Manuel InsideTest

InsideTest permet de lancer des tests dans le module à tester lui-même, afin d'avoir une suveillance permanente en cours de développement.

Le fonctionnement est cependant assez différent des autres modules de test, il est spécialement pensé pour les tests répétitifs, c'est-à-dire qui testent le résultat de nombreuses valeurs. L'idée est alors que créer un test, puis de passer ces valeurs (appelées `sujets`) par ce test.
Par exemple :

```
import { InsideTest, page, mouse } from '../../system/InsideTest/inside-test.lib.js'

InsideTest.reset(); // pour le moment, juste pour l'index du test

// Un test qui doit vérifier que les valeurs sont bien égales à 4
var test = new InsideTest({
    error: '%{devrait} être égal à 4.'
  , eval: function(sujet){
      return sujet == 4
    }
  , afterServerEval: function(resultat) {
      // appelé si on travail avec une opération serveur
    }
})
// On fait les tests
test.with(2) // => une erreur en console "«« 2 »» devrait être égal à 4."
test.with(3) // => idem
test.with(4) // succès (invisible par défaut)
```

```
test.with(2 + 2) // => succès

test.withNegate(4) // => Erreur "«« 4 »» ne devrait pas être égal à 4."
test.withNegate(3) // => succès
```

Définition des messages

Comme on peut le voir ci-dessus, on définit simplement les messages d'erreur à l'aide d'un texte simple qui doit pouvoir marcher (mais pas forcément) avec les tests négatifs. On utilise pour ça la valeur template `%{devrait}` ou `%{doit}` qui suivant le résultat se transformera en "devrait"/"ne devrait pas" ou "doit"/"ne doit pas".

Les tests ne produisent aucun retour en cas de succès, ils ne sont là que pour déceler les problèmes.

Mise en forme plus complexe des messages

Pour une mise en forme plus adaptée aux circonstances, on peut utiliser les balises `%{sujet}`. Par exemple :

```
var test = new InsideTest({
  error: 'Quand le sujet est %{sujet}, le résultat %{devrait} être bon.'
  ...
})
```

Plusieurs messages

On peut envoyer plusieurs messages au cours du même test si nécessaire, ou un seul mais qui interrompt le test avant la fin. Par exemple, imaginons un test qui cherche à savoir si tous les prénoms voulus sont bien là. On pourrait avoir :

```
var test = new InsideTest({
    error: '%{devrait} passer.'
  , eval: function(){
      var une_erreur = false
      if (page.not.contains('Marion')) {
        InsideTest.error = "Marion devrait être dans la liste"
        // note error= est une méthode cumulative
        une_erreur = true
      }
      if (page.not.contains('Philippe')){
        InsideTest.error = "Philippe devrait être dans la liste"
        une_erreur = true
      }
      if (page.not.contains('Élie')){
        InsideTest.error = "Élie devrait être dans la liste"
        une_erreur = true
      }
```

```
      return une_erreur
    }
})
```

> **Note importante** : contrairement aux apparences, si les trois noms sont absents de la page, les 3 messages seront affichés.. La méthode `error =` est cumulative.

Activation des tests

- Les définir dans chaque module ou dans des modules séparés,
- mettre une constante `INSIDE_TESTS` à la valeur `true` avant l'appel de la méthode suivante,

```
const INSIDE_TESTS = true
```

- une fois le programme chargé (dans le `$(document).ready` par exemple), placer la commande :

```
InsideTest.run()
```

Les méthodes de test

Toutes ces méthodes s'appliquent à une instance `InsideTest` qui définit au moins la méthode `eval()` :

```
var test = new InsideTest({
  eval: function(){return true}
})
```

.with(sujet[, expected])

Quand on veut seulement tester un sujet.

Si `expected` est défini, la méthode se comporte comme `.withExpected`

Bien prendre en compte les problèmes d'égalité.

.withNegate(sujet[, expected])

Inverse de la précédente.

.withExpected(sujet, expected)

Fournit le `sujet` à la méthode de test et espère qu'elle retournera le résultat `expected`. Bien prendre en compte les problèmes d'égalité.

Exemple :

```
var test = new InsideTest({
    error: '%{devrait} correspondre.'
  , eval: function(sujet){
      return `Bonjour, ${sujet} !`
    }
})

test.withExpected('John', 'Bonjour, John !') // => succès
test.withExpected('Renée', 'Bonjour, Renée !') // => succès
test.withExpected('Al', 'Au revoir, Al')
// => échec. En console :
// «« Al »» devrait correspondre.
// Attendu : "Au revoir, Al"
// Obtenu  : "Bonjour, Al !"
```

> **Note** : bien comprendre que la méthode `eval` du test doit retourner la valeur qui sera comparée à `expected`. La méthode ne doit pas retourner quelque chose qui s'apparenterait à `traitement(sujet) == expected`

.withExpectedNegate(sujet, expected)

Inverse de la précédente.

.equal(sujet, expected)

Teste l'égalité entre `sujet` et `expected`.

Cette méthode prend en compte les problèmes d'égalité.

.exec()

Pour un test sans sujet.

Par exemple :

```
var test = new InsideTest({
    error: '%{doit} créer une instance Scenario'
  , eval: function(){
      const sce = new Scenario()
      return sce instanceof Scenario
    }
})
test.exec() // => true si tout se passe bien
```

Travail complexe quand tests côté serveur

Les tests se compliquent lorsqu'on doit travailler avec le serveur avec WAA car la méthode `WAA.send` ne renvoie pas son message à `InsideTest` puisque c'est un module.

- On passe par **`IT_WAA.send`** côté javascript
- **`WAA.send(class:'IT_WAA', method:'receive', data:{testId:..., result:..., data:...})`** côté serveur (noter que les données `:testId`, `:result` et `:data` sont obligatoires)
- la méthode **`afterServerEval`** d'InsideTest si on doit poursuivre le test avec le résultat.

Exemple concret d'utilisation

On définit le test :

```
var test ;

test = new InsideTest({
    error: "Le test %{doit} passer."
  , eval: (sujet) => {
      // Envoi au serveur
      IT_WAA.send({class:'MonApp',method:'maMethod', data:{sujet:sujet}})
      return true // Pour ne pas faire de faux négatif
    }
  , afterServerEval:(data)=>{... on la verra plus bas ...}
})
```

Côté serveur :

```
class MonApp
  def self.maMethod(data)

    # Le test
    ok = sujet > 4
```

```
    error = ok ? nil : "Le sujet #{sujet} devrait être > 4."
    result = {ok: ok, error: error}

    # Renvoie au client
    WAA.send(class:'IT_WAA',method:'receive', data:{
        testId:     data['testId'],
        testIndex:  data['testIndex'],
        result:     result,
        data:       {time: Time.now.to_i}
      })
  end
end
```

Et retour côté client :

```
test = new InsideTest({
  //...
  , afterServerEval:(data) => {
      if ( data.result.ok ) {
        // Traitement si c'est bon
      }
      return data.result.ok
    }
})
```

```
Lorsqu'une méthode eval de test doit appeler WAA.send, elle prévient l'application
qu'elle le fait.
Un traitement se fait côté serveur, qu'on imagine long.

Lorsque InsideTest arrive en fin des tests, il interroge l'application pour savoir
si un travail serveur est encore en train de se faire.
Si c'est le cas, il attend avant de montrer les résultats.

Lorsque les tests côté serveur ont été effectués, le serveur prévient l'application.
Lorsque InsideTest interroge l'application, on lui dit alors que c'est fini et on
lui donne les résultats.

InsideTest peut alors clore l'opération et afficher les résultats.
```

Concrètement, tout ça se passe avec la classe **`IT_WAA`** qu'il faut charger par `IT_WAA.js` comme un script normal dans l'application (ça se fait automatiquement par `InsideTest.install()` qui ajoute une balise pour ce script.

Ensuite, on appelle les méthodes serveur avec :

```
IT_WAA.send({data})
```

… où `data` contient les propriétés normales, auxquelles sont ajoutées par le programme la propriété `testId` qui devra être remonté ensuite pour savoir que le test a fini son travail.

> Cela signifie qu'il faut appeler des méthodes propres aux tests, qui seront susceptibles de retourner ce identifiant du test. Dans le cas contraire, les tests ne s'arrêteront jamais (un timeout les arrêtera cependant)

Côté serveur, on se sert de la méthode `WAA.send` normale, mais en appelant :

```
WAA.send(class:'IT_WAA', method:'receive', data:{<data>})
```

> Rappel : `<data>` doit impérativement définir `testId`, l'identifiant du test.

```
// Données :
// --------

testId    : IDentifiant du test {InsideTest}
testIndex : Index du test précis dans le stack du InsideTest[testId]
            // Chaque fois qu'on a un test.with("sujet") par exemple,
            // ça crée un nouvel index.
```

Fonctionnement

Dans `InsideTest`, une fois que tous les tests ont été joués par **`InsideTest.runStack()`**…

… on se met en attente dans la méthode **`InsideTest.awaitForAllServerCheckDone()`**. Cette méthode interroge `IT_WAA` pour savoir si la classe est en activité.

Dans **`IT_WAA`**, si aucun test serveur n'a été lancé, la propriété **`working`** (`IT_WAA.working`) est à `false` et l'on peut achever les tests et afficher le résultat.

En revanche, si des tests serveurs ont été lancés (rappel : depuis un inside-test normal, mais avec `IT_WAA.send`), alors la méthode **`InsideTest.checkerServerResultats`** est appelée tous les demi-secondes pour relever dans **`IT_WAA.stackServerResultats`** tous les résultats remontés par le serveur (rappel : grâce à l'appel `WAA.send(class:'IT_WAA', method:'receive', data:{testId:<test-id>, result:{ok:..., error…}, data:{<autres données}})` côté ruby.)

Annexe

Problèmes d'égalité

Avec Javascript, les égalités ne sont pas pratiques... Par exemple, `[1,2,3]` ne sera pas égal à `[1,2,3]`...

Quand on veut tester une telle égalité, soit on utilise `JSON.stringify` (ou `JString(…)`) à l'intérieur de la fonction :

```
var test = new InsideTest({
  eval: function(sujet, expected){
    // ...
    return JString(sujet) == JString(expected)
  }
})
```

... Soit on utilise la méthode de test `equal`.