Manuel des tests

Introduction

Les « FITests » (« From Inside Tests ») permettent de lancer des tests de l'intérieur même de l'application. De ce fait, leur utilisation est simplissime en regard des autres tests qui, pour les tests unitaires, d'intégration et fonctionnels nécessitent toujours des réglages particuliers et délicats.

Principaux généraux fondateurs

De l'intérieur

Comme indiqué, les tests se lancent de l'intérieur même de l'application à tester, c'est-à-dire qu'ils ont

accès à tous ce que peut atteindre l'application. Pas de modules à requérer, etc.

Sujet complexe une lettre

La plupart des sujets complexes) utilisés dans la formule `expect(<sujet complexe>)` fonctionnent en une lettre minuscule. Par exemple, si l'on parle d'un élément du dom, ce sera la lettre `d` comme « DOM » : `d("#monDiv")`. Si c'est un fichier, ce sera `f` comme « file » : `f("mon/path")`. Voir tous les « sujets complexes une lettre »

> Noter de ce fait qu'il ne faut pas utiliser des méthodes en une lettre minuscule dans l'application (il ne faut donc pas tester une application uglifiée et minifyisée).

Utilisation des FITests

Pour utiliser les FIT-Tests (persos) comme ici, on doit :

- mettre le dossier `TestsFIT` dans un dossier librairie de l'application,
- le requérir avec `require("./path/to/fitests/folder/")`.
- créer un dossier `__TestsFIT__` à la racine de l'application pour définir les tests,
- définir dans le fichier `./__TestsFIT__/config.json` la configuration des tests. cf. Configurer les tests

- Dans `package.json` de l'application, on ajoute :

```
"scripts":{
  //...
  "testsfits": "MODE_TEST=true npm start"
  //...
}
```

- On peut définir dans `./__TestsFIT__/support/` les fichiers et méthodes utiles aux tests de l'application courante.

- On définit la méthode `App.runtests()` de cette manière (on peut définir une autre méthode dans un autre objet, mais alors il devra être accessible et utilisé pour lancer les tests) :

```
App.runtests = function(){
    NONE === typeof(Tests) && ( global.Tests = require('./path/to/folder/fitests')
  )
    Tests.run()
  }
```

- On définit les tests dans le dossier `./__TestsFIT__/`.

Pour lancer les tests :

- Le plus simple est de faire un menu qui appelle la méthode `App.runtests()` (ou directement la méthode `Tests.run()` si l'objet `Tests` est déjà requis).
- On peut aussi lancer l'application et ouvrir la `console dev` dans laquelle on tape `App.runtests()`.

Note : le fichier `config.json` va permettre de filtrer les tests à passer. Cf. Liste des tests à lancer pour le détail. Pour le moment, on ne peut pas le faire en ligne de commande.

Configurer les tests

On configure les tests dans le fichier `./__TestsFIT__/config.json`.

`trace`
: Si true, on garde au maximum tout ce qui a été créé/modifié à la fin des tests, pour pouvoir vérifier des choses.
Par exemple, à la fin des tests, les fichiers créés à la volée (dans support/files), ou les mails écrits (dans support/mails) sont automatiquement détruits. Si `trace` est mis à `true` dans la configuration des tests, tous ces éléments sont conservés à la fin des tests (mais détruits au prochain lancement).

`onlyFolders`
: Liste des seuls dossiers, dans `./__TestsFIT__/`, qu'il faut traiter.
Ou null.

`random`
: Si true, les tests et les cas sont traités en ordre aléatoire.
Noter que pour les cas, ce sont seulement les cas d'un même test qui sont mélangés.

`fail_fast`
: Si true, les tests s'arrêtent à la première failure rencontrée. Sinon, on va jusqu'au bout des tests.

`regFiles`
: Expression régulière à utiliser pour filtrer les fichiers de test.
Elle sera escapée, dont inutile de le faire.

`regNames`
: Expression régulière à utiliser pour filtrer les tests à jouer.

`regCases`
: Expression régulière à utiliser pour filtrer les cas à jouer.
On peut utiliser par exemple "ONLY" et ajouter « ONLY » au seul cas qu'on veut traiter pour ne traiter que lui.

Définition d'une feuille de test

On appelle « feuille de test » un fichier `js` définissant le ou les tests à exécuter pour un cas particulier (ou toute l'application, si elle est simplissime).

La structure de ce fichier est :

```
'use strict'

var t = new Test("Le titre du test (affiché en titre)")

t.case("Un cas particulier du test", async /* [1] */ () => {
  // ici les tests et assertions
  // [1] Si le cas contient des assertions asynchrones ou des waits.
})

t.case("Un autre cas particulier du test, asynchrone", async () => {
  // Ici les tests des autres cas

  // Pour gérer l'asynchronicité
  await expect(domId).toExistsInDom({success: 'Le truc existe', onlySuccess:true})
  // OU :
  await expect(domId).toExistsInDom({onlySuccess:'Le truc existe'})
  // ... on poursuit les tests avec…
```

```
})

t.case("Un cas avec une attente", async () => {

  await wait(2000)
  // Le code à exécuter 2 secondes plus tard

})

// etc.

module.exports = [t]
```

Noter l'utilisation d'une fonction `async` pour pouvoir utiliser `await`.

Syntaxe `describe`

On peut utiliser aussi la syntaxe en `describe` qui ne nécessite pas d'export :

```
describe("Nom du test", () => {

  this.before(() => {/* à faire au début (Promise) */})
  this.after(()=> {/* à faire à la fin (Promise) */})

  this.case("nom du cas", async () => {

    /* ... les tests ici ... */

  })

  this.case("Autre cas", async () => {
    //...
  })

})
```

Mot clé `not`

On utilise le mot clé `not` pour inverser le sens de l'assertion.

```
expect(2+6).equals(7)
// => échec

expect(2+6).not.equals(7)
// => succès
```

Mot clé `strictly`

On utilise le mot clé `strictly` pour demander une comparaison stricte.

```
expect('12').equals(12)
// => succès
```

```
expect('12').strictly.equals(12)
// => échec
```

Les expectations

On appelle ici « expectation » la partie d'une assertion complète. Elle se résume à :

```
expect(sujet[, options])
```

Le sujet peut être un sujet complexe ou tout autre élément qui peut être comparé.

`options`, pour le moment, ne sert qu'à décrire comment sera présenté le sujet dans les messages. Au plus simple, on peut mettre simplement en string la valeur dont l'on veut voir désigner le sujet.

Par exemple :

```
expect(2+2).is(4)
//=> écrit "4 est bien égal à 4"

expect(2+2,'2+2').is(4)
// ou
expect(2+2,{sujet:'2+2'}).is(4)
// => écrivent "2+2 est bien égal à 4"
```

Les Assertions

Les assertions s'utilisent de cette manière :

```
  expect(<sujet|valeur>).<assertion>(<valeur attendue>)
```

Par exemple, pour tester qu'une fonction existe pour un objet, on utilise l'assertion `responds_to` :

```
  expect(object).responds_to('functionName')
```

Toutes les assertions utilisables sont définies dans le dossier `required/Assertions/`.

On peut définir dans le dossier `support/assertions` les assertions propres à l'application testée.

Assertions usuelles

Les assertions usuelles, standards, sont définies dans le dossier `FITests/lib/required2/Assertions/`.

Création d'assertions

On crée les nouvelles assertions, propres à toute l'application dans le dossier `__TestsFIT__/support`, à l'aide de :

```
const MesAssertions = {
  uneAssertion(){
```

```
    //...
  }
, uneAutreAssertion(){
    //...
  }
}

FITExpectation.add(MesAssertions)
```

Noter qu'avec la définition ci-dessus, les assertions seront utilisables pour n'importe quel sujet. Pour faire des assertions propres à des sujets particuliers, utiliser les sujets complexes.

Ensuite, on peut tout simplement faire :

```
describe("En utilisation mes assertions", function(){
  this.case("J'utilise la première", async () => {
    expect('monsujet').uneAssertion()
  })
  this.case("J'utilise la seconde", async () => {
    expect('monsujet').not.uneAutreAssertion({onlyFailure:true})
    // OU :
    expect('monsujet').not.uneAutreAssertion({onlyFailure:'Le message d'échec'})
  })
  this.case("J'utilise la seconde", async () => {
    expect('monsujet').strictly.unestrict({onlySuccess:true})
  })
})
```

Codage de l'assertion

Une assertion est composée de deux parties :

```
affirmation(expected[, options]){
  /**
    Une instance pour travailler le résultat de façon plus souple
  **/
  let resultat = new FITResultat(this,{
        sujet: "Le sujet de l'assertion",
      , verbe: "LeVerbe" // doit être connu de FITExpectation.positivise
      , comp_verbe: "complement optionnel" // p.e. 'visible' si verbe 'est'
      , objet:  "L'objet traité par l'assertion"
      , options: options || {}
    })
    /**
      Estimation principale
      =====================

      Estimation, comparaison de la valeur actuelle et attendue
      éventuellement en fonction de 'not' et 'strictly'
      Cette partie doit définir `pass` qui sera TRUE si c'est un succès
      (quelle que soit la valeur de 'not') et FALSE si c'est un échec.

      On ne doit pas mettre de messages dans ce validIf, contrairement aux
      conditions suivantes.
    **/
    let estimation = ...
    resultat.validIf(estimation)
```

```
    /**
      Peut-être y a-t-il d'autres conditions
    **/
    if ( resultat.valid && options.doit_etre_grand ){
      estimation = /* est-il grand ? */
      resultat.validIf(estimation, "il est grand", "il n'est pas grand")
      // Le premier message sera ajouté si c'est un succès, le second message,
      // précédé de "mais" sera ajouté en cas d'échec
    }

    /**
      D'autres sous estimations…
    **/

    /**
      Ajouts, en cas d'échec, d'un message plus complet qui permettra de
      vraiment comprendre d'où vient l'erreur.
    **/
    if ( resultat.invalid ) {
      resultat.detailFailure = "Le message détaillant l'échec."
    }

    /**
      On produit l'assertion proprement dite, qui sera écrite dans le
      rapport d'erreur comme un succès ou un échec
    **/
    assert(resultat)
}
```

Version encore plus courte :

```
affirmation(expected, options){
  const esti = ...
  const pass = this.positive === esti
  const expe = ... // expected à écrire
  const msgs = this.assertise('<verbe>', '<complément>', this.subject, expe)
  assert(pass, ...msgs, options)
}
```

Les options peuvent déterminer si le message ne doit s'afficher que si la condition est `false`, pas exemple, avec `onlyFailure:true`. Cf. Options des assertions.

Rédaction des messages positifs et négatifs

Pour simplifier la rédaction des messages de la méthode `assert`, on peut utiliser les méthodes `assertise` et `positivise` des instances de `FITExpectation` qui connait déjà un certain nombre de verbes. On lui envoie un verbe de base qu'elle doit connaitre (par exemple « est »), optionnellement un complément, et elle revoit les deux messages de succès (`success`) et d'échec (`failure`).

> Comme la méthode `positivise` est une méthode de l'instance `FITExpectation`, donc de l'expectation elle-même, elle sait si l'assertion est positive ou non et renvoie le message en conséquence.

Par exemple, avec une assertion négative (`not`) :

```
this.positivise('est', 'égal à')

// => {
//      success: "n'est pas égal à"
//      failure: "ne devrait pas être égal à"
//    }
```

Je peux donc définir :

```
  expect(actual).strictly_equals(expected)
```

J'implémente :

```
equals(expected, options){
  const esti = this.strict ? (this.sujet === expected) : (this.sujet == expected)
  const pass = this.positive === esti
  const msgs = this.assertise('est', 'égal à', actual, expected)
  assert(pass, ...msgs, options)
}
```

Options des assertions

Note : pour les sujets complexes, on peut définir toutes ces valeurs dans la propriété `options`.

`onlyReturn`
: Si `true`, on ne produit pas de résultat, on renvoie seulement la valeur booléenne de l'assertion.

`onlyFailure`
: si `true` (ou le message d'échec), le succès reste silencieux, seul la failure écrit un message.

`onlySuccess`
: si `true` (ou le message de succès), la failure reste silencieuse, seul le succès écrit un message.

`success`, `failure`
: Forcer un message de succès ou d'échec différent du message par défaut.
On peut aussi mettre explicitement `success:false` ou `failure:false` dans les options (dernier argument de l'assertion) pour indiquer de ne pas écrire de message.

`ref`
: si `true`, on indique la valeur du sujet dans certaines assertions.
Par exemple, pour l'égalité, au lieu du message "La somme est juste", on obtiendra "la somme (2+2:number) est juste".

Sujets complexes (`expect(sujet)`)

Les « sujets complexes » sont une des fonctionnalités les plus puissantes des *FITests*. Il permet de définir un comportement propre à l'application de façon très simple.

Les « sujets complexes » permettent de définir des sujets propres à l'application — donc des éléments à mettre en premier argument d'un `expect` — avec tout ce qu'il faut pour les estimer. Imaginons par exemple qu'une classe `EventForm` permette de générer des formulaires (instances). Soit `EventForm.current`, dans l'application, la propriété qui retourne l'instance du formulaire au premier plan, le formulaire courant. On peut faire un sujet de ce formulaire courant.

```
class CurrentFormSubject extends FITSubject {
constructor(){
  super('CurrentEventForm')
```

```
  this.assertions = {
      is_opened: this.is_opened.bind(this)
    , is_closed: this.is_closed.bind(this)
  }
}
is_opened(options){
  let resultat = this.newResultat('est','ouvert',options)
  resultat.validIf(DOM.displays('#form....'))
  assert(resultat)
}
is_closed(){
  // ...
  assert(resultat)
}
}

// Pour obtenir une nouvelle instance chaque fois
Object.defineProperties(global,{
  CurrentForm:{get(){return new CurrentFormSubject() }}
})
```

De cette manière, on pourra faire :

```
expect(CurrentForm).is_opened()
```

Ci-dessous, c'est l'ancienne tournure :

```
// Dans un fichier de `__TestsFIT__/support/`
const sub = new FITSubject("Le formulaire courant")

global.CurrentForm = sub // pour l'exposer
```

On va pouvoir déterminer plusieurs propriétés de cet instance dont les plus générales sont :

- la valeur (le premier argument habituel de `expect`) : `sub.value = ...`
- le nom (à marquer dans les messages) : `sub.subject_message = "..."`
- les assertions : `sub.assertions = ... objet contenant les assertions.`
- Les options : `sub.options = {prop:value, prop:value, ...}`

Par exemple :

```
const sub = new FITSubject("Mon formulaire courant")
sub.value = EventForm.current // retourne l'instance du formulaire courant
sub.subject_message = "Le formulaire courant"
sub.options = {noRef: true}
sub.assertions = {
  est_ouvert(){
    //... on teste pour voir s'il est ouvert
    assert(
      ok, ...
    )
  }
, est_bien_rempli(){
    // ...
  }
// etc.
```

```
}
```

Dans le cas où la valeur doit changer dynamiquement, on peut faire une sous-classe de `FITSubject`.

```
class MonSousSujet extends FITSubject {
  constructor(){
    super(this.name)
    this.name = 'Mon sous-sujet'
    ///...
    this.value = // une valeur dynamique, par exemple la fenêtre courante
    this.assertions = {
      est_bien(){
        //...
        assert(/*...*/)
      }
    , est_avant(quoi){/* ... */}
    }
    // Pour pouvoir utiliser les données et les méthodes de cette instance
    // car le mot `this` ne fera pas référence à cette instance (les méthodes
    // sont "collées" aux expectations)
    this.assertions.i = this
  }
}

// Pour créer une nouvelle instance dynamique à chaque appel
Object.defineProperties(global,{
  // Créera une nouvelle instance à chaque appel
  MonSousSujet:{get(){return new MonSousSujet()}}
})
```

Il suffit ensuite de l'utiliser comme :

```
  const subj = MonSousSujet // une instance toute fraiche, donc avec la fenêtre
  expect(subj).est_bien()
  expect(subj).est_avant('ca')
```

Sujets complexes existants

Sujets complexes une lettre

`a` `a([Array|List])`
: Un Array. `a(1,2,3)`

`b` `b(Boolean)`
: Un booléen.
: Pour estimer si c'est vrai ou faux.

`d` `d(String|DOMElement|jQuerySet)`
: Un élément DOM. `d("div#monDiv")`

`f` `f(String|File)`
: Un fichier. `f("/path/to/my/folder/")`

`n` `n(Number|Integer|Float)`
: Un nombre. `n(12+23)`

`o` `o(Object)`
: Un objet. `o({prop:value,prop:value})`

`x x('test à évaluer')`
: Une expectation — et seulement une expectation — à évaluer.
Utile uniquement pour les tests de FITests.
P.e. `expect(x('expect(a(12,2)).contains(12)').succeeds()`
Noter que `{onlyReturn:true}` sera ajouté à la fin pour que l'évaluation ne produise pas de résultat écrit. Sinon, une failure attendue apparaitrait comme une failure.
Les deux assertions possibles sont `succeeds` (quand une réussite est attendue) et `fails` (quand un échec est attendu).

Élément DOM courant (`FocusedElement`)

Le sujet complexe `FocusedElement` permet de travailler et tester l'élément courant, par exemple pour voir si c'est bien le focus courant.

À la base, pour tester si l'élément courant est le bon, on se sert de l'identifiant DOM et l'on fait :

```
expect(FocusedElement).is({id:'identifiant_attendu'})
```

Méthodes pratiques

wait(`<durée>`, `<message>`)

Permet d'attendre avant de poursuite.

```
t.case("Un cas d'attente", async () => {
  // ...

  await wait(3000, "J'attends 3 secondes")
  //... on peut poursuivre 3 secondes plus tard

})
```

Méthode `waitFor`

Attend qu'une condition soit vraie (premier argument) avant de poursuivre.

```
  await waitFor(this_condition_must_be_true[, options])
```

`options` peut définir un `timeout` qui produira une erreur s'il est atteint. Sinon, on utilise le timeout général défini (`Tests.TIMEOUT`).

Méthodes pratiques sur les fichiers/dossier

`removeFile(<fpath>[, <humanName>])`
: Permet de détruire le fichier `fpath` désigné par `humanName`.
Par exemple : `removeFile('./mon/fichier.js', "Mon fichier")`
Note : la fonction produit une erreur fatale si le fichier n'a pas pu être détruit.

Liste des tests à lancer

Pour filtrer les tests à lancer, on se sert du fichier de configuration `config.json`.

Exécutions avant et après

Noter la simplicité du nommage :

- `before_tests` concerne le code à jouer avant tous les tests (`tests` au pluriel)
- `before_test` (ou `before`) concerne le code à jouer avant un test particulier (`test` au singulier)
- `before_case` concerne le code à jouer avant les cas (correspond au `before_each` des autres frameworks de test).

Exécutions avant et après la suite entière de tests

Pour le code à évaluer avant le test courant, voir exécution avant et après le test courant

Pour définir le code à jouer avant ou après l'ensemble de la suite de **tous les tests**, on implémente les méthodes `beforeTests` et `afterTests` respectivement dans les fichiers `./__TestsFIT__/before_tests.js` et `./__TestsFIT__/after_tests.js`.

Par exemple :

```
// Dans before_tests.js
'use strict'

module.exports = async /* ou pas */ () => {
  alert("Je dois afficher ça.")
}
```

Les deux méthodes, par défaut, sont considérées comme asynchrones.

Par exemple, pour attendre deux secondes avant de lancer les tests, permettant au testeur de mettre l'application au premier plan :

```
// Dans ./__TestsFIT__/before_tests.js

'use strict'
module.exports = function(){
  Console.bluebold("Merci d'activer l'application")
  return new Promise((ok,ko)=> {setTimeout(ok,2000)})
  // ou :
  // return wait(2000)
}
```

Exécution avant et après le test courant

Pour jouer du code avant et après la feuille de test courant, le test courant, utiliser les méthodes `after` (ou `after_test`) et `before` (ou `before_test`) :

```
  var t = new Test("mon test")

  t.before(<promesse>)
  t.after(<promesse>)
```

```
  t.case(...)
  t.case(...)
```

Le cas classique, dans Film-Analyzer, consiste à charger une analyse de film avant de procéder aux tests. On procède ainsi :

```
var t = new Test("Mon test sur une analyse")

t.before(FITAnalyse.load.bind(FITAnalyse, 'dossier/test'))

t.case("Premier test", ()=>{
  //... Je peux exécuter ici un tests sur l'analyse chargée, après son
  //... chargement.
})
```

Code à exécuter avant ou après chaque cas

Pour exécuter du code asynchrone ou non avant et après chaque cas, on utilise les méthodes de test `before_case` et `after_case` :

```
const test = new Test("Mon test")

test.before_case( () => { /* code à exécuter avant chaque cas */ })
test.after_case( () => { /* code à exécuter avant chaque cas */ })
```

Simuler des touches clavier

Pour toutes les méthodes de tests concernant les touches clavier, cf. le fichier rassemblant toutes les méthodes.

Exécution d'une action (`action`)

Pour mettre en valeur une action à exécuter (et s'assurer qu'elle fonctionne), on peut utiliser le mot-clé-fonction `action` :

```
action("L'action que je dois faire", ()=>{
  // Le code de l'action, par exemple :
})
```

Par exemple :

```
action("L'utilisateur clique sur le bouton OK", () => {
  $('#monBoutonOK').click()
})
```

De cette manière, dans l'énoncé des tests, on peut suivre toutes les actions accomplies.

L'action peut être aussi asynchrone et doit alors être précédée du mot clé `await` :

```
await action("Je fais une action asynchrone", async () => {
  keyPress('n')
  wait(1000)
  keyPress('i')
```

```
  })
```

Textes écrits dans le suivi

En dehors des messages des assertions elles-mêmes, on peut trouver ces textes dans le suivi du tests :

- Cas entièrement à implémenter (`pending`)
- Test à implémenter plus tard (`tester`)

Sommaire des opérations

On peut écrire un sommaire des opérations qui sont testés, n'importe où (dans le describe, dans le case), à l'aide la méthode `sumarize` :

```
describe("Mon tests", async function(){
  this.case("Le cas courant", async () => {
    sumarize(`
      - on fait ceci pour voir cela
      - et puis ensuite on teste ça
      - et on devrait obtenir tel résultat.
      `)
  })
})
```

Cas entièrement à implémenter (`pending`)

On utilise le mot-clé-fonction `pending` pour déterminer un cas entièrement à implémenter.

```
t.case("Mon tests pas implémenté", ()=>{
  pending("Faire ce test plus tard")
})
```

> On peut ne rien mettre en paramètre, ce qui indiquera "TODO" dans les tests.

Noter que contrairement à `tester`, qui n'influence pas le résumé total des tests à la fin (sa couleur), ce `pending` empêche les tests de passer au vert.

Test à implémenter plus tard (`tester`)

Lorsqu'un test ponctuel — à l'intérieur d'un cas long — est compliqué ou délicat et qu'on ne veut pas l'implémenter tout de suite, on peut le remplacer par le mot-clé-fonction `tester`.

```
tester("<message du test à faire>")
```

C'est le message `<message du test à faire>` qui apparaitra en rouge gras dans le suivi des tests, indiquant clairement que ce test sera à implémenter.

On peut se servir de ce mot-clé, par exemple, pour définir rapidement tous les tests à faire. Puis les implémenter dans un second temps.

Tests des FITests

Les tests des FITests se trouvent dans `./__TestsFIT__/FIT`. Pour les lancer, il suffit donc de régler `onlyFolders: ['FIT']` dans `config.json`.

Rationnalisation des termes

Les termes sont rationnalisés pour tous les tests :

`actualValue`
: Valeur actuelle réelle à comparer à `expectedValue`.
Elle est souvent passée en premier argument de `expect` mais c'est loin d'être toujours le cas.

`expectedValue`
: Valeur attendue réelle. À comparer à `actualValue`.

`sujet`
: Le sujet humain à écrire dans les messages.

`objet`
: L'objet humain à écrire dans les messages. C'est parfois la valeur attendue, `expectedValue`, stringifiée ou JSONisée.

Expectations d'expectations

Dans ces tests, pour dire qu'un test doit produire un succès ou, particulièrement, un échec (donc pour produire un succès qui dit qu'un test aurait du être un échec), on utilise le [sujet complexe](#) `x`. Par exemple :

```
expect(x('expect(12).is(13)')).fails()
```

Cette expectation évalue en silence `expect(12).is(13)`, qui retourne false, conformément à ce que demande `fails()`.

Mais les choses sont beaucoup plus complexe pour les tests des tests asynchrone. Il faut alors transmettre une méthode à `x`. Sa forme générale est :

```
expect(x(
  async function(){return await expect(...).assertion(..., {onlyReturn:true})}
)).fails()
```

Bien noter les `async`, `return` et `await` qui sont tous indispensables. Surtout le `return` qu'on risque de facilement oublier et qui porduirait un échec chaque fois.

Pour que le `sujet` affiché ne soit pas la fonction, on peut utiliser :

```
expect(
  x(
    async function(){return await expect(...).assertion(..., {onlyReturn:true})}
  ),
  'expect(...).assertion(...)'
).fails()
```

> Noter qu'on ne remet pas le `{onlyReturn:true}` qui ne sert qu'à faire un test silencieux, donc qui ne produira pas de message de succès ou d'échec.

Toutes les méthodes et expectations

Les listes des méthodes et des expectations est tenue à jour dans un fichier HTML du dossier `FITests/Manuel`.

Pour pouvoir actualiser la liste des méthodes et expectations de tests, on peut ajouter dans le fichier `pakcage.json` de l'application :

```
"scripts":{
  "collect-fit-methods": "ruby FITests/exe/build_methods_list.rb",
  "open-fit-methods": "open FITests/Manuel/Manuel_FITests_METHS.html",
}
```

Puis jouer `npm run collect-fit-methods` pour actualiser la liste et `npm run open-fit-methods` pour ouvrir le fichier HTML contenant toutes les méthodes.

Description des méthodes et expectations

Pour qu'une méthode ou une expectation soit répertoriée, il suffit de bien la définir au-dessus à l'aide des marques spéciales :

```
/*
  @method (returned) nomDeLaMethode(provided)
  @description Ici une description de la méthode
  @note Des notes éventuelles.
  @provided
    :arg1 {Type} Le premier argument
    :arg2 {Type} Le deuxième argument
    ...
  @returned
    :prop1 {Type} La première propriété retournée
    :prop2 {Type} La seconde propriété
    ...
  @usage let res = await nomDeLaMethode(data)

*/
```

Noter qu'il faut obligatoirement `/*` et `*/` pour encadrer la définition. S'il y a deux astérisques, ça ne fonctionnera pas.