TDD avec Django

Version 0.9.1

Gérard Rozsavolgyi

octobre 03, 2017

Table des matières

CHAPITRE 1

Ce cours au format epub

Django TDD en accéléré format epub

CHAPITRE 2

Ce cours au format pdf

Django TDD en accéléré en pdf

CHAPITRE 3

Tables des matières :

3.1 TDD avec le Framework Django

3.1.1 Le Framework Django

— Développé pour un journal local de la ville de Lawrence dans le Kansas
— But : Réaliser une sorte de CMS, simple à utiliser pour les non informaticiens
— En Open Source depuis 2005
— Beaucoup d'évolutions depuis
— Framework Web de référence
— Cache
— Migrations
— Authentification
— Internationalisation, Unicode
— Gestion complète des exceptions
— Bonne documentation

3.1.2 Framework Web généraliste offrant

— MVT = Modèle Vue Template
— Système de templates
— ORM = Object Relational Mapper (comme SQLAlchemy ou Doctrine)
— Serveur Web intégré
— Interface d'Admin complète, souple et extensible

3.1.3 Versions

— OpenSource (BSD) en 2005
— Version 1.0 en 2008
— Version 1.4 LTS en 2012
— Version 1.8 LTS en
— Actuelle (2017) : 1.11
— Version 1.11 sera la dernière avec le support Python 2

3.2 Installation Django

Indice : Il faut savoir utiliser un virtualenv en Python . . .

3.2.1 Virtualenv

— Environnement virtuel Python
— Permet d'installer ses propres paquets
— Peut ou non utiliser les libs présentes dans le système
— Permet de fixer l'environnement logiciel nécéssaire à un projet
— Habituellement, 1 virtualenv par projet
— Pas besoin de le sauvegarder, juste les *requirements* qui contiennent la liste des dépendances à installer

3.2.2 Création

```
virtualenv -p python3 venv3
```

ou plus rarement avec les librairies déjà installées sur votre système :

```
virtualenv --system-site-packages -p python3 venv3
```

3.2.3 Activation virtualenv

```
source venv3/bin/activate
```

3.2.4 Installation de Django

```
pip install  django
```

3.2.5 Lancement du serveur

```
./manage.py runserver
```

3.2.6 Test :

Localhost:8000 (http ://localhost :8000)

3.2.7 Initialiser projet git et premiers commits

Vérifiez que vous avez déjà fait les réglages de base de git (user, email, editeur) :

```
git config --list
```

Vous devriez avoir user.name, user.email et core.editor de configurés. Sinon :

```
git config --global user.name "Alice Torvalds"
git config --global user.email "alice@linux.org"
git config --global core.editor emacs
```

3.2.8 .gitignore

On se place à la racine du projet , on y ajoute un fichier .gitignore contenant :

```
__pycache__
local_settings.py
```

Le fichier *.gitignore* contient les fichiers ou dossiers qui doiventéchapper au versionnage. Ici nous indiquons que tous les dossiers *__pycache__* doivent être ignorés ainsi que le fichier *local_settings.py* qui peut contenir des informations sensibles (comme des logins et passwd de BD qui n'ont pas à être versionnés).

3.2.9 Puis on initie un dépôt git

```
git init
git add .
```

Attention : On ne versionne pas le virtualenv Python. Il est propre à chaque environnement et on ne le change pas d'emplacement une fois qu'il est créé !

On peut extraire les dépendances de notre projet (pour l'instant essentiellement une version spécifique de django) avec la sous-commande *freeze* de pip.

3.2.10 requirements

```
pip freeze > requirements.txt
```

On stocke ces dépendances dans le fichier *requirements.txt* et on peut versionner le fichier *requirements.txt* obtenu :

```
git add requirements.txt
git commit -m "ajout requirements.txt au projet"
```

3.3 Création du projet de base

3.3.1 La commande django-admin

Permet de créer un projet, une application ou d'inspecter un projet existant (base de données ,etc)

```
django-admin startproject ProjetEssai
```

Puis consultons l'arborescence du projet créé :

```
tree ProjetEssai
```

qui donne :

```
ProjetEssai/
|-- ProjetEssai
|   |-- __init__.py
|   |-- __pycache__
|   |   |-- __init__.cpython-35.pyc
|   |   |-- settings.cpython-35.pyc
|   |   |-- urls.cpython-35.pyc
|   |   `-- wsgi.cpython-35.pyc
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
|-- db.sqlite3
|-- manage.py
```

3.3.2 La commande django-admin

Sans arguments, donne les sous-commandes disponibles.

```
Type 'django-admin help <subcommand>' for help on a specific␣
↪subcommand.

 Available subcommands:

 [django]
     check
     compilemessages
     createcachetable
     dbshell
     diffsettings
     dumpdata
     flush
     inspectdb
     loaddata
     makemessages
     makemigrations
     migrate
     runserver
     sendtestemail
     shell
     showmigrations
     sqlflush
     sqlmigrate
     sqlsequencereset
     squashmigrations
     startapp
     startproject
     test
     testserver

     Note that only Django core commands are listed as settings are␣
↪not properly configured (error: Requested setting INSTALLED_APPS,␣
↪but settings are not configured. You must either define the␣
↪environment variable DJANGO_SETTINGS_MODULE or call settings.
↪configure() before accessing settings.).
```

Danger : Attention, django-admin et manage.py ne sont pas strictement identiques même si ils sont très semblables. Django-admin se charge des tâches administratives et manage.py est créé dans chaque projet que vous réalisez et assure certaines tâches automatiquement.

3.3.3 Modif .gitignore

La base de données n'a pas normalement vocation à être versionnée. On va donc l'ajouter au .gitignore et commiter.

```
echo db.sqlite3 >> .gitignore
git commit -am "ajout fichier db au .gitignore"
```

3.3.4 Creation d'une app dans le projet

```
django-admin startapp planning
```

Puis observons l'arborescence obtenue :

```
tree ProjetEssai
```

qui donne :

```
ProjetEssai/
|-- ProjetEssai
|   |-- __init__.py
|   |-- __pycache__
|   |   |-- __init__.cpython-35.pyc
|   |   |-- settings.cpython-35.pyc
|   |   |-- urls.cpython-35.pyc
|   |   `-- wsgi.cpython-35.pyc
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
|-- db.sqlite3
|-- manage.py
|-- planning
|   |-- __init__.py
|   |-- admin.py
|   |-- apps.py
|   |-- migrations
|   |   `-- __init__.py
|   |-- models.py
|   |-- tests.py
|   `-- views.py
```

Commitons

3.4 Installation et tests simples de notre app

3.4.1 Complétons les settings de notre projet

Nous ajoutons l'app *planning* et aussi la *debug_toolbar* aux INSTALLED_APPS Puis les panels que l'on souhaite voir afficher, le middleware nécessaire, et les IPs autorisées. On peut également configurer la langue, le timezone, le format des dates, etc.

```
DEBUG = True

DEBUG_TOOLBAR_PANELS = [
'debug_toolbar.panels.versions.VersionsPanel',
'debug_toolbar.panels.timer.TimerPanel',
'debug_toolbar.panels.settings.SettingsPanel',
'debug_toolbar.panels.headers.HeadersPanel',
'debug_toolbar.panels.request.RequestPanel',
'debug_toolbar.panels.sql.SQLPanel',
'debug_toolbar.panels.staticfiles.StaticFilesPanel',
'debug_toolbar.panels.templates.TemplatesPanel',
'debug_toolbar.panels.cache.CachePanel',
'debug_toolbar.panels.signals.SignalsPanel',
'debug_toolbar.panels.logging.LoggingPanel',
'debug_toolbar.panels.redirects.RedirectsPanel',
]

ALLOWED_HOSTS = []


# Application definition

INSTALLED_APPS = [
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'debug_toolbar',
'planning',
]

INTERNAL_IPS = [
'localhost',
'127.0.0.1',
]

MIDDLEWARE = [
'django.middleware.security.SecurityMiddleware',
'django.contrib.sessions.middleware.SessionMiddleware',
'django.middleware.common.CommonMiddleware',
```

```
'django.middleware.csrf.CsrfViewMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware',
'django.contrib.messages.middleware.MessageMiddleware',
'django.middleware.clickjacking.XFrameOptionsMiddleware',
'debug_toolbar.middleware.DebugToolbarMiddleware',
]

../..

LANGUAGE_CODE = 'fr-FR'
TIME_ZONE = 'Europe/Paris'
```

Commitons.

3.4.2 Complétons le fichier tests.py de notre app planning avec un test bidon

```
from django.test import TestCase

class StringTest(TestCase):
    def test_concatene(self):
        """Test bidon"""
        self.assertEqual("Bon"+"Jour", "hello")
```

3.4.3 Puis lançons le test depuis Django :

```
./manage.py test
```

Il échoue logiquement. Corrigeons à présent cela . . .

3.4.4 Un test unitaire plus précis

Que fait le test suivant ?

```
from django.core.urlresolvers import resolve
from planning.views import home

class HomeTest(TestCase):

def test_root_planning_resolue_par_home_view(self):
    vue = resolve('/planning/')
    self.assertEqual(vue.func, home)
```

On lance le test, on constate qu'il échoue, et on complète le code nécessaire pour qu'il passe.

3.4.5 Passage de test home

On commence par déclarer dans le fichier général *urls.py* que les routes commençant par *planning* sont décrites dans le fichier *urls.py* de l'app planning et on intègre la debug_toolbar.

```
from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
        url(r'^admin/', admin.site.urls),
        url(r'^planning', include('planning.urls'))
]
# Pour la debug_toolbar:
from django.conf import settings

if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        url(r'^__debug__/', include(debug_toolbar.urls)),
                ] + urlpatterns
```

Puis on met en place le fichier *urls.py* de l'app *planning* :

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'/', views.home),
]
```

On reteste :

```
./manage.py test
```

Le test ne passe pas sauf si on implémente la méthode home() dans le views.py de l'app planning :

```
#views.py
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.

def home(request):
    return HttpResponse('<h1>Hello World!</h1>')
```

Maitenant, ça passe ! ! !

3.4.6 On peut à présent décrire notre modèle puis l'afficher dans l'admin

3.4.7 Modèle ToDo

```
#models.py
class ToDo(models.Model):
    name = models.CharField(max_length=250)
    begin_date = models.DateField(auto_now_add=True)
    end_date = models.DateField()
    done = models.BooleanField(default=False)

    def __str__(self):
        return self.name
```

3.4.8 Migrations

3.4.9 Pour que Django prenne en compte les modifications de structure de la BD

```
./manage.py makemigrations
./manage.py sqlmigrate planning 0001
./manage.py migrate
```

— la commande *makemigrations* prépare les migrations
— la commande *sqlmigrate* décrit en sql la migration qui va se faire en base de données
— La commande *migrate* l'effectue

3.4.10 Et pour voir le nouveau modèle dans l'admin de Django

Complétons le fichier *admin.py*

```
# admin.py
from django.contrib import admin
from planning.models import ToDo


class ToDoAdmin(admin.ModelAdmin):
    list_display=(
    'name', 'begin_date', 'end_date','done',
)

admin.site.register(ToDo,ToDoAdmin)
```

3.4.11 Ajout de debug-toolbar

On peut ajouter django-debug-toolbar à notre venv pour voir cet outil à l'oeuvre :

```
pip install django-debug-toolbar
```

Corrigez, les requirements :

```
pip freeze > requirements.txt
```

Commitez.

3.4.12 Créons un superuser dans Django pour accéder à l'admin

```
./manage.py createsuperuser
```

Puis visitez l'admin : Localhost:8000/admin/ (http ://localhost :8000/admin/) sous cette identité. Créez quelques ToDo. Observez la debug-toolbar et essayez ses différents Panels.

3.4.13 Améliorons un peu le rendu de l'admin des ToDo

```
@admin.register(ToDo)
class ToDoAdmin(admin.ModelAdmin):
        fieldsets = [
                ('ToDo Details', {'fields': ['name', 'done']}),
                ('ToDo Dates', {'fields': ['begin_date', 'end_date
↪']}),
        ]

        readonly_fields = ('begin_date',)

        list_display = ('name', 'begin_date', 'end_date','done',)
        list_editable = ('done',)
        list_display_links = ('name', 'date_reviewed',)
        list_filter = ('done',)
        search_fields = ['name',]
```

Testez. Remarquez le décorateur qui sert à enregistrer directement le modèle ToDo dans l'admin et aussi les améliorations dans l'affichage des ToDo dans l'admin. Commitez.

3.4.14 Enrichissement du Modèle ToDo

Ajoutons par exemple le fait de mettre automatiquement *end_date* du Todo à *now* dès que le booléen *done* passe à *True*. Il faut pour cela surcharger la méthode *save()* de *Model* (avec ses args et kwargs, pourquoi à votre avis ?) et aussi ne pas oublier de faire appel à la méthode save() de la superclasse ...

```
#models.py
from django.db import models
from django.utils.timezone import now

class ToDo(models.Model):
    name = models.CharField(max_length=250)
    begin_date = models.DateField(auto_now_add=True)
    end_date = models.DateField()
    done = models.BooleanField(default=False)

    def __str__(self):
        return self.name

    def save(self, *args, **kwargs):
        if (self.done and self.end_date is None):
                self.end_date = now()

        super(ToDo, self).save(*args, **kwargs)
```

Réalisons les migrations.

```
./manage.py makemigrations
./manage.py migrate
```

Testez dans l'admin. Commitez.

3.5 Modèle et admin de Django

On peut à présent décrire notre modèle puis l'afficher dans l'admin.

3.5.1 Modèle ToDo

```
#models.py
class ToDo(models.Model):
    name = models.CharField(max_length=250)
    begin_date = models.DateField(auto_now_add=True)
    end_date = models.DateField()
    done = models.BooleanField(default=False)

    def __str__(self):
        return self.name
```

3.5.2 Migrations

3.5.3 Pour que Django prenne en compte les modifications de structure de la BD

```
./manage.py makemigrations
./manage.py sqlmigrate planning 0001
./manage.py migrate
```

— la commande *makemigrations* prépare les migrations
— la commande *sqlmigrate* décrit en sql la migration qui va se faire en base de données
— La commande *migrate* l'effectue

3.5.4 Et pour voir le nouveau modèle dans l'admin de Django

Complétons le fichier *admin.py*

```
# admin.py
from django.contrib import admin
from planning.models import ToDo


class ToDoAdmin(admin.ModelAdmin):
    list_display=(
    'name', 'begin_date', 'end_date','done',
)

admin.site.register(ToDo,ToDoAdmin)
```

3.5.5 Ajout de debug-toolbar

On peut ajouter django-debug-toolbar à notre venv pour voir cet outil à l'oeuvre. (les autres réglages nécessaires ont été faits dans settings.py)

```
pip install django-debug-toolbar
```

Corrigez, les requirements :

```
pip freeze > requirements.txt
```

Commitez.

3.5.6 Créons un superuser dans Django pour accéder à l'admin

```
./manage.py createsuperuser
```

Puis visitez l'admin : Localhost:8000/admin/ (http ://localhost :8000/admin/) sous cette identité. Créez quelques ToDo. Observez la debug-toolbar et essayez ses différents Panels.

3.5.7 Améliorons un peu le rendu de l'admin des ToDo

```
@admin.register(ToDo)
class ToDoAdmin(admin.ModelAdmin):
        fieldsets = [
                ('ToDo Details', {'fields': ['name', 'done']}),
                ('ToDo Dates', {'fields': ['begin_date', 'end_date
↪']}),
        ]

        readonly_fields = ('begin_date',)

        list_display = ('name', 'begin_date', 'end_date','done',)
        list_editable = ('done',)
        list_display_links = ('name', 'date_reviewed',)
        list_filter = ('done',)
        search_fields = ['name',]
```

Testez. Remarquez le décorateur qui sert à enregistrer directement le modèle ToDo dans l'admin et aussi les améliorations dans l'affichage des ToDo dans l'admin. Commitez.

3.5.8 Enrichissement du Modèle ToDo

Ajoutons par exemple le fait de mettre automatiquement *end_date* du Todo à *now* dès que le booléen *done* passe à *True*. Il faut pour cela surcharger la méthode *save()* de *Model* (avec ses args et kwargs, pourquoi à votre avis ?) et aussi ne pas oublier de faire appel à la méthode save() de la superclasse ...

```
#models.py
from django.db import models
from django.utils.timezone import now

class ToDo(models.Model):
    name = models.CharField(max_length=250)
    begin_date = models.DateField(auto_now_add=True)
    end_date = models.DateField()
    done = models.BooleanField(default=False)

    def __str__(self):
        return self.name

    def save(self, *args, **kwargs):
```

```
        if (self.done and self.end_date is None):
                self.end_date = now()

        super(ToDo, self).save(*args, **kwargs)
```

Réalisons les migrations.

```
./manage.py makemigrations
./manage.py migrate
```

Testez dans l'admin. Commitez.

3.6 Tester une application Python Django - TDD

Nous allons à présent nous attaquer à une problématique fondamentale dans toute application qu'elle soit Web, mobile ou autres : Les tests.

3.6.1 TDD

TDD veut dire *Test Driven Development* c'est à dire *Développement dirigé par les tests* C'est une démarche mise en avant en *Méthodologie Agile* Elle consiste en général en l'application des points suivants :

- écrire un test
- vérifier qu'il échoue (car le code qu'il teste n'existe pas)
- commiter votre code
- écrire juste le code suffisant pour passer le test
- vérifier que le test passe
- faire un commit
- procéder à un refactoring du code, c'est-à-dire l'améliorer en gardant les mêmes fonctionnalités.
- vérifier que le test passe toujours
- commiter
- etc.

3.6.2 Intérêt de la démarche :

Les avantages principaux de cette démarche sont :

- Préciser au mieux les spécifications du code et l'API envisagée
- Ceci oblige à faire des choix de conception qui restent parfois trop dans le flou au début du développement
- Plus tard, disposer d'une large base de tests est une riche pour une application car elle permet de vérifier à tout moment que les tests installés ne sont pas mis en défaut par de nouveaux développements ou des refactoring de code

Tous les langages de programmation disposent de Frameworks de tests. Par exemple Java offre *JUnit*, PHP quand à lui propose *PHPUnit*. Python propose *unittest* et permet aussi d'utiliser facilement la librairie selenium pour réaliser des tests fonctionnels.

Prérequis

Installer selenium via pip et phantomjs via npm :

```
pip install selenium
npm install phantomjs-g
```

Eventuellement, installer selenium-standalone, qui offre plus de possibilités pour utiliser des browsers différents.

```
npm install selenium-standalone@latest -g
selenium-standalone install
selenium-standalone start
```

Pour vérifier les paquets npm globalement installés :

```
npm list -g --depth=0
```

Pour vérifier les paquets npm localement installés :

```
npm list  --depth=0
```

Puis écrivons notre premier test fonctionnel dans le dossier Tests :

```
from selenium import webdriver
import time

browser = webdriver.Chrome(executable_path='./chromedriver')
time.sleep(3)
browser.get('http://localhost:8000')

assert 'Django' in browser.title
browser.quit()
```

Pour tester :

```
python functional-test1.py
```

Danger : python est python35 normalement (celui du venv que vous avez créé) Vous devrez remettre temporairement DEBUG = FALSE pour que ce teste passe. Pourquoi ?

Pour l'instant notre test échoue si notre serveur est arrété ou si il est en mode DEBUG. Commitons.

Relançons le serveur . . .

```
./manage.py runserver
```

Puis relançons le test et constatons qu'il passe. Commitons.

Un test fonctionnel un peu plus riche fondé sur une *User Story*

```
from selenium import webdriver
import unittest

class UnitTest1(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Chrome(executable_path='./
↪chromedriver')
        self.browser.implicitly_wait(3)

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

    def test_can_start_a_list_and_retrieve_it_later(self):
        # Alice se rend sur le site localhost:8000
        # et escompte y trouver une app de Planning
        self.browser.get('http://localhost:8000/planning')

        # Elle remarque que le mot Planning
        # figure dans le titre de la page
        self.assertIn('Planning', self.browser.title)  #
        self.fail('Test terminé !')

        # Elle peut aussi créer une nouvelle tâche ...
        # Puis constater qu'elle figure bien
        # dans la page qui liste les tâches ...
```

3.7 Tester une application Django

3.7.1 Un test unitaire bébête :

```
# A mettre dans le fichier tests.py de l'app Django
from django.core.urlresolvers import resolve
from django.test import TestCase
from lists.views import home
from django.template.loader import render_to_string
```

```
class StringTest(TestCase):
    '''Test unitaire bidon'''
    def test_concatene(self):
        self.assertEqual("Bon"+"jour", "Bonjour")
```

3.7.2 Un test qui vérifie la méthode gérant la racine du site :

```
# A mettre dans le fichier tests.py de l'app Django
from django.core.urlresolvers import resolve
from django.test import TestCase
from lists.views import home


class HomePageTest(TestCase):
'''
Test unitaire de la page accueil sur la racine du projet
On vérifie que la méthode home() est bien invoquée sur /
'''
def test_root_url_resolves_to_home_view(self):
  found = resolve('/')
  self.assertEqual(found.func, home)
```

3.7.3 Puis un test vérifiant si le contenu présenté correspond bien à celui du template home.html :

```
# A mettre dans le fichier tests.py de l'app Django
from django.core.urlresolvers import resolve
from django.test import TestCase
from lists.views import home
from django.template.loader import render_to_string


class HomePageTestContent(TestCase):
  '''Test Unitaire pour vérifier si le contenu de la page d'accueil␣
↪est bien retourné par home()'''
  request = HttpRequest()
  response = home(request)
  expected_html = render_to_string('home.html')
  self.assertEqual(response.content.decode(), expected_html)
```

Pour tester :

```
./manage.py test
```

Constatez l'échec du test. Puis ajouter la route et la vue correspondante pour que le test passe ...

3.8 Faire des tests plus complets avec Selenium

3.8.1 Un test d'existence d'un élément ayant une id spécifique :

```
# A mettre dans le fichier tests.py
# de l'app Django
from selenium import webdriver
import time

browser = webdriver.Firefox()
time.sleep(10)
browser.get('http://www.univ-orleans.fr')
assert 'Université' in browser.title

elem = browser.find_element_by_id('banniere')
assert(elem is not None)

browser.quit()
```

3.8.2 Un test qui remplit un formulaire puis vérifie un nombre de liens

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
    import time
    browser = webdriver.Chrome()

    browser.get('http://www.univ-orleans.fr')


    elem = browser.find_element_by_name('search_block_form')
    elem.send_keys('iut orleans' + Keys.RETURN)
    liens = browser.find_element_by_class_name('content').find_
↪elements_by_tag_name('a')

    # Vérifions qu'il y a au moins 3 liens
    # sur cette recherche
    assert(len(liens)>=3)
    # Puis affichons ces liens
    for l in liens:
        print(l.get_attribute('href'))
```

```
    time.sleep(5)
    browser.quit()
```

3.8.3 Puis un test vérifiant une authentification sur une page Web :

Prenons une page d'authentification qui attend comme login "root" avec comme password "azerty" :

```
<?php
// Definition des constantes et variables
define('LOGIN','root');
define('PASSWORD','azerty');
$errorMessage = '';
// Test de l'envoi du formulaire
if(!empty($_POST))
{
  // Les identifiants sont transmis ?
  if(!empty($_POST['login']) && !empty($_POST['password']))
  {
    // Sont-ils ceux attendus ?
    if($_POST['login'] != LOGIN)
    {
      $errorMessage = 'Mauvais login !';
    }
    elseif($_POST['password'] != PASSWORD)
    {
      $errorMessage = 'Mauvais password !';
    }
    else //tout va bien
    {
    // On ouvre la session
    session_start();
    // On enregistre le login en variable de session
    $_SESSION['login'] = LOGIN;
    // On redirige vers le fichier suite.php
    header('Location: suite.php');
  }
}
else
{
  $errorMessage = 'Veuillez inscrire vos identifiants svp !';
}
}
?>

<!DOCTYPE HTML>
<html lang="fr">
<head>
<title>Formulaire d'authentification</title>
```

```
</head>
<body>
<?php
if (!empty($errorMessage)){
  echo $errorMessage;
}
?>
<form action="authentification.php" method="post">
<fieldset>
<legend>Identifiez-vous</legend>
<p>
<label for="login">Login :</label>
<input type="text" name="login" value="" />
</p>
<p>
<label for="password">Password :</label>
<input type="password" name="password" value=""/>
<input type="submit" value="Se logguer" />
</p>
</fieldset>
</form>
</body>
</html>
```

3.8.4 Puis écrivons un test fonctionnel correspondant à une authentification réussie :

```
from selenium import webdriver

baseurl = "http://localhost/~roza/authentification.php"
username = "root"
password = "azerty"

xpaths = { 'loginBox' : "//input[@name='login']",
           'passwordBox' : "//input[@name='password']",
           'submitButton' :   "//input[@type='submit']"
         }

mydriver = webdriver.Chrome(executable_path='./chromedriver')
mydriver.get(baseurl)
mydriver.maximize_window()

#Clear Username TextBox if already allowed "Remember Me"
mydriver.find_element_by_xpath(xpaths['loginBox']).clear()

#Write Username in Username TextBox
mydriver.find_element_by_xpath(xpaths['loginBox']).send_
 ↪keys(username)
```

```
#Clear Password TextBox if already allowed "Remember Me"
mydriver.find_element_by_xpath(xpaths['passwordBox']).clear()

#Write Password in password TextBox
mydriver.find_element_by_xpath(xpaths['passwordBox']).send_
 ↪keys(password)

#Click Login button
mydriver.find_element_by_xpath(xpaths['submitButton']).click()

print(mydriver.page_source)

assert "Bienvenue ici root" in mydriver.page_source
mydriver.close()
```

3.9 Alice démarre avec git :

3.9.1 Paramétrage et initialisations :

On configure d'abord ses paramètres

```
git config --global user.name "Alice Torvalds"
git config --global user.email "alice@kernel.org"
```

3.9.2 Création d'un dossier local versionné

```
mkdir monprojet
cd monprojet
git init
```

Si vous avez déja du contenu :

```
git add .
```

3.9.3 Création d'un dépot « monprojet » sur gitlab

— Privé
— Public
— ou Interne à gitlab

Bitbucket (https ://bitbucket.org/) offre également la possibilité d'avoir des dépôts privés de taille limitée.

Github (https ://github.com/) offre les dépôts public et fait payer les dépôts privés.

3.9.4 Connexion entre le local et le gitlab :

Eventuellement : git config push.default simple

```
git remote add origin https://gitlab.com/alice/monprojet.git
git push -u origin master
```

ou simplement :

```
git push
```

par la suite

3.9.5 Réalisation d'une fonctionnalité par Alice :

- Alice prend une chose à réaliser et implémente le code nécessaire
- Alice fait les tests et vérifie que ça marche
- git commit -am « Message de commit »

3.9.6 Alice pousse son master sur son remote :

```
git push -u origin master
```

3.10 Bob travaille avec Alice grâce à git :

Bob fait d'abord comme Alice pour paramétrer et initialiser son dépôt local.

3.10.1 Bob vérifie qu'il a bien 2 remotes :

- **Le sien, origin qu'il crée au besoin en faisant :**

  ```
  git remote add origin https://gitlab.com/bob/monprojet.git
  ```

- **celui d'Alice qu'il ajoute :**

  ```
  git remote add alice https://gitlab.com/alice/monprojet.git
  ```

- il tape git remote -v pour vérifier ses remotes
- **Si il se trompe :**

  ```
  git remote remove alice
  ```

3.10.2 Bob récupère le master d'Alice :

```
git fetch Alice master
```

3.10.3 Bob consulte la branche locale correspondant au master d'Alice :

```
git branch -av
git checkout Alice/master
```

puis vérifie que le code d'Alice est correct

3.10.4 Bob revient dans son master :

```
git checkout master
```

3.10.5 Bob merge le travail d'Alice et pousse les modifs dans son dépôt distant :

```
git merge Alice/master
git push
```

Puis detruit la branche locale d'Alice :

```
git branch -d Alice/master
```

3.11 Alice se met à jour :

— ajoute le remote de Bob
— **fetche le master de Bob pour se mettre à jour :**

```
git fetch Bob master
```

— **Fusionne :**

```
git merge Bob/master
```

3.12 Alice travaille sur une branche git :

Alice doit par exemple intégrer une feature de connexion à une base de données. Elle va pour cela créer une branche *bd* dédiée à la réalisation de cette feature et se placer dedans.

3.12.1 Création et choix de la branche :

```
git checkout -b bd
```

Elle fait ensuite son travail, le teste puis :

```
git commit -am "Intégration BD"
```

3.12.2 Alice pousse sa branche sur son remote :

```
git push origin bd
```

3.13 Bob et la branche d'Alice :

3.13.1 Bob récupère la branche d'Alice :

```
git fetch Alice bd
```

3.13.2 Bob consulte la branche d'Alice :

S'il le souhaite, Bob consulte la liste des branches disponibles puis se place dans la branche d'Alice pour faire une petite revue du code de sa collaboratrice…

```
git branch -av
git checkout Alice/bd
```

3.13.3 Bob revient dans sa branche master :

```
git checkout master
```

3.13.4 Bob merge la branche d'Alice et pousse les modifs :

```
git merge Alice/bd
git push
```

3.14 Alice récupère la dernière version du master :

3.14.1 Alice fetche le master de Bob pour se mettre à jour :

```
git fetch Bob master
git merge Bob/master
```

3.14.2 Alice efface sa branche bd :

```
git branch -d bd
```

CHAPITRE 4

GIT

Tout bon développeur doit aujourd'hui savoir utiliser un système de gestion de version pour son code et pour collaborer. Git est aujourd'hui le plus répandu. Vous trouverez à la fin de ce cours un rappel des principales commandes git pour démarrer :

GIT start

et quelques commandes pour travailler à plusieurs sur un projet avec les branches git :

GIT branches

CHAPITRE 5

Références

— Doc Django (https ://docs.djangoproject.com/fr/1.10/)
— Tutoriel Django (https ://docs.djangoproject.com/fr/1.10/intro/tutorial01/)
— TDD with Python (http ://www.obeythetestinggoat.com/pages/book.html)
— Tuto Django TDD (http ://www.marinamele.com/taskbuster-django-tutorial)

CHAPITRE 6

Index et recherche

— genindex
— search

Index