQuery
- Type xUnit (JUnit, PHPUnit...) : basés sur des assertions
- Plutôt destiné à du code client
- Installation
  - npm install --save-dev qunitjs
  - bower install --save-dev qunit
- Lancement des tests
  - Ouverture du fichier .html
  - grunt-contrib-qunit
  - karma-qunit





# Tests automatisés - QUnit

```
<!-- runner.html -->
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit Example</title>
  <link rel="stylesheet" href="node_modules/qunitjs/qunit/qunit.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="calculette.js"></script>
  <script src="node_modules/qunitjs/qunit/qunit.js"></script>
  <script src="calculette-test.js"></script>
</body>
</html>
```

```
// calculette-test.js
QUnit.test("Test addition", function(assert) {
  assert.equal(calculette.ajouter(2, 3), 5, "2 + 3 = 5");
});
```

The screenshot shows a web browser window titled "QUnit Example". The address bar indicates the URL is "localhost:63342/JavaScript/Tests/QUnit/testCalculette.html". The main content area displays the QUnit test results:

- QUnit 1.19.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.85 Safari/537.36**
- Tests completed in 7 milliseconds.
- 1 assertions of 1 passed, 0 failed.
- 1. Test addition (1) Rerun**
- A single test result is shown: **1. 2 + 3 = 5** (status: **1 ms**)
- Source: at <http://localhost:63342/JavaScript/Tests/QUnit/testCalculette.html:14:11>

# Tests automatisés - Jasmine



- Crée en 2010
- Type BDD (Behavior-Driven Development)
- Fonctionne pour le browser ou node.js
- Installation et lancement des tests (node)  
`npm install -g jasmine`  
`jasmine init`  
`jasmine`
- Installation et lancement des tests (browser)  
`npm install --save-dev jasmine-core`  
`SpecRunner.html`  
`karma`





# Tests automatisés - Jasmine

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner v2.3.4</title>

  <link rel="shortcut icon" type="image/png" href="node_modules/jasmine-core/images/jasmine_favicon.png">
  <link rel="stylesheet" href="node_modules/jasmine-core/lib/jasmine-core/jasmine.css">

  <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></script>
  <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine-html.js"></script>
  <script src="node_modules/jasmine-core/lib/jasmine-core/boot.js"></script>

  <!-- include source files here... -->
  <script src="calculette.js"></script>

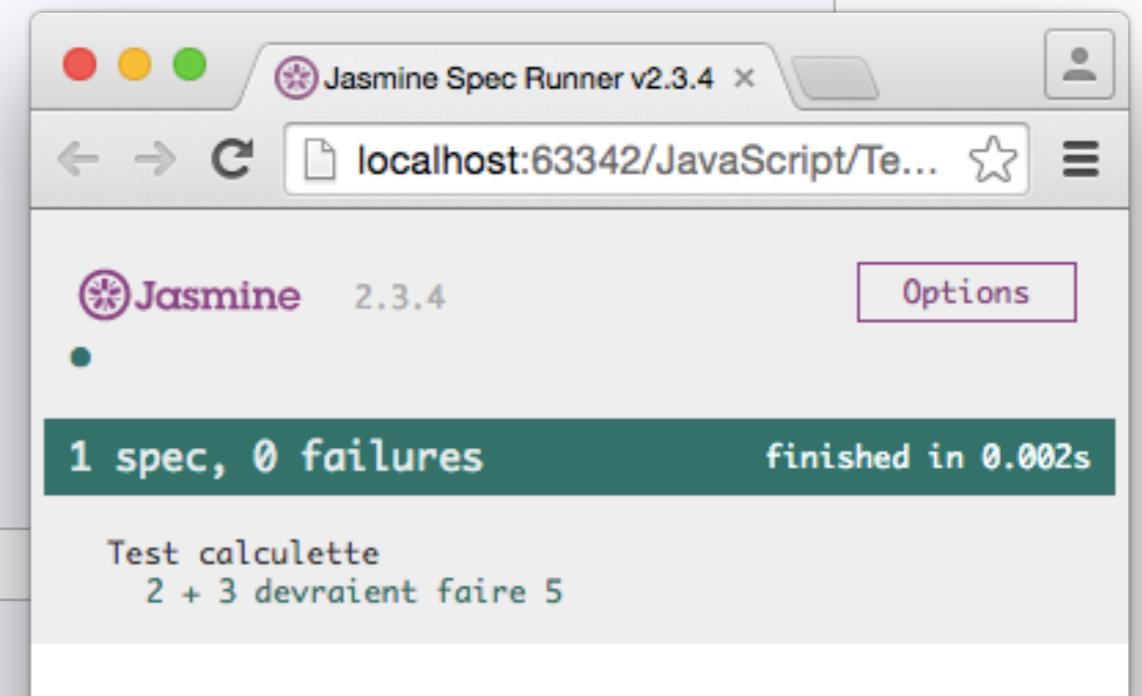
  <!-- include spec files here... -->
  <script src="spec/CalculetteSpec.js"></script>
</head>

<body>
</body>
</html>
```

```
describe("Test calculette", function() {

  it("2 + 3 devraient faire 5", function() {
    expect(calculette.ajouter(2, 3)).toEqual(5);
  });

});
```



# Tests automatisés - Mocha



- Crée en 2011
- Type assert ou BDD (le framework est flexible)
- Fonctionne pour le browser ou node.js
- Installation et lancement des tests (node)  
`npm install -g mocha`  
`mocha`
- Installation et lancement des tests (browser)  
`npm install -g mocha`  
`mocha init`  
`npm install chai`  
`runner.html`  
`karma`





# Tests automatisés - Mocha

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mocha</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="mocha.css" />
  </head>
  <body>
    <div id="mocha"></div>
    <script src="mocha.js"></script>
    <script src="node_modules/chai/chai.js"></script>
    <script>mocha.setup('bdd');</script>
    <script src="src/calculatrice.js"></script>
    <script src="test/calculatrice-test.js"></script>
    <script>
      mocha.run();
    </script>
  </body>
</html>
```

```
var assert = chai.assert;

describe('Test Addition', function() {
  it('2 + 3 devraient faire 5', function () {
    assert.equal(5, calculatrice.ajouter(2, 3));
  });
});
```



# Tests automatisés - Voir aussi



- Coverage :
  - Istanbul : <https://istanbul.js.org>
- Doubles :
  - Sinon : <http://sinonjs.org>
- Parallélisation des tests :
  - Jest : <https://facebook.github.io/jest/>
  - AVA : <https://github.com/avajs/ava>
- Tests End-to-End :
  - Selenium : <http://www.seleniumhq.org>
  - Webdriver : <http://webdriver.io>
- PAAS de tests :
  - Sauce Labs : <https://saucelabs.com>
  - Browser Stack : <https://www.browserstack.com>