Node.js

Christophe Lecoutre
lecoutre@cril.fr

IUT de Lens - CRIL CNRS UMR 8188
Université d'Artois
France

Septembre 2017

Plan

Node

Express

React

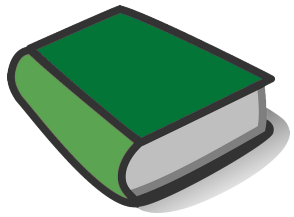
Plan

Node

Express

React

Ressources



- Node.js Design Patterns (2nd Edition), Mario Casciaro and Luciano Mammino, 2016, Packt Publising

- nodejs.org

Node.js

Node.js is a very powerful JavaScript-based framework/platform

- built on Google Chrome's JavaScript V8 Engine,
- used to develop I/O intensive web applications (video streaming sites, single-page applications, etc.),
- open source, completely free, and used by thousands of developers around the world.

Node version Manager

Installation et Guide :

https://github.com/creationix/nvm#install-script

```
curl -o- https:... | bash // installer nvm
nvm install node  // installer node (latest release)
nvm install --lts node // long-term support
command -v nvm // vérifier l'installation
node -v // version
nvm list // lister les versions
nvm list-remote // lister les versions distantes
nvm use 4 // utiliser la version 4
nvm use 8.6 // utiliser la version 8.6
nvm alias default 4.5 // par default, la version 4.5
```

Remarque

Un répertoire `.nvm` est créé.

Node - REPL Terminal

```
$ node
> x = 10
10
> let y = 10;
undefined
> x + y
20
> console.log("Hello World")
Hello World
undefined
> for (let i=0; i<3; i++)
... console.log(i);
0
1
2
undefined
>
```

Remarque

Taper ctrl-c, deux fois, pour quitter.

Node - Evaluer un fichier

Soit le fichier *test.js* suivant :

```
'use strict';

console.log("Hello World")
for (let i=0; i<3; i++)
  console.log(i);
```

En éxecutant :

```
node test.js
```

On obtient :

```
Hello World
0
1
2
```

Créer un serveur très rudimentaire

Soit le fichier *server.js* suivant :

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

http.createServer((request, response) => {
   // Send the HTTP header :  200 (OK) and text/plain
   response.writeHead(200,{'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"
   response.end('Hello World\n');
}).listen(8081);

// Server console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

On le lance avec `node server.js`

Créer une application avec npm

Commandes à executer pour créer une application avec une dépendance sur trois modules.

```
mkdir appli
cd appli
npm init
npm install babel-cli babel-preset-es2015 express --save
```

On obtient un fichier *package.json* comme suit :

```
{
  "name": "appli",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "toto",
  "license": "ISC",
  "dependencies": {
    "babel-cli": "^6.26.0",
    "babel-preset-es2015": "^6.24.1",
    "express": "^4.15.5"
  }
}
```

Soit le fichier *index.js* suivant à la racine du projet :

```
import express from "express";

const PORT = 3000;
const app = express();

app.get("/", (req, res)
  => res.json({status: "Test API"}));
app.listen(PORT, ()
  => console.log(`Test API on Port ${PORT}`));
```

En remplaçant dans *package.json* la ligne "test" de "scripts" par:

```
"start": "babel-node --presets 'es2015' index.js"
```

On peut :

- lancer le serveur avec `npm start`
- lancer le client (browser) sur `localhost:3000`

Remarque

Les modules sont placés dans le répertoire *node_modules*.

Node Package Manager

De nombreux modules accessibles par la commande *npm* ou sur le site https://npmjs.com
De nombreuses commandes en ligne :

```
npm init // initier un projet
npm install module [--save | --save-dev]
npm remove module  [--save | --save-dev]
npm update module
npm list
npm -v // version
npm addUser
```

Remarque

Les options `--save` et `--save-dev` font référence aux champs 'dependencies' et 'devDependencies' du fichier package.json.

Remarque

On peut aussi utiliser `npm -g` pour gérer les commandes au niveau globale (du système), et non seulement au niveau du projet.

Node Package Manager

Nommage des modules. Voir https://docs.npmjs.com/misc/semver

```
version Must match version exactly
>version Must be greater than version
>=version
<version
<=version
~version "Approximately equivalent to version"
^version "Compatible with version"
*
```

Remarque

A noter que les numéros de version de la forme x.y.z indiquent :

- x : niveau majeur
- y : niveau mineur
- z : patch

Module *fs*

The Node File System (fs) module can be imported with the statement *require*.

Soit le fichier *reading.js* suivant :

Example

```
const fs = require("fs");

// Asynchronous read
fs.readFile('titi.txt', (err, data) => {
  if (err)
    return console.error(err);
  console.log("Async. " + data.toString());
});

// Synchronous read
console.log("Sync. " + fs.readFileSync('titi.txt'));

console.log("Program Ended");
```

If the file *titi.txt* is :

```
Bonjour à tous.
Comment allez-vous ?
```

and you execute:

```
node reading.js
```

then the output is:

```
Sync. Bonjour à tous.
Comment allez-vous ?

Program Ended
Async. Bonjour à tous.
Comment allez-vous ?
```

Module *os*

Example

```
const os = require("os");

os.hostname(); // ludwig
os.type();     // Linux
os.platform(); // linux
os.arch();     // x64
(os.totalmem/1e6).toFixed(1) + 'MB'; // 1042.2 MB
(os.freemem/1e6).toFixed(1) + 'MB';  // 195.8 MB
console.log(os.cpus());
[ {
    model: 'Intel(R) Core(TM) i7 CPU 860  @ 2.80GHz',
    speed: 2926,
    times: { user: 252020, nice: 0, sys: 30340, idle:
        1070356870, irq: 0 }
  },
  {
    model: 'Intel(R) Core(TM) i7 CPU 860  @ 2.80GHz',
    speed: 2926,
    times: { user: 306960, nice: 0, sys: 26980, idle:
        1071569080, irq: 0 }
  } ]
```

Module *process*

Each Node.js process has a set of built-in functionality, accessible through the global process module, which doesn't need to be required.

```
process.stdout.write("Hello World!" + "\n");
process.argv.forEach(
  (val, ind, array) => console.log(ind + ': ' + val));
console.log(process.execPath);
console.log(process.pid, process.version);
console.log(process.platform, process.title);
process.on('exit', () => console.log('exiting'));
process.on('uncaughtException',
  (err) => console.error('Uncaught error ', err.stack));
```

If the code above is in `test.js` and execute `node test.js`:

```
Hello World!
0: /usr/bin/node
1: /home/lecoutre/Desktop/proMern/test.js
/usr/bin/node
23289 'v8.6.0'
linux node
exiting
```

Module *url*

The url module provides utilities for URL resolution and parsing.

Example

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

let p = url.parse('http://www.arte.fr/people?name=alice');
console.log(p.protocol);  // http:
console.log(p.host);      // www.arte.fr
console.log(p.query);     // name=alice
console.log(p.pathname);  // /people
console.log(p.search);    // ?name=alice
```

Module *path*

Si on exécute le fichier suivant :

Example

```
const path = require("path");

console.log(path.normalize('/d1/d2//d3/d4/tab/..'));
console.log(path.join('/d1', 'd2', 'd3/d4', 'tab', '..'));
console.log(path.resolve('main.js'));
console.log(path.extname('main.js'));
console.log(path.join(__dirname,'tutu'));
console.log(path.join(__dirname,__filename));
```

on obtient :

```
/d1/d2/d3/d4
/d1/d2/d3/d4
/home/lecoutre/appliTest/main.js
.js
/home/lecoutre/appliTest/tutu
/home/lecoutre/appliTest/home/lecoutre/appliTest/path1.js
```

Module *EventEmitter*

```
const EventEmitter = require('events').EventEmitter;
const ee = new EventEmitter();

const listener1 = () => console.log('listener 1');
ee.on('bip', listener1);
ee.on('bip', () =>  console.log('listener 2'));

let nListeners = EventEmitter.listenerCount(ee, 'bip');
console.log(nListeners + " Listeners");

ee.emit('bip'); // fire the event
ee.removeListener('bip', listener1);
console.log("Listener 1 will not listen now.");
ee.emit('bip'); // fire the event

nListeners = EventEmitter.listenerCount(ee, 'bip');
console.log(nListeners + " Listeners");

console.log("Program Ended.");
```

If the previous code is in file *listening.js* and we execute:

```
node listening.js
```

then the output is:

```
2 Listeners
listener 1
listener 2
Listener 1 will not listen now.
listener 2
1 Listeners
Program Ended.
```

Streams

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams:

- Readable – Stream which is used for read operation.
- Writable – Stream which is used for write operation.
- Duplex – Stream which can be used for both read and write.
- Transform – A type of duplex stream where the output is computed based on input.

Each type of Stream is an EventEmitter instance and throws several events at different instance of times. For example, some of the commonly used events are:

- data – This event is fired when there is data is available to read.
- end – This event is fired when there is no more data to read.
- error – This event is fired when there is any error.
- finish – This event is fired when all the data has been flushed to underlying system.

Soit le fichier *stream1.js* suivant :

```
const fs = require("fs");

let data = '';

const rs = fs.createReadStream('titi.txt');
rs.setEncoding('UTF8');
rs.on('data', (chunk) => data += chunk);
rs.on('end', () => console.log(data));
rs.on('error', (err) => console.log(err.stack));

console.log("Program Ended");
```

If we execute:

```
node stream1.js
```

then the output is:

```
Program Ended
Bonjour à tous.
Comment allez-vous ?
```

N'y a-t-il pas un problème avec l'exécution en séquence ici ?

```
const fs = require("fs"), zlib = require('zlib');

function compress(filename) {
  fs.createReadStream(filename)
    .pipe(zlib.createGzip())
    .pipe(fs.createWriteStream(filename + '.gz'))
    .on('finish', () => console.log('finished');
  console.log("Compressed: " + filename + '.gz');
}

function uncompress(filename) {
  fs.createReadStream(filename)
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream(filename.slice(0,-3)));
  console.log("Decompressed: " + filename.slice(0,-3));
}

compress('titi.txt');
uncompress('titi.txt.gz');
```

Créer un serveur de fichiers simple

Le serveur :

```
const http = require('http'), fs = require('fs');
const url = require('url');

http.createServer((req, res) => {
  const pathname = url.parse(req.url).pathname;
  console.log("Req for " + pathname + " received.");

  // Read the reqed file content from file system
  fs.readFile(pathname.substr(1), (err, data) => {
    if (err) {
       res.writeHead(404, {'Content-Type': 'text/html'});
       console.log(err);
    } else {
       res.writeHead(200, {'Content-Type': 'text/html'});
       res.write(data.toString());
    }
    res.end();
 });
}).listen(8081);

console.log('Server running at http://127.0.0.1:8081/');
```

Créer un serveur de fichiers simple

Le client :

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

// Options for the request
const options = { host: 'localhost', port: '8081', path: '
    /test.html'  };
// Callback function for dealing with the response
const callback = (response) => {
  let s = '';
  response.on('data', (data) => s += data);
  response.on('end', () => console.log(s));
}
// Make a request to the server
const req = http.request(options, callback);
req.end();
```

Remarque

Lorsque le serveur est lancé, puis le client, on obtient en retour le contenu du fichier dans la console du client.

Système de modules (CommonJS)

Créer un module revient à construire un objet ou une fonction et à le rendre accessible par des instructions liées à l'exportation.

Pour bien comprendre les choses, voici ce que Node gère automatiquement autour du code d'un fichier *kit.js*.

```
var module = { exports: {} };
var exports = module.exports;
  // le code du fichier kit.js où il faut modifier
  // l'une des deux variables (au choix)
return module.exports;
```

Pour importer le module dans un autre fichier *test.js*, on écrira simplement :

```
const module = require('./kit.js');
 // le code du fichier test.js
```

Exporter un objet

Un fichier *person.js* :

```
const obj = {
    name : 'Alice',
    introduce() {
      console.log('Hello ' + exports.name);
    }
}
module.exports=obj;
```

Un fichier *appli.js* :

```
const person = require('./person.js');
console.log('Name : ' + person.name);
person.introduce();
```

L'exécution de *appli.js* :

```
node ./appli.js
```

Exporter une fonction

Un fichier *amanda.js* :

```
function calculate(a, x, n) {
  if (x === 1) return a*n;
  return a*(1-Math.pow(x,n))/(1-x);
}
module.exports = calculate;
```

Un fichier *tyler.js* :

```
function calculate(r) {
  return 4/3*Math.PI*Math.pow(r,3);
}
module.exports = calculate;
```

Un fichier *appli.js* :

```
const geometricSum = require('./amanda.js');
const sphereVolume = require('./tyler.js');
console.log(geometricSum(1,2,5) + ' ' + sphereVolume(2));
```

Types de modules

Il y a trois types de modules :

- les *core* modules, fournis par Node. On donne leur nom directement comme par exemple :

```
require('fs')
require('http')
```

- les *npm* modules, installés par cette commande. On donne leur nom directement comme par exemple :

```
require('express')
require('debug')
```

- les *file* modules, construits par l'utilisateur. On donne leur nom, toujours en préfixant par /, ou ./, ou encore ../ comme par exemple :

```
require('./amanda.js')
require('../person.js')
```

Remarque

Certains core modules sont globaux et ne nécessitent même pas d'instruction `require`. Par exemple, *process* and *buffer*.

Modules npm

Ils sont installés dans le répertoire *node_modules*.

Remarque

Lors d'une instruction `require(x)`, où x désigne un npm module, x sera recherché dans *node_modules*. S'il n'est pas trouvé, il sera recherché dans *node_modules*, s'il existe, du répertoire parent. Et ainsi de suite.

Attention

1. Ne pas glisser ses propres modules dans *node_modules*.
2. Ne pas utiliser plusieurs `require` sur le même module, ou alors comprendre qu'un cache est utilisé.
3. On peut détruire *node_modules* et réinstaller toutes les dépendances placées dans *package.json* avec `npm install`

Système de modules (ES6 modules)

Every ES6 module needs to be represented by a separate .js file. An ES6 file can export and import any number of variables.

Exporting and importing comes in different formats:

```
export {name};
export {name1, name2, name3};
export {name1 as alias1, name2 as alias2};
export {name as default};
export {name1 as default, name2 as alias2, name3};
export default function() { ... };
export default class ... ;
export {name1, name2} from 'anotherModule';
export * from 'anotherModule';
```

```
import defaultName from 'module';
import * as alias from 'module';
import { mame } from 'module';
import { name as alias } from 'module';
import { name1 , name2 } from 'module';
import defaultName , { name [ , [...] ] } from 'module';
import defaultName, * as alias from 'module';
import 'module';
```

Examples

File *test.js* :

```
function cube(x) {
  return x * x * x;
}
const truc = Math.PI + Math.SQRT2;
export { cube, truc };
```

```
import { cube, truc } from 'test';
console.log(cube(3)); // 27
console.log(truc);    // 4.555806215962888
```

File *test.js* :

```
export default function cube(x) {
  return x * x * x;
}
```

```
import cube from 'test';
console.log(cube(3)); // 27
```

File *mathModules/logarithm.js* :

```
function getLN2() { return Math.LN2; }
function getLN102() { return Math.LN10; }
export { getLN2, getLN10 };
```

File *mathModules/trigonometry.js* :

```
function getSin(v) { return Math.sin(v); }
function getCos(v) { return Math.cos(v); }
export { getSin, getCos };
```

File *mathStuff.js* :

```
import * as logarithm from 'mathModules/logarithm';
import * as trigonometry from 'mathModules/trigonometry';
export default { logarithm, trigonometry }
```

File *appli.js* :

```
import math from 'mathStuff.js';
console.log(math.trigonometry.getSin(3));
console.log(math.logarithm.getLN2(3));
```

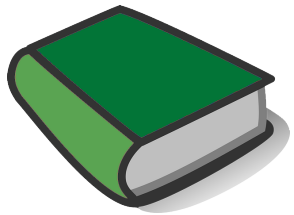
Plan

Node

Express

React

Ressources



- Express in Action,
  Evan M. Hahn,
  2016, Manning

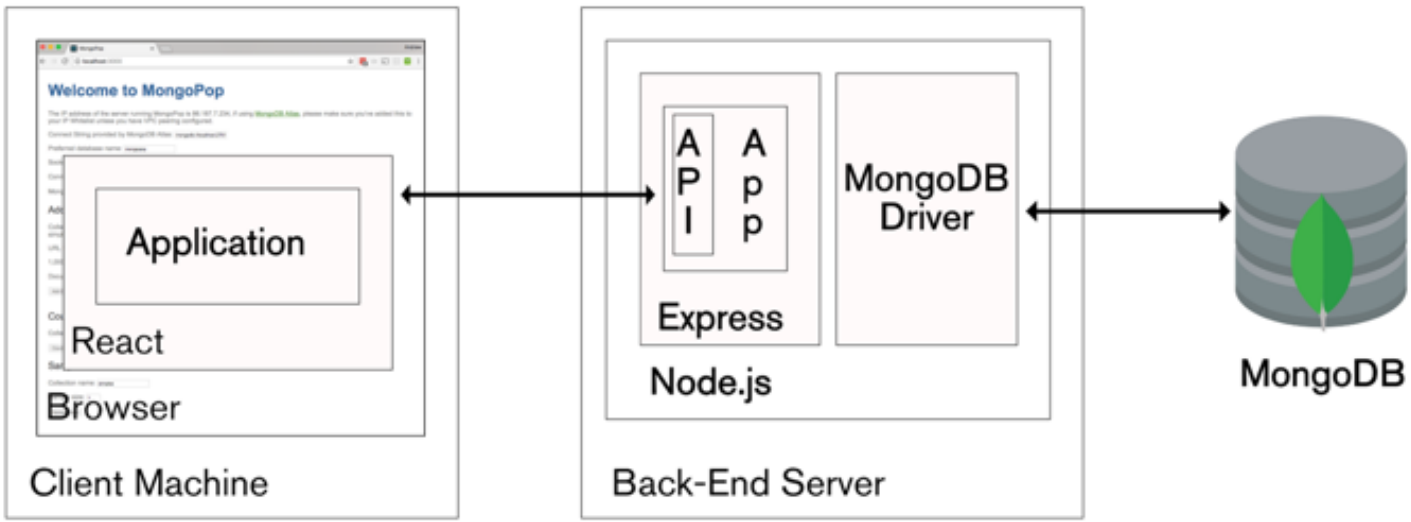
- http://expressjs.com/

Express

Express est un framework JavaScript :

- construit sur Node
- permettant de développer facilement des serveurs web

Il est présent dans la suite MERN (et également dans la suite MEAN).



Remarque

Express ajoute de nombreuses fonctionalités par rapport au service élementaire http de Node.

Serveur Express

Pour construire un serveur, il suffit d'exécuter la fonction retournée par le module *express*.

Example

```
const express = require("express");
const app = express();

app.get('/', function(request, response) {
    response.send("Hello world!");
});

app.listen(3000, () => console.log("Waiting at 3000"));
```

Remarque

The Express application object can be referred from the request object and the response object with a field called *app*.

Major Features

- middleware : requests flow through a stack, which is an array of functions
- routing : functions associated with URLs
- extensions to request and response objects : extra methods and properties for developer convenience
- views : HTML dynamically rendered after choosing one template engine (e.g., React)

Middlewares

The signature of a middleware function (that must be passed to `app.use()`).

```
function(request, response, next) { ... }
```

- request is an object that represents the incoming HTTP request
- response is an object that represents the outgoing HTTP response
- next is a function that will go the next middleware when called

Une fonction middleware met fin à une requète en appelant `res.end`.

Remarque

les méthodes `res.send`, `res.sendFile` et `res.json` appellent indirectement `res.end`.

Important:

- chaque middleware peut modifier la requète ou la réponse
- il faut que l'un des middlewares réponde à la requète (sinon ?)
- pour passer au middleware suivant, appeler le callback *next()*

Example

```
const app = require("express")();

app.use(function(request, response, next) {
    console.log("First");
    next();
});
app.use(function(request, response, next) {
    console.log("Second");
    next();
});
app.use((req, res) => res.send("Done"));
app.listen(3000, () => console.log("Waiting at 3000"));
```

Illustration with *morgan*

Morgan is a nice logger for Express. In the code below, a password is sent provided that the current number of minutes at the clock is a multiple of 2.

Example

```
const app = require("express")();
const logger = require("morgan");

app.use(logger("short"));
app.use(function(req, res, next) {
    const t = new Date().getMinutes();
    if (t % 2 == 0)
        next();
    else
        res.status(403).end("Not authorized");
});
app.use((req, res) => res.send("Password: toto"));

app.listen(3000, () => console.log('Server started'));
```

If the code from the previous file is put in a file called *test.js*, to run the server, type

```
node test.js
```

An alternative is to have *package.json* as follows:

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
   "scripts": {
     "start": "node test.js",
     "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.2",
    "morgan": "^1.9.0"
  }
}
```

and type:

```
npm start
```

Illustration with *express.static*

To send static files, we can use *express.static*. It suffices to indicate the name of a directory where files can be downloaded.

Example

```
const express = require("express");
const path = require('path');
const http = require('http');

const app = express();

app.use(express.static(path.resolve(__dirname,'public')));
app.use(function(req, res, next) {
  res.writeHead(200, {'content-Type': 'text/plain' });
  res.end("Looks like you didn't find a static file.");
});

app.listen(3000, () => console.log('Server started'));
```

Try `http://localhost:3000/titi` before and after building a text file `public/titi`.

Error-handling Middlewares

Lorsqu'un problème survient, il faut appeler *next()* avec un argument qui représente l'erreur. Par exemple:

```
next(new Error('Something bad happened!'));
```

L'erreur est alors traitée par la chaine de error-handling middlewares, qui prennent quatre arguments (le premier étant l'erreur).

- Pour passer de l'un à l'autre, appeler *next* avec un argument.
- Pour quitter la chaîne, appeler *next* sans argument.

Two error-handling middlewares are present here:

- the first one, used for logging
- the second one, used to send a response

Example

```
app.use((req, res, next) => {
  res.sendFile('titi', function(err) {
    if (err)
      next(new Error('Error when sending file'));
  }
});
app.use((err, req, res, next) => {
  console.log(err);
  next(err);
});
app.use((err, req, res, next) => {
  res.status(500);
  res.send('Internal server error');
});
```

Routing

For reading web pages, it suffices to call `app.get`.

Example

```
const express = require("express");
const path = require('path');
const http = require('http');

const app = express();

app.use(express.static(path.resolve(__dirname,'public')));
app.get('/', (req, res) => res.end("Home page!"));
app.get('/about', (req, res) => res.end("About page!"));
app.get('/jobs', (req, res) => res.end("Jobs page!"));
app.use((req,res) => res.status(404).end('404!'));

app.listen(3000, () => console.log("Waiting at 3000"));
```

Try `http://localhost:3000/`, `http://localhost:3000/about`, `http://localhost:3000/jobs` and `http://localhost:3000/titi`.

Parameters in Routing

A parameter can be defined by using the character ':'. You can get its value with `req.params`.

Example

```
app.get('/users/:userId', (req, res) => {
  const userId = req.params.userId;
  res.send('Id = ' + userId);
});
```

This will not match `/users/` and `/users/12/posts` but this will match `/users/toto`. If we only want integers for user ids :

Example

```
app.get(/^\/users\/(\d+)$/, (req, res) => {
  const userId = req.params[0];
  res.send('Id = ' + userId);
});
```

Parameters in Queries

In URLs, you can provide information under the form of a query. You get this information with `req.query`.

Example

```
app.get('/search', (req, res) => {
   Object.entries(req.query)
         .forEach(t => console.log(t[0] + ':' + t[1]));
   res.send('searching...');
});
```

If you hit:

```
http://localhost:3000/search/?a=10&b=7&c=titi
```

you get in the server console:

```
a:10
b:7
c:titi
```

Using Auxilliary Routers

To get an auxilliary router object, just call `express.Router()`.

Example

```
const sub = require("express").Router();
sub.get('/posts', (req, res) => res.end("Posts page!"));
sub.get('/mails', (req, res) => res.end("Mails page!"));
module.exports = sub;
```

If the code above is in file *sub.js* and you run the following server:

Example

```
const app = require("express")();
app.use('/students', require('./sub'));
app.use((req,res) => res.status(404).end('404!'));
app.listen(3000, () => console.log("Waiting at 3000"));
```

then, you get the pages with:

- `http://localhost:3000/students/posts`
- `http://localhost:3000/students/mails`

Object Request

A lot of useful information in the first object passed to each middleware:

- req.app: a reference to the instance of the Express application that is using the middleware
- req.header
- req.body: object containing key-value pairs of data submitted in the request body (POST and PUT). It is undefined, except if you use body-parsing middleware such as body-parser.
- req.hostname
- req.ip : the remote IP address of the request
- req.method
- req.url, req.originalURL
- req.protocol
- req.path
- req.query

Object Response

A lot of useful information in the second object passed to each middleware:

- res.app: a reference to the instance of the Express application that is using the middleware
- res.end(): ends the response process. Use to quickly end the response without any data. If you need to respond with data, instead use methods such as res.send() and res.json().
- res.json(): sends a response (with the correct content-type) that is the parameter converted to a JSON string using JSON.stringify()
- res.redirect(): Redirects to the specified URL
- res.send(): Sends the HTTP response. The body parameter can be a buffer (application/octet-stream), a string (text/html), an object or an array (both being stringified in JSON).
- res.sendFile(): Transfers the specified file
- res.status(): Sets the HTTP status for the response.

CRUD

HTTP methods map to CRUD (Create, Read, Update, Delete) operations.

- Create : POST
- Read : GET
- Update : PUT and PATCH
- Delete : DELETE

The methods that can be used with Express are:

- app.all()
- app.post()
- app.get()
- app.put()
- app.delete()

CRUD

Example

```
const app = require("express")();

app.all('/', (req, res, next) => {
  console.log('a request ' + req.method);
  next();
});
app.get('/', (req, res) => res.send('get request'));
app.post('/', (req, res) => res.send('post request'));
app.put('/', (req, res) => res.send('put request'));
app.delete('/', (req, res) => res.send('delete request'));

app.listen(3000, () => console.log("Waiting at 3000"));
```

You can hit the server as follows:

- curl -X PUT http://localhost:3000/
- curl -X POST http://localhost:3000/

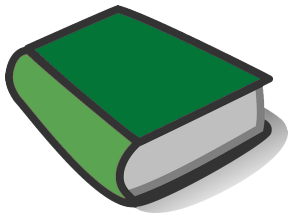
Plan

Node

Express

React

Ressources



- Pro MERN Stack, Vasan Subramanian, 2017, Apress

- https://reactjs.org/

React from a CDN

Use something like this in the head element of your page:

Example

```
<head>
  <meta charset="utf-8" >
  <title>Hello World!</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/
      react/15.2.1/react.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/
      react/15.2.1/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/
      babel-core/5.8.23/browser.min.js"></script>
</head>
```

Scripts for React

Example

```
<body>
  <div id="container"></div>

  <script type="text/babel">
    let container = document.getElementById('container');
    let component = <h1>Hello World!</h1>;
    ReactDOM.render(component, container);
  </script>
</body>
```

React Classes

It suffices to build a class extending React.Component, which contains a method *render*.

Example

```
const container = document.getElementById('contents');
class Test extends React.Component {
  render() {
    return (
      <div>Test !</div>
    );
  }
}
ReactDOM.render(<Test />, container);
```

Remarque

Il faut auto-fermer la balise en utilisant le caractère '/'.

Composing Components

Example

```
class Sub1 extends React.Component {
  render() { return ( <div> Sub 1 </div> ); }
}
class Sub2 extends React.Component {
  render() { return ( <div> Sub 2 </div> ); }
}
class Test extends React.Component {
  render() {
    return (
      <div>
        <Sub1 />
        <hr />
        <Sub2 />
      </div>
    );
  }
}
ReactDOM.render(<Test />, document.getElementById('ct'));
```

Passing Data

Any data can be passed by introducing some user-defined attributes, which are accessed in the React component with *this.props*

Example

```
class Table extends React.Component {
  render() {
    const borderedStyle = {border: "1px solid silver",
        padding: 6};
    return (
      <table style={{borderCollapse: "collapse"}}>
        <tr>
          <th style={borderedStyle}>Id</th>
          <th style={borderedStyle}>Title</th>
        </tr>
        <Row id={1} title="One" />
        <Row id={2} title="Two" />
      </table>
    );
  }
}
```

Passing Data

Example

```
class Row extends React.Component {
  render() {
    const borderedStyle = {border: "1px solid silver",
        padding: 4};
    return (
      <tr>
        <td style={borderedStyle}>{this.props.id}</td>
        <td style={borderedStyle}>{this.props.title}</td>
      </tr>
    );
  }
}
```

Remarque

For the attribute `style`, you pass in an object containing key-value pairs. The keys are same as the CSS style names, except that camel case is used (e.g., `border-collapse` is replaced by `borderCollapse`).

Property Validation

Properties being passed can be validated against a specification, supplied by a static getter function `propTypes()` in the class of the React component, and returning an object containing key-value pairs:

- the name of the property is the key
- the validator is the value, which is one of the many constants exported by `React.PropTypes`.

Example

```
static get propTypes() {
  return {
    id: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
  };
}
```

An alternative is:

```
Row.propTypes = {
  id: React.PropTypes.number.isRequired,
  title: React.PropTypes.string
};
```

Using Children

Another way to pass data is by using the content of the HTML-like node of the component. This can be accessed with `this.props.children`

Example

```
class BorderWrap extends React.Component {
 render() {
   const borderedStyle = {border: "1px solid silver",
       padding: 6};
    return (
      <div style={borderedStyle}>
        {this.props.children}
      </div>
    );
  }
}
-----
<BorderWrap>
   <ExampleComponent />
</BordeWrap>
```

Passing objects

Of course, it is possible to pass objects.

Example

```
const persons = [
 { id: 1, name: 'Toto', born: new Date('2016-08-15') },
 { id: 2, name: 'Titi', born: new Date('2016-08-16') }
];
-----
<Group persons={persons} />
-----
class Group extends React.Component {
  render() {
    const rows = this.props.persons.map(
      p => <Row key={p.id} person={p} />);
    return ( <table> {issueRows} </table> );
  }
}
```

Remarque

It is important, for efficiency reasons, to associate a unique key with each component of an array/iterator.

Passing functions

The way to communicate from a child component to its parent is by passing callbacks from the parent to the child, which it can call to achieve specific tasks.

Example

```
hh
```

React state

The state of a React component is managed by `this.state`.
Whenever the state changes, it triggers a rerender of the component and the view automatically changes.

Example

```
class Things extends React.Component {
  constructor() {
    super();
    this.state = { things: [] };
    this.add = this.add.bind(this);
  }
  add(thing) {
    const t = this.state.things.concat(thing);
    this.setState({ things: t });
  }
  render() { return (
    <div>
      <ListThings things={this.state.things} />
      <AddThing add={this.add} />
    </div>
  ); }
}
```

Example

```
class AddThing extends React.Component {
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleSubmit(e) {
    e.preventDefault();
    const form = document.forms.formAdd;
    this.props.add({
      f1: form.field1.value,
      f2: form.field2.value,
    });
    form.field1.value = ""; form.field2.value = "";
  }
  render() { return (
    <div>
      <form name="formAdd" onSubmit={this.handleSubmit}>
        <input name="field1" />
        <input name="field2" />
        <button>Add</button>
      </form>
    </div>
  ); }
}
```

React with Express

Example

```
const app = require('express')();
app.use(require('express').static('static'));
app.use(require('body-parser').json());

const things = [  ... ];

app.get('/things', (req, res) => {
  const metadata = { count: things.length };
  res.json({ _metadata: metadata, records: things });
});

app.post('/things', (req, res) => {
  const newThing = req.body;
  newThing.created = new Date();
  things.push(newThing);
  res.json(newThing);
});

app.listen(3000, () => console.log('App at 3000'));
```

Example

```
class Things extends React.Component {
  constructor() {
    super();
    this.state = { things: [] };
    this.add = this.add.bind(this);
  }

  componentDidMount() {
    this.loadData();
  }

  loadData() {
    fetch('/things').then(response =>
      response.json()
    ).then(data => {
      console.log("Total count:", data._metadata.count);
      this.setState({ things: data.records });
    }).catch(err => {
      console.log(err);
    });
  }
```

Example

```
  add(newThing) {
    fetch('/things', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(newThing),
    }).then(response => {
      if (response.ok) {
        response.json().then(th => {
          th.created = new Date(th.created);
          const t = this.state.things.concat(th);
          this.setState({ things: t });
        });
      } else
        response.json().then(err => alert(err.message));
    }).catch(err => alert("Error: " + err.message));
  }

  render() { return (
    <div>
      <ListThings things={this.state.things} />
      <AddThing add={this.add} />
    </div>
  ); }
}
```

Important Details

The package body-parser is necessary to parse various types of request bodies including URL-encoded form data and JSON.

```
const bodyParser = require('body-parser');
...
app.use(bodyParser.json());
```

The popular library jQuery is an easy way to use the $.ajax() function. But modern browsers support asynchronous calls natively via the Fetch API:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/fetch
    /1.0.0/fetch.min.js">
</script>
```

An Ajax call using fetch(), is similar to $.ajax(). It takes in the path of the URL to be fetched, and returns a promise with the response as the value. Concerning the response:

- we can use the json() method to parse it
- date strings must be converted into date objects

Important details

It may be relevant to initialize the state after the component has been mounted (in the constructor, it may cause problems). This is possible by putting the following method in the React class of the compornent:

```
componentDidMount() {
  // initialize state here
}
```

Remarque

If the state contains an array to be modified, you need to make a copy or a clone of the existing array, append to it, and put it in the state. You must call `setState()`.

Stateless Components

Some components are stateless ; typically, only containing a `render()` method.

For performance reasons, it is recommended that such components are written as functions rather than classes: functions that take in `props` and just renders based on it.

Example

```
function Row(props) {
  const borderedStyle = {border: "1px solid silver",
      padding: 4};
  return (
    <tr>
      <td style={borderedStyle}>{props.id}</td>
      <td style={borderedStyle}>{props.title}</td>
    </tr>
  );
}
```

JS and JSON

- to convert JS objects into JSON strings, use *JSON.stringify*
- to convert JSON strings into JS objects, use *JSON.parse*

Example

```
let guys = {
    toto : {
        age : 15,
        city : 'Lille'
    },
    titi : {
        age : 28,
        city : 'Paris'
    }
}

let s = JSON.stringify(guys).replace(/city/g, 'town');
console.log(s);
let guys2 = JSON.parse(s);
guys2.toto.age = 16;
console.log(guys2);
console.log(JSON.stringify(guys2));
```