*Tester l’interface de votre site*

**Pourquoi écrire des tests ?**

Vérifier manuellement les pages et fonctionnalités de son site n’est pas suffisant, le mieux est de rendre tout ça automatique ! Cela le protège des changements qui pourraient éventuellement casser quelque chose.

Pour tester la partie front end (en JavaScript) d’un site on peut utiliser le Framework Jasmine.

**Le BDD ou *B*ehavior *D*riven *D*evelopment**

Le BDD a pour but d’identifier exactement ou notre code de test doit nous amener. Elle se base sur une syntaxe simple destinée à décrire une situation théorique, afin de rendre plus clairs les scenarios à considérer dans l’écriture de nos tests :

1. Given
2. When
3. Then
4. And

Ou, en français :

1. Etant donné
2. Quand
3. Alors
4. Et

Exemples :

* Given a user is in Spain
* When they visit the homepage
* Then they should see the Spanish version of site
* Given a user is in USA
* When they visit the homepage
* Then they should not see the French version of site
* Given player 1 is on square 3
* When player 2 attempts to take square 3
* Then player 2 should receive an error
* And they should not be able to take square

**Le TDD ou *T*est *D*riven *D*evelopment**

Le BDD n’a pas vocation à remplacer le BDD, s’en est la suite logique Ils se complètent l’un à l’autre :

1. Identification des comportements avec le BDD
2. Ecriture de tests pour ces mêmes comportements avec le TDD

Le but du TDD est d’écrire une solution la plus basique possible pour faire passer chaque test précédemment identifié. Cela passe par le respect de trois règles simples mais indissociables l’une de l’autre :

1. Ecrire un test qui échoue avant d’écrire le code lui-même
2. Ne pas écrire de test plus compliqué que nécessaire
3. Ne pas écrire plus de code que nécessaire, juste assez pour faire passer le test qui échoue

C’est le cycle **« red-green-refactor »** ou **« Rouge-vert-refactor »** en français, soit :

1. Ecrire un test qui échoue : Résultat rouge
2. Ecrire le code le plus simple possible qui fait passer le test en 1. : Résultat vert
3. Améliorer le test si besoin est : Refactoring

**Définir des suites et des spécifications**

Les suites et les spécifications sont les deux éléments nécessaires à la mise en place de tests. Elles permettent en effet de bien structurer notre code en l’organisant par fonctionnalité. Au final nos tests ne seront intégralement composés que de suites et de spécifications.

Les suites :

Comme leur nom l’indique, les suites sont en fait un ensemble de micro-tests qui vont se concentrer sur une fonctionnalité donnée.Une suite peut bien évidemment contenir d’autres suites en son sein.

Par exemple, une suite pour vérifier qu’un lecteur vidéo fonctionne correctement devra tester :

* Est-ce que la vidéo se lit bien automatiquement lorsque la page est chargée
* Est-ce que la vidéo se lance quand quelqu’un clique sur le bouton de lecture
* Est-ce que le bouton d’arrêt arête effectivement la lecture
* Est-ce que le lecteur vidéo s’arrête lorsque la vidéo est finie
* Etc…

Les suites dans Jasmine sont écrites grâce à la fonction « **describe()**»:

*describe(”when video is played”, function() {*

*// test video playing functionalities here*

*);*

Les spécifications :

Là où les suites déclarent ce qui va être testé, les specs elles décrivent ce qui doit normalement se passer. Leur syntaxe est peu ou prou la même que celle des suite, en utilisant cette fois ci la fonction « **it()**» :

*describe(“Video player”, function() {*

*describe(“when the page loads”, function() {*

*it(”should automatically play the video”, function() {*

*// test video playing functionalities here*

*);*

*});*

*});*

**Gérer les résultats attendus**

Après avoir regroupé nos tests de manière logique, il faut s’assurer que les valeurs sorties par le code correspondent aux valeurs attendues. Cela se fait au moyen d’expectations (aussi appelées « assertions ») et de matchers.

Les expectations :

Un certain nombre bien utiles d’expectations sont nativement proposées par Jasmine mais évidemment il reste également la possibilité d’en créer des personnalisées, grâce à la fonction « **expect()** » :

*expect(videoPlaying).toBe(true);*

La fonction expect() fonctionne de la même manière que les fonctions **describe()** et **it()** à la différence que cette première ne prend qu’un seul argument (la valeur attendue) et qu’elle fonctionne de pair avec un matcher (ici « **toBe() »**).

Les matchers :

Les matchers viennent compléter les assertions. Ce sont eux qui vont comparer les valeurs attendues avec celles obtenues lors de l’exécution du code de test. Il est également possible de créer ses propres matchers mais ceux proposés par Jasmine devraient déjà couvrir la quasi-totalité des besoins.

Liste des matchers fournis avec Jasmine :

* toBeNull(x)
* toBeUndefined()
* toBeDefined()
* throw(e)
* toBeFalsy()
* toBeTruthly()
* toEqual()
* toBeGreaterThan(x)
* toBeLessThan(x)
* toBe(x)
* toContain(x)
* toMatch(x)

Pour lancer l’exécution des tests il suffit d’ouvrir le fichier « **SpecRunner.html** » présent à la racine de Jasmine.

Jasmine propose deux fonctions qui permettent d’exécuter du code avant ou après chaque assertion (spécifications). Ce sont les fonctions « **beforeEach()** » et « **afterEach()** ».