Manuel des MiniTests (de l'intérieur)

> Note : c'est une version allégée des inside-tests (qui ne me plaisent pas encore)

Mise en place des tests

- Implémenter **le fichier** `./ajax/_scripts/MiniTest/before_suite.rb` pour définir ce qu'il faut faire au début des tests (avant le premier.
- Implémenter **le fichier** `./ajax/_scripts/MiniTest/after_suite.rb` pour définir ce qu'il faut faire à la fin des tests (par exemple remettre la configuration du départ)
- Implémenter **le fichier** `./ajax/_scripts/MiniTest/gel.rb` pour définir comment produire les gels.
- Implémenter **le fichier** `./ajax/_scripts/MiniTest/degel.rb` pour définir comment dégeler les gels.
- Implémenter **tous les tests dans le dossier** `./js/MiniTests/`. Ce sont de simples fichiers javascript. cf.leur implémentation.
- Implémenter le fichier définissant la **liste des tests à jouer**, dans `./js/MiniTests/_tests_list.js`.

Liste des tests à jouer

La liste des tests à jouer doit être définie dans le fichier `./js/MiniTests/_config.js` grâce au code :

```
// _config.js
'use strict'

MiniTest.config = {
  ...
}

MiniTest.tests_list = [
    '<affixe premier test>'
  , '<affixe second test>'
  , ...
  , '<affixe dernier test>'
]
```

Expectations

Les "expectations" (attentes) ont toutes la forme :

```
expect(<sujet>).<verbe>(<valeur>).else(<message d'erreur>)
```

Par exemple :

```
expect(2 + 2).eq(4).else("2 + 2 devrait être égal à 4. Il vaut #actual.")
```

La valeur attendue et la valeur réelle, lorsqu'elles peuvent s'exprimer, peuvent être introduire dans les messages en utilisant respectivement `#expected` et `#actual`.

Toutes les méthodes d'expectation sont défines dans le fichier js/required-2/system/MiniTests/Expectations.js.

Pour une liste complète des méthodes d'expectation, ouvrir une fenêtre de Terminal au dossier de l'application et jouer :

```
> bin/minitest-expectations
```

Réinitialisation de l'application

Les tests s'exécutant à l'intérieur même de l'application, il est nécessaire de la ré-initialiser au début de chaque test.

Pour ce faire :

- créer une méthode **`App.resetBeforeTest()`** qui réinitialisera toutes les valeurs nécessaires (à suivre de près au cours du développement),
- définir dans le fichier `./js/MiniTests/_config.js` que l'application doit être ré-initialisée à chaque nouveau tests :

  ```
  MiniTest.config = {
    ...
    , reset_before_each_test: true
    ...
  }
  ```

- OU ALORS appeler cette méthode à chaque début de test :

  ```
  MiniTest.add("Mon test avec réinitialisation", async function(){
    App.resetBeforeTest()
    degel("mon-gel-utile")

    // ... le test ...

    return ok
  })
  ```

Noter que le reset de l'application sera également **appelé après chaque dégel**. Prendre en compte le fait que dans ces cas-là, la réinitialisation se fait deux fois : une fois avant le dégel, avant le test, et une fois après le dégel. Si la réinitialisation est longue, les tests peuvent être très ralentis.

Les Gels

Pour utiliser des gels, on invoque les méthodes d'helpers `gel` et `degel`.

Utilisation d'un gel

```
MiniTest.add("Mon test qui utilise un gel", async function(){

  await gel("mon-gel")

  // ... procéder ici au test à partir du gel ...

  return resultat
})
```

Production d'un gel

```
MiniTest.add("Mon test pour créer un gel", async function(){

  // ... des opérations qui doivent créer un état ...

  await gel("nom du gel à partir de l'état courant", `Description
La description précise du gel.
L'état où l'on en est.
`)

  return ok
})
```

Pour fonctionner, il faut définir pour chaque application la méthode de gel et de dégel. Ce sont des fichiers à mettre dans `./ajax/_scripts/MiniTest/` avec les noms `gel.rb` et `degel.rb`.

```
# gel.rb

gel_name = Ajax.param(:gel_name)
gel_desc = Ajax.param(:gel_description)

# ... opérations à faire pour produire le gel voulu ...

Ajax << {message: "Gel #{gel_name} produit avec succès."}
```

```ruby
# degel.rb

gel_name = Ajax.param(:gel_name)

# ... Opérations à faire pour dégeler le gel ...

Ajax << {message: "Gel #{gel_name} dégelé avec succès."}
```

Les Tests

Implémentation d'un test

La base de la définition d'un test est :

```
'use strict'

MiniTest.add("<nom de mon test>", async function(){

  // ... opérations de test ...

  return resultat
})
```

On notera :

- la fonction asynchrone en seconde paramètre,
- le retour, toujours, d'un résultat. Ce résultat peut être [de différents types][#retours-test].

Retour des tests

Le retour des tests peut avoir des valeurs très différentes. Je les résume ici, et les illustre plus bas :

> Note : on parle ici des valeurs réelles, strictes (`null` n'est pas `false`)

- `true` pour un test réussi, qui affiche son nom (et son path si nécessaire),
- `false` pour un test raté, qui affiche seulement son nom et son path,
- `null` pour un **test réussi**, quand c'est un message précis qui doit être renvoyé comme résultat.
- un string valeur 'pending' pour un test en attente (non encore implémenté),
- un `String` quelconque, différent de 'pending', pour un message d'erreur précis.

Exemple de motif précis d'erreur

```
MiniTest.add("Mon test avec erreur précise", async function(){
  let motif_echec = null // en cas de succès

  // ... des opérations ...
```

```
  await waitFor(page.has.bind(page,'div#mon-div')).catch(ret=>{
    // On définit ici le motifi d'échec
    motif_echec = "Le div #mon-div est introuvable. Je dois renoncer."
  })

  // Pour interrompre tout de suite
  if ( motif_echec != null ) return motif_echec

  // Ou :

  if ( motif_echec == null ) {

    // ... pour poursuivre le test ...

  }

  return motif_echec // on le retourne
})
```

Messages en cours de test

Dans tous les cas, pour afficher des messages provisoires ou non, on peut utiliser les méthodes de console habituelles :

```
console.log(...)
console.error(...)
console.warn(...)
```

Mais ces messages “pollueront” un peu les retours de test.

On peut utiliser plus profitablement la méthode `log` qui permet de ne rien produire en mode “quiet” avec un `debug_level` à 0 (défini dans la configuration des tests).

```
log("Mon message sensible", 1)
// Ce message ne s'affichera que si les debug_level est supérieur ou
// égal à 1

log("Mon message insensible", 9)
// Ce message ne s'affichera que si le debug_level est supérieur ou
// égal à 9
```

Configuration des tests

La configuration des tests est définie dans le fichier `./js/MiniTests/_config.js`

On trouve :

```
MiniTest.config = {

  // Juste pour la virgule
  config: true

  // Pour définir le mode de débuggage. Tous les messages
  // en dessous de cette valeur seront affichés.
  , debug_level: 0

  // Pour réinitialiser l'application avant chaque test
  // cf. la section qui en parle
  , reset_before_each_test: true
}
```

Méthodes d'helpers

Méthodes propres à l'application

Avant toute chose il convient de rappeler que puisque ces mini-tests se font à l'intérieur même de l'application, on peut utiliser toutes les méthodes de cette application (notamment pour faire des tests unitaires mais aussi pour créer des méthodes d'helpers propres). Ces méthodes peuvent même se trouver dans des modules du dossier `./js/MiniTests/` qui ne seront pas des tests.

Méthodes de test de la page

page.has(`selector`[, `<attributs>`])

Retourne `true` si la page contient le sélecteur et ses attributs.

`<attributs>` peut définir aussi `in` pour chercher dans un container particulier et `count` pour trouver exactement le nombre d'éléments voulus.

page.contains(`<message>`)

Retourne `true` si la page contient le message.

Méthodes d’interaction avec la page

page.click(`<selector>`[, `<params>`])

Pour cliquer sur l’élément spécifié.

Par exemple :

```
MiniTest.add("mon test du click", async function(){

  page.click('button#mon-bouton', {in: "div#mon-div"})

})
```