# Les notions de test unitaire et test fonctionnel

# Table des matières

# I. Contexte

**Durée** : 1 h 30

**Environnement de travail** : Visual Studio Code

**Pré-requis** : Bases de JavaScript

***Contexte***

L'écriture de code entraîne obligatoirement un risque d'erreur et d'introduction de bugs dans l'application. C'est pourquoi il est important de tester le code, aussi bien dans le fond (d'un point de vue technique) que dans la forme (d'un point de vue fonctionnel). Au fur et à mesure que le code de l'application va grossir, il va être impossible de tester manuellement tous ces aspects. Heureusement, des solutions existent pour écrire différents types de tests et automatiser leur lancement.

# II. Notions générales

### Objectif

- Définir ce qu'est un test unitaire et son utilité

### Mise en situation

Les tests unitaires permettent de gérer les différentes erreurs potentiellement commises par un développeur et assurent la bonne maintenabilité du code dans le temps. Il y a différents types de tests et des patterns précis à respecter pour s'assurer du bon fonctionnement de ceux-ci.

***Définition*** **Test unitaire**

Un **test unitaire** permet de tester une partie précise du code. Par exemple, une méthode en particulier. Ils doivent donc être concis. Un test unitaire doit pouvoir être joué de manière isolée, c'est-à-dire sans que des données externes ou une action de l'utilisateur ne puissent influer sur le résultat du test. Par conséquent, toutes les données utilisées devront être maîtrisées lors du test. Par exemple, il n'est pas souhaitable de faire de vrais appels à la base de données. Il est préférable d'utiliser des objets créés durant le test, qui vont simuler la base de données.

Pour être efficace, un test unitaire doit tester le cas passant, mais aussi les cas aux limites.

Ces tests pourront être joués manuellement par le développeur sur son IDE, ou lancés automatiquement lors d'un processus d'intégration continue.

***Exemple***

Notre application comporte un formulaire de saisie de carte bleue, qui vérifie au cours de la saisie la validité des informations grâce à l'algorithme de Luhn.

Notre test unitaire se concentrera uniquement sur la méthode qui vérifie la validité du numéro de carte bleue saisi.

```
1 function validCreditCard(cardNumber) {
2   if(/*cardNumber vérifie l'algorithme de Luhn*/) {
3     return true;
4   } else {
5     return false;
```

```
  }
}
```

Le paramètre `cardNumber` sera défini dans l'écriture du test, et ne dépendra pas d'une entrée utilisateur.

***Méthode*** **Écrire un test unitaire, le pattern AAA**

Le pattern AAA (*Arrange*, *Act*, *Assert*) est le modèle de rédaction de test unitaire le plus utilisé. Il consiste à diviser la rédaction d'un test unitaire en 3 parties :

1. **Arrange** : Initialisation des objets, définition des valeurs transmises à la méthode à tester.
2. **Act** : Appel de la méthode testée avec les paramètres définis dans la section *Arrange*.
3. **Assert** : Vérification que la méthode testée est conforme à ce que l'on attend.

***Syntaxe*** **À retenir**

- Les tests unitaires sont un aspect primordial du travail de développeur. Ils permettent de s'assurer du bon fonctionnement du code, quelles que soient les actions de l'utilisateur ou les données en entrées et sorties.

## III. Exercice : Appliquez la notion

### Question

À partir de la fonction suivante, qui retourne le jour de la semaine, écrivez une fonction JavaScript qui testera la fonction ci-dessous en vous inspirant du pattern AAA.

```
const getDayOfDate = (day, month, year) => {
  const d = new Date(year, month, day);
  switch (d.getDay()) {
    case 0:
      return 'Dimanche'
      case 1:
      return 'Lundi'
      case 2:
      return 'Mardi'
      case 3:
      return 'Mercredi'
      case 4:
      return 'Jeudi'
      case 5:
      return 'Vendredi'
      case 6:
      return 'Samedi'
  }
}
```

Pour réaliser ce test, on utilisera :

- Arrange : donnée de la date du 15 mai 2020.
- Act : comparaison avec le résultat attendu.
- Assert : validation avec un `boolean` en retour.

## IV. Les différents types de tests

**Objectifs**

- Connaître les différents types de tests
- Comprendre l'utilité des différents types de tests

**Mise en situation**

Tester un script de quelques lignes réalisé par un seul développeur est très rapide, et l'on peut facilement identifier les erreurs. Mais sur un projet de plus grande envergure, avec parfois plusieurs développeurs, il devient délicat (voire impossible) de tester manuellement tout le programme à chaque ajout de fonctionnalités.

C'est pourquoi il est judicieux d'écrire des tests automatisés, qui pourront êtres joués manuellement et automatiquement dans un cycle d'intégration continue.

***Définition*** **Test fonctionnel**

Le test fonctionnel (ou test système) a pour but de tester le fait que l'application est bien conforme avec les spécifications fonctionnelles. Ces tests permettent de tester un comportement complet, à la différence des test unitaires qui ne testent qu'une petite partie. Ainsi, les fonctionnalités vont être testées en simulant les actions de l'utilisateur (clic, saisie, etc.).

***Exemple***

Dans le cas d'un formulaire de saisie de carte bleue, le test fonctionnel va simuler les actions de l'utilisateur (saisie du numéro de carte, de la date d'expiration, du cryptogramme, du nom) et cliquer sur le bouton de soumission du formulaire. Le test sera validé si la carte bleue est bien enregistrée.

***Remarque*** **Tests unitaires vs tests fonctionnels**

Si l'on reprend l'exemple d'un formulaire :

- le **test unitaire** va tester le type de donnée unitairement. Si nous avons un `input` de type `mail`, on vérifiera la réaction du code, dans le cas où le format du mail est valide et dans le cas où le format du mail n'est pas valide.
- Le **test fonctionnel** testera en revanche la réaction du code à l'envoi du formulaire. Si une des données n'est pas valide, que se passe-t-il ? Les bonnes informations sont-elles enregistrées en base de données ? etc.

**Les autres types de tests**

Il existe d'autres types de tests tout aussi importants :

- **Le test d'intégration**, qui est l'étape arrivant après le test unitaire. Son rôle est de tester le bon fonctionnement des différentes parties du code (testées unitairement) entre elles.

  Par exemple : "La saisie d'un numéro de carte bleue déclenche la vérification de la validité du numéro".

- **Le test End-to-End (ou E2E)**, qui est l'étape arrivant après le test fonctionnel. Son rôle est, comme son nom l'indique, de tester l'application de bout en bout, permettant ainsi de valider le bon fonctionnement de scénarios métier complexes.

  Par exemple :

  - L'utilisateur se connecte avec le rôle "Client",
  - Clique sur le lien "Ajoutez une carte bleue",
  - Le formulaire de saisie d'une carte bleue s'affiche,
  - L'utilisateur saisit le numéro de carte valide dans le champ "Numéro de carte",

- Le champ est entouré de vert pour signaler que le numéro est valide,
- L'utilisateur clique sur "Valider",
- ...
- Un message s'affiche avec le texte "Carte créée avec succès".

- **Le test Smoke** (ou *Sanity testing*) est un test simple d'une fonctionnalité critique. Ce peut être un test unitaire ou fonctionnel. Ce test est critique : s'il échoue, il est inutile de lancer les autres tests.

  Par exemple : "Est-ce que le clic sur le bouton "Ajouter une carte bleue" ouvre le formulaire de saisie ?".

***Syntaxe*** **À retenir**

- Il existe plusieurs formes de tests avec des utilités particulières, du test unitaire (pour tester une méthode sans impacts extérieurs) au test E2E (pour valider le fonctionnel de l'application complète).

## V. Exercice : Appliquez la notion

### Question 1

Un site de e-commerce veut faire des tests sur l'ajout d'articles à un panier.

Cette fonctionnalité est structurée de cette manière :

- À côté de chaque article, il y a un champ de formulaire permettant de rentrer le nombre d'articles et un bouton *Ajoutez au panier*.
- Lorsque l'on ajoute au panier un article, cela envoie une requête au serveur, qui enregistre dans la base de données les éléments du panier.

À partir du code ci-dessous, écrivez les tests unitaires permettant de tester si le type de données concernant la quantité d'articles est bien un `number`.

Testez la fonction avec 2 et avec '2'. Si un des tests retourne *false*, corrigez le problème.

```
1 const getArticleNumber = (numberOfArticles) => {
2    return numberOfArticles;
3 }
```

Pour réaliser les tests unitaires, on utilisera :

- Arrange : 2 et '2'
- Act : comparaison avec le résultat attendu
- Assert : validation avec un `boolean` en retour

### Question 2

Quel pourrait être le test fonctionnel de cette fonctionnalité (il n'y a pas besoin d'écrire de code, une explication logique suffira) ?

## VI. Node.js et NPM

**Objectifs**

- Découvrir Node.js et NPM
- Savoir utiliser Node.js et NPM pour du développement front-end en JavaScript

### Mise en situation

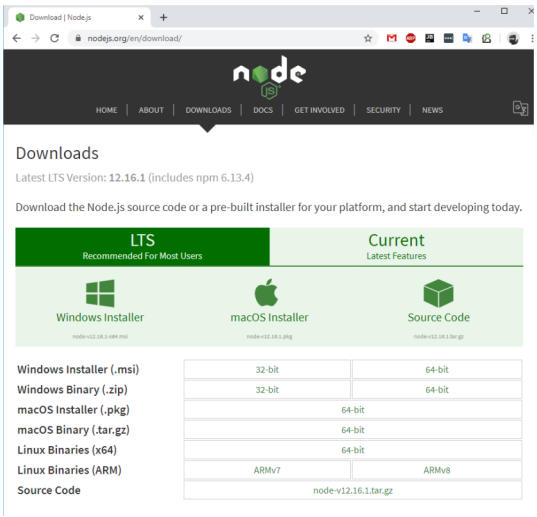
Pour utiliser les librairies de tests, nous allons devoir installer un environnement spécifique : **Node.JS** et **NPM**. Avant de passer aux tests, il est important de savoir comment utiliser cet environnement dans le cadre d'un développement front-end.

Node.js est une plateforme logicielle *open source*, qui s'appuie sur le moteur JavaScript V8, utilisé notamment sur Chrome, et qui permet d'exécuter du code JavaScript sur un serveur.

Il dispose également d'un gestionnaire de paquets appelé NPM, le plus grand écosystème de librairies *open source* du monde.

### Node.js

Pour installer Node.js, il suffit de se rendre sur le site du projet nodejs.org[1], et dans la section *Download* de choisir *Installer* (ou *Binary* pour Linux) selon l'OS utilisé, et de lancer l'installation.




Node.js propose deux versions :

- **Current** : dernière version majeure de Node.js, elle peut comprendre des *breaking changes*, et le support ne sera pas forcément assuré.

  Cette version est plutôt destinée aux développeurs voulant essayer les dernières nouveautés.

- **LTS** (*Long-Term Support*) : désigne une version dont le support technique sera assuré sur une longue période.

  C'est cette version qui devra être utilisée en production et dans un cycle de développement classique.

***Conseil***

Après l'installation, un redémarrage est souvent nécessaire.

Pour tester la bonne installation de Node.js, on peut par exemple écrire un programme JavaScript simple, et l'exécuter à partir de la console.

```
1 //Fichier prog.js
2 console.log('Mon programme executé avec Node.js');
```

```
1 cmd> node prog.js
2 Mon programme executé avec Node.js
```

[1] https://nodejs.org/en/download/

***Définition*** **NPM**

NPM est le gestionnaire de paquets officiel de Node.js :

- **Paquet** (*package*) : un paquet est ensemble de fonctions utilitaires regroupées et mises à disposition pour pouvoir être réutilisées ou exécutées sans avoir à réécrire le code.

  Chaque paquet a un numéro de version permettant de garantir la cohérence dans leur utilisation.

- **Gestionnaire de paquets** : un gestionnaire de paquets est un outil qui va permettre de télécharger, de mettre à jour et de supprimer des paquets référencés dans une application, et de gérer leurs dépendances.

  Si le paquet A référence le paquet B et que notre application référence le paquet A, alors NPM téléchargera le paquet A et le paquet B.

  NPM étant le gestionnaire de paquets officiel de Node.js, il est compris dans l'installation de Node.js.

Pour tester la bonne installation de l'exécutable NPM, on peut exécuter la commande suivante dans la console :

```
cmd> npm --version
6.5.0
```

***Méthode*** **Utilisation de NPM**

Pour gérer les différentes versions des paquets dans une application, NPM se base sur le fichier `package.json` présent à la racine de l'application. Si le fichier `package.json` n'existe pas, il faudra le créer à l'aide de l'exécutable NPM et de la commade `npm init`. Il suffit ensuite de suivre les instructions dans la console pour remplir la description du projet dans le `package.json`.

```
cmd> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (app-test)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to D:\app-test\package.json:

{
  "name": "app-test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

```
34
35
36 Is this ok? (yes)
```

Pour ajouter un paquet à l'application, il faudra demander à l'exécutable NPM de le télécharger et de l'installer, avec la commande `npm install <nom-du-paquet>`.

Les paquets seront ajoutés au dossier `node-modules` dans le dossier de l'application.

```
cmd> npm install cowsay
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN app-test@1.0.0 No repository field.

+ cowsay@1.4.0
added 10 packages in 4.585s
```

```
//Le fichier package.json
{
  "name": "app-test",
  "version": "1.0.0",
  "description": "my description",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cowsay": "^1.4.0"
  }
}
```

On pourra alors utiliser le paquet téléchargé dans notre application.

```
'use strict';

const {say} = require('cowsay');
const message = say({ text: 'Mon premier import de package !' });

console.log(message);
```

```
cmd> node prog.js
 _________________________________
< Mon premier import de package ! >
 ---------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

```

***Syntaxe*** **À retenir**

- Node.js est une plateforme d'exécution (*runtime*) qui permet d'exécuter du JavaScript sans passer par le navigateur.
- Il est fourni avec son gestionnaire de paquets NPM, qui permet de télécharger des paquets choisis dans la plus grande collection *open source* du monde et de gérer leurs versions.

***Complément***

Node.js[1]

NPM[2]

## VII. Exercice : Appliquez la notion

### Question

Cet exercice a pour but de vous familiariser avec l'environnement nécessaire à l'utilisation de Node.js.

1. Téléchargez et installez **Node.js**.
2. Créez un nouveau dossier et ouvrez l'invite de commande à partir de cet emplacement.
3. Initialisez un nouveau projet via la commande `npm init`.
4. Créez un fichier JavaScript et écrivez un script qui retournera *Hello world* en console.
5. Exécutez le script en console.

#### Indice :

Si vous utilisez Visual Studio Code, vous pouvez accéder au terminal intégré via le menu **view-terminal**. Il ouvrira le terminal à l'emplacement du dossier sélectionné.

## VIII. Les outils de tests JavaScript

#### Objectifs

- Connaître plusieurs outils de tests JavaScript
- Apprendre les bases de l'utilisation des outils de tests

#### Mise en situation

Pour écrire et exécuter des test unitaires ou d'intégration, l'écosystème JavaScript propose plusieurs outils avancés. Nous allons voir ici l'utilisation d'un de ces outils, Mocha, qui est représentatif de l'utilisation des outils les plus performants.

#### Mocha

Mocha est un framework de test JavaScript. C'est un outil simple, extensible et rapide qui peut être utilisé pour écrire des tests unitaires ou d'intégration.

Voyons comment l'utiliser.

Mocha est utile pour écrire et lancer des tests, mais cette fonctionnalité n'aura pas sa place en production. C'est pourquoi, lors de l'installation avec l'exécutable NPM, nous pourrons préciser `--save-dev`, qui enregistrera la référence dans la catégorie dépendances de développement du `package.json` et ignorera ce package lors d'un déploiement en production.

```
cmd> npm install --save-dev mocha
```

```
//Fichier à tester : math.service.js

module.exports = {
    //Fonction d'addition des 2 paramètres d'une fonction
```

[1] https://nodejs.org/en/
[2] https://www.npmjs.com/

```
    sum : function(firstValue, secondValue){
       return firstValue + secondValue;
    },

    //Fonction de multiplication des 2 paramètres d'une fonction
    multiply : function(firstValue, secondValue){
        //Pour l'exemple, cette fonction est boguée
        return firstValue + secondValue;
    }
};
```

```
//Fichier tests mocha : math.service.test.js

var assert = require('assert');
var mathFunctions = require('./math.service');

//On crée un groupe que l'on nomme Calculatrice,
//qui regroupera tous les tests sur la feature Calculatrice de l'application
describe('Calculatrice', function(){

    //On crée un sous-groupe que l'on nomme Operations,
    // qui regroupera les tests sur les
    // Additions, Multiplications, etc..
    describe('Operations', function(){
        it('Fonctions mathématiques - Sum OK', function(done) {
            //Arrange
            let firstParam = 1;
            let secondParam = 2;

            //Act
            let result = mathFunctions.sum(firstParam, secondParam);

            //Assert
            assert.equal(result, 3);
            done();
        });

        it('Fonctions mathématiques - Multiply OK', function(done) {
            //Arrange
            let firstParam = 3;
            let secondParam = 2;

            //Act
            let result = mathFunctions.multiply(firstParam, secondParam);

            //Assert
            assert.equal(result, 6);
            done();
        });
    });
});
```

- Arrange : On définit les données d'entrée de la méthode à tester.
- Act : On lance la méthode à tester avec les données définies.
- Assert : On vérifie que le résultat correspond aux attentes.

  Ici, on utilisera la librairie `Assert`, incluse avec Mocha, pour vérifier plus facilement que les valeurs retournées correspondent à ce que l'on attend.

L'utilisation d'autres packages d'assertion plus évolués est possible avec Mocha. Pour cela, il suffira des les installer avec l'exécutable NPM et de les importer dans le fichier de test de la même façon qu'`Assert` est importé ici (`var assert = require('assert');`).

Pour lancer les tests :

```
cmd> mocha math.service.test.js

  Calculatrice
    Operations
      √ Fonctions mathématiques - Sum OK
      1) Fonctions mathématiques - Multiply OK

  1 passing (22ms)
  1 failing

  1) Calculatrice
       Operations
         Fonctions mathématiques - Multiply OK:

      AssertionError [ERR_ASSERTION]: 5 == 6
      + expected - actual

      -5
      +6

      at Context.<anonymous> (math.service.test.js:33:20)
```

Le résultat du test nous indique que le test `Fonctions mathématiques - Sum OK` est valide, mais que le test `Fonctions mathématiques - Multiply OK` a échoué. Mocha nous informe sur le résultat attendu et sur la différence avec le résultat obtenu.

Mocha est un framework de test assez basique. La philosophie est que l'outil pourra être enrichi par l'ajout de paquets choisis par le développeur.

C'est pourquoi, dans un cas d'utilisation plus complexe, il sera nécessaire d'utiliser des paquets tels que :

- **Chai** : pour avoir des assertions plus avancées ou plus lisibles.
- **Sinon** : pour créer des `Fake Server` (serveurs fictifs) afin d'isoler les appels et de maîtriser les réponses dans le jeu de test.

***Complément*** **Les autres frameworks de tests**

**Jest** ou **Jasmine** sont des frameworks de tests concurrents de Mocha. Ils s'utilisent de façon très similaire, à la différence que Jest et Jasmine se revendiquent comme des frameworks complets, c'est-à-dire que le développeur n'aura pas besoin d'ajouter des paquets externes.

**Lequel choisir ?**

Comme toujours, cela va dépendre de vos besoins précis sur le projet. Si c'est un gros projet, qui nécessite d'avoir un outil parfaitement adaptable à chaque cas de figure, Mocha sera probablement le plus adapté.

Si, au contraire, le projet est de taille plus petite, Jest ou Jasmine éviteront une importante phase de configuration de l'outil de test.

***Syntaxe*** **À retenir**

- Le très riche écosystème de JavaScript nous offre plusieurs paquets aboutis pour écrire et lancer des test unitaires, d'intégration ou fonctionnels.
- Ces outils devront être choisis par le développeur au démarrage du projet, en fonction des besoins à venir.

***Complément***

Mocha[1]

Jest[2]

Jasmine[3]

Chai[4]

Sinon[5]

## IX. Exercice : Appliquez la notion

### Question

Dans l'exercice précédent, nous avons installé un environnement Node.js. Nous allons à présent implémenter un premier test simple sur une fonction JavaScript.

Dans le dossier créé précédemment :

- Créez un fichier `name.function.js`.
- Créez un fichier `test.js`.
- Copiez-collez le code ci-dessous dans le fichier `name.function.js`.
- Installez la librairie Mocha via NPM : https://mochajs.org/.

```
module.exports = {
  formatName : (name) => {
    const names = name.split('-');
    const firstLetter = name.substring(0, 1).toUpperCase();
    const otherLetter = name.substring(1).toLowerCase();
    return `${firstLetter}${otherLetter}`;
  }
}
```

Dans le fichier `test.js` et en utilisant Mocha, testez que la première lettre de 'Pierre' soit 'P'.

## X. Auto-évaluation

### A. Exercice final

#### Exercice

Exercice

Quel principe ne fait pas partie du cycle de développement d'un test unitaire ?

[1] https://mochajs.org/
[2] https://jestjs.io/
[3] https://jasmine.github.io/
[4] https://www.chaijs.com/
[5] https://sinonjs.org/

- ○ Écrire le test
- ○ Lier le test à la base de données
- ○ Jouer tous les tests
- ○ Écrire le code de production

Exercice

Parmi ces types de tests, lesquels existent ?

- ☐ Tests fonctionnels
- ☐ Tests unitaires
- ☐ Tests d'intégration
- ☐ Tests de variabilité

Exercice

Quelle est la plateforme permettant d'exécuter du code JavaScript sur un serveur ?

Exercice

Quel est le gestionnaire de paquets officiel de Node.js ?

Exercice

Quelle est la commande permettant d'ajouter un paquet à l'application ?

- ○ npm <nom du paquet>
- ○ npm run <nom du paquet>
- ○ npm start <nom du paquet>
- ○ npm install<nom du paquet>

Exercice

Quel fichier à la source d'une serveur Node permet de voir en un coup d'œil les paquets installés ?

Exercice

Parmi ces réponses, lesquelles sont des frameworks de tests ?

- ☐ Jest
- ☐ Jasmine
- ☐ Mocha
- ☐ Trunk

Exercice

Quelle fonction JavaScript native permet d'importer une librairie dans un script Node.js ?

Exercice

Quelle instruction faut-il taper en invite en commande pour voir la version de NPM installée sur votre ordinateur ?

Exercice

Quelle instruction faut-il ajouter lorsqu'on installe un paquet via NPM pour que celui-ci ne soit pas pris en compte en production ?

## B. Exercice : Défi

On vous demande d'implémenter des tests unitaires sur une fonction d'un logiciel de caisse, qui calcule le montant de monnaie à rendre.

Le résultat pour un achat de 22,30 € et un paiement de 30 € sera retourné sous le format :

```
/*
[
{value: 5, number: 1},
{value: 2, number: 1},
{value: 0.5, number: 1},
{value: 0.2, number: 1}
]
*/
// => 1 billet de 5€, 1 piece de 2€, 1 piece de 0.5€ et 1 piece de 0.2€ soit un total de 7.70€
```

### Question

Implémentez deux tests unitaires sur la fonction `giveMoney()`.

Pour réaliser cet exercice, vous devrez partir du dossier créé :

- Créez deux fichiers : **money.js** dans lequel vous copierez le code ci-dessous, et **test.js** dans lequel vous écrirez vos tests unitaires.
- Installez Mocha.
- Dans le fichier **package.json**, remplacez `"test": "echo \"Error: no test specified\" && exit 1"` par `"test": "mocha"`.
- Premier test : lorsque le montant donné par le client est supérieur au prix d'achat.
- Deuxième test : lorsque le montant donné par le client est inférieur au prix d'achat.

Pour lancer le test, il faudra taper la commande `npm test`.

```
// le parametre price représente le montant des achats du client
// le parametre mount représente la somme donnée par le client

module.exports = {
   giveMoney : (price, mount) => {
    if (mount > price) {
      // les différentes valeurs possibles
      const typeOfMoney = [500, 200, 100, 50, 20, 10, 5, 2, 1, 0.5, 0.2, 0.1, 0.05, 0.02,
0.01];
      // le tableau qui sera retourné
      const render = []
      // calcul de la différence entre le montant donné par le client et le montant des achats
      let MoneyToGive = (mount*100 - price*100)/100;
      // On boucle sur le tableau des valeurs
      typeOfMoney.forEach((money => {
```

```
        // on calcul le quotient entre le montant à rendre et chaque valeurs
        const reste = MoneyToGive/money
        // si le quotient est supérieur ou égale à 1 alors
        if (reste >= 1) {
          // On crée un objet avec la valeur et le reste arrondi à l'entier inférieur
          const tempory = {value: money, number: Math.floor(reste)}
          // On ajoute cet objet dans le tableau final
          render.push(tempory)
          // On recalcul le nouveau montant à rendre
          MoneyToGive = (MoneyToGive*100 - money*tempory.number*100)/100
        }
      }))
      return render;
    } else {
      return `Il manque ${(price*1000 - mount*1000)/1000}€`
    }
  }
}


```

**Indice :**

Mocha teste l'égalité sur des types primitifs : cherchez comment tester l'égalité sur des objets.

Les opérations sont multipliées par 100, puis divisées par 100 à cause de la gestion des décimales en JavaScript.

## Solutions des exercices

### Exercice p. Solution n°1

En regardant le calendrier, on sait que cette date correspond à un vendredi. On doit donc tester que la fonction retourne *vendredi* en fonction de la date.

```
const test = () => {
    const act = [15, 4, 2020];

    if (getDayOfDate(...act) === 'Vendredi') {
      return true;
    } else {
      return false;
    }
}

console.log(test());
```

### Exercice p. Solution n°2

On va créer deux tests unitaires :

```
const testWithNumber = () => {
    const act = 2;

    if (typeof getArticleNumber(act) === 'number') {
      return true;
    } else {
      return false;
    }
}

const testWithString = () => {
    const act = '2';

    if (typeof getArticleNumber(act) === 'number') {
      return true;
    } else {
      return false;
    }
}

console.log(testWithNumber()); // true
console.log(testWithString()); // false
```

`testWithString()` retourne *false*, il faut donc modifier la fonction `getArticleNumber()`.

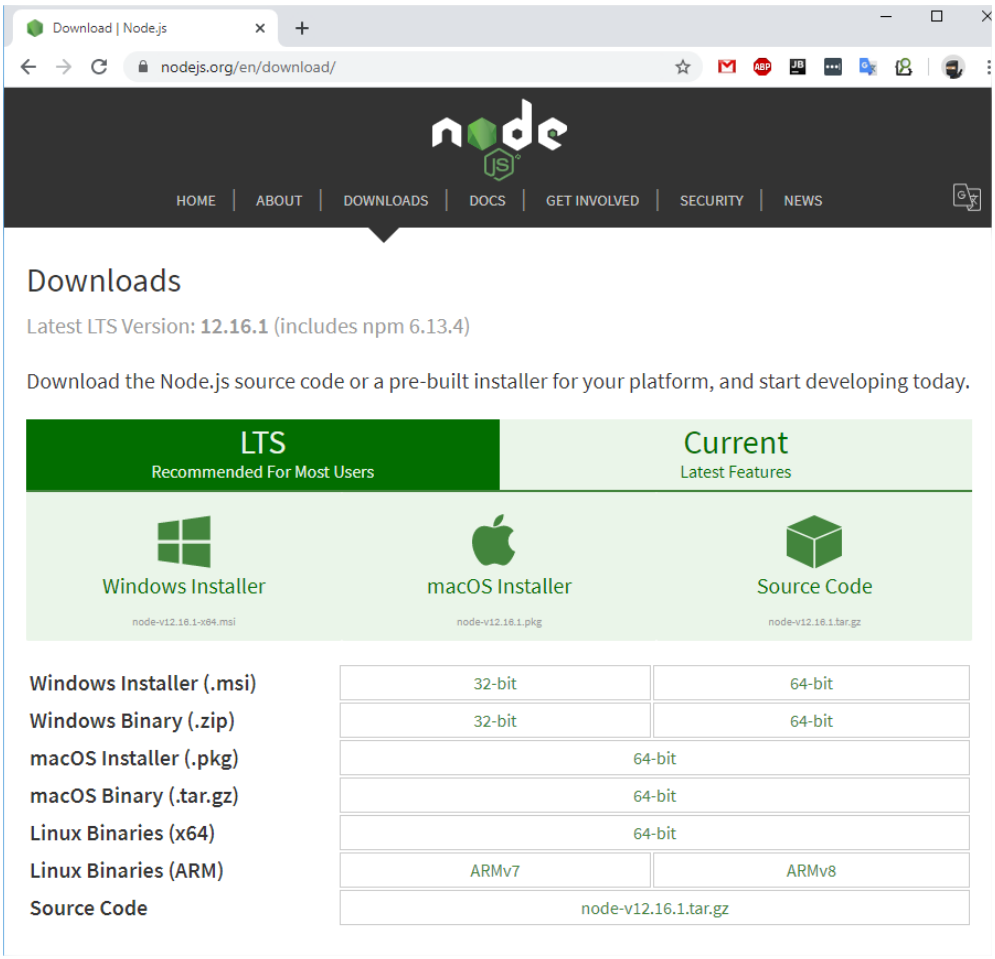
```
const getArticleNumber = (numberOfArticles) => {
   return Number(numberOfArticles);
}


console.log(testWithNumber()); // true
console.log(testWithString()); // true
```

### Exercice p. Solution n°3

Le test fonctionnel doit tester une fonctionnalité dans son ensemble. Ici, on testera que le panier a correctement été mis à jour dans la base de données.

### Exercice p. Solution n°4

Étape 1 : rendez-vous sur le site de Node.js[1] et téléchargez la version `LTS`, puis installez Node.

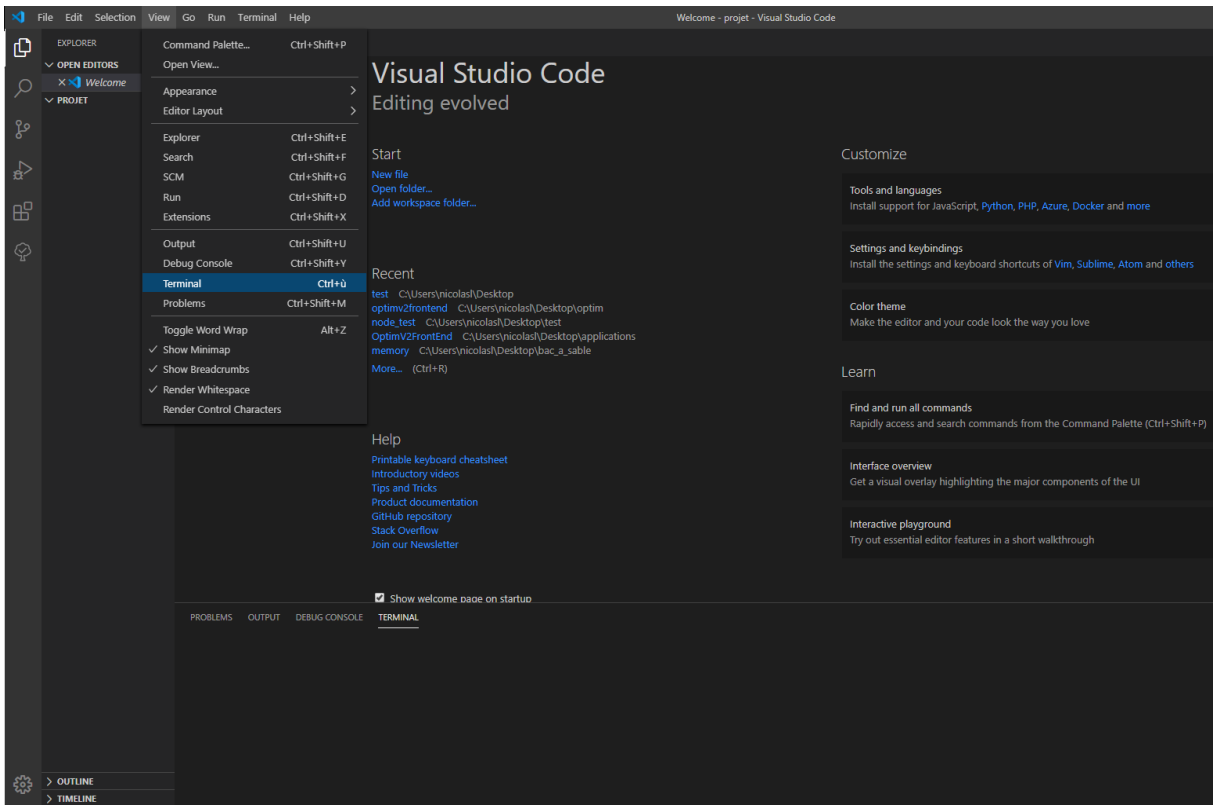



Étape 2 : créez un nouveau dossier et nommez-le à votre convenance. Si vous utilisez Visual Studio Code, ouvrez le dossier à partir de l'éditeur. Puis allez dans le menu **vue -> terminal**.

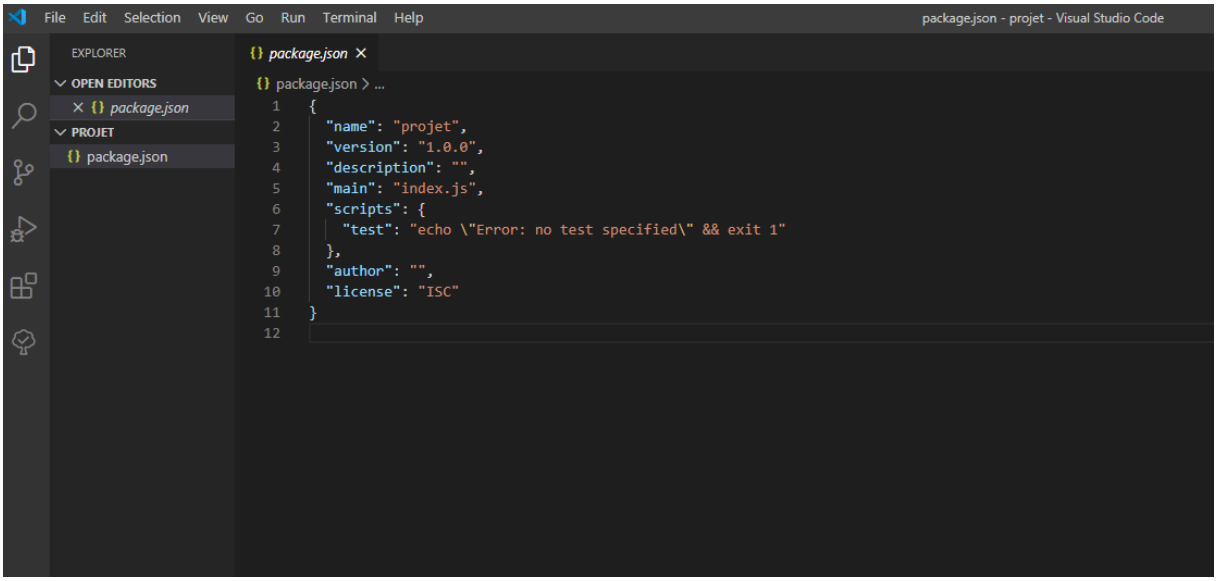
Si vous n'utilisez pas Visual Studio Code :

- Si vous êtes sous Windows, pour ouvrir un terminal de commande à partir du dossier, ouvrez le dossier et faites **Shift + click droit**. Sélectionnez **Ouvrir la fenêtre PowerShell ici**
- Si vous êtes sur Linux, en fonction de votre distribution, ouvrez le terminal avec le raccourci Ctrl+Alt+T ou alternativement, retrouvez l'application Terminal dans la liste des applications
- Si vous êtes sur Mac, ouvrez Spotlight et cherchez "terminal", alternativement, retrouvez l'application Terminal dans la liste des applications

[1] https://nodejs.org/en/

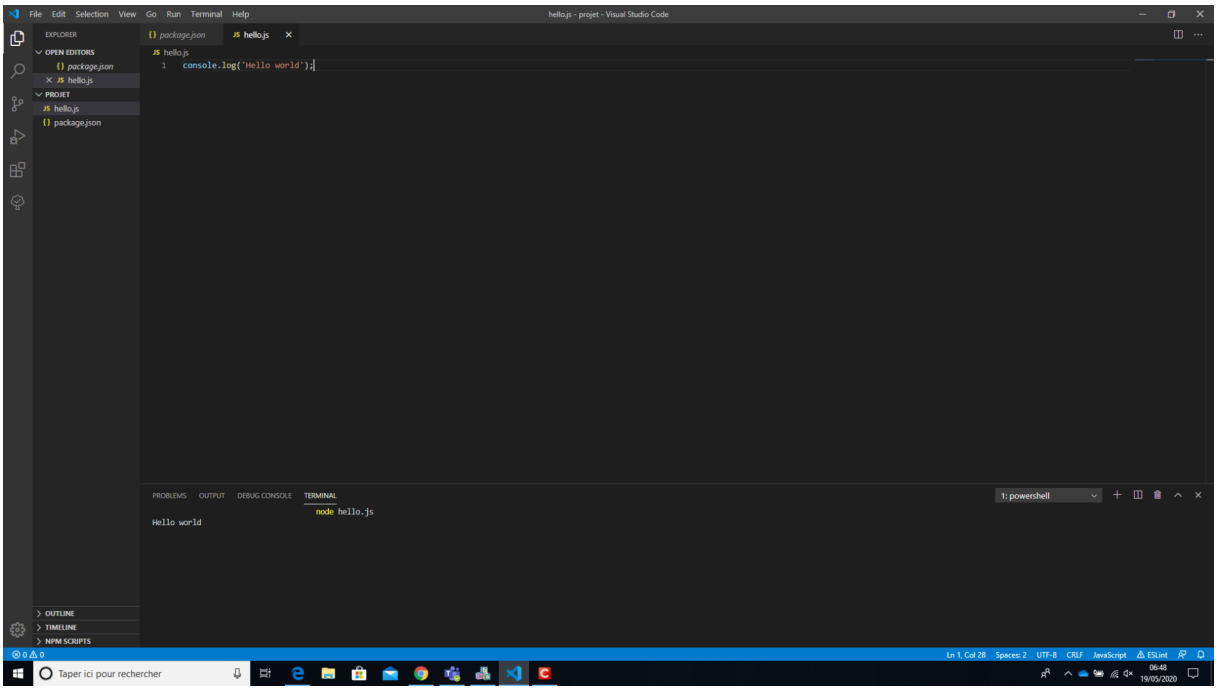



Étape 3 : à partir du terminal, tapez la commande `npm init` et suivez les instructions. Vous devriez alors avoir un fichier `package.json` qui est apparu.




```
1 console.log('Hello world');
```

```
1 node hello.js
```




### Exercice p. Solution n°5

```
const nameFunction = require('./name.function');
var assert = require('assert');

// On écrit notre fonction de test
describe('NameFunction', () => {
  describe('Render', () => {
    // Test lorsque le montant donné par le client est supérieur au prix d'achat
    it('first letter is uppercase', function(done) {
      //Arrange
      const name = 'pierre'

      // Act
      const result = nameFunction.formatName(name);

      // Assert
      assert.equal(result.substring(0, 1), 'P');
      done();
    })
  })
})
```

### Exercice p. 13 Solution n°6

Exercice

Quel principe ne fait pas partie du cycle de développement d'un test unitaire ?

- ○ Écrire le test
- ◉ Lier le test à la base de données
- ○ Jouer tous les tests

- ○ Écrire le code de production

Exercice

Parmi ces types de tests, lesquels existent ?

- ☑ Tests fonctionnels
- ☑ Tests unitaires
- ☑ Tests d'intégration
- ☐ Tests de variabilité

Exercice

Quelle est la plateforme permettant d'exécuter du code JavaScript sur un serveur ?

Node.js
Exercice

Quel est le gestionnaire de paquets officiel de Node.js ?

npm
Exercice

Quelle est la commande permettant d'ajouter un paquet à l'application ?

- ○ npm <nom du paquet>
- ○ npm run <nom du paquet>
- ○ npm start <nom du paquet>
- ◉ npm install<nom du paquet>

Exercice

Quel fichier à la source d'une serveur Node permet de voir en un coup d'œil les paquets installés ?

package.json
Exercice

Parmi ces réponses, lesquelles sont des frameworks de tests ?

- ☑ Jest
- ☑ Jasmine
- ☑ Mocha
- ☐ Trunk

Exercice

Quelle fonction JavaScript native permet d'importer une librairie dans un script Node.js ?

require()
Exercice

Quelle instruction faut-il taper en invite en commande pour voir la version de NPM installée sur votre ordinateur ?

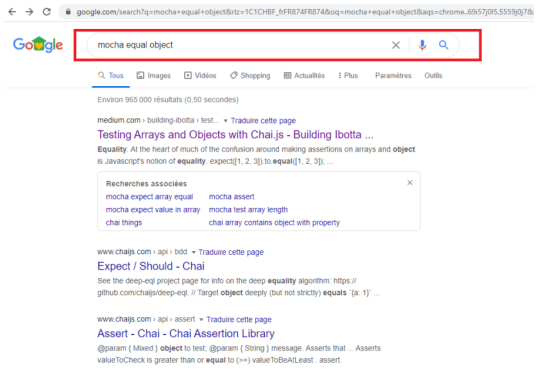
npm --version
Exercice

Quelle instruction faut-il ajouter lorsqu'on installe un paquet via NPM pour que celui-ci ne soit pas pris en compte en production ?

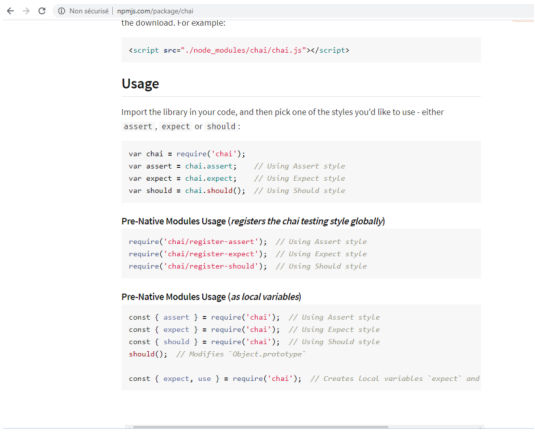
-dev

**Exercice p. Solution n°7**

La première problématique est de trouver comment comparer deux objets. Pour cela, une recherche rapide suffit. Le premier résultat est un article qui semble donner la réponse souhaitée. Effectivement, il explique comment tester une égalité avec Chai, mais ne nous explique pas comment installer les méthodes de la librairie dans notre script.




Pour cela, une petite recherche supplémentaire s'impose, avec les mots-clés `chai npm`.




```
// On importe le module contenant la fonction giveMoney()
const moneyFunctions = require('./money');
// On importe les méthodes de la libraire chai afin de pouvoir comparer des objets
const chai = require('chai');
const expect = chai.expect;    // Using Expect style
const assert = chai.assert;  // Using Assert style

// On écrit notre fonction de test
describe('Money', function() {
  describe('Render', function() {
    // Test lorsque le montant donné par le client est supérieur au prix d'achat
    it('fonction giveMoney OK if price < mount', function(done) {
      //Arrange
      const price = 22.30;
      const mount = 30;

      // Act
      const result = moneyFunctions.giveMoney(price, mount);

      // Assert
      expect(result).to.eql([{value: 5, number: 1},{value: 2, number: 1},{value: 0.5, number: 1},{value: 0.2, number: 1}])
      done();
    })
    // Test lorsque le montant donné par le client est inférieur au prix d'achat
    it('fonction giveMoney OK if price > mount', function(done) {
```

```
26        //Arrange
27        const price = 30;
28        const mount = 20;
29
30        // Act
31        const result = moneyFunctions.giveMoney(price, mount);
32
33        // Assert
34        expect(result).to.eql(`Il manque ${(price*1000 - mount*1000)/1000}€`)
35        done();
36      })
37    })
38  })
39
```