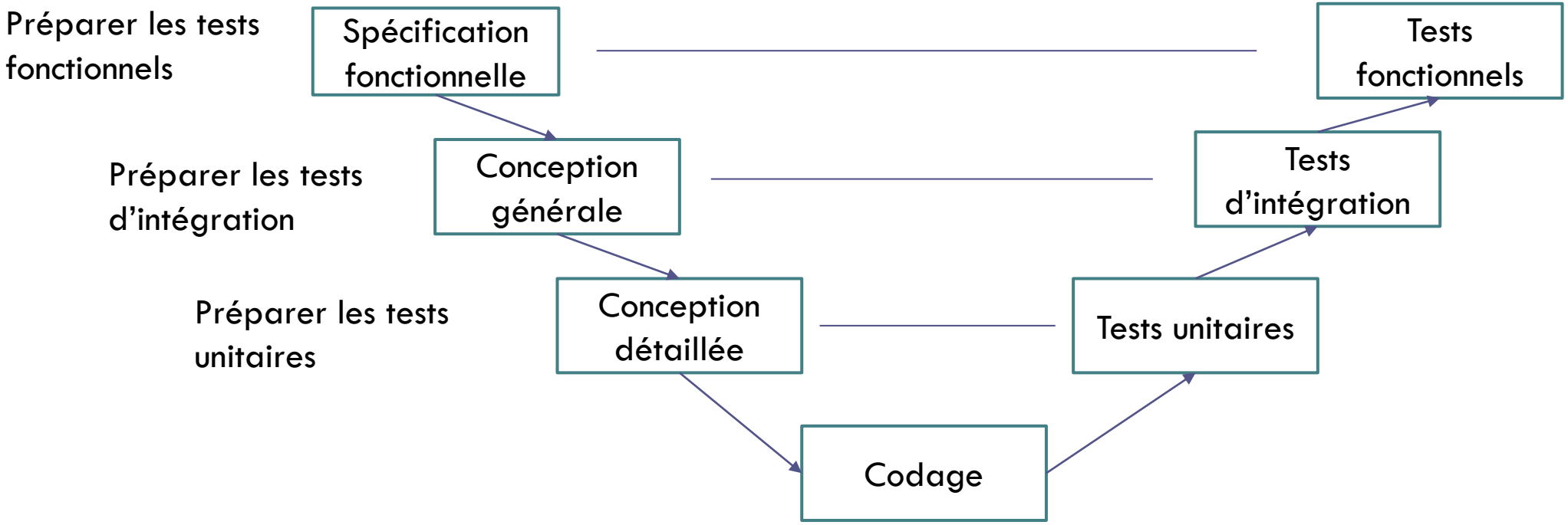
Qualité

Plan

- Tests : vérification et validation
- Analyse du code
- Bonnes pratiques

Vérification et validation

Tests



Vérification et validation

Tests

- Il existe plusieurs types de test
  - Tests unitaires :
    - Tester (vérifier) chaque composant (fonctions, méthodes, objets, etc.) indépendamment des autres
  - Test d'intégration :
    - Ces tests sont exécutées pour valider l'intégration des différents modules entre eux et dans leur environnement exploitation définitif.
    - Ils permettront de mettre en évidence des problèmes d'interfaces entre différents modules.
  - Tests fonctionnels
    - Tests qui servent à vérifier que les fonctionnalités développées correspondent au cahier des charges

Tests unitaires

- **Couverture de code** : est une mesure utilisée pour décrire le taux de code source exécuté d'un programme quand une suite de test est lancée [Wikipédia]
  - Attention un code peut être couvert mais mal testé. Il est intéressant de surveiller l'évolution /historique de ce taux et vérifiez qu'il chute pas brusquement.
- Objectifs des tests unitaires
  - Localiser les sources des erreurs
  - Augmenter la couverture de code (instructions, branchements, appels aux fonctions, …)
- Outils : Jest, Jasmine, Mocha

Test : qualité

- Adopter une stratégie tel que le développement piloté par les test
  - Concevoir les scénarios de test durant la phase de conception bien avant de commencer à coder
- Automatiser les tests => répétitifs, plusieurs itérations
- Rapide et facile à lancer (configuration, dépendances, etc.)
- Reproductible et fiable (non aléatoire)
- Facile à interpréter les échecs et repérer les erreurs
  - Tests unitaire vs tests d'intégration

Rapport de couverture de code

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	21.14	7.69	20.59	21.14	
login-api/src	62.5	50	66.67	62.5	
app.ts	0	100	0	0	25-46
config.ts	80	100	66.67	80	34-41
context.ts	90.91	50	100	90.91	59
environment.ts	0	0	0	0	
logging.ts	100	100	100	100	
runtime.ts	0	0	0	0	
login-api/src/lambda	23.53	12.5	25	23.53	
index.ts	25	16.67	100	25	18-30
serverless.ts	22.22	0	0	22.22	19-54
login-api/src/middleware	0	0	0	0	
context.ts	0	0	0	0	7-20

- Code de couleurs (rouge, jaune, vert)
- Plusieurs métriques
  - Par nombre d'instructions, de branchements, de fonctions et de lignes testés
- Lignes non testées

Environnement de test

- Il existe plusieurs outils de test automatisés pour Javascript
  - Jasmin
  - Mocha
  - Jest
  - Etc.

JEST

- Installation :
  - >npm install -- save-dev jest
- Configuration (Package.json)
- Lancer le test :
  - >npm run test

```
{
  "name": "myapp",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www",
    "test": "jest"
  },
  "jest": {
    "collectCoverage":true
    },
  "dependencies": {
.....
```

Créer une suite de tests

- Crée un dossier « test/ » ?
- Créer un fichier *.test.js * : nom de votre suite de tests
- Il existe trois méthodes principales dans un fichier de test :
  - describe() - C'est une suite de scripts de test qui donne une description externe pour la suite de tests.
  - test() - C'est le plus petit cas de test unitaire écrit pour être exécuté. La chaîne entre guillemets représente le nom du test.
  - expect() - C'est une assertion. Chaque instruction test() a une fonction expect() qui prend une valeur et attend un retour sous forme vraie.

- Argument « done » : Jest attendra que le callback done soit appelé avant de terminer le test.

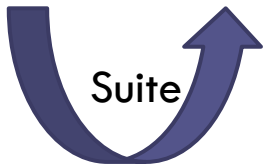
```
const DB= require ("../model/db.js");
const model= require ("../model/user.js");
describe("Model Tests", () => {

    beforeAll(() => {
        // des instructions à exécuter avant le lancement de cette suite de tests
    });

    afterAll((done) => {
        function callback (err){
            if (err) done (err);
            else done();

        }
        DB.end(callback);

    });

    test ("read user",()=>{
       nom=null;
       function cbRead(resultat){
        nom = resultat[0].nom;
        expect(nom).toBe("test");

       }

         model.read("test@test.fr",  cbRead);

    });

})
```

Les Matchers Jest
Egalité


```
test("equality matchers", () => {

  expect(2*2).toBe(4);

  expect(4-2).not.toBe(1);

})
```

Les Matchers Jest
Comparaison booléenne


```
test("truthy operators", () => {

   var name= "une simple chaine";
   var n = null;

   expect(n).toBeNull();
   expect(name).not.toBeNull;

   // name has a valid value

   expect(name).toBeTruthy();

   //fail - as null is non success

   expect(n).toBeTruthy();

   // pass - null treated as false or negative

    expect(n).toBeFalsy()

    // 0 - treated as false

    expect(0).toBeFalsy()
})
```



Les Matchers Jest
Comparaison entre nombres


```
test("numeric operators", () => {
  var num1 = 100;
  var num2 = -20;
  var num3 = 0;
  // greater than
  expect(num1).toBeGreaterThan(10);

  // less than or equal

  expect(num2).toBeLessThanOrEqual(0);

  // greater than or equal

  expect(num3).toBeGreaterThanOrEqual(0);
 })
```

Les Matchers Jest

Expressions régulières sur les chaines de caractères


```
test("string matchers",() => {

  var string1 = "une simple chaine"

  // test for success match

  expect(string1).toMatch(/test/);

  // test for failure match

  expect(string1).not.toMatch(/abc/)

 })
```

Tests sur des routes HTTP

- Pour préparer les entrées d'une route http on utilise le module « supertest »
- Installation : npm install -- save-dev supertest

```
const request = require("supertest");
const app = require("../app");

describe("Test the root path", () => {
  test("It should response the GET method", done => {
    request(app)
      .get("/")
      .then(response => {
        expect(response.statusCode).toBe(200);
        done();
      });
  });
});
```

Tests sur des routes HTTP

- Plusieurs types de tests http :
  - Le code http retourné (200, 404, …)
    - expect(200)
  - Le type de contenu retourné
    - expect("Content-Type", /json/)
  - Le contenu
    - json : expect((res) => {
    - res.body.data.length = 1;
    - res.body.data[0].email = "test@example.com";
    - })

Tests sur des routes HTTP

- Faire un « post », envoyer des données et faire des vérifications
  - request(app)
  - .post("/send")
  - .expect("Content-Type", /json/)
  - .send({
  - email: "francisco@example.com",
  - })
  - .expect(201)
  - Etc.

Test logique vs test présentation

- Test de logique
  - Les tests logiques exécuteront des tests unitaires et d'intégration par rapport à notre domaine logique. Ce sera tester uniquement JavaScript, déconnecté de toute fonctionnalité de présentation

- Test de présentation

  - Test de page
    - Comme son nom l'indique, teste la présentation et la fonctionnalité frontale d'une page. Cela peut impliquer à la fois des tests unitaires et des tests d'intégration.
    - Outils : mocha, Jest, …

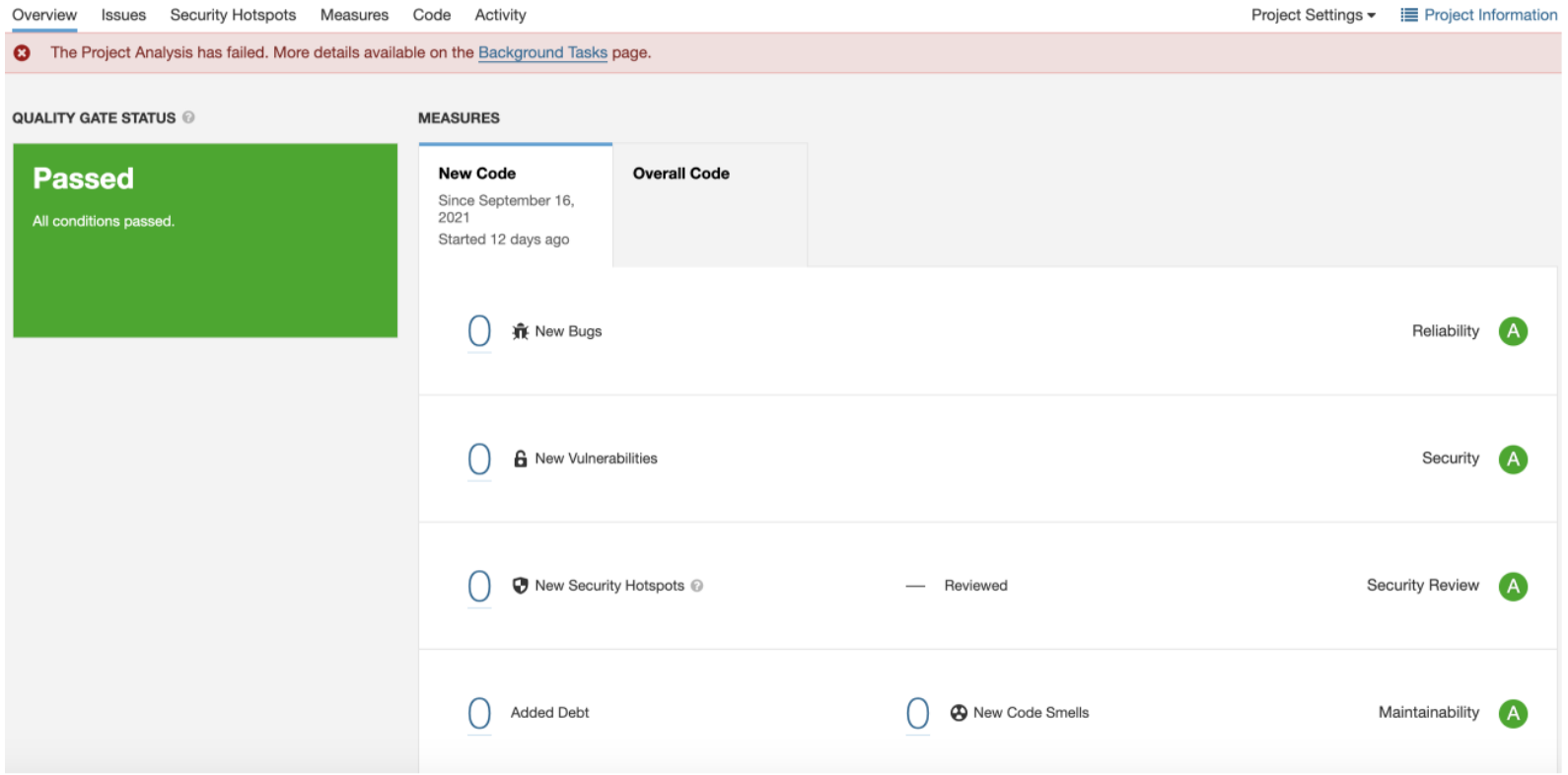
  - Tests inter-pages
    - Les tests inter-pages impliquent de tester des fonctionnalités qui nécessitent une navigation à partir d'un page à une autre. Par exemple, le processus de paiement sur un site de commerce électronique s'étend sur plusieurs pages. Étant donné que ce type de test implique intrinsèquement plus d'un composant, il est généralement considéré comme un test d'intégration.
    - Tools : Zombie.js

Analyse du code

- Analyse du code (Linting)
  - Cet étape ne consiste pas à trouver des erreurs (le code n'est pas exécuté), mais (analyser) des erreurs potentielles. Le concept général est d'identifie les zones qui pourraient représenter des erreurs possibles ou des constructions fragiles cela pourrait conduire à des erreurs dans le futur.
  - Outils : ESLint , JSHint, SonarLint, SonarQube
  - Plugins selon la nature de votre projet

- Vérifications des liens
  - s'assurer qu'il n'y a pas de liens brisés sur votre site
  - Tools : LinkChecker

Analyse du code

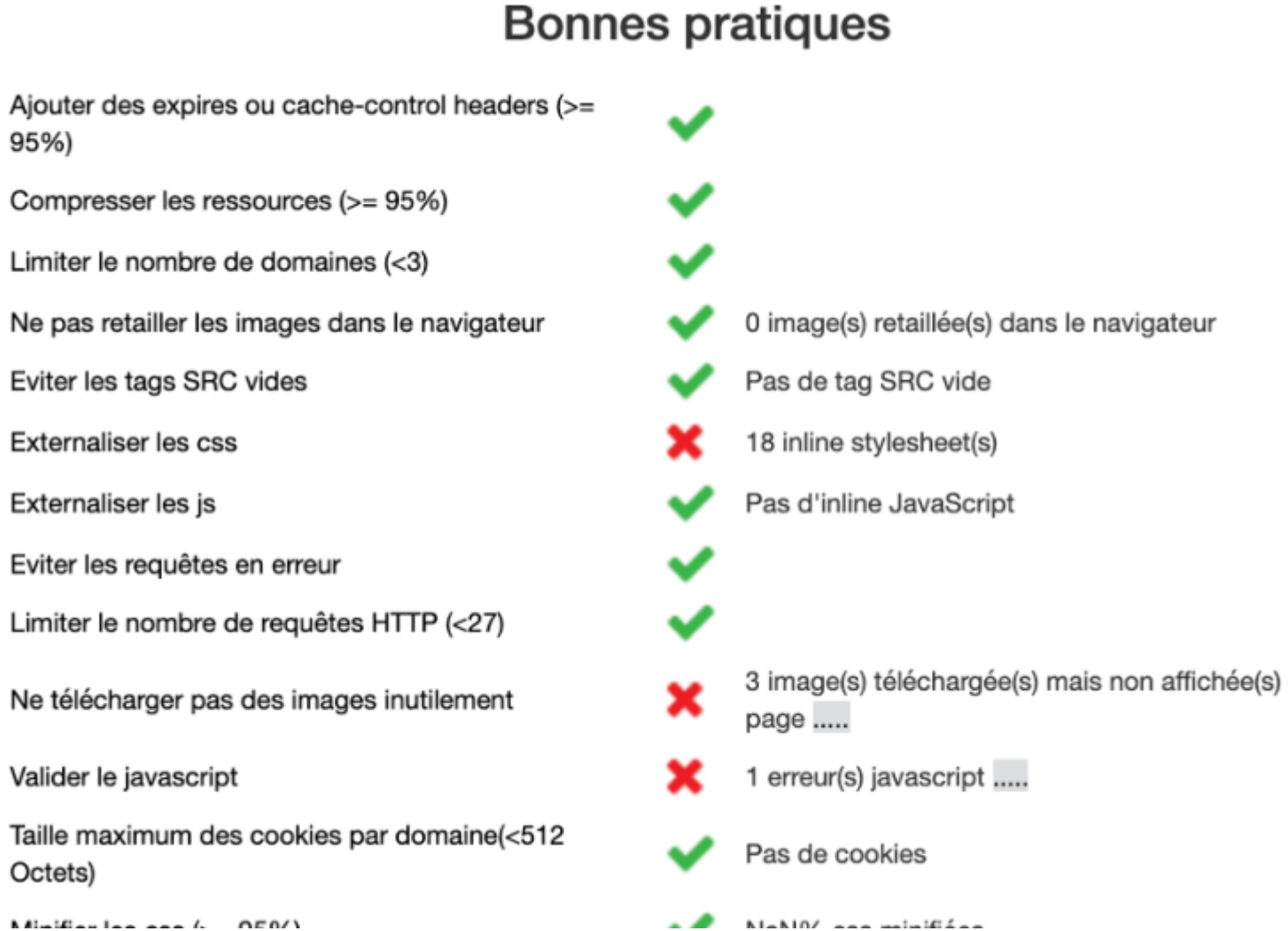


Aperçu SonarQube en local

Analyse du code
Eco-Index & éco-conception web

- L'extension Green IT : vérifier qu'un site respecte les bonnes pratiques d'éco conception web

- https://collectif.greenit.fr/ecoconception-web/115-bonnes-pratiques-eco-conception_web.html



Bonnes pratiques

Ajouter des expires ou cache-control headers (>= 95%)	✔	
Compresser les ressources (>= 95%)	✔	
Limiter le nombre de domaines (<3)	✔	
Ne pas retailler les images dans le navigateur	✔	0 image(s) retaillée(s) dans le navigateur
Eviter les tags SRC vides	✔	Pas de tag SRC vide
Externaliser les css	✖	18 inline stylesheet(s)
Externaliser les js	✔	Pas d'inline JavaScript
Eviter les requêtes en erreur	✔	
Limiter le nombre de requêtes HTTP (<27)	✔	
Ne télécharger pas des images inutilement	✖	3 image(s) téléchargée(s) mais non affichée(s) page .....
Valider le javascript	✖	1 erreur(s) javascript .....
Taille maximum des cookies par domaine(<512 Octets)	✔	Pas de cookies
Minifier les css (< 95%)	✔	NaN% css minifiées

Analyse du code
Eco-Index & éco-conception web

Minifier les js (>= 95%)	✗	0% js minifié .....
Pas de cookie pour les ressources statiques	✓	Aucun cookie
Eviter les redirections	✓	
Optimiser les images bitmap	✓	Pas d'images bitmap à optimiser
Optimiser les images svg	✓	Pas de svg à optimiser
Ne pas utiliser de plugins	✓	Aucun plugin
Fournir une print css	✗	Pas de print css
N'utilisez pas les boutons standards des réseaux sociaux	✓	Pas de bouton standard de réseau social trouv
Limiter le nombre de fichiers css (<3)	✓	Pas plus de 2 fichiers css
Utiliser des ETags (>= 95%)	✓	
Utiliser des polices de caractères standards	✓	Pas de polices de caractères spécifiques

- Les 115 bonnes pratiques :
  - 6 catégories : Spécification, conception, réalisation, production, utilisation et Support / maintenance / fin de vie
  - Livre : https://github.com/cnumr/best-practices/

Bonnes pratiques Node

Source : https://twitter.com/nodepractices/

- Structure de projet
- Gestion des erreurs
- Style du code
- Tests et pratiques générales de qualité
- Pratiques de mise en production
- Sécurité
- Performance
- Pratiques de Docker

Bonnes pratiques
Structure de projet

- Organisez votre projet en composants (dossier par composant : models, routes, tests, etc.)
  - Faciliter la maintenance

- Organisez vos composants en strates, gardez la couche web à l'intérieur de son périmètre
  - Couche service, couche accès aux données

Bonnes pratiques
Structure de projet

- Externalisez les utilitaires communs en paquets NPM
  - Regrouper certains vos codes et exposés-les en tant que paquets NPM privé
  - Créer vos propres dépendances
  - Réutilisation de ces dépendances dans plusieurs de vos projets
- Séparez Express 'app' et 'server' comme on le fait déjà :
  - Dossier ‘www’
  - Fichier ‘app.js’

Bonnes pratiques
Structure de projet

- Utilisez une configuration respectueuse de l'environnement, sécurisée et hiérarchique
  - garantir que
    - (a) les clés peuvent être lues depuis un fichier ET à partir de la variable d'environnement
    - (b) les secrets sont conservés hors du code source
    - (c) la configuration est hiérarchique pour une recherche plus simple.
  - Certains paquets peuvent gérer la plupart de ces points comme rc, nconf, config et convict.

Bonnes pratiques
Gestion des erreurs

- Utilisez Async-Await ou les promesses pour le traitement des erreurs asynchrones
- Utilisez uniquement l'objet intégré Error

```
if(!productToAdd)
    throw new Error('Comment puis-je ajouter un nouveau produit lorsqu\'aucune valeur n\'est fournie ?');
```

```
if(!productToAdd)
    throw ('Comment puis-je ajouter un nouveau produit lorsqu\'aucune valeur n\'est fournie ?');
```

- Distinguez les erreurs opérationnelles des erreurs de programmation
  - Erreur sur les entrées d'une API
  - Erreur d'une variable indéfinie => inconnue
- Etc.

Bonnes pratiques
Style du code

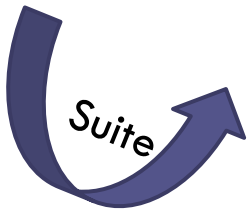
- Utilisez un linter comme « Eslint » couvrant JavaScript vanilla
- Plugins spécifiques à Node.js
  - eslint-plugin-node
- Commencez les accolades d'un bloc de code sur la même ligne
  - Exemple

```
// À faire
function someFunction() {
  // bloc de code
}

// À éviter
function someFunction
{
  // bloc de code
}
```

Bonnes pratiques
Style du code

- Séparez correctement vos instructions

```
// À faire

function doThing() {
    // …
}

doThing()

// À faire

const items = [1, 2, 3]
items.forEach(console.log)
```




```
// À éviter — lève une exception
const m = new Map()
const a = [1,2,3]
[...m.values()].forEach(console.log)
> [...m.values()].forEach(console.log)
>  ^^^
> SyntaxError: Unexpected token ...

// À éviter — lève une exception
const count = 2 // il essaie d'exécuter 2(), mais 2 n'est pas une fonction
(function doSomething() {
  // faire quelque chose d'incroyable
})()
/* placez un point-virgule avant la fonction immédiatement invoquée, après la définition de const, enregistrez la valeur de retour de la fonction anonyme dans une variable ou évitez tous les IIFE */
```

Bonnes pratiques
Style du code

- Nommez vos fonctions
  - Nommez toutes les fonctions, y compris les fermetures (closures, NdT) et les fonctions de rappel. Évitez les fonctions anonymes
- Utilisez des conventions de nommage pour les variables, les constantes, les fonctions et les classes
  - Utilisez LowerCamelCase lorsque vous nommez des constantes, des variables et des fonctions
  - Utilisez UpperCamelCase (première lettre en majuscule également) lorsque vous nommez des classes
  - Utilisez des noms évocateurs, mais efforcez-vous de les garder concis.

Bonnes pratiques
Style du code

- Préférez const à let. Laissez tomber le var
  - cont => ne pas utiliser la même variable pour différentes utilisations => code clair
  - Let : si une variable doit être réaffectée
  - var à ne pas utiliser en ES6
- Utilisez en premier require pour les modules, pas dans des fonctions internes
  - au début de chaque fichier require pour les modules, avant et en dehors de toute fonction

Bonnes pratiques
Style du code

- Utilisez l'opérateur === (strict)
- Utilisez Async Await, évitez les fonctions de rappel
- Utiliser les expressions de fonction fléchée (=>)

Bonnes pratiques
Tests et pratiques générales de qualité

- Au minimum, écrivez des tests API (pour chaque composant)
  - Postman, jest (avec supertest)

- Incluez 3 parties dans chaque nom de test
  - Indiquez dans le nom du test ce qui est testé (élément du test), dans quelles circonstances et quel est le résultat attendu.

```
//1. unité testée
describe('Service Produits', () => {
  describe('Ajoute un nouveau produit', () => {
    //2. scénario et 3. attente
    it('Quand aucun prix n\'est spécifié, alors le statut du produit est en attente d\'approbation', () => {
      const newProduct = new ProductService().add(...);
      expect(newProduct.status).to.equal('validationEnAttente');
    });
  });
});
```

Bonnes pratiques
Tests et pratiques générales de qualité

- Structurez vos tests avec le format AAA

```
describe.skip('Classification des clients', () => {
    test('Lorsque le client a dépensé plus de 500 $, il doit être classé comme premium', () => {
        //Arrange (Préparer)
        const customerToClassify = {spent:505, joined: new Date(), id:1}
        const DBStub = sinon.stub(dataAccess, 'getCustomer')
            .reply({id:1, classification: 'ordinaire'});

        //Act (Agir)
        const receivedClassification = customerClassifier.classifyCustomer(customerToClassify);

        //Assert (Vérifier)
        expect(receivedClassification).toMatch('premium');
    });
});
```

Bonnes pratiques
Tests et pratiques générales de qualité

- Détectez les problèmes de code avec un linter
- Évitez les tests globaux, ajoutez des données pour chaque test
- Inspectez en permanence les dépendances vulnérables
  - Npm audit
- Étiquetez vos tests (intégrations, sans IO, etc.) pour savoir quoi lancer et quand lancer.
  - Isoler bien le scénario de test : « pour éviter le chevauchement de test et expliquer facilement le déroulement du test, chaque test doit ajouter et agir sur son propre ensemble d'enregistrement de la base de données. Chaque fois qu'un test a besoin de récupérer ou de présumer l'existence de certaines données de la BD - il doit explicitement ajouter ces données et éviter de modifier tout autre enregistrement. »

Bonnes pratiques
Tests et pratiques générales de qualité

- Vérifiez votre couverture de test, cela aide à identifier les mauvaises conception de test

- Inspectez les paquets obsolètes
  - npm outdated ou npm-check-updates)

- Refactorisez régulièrement à l'aide d'outils d'analyse statique
  - Retravailler le code : est facilité par l'ide
    - Changer le nom d'un fichier
    - Changer le nom d'une variable,