Les `Case-méthodes`

- Méthodes du module `ModuleRouteMethods`
- L'objet `html`
- Méthodes-case `methode` , `not_methode` , `methode?` et `methodeS`
    - Arguments des cases-méthodes plurielles
- Méthodes-case de l'objet-case ROUTE
- Méthodes-case de l'objet-case FORM
- Définition

Définition

Les “case-méthodes” ou “méthodes de cas” correspondent aux `it` de RSpec, mais ne sont pas tout à fait identiques.

L'objet `html`

L'objet `html` correspond à la `page` de RSpec en mode d'intégration. Il fait référence à la balise de même nom de la page HTML, donc la page intégrale, avec son `head` et son `body` .

C'est un objet propre au test de classe `SiteHtml::TestSuite::HTML` qui utile `Nokogiri::HTML` pour gérer son code et procéder aux tests.

Il doit s'employer à l'intérieur d'une méthode de test comme `test_route` ou `test_form` pour ne donner que ces deux exemples.

Quelques méthodes de l'objet `html` :

~~~ruby

html.has_title("Mon beau titre", 3, {strict: true}) # => Produit un succès si la page contient exactement #

### Mon beau titre

html.has_tag?("div#estula") # Retourne true si la page contient la balise div#estula

html.has_messages(["Bonjour", "Revoir", "Encore"], {strict: true}) # => Produit un succès si la page contient exactement les # message flash spécifiés.
~~~

```
ICI ICI ICI

<a name='methodesmoduleroutemethodes'></a>

## Méthodes de cas du module `ModuleRouteMethods`

* [respond & dérivées](#methoderesponds)
* [has_tag & dérivées](#methodtesthastag)
* [has_title & dérivées](#methodehastitle)
* [`has_message` & dérivées](#casemethodhasmessage)
* [`has_error` et dérivée](#methodecasehaserror)


<a name='methoderesponds'></a>

#### `responds`

Méthode directe (c'est-à-dire qu'elle s'emploie seule, comme une méthode de la méthode de test). Produi
t un succès si la commande renvoie une page valide (code 200), une failure dans le cas contraire.

        responds
```

```
        ________________________________________________________
```

FORMES
—— ||
Droite | Négative | Interrogation |
——
Simple || responds | not_responds | responds? | ----------||---------------|--------------------|-------------------|

Pluriel || --- | --- | --- |

```
<a name='methodtesthastag'></a>

#### `has_tag`

Méthode de l'objet `html`.

Produit un succès si la page contient le tag spécifié.

@syntaxe

        has_tag <tag>[, <options>]

        ou

        has_tags( [<tag>, <tag>, <tag>] ) # test une liste de balises
```

```
        ___________________________________________________________
```

FORMES
——
|| Droite | Négative | Interrogation | ----------||---------------|---------------------|-------------------| Simple || has_tag | has_not_tag | has_tag? | ----------||---------------|---------------------|-------------------|

Pluriel || has_tags | has_not_tags | has_not_tag? |

```
@exemples

    test_route "ma/route" do
      html.has_tag "div#mondiv.soncss"
      html.has_tag "div", {id:"mondiv", class:'soncss'}
      html.has_not_tag "span", {text: /Bienvenue !/}
      html.has_tags(['div#premier', 'div#deuxieme', 'div#troisieme'])
    end


<a name='methodehastitle'></a>

#### `has_title`

Méthode de l'objet `html`.

Produit un succès si la page contient le titre spécifié, au niveau spécifié (if any)

@syntaxe

        html.has_title <titre>[, <niveau>][, <options>]
```

```
      ________________________________________________________
```

FORMES
—— ||
Droite | Négative | Interrogation |
——
Simple || has_title | has_not_title | has_title? | ----------||---------------|-------------------|------------------|

Pluriel || has_titles | has_not_titles | --- |

```
<a name='casemethodhasmessage'></a>

## `has_message`

Méthode de l'objet `html`.

    html.has_message <message>[, <options>]
```

```
      ________________________________________________________
```

FORMES
—— ||
Droite | Négative | Interrogation | ----------||---------------|------------------ |------------------| Simple || has_message | has_not_message | has_message? | ----------||---------------|-------------------|------------------|

Pluriel || has_messages | has_not_messages | --- |

```
<a name='methodecasehaserror'></a>

### `has_error` et dérivées

Méthode de l'objet `html`.

    test_form "user/login", dataform do

      ...

      html.has_error <erreur>[, <options>]

    end

Produit un succès si la page contient le message d'erreur spécifié en argument, produit une failure dans le cas contraire.
```

```
      ________________________________________________________
```

FORMES
——— || Droite | Négative | Interrogation | ----------||---------------|------------------ |------------------| Simple || has_error | has_not_error | has_error? | ----------||---------------|-------------------|------------------|

Pluriel || has_errors | has_not_errors | has_errors? |

```
---------------------------------------------------------------------


<a name='hashashnotandinterrogation'></a>

## Méthodes-case `methode`, `not_methode`, `methode?` et `methodeS`

La plupart des méthodes de test possèdent SIX états différents qui correspondent à six actions et retou
rs différents.

Chaque méthode peut être DROITE ou INVERSE (NÉGATIVE). C'est la différence entre `has` et `has_not` ou
`is` et `is_not`.

Chaque méthode peut être INTERROGATIVE. C'est la différence entre `has`, qui produit un cas de test (do
nc une failure ou un succès) et la méthode `has?` qui ne fait que retourner le résultat de l'évaluation
, true ou false.

Et enfin, chacune de ces méthodes (ou presque) peut être SIMPLE (SINGULIÈRE) ou PLURIELLE. C'est la dif
férence entre `has_truc` et `has_trucs` qui permet de tester plusieurs éléments en même temps.

<a name='argumentdescasesmethodesplurielles'></a>

### Arguments des cases-méthodes plurielles

Le premier argument des méthodes de cas plurielles est une liste d'élément à tester. Chaque ÉLÉMENT de
cette liste peut être :

* soit un Array contenant [tag, options, autres arguments éventuels],
* soit un argument seul.

Par exemple :

    has_tags ["premier#tag", "div#interieur"]

    has_tags [
      ["span", {class: "sacss", count:5}],
      "div#seul"
    ]

Le deuxième argument transmis à une méthode plurielle est un `Hash` d'options qui sera appliqué à tout
élément de la liste.

Par exemple, l'appel à :

~~~ruby

    has_tags ["div#bonjour", "div#aurevoir"], {class:'voir'}
```

… entrainera les appels à :

~~~ruby

```
has_tag "div#bonjour", {class:'voir'}
has_tag "div#aurevoir", {class: 'voir'}
```

```
Noter que ces options seront ajoutées à la requête sans vérification. Donc si un élément de la liste le
s définit déjà, ces options seront ajoutées à la fin et une erreur de nombre d'arguments se produira.

---------------------------------------------------------------------

<a name='methodesdetestderoute'></a>


### Méthodes de tests de route (méthodes-cases)
```
~~~

Ce que j'appelle les "méthodes de tests de route", ce sont les méthodes qui testent une page particuliè
re appelée par une route, donc une URL moins la partie local.

Par exemple, si l'url complète est `http://www.atelier-icare.net/overview/temoignages` alors la route e
st `overview/temoignagnes`. Cette route peut également définir des variables après un "?".

La formule de base du test d'une route est la suivante :

```
        test_route "la/route?var=val"[, options] do |r|
          <methode>
          <methode> <parametres>
        end
```

`options` peut être le libellé à donner au test (`String`) ou un `Hash` contenant : les [options de fin
 de méthode](#optionsdefinmethodestest)

Liste des méthodes :

Cet objet-test hérite de toutes les [méthodes du module `ModuleRouteMethods`](#methodesmoduleroutemetho
des)

Méthodes-cases de l'objet-case formulaire

L'autre grande chose à faire avec les pages, c'est le remplissage de formulaires. La méthode-test ci-de
ssous est la méthode principale qui s'en charge :

```
        test_form "la/route", <data> do

          <case-methode>

        end
```

Les `data` doivent permettre de trouver le formulaire et de remplir les champs du formulaire avec les v
aleurs proposées, sous la forme :

```
        {
          id:     "ID du formulaire (if any)",
          name:   "NAME du formulaire (if any)",
          action: "ACTION du formulaire, if any",
          fields: { # Champs du formulaire
            'id ou name de champ' => {value: <valeur à donner> },
            'id ou name de champ' => {value: <valeur à donner> }
          }
        }
```

Toutes les méthodes-cases :

Toutes les [méthodes du module `ModuleRouteMethods`](#methodesmoduleroutemethodes) sont héritées donc o
n peut les utiliser avec les formulaires.

Méthodes propres :

* [`exist` - test de l'existence du formulaire](#testexistenceformulaire)
* [`fill_and_submit` - remplissage du formulaire](#testremplissageformulaire)

`exist` - Test de l'existence du formulaire

~~~ruby

    test_form "mon/formulaire", data do
      exist
    end
~~~

Seule la forme interrogative existe : `exist?`

`fill_and_submit` - remplissage et soumission du formulaire

~~~ruby

```
test_form "mon/formulaire", data_form do

  fill_and_submit

end
```

```
Données de formulaire :

~~~ruby

  data_form = {

    id: "ID du formulaire",
    action: "ACTION/DU/FORMULAIRE",
    fields: {
      <field_id>: {name: <field[name]>, value:"<field_value_expected_or_send>"},
      <field id>: {name: <field[name2]>, value:"<field value expected or send>"},
      etc.
    }
  }
```

> Note : La propriété `:name` est impérative pour les champs s'ils doivent être pris en compte pour la simulation de remplissage.

Les données utilisées seront celles transmises dans `data_form` (enregistrées à l'instanciation du formulaire) mais on peut également en transmettre d'autres à la volée qui seront mergées DE FAÇON INTELLIGENTE (#) avec les données originales. Par exemple :

~~~ruby

```
test_form "signup", data_signup do |f|

  fill_and_submit(pseudo: nil) # le :pseudo sera mis à nil
  has_error "Vous devez soumettre votre pseudo !"n {strict: true}

  ...

  fill_and_submit(password_confirmation: nil)
  has_error "La confirmation du mot de passe est requise."
```

~~~

(#) *DE FAÇON INTELLIGENTE* signifie qu'on peut définir simplement la valeur d'un champ sans mettre `dform[:fields][:id_du_field][:value] = "la valeur"`. Il suffit de faire :

```
id_du_field: "la valeur"
```

… et la méthode `fill_and_submit` comprendra qu'il s'agit du champ.
```