Test de l'application javascript

- Introduction
- Fonctionnement
- Composition des tests
- Liste des assertions

Introduction

J'ai mis en place un système très simple de test pour cette application entièrement en javascript.

Fonctionnement

Les tests sont chargés directement avec l'application. Mais c'est un nouveau fichier HTML qui est construit, pour ne pas toucher l'application elle-même. Donc, un fichier `test.html` est fabriqué à partir du fichier `partition.html`, en remplaçant la marque `<!-- TEST -->` par les balises script d'inclusion.

Parmi ces scripts, il y a les scripts `tests/system` qui chargent l'objet `Tests` qui va gérer les tests. Au chargement de la page (à la fin), la méthode `Tests.run` est appelée, qui va lancer tous les tests instanciés.

Chaque fichier est une instance de `Test`, mais on peut très bien imaginer d'avoir plusieurs instances de tets dans le même fichier. Instancier un test consiste à utiliser la méthode `new` et à définir la méthode `run` :

```
// Création de l'instance de test. En l'instanciant, il est enregistré
// dans la liste des tests à jouer.
var test = new Test('Nom de mon test');

// Méthode appelée par Tests.run lorsqu'elle boucle sur tous les tests
// instancié.
test.run = function(){

  // Je suis la méthode qui sera appelée par le moteur de test

  // Une méthode définie plus bas (pour la clarté)
  this.un_test_particulier();
}

test.un_test_particulier = function(){
  given('Une situation de départ');
  // Le travail
```

```
  // Les assertions pour vérifier
}
```

Les fiches de test doivent impérativement se trouver dans le dossier `./template/tests/tests/`. Dès qu'une fiche de test est créée, il faut lancer en console (Terminal) l'utilitaire `./utils/test.rb` qui va préparer les tests. `test.rb` va ajouter les balises d'inclusion de script dans le fichier HTML : tous ceux du dossier system, sans distinction, puis ceux désignés ou non par le premier argument :

```
  > ./utils/tests.rb
  # => Prépare le fichier test.html avec TOUS les tests

  > ./utils/tests.rb ts_mon_test
  # => Prépare le fichier test.html seulement pour la feuille
  #    `./tests/tests/ts_mon_test.js`

  > ./utils/tests.rb dossier/unit/
  # => Prépare le fichier test.html avec tous les tests du dossier
  #    `./tests/tests/dossier/unit/`
```

On comprend donc que le fichier `test.html` doit être produit chaque fois qu'on change de test. En revanche, lorsqu'un feuille de test est modifiée, il suffit de recharger le fichier `test.html` pour prendre en compte les changements.

Ensuite, il suffit de charger le fichier `test.html` (créé par `test.rb`) et de lire la console de Firebug pour avoir le résultat du test.

Composition des tests

Comme nous l'avons vu, la base du fichier test est simplement :

```
// Création de l'instance de test. En l'instanciant, il est enregistré
// dans la liste des tests à jouer.
var test = new Test('Nom de mon test');

// Méthode appelée par Tests.run lorsqu'elle boucle sur tous les tests
// instancié.
test.run = function(){

  // Je suis la méthode qui sera appelée par le moteur de test

  // Une méthode définie plus bas (pour la clarté)
  this.un_test_particulier();
}
```

```
test.un_test_particulier = function(){
  given('Une situation de départ');
  // Le travail
  // Les assertions pour vérifier
}
```

On utilise ensuite, à l'intérieur, des *assertions* pour tester. Les assertions (méthodes) sont pour la plupart construites sur un modèle volontairement simple :

```
<nom_assertion>(
  <evaluation>,
  <message en cas de succès>,
  <message en cas d'échec>
)
```

L'assertion la plus simple vérifie par exemple simplement que le premier argument soit vrai :

```
assert(
  2 + 2 == 4,
  'Deux + deux est bien égal à quatre',
  'Deux + deux devrait être égal à quatre'
)
```

L'assertion ci-dessus produira « Deux + deux est bien égal à quatre ».

Vous trouverez la liste des assertions ci-dessous.

Assertions

assert(evaluation, message_success, message_failure)

Assertion de base qui génère le message de succès `message_succes` quand `evaluation` est `true` et génère le message de failure `message_failure` dans le cas contraire.

```
assert(
  true,
  'Je suis vrai',
  'Je suis faux'
  )
```

```
assert(
  2 > 4,
  'Deux est bien inférieur à 4',
```

```
'Deux ne devrait pas être supérieur à 4'
)
```

Noter que pour les assertions négatives, il suffit d'utiliser `!evaluation`.

assert_classes(jqDes, classes) / assert_not_classes()

Produit un succès si l'élément ou la liste d'éléments désignés par `jqDes` possèdent les classes (Array) désignées par `classes`. Produit une failure dans le cas contraire.

```
var mesDivs = document.getElementsByClassName('divisors');
assert_classes(mesDivs, ['good', 'one'])
```

assert_position(jqDes, hposition[, tolerance]) / assert_not_position()

Produit un succès si l'élément désigné par `jqDes` se trouve dans les positions définies par `hpositions` avec une tolérance optionnelles de `tolerance`.

`hpositions` est une table définissant les valeurs (nombre de pixels, sans unité) de `x` ou `left`, `y` ou `top`, `w` ou `width`, `h` ou `height`.

```
var monDiv = document.getElementById('monDiv');
assert_position(monDiv, {x: 100, y: 200});
```

L'assertion ci-dessus génère un succès si le node `#monDiv` se trouve a un `left` de 100 pixels et un `top` de 200 pixels.

```
var monDiv = document.getElementById('monDiv');
assert_not_position(monDiv, {w: 100, h:30}, 10);
```

Le code ci-dessus génère une failure si l'objet `#monDiv` a une largeur comprise entre 90 et 110 et une hauteur comprise entre 20 et 40.

On peut transmettre une liste (Array) le nœuds à la méthode.

assert_visible(jqDes) / assert_not_visible

Produit un succès si l'élément désigné par `jqDes` est visible, une failure dans le cas contraire.

```
assert_visible('#monDiv');

assert_not_visible('.mesDivs');
```