# Projet fil rouge de test - Facadia

Tom Avenel

# Table des matières



# Présentation du projet

**Lien**

Ce projet est très fortement inspiré du tutoriel OpenClassrooms : testez vos applications frontend avec javascript.

Récupérer les sources du projet :
`git clone https://github.com/tdimnet/testez-vos-applications-front-end-avec-javascript`

Pour ce projet, vous aurez besoin de créer un compte sur l'API WeatherCast : https://weatherstack.com/

Notre projet fil rouge permet de suivre des façades par le biais de capteurs. Ces capteurs nous donnent des informations telles que le degré d'humidité ou les coordonnées géographiques. Grâce à l'API WeatherCast et aux coordonnées GPS de chaque capteur, nous récupérons les données météorologiques pour chaque façade.

Le projet comporte quatre pages :

- Une page de connexion où on va devoir rentrer son adresse e-mail et son mot de passe ;
- La page d'accueil, qui comprend 34 capteurs récupérés via une requête HTTP ;
- La page "façade", qui donne des détails sur une façade ;
- La page d'ajout d'une façade. Elle comprend un formulaire.

Pour démarrer le projet, vous pouvez utiliser l'extension VSCode live server de ritwickdey. L'extension ajoute un bouton "Go Live" en bas à droite de l'interface.

# Tests unitaires

## Framework Jest

Les tests unitaires seront écrits avec le framework `Jest` (déjà intégré dans le projet). Voir le TP d'introduction à Jest avant d'effectuer cette partie.

## Test unitaire : pagination

Se déplacer dans la branche `partie-1/chapitre-2-a` :

```
git checkout partie-1/chapitre-2-a
```

L'application possède une fonction de pagination dans `js/pages/common/pagination/index.js` : `getNumberOfPages(numberOfSensors)`

Cette fonction prend un paramètre : `numberOfSensors` . Cela correspond au nombre total de capteurs récupérés dans le fichier `homepage-data.json` . Pour cette fonction, je peux écrire quatre tests :

- Un cas de test où je lui passe 12 en paramètre. Ici, je ne vais pas tester le résultat de la fonction, mais simplement vérifier si la fonction me retourne bien quelque chose. On utilisera : `expect().toBeDefined()`
- Un cas de test où je lui passe 0 en paramètre, soit aucun capteur. Et je teste si le nombre de pages retourné est 0.
- Un cas de test où je lui passe 7. Le résultat retourné par la fonction devrait être de 1, car j'affiche 8 capteurs par page. Autrement dit, il n'y aura qu'une seule page.
- Enfin, un cas où je lui passe 34, et où je teste si j'ai bien 5 pages.

**Exercice**

Écire les tests correspondants.
La correction est dans la branche : `git checkout partie-2/chapitre-1`.

## Test unitaire : isInTestEnv

Se placer dans la branche : `git checkout partie-2/chapitre-2-c`

Écrire le test correspondant pour la fonction `isInTestEnv()` du fichier `js/utils/env/index.js`

**Test 1**

- **Given** : Je suis en environnement de test
- **When** : J'appelle la fontion `isInTestEnv()`
- **Then** : La fonction me retourne le booléen `true`

**Test 2**

- **Given** : Je suis en environnement de test
- **When** : J'appelle la fontion `isInTestEnv()` et que je précise que je ne suis pas en environnement de test
- **Then** : La fonction me retourne le booléen `false`

### Test unitaire : retrieveSensorsData

Rester dans la branche : `git checkout partie-2/chapitre-2-c`

Écrire le test correspondant pour la fonction `retrieveSensorsData()` du fichier `js/utils/api/index.js`

- **Given** : Je suis en environnement de test
- **When** : J'appelle la fontion `retrieveSensorsData()` pour récupérer les données des capteurs de la page d'accueil
- **Then** : La fonction me retourne bien les données des façades mockées.

On utilisera pour cela un mock des données de la page d'accueil :

```
import { data } from '../../../data/mock-homepage-data'
```

**Exercice**

La correction est dans la branche : `partie-2/chapitre-2-c-solution`

### Plan de tests unitaires

Rester dans la branche `partie-2/chapitre-2-c`. Inspecter le code, particulièrement les fonctions utilitaires de `js/utils` et ajouter les tests unitaires correspondants. On pourra modifier le code du programme au besoin.

## Tests d'intégration

### DOM Testing

Les tests d'intégration seront écrits grâce à :

- `Jest` comme framework d'exécution des tests. `Jest` est un framework de tests unitaires mais est utilisé ici comme méchanisme de sélection, d'exécution et de vérification des tests. C'est souvent comme cela que l'on exécute des tests automatiques à tous les niveaux : on réutilise un framework de tests unitaires pour tourner d'autres tests.
- la librairie `DOM Testing` permet de réaliser les actions de sélection d'éléments dans le Domain Object Model (DOM), nécessaires durant les tests d'intégration. On utilisera l'implémentation de la `Testing Library` qui est la nouvelle norme commune de sélection d'éléments dans le DOM : https://testing-library.com/docs/dom-testing-library/intro/ . Il

est possible d'utiliser `Jest` seul (`Jest` simule le DOM de test) mais la sélection des éléments dans le DOM est alors très compliquée.

Exemple de test d'intégration sur le DOM :

```
/**
* @jest-environment jsdom
*/

// Ici j'importe DOM Test Library
import { getByTestId } from '@testing-library/dom'

describe('Sample 1 Integration Test Suites', () => {
    it('should display "Hello, Tom"', () => {
        // Je crée un nouveau noeud
        const $wrapper = document.createElement('div')

        // Je lui injecte du HTML
        $wrapper.innerHTML = `
            <div id="root">
                <h1 data-testid="hello">Hello, Tom</h1>
            </div>

        // Je teste le resultat
        expect(getByTestId($wrapper, "hello").textContent).toEqual("Hello, Tom")
    })
})
```

- `@jest-environment jsdom` : ce commentaire est à mettre dès le début du fichier. Il permet d'indiquer à `Jest` de simuler un DOM sur lequel travailler dans l'environnement de test.
- `<h1 data-testid=...` : ajoute un attribut `data-testid` récupéré ensuite avec `getByTestId()`

**Lien**

La difficulté des tests sur le DOM vient principalement de la sélection des éléments.
Voir :

- la documentation de la `Testing Library` : https://testing-library.com/docs/queries/about/.
- la documentation des matchers `jest-dom` : https://github.com/testing-library/jest-dom

Voir aussi la page du tutoriel OpenClassrooms dédié

## Premiers tests du DOM

### Tests du routeur

Vous pouvez vous mettre sur la branche `git checkout partie-3/chapitre-2-a`

Pour réaliser les scénarios de tests, on pourra par exemple utiliser les instructions suivantes :

```
document.location = '/#/home' // pour aller sur la page 'home' (optionel)
await router() // pour activer le router
```

**Test 1**

- **Given** : En tant qu'utilisateur
- **When** : je vais sur l'URL `/`
- **Then** : je souhaite voir la page de connexion s'afficher avec le titre : "Veuillez vous connecter"

**Test 2**

- **Given** : En tant qu'utilisateur
- **When** : je vais sur l'URL `/#/home`
- **Then** : je souhaite voir la page d'accueil des capteurs s'afficher avec le titre "Vos capteurs"

**Test 3**

- **Given** : En tant qu'utilisateur
- **When** : je vais sur l'URL `/#/facade-details`
- **Then** : je souhaite voir la page d'accueil des capteurs s'afficher avec le titre "Détails du capteur"

**Test 4**

- **Given** : En tant qu'utilisateur
- **When** : je vais sur l'URL `/#/add-sensor`
- **Then** : je souhaite voir la page d'accueil des capteurs s'afficher avec le titre "Ajout d'un nouveau capteur"

**Formulaire de connexion**

Vous pouvez vous mettre sur la branche `git checkout partie-3/chapitre-2-b`

Ajouter les tests d'intégration pour le formulaire de connexion (fichier `js/utils/signInForm/index.js`) correspondant aux scénarios suivants.

On commencera par se rendre sur le formulaire dans chacun des tests :

```
import SignInPage from '../../pages/signIn/index'
import { handleSignInForm } from './index'

beforeEach(() => {
    document.body.innerHTML = SignInPage.render()
    handleSignInForm()
})

afterEach(() => {
    document.body.innerHTML = ''
})
```

**Test 1**

- **Given** : en tant qu'utilisateur déconnecté
- **When** : je complète les informations du formulaire avec une erreur dans l'e-mail thomas@thomas.com au lieu de thomas@facadia.com
- **When** : je clique sur le bouton submit
- **Then** : un message d'erreur s'affiche

**Test 2**

- **Given** : en tant qu'utilisateur déconnecté
- **When** : je complète les informations du formulaire avec la bonne adresse e-mail thomas@facadia.com
- **When** : je clique sur le bouton submit
- **Then** : un message d'erreur pour le mot de passe s'affiche

**Test 3**

- **Given** : en tant qu'utilisateur déconnecté
- **When** : je complète les informations du formulaire avec la bonne adresse e-mail thomas@facadia.com et le mauvais mot de passe `qwerty`
- **When** : je clique sur le bouton submit
- **Then** : un message d'erreur pour le mot de passe s'affiche

**Test 4**

- **Given** : en tant qu'utilisateur déconnecté
- **When** : je complète les informations du formulaire avec la bonne adresse e-mail thomas@facadia.com et le bon mot de passe `azerty`
- **When** : je clique sur le bouton submit
- **Then** : aucun message d'erreur ne s'affiche

Les corrections sont dans la branche : `partie-3/chapitre-2-b-solution`

### Plan de tests d'intégration

**Exercice**

Ajouter un ensemble de tests d'intégration permettant de valider le programme.

## Tests end-to-end

### Framework Selenium

Les tests end-to-end sont des tests qui simulent les interactions d'un vrai utilisateur (dans le navigateur donc), il seront réalisés en `Selenium`. Voir le TP d'introduction à Selenium avant d'effectuer cette partie.

### Premiers tests end-to-end

**Exercice**

En utilisant Selenium, implémenter les scénarios de tests suivants. Pensez à lancer l'application en parallèle !

**Test 1**

- **Given** : en tant qu'utilisateur déconnecté
- **When** : je complète les informations du formulaire avec la bonne adresse e-mail thomas@facadia.com et le bon mot de passe `azerty`
- **When** : je clique sur le bouton submit
- **Then** : je suis redirigé vers la page d'accueil des capteurs

**Test 2**

- **Given** : en tant qu'utilisateur connecté
- **When** : je suis sur la page d'accueil des capteurs
- **Then** : le noeud avec la classe `section-title` est présent
- **Then** : le noeud avec la classe `section-title` a le titre `Vos capteurs`

**Test 3**

- **Given** : en tant qu'utilisateur connecté
- **When** : je suis sur la page d'un capteur
- **Then** : le noeud avec la classe `section-title` est présent
- **Then** : le noeud avec la classe `section-title` a le titre `Détails du capteur 7`

### Plan de tests end-to-end

**Exercice**

Décrire un plan de tests complet permettant de valider le fonctionnement du programme d'un point de vue utilisateur (il s'agira donc de réaliser la recette fonctionnelle du projet). Implémenter une automatisation de cette recette fonctionnelle en utilisant le framework d'automatisation de navigateurs `Selenium`.

- On utilisera un design pattern de `PageObject`, c'est-à-dire que les pages et les éléments de l'interface graphique seront décrits dans des classes de tests dédiées (voir cours).
- On pourra également lancer l'application et réaliser des tests de fonctionalité à la main - ceux-ci seront à documenter dans le rapport.

## Rendus attendus

Il est attendu pour chaque groupe une **archive** contenant : - Le code source de l'application - Les tests unitaires et d'intégration `Jest` ; - Les tests `Selenium` dans le langage de votre choix ; - Le rapport.

Le barême est le suivant :

- 1 note sur la partie tests unitaires et d'intégration (/20) ;
- 1 note sur la partie tests end-to-end (/20) ;
- Chaque partie a le même barême :
  - 3 points pour la partie tests unitaires (ou tests end-to-end) du rapport ;
  - 6 points pour la couverture de tests (nombre de tests, choix des scénarios, ...)
    - il est attendu une bonne couverture par lignes de code pour les tests unitaires sur les classes métier du backend et du frontend (environ 80%) ;
    - il est attendu au moins 5 tests `Selenium`.
  - 6 points pour la qualité des tests (design patterns utilisés, lisibilité du code de test, ...)
  - 5 points pour le développement de l'application (aboutissement du projet, qualité du code, ...)
- Le rapport sur la réalisation de l'étude de cas devra contenir :
  - La description des plans de tests pour les tests d'intégration et les tests end-to-end
  - Expliquer les choix de réalisations de tests, par exemple : donner un exemple expliquant pourquoi vous avez choisi de tester plus fortement certaines parties du code que d'autres.

## Legal