

Séance 12 :

→ **REACT.JS** – Introduction aux tests unitaires

INF37407 – Technologie de l'inforoute

Prof. Yacine YADDADEN, Ph. D.

Plan

1. Introduction aux tests
2. Outils & environnement
3. Premiers tests
4. Tests avancés essentiels
5. Tests unitaires dans React
6. Bonnes pratiques & pièges à éviter
7. Conclusion
8. Questions & Discussion

Introduction aux tests

→ Objectif du cours

- ✓ Comprendre **l'intérêt des tests unitaires** (qualité, fiabilité, non-régression).
- ✓ Découvrir les outils modernes : **Jest** (ou **Vitest**).
- ✓ Écrire ses **premiers tests simples** sur des fonctions JS & TS.
- ✓ Intégrer les tests dans un **workflow moderne**.

Introduction aux tests

→ Pourquoi tester ?

- ✓ **Réduire les bugs**

- Détecter rapidement les erreurs avant la mise en production.

- ✓ **Favoriser le *refactoring***

- Modifier et améliorer le code en confiance, sans tout retester manuellement.

- ✓ **Améliorer la qualité du code**

- Forcer une meilleure structure, des fonctions plus claires et plus prévisibles.

- ✓ **Gagner du temps sur le long terme**

- Moins de débogage, moins de régressions, moins de chaos lors des évolutions.

- ✓ **Assurer la non-régression**

- Garantir que les nouvelles fonctionnalités ne cassent pas les anciennes.

Introduction aux tests

→ Qu'est-ce qu'un test unitaire ?

- **Teste une unité de code isolée**

→ Une fonction, une méthode ou un petit composant testé séparément du reste.

- **Rapide, automatique et indépendant**

→ Chaque test doit pouvoir s'exécuter seul, sans dépendance externe.

- **Règle AAA : Arrange → Act → Assert**

1. *Arrange* : préparer les données

2. *Act* : exécuter la fonction

3. *Assert* : vérifier le résultat attendu

- **Exemple simple**

→ Tester $sum(a, b)$ doit confirmer que $sum(2, 3) = 5$.

Outils & environnement

→ Frameworks de tests

- **Jest** (*le plus populaire*)
 - Utilisé dans la majorité des projets React/Node. Simple, complet, batteries incluses.
- **Vitest** (*rapide et moderne*)
 - Alternative légère compatible Vite. Très rapide, excellent pour TypeScript.
- **Mocha** ou **Chai** (*historique*)
 - Frameworks plus anciens, flexibles mais nécessitent plus de configuration.
- **Cypress**
 - Utilisé pour les tests end-to-end (E2E) : interaction complète avec l'application.

Outils & environnement

→ Installation & configuration

- **Installer Jest**

- *npm install --save-dev jest*
- Framework complet, recommandé pour la majorité des projets JS/TS.

- **Installer Vitest**

- *npm install --save-dev vitest*
- Ultra rapide, parfait pour les projets basés sur **Vite**.

- **Support TypeScript**

- *npm install --save-dev ts-jest @types/jest*
- Permet d'exécuter des tests directement en TypeScript.

Outils & environnement

→ Structure d'un projet de tests

- *src/* → **code source**
 - Contient les fonctions, classes, composants à tester.
- *tests/* ou *__tests__/* → **fichiers de tests**
 - Organisation recommandée pour isoler clairement les tests.
- **Convention de nommage**
 - *maFonction.test.js* ou *maFonction.test.ts*
- **Exécution des tests**
 - La commande → *npm test*
 - Lance tous les tests du projet.

Premiers tests JS & TS

→ Premier test en JavaScript



```
1 function sum(a, b) {  
2   return a + b;  
3 }
```

Fonction à tester



```
1 test("sum adds numbers", () => {  
2   expect(sum(2, 3)).toBe(5);  
3 });
```

Test unitaire

- On vérifie une **seule logique simple**.
- *test()* définit un scénario.
- *expect()* compare le résultat obtenu au résultat attendu.
- *toBe()* vérifie une égalité stricte.

Premiers tests JS & TS

→ Premier test en TypeScript



```
1 export const multiply = (a: number, b: number): number => a * b;
```

Fonction à tester



```
1 test("multiply works", () => {  
2   expect(multiply(4, 2)).toBe(8);  
3 });  
4
```

Test unitaire

Premiers tests JS & TS

→ Les *matchers*

- *Matchers* les plus utilisés :
 - **toBe()** → comparer une valeur primitive (égalité stricte)
 - **toEqual()** → comparer objets et tableaux
 - **toBeTruthy()** ou **toBeFalsy()** → vérifier la véracité
 - **toThrow()** → tester qu'une fonction génère une erreur



```
1 expect([1, 2]).toContain(1);
```

Tests avancés essentiels

→ Tester l'asynchrone (async & await)

- Les tests doivent attendre la résolution de la promesse.



```
1  async function fetchData() {  
2      return "done";  
3  }  
4  
5  test("async example", async () => {  
6      const result = await fetchData();  
7      expect(result).toBe("done");  
8  });
```

Tests avancés essentiels

→ Mocking simple

Pourquoi *mock* ?

- Isoler la fonction
 - Tester uniquement la logique ciblée sans dépendre d'autres modules.
- Simuler une API, une base de données ou un module externe
 - Éviter les appels réels, accélérer les tests et les rendre fiables.

Tests avancés essentiels

→ Coverage & analyse de couverture

À quoi sert la couverture de code ?

- Mesurer la proportion du code testé
 - Combien de lignes, branches, fonctions sont réellement exécutées
- Identifier les zones non testées
 - Utile pour repérer les risques de bugs ou de régressions.
- La commande : *npm test -- --coverage*
 - Génère un rapport détaillé dans le dossier *coverage/*
- **Indicateurs principaux** : **Statements** : instructions exécutées, **Branches** : conditions (if, switch...), **Functions** : fonctions appelées, **Lines** : lignes réellement couvertes.

Tests unitaires avec React

- Exemple avec React Testing Library



```
1 render(<Button label="OK" />);  
2 expect(screen.getByText("OK")).toBeInTheDocument();
```

- Principes essentiels

- Tester le comportement

- Vérifier ce que l'utilisateur voit ou fait, pas le code interne.

- Ne pas tester l'implémentation

- Éviter de dépendre de la structure interne des composants.

- Interagir comme un utilisateur

- Utiliser screen, fireEvent, userEvent pour tester :

Bonnes pratiques & pièges à éviter

- Un test = un comportement
 - Chaque test doit vérifier une seule logique précise.
- Noms explicites
 - Les tests doivent être lisibles et compréhensibles sans effort.
- Tests isolés & déterministes
 - Un test ne dépend jamais d'un autre. Le résultat doit toujours être le même.
- Ne pas *mock* inutilement
 - Utiliser les mocks uniquement pour isoler une dépendance externe.
- Courtes PR + tests automatiques
 - Favoriser des petites modifications, validées automatiquement via CI.

Conclusion

- Les **tests unitaires** améliorent la qualité, la stabilité et la maintenabilité du code.
- **Jest** et **Vitest** sont les outils modernes les plus utilisés en JS/TS.
- **TypeScript** renforce la fiabilité grâce au typage statique.
- L'intégration des tests dans une **CI** (ex. : GitHub Actions) automatise les vérifications à chaque commit.

Questions & Discussion

Bibliographie

1. Templier, Thierry & Gougeon, Arnaud (2007). JavaScript pour le Web 2.0 Programmation objet, DOM, Ajax, Prototype, Dojo, Script.aculo.us, Rialto. Éditions Eyrolles.
2. Porteneuve, Christophe (2008). Bien développer pour le Web 2.0 : Bonnes pratiques Ajax. Éditions Eyrolles.
3. Engels, Jean (2012). HTML5 et CSS3 : Cours et exercices corrigés. Éditions Eyrolles.
4. Martin, Michel (2014). HTML5, CSS3 & jQuery : Créez votre premier site web. Éditions Pearson.