Chapitre 4 - Réusinage du code et templates

Pourquoi tout ces tests ? Vous pouvez penser que la méthode est trop lente , mais au final la complexité croissantes des interactions peut aboutir à l'impossibilité de réusiner son code (modification du code sans changement de fonctionnalités). Plus un système devient complexe et moins vous pourrez avoir l'intégralité des intéractions en mémoire, les erreurs seront inévitable et sans tests trouver leur origine sera impossible.

Le TDD limite la taille du code, guide dans la conception et réduit la "dette technique" c'est à dire le temps qu'il faudra utiliser pour effectuer les maintenances corrective et évolutive dans le futur.

Allez on retourne au boulot. Humm d'ailleurs je suis un peu perdu mais on en était où déjà ? Ah oui utilisons notre TF pour savoir ! (Encore eux lol)

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "functional_tests.py", line 20, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
AssertionError: Finish the test!

----------------------------------------------------------------------
Ran 1 test in 4.383s

FAILED (failures=1)
```

(Si vous avez une erreur vérifiez que vous avez bien lancé le server dans un autre shell avec `$ ./manage.py runserver`)

Ok notre TF échoue volontairement du fait de la méthode fail avec le message 'Finish the test'.
Ah oui ca me revient, notre test vérifiant 'To-Do' dans le titre de la page est passé, mais il reste la suite de la user storie à tester.

> A faire:
>
> - [ ] Continuer la rédaction du TF pour le reste de la US.

Bon bah c'est parti, on complète le TF !

> functional_tests.py

```
import time
import unittest
```

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()

    def tearDown(self):
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):
        # 1 - Scénario "Robert découvre le site":
        # Etant donné Robert,
        #  un visiteur qui a entendu parler de notre site
        # Quand il saisit l'url de notre site via son navigateur
        self.browser.get('http://localhost:8000')
        # Alors il peut lire "To-Do" dans l'onglet,
        self.assertIn('To-Do', self.browser.title)
        # Et dans un header sur la page.
        header_text = self.browser.find_element_by_tag_name('h1').text
        self.assertIn('To-Do', header_text)
        # Et on lui propose de saisir une note de texte.
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertEqual(
            inputbox.get_attribute('placeholder'),
            'Enter a to-do item'
        )

        # 2 - Scénario "Robert ajoute une note":
        # Etant donné Robert (toujours lui !)
        #  un visiteur sur notre page d'accueil.
        # Quand il ajoute du texte ("Acheter du pain"),
        #  dans la zone de texte
        inputbox.send_keys('Acheter du pain')
        # Et qu'il appuie sur entrée
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)
        # Alors sa note est ajoutée dans un tableau,
        #  sur la même page avec le numéro de la note devant.
        table = self.browser.find_element_by_id('id_list_table')
        rows = table.find_elements_by_tag_name('tr')
        self.assertTrue(
            any(r.text == '1: Acheter du pain' for r in rows)
        )

        # Echec volontaire du test
        self.fail('Finish the test!')
        # [suite des scénarios pour plus tard ...]
```

Explications:

On va pouvoir manipuler les éléments du DOM grâce à des méthodes de notre instance 'browser' qui "représente" le navigateur. Ces méthodes sont par exemple find_element_by_id() pour localiser des éléments du DOM.

On va aussi simuler la manipulation du clavier en utilisant la méthode send_keys() et la classe Keys, (ne pas oublier de l'importer !), pour les touches spéciales.

Le time.sleep est là pour être sur, que le navigateur ait le temps de charger la page, avant de faire l'assertion de test qui suit.

Enfin, attention, la methode find_elements_by_tag_name() retourne une liste d'éléments.

On lance le TF:

```
$ python functional_tests.py
[ ... ]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: h1
```

Alors le TF n'arrive pas a trouver un header. C'était prévu. Onvient de faire un gros changement de notre Test Fonctionnel et en général il vaut mieux faire un commit à la suite.

```
$ git commit -am "TF improvement
> FT now checks that visitor can add item to page."
```

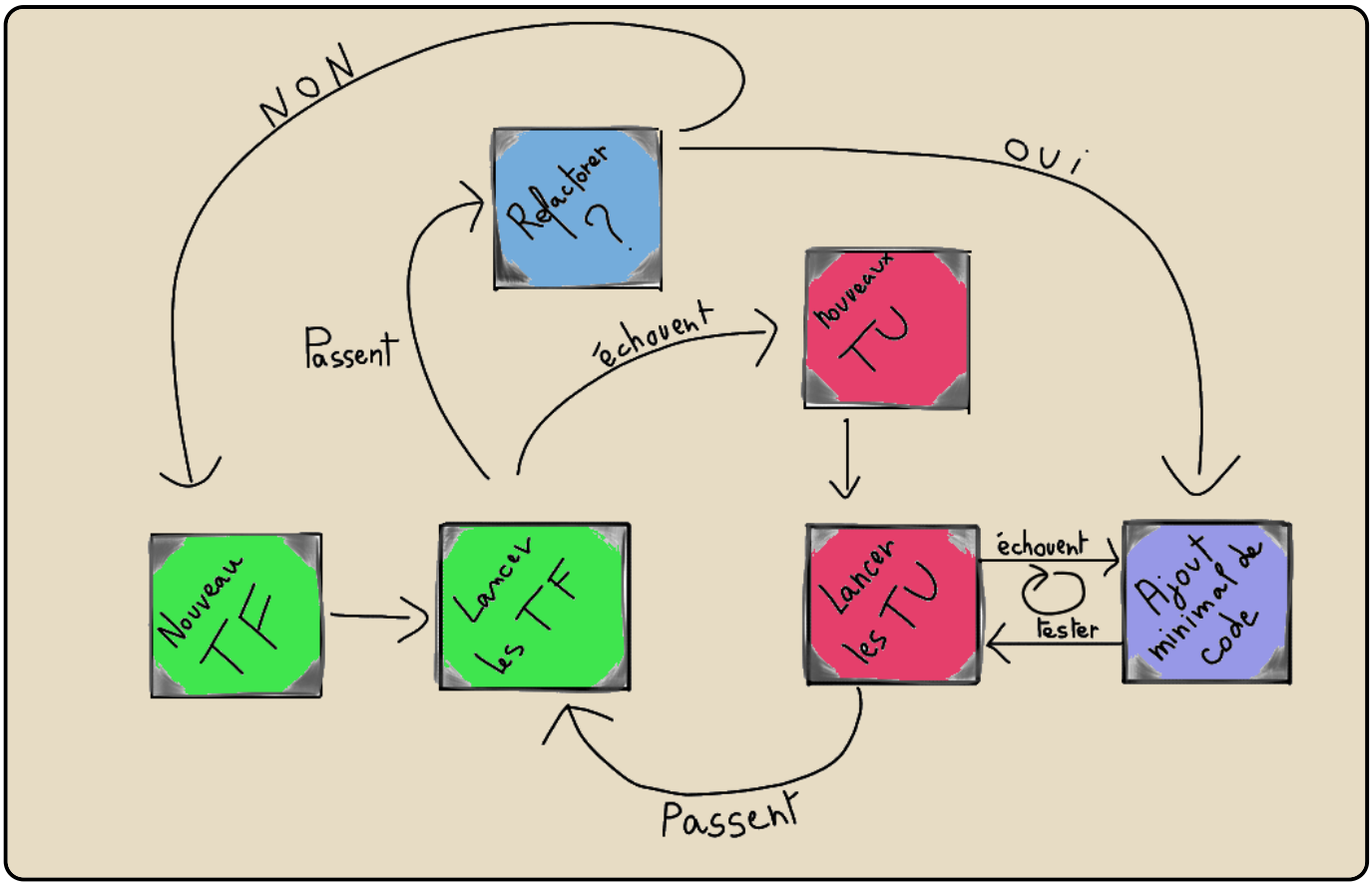
Et on peut cocher notre liste ca fait toujours du bien 😃

> A faire:
>
> - [x] Continuer la rédaction du TF pour le reste de la US.

Réusinage de code: Utilisation des templates.

On va s'interesser au "réusinage de code" ou refactoring en anglais. Cette étape consiste à **améliorer** notre code sans que la **fonctionnalité** de celui ci soit changé.

Aller un petit croquis pour rappeler le processus BDD et TDD:



Une règle importante est de ne pas tester des constantes, mais la logique du code, le controle du flux des données et les configurations. cela paraît une évidence mais nous l'avons fait dans nos tests unitaires lorsque nous avons testé notre vue et le html en dur quelle renvoyait.

Pour générer les pages html nous allons utiliser les templates (gabarits en français)

Ce sont des objets qui permettent de générer dynamiquement du contenu grâce à un langage de template et à des objets context

Un exemple:

```
>>> from django.template import Context, Template
>>> template = Template("My name is {{ my_name }}.")

>>> context = Context({"my_name": "Adrian"})
>>> template.render(context)
"My name is Adrian."

>>> context = Context({"my_name": "Dolores"})
>>> template.render(context)
"My name is Dolores."
```

On utilise ici le même gabarits pour générer une phrase, seule la variable "my_name" de l'objet context change.

Vous pouvez aussi faire hériter les templates entre eux pour les superposer et éviter de réécrire plusieurs fois le meme contenu HTML.

Pour illustrer l'héritage de templates, je vous propose un croquis d'un théatre à l'italienne (vous allez comprendre !):

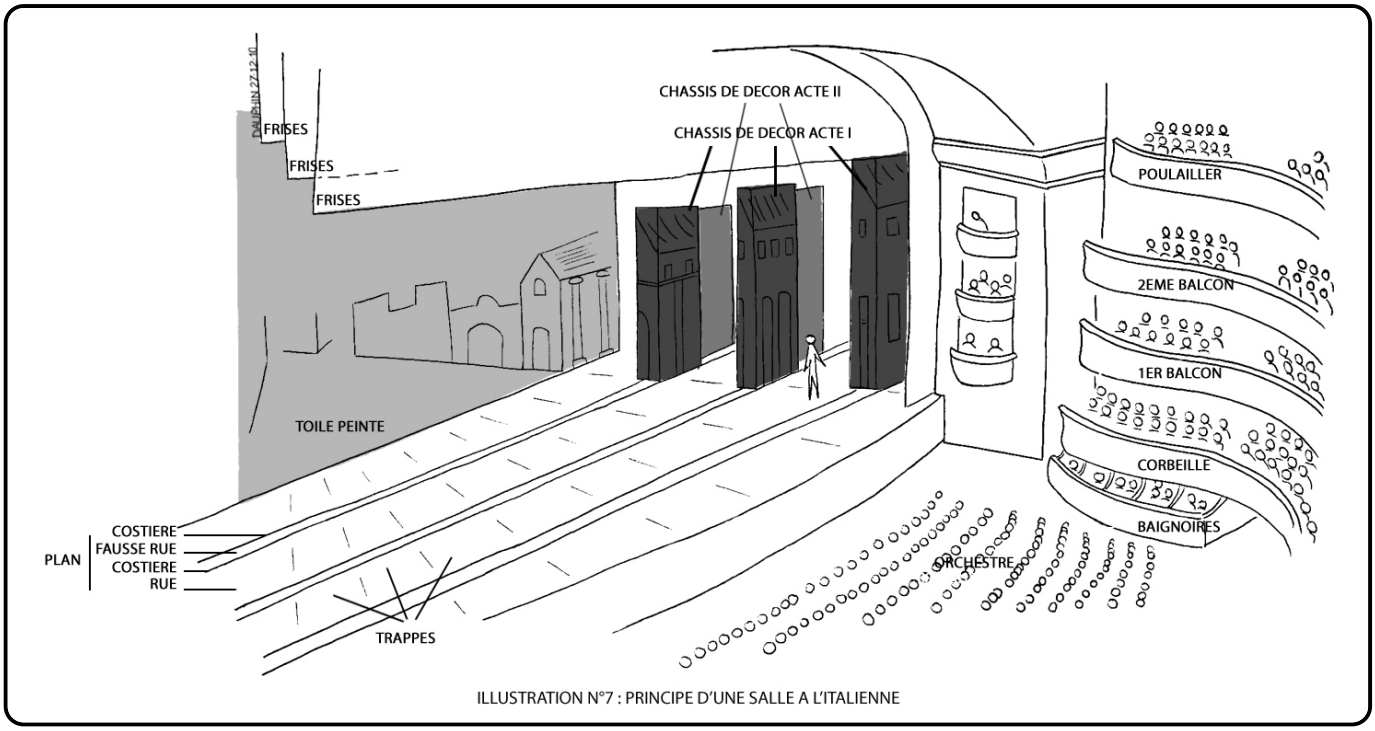
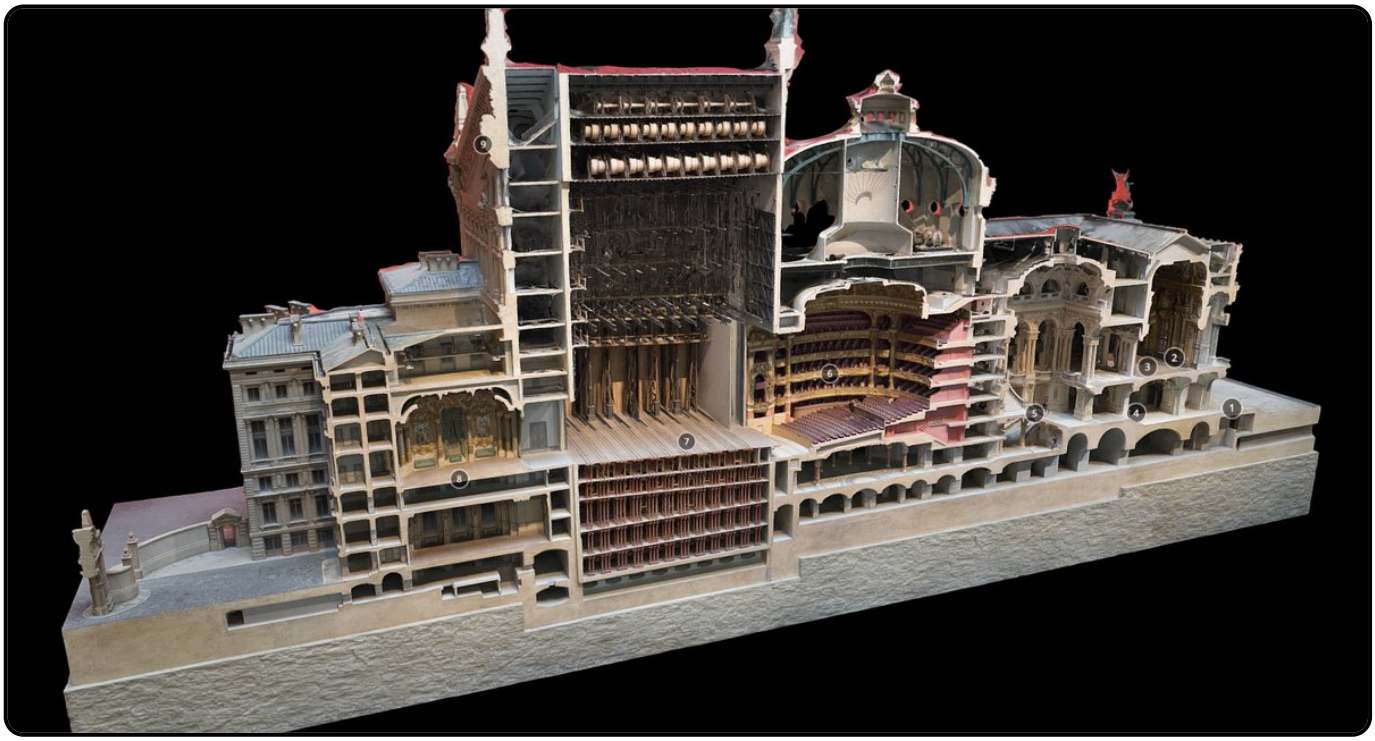


ILLUSTRATION N°7 : PRINCIPE D'UNE SALLE A L'ITALIENNE

Les décors sont coulissants et peuvent se superposer à une toile de fond ou à d'autres décors ! La production de visuels est modulaire! Les templates c'est le meme systeme, mais on peut en plus changer dynamiquement chaque décors !

Allez je vous mets la coupe de l'Opéra Garnier, juste parce que c'est beau, à vous de trouver où sont les "templates":



Bon c'est bien beau tout ca mais revenons a notre code.

> A faire:

> - [ ] Refactorer notre vue en utilisant les templates

Nous allons créer un dossier templates dans notre application et y ajouter un fichier home.html ou nous allons mettre notre HTML:

> lists/templates/home.html

```
<html>
    <title>To-Do</title>
</html>
```

Et nous allons refondre notre vue pour qu'elle utilise ce template.

```
from django.shortcuts import render

def home_page(request):
    return render(request, 'home.html')
```

On utilise la fonction render qui va retourner un objet HttpResponse et rechercher automatiquement les templates dans les répertoires /templates de nos applications.

On lance les tests unitaires pour voir si ca marche :

```
$ ./manage.py test lists/
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.E
======================================================================
ERROR: test_to_do_in_title_home_page (lists.tests.HomePageTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/tuto/todo-tdd/lists/tests.py", line 14, in
test_to_do_in_title_home_page
    response = home_page(request)
  File "/home/tuto/todo-tdd/lists/views.py", line 4, in home_page
    return render(request, 'home.html')
  File "/home/tuto/.local/share/virtualenvs/todo-tdd-
rzjrtEdZ/lib/python3.6/site-packages/django/shortcuts.py", line 36, in
render
    content = loader.render_to_string(template_name, context, request,
using=using)
  File "/home/tuto/.local/share/virtualenvs/todo-tdd-
rzjrtEdZ/lib/python3.6/site-packages/django/template/loader.py", line 61,
in render_to_string
    template = get_template(template_name, using=using)
  File "/home/tuto/.local/share/virtualenvs/todo-tdd-
rzjrtEdZ/lib/python3.6/site-packages/django/template/loader.py", line 19,
in get_template
```

```
    raise TemplateDoesNotExist(template_name, chain=chain)
django.template.exceptions.TemplateDoesNotExist: home.html
```

Arggh ca ne fonctionne pas ! Pourtant on a bien mis notre template home.html dans un dossier lists/templates et notre vue utilise la fonction render() qui va chercher automatiquement les templates s'ils sont dans un dossier /templates... Bon bah il va falloir lire l'erreur en détail ...

> - On fait appel à notre vue home_page
> - Elle retourne la fonction render avec notre template home.html
> - Après nous avons `content = loader.render_to_string([...])`, ici django utilise un loader pour trouver le template, c'est à partir de la que ca bloque...

En fait le loader va pouvoir trouver les dossiers /templates de nos applications uniquement si celles-ci sont installées. En effet lors de la création de notre app 'lists' à l'aide de `./startapp lists` nous ne l'avons pas ajoutée à notre projet. Pour cela il faut simplement rajouter, le nom de notre app au fichier /superlists/settings.py dans la variable INSTALLED_APPS :

> superlists/settings.py

```
# Application definition
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists', # new
]
```

Normalement maintenant le loader de template utilisé par la méthode render va pouvoir trouver notre dossier templates de notre app 'lists'.

On relance les tests pour vérifier !

```
$ ./manage.py test lists/
[...]
OK
```

Top ! On a réussie a réfactorer en utilisant les templates ! (Si vous avez une erreur peut etre que vous avez rajouté des lignes vides à la fin de votre fichier home.html ?)

> A faire:
>
> - [x] Refactorer notre vue en utilisant les templates

Nous avons refactoré notre code, mais nos tests unitaires continuent de vérifier des constantes, en l'occurence de vérifier notre HTML en dur.

> A faire:
>
> - [x] Refactorer notre vue en utilisant les templates
> - [ ] Refactorer nos tests unitaires.

Allez on refactor nos tests unitaires:

> lists/tests.py

```
# from django.http import HttpRequest
# from django.urls import resolve
from django.test import TestCase
# from lists.views import home_page

class HomePageTest(TestCase):

    # def test_root_url_resolve_to_home_page_view(self):
    #     my_view = resolve('/')
    #     self.assertEqual(my_view.func, home_page)

    # def test_to_do_in_title_home_page(self):
    def test_uses_home_template(self):
        # request = HttpRequest()
        # response = home_page(request)
        response = self.client.get('/')

        # html = response.content.decode('utf8')
        # self.assertTrue(html.startswith('<html>'))
        # self.assertIn('<title>To-Do</title>', html)
        # self.assertTrue(html.endswith('</html>'))
        self.assertTemplateUsed(response, 'home.html')
```

j'ai mis en commentaires les portions de test inutiles après réusinage du code.
En effet, on utilise maintenant le client de test "client" qui se comporte comme un navigateur simple. On comprend que si la reponse du client, pour une requête GET de l'url racine, possède bien le template 'home.html'.
Alors toutes les autres lignes sont vérifiées et elles deviennent superficielles, ont peut les supprimer ! (elles étaient la pour avancer doucement et comprendre le cheminement des objets HttpRequest et HttpResponse)

Notre classe de tests unitaires ressemble maintenant à cela:

> lists/tests.py

```
from django.test import TestCase

class HomePageTest(TestCase):
```

```
    def test_uses_home_template(self):
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')
```

Allez on vérifie que le test unitaire passe:

```
$ ./manage.py test lists/
[...]
OK
```

Et que notre test fonctionnel échoue toujours au meme endroit:

```
$ python functional_tests.py
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: h1
```

> A faire:
>
> - [x] Refactorer notre vue en utilisant les templates
> - [x] Refactorer nos tests unitaires.

Ah ca fait du bien de cocher une case. On a refactoré notre code et nos test unitaires, en général c'est un bon moment pour un commit:

```
$ git add .
```

```
$ git commit -m "Refactor home_view to use template
> add lists app to the project's setting INSTALLED_APPS
> Refactor the unit tests"
```