

Développement Web avancé : *MEAN*

Mongodb, Express, AngularJS, Nodejs

E. RAMAT

Université du Littoral - Côte d'Opale

23 juin 2016



Sommaire

1 Introduction

2 Node.js

3 Express

4 MongoDB

Plan

1 Introduction

2 Node.js

3 Express

4 MongoDB

Introduction

Objectifs

- comprendre une application web moderne
- introduire le concept d'application multi-tiers
- mettre en place une architecture *Full-stack Javascript*

Compléments abordés

- MongoDB : un système de bases de données *no-sql* basées sur les documents et BSON/JSON
- Express : un framework Web pour nodejs
- AngularJS : un framework JS côté client basé MVC
- NodeJS : une plateforme serveur basé sur le moteur JS V8
- des gestionnaires de modules et de dépendances :
 - ▶ npm : le gestionnaire de modules de nodejs
 - ▶ bower : idem pour la couche cliente

Introduction

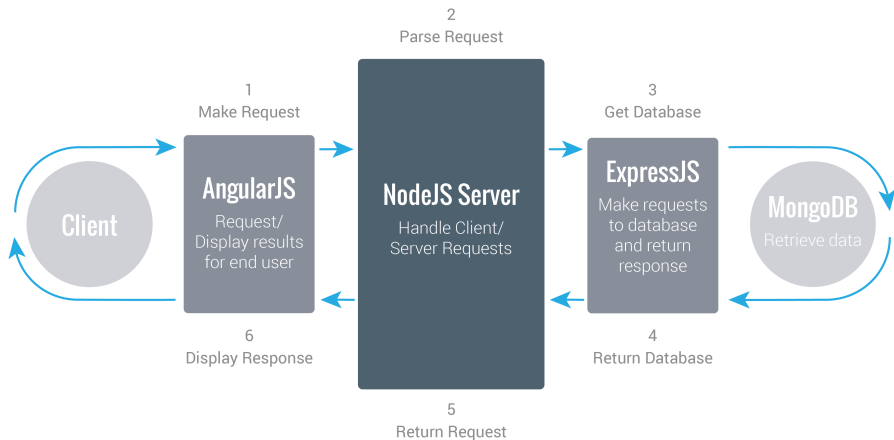
MEAN stack



express

Introduction

Architecture générale



Introduction

Les outils - client

Le debugger

- sous Iceweasel/Mozilla, le plugin Firebug
- sous Chrome, c'est intégré

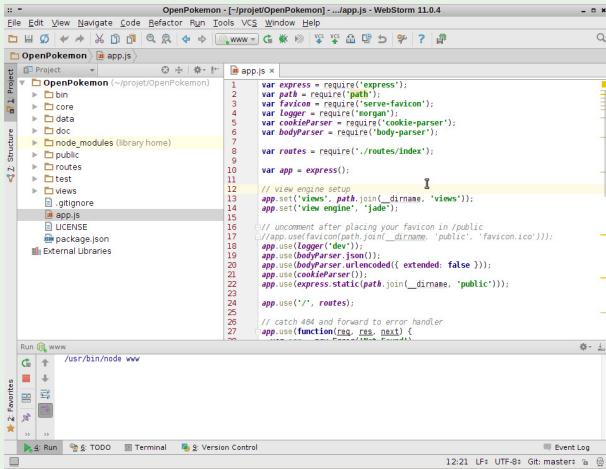
Possibilités

- navigation dans le DOM et modifier des éléments
- tracer les transferts réseaux (méthode, type de ressource, taille, temps, ...)
- visualiser les sources javascript et les fichiers css téléchargés
- exécuter pas à pas des scripts JS, visualiser et modifier des variables JS
- visualiser et modifier les cookies, les bases de données locales (localStorage en HTML5)
- voir le contenu de la console (sortie des scripts JS)

Introduction

Les outils - IDE

Jetbrains - Webstorm



Introduction

Les outils - client

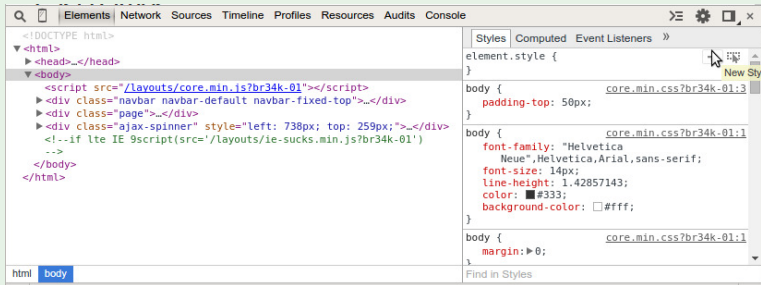
Webstorm

- colorisation, mise en forme automatique, completion, ...
- exécution selon divers configurations (node.js, tests unitaires, ...)
- debug côté serveur
- analyse de code (qualité)
- versionning (git)

Introduction

Les outils - client

Sous Chrome

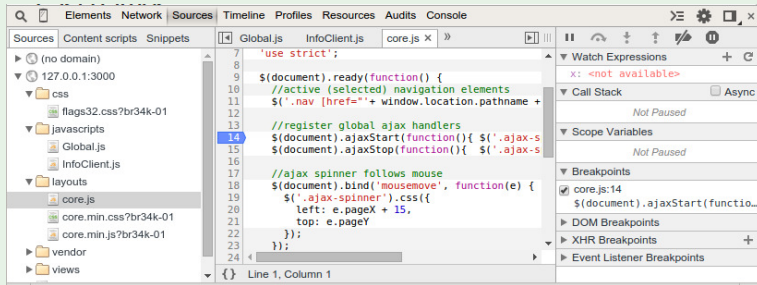


- la vue du DOM
- les éléments de CSS qui s'appliquent sur le document
- on peut aussi cliquer sur un élément d'une page pour obtenir sa représentation dans le DOM et les styles appliqués

Introduction

Les outils - client

Sous Chrome

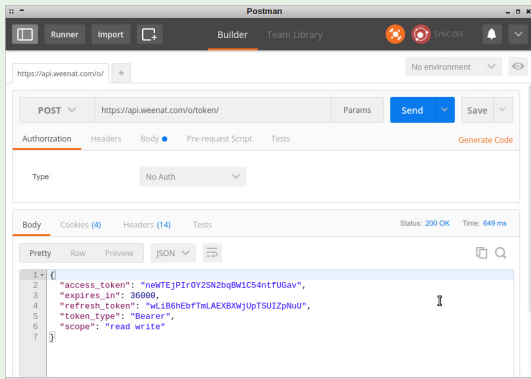


- arborescence des fichiers téléchargés (.js et .css)
- source d'un fichier javascript
- définition d'un point d'arrêt
- suivi de variables

Introduction

Les outils - client

Client HTTP - Postman



- définition de requêtes HTTP (méthode, url, headers, ...)
- affichage des réponses (json, raw, ...)

Plan

1 Introduction

2 **Node.js**

3 Express

4 MongoDB

Node.js

Introduction

V8

- V8 est un moteur Javascript développé par Google
- écrit en C++ et utilisé par Chrome
- compilation à la volée du code JS en code natif
- NodeJS est basé sur V8

NodeJS

- environnement d'exécution de code JS
- applications serveur et réseau (mais aussi scripting général)
- créé par Ryan Dahl et première release le 27 mai 2009
- dernière version stable : 4.2.0
- dernière version : 6.2.1

Avantages

- rapide : basé sur V8, une boucle événementielle et processus non-bloquant
- simple : peu de couche, mécanisme simple, ...
- grande communauté active
- npm : le gestionnaire de modules

Attention

- Node.js propose un Javascript conforme à la spécification ECMAScript6 (Harmony)
- ce n'est pas le même que celui dispose dans un navigateur (pas de DOM, par exemple !)

Philosophie

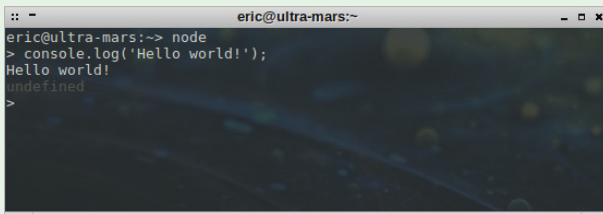
- influencé par Unix
- un noyau très réduit et extensible
- des petits modules (“un module = une fonctionnalité claire”) → problème de la gestion des dépendances → npm
- *reactor pattern* : le coeur de node.js (mono-thread, E/S non bloquant et asynchrone)

Node.js

Introduction

Command-line interface - *CLI*

- interface en ligne de commande pour tester du code node.js
- lanceur de scripts

A screenshot of a terminal window titled 'eric@ultra-mars:~'. The prompt is 'eric@ultra-mars:~>'. The user enters 'node', and the prompt changes to '>'. The user then enters 'console.log('Hello world!');', and the terminal outputs 'Hello world!'. The prompt returns to '>'. The user enters 'undefined', and the terminal outputs 'undefined'. The prompt returns to '>'.

```
eric@ultra-mars:~> node
> console.log('Hello world!');
Hello world!
undefined
>
```

Node.js

Introduction

Gestionnaire de modules - *npm*

- gestion des modules disponibles sur le dépôt npmjs.org
- gestion des dépendances d'un module
- deux modes d'installation :
 - ▶ local : au sein du projet → création d'un répertoire `node_modules`
 - ▶ global : dans le système (sous Linux, dans `/usr/local/lib/node_modules`)

Installation

- local : installation de la dernière version de la branche 4.13 du module `express`

```
$ npm install express@4.13.x
```
- global : installation globale de la dernière version du module `gulp`

```
$ npm install -g gulp
```
- mise à jour :

```
$ npm update -g gulp
```

Gestionnaire de modules - *package.json*

- la liste des modules d'une application peuvent être définie dans un fichier
- `package.json` est un fichier de définition de l'application
- le contenu :
 - ▶ le nom de l'application, sa version, une description, les auteurs, le dépôt du source, la licence, ...
 - ▶ la commande de lancement (*npm start*)
 - ▶ les dépendances

`npm install`

- possibilité d'ajout de commandes au moment de l'installation
- par exemple, la compilation de module écrit C/C++ (via *node-gyp*)

Node.js

Introduction

package.json

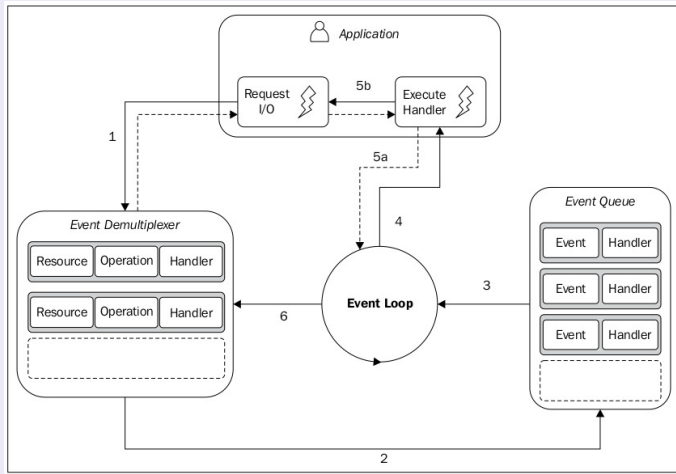
```
{
  "name": "OpenPokemon",
  "version": "0.0.1",
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "express": "~4.13.1",
    "jade": "~1.11.0"
  }
}
```

Le fichier `package.json` se place à la racine de l'application.

Node.js

Introduction

Reactor pattern



Reactor pattern - Explications

- 1. l'application génère une nouvelle opération d'entrée/sortie et génère une requête de traitement ; elle est non-bloquant ; l'application reprend la main ;
- 2. dès que la requête est traitée par le *Event Demultiplexer*, alors un événement est envoyé dans la *Event Queue* avec son handler (la fonction appelée à la fin du traitement) ;
- 3. l'*Event Loop* prend un et un seul événement dans la file d'attente ;
- 4. l'handler est exécuté ;
- 5. à la fin de l'exécution de l'handler, l'*Event Loop* reprend la main et vérifie si de nouvelles requête de traitement ne sont pas apparu ;
- 6. s'il n'y a plus de tâches dans la file d'attente (*Event Queue*) alors la boucle est mise en attente.

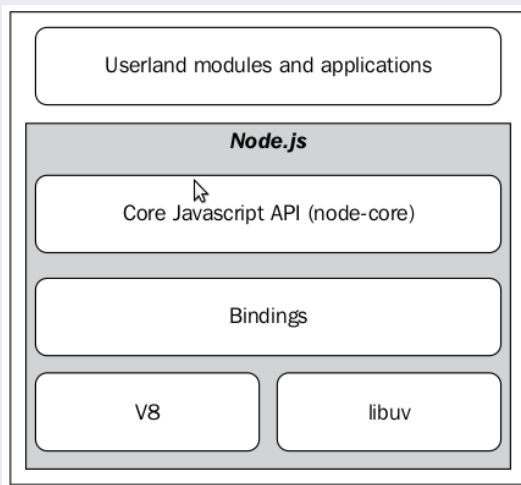
libuv

- ce mécanisme est implémenté dans la *libuv* qui est au coeur de node.js
- le demultiplexeur est en général un pool de threads

Node.js

Introduction

Architecture



Le pattern *callback*

- le pattern *reactor* repose sur les handlers
- un handler est une fonction qui sera appelée à la fin de la requête
- un moyen : les *callbacks* → closure
- un *callback* est une fonction qui est passée à une autre fonction → *continuation-passing style* - CPS

Closure

La *closure* permet de définir des variables et des fonctions au sein de fonctions dont la portée s'applique à tous les enfants.

Closure - exemple

```
var f = function (data, x) {  
  var sum = 0;  
  
  data.forEach(function (y) {  
    sum += y * x;  
  });  
  return sum;  
}
```

Le paramètre *x* et la variable *sum* sont visibles dans la fonction définie comme callback de *forEach*.

Callback - conventions

- le callback est toujours le dernier paramètre

```
fs.readFile(filename, options, callback)
```

- s'il y a une gestion d'erreur alors c'est le premier paramètre de la fonction callback

```
fs.readFile(filename, options, function(err, data) {  
  if (err) {  
    handleError(err);  
  } else {  
    processData(data);  
  }  
});
```

Node.js

Fondements - Module

Constat

En Javascript, aucun mécanisme d'inclusion de source dans un autre fichier source

Module

- permet d'étendre les possibilités de node.js (*require*)
- sorte de librairie
- node.js fournit des librairies

Modules node.js

- *fs* : manipulation du système de fichiers (*file system*)
- *http* : serveur HTTP
- *util* : fonctions utilitaires (format, test de type, ...)
- *path* : manipulation de chemins
- *net* : protocoles réseau et socket
- ...

Module - définition

- le module est une simple valeur :

```
module.exports = 'hello';
```

- le module est une simple fonction :

```
module.exports = function() { return 'hello'; }
```

- le module est un ensemble de fonctions :

```
module.exports.hello = function() { return 'hello'; }  
module.exports.bye = function () { return 'bye'; }
```

- le module est aussi un ensemble de fonctions via un objet :

```
module.exports = {  
  hello: function() { return 'hello'; },  
  bye: function () { return 'bye'; }  
};
```

Un module peut exporter une classe.

Module - importation

- import d'une simple valeur :

```
var hello = require('./lib');  
console.log(hello);
```

- le module est une simple fonction :

```
var hello = require('./lib');  
console.log(hello());
```

- le module est un ensemble de fonctions :

```
var lib = require('./lib');  
console.log(lib.hello());
```

Module et classe

```
module.exports = (function () {  
  var self = {};  
  
  self.A = function (_a) {  
    var a;  
    this.f = function () { ... };  
    var init = function (_a) { a = _a; };  
  
    init(_a);  
  };  
  self.B = function (...) { ... };  
  
  return self;  
})();
```

Explications

- l'export est le résultat de l'appel d'une fonction
- la fonction crée un objet *self*
- l'objet contient la déclaration de 2 classes (A et B)

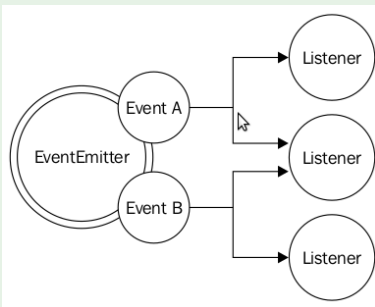
Node.js

Fondements - *EventEmitter*

Définition

Au lieu d'invoquer une fonction de type *callback*, une fonction asynchrone peut émettre des événements et d'autres fonctions/objets sont en attente de ces événements.

Schéma



Node.js

Fondements - *EventEmitter*

Exemple - émetteur

```
var EventEmitter = require('events').EventEmitter;

function f(...) {
  var emitter = new EventEmitter();

  emitter.emit('event1', args);
  emitter.emit('event2', args);

  return emitter;
}
```

Exemple - récepteur

```
f(args).on('event1', function (...) { ... })
      .on('event2', function (...) { ... });
```

once

La fonction *once* permet de déclarer un listener mais après la première invocation, le listener est détruit.

Définition

- un *Promise* est une abstraction à une fonction asynchrone de retourner un objet (**la promesse**) qui représente l'éventuel résultat de la fonction
- la promesse est suspendue tant que la fonction n'a pas délivrée son résultat
- soit la fonction réussit (*resolve*) soit elle échoue (*reject*)

Exemple

La première fonction définie dans le *then* est appelée si la fonction réussit ; sinon la seconde est invoquée pour traiter l'erreur

```
asyncOperation(arg).then(function(result) {  
  ...  
}, function(err) {  
  ...  
});
```

Node.js

Fondements - *Promise*

Promise et node.js

De nombreux modules offrent des fonctions à base de *promise*.

Comment construire une fonction avec une promesse ?

```
var f = function (u, callback) {  
    // traitement et produit d  
    callback(d);  
};  
  
var g = function (u) {  
    return new Promise(function (resolve, reject) {  
        f(u, function (d) { resolve(d); });  
    });  
});  
  
g(u).then(function (d) { ... });
```

La fonction *f* est maintenant asynchrone mais ATTENTION il faut que le traitement fasse appel à des E/S sinon aucun intérêt !

Comment synchroniser plusieurs promesses ?

```
Promise.all(list.map(function (u) {  
  return new Promise(function (resolve, reject) {  
    f(u, function (d) {  
      resolve(d);  
    });  
  });  
})).then(function (r) { ... });
```

Explication

- la fonction *all* permet de créer un ensemble de promesses synchronisées ;
- *then* est invoqué quand toutes les promesses sont réalisées
- *list.map* applique la fonction sur chaque élément de la liste qui construit une promesse

Quelques objets et méthodes

- *process* : tâche principale de l'instance de node.js
- *console* : sortie console (stdout et stderr)
- *timeout* et *interval* : gestion des timers

Process

- *process.argv* : accès aux arguments passés au lancement
- *process* est un *EventEmitter* ; on peut réagit à certains événement

```
process.on('exit', function() { ... }); // fin du processus  
process.on('SIGINT', function() { ... }); // arret via ctrl + d
```

Module *fs*

- surcouche JS des fonctions C (POSIX)
- deux formes : synchrone ou asynchrone
- en synchrone, les erreurs sont gérées via les exceptions

Lecture d'un fichier

- asynchrone

```
fs.readFile('/etc/passwd', function (err, data) {  
  if (err) throw err;  
  console.log(data);  
});
```

- synchrone

```
try {  
  var data = fs.readFileSync('/etc/passwd');  
} catch (err) {  
  ...  
}
```

Module vm

- compiler et exécuter un code JavaScript sous forme d'une chaîne de caractères
- vm : un environnement de compilation et d'exécution dans un programme node.js
- création d'un contexte via la notion de "bac à sable" (*sandbox*)

Exemple avec passage d'objets

```
var util = require('util'),
    vm = require('vm'),
    initSandbox = {
      name: 'RAMAT',
      count: 2
    },
    context = vm.createContext(initSandbox);

vm.runInContext('++count; _name_+=_"Eric"; _organization_=_ "ULCO"', context);
console.log(util.inspect(context));

// { name: 'RAMAT Eric', count: 3, organization: 'ULCO' }
```

Module *path*

- construction et normalisation de chemin
- pas de vérification de l'existence

Contruction et normalisation

```
var p1 = path.join('/var/', '/log/', '/syslog'); // /var/log/syslog
var p2 = path.normalize('/var//log/syslog'); // /var/log/syslog
```

Définition

- http : module client / serveur
- propose une classe *Server* et une méthode de construction *createServer*
- un serveur est une sous-classe d'*EventEmitter*

Exemple

```
var http = require('http');
var server = http.createServer();

server.listen(8080);
server.on('request', function(request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
});
```


Explications

- le serveur écoute les requêtes sur le port 8080
- le paramètre *request* contient la requête HTTP (url, method, header, ...)
- le paramètre *response* permet de construire la réponse

GET/POST/PUT

Si la méthode est :

- GET : `request.url` contient l'url et permet d'identifier la ressource demandée
- POST et PUT : des données sont en général transmises → nécessité d'écouter 'data' et 'end'

Exemple

```
var body = [];  
request.on('data', function(chunk) {  
  body.push(chunk);  
}).on('end', function() {  
  body = Buffer.concat(body).toString();  
});
```

Explications

- sur 'data', chaque paquet de données est stocké dans une liste
- lors de la réception de 'end', *body* devient une chaîne de caractères par concaténation des éléments du vecteur

Réponse

- la variable *response* est un *WritableStream*
- elle permet de construire la réponse au client :

- ▶ *status code* : 2xx, 3xx, 4xx, 5xx, ...

```
response.statusCode = 404;
```

- ▶ le *header* : une réponse en JSON par exemple

```
response.setHeader('Content-Type', 'application/json');
```

Réponse HTML

```
response.write('<html>');
response.write('<body>');
response.write('<h1>Hello, World!</h1>');
response.write('</body>');
response.write('</html>');
response.end();
```

Explications

- chaque ligne est stockée dans la réponse via la méthode *write*
- la fin de l'écriture est marquée par un appel à *end*

Réponse JSON

```
var user = {  
  id: 10,  
  name: 'RAMAT_Eric',  
  organization: 'ULCO'  
};  
  
response.write(JSON.stringify(user));  
response.end();
```

Explications

Avant l'écriture dans le flux de réponse, l'objet est converti en chaîne de caractères.

Réponse JSON

```
var user = {  
  id: 10,  
  name: 'RAMAT_Eric',  
  organization: 'ULCO'  
};  
  
response.write(JSON.stringify(user));  
response.end();
```

Explications

Avant l'écriture dans le flux de réponse, l'objet est converti en chaîne de caractères.

Middleware

- nécessité de traiter les informations des headers
- par exemple, analyse l'url ou les cookies
- un premier module minimal : `connect`
 - ▶ création de serveur HTTP
 - ▶ spécification de routes et des réponses

Exemple

```
var connect = require('connect');  
  
var hello = function (req, res) { res.end('Hello World!'); }  
  
var app = connect();  
  
app.use('/', hello);  
app.listen(8080);
```

Un autre module minimaliste : `request`

Plan

1 Introduction

2 Node.js

3 **Express**

4 MongoDB

Express

- un framework web (plus évolué que connect)
- une gestion de route
- une séparation traitement / vue
- introduction d'un langage de template (jade, par défaut)

Route

C'est la chaîne de caractères après le / de l'url

- la plus simple : /
- une plus complexe : */user/signin*
- avec des données : */user/:id/status*

Jade

- écriture simplifiée de l'HTML
- manipulation de valeur issues des traitements
- structure d'itération, de conditionnelle, ...
- "héritage" entre les fichiers

Un exemple : layout

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

Explications

- structure globale des pages
- *title* est une variable (à renseigner dans le traitement)
- les attributs des balises sont mises en paramètre et séparés par une virgule
- définition d'un bloc *rightarrow* le bloc content devra être défini dans les pages enfants sinon vide

Un exemple

```
extends layout

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

Explications

- héritage de la vue layout
- définition du bloc content
- utilisation de la variable title dans un texte

Express

Organisation

```
create : .  
create : ./package.json  
create : ./app.js  
create : ./public  
create : ./routes  
create : ./routes/index.js  
create : ./routes/users.js  
create : ./views  
create : ./views/index.jade  
create : ./views/layout.jade  
create : ./views/error.jade  
create : ./bin  
create : ./bin/www  
create : ./public/javascripts  
create : ./public/images  
create : ./public/stylesheets  
create : ./public/stylesheets/style.css  
  
install dependencies:  
$ cd . && npm install  
  
run the app:  
$ DEBUG=test:* npm start
```

Explications

- `public` : fichiers statiques (js côté client, css, html, images)
- `views` : fichiers jade
- `routes` : définition des routes et des traitements

Principe

- à la réception d'une requête, la requête peut avoir besoin de prétraitement
- les middlewares mettent en place une chaîne de fonctions
- ces fonctions s'exécutent l'une après l'autre
- utilisation de la méthode `use`
- `use` permet :
 - ▶ d'activer des middlewares
 - ▶ de spécifier des routes

Un exemple

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

Explications

- un middleware prend en paramètre la requête et la réponse
- le troisième paramètre est le middleware suivant
- il réalise un traitement (dans l'exemple, construction d'une erreur et fin de la chaîne normale)

Un autre exemple

```
app.use(express.static(path.join(__dirname, 'public')));
```

Explications

- activation du middleware `express.static`
- objectif : donner accès aux éléments du répertoire public de l'application

La classe router

- la méthode `use` permet de définir des routes
- pour les routes plus complexes, utilisation de la classe `Router`
- possibilité de :
 - ▶ spécifier la méthode
 - ▶ définir des variables

Quelques exemples

- méthode GET, à la racine, passage de la variable `title` et lien avec le fichier template jade via `render`

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

- méthode PUT avec un paramètre et définition de la fonction de traitement dans un module

```
router.put('/user/:id/update', user.update);
```

Attention

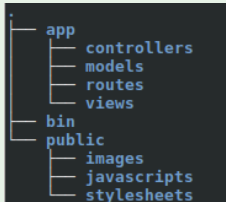
Chemin relatif au chemin défini avec `use`

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

Model-View-Controller

- Express génère les routes et les vues → les traitements peuvent être vus comme des contrôleurs
- si présence d'une base de données, une partie modèle peut apparaître

Arborescence



Exemple - la route

Importation du module main et utilisation de la fonction index

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

router.get('/', main.index);

module.exports = router;
```

Exemple - la contrôleur

Définition d'une fonction index qui exportée

```
module.exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

Intégration d'une lib css/js en jade

Après installation de jQuery et de Bootstrap dans /public/ index

```
doc html
html
  head
    link(rel='stylesheet', href='/bootstrap/css/bootstrap.css')
body
  block content
    script(src='/javascripts/jquery.min.js')
    script(src='/bootstrap/js/bootstrap.min.js')
```

Utilisation de classe et d'id en jade

- construction d'un panel centré avec une entête
- notation :
 - ▶ `<balise>.<class>` pour les classes
 - ▶ `<balise>#<id>` pour les ids

Code jade

```
div.row.vertical-offset-100
  div.col-md-8.col-md-offset-2
    div.panel.panel-primary
      div.panel-heading
        a.glyphicon.glyphicon-chevron-left(href="/devices")
        h3.panel-title #{name}
        a.glyphicon.glyphicon-cog.pull-right(href="/config/device/5")
      div.panel-body
        div#tab
```

Rendu

◀ Appareil P+ 15188



Plan

1 Introduction

2 Node.js

3 Express

4 MongoDB

MongoDB

- base de données de type no-sql
- basé sur JSON et BSON (Binary JSON)
- deux concepts :
 - ▶ collection (les tables en SQL)
 - ▶ document (les enregistrements en SQL)
- tous les documents d'une collection possèdent la même structure (schéma)
- les documents s'expriment à l'aide d'un JSON auquel est ajouté un identifiant unique (`_id`) via un type (*ObjectId*)

Un document

```
{
  "__v" : 0,
  "_id" : ObjectId("5580807984e25f684782fd4d"),
  "color" : "black",
  "turns" : [ "BPRA1", "WPRA2" ]
  "game" : ObjectId("54820a038760269cec59a245"),
  "opponent" : { "id" : ObjectId("553e5ab39093dcb720ca872d") },
  "userCreated" : {
    "id" : ObjectId("55807f5f84e25f684782fd49"),
    "timeCreated" : ISODate("2015-12-14T21:35:23.966Z")
  }
}
```

Explications

- l'attribut name est un string
- l'attribut turns est un tableau de string
- l'attribut game est une référence à un autre document (sous-document)

Tous les types stockables dans un JSON sont possibles (même les dates).

Requêtes

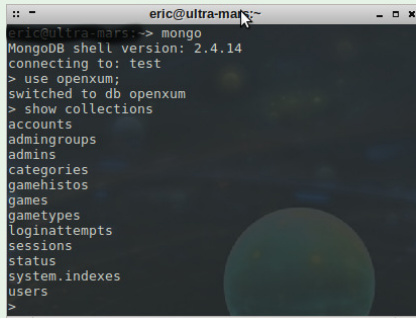
- le langage SQL ne s'applique pas
- à la place, un ensemble de méthodes :
 - ▶ insert / save : création d'un document
 - ▶ update / save : mise à jour d'un document
 - ▶ remove : suppression d'un ou plusieurs documents
 - ▶ find : recherche d'un ou plusieurs documents selon des conditions

MongoDB

Introduction

Console

L'interaction avec mongoDB se fait via une console.

A screenshot of a terminal window titled 'eric@ultra-mar...'. The prompt is 'eric@ultra-mars:~>'. The user has entered 'mongo', which has started the 'MongoDB shell version: 2.4.14'. The prompt is now 'mongo>'. The user has entered 'use openxum;', which has switched the database to 'openxum'. The prompt is now 'openxum>'. The user has entered 'show collections', which has listed the following collections: 'accounts', 'admingroups', 'admins', 'categories', 'gamehistos', 'games', 'gametypes', 'loginattempts', 'sessions', 'status', 'system.indexes', and 'users'. The prompt is now 'openxum>'.

```
eric@ultra-mars:~> mongo
MongoDB shell version: 2.4.14
connecting to: test
> use openxum;
switched to db openxum
> show collections
accounts
admingroups
admins
categories
gamehistos
games
gametypes
loginattempts
sessions
status
system.indexes
users
>
```

Explications

- sélection d'une base de données (openxum)
- `show collections` : liste des collections

CRUD

CRUD = opérations Create / Read / Update / Delete

Création

Plusieurs possibilités :

- avec *insert*

```
db.users.insert({ "name" : "RAMAT_Eric",  
                  "mail" : "eric.ramat@gmail.com",  
                  "username" : "ericR" });
```

- avec *update* et le flag *upsert* (pour indiquer que si le document n'existe pas alors création)

```
db.users.update({ "username" : "ericR" }, { "name" : "RAMAT_Eric",  
                                             "mail" : "eric.ramat@gmail.com",  
                                             "username" : "ericR" }, { upsert: true });
```

Création

- avec `save` et si l'objet ne possède pas d'id

```
db.users.save({ "name" : "RAMAT_ERIC",  
               "mail" : "eric.ramat@gmail.com",  
               "username" : "ericR" });
```

Recherche d'un ou plusieurs documents

- tous les documents d'une collection

```
db.users.find();  
db.users.find({ });
```

- avec une condition sur un attribut

```
db.users.find({ "username" : "ericR" });
```

Projection

- par défaut, tous les attributs sont retournés
- possibilité de sélectionner les attributs

```
db.users.find({ "username" : "ericR" }, { name : 1, mail : 1});
```

- `_id` est toujours retourné

Recherche plus complexe

En combinant des opérateurs logiques et de comparaison

- et/ou

```
db.users.find({ $or { "username": "ericR",  
"username": "ericRamat" } }));
```

- appartenance à un ensemble

```
db.users.find({ "username": { $in: [ "ericR", "ericRamat" ] } });
```

- comparaison

```
db.users.find({ "age": { $gt: 30 } }));
```


Recherche d'un document

Recherche d'un seul document

```
db.users.findOne({ "username": "ericR" });
```

Recherche combinée

- recherche et suppression

```
db.users.findOneAndDelete({ "username": "ericR" });
```

- recherche et mise à jour (condition de sélection et opération de mise à jour)

```
db.users.findOneAndUpdate({ "name" : "E. RAMAT" }, { $inc: { "age" : 1 } })
```

Recherche avec sous-document

Les conditions peuvent s'exprimer sur les sous-documents

```
db.users.find({ "address.city": "Calais" })
```

Address est un sous-document du document *Users* et *city* est un attribut du sous-document

Suppression

La méthode `remove` + une condition (même structure que pour `find`)

- tous les documents respectant la condition

```
db.users.remove({ "age": { $gt: 30 } });
```

- seulement le premier document respectant la condition

```
db.users.remove({ "age": { $gt: 30 } }, true);
```

MongoDB

Mongoose

Mongoose

- module node.js pour MongoDB
- définition de schémas
- validation des documents
- fonctions de manipulation (CRUD)

Connection

```
var mongoose = require('mongoose');  
var app = express();  
  
app.db = mongoose.connect('mongodb://localhost/database');
```

Possibilité de passer en paramètre un couple user/password

Schéma

- définition de la structure des documents
- définition des sous-documents
- typage des champs, valeur par défaut, contraintes
- validation

Exemple

```
var userSchema = new mongoose.Schema({
  name: String,
  email: String,
  createdAt: Date
});

mongoose.model( 'User', userSchema );

var User = mongoose.model( 'User' );
```

Exemple - Valeur par défaut

```
createdOn: { type: Date, default: Date.now }
```

Exemple - Unicité

```
email: { type: String, unique:true}
```

Exemple - Validation

- attribut obligatoire

```
name: { type: String, required: true }
```

- intervalle

```
age: { type: Number, min: 0, max: 120 }
```

- fonction de validation

```
email: {  
  type: String,  
  validate: {  
    validator: function(v) {  
      return /^[^\w-\.] + @ ([\w-] + \. ) + [\w-] {2,4} ) ? $ / . test ( v );  
    },  
    message: '{VALUE} is not a valid email address!'   
  }  
}
```

Sous-documents

- un sous-document est un document dans un document
- avec Mongoose, un schema peut contenir un schema

```
var citySchema = new mongoose.Schema({ name: 'string' });

var userSchema = new mongoose.Schema({
  name: string,
  city: citySchema
})
```


Création

Plusieurs méthodes :

- save

```
var newUser = new User({
  name: 'Eric_Ramat',
  email: 'ramat@lisic.univ-littoral.fr',
  createdAt : Date.now()
}).save( function( err ){
  if(!err){
    console.log('User saved!');
  }
});
```

Création - suite

- create

```
User.create({
  name: 'Eric_Ramat',
  email: 'ramat@lisic.univ-littoral.fr',
  createdAt : Date.now()
}, function( err, user ){
  if(!err){
    console.log('User saved!');
    console.log('_id of saved user: ' + user._id);
  }
});
```

Accès au document créé.

Requête

- idem que sous MongoDB : `find`

```
User.find({'name': 'Eric_Ramat'},  
function (err, users){  
  if (!err) {  
    console.log(users);  
  }  
});
```

- *users* est un tableau mais si le résultat est un unique document
- `Model.find(conditions, [fields], [options], [callback])`

Deux autres méthodes

- une seule instance :

```
Model.findOne(query)
```

- recherche par l'ID :

```
Model.findById(ObjectID)
```

Copyright

Auteur

Éric Ramat *ramat@lisic.univ-littoral.fr*

Licence

Copyright (C) 2016 - LISIC - ULCO

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation ; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".