TP – Pipeline CI (Node.js + GitHub + SonarCloud + Jest)

Auteur : Yassine BOUFENNECHE
Établissement : ESME
Date : 6 novembre 2025

Objectifs pédagogiques

À l'issue de ce TP, vous saurez :

- Créer un projet (mini) **Node.js** versionné sur **GitHub** ;
- Mettre en place une **intégration continue (CI)** avec **GitHub Actions** ;
- Analyser la **qualité du code** avec **SonarCloud** ;
- Travailler en équipe via des **Pull Requests (PR)** et des **revues de code** ;
- Écrire des **tests unitaires** avec Jest et garantir une **couverture minimale de 80 %**.

Étape 1 — Initialiser le projet

Objectif

Mettre en place la base du projet Node.js et effectuer le premier commit sur GitHub.

1. Créer le dossier du projet

Créer un dossier `mini-node-ci` et s'y rendre.

2. Initialiser le projet Node.js

Créer un projet Node.js, installer Jest / ESLint et configurer la commande de test avec couverture:

```
npm init -y
npm pkg set type="module"
npm install --save-dev jest@29 eslint
npm pkg set scripts.test="jest --coverage"
```

Astuce : La commande `npm pkg set` permet de modifier directement le fichier `package.json` sans l'ouvrir.

3. Créer un fichier `.gitignore`

Créer un `.gitignore` (évite de versionner `node_modules`, `coverage`, etc.) avec le contenu de `Node.gitignore` depuis GitHub, ou créez le fichier rapidement :

```
curl -L https://raw.githubusercontent.com/github/gitignore/main/Node.gitignore -o
.gitignore
```

- Ce fichier indique à Git quels fichiers **ne doivent pas être versionnés** (`node_modules`, `coverage`, etc.).

4. Créer le dépôt Git

- Initialiser un dépôt Git.
- Créez un commit avec le message: "*initialisation du projet*"

Important :
Ce premier commit **ne contient pas encore de code applicatif**. Il ne s'agit que de la base du projet.

5. Publier le projet sur GitHub

1. Créez un **nouveau dépôt vide** sur GitHub nommé `mini-node-ci`.
2. Liez-le votre dépôt local.
3. Poussez la première version du projet.

Étape 2 — Créer une branche et une Pull Request

Objectif

Travailler sur une nouvelle fonctionnalité dans une **branche dédiée**, ajouter du code, des tests unitaires et ouvrir une **Pull Request (PR)**.

1. Créer une branche de fonctionnalité

1. Créer une branchde de fonctionalité nommée `feature/premier-test`.
2. Basculer sur cette nouvelle branche.

Les branches de type `feature/...` permettent d'ajouter une fonctionnalité sans impacter la branche principale.

2. Créer le code et les tests

Créez les fichiers suivants :

`src/sum.js`

```
export function sum(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new TypeError('Both arguments must be numbers');
  }
  return a + b;
}
```

`__tests__/sum.test.js`

```
import { sum } from '../src/sum.js';
```

```
describe('sum', () => {
  test('additionne deux nombres', () => {
    expect(sum(2, 3)).toBe(5);
  });

  test('gère les nombres négatifs', () => {
    expect(sum(-2, 5)).toBe(3);
  });

  test('rejette les valeurs non numériques', () => {
    expect(() => sum('2', 3)).toThrow(TypeError);
  });
});
```

`jest.config.js`

```
export default {
  testEnvironment: 'node',
  collectCoverage: true,
  coverageDirectory: 'coverage',
  coverageReporters: ['text', 'lcov'],
  collectCoverageFrom: ['src/**/*.js'],
  coverageThreshold: {
    global: { branches: 80, functions: 80, lines: 80, statements: 80 },
  },
};
```

Astuce : Les tests doivent couvrir les cas “heureux” (OK) et les cas “erreurs” (exception, entrées invalides).

3. Lancer les tests localement

> Jest nécessite parfois une option pour les modules ES en Node 18+.

```
export NODE_OPTIONS=--experimental-vm-modules
npm test
```

Vérifiez que tous les tests passent et que la couverture est **≥ 80 %**. Vous devez voir un rapport de couverture **≥ 80 %**

- Quelles sont les **quatre métriques** mesurées par Jest (branches, functions, lines, statements) ?
- Que vous disent ces métriques sur la **qualité** de vos tests ?

Versionner les fichiers

1. Faites un nouveau commit avec le message *"feat: ajout de la fonction sum et des test unitaires"*.
2. Poussez la branche locale `feature/premier-test` vers le dépôt distant.

5. Créer une Pull Request (PR)

Sur GitHub :

1. Cliquez sur **Compare & pull request**.
2. Base : `main` | Compare : `feature/premier-test`.
3. Renseignez un titre et une description.

Exemple :

```
## Objet
Ajout de la fonction `sum` et des tests unitaires.

## Détails
- Fonction sum(a, b) avec vérification de type
- Tests unitaires complets (cas positifs et erreurs)
- Couverture minimale : 80 %

## Checklist
- [ ] Tests Jest verts
- [ ] CI GitHub Actions OK
- [ ] SonarCloud : Quality Gate Passed
```

4. Valider la création de du PR

Collaboration :

- Un camarade commente la PR (propose des améliorations).
- Vous corrigez localement (faites des modifs: ex. rajouter des commentaires dans le code).
- Faites un nouveau commit et poussez les changements:

```
git add .
git commit -m "fix: corrections after review"
git push
```

- Observez la PR sur GitHub.

Après l'étape 4 (pipeline CI), chaque nouveau push relance automatiquement la CI. Avant cela, il n'y a pas encore de checks sur GitHub : on exécute les tests en local avec `npm test`.

Étape 3 — Configurer SonarCloud

Objectif

Analyser la qualité du code et suivre la couverture.

1. Connectez-vous à https://sonarcloud.io.
2. Selectionnez le plan gratuit.
3. Créez une **organisation** et ajoutez votre projet GitHub.

4. Générez un **token**:
    - Clique sur ton avatar → My Account → Security.
    - Dans la section “Tokens”, clique sur Generate Token.
    - Donne-lui un nom, par exemple : `node-js-project`
5. Ajoutez le token dans votre dépôt GitHub sous le nom `SONAR_TOKEN`:
   Settings → Secrets and Variables → Actions → New repository secret.
6. Désactiver *Automatic Analysis* sur SonarCloud
    - *Votre projet → Administration → Automatic Analysis* -> Désactiver

`sonar-project.properties`

```
sonar.organization=votre-organisation
sonar.projectKey=votre-organisation_mini-node-ci
sonar.projectName=mini-node-ci
sonar.sources=src
sonar.tests=__tests__
sonar.javascript.lcov.reportPaths=coverage/lcov.info
sonar.qualitygate.wait=true
```

SonarCloud évaluera vos **tests, duplications, complexité, bugs potentiels** et plus encore.

Étape 4 — Créer le pipeline CI GitHub Actions

Objectif

Automatiser l’exécution des tests et l’analyse SonarCloud.

Créez le fichier `.github/workflows/ci.yml` :

```yaml
name: CI

on:
  push:
    branches: [ "main", "develop", "feature/**" ]
  pull_request:
    branches: [ "main", "develop" ]

jobs:
  build-test-analyze:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Setup Node.js
        uses: actions/setup-node@v4
```

```
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test -- --coverage
        env:
          NODE_OPTIONS: --experimental-vm-modules

      - name: SonarCloud Scan
        uses: SonarSource/sonarcloud-github-action@v2
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Remarque : Le plan gratuit de SonarCloud permet uniquement d'analyser la branche main.

- Fusionner la branche `release/premier-test` dans la branche `main`.
- Pousser la branche main vers le dépôt distant.
- Observer l'avancement de l'éxecution du pipeline dans l'onglet **Actions** sur GitHub.

Étape 5 — Protéger la branche principale

Objectif

Éviter les push directs sur `main`.

Sur GitHub → *Settings* → *Branches* → *Branch protection rules* :

- Require pull request before merging
- Require status checks to pass
- Require branches to be up to date before merging

Cela oblige à passer **obligatoirement par une PR** et à avoir la **CI verte** avant de fusionner.

1. Faites des modifications au niveau de votre code sur la branche `main`.
2. Faites un commit.
3. Essayer de pousser ces mangements.

- Que se passe-t-il ?
- Expliquez.

Étape 6 — Renforcer la qualité : nouveau module, couverture ≥ 80 %, ESLint & CI

Objectif

- Ajouter un **nouveau module** JavaScript et ses **tests**.

- Vérifier la **couverture locale** (`npm test`) et atteindre **≥ 80 %**.
- Installer et configurer **ESLint** (règles de code).
- **Ajouter `npm run lint` au pipeline** pour **bloquer la CI** si le code n'est pas conforme.

6.1 Ajouter un module JavaScript

ATTENTION : dans ce qui suit, vous travaillez directement sur la branche `main`

1. Créez un nouveau fichier `src/calc.js` :

```
export const calc = {
  add: (a, b) => a + b,
  sub: (a, b) => a - b,
  mul: (a, b) => a * b,
  div: (a, b) => {
    if (b === 0) throw new RangeError('Division by zero');
    return a / b;
  },
};
```

2. Vérifier la couverture du code en local.

6.2 Rajouter des tests unitaires

1. Créez son fichier de tests `__tests__/calc.test.js` suivant:

```
import { calc } from '../src/calc.js';

describe('calc', () => {
  test('add/sub/mul work', () => {
    expect ... ; // TODOD
    expect ... ; // TODOD
    expect ... ; // TODOD
  });

  test('div works', () => {
    expect ... ; // TODOD
  });

  test('div by zero throws', () => {
    expect(() => calc.div(1, 0)).toThrow(RangeError);
  });
});
```

2. Complétez le fichier pour implémenter les tests unitaires.

3. Re-vérifier la couverture du code en local.

Attendu : les tests passent, et le rapport de couverture (`coverage/lcov.info`) est généré.
Exigence : si la couverture < 80 %, **rajoutez des tests** jusqu'à atteindre le seuil.

6.3 Installer et configurer ESLint

1. Installez ESLint :

```
npm i -D globals
npx eslint --init
```

2. Créez le fichier`npm run lint` :

```
import globals from "globals";

export default [
  {
    ignores: ["node_modules/**", "coverage/**"],
  },

  {
    files: ["**/*.js"],
    languageOptions: {
      ecmaVersion: 2022,
      sourceType: "module",
      globals: {
        ...globals.node,
      },
    },
    rules: {
      "no-unused-vars": "error",
      "no-undef": "error",
      eqeqeq: ["error", "always"],
    },
  },

  {
    files: ["__tests__/**/*.js"],
    languageOptions: {
      globals: {
        ...globals.jest,
      },
    },
    rules: {
      // On peut alléger certaines règles dans les tests si besoin
    },
  },
];
```

3. Ajoutez un script **lint** dans `package.json` :

```
{
  "scripts": {
    "lint": "eslint .",
    "test": "jest --coverage"
  }
}
```

4. Lancez la commande `npm run lint` et observez le résultat.
5. Modifiez le fichier `src/calc.js` comme suit:

```
export const calc = {
  add: (a, b) => {
    return a + b;
  },

  sub: (a, b) => {
    const tmp = a - b;   //  variable non utilisée (no-unused-vars)
    return a - b
  },

  mul: (a, b) => {
    if (a == 0) return 0 //  "==" au lieu de "===" (eqeqeq)
    return a * b
  },

  div: (a, b) => {
    if (b === 0) throw new RangeError("Division by zero")
    return a / b
  }
}
```

6. Re-lancez la commande `npm run lint` et observez le résultat à nouveau.

6.4 Bloquer la CI si le lint échoue

1. Dans `.github/workflows/ci.yml`, ajoutez un **step lint** avant les tests :

```
- name: Lint
  run: npm run lint
```

2. Committez et poussez le code vers la branche distante `main`.
3. Le pipeline passe ? Pourquoi ?

Pourquoi le step lint avant les tests ?

Pour **échouer vite** si le code n'est pas conforme.
Ça économise du temps et rend le feedback plus clair pour les étudiants.

Annexe — Ressources et rappels

Créer un projet Node.js

Un projet Node.js se crée en initialisant un dossier avec **npm** :

```
mkdir mini-node-ci
cd mini-node-ci
npm init -y
```

- Cette commande crée automatiquement un fichier `package.json` qui décrit ton projet (dépendances, scripts, métadonnées…).
- Ensuite, installe les dépendances nécessaires.

Le fichier `package.json`

Ce fichier est le **cœur** d'un projet Node.js. Il indique :

- le nom du projet, la version, la licence ;
- les dépendances (`dependencies` et `devDependencies`) ;
- les scripts à exécuter (`npm run ...`).

Exemple simplifié :

```
{
  "name": "mini-node-ci",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "start": "node src/index.js",
    "test": "jest --coverage",
    "lint": "eslint ."
  },
  "devDependencies": {
    "eslint": "^9.0.0",
    "globals": "^15.0.0",
    "jest": "^29.0.0"
  }
}
```

> Les clés dans l'objet `"scripts"` permettent de créer des raccourcis pour lancer facilement les commandes.

Jest

Jest est un framework de tests très populaire pour JavaScript.

1. **Créer un fichier à tester** :

```
// src/sum.js
export function sum(a, b) {
  return a + b;
}
```

2. **Créer le test associé** :

```
// __tests__/sum.test.js
import { sum } from '../src/sum.js';

describe('sum function', () => {
  test('adds numbers correctly', () => {
    expect(sum(2, 3)).toBe(5);
  });
});
```

3. **Lancer les tests** :

```
npm test
```

4. **Résultat attendu** :

```
PASS  __tests__/sum.test.js
 ✓ adds numbers correctly (3 ms)
```

Le fichier `jest.config.js`

Ce fichier configure Jest selon tes besoins (couverture, dossiers, etc.) :

```
export default {
  testEnvironment: "node",
  collectCoverage: true,
  collectCoverageFrom: ["src/**/*.js"],
  coverageDirectory: "coverage",
  coverageThreshold: {
    global: {
      branches: 80,
      functions: 80,
```

```
      lines: 80,
      statements: 80
    }
  }
};
```

- `collectCoverage`: active la collecte de couverture.
- `collectCoverageFrom`: spécifie les fichiers à inclure.
- `coverageThreshold`: bloque les tests si la couverture est insuffisante.

ESLint — Analyse statique du code

ESLint vérifie ton code sans l'exécuter.
Il signale :

- les erreurs de syntaxe (`no-undef`, `no-unused-vars`),
- les incohérences (`eqeqeq`, `semi`),
- et peut même corriger automatiquement certains problèmes.

Lancer le lint :

```
npm run lint
```

Les fichiers YAML (`.yml` ou `.yaml`)

- Les fichiers YAML sont des fichiers **de configuration lisibles**.
- Ils utilisent l'indentation (espaces) pour représenter la structure.
  Exemple :

```
name: Example
on:
  push:
    branches: [main]
jobs:
  say-hello:
    runs-on: ubuntu-latest
    steps:
      - name: Print message
        run: echo "Hello, world!"
```

> YAML = "Yet Another Markup Language" : très utilisé pour GitHub Actions, Docker Compose, Kubernetes, etc.

Structure d'un pipeline GitHub Actions

- Un workflow GitHub Actions est défini dans `.github/workflows/ci.yml`.
- Sa structure suit ce modèle :

```yaml
name: CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 20

      - name: Install dependencies
        run: npm ci

      - name: Lint
        run: npm run lint

      - name: Run tests
        run: npm test

      - name: SonarCloud analysis
        uses: SonarSource/sonarcloud-github-action@v2
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Décomposition :

Élément	Rôle
`on:`	Déclencheurs du pipeline (push, PR, etc.)
`jobs:`	Ensemble de tâches exécutées (ici `build-test`)
`runs-on:`	Environnement d'exécution (Ubuntu, Windows, etc.)
`steps:`	Étapes successives du job
`uses:`	Réutilise une action publique (ex. `actions/checkout`)
`run:`	Exécute une commande shell