




TESE510 - Atelier - Industrialisation des Processus de Tests - Support Partie 1

👤 Owner	 Julien Couraud
🏢 Module for	EPSI Bordeaux
🎓 Promotion	Bachelor 3
📅 Cursus	DEV FS - IA
☰ Year	2024-2025 S2

1. Starter-kit Next.js avec Jest

Dans cette partie, nous allons préparer un socle applicatif avec le framework FullStack Next.js et nous écrirons quelques tests unitaires autour d'une fonctionnalité.

1.1 Exemple "with-jest"

Starter kit: <https://github.com/vercel/next.js/tree/canary/examples/with-jest>

- Création du projet en local

```
npx create-next-app --example with-jest with-jest-app
```

- Ajout du projet sur Github en tant que répertoire existant

<https://docs.github.com/fr/desktop/adding-and-cloning-repositories/adding-an-existing-project-to-github-using-github-desktop>

- Execution de l'application en local

```
npm run dev
```

- Execution des tests en local

```
npm run test
```

2. Tests unitaires Full Stack, avec exécution en local

2.1 Tests existants sur le socle

- Dossier `__tests__` : tests globaux et relatifs à l'application en elle même (build ok, setup layout ok, home page ok, etc...)
→ Fichiers
`xxxx.test.tsx` : tests relatifs à des petits composants graphiques qui ont leur propre logique d'implémentation. Il est très recommandé de placer ces fichiers de tests au même endroit que la déclaration du composant dans l'arborescence du projet, notamment pour faciliter la réutilisabilité.

2.2 Fonctionnalité côté client et ses tests

- Le fichier `counter.tsx` contient un composant React implémentant une logique de compteur s'incrémentant au clic sur un bouton.
- En l'intégrant à la page d'accueil, nous pouvons observer en local comment se comporte graphiquement le composant.

```
//app/page.tsx

export const metadata = {
  title: "App Router",
};

import Counter from "app/counter";

export default function Page() {
  return (
    <>
      <h1>App Router</h1>
      <Counter />
    </>
  );
}
```

```
);  
}
```

- Dans l'implémentation du composant, cassons volontairement la logique d'incrémentation qui semble avoir été testé dans le fichier de test associé au composant.

```
//app/page.tsx  
"use client";  
  
import { useState } from "react";  
  
export default function Counter() {  
  const [count, setCount] = useState(0);  
  return (  
    <>  
      <h2>{count}</h2>  
      /*  
      <button type="button" onClick={() => setCount(count +  
1)}>  
      */  
      <button type="button" onClick={() => setCount(count -  
1)}>  
        +  
      </button>  
    </>  
  );  
}
```

- A l'exécution des tests unitaires de l'application, nous observons bien une erreur liée à la fonctionnalité de compteur. Les tests présents ont donc prévenu une régression d'une fonctionnalité et peuvent être considérés (au moins pour ce cas d'usage) plutôt efficaces.

2.3 Les briques d'un outillage de tests

- **Assertions**

La brique élémentaire de nos tests est une assertion. Il s'agit tout simplement d'un morceau de code qui vérifie qu'une

condition est bien remplie, à défaut de quoi elle lève une erreur adaptée.

En pratique, on utilisera des bibliothèques d'assertions déjà disponibles, plus ou moins riches et extensibles. Cela ne nous empêche pas, si besoin, d'écrire les nôtres.

- **Tests et suites**

Un test est un petit bloc de code qui pose une question précise et en vérifie la réponse. Il peut s'agir d'une question très élémentaire, ou d'une question déjà plus élaborée.

Un test est toujours constitué des mêmes étapes :

- Si besoin, mise en place (setup)
- Stimulus (appel de fonction, simulation d'événement, etc.)
- Vérification des attentes (au moyen des assertions)

Les tests sont généralement regroupés en suites de tests, à raison généralement d'une suite par fichier.

- **Harnais**

Un harnais de test est un programme qui agit en quelque sorte comme le chef d'orchestre de vos tests. Classiquement, il permet de :

- Trouver vos fichiers de tests (suites)
- Fournir une syntaxe de structuration de ces fichiers.
- Exécuter ces fichiers dans un environnement de test approprié (protection contre les exceptions, timeouts, etc.)
- Fournir un rapport sur les tests ayant réussi ou échoué
- En développement, surveiller vos fichiers de test et codes sources testés, pour relancer les tests appropriés lors d'une modification de fichier.

Certains harnais font davantage, en fournissant par exemple une bibliothèque d'assertions intégrée, en mesurant la couverture de tests, etc.

- **Couverture de tests**

Lorsqu'on commence à avoir des tests, il devient possible de mesurer la couverture de tests, c'est-à-dire le pourcentage de notre code, à l'expression près, qui est sollicité par les tests. On peut alors repérer les parties non testées, ou insuffisamment

testées, et savoir ainsi où concentrer nos prochains efforts d'écriture de tests. Divers services permettent d'utiliser la couverture de tests comme critère de blocage pour l'intégration de nouveau code à nos projets, dans le cadre par exemple de pull requests sur GitHub. CodeCov et Coveralls, par exemple, permettent de définir des exigences de taux absolu plancher, ou l'interdiction de faire baisser le taux existant, pour autoriser une pull request à être fusionnée dans sa branche destinataire. À partir d'un certain taux de couverture, généralement aux alentours de 90–95%, on peut commencer à considérer que des tests « verts » (qui passent tous sans problème) constituent une autorisation de déploiement légitime, qui peut alors être automatisé : on parle de déploiement continu.

2.4 Exercice : Implémentation d'une Calculatrice en Next.js

Dans cet exercice, vous allez implémenter un **composant React** permettant d'effectuer les quatre opérations de base : **addition, soustraction, multiplication et division**.

L'objectif est de réutiliser ce qui a été vu précédemment tout en optimisant l'implémentation afin d'éviter la duplication de code.

Consignes :

1. Créez un composant **Calculator** qui comprend :
 - Deux champs de texte pour saisir les nombres `a` et `b`
 - Un champ de sélection (`select`) pour choisir l'opérateur
 - Un bouton `submit` pour effectuer le calcul
 - Une zone d'affichage du résultat
2. Implémentez la logique de calcul dans une fonction dédiée.
3. Gérez les cas particuliers, notamment :
 - La division par zéro
 - L'utilisation d'un opérateur invalide
4. Évitez la duplication de code et optimisez la gestion des opérations.

Exemple de composant **Calculator.tsx** :

```

// app/calculator.tsx

"use client";

import { useState } from "react";

export default function Calculator() {
  const [a, setA] = useState("");
  const [b, setB] = useState("");
  const [operator, setOperator] = useState("+");
  const [result, setResult] = useState<number | string>
  ("");

  function calculate() {
    const numA = parseFloat(a);
    const numB = parseFloat(b);
    if (isNaN(numA) || isNaN(numB)) {
      setResult("Veuillez entrer des nombres valide
s");
      return;
    }
    switch (operator) {
      case '+': setResult(numA + numB); break;
      default: setResult("Opérateur non supporté");
    }
  }

  return (
    <div>
      <input type="text" value={a} onChange={(e) => s
etA(e.target.value)} placeholder="Nombre A" />
      <select value={operator} onChange={(e) => setOp
erator(e.target.value)}>
        <option value="+">+</option>
        <option value="-">-</option>
        <option value="*">*</option>
        <option value="/">/</option>
      </select>

```

```

        <input type="text" value={b} onChange={(e) => s
etB(e.target.value)} placeholder="Nombre B" />
        <button onClick={calculate}>Calculer</button>
        <h3>Résultat : {result}</h3>
    </div>
  );
}

```

Exemple de test avec Jest :

```

// app/calculator.test.tsx

import { render, screen, fireEvent } from "@testing-librar
y/react";
import Calculator from "../app/Calculator";

test("Addition fonctionne correctement", () => {
  render(<Calculator />);
  const inputA = screen.getByPlaceholderText("Nombre A");
  const inputB = screen.getByPlaceholderText("Nombre B");
  const select = screen.getByRole("combobox");
  const button = screen.getByText("Calculer");
  const result = screen.getByText(/Résultat/i);

  fireEvent.change(inputA, { target: { value: "5" } });
  fireEvent.change(inputB, { target: { value: "3" } });
  fireEvent.change(select, { target: { value: "+" } });
  fireEvent.click(button);

  expect(result).toHaveTextContent("8");
});

```

À vous de jouer :

Ajoutez des tests pour les autres opérations et assurez-vous que le composant fonctionne correctement dans l'application Next.js.

2.5 Ouverture d'une route API côté serveur et ses tests

Dans cette section, nous allons implémenter une **route API** dans Next.js qui permettra de stocker temporairement l'**historique des opérations** effectuées par la calculatrice.

Objectifs :

1. **Créer une route API** `/api/history` qui stocke les opérations réalisées dans une variable temporaire côté serveur.
2. **Modifier le composant Calculator** pour envoyer chaque opération réalisée à l'API.
3. **Créer un test Jest** pour vérifier le bon fonctionnement de l'API.

Implémentation de la route API `/api/history.ts` :

```
// app/api/history/route.ts

import { NextRequest, NextResponse } from "next/server";

let history: { a: number; b: number; operator: string; result: number | string }[] = [];

export async function GET() {
  return NextResponse.json(history);
}

export async function POST(req: NextRequest) {
  try {
    const body = await req.json();
    const { a, b, operator, result } = body;

    if (typeof a !== "number" || typeof b !== "number" ||
      typeof result !== "number" && typeof result !== "string") {
      return NextResponse.json({ error: "Invalid data format" }, { status: 400 });
    }

    history.push({ a, b, operator, result });
  }
}
```



```

        return NextResponse.json({ success: true });
    } catch (error) {
        return NextResponse.json({ error: "Invalid request"
    }, { status: 400 });
    }
}

```

Modification du composant Calculator pour interagir avec l'API :

```

// app/calculator.tsx

"use client";

import { useState } from "react";

export default function Calculator() {

    // ...

    const [history, setHistory] = useState<{ a: number; b:
number; operator: string; result: number | string }[]>([]);

    async function calculate() {

        // ...

        setResult(operationResult);

        // Envoyer l'opération à l'API
        await fetch("/api/history", {
            method: "POST",
            headers: { "Content-Type": "application/json"
        },
            body: JSON.stringify({ a: numA, b: numB, operat
or, result: operationResult })),
        });
    }
}

```

```

        // Mettre à jour l'historique
        fetch("/api/history")
            .then(res => res.json())
            .then(data => setHistory(data));
    }

    return (
        <div>
            <input type="text" value={a} onChange={(e) => setA(e.target.value)} placeholder="Nombre A" />
            <select value={operator} onChange={(e) => setOperator(e.target.value)}>
                <option value="+">+</option>
                <option value="-">-</option>
                <option value="*">*</option>
                <option value="/">/</option>
            </select>
            <input type="text" value={b} onChange={(e) => setB(e.target.value)} placeholder="Nombre B" />
            <button onClick={calculate}>Calculer</button>
            <h3>Résultat : {result}</h3>
            <h3>Historique :</h3>
            <ul>
                {history.map((entry, index) => (
                    <li key={index}>
                        {entry.a} {entry.operator} {entry.b} = {entry.result}
                    </li>
                ))}
            </ul>
        </div>
    );
}

```

Test Jest pour l'API /api/history :

- Ajout de la librairie `node-mocks-http` (<https://www.npmjs.com/package/node-mocks-http>)

```
npm i --save-dev node-mocks-http
```

- Ajout du polyfill "whatwg-fetch"
(<https://www.npmjs.com/package/whatwg-fetch>)

```
// jest.setup.js

// ...

import "whatwg-fetch"; // Polyfill pour Fetch API
```

- Un exemple basique de tests pour les méthodes GET et POST.

```
// app/api/history/route.test.ts

import { NextRequest } from "next/server";
import { GET, POST } from "../route"; // Import des handlers
de la route
import httpMocks from "node-mocks-http";

jest.mock("next/server", () => ({
  NextResponse: {
    json: jest.fn((data) => ({ json: data, status: 200 })),
  },
}));

describe("API /api/history", () => {
  it("devrait appeler GET et retourner une réponse", async
  () => {
    const req = httpMocks.createRequest({ method: "GET" });
    const res = httpMocks.createResponse();

    const spy = jest.spyOn(global, "fetch"); // Vérifier si
    une requête est faite

    await GET(req, res);
```

```

    expect(spy).not.toHaveBeenCalled(); // On ne veut pas d
e fetch ici
    expect(res._getStatusCode()).toBe(200);
    expect(res._getData()).toBeDefined();
  });

  it("devrait appeler POST et stocker une opération", async
  () => {
    const req = httpMocks.createRequest({
      method: "POST",
      body: {},
    });
    const res = httpMocks.createResponse();

    await POST(req, res);

    expect(res._getStatusCode()).toBe(200);
    expect(res._getData()).toBeDefined();
  });
});

```

2.6 Exercices pour approfondir

- Utilisez les différentes méthodes disponibles et proposées par Jest pour affiner vos tests, notamment sur la logique métier importante (<https://jestjs.io/fr/docs/api>).
- Entraînez-vous sur toutes les méthodes d'un CRUD pour
- Utilisez la librairie `supertest` (<https://www.npmjs.com/package/supertest>), qui propose un haut niveau d'abstraction et propose donc une écriture de tests plus simple et productive.

2.7 Comment écrire un test qui apporte de la valeur ?

Écrire des tests automatiques est une activité chronophage. Déjà, l'écriture en elle-même demande du temps. Mais ce qui en prend le plus, c'est la maintenance du test.

De plus, comme pour n'importe quel code, il est sujet à la dette technique. Il est donc utile de l'améliorer régulièrement, de le refactorer, de le nettoyer, etc.

À un moment ou un autre, il arrivera aussi que le test ne passe plus. Dans ce cas, il va falloir investiguer et mener l'enquête afin de savoir si le problème vient du code ou du test. Autrement dit, il faudra déterminer s'il y a vraiment une erreur ou si le test n'est plus à jour.

Enfin, le code qui est testé a de grandes chances d'évoluer.

Donc il va falloir encore passer du temps à corriger le test pour qu'il reflète les évolutions.

Vous l'avez compris : tester, c'est très bien et cela permet de gagner un temps précieux. Mais cette activité nécessite également d'y consacrer régulièrement du temps, comme pour n'importe quel autre code qui compose votre application.

Il faut donc réussir à être pragmatique et à essayer d'écrire le moins de tests possible tout en s'assurant qu'ils sont efficaces et utiles.

Passons à quelques astuces vous permettant de savoir ce qui donne de la valeur à un test.

Avoir des tests inutiles est assez risqué.

En effet, ils vous donnent une fausse impression de sécurité. Vous avez énormément de tests, alors vous vous croyez très bien protégé, alors qu'en fait, vous ne l'êtes pas du tout !

De plus, si vous avez produit beaucoup de code qu'il va falloir faire évoluer, maintenir, remanier, nettoyer... Vous perdrez beaucoup de temps perdu à maintenir les tests, mais aucun bénéfice à l'horizon.

Tester le code le plus important en premier

Il faut tester le code qui est le plus important en premier. Tester le code qui fait la force de votre application, de votre entreprise. C'est en général le code métier, celui pour lequel toute l'application a été écrite.

Vous avez un site d'e-commerce ? Il faut vérifier qu'un client puisse bien ajouter un produit à son panier et le payer. Pouvoir changer la date d'anniversaire de son chien est sans doute une fonctionnalité intéressante offerte par votre site, mais elle peut être testée après le plus important.

Un test doit avoir une forte probabilité de montrer des régressions

Cela se traduit notamment par le fait de tester un nombre important de lignes et de fonctionnalités. Mais pas seulement. Ce n'est pas toujours qu'une question de lignes, mais aussi de signification. Tester du code métier, c'est bien, mais tester du code trivial n'a pas d'intérêt.

Un test doit avoir une probabilité faible de montrer des faux positifs

Un faux positif correspond à un test qui échoue alors que la fonctionnalité testée est opérationnelle. Le test ne devrait pas échouer, il s'agit d'une fausse alarme.

Un faux positif peut arriver à plusieurs occasions, par exemple lors d'une phase de refactoring, lorsque le test ne correspond plus au code que l'on a créé.

Si un test échoue pour de mauvaises raisons, alors on risque de ne plus prêter attention au résultat du test et de l'ignorer. La conséquence est grave, car on risque de laisser passer une régression bien réelle.

Un faux positif a plus de chance d'arriver lorsque le test est directement lié aux détails d'implémentation. Pour réduire les risques, il faut découpler le test de l'implémentation du système à tester. Vous pouvez traiter votre code comme une boîte noire, par exemple, comme si vous ne saviez pas ce qu'il y a à l'intérieur. Vous utilisez les entrées et vérifiez les sorties, sans chercher forcément à vérifier toutes les étapes possibles.

Un test doit apporter un feedback rapide

C'est très important, car plus on a un retour rapide, moins on perd de temps à partir dans la mauvaise direction.

Il faut donc que l'intégralité des tests puisse s'exécuter très rapidement.

Des tests très complets (de bout en bout, voire d'interfaces) sont particulièrement intéressants pour capturer les régressions sans faux positif. Malheureusement, ils sont en général très longs à exécuter, ce qui est un problème pour avoir un feedback rapide.

Un test doit avoir un coût de maintenance réduit

En général, c'est fortement lié à la taille du test. Plus il y a de lignes de codes dans un test, plus la maintenance sera compliquée et prendra du temps.

Il est plus facile de lire du code ou de le modifier lorsqu'il y a peu de lignes.

Écrivez des tests simples, qui ne testent pas énormément de choses. Il vaut mieux écrire 5 petits tests qu'un gros test qui teste 5 choses. Cela sera plus simple de déterminer celui qui échoue et simplifiera le code global.

Si le test porte sur des détails d'implémentation, il y aura non seulement un risque plus élevé de montrer des faux positifs, mais il sera beaucoup plus fragile au refactoring. Si je teste trois méthodes et qu'après un refactoring, il ne me reste qu'une seule méthode, alors il va falloir que je modifie les tests, ou que je les réécrive complètement.