

# Tester avec Node.js

Joseph AZAR – Oct 2022

Les tests vous permettent d'identifier les bogues dans votre code plus rapidement et plus efficacement. Les cas de test doivent être écrits pour vérifier que chaque morceau de code produit la sortie ou les résultats attendus. Les tests peuvent agir comme une forme de documentation pour la logique de votre programme.

Les tests unitaires (Unit Tests) sont un type spécifique de test où des composants individuels ou des unités de code sont testés. Les petits tests unitaires fournissent une spécification granulaire par rapport à laquelle votre programme peut être testé. S'assurer que votre base de code est couverte par des tests unitaires facilite le processus de développement, de débogage et de refactorisation en fournissant une mesure de base de la qualité sur laquelle vous pouvez travailler. Disposer d'une suite de tests complète peut permettre d'identifier les bogues plus tôt, ce qui peut économiser du temps et de l'argent, car plus un bogue est détecté tôt, moins il est coûteux à corriger.

Il existe plusieurs frameworks et approches pour les tests unitaires tels que :

- tape
- Mocha
- Jest
- Stubbing HTTP requests
- Puppeteer
- Configuring Continuous Integration tests (Configuration des tests d'intégration continue)

Dans ce codelab, nous nous concentrerons sur le framework **Mocha**.

## Premiers pas avec Node.js et Mocha

Mocha est une bibliothèque de test pour Node.js, créée pour être simple, extensible et rapide. Il est utilisé pour les tests unitaires et d'intégration, et c'est un excellent candidat pour le BDD (Behavior Driven Development).

En génie logiciel, le développement axé sur le comportement (Behavior Driven Development) est un processus de développement logiciel agile qui encourage la

collaboration entre les développeurs, les experts en assurance qualité et les représentants des clients dans un projet logiciel.

Nous allons commencer par créer un répertoire vide.

```
$ mkdir testing
```

```
$ cd testing
```

Le gestionnaire de paquets officiel de Node - npm - nous aidera à configurer un projet vide. L'option de ligne de commande **init** de npm lancera un assistant, qui créera un **package.json** pour notre projet.

Pour les besoins de ce tutoriel, répondez à ses questions comme suit :

```
1 {
2   "name": "testing",
3   "version": "0.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1"
8   },
9   "author": "",
10  "license": "ISC"
11 }
```

La structure de notre projet contiendra un répertoire d'application avec le code principal et un répertoire de test avec les fichiers de test Mocha :

```
$ mkdir app
```

```
$ mkdir test
```

Ensuite, nous allons installer notre framework de test et une bibliothèque d'attentes (expectation library) appelée **Chai** qui remplace bien la fonction d'assertion (**assert**) standard de Node. **Remarque : vous aurez peut-être besoin de la version 16 de nodejs pour terminer ce projet.**

```
$ npm install mocha
```

```
$ npm install chai
```

Pendant la phase de test du serveur, nous aurons besoin d'un moyen d'envoyer des requêtes HTTP. Le module "**request**" est un excellent choix. Installez-le comme suit :

```
$ npm install request
```

Enfin, nous aurons également besoin du package Express qui définit un simple DSL (langage spécifique au domaine ⇒ Domain Specific Language) pour le routage et la gestion des requêtes HTTP entrantes :

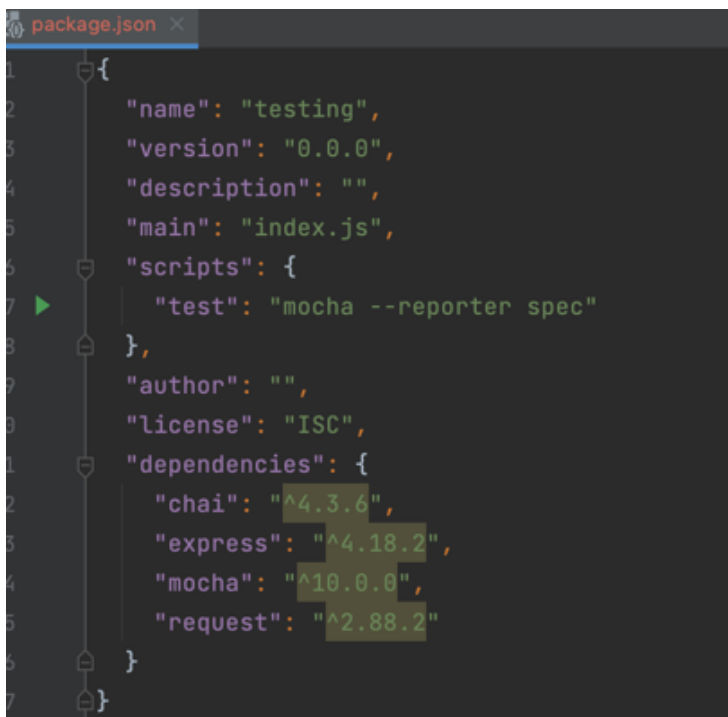
```
$ npm install express
```

Nous allons configurer la commande test dans le fichier **package.json**, afin d'exécuter nos tests simplement en exécutant `npm test` depuis la ligne de commande.

La commande suivante est utilisée pour invoquer le binaire Mocha installé localement dans le répertoire **./node\_modules** :

```
$ npx mocha --reporter spec
```

Ensuite, nous mettrons à jour la commande de test dans **package.json** pour contenir la commande ci-dessus. Ce fichier devrait maintenant ressembler à ceci :



```
1 {
2   "name": "testing",
3   "version": "0.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "mocha --reporter spec"
8   },
9   "author": "",
10  "license": "ISC",
11  "dependencies": {
12    "chai": "^4.3.6",
13    "express": "^4.18.2",
14    "mocha": "^10.0.0",
15    "request": "^2.88.2"
16  }
17 }
```

Nous suivrons la pratique TDD (développement piloté par les tests ⇒ Test Driven Development) et commencerons notre projet en créant des tests avant la mise en œuvre.

Commençons par créer un fichier de test :

```
$ touch test/convert.js
```

Mocha nous donne la possibilité de décrire les fonctionnalités que nous implémentons en nous donnant une fonction de description qui résume nos attentes. Le premier argument

est une chaîne simple qui décrit la fonctionnalité, tandis que le second argument est une fonction qui représente le corps de la description.

Description de notre convertisseur de couleurs :

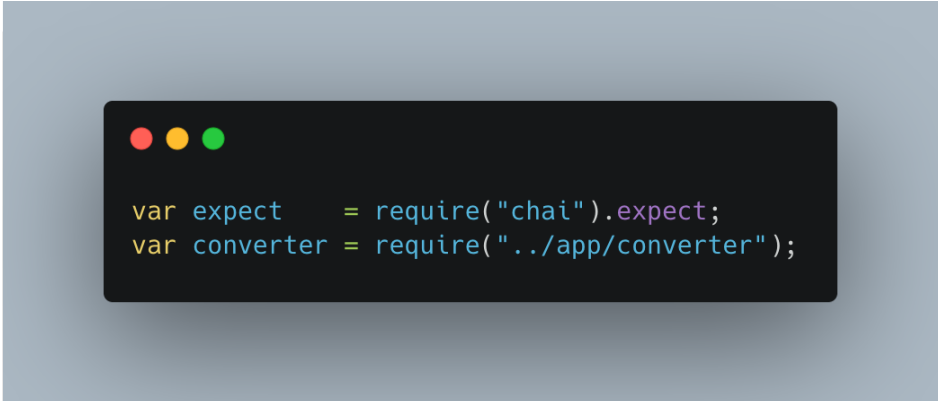
```
describe("Convertisseur de code couleur", function() {  
  describe("Conversion RVB en Hex", function() {  
    it("convertit les couleurs de base", function() {  
  
    });  
  });  
  
  describe("Conversion Hex vers RVB", function() {  
    it("convertit les couleurs de base", function() {  
  
    });  
  });  
});
```



Dans le corps de cette description, nous avons créé des segments qui représentent notre fonctionnalité un peu plus en détail.

Nous pouvons mettre en place une chose concrète que nous testons en utilisant la fonctionnalité **it**. La fonction **it** est très similaire à la fonction **describe**, sauf que nous ne pouvons mettre des attentes que dans le corps de la fonction **it**.

Nous pouvons maintenant introduire notre première attente. Nous utiliserons la bibliothèque **Chai** et son mot-clé **expect** pour comparer le résultat de l'implémentation de notre fonctionnalité et le résultat que nous espérons obtenir. Bien sûr, nous devons d'abord importer la bibliothèque **Chai** :



```
var expect = require("chai").expect;
var converter = require("../app/converter");
```

Notez que notre code d'implémentation n'existe pas à ce stade, mais cela ne doit pas nous déranger, et nous pouvons écrire nos tests comme si l'implémentation existait déjà.

Dans le corps de notre première fonction **it**, nous allons tester le convertisseur de couleur **rgbToHex**. Comme son nom l'indique, il convertit un tuple (rouge, vert, bleu) en hexadécimal. Nous allons le tester en convertissant les couleurs de base. Le fichier **test/converter.js** final ressemble à :

```
var expect = require("chai").expect;
var converter = require("../app/converter");

describe("Convertisseur de code couleur", function() {
  describe("Conversion RVB en Hex", function() {
    it("convertit les couleurs de base", function() {
      let redHex = converter.rgbToHex(255, 0, 0);
      let greenHex = converter.rgbToHex(0, 255, 0);
      let blueHex = converter.rgbToHex(0, 0, 255);

      expect(redHex).to.equal("ff0000");
      expect(greenHex).to.equal("00ff00");
      expect(blueHex).to.equal("0000ff");
    });
  });
});
```

```
describe("Conversion Hex vers RVB", function() {  
  it("convertit les couleurs de base", function() {  
    let red    = converter.hexToRgb("ff0000");  
    let green  = converter.hexToRgb("00ff00");  
    let blue   = converter.hexToRgb("0000ff");  
  
    expect(red).to.deep.equal([255, 0, 0]);  
    expect(green).to.deep.equal([0, 255, 0]);  
    expect(blue).to.deep.equal([0, 0, 255]);  
  });  
});  
});
```

Remarquez la partie **.to.deep.equal** de l'extrait ci-dessus. C'est ce qu'on appelle un **matcher**, et il fait correspondre le résultat d'une caractéristique avec une valeur attendue.

Introduction

Expect / Should

Assert

Plugin Utilities

Online Test Suite

language chains

not

deep

nested

own

ordered

any

all

a

include

## .deep

Causes all `.equal`, `.include`, `.members`, `.keys`, and `.property` assertions that follow in the chain to use deep equality instead of strict (`===`) equality. See the `deep-eql` project page for info on the deep equality algorithm: <https://github.com/chaijs/deep-eql>.

```
// Target object deeply (but not strictly) equals `{a: 1}`
expect({a: 1}).to.deep.equal({a: 1});
expect({a: 1}).to.not.equal({a: 1});

// Target array deeply (but not strictly) includes `{a: 1}`
expect([{a: 1}]).to.deep.include({a: 1});
expect([{a: 1}]).to.not.include({a: 1});

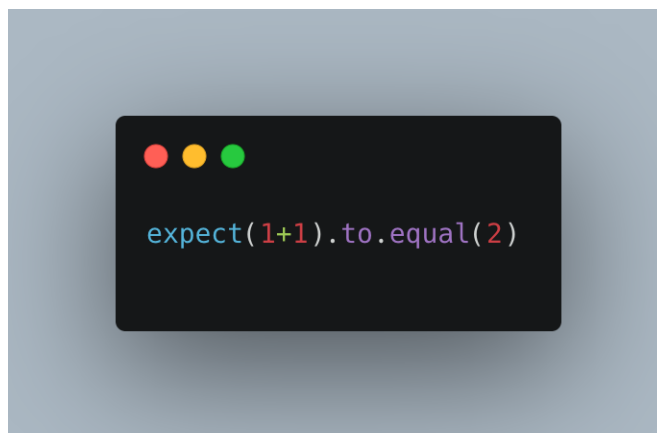
// Target object deeply (but not strictly) includes `x: {a: 1}`
expect({x: {a: 1}}).to.deep.include({x: {a: 1}});
expect({x: {a: 1}}).to.not.include({x: {a: 1}});

// Target array deeply (but not strictly) has member `{a: 1}`
expect([{a: 1}]).to.have.deep.members([{a: 1}]);
expect([{a: 1}]).to.not.have.members([{a: 1}]);

// Target set deeply (but not strictly) has key `{a: 1}`
expect(new Set([[{a: 1}]])).to.have.deep.keys([[{a: 1}]]);
expect(new Set([[{a: 1}]])).to.not.have.keys([[{a: 1}]]);

// Target object deeply (but not strictly) has property `x: {a: 1}`
expect({x: {a: 1}}).to.have.deep.property('x', {a: 1});
expect({x: {a: 1}}).to.not.have.property('x', {a: 1});
```

Il existe bien sûr de nombreux autres matchers définis dans la bibliothèque Chai (<https://www.chaijs.com/api/bdd/>), qui peuvent correspondre aux attentes selon divers critères. Par exemple, pour vérifier l'égalité de deux objets simples, on pourrait écrire :



Dans l'exemple ci-dessus, nous avons utilisé **deep.equal** car nous comparons deux objets imbriqués. La partie **.deep** indique à Chai de faire correspondre tous les éléments des tableaux, un par un.

Si nous exécutons `npm test` à ce stade, il se plaindra d'un fichier d'implémentation manquant pour notre convertisseur de couleurs. Nous allons implémenter notre convertisseur dans le fichier suivant :

**\$ touch app/convert.js**

La première description de notre convertisseur de couleur décrit une conversion de couleur RVB en hexadécimal. Il prend trois arguments et renvoie une chaîne qui représente la couleur dans sa représentation hexadécimale.

Nous utiliserons la méthode **toString** des nombres entrants, combinés avec le nombre 16, pour invoquer une conversion en représentation hexadécimale.


```
exports.rgbToHex = function(red, green, blue) {  
  var redHex = red.toString(16);  
  var greenHex = green.toString(16);  
  var blueHex = blue.toString(16);  
  return pad(redHex) + pad(greenHex) + pad(blueHex);  
};  
  
function pad(hex) {  
  return (hex.length === 1 ? "0" + hex : hex);  
}
```

Notez que nous avons complété la valeur résultante avec un préfixe zéro si elle ne contient qu'un seul caractère, car une représentation hexadécimale valide doit toujours contenir deux caractères.

Pour implémenter la fonction qui convertit la représentation hexadécimale en RVB, nous utiliserons la fonction `parseInt` avec la base 16 pour convertir des parties des chaînes entrantes en valeurs RVB décimales valides.

```
exports.hexToRgb = function(hex) {  
  var red = parseInt(hex.substring(0, 2), 16);  
  var green = parseInt(hex.substring(2, 4), 16);  
  var blue = parseInt(hex.substring(4, 6), 16);  
  return [red, green, blue];  
};
```





```
// app/converter.js
exports.rgbToHex = function(red, green, blue) {
  var redHex   = red.toString(16);
  var greenHex = green.toString(16);
  var blueHex  = blue.toString(16);
  return pad(redHex) + pad(greenHex) + pad(blueHex);
};

function pad(hex) {
  return (hex.length === 1 ? "0" + hex : hex);
}

exports.hexToRgb = function(hex) {
  var red   = parseInt(hex.substring(0, 2), 16);
  var green = parseInt(hex.substring(2, 4), 16);
  var blue  = parseInt(hex.substring(4, 6), 16);
  return [red, green, blue];
};
```

Après cette étape, nous pouvons exécuter nos tests avec **npm test** et devrions voir que tous nos tests réussissent :

```
(base) josephazar@wifi-edr-h-204-153 testing % npm test
```

```
> testing@0.0.0 test
> mocha --reporter spec
```

```
Convertisseur de code couleur
```

```
Conversion RVB en Hex
```

```
✓ convertit les couleurs de base
```

```
Conversion Hex vers RVB
```

```
✓ convertit les couleurs de base
```

```
2 passing (3ms)
```

## Implémentation du serveur Web

Dans cette étape, nous allons exposer le convertisseur de couleurs via une API HTTP et démontrer l'écriture de tests pour le code asynchrone à l'aide de **Mocha**.

Tout d'abord, nous allons créer un fichier de test :

```
$ touch test/server.js
```

Comme dans le test précédent, nous aurons besoin de **chai**. Pour tester la requête HTTP, nous aurons également besoin du package de **request**.

```
var expect = require("chai").expect;
var request = require("request");
```

Voici la description de la fonctionnalité que nous voulons implémenter, joliment présentée avec la description de Mocha. Nous allons stocker le chemin complet vers la ressource que nous voulons tester dans une variable. Avant d'exécuter les tests, nous allons

exécuter notre serveur Web sur le port localhost 3000. Notez que dans une plus grande suite de tests, il est probablement plus facile et plus agréable de mettre la partie hôte des URL dans une constante globale et de la réutiliser dans tous les tests.

Pour faire une requête, nous utiliserons le package **request**. Nous devons lui transmettre deux arguments : une URL à visiter et une fonction à invoquer lorsque la requête est terminée. Nous allons configurer nos attentes à l'intérieur de ces fonctions de rappel.

```
var expect = require("chai").expect;
var request = require("request");

describe("API de conversion de code couleur", function() {
  describe("Conversion RVB en Hex", function() {
    var url = "http://localhost:3000/rgbToHex?red=255&green=255&blue=255";

    it("retourne le statut 200", function(done) {
      request(url, function(error, response, body) {
        expect(response.statusCode).to.equal(200);
        done();
      });
    });

    it("retourne la couleur en hexadécimal", function(done) {
      request(url, function(error, response, body) {
        expect(body).to.equal("ffffff");
        done();
      });
    });
  });
});

describe("Conversion Hex vers RVB", function() {
  var url = "http://localhost:3000/hexToRgb?hex=00ff00";
```

```

it("retourne le statut 200", function(done) {
  request(url, function(error, response, body) {
    expect(response.statusCode).to.equal(200);
    done();
  });
});

it("retourne la couleur en RVB", function(done) {
  request(url, function(error, response, body) {
    expect(body).to.equal("[0,255,0]");
    done();
  });
});
});

```

Si nous exécutons le code ci-dessus sans la commande **done**, quelque chose d'étrange se produira. Rien n'échouera, mais, en même temps, aucune attente ne sera vérifiée. Cela se produit parce que nous n'avons pas donné à **Mocha** suffisamment de temps pour attendre la fin des requêtes. En d'autres termes, le code à l'intérieur du rappel de la requête n'est jamais réellement exécuté.

Heureusement, **Mocha** nous donne une belle abstraction pour ce problème. Pour chaque **it** qui doit attendre une valeur de réponse, nous allons injecter une fonction de rappel **done** et ne l'appeler que lorsque nos attentes ont été exécutées. De cette façon, **Mocha** saura qu'il doit attendre certaines des attentes.

```

// test/server.js
var expect = require("chai").expect;
var request = require("request");

describe("API de conversion de code couleur", function() {
  describe("Conversion RVB en Hex", function() {
    var url = "http://localhost:3000/rgbToHex?red=255&green=255&blue=255";
    it("retourne le statut 200", function(done) {
      request(url, function(error, response, body) {
        expect(response.statusCode).to.equal(200);
        done();
      });
    });
    it("retourne la couleur en hexadécimal", function(done) {
      request(url, function(error, response, body) {
        expect(body).to.equal("ffffff");
        done();
      });
    });
  });
  describe("Conversion Hex vers RVB", function() {
    var url = "http://localhost:3000/hexToRgb?hex=00ff00";

    it("retourne le statut 200", function(done) {
      request(url, function(error, response, body) {
        expect(response.statusCode).to.equal(200);
        done();
      });
    });
    it("retourne la couleur en RVB", function(done) {
      request(url, function(error, response, body) {
        expect(body).to.equal("[0,255,0]");
        done();
      });
    });
  });
});

```

Créons un nouveau fichier pour notre implémentation d'API :

**\$ touch app/server.js**

Nous utiliserons le framework **Express** pour créer une API Web simple et inclurons également notre précédente implémentation de convertisseur de couleurs. Nous allons définir les routes et écouter le port 3000. Le code est le suivant :

```
// app/server.js
```

```
var express = require("express");
```

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

```
var converter = require("./converter");
```

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

```
    var red    = parseInt(req.query.red, 10);
```

```
    var green  = parseInt(req.query.green, 10);
```

```
    var blue   = parseInt(req.query.blue, 10);
```

```
    var hex = converter.rgbToHex(red, green, blue);
```

```
    res.send(hex);
```

```
});
```

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

```
    var hex = req.query.hex;
```

```
    var rgb = converter.hexToRgb(hex);
```

```
    res.send(JSON.stringify(rgb));
```

```
});
```

```
app.listen(3000);
```



```
// app/server.js
var express = require("express");
var app = express();
var converter = require("./converter");

app.get("/rgbToHex", function(req, res) {
  var red = parseInt(req.query.red, 10);
  var green = parseInt(req.query.green, 10);
  var blue = parseInt(req.query.blue, 10);
  var hex = converter.rgbToHex(red, green, blue);
  res.send(hex);
});

app.get("/hexToRgb", function(req, res) {
  var hex = req.query.hex;
  var rgb = converter.hexToRgb(hex);
  res.send(JSON.stringify(rgb));
});

app.listen(3000);
```

Pour exécuter nos tests, nous devons d'abord exécuter notre serveur :

**\$ node app/server.js**

Nous pouvons maintenant exécuter nos tests avec **npm test** dans une session de terminal séparée :

```
(base) josephazar@wifi-edr-h-204-153 testing % npm test
```

```
> testing@0.0.0 test /Users/josephazar/Desktop/NodeJs/Web_Dev_2022_2023/nodejs-project/testing
```

```
> mocha --reporter spec
```

#### Convertisseur de code couleur

##### Conversion RVB en Hex

✓ convertit les couleurs de base

##### Conversion Hex vers RVB

✓ convertit les couleurs de base

#### API de conversion de code couleur

##### Conversion RVB en Hex

✓ retourne le statut 200

✓ retourne la couleur en hexadécimal

##### Conversion Hex vers RVB

✓ retourne le statut 200

✓ retourne la couleur en RVB

6 passing (27ms)