Objectifs de la feuille

- Tester le modèle
- Tester les vues
- Tester les formulaires
- Couverture de test

Introduction

Mettre en place des tests unitaires dans une app Flask est une excellente pratique. Voici comment structurer des tests unitaires pour les routes liées aux auteurs, étape par étape. Commençons par l'installation de pytest pour Flask et des outils permettant de mesurer la couverture de test :

```
$ pip install pytest pytest-flask coverage
```

La configuration des tests

Dans le répertoire tests, créez le fichier conftest.py, fichier qui portera la configuration Flask pour les tests. Notamment, cela permet de créer une base de test et une application de test pour éviter de polluer notre base de production et l'application associée. Il permet également de créer un jeu de données d'essais pour les tests. Voici à quoi ressemble votre fichier conftest.py :

```
import pytest
from monApp import app,db
from monApp.models import Auteur

@pytest.fixture
def testapp():
        app.config.update({"TESTING":True,"SQLALCHEMY_DATABASE_URI":
        "sqlite:///:memory:","WTF_CSRF_ENABLED": False})

        with app.app_context():
                db.create_all()
                # Ajouter un auteur de test
                auteur = Auteur(Nom="Victor Hugo")
                db.session.add(auteur)
                db.session.commit()
        yield app

        # Cleanup après les tests
        with app.app_context():
                db.drop_all()

@pytest.fixture
def client(testapp):
        return testapp.test_client()
```

Vous pouvez déjà lancer les tests pour voir ce qu'il en est : $ coverage run -m pytest

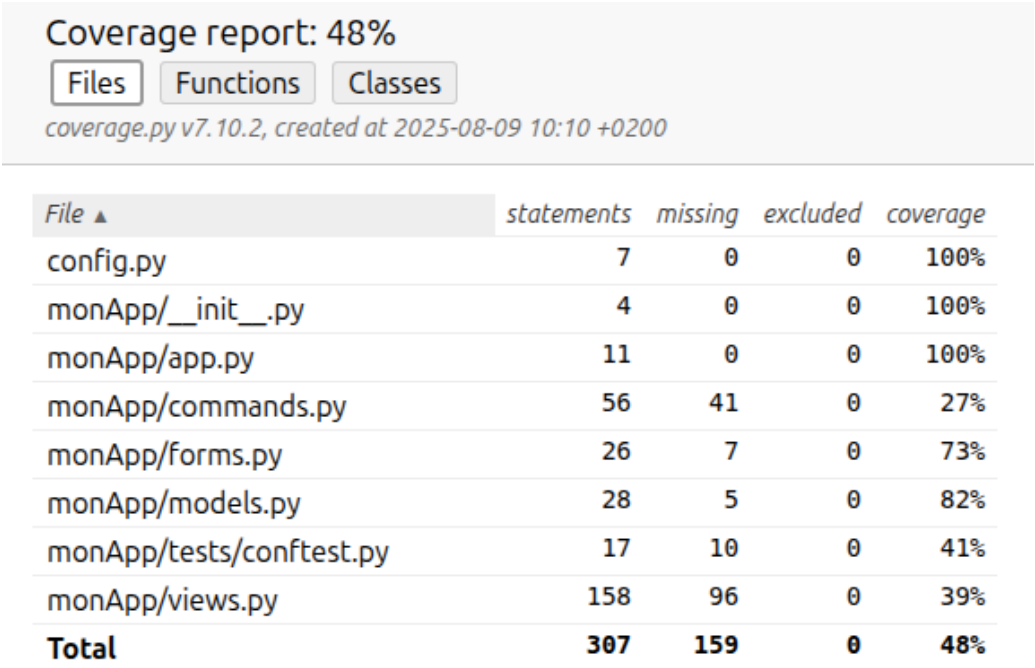
Et même générer un rapport dans le terminal : $ coverage report -m

```
Name                       Stmts   Miss  Cover   Missing
--------------------------------------------------------
config.py                      7      0   100%
monApp/__init__.py             4      0   100%
monApp/app.py                 11      0   100%
monApp/commands.py            56     41    27%   10-41, 47-48, 56-63, 70-80
monApp/forms.py               26      7    73%   25-31
monApp/models.py              28      5    82%   9, 12, 25, 32, 38
monApp/tests/conftest.py      17     10    41%   7-18, 22
monApp/views.py              158     96    39%   11-15, 19, 23, 27-28, 33-35, 40-51, 55-57, 62-63, 68-77, 82-84, 89-97, 101-102, 107-109, 114-126, 130-132, 137-138, 143-154, 159-161,
 166-176, 180-191, 197-198, 201
--------------------------------------------------------
TOTAL                        307    159    48%
```

Le rapport vous indique là où il y a des manquements. Si vous ne trouvez pas cela très exploitable, vous pouvez toujours générer ce rapport au format HTML : ,

$ coverage html

Ci-dessous le résultat. Vous n'avez plus les indications des lignes non testées mais en cliquant sur les différents liens, vous pouvez les retrouver.



File ▲	statements	missing	excluded	coverage
config.py	7	0	0	100%
monApp/__init__.py	4	0	0	100%
monApp/app.py	11	0	0	100%
monApp/commands.py	56	41	0	27%
monApp/forms.py	26	7	0	73%
monApp/models.py	28	5	0	82%
monApp/tests/conftest.py	17	10	0	41%
monApp/views.py	158	96	0	39%
Total	**307**	**159**	**0**	**48%**

Tests unitaires du modèle

Nous allons nous occuper du modèle testé à 82 %. Créez le répertoire unit dans tests/ , puis créez un fichier __init__.py (vide) et test_models_auteur.py avec le contenu suivant :

```
from monApp.models import Auteur

def test_auteur_init():
    auteur = Auteur("Cricri DAL")
    assert auteur.Nom == "Cricri DAL"

def test_auteur_repr(testapp):    #testapp est la fixture définie dans conftest.py
    with testapp.app_context():
        auteur=Auteur.query.get(1)
        assert repr(auteur) == "<Auteur (1) Victor Hugo>"
```

Le modèle est à présent testé à 89 %.

```
Name                                    Stmts   Miss  Cover   Missing
---------------------------------------------------------------------
config.py                                   7      0   100%
monApp/__init__.py                          4      0   100%
monApp/app.py                              11      0   100%
monApp/commands.py                         56     41    27%   10-41, 47-48, 56-63, 70-80
monApp/forms.py                            26      7    73%   25-31
monApp/models.py                           28      3    89%   25, 32, 38
monApp/tests/conftest.py                   17      1    94%   24
monApp/tests/unit/__init__.py               0      0   100%
monApp/tests/unit/test_models_auteur.py     8      0   100%
monApp/views.py                           158     96    39%   11-15, 19, 23, 27-28, 33-35, 40-51, 55-57, 62-63, 68-77, 82-84, 89-97, 101-102, 107-109, 114-126, 130-132, 137-138,
3-154, 159-161, 166-176, 180-191, 197-198, 201
---------------------------------------------------------------------
TOTAL                                     315    148    53%
```

Je vous conseille de vous créer un petit fichier .coveragerc à la racine du Projet TutoFlask pour indiquer les fichiers à ne pas prendre en compte dans le taux de couverture de test :

```
[run]
omit =
   */tests/*
   */__init__.py
   */config.py
```

```
Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
monApp/app.py          11      0   100%
monApp/commands.py     56     41    27%   10-41, 47-48, 56-63, 70-80
monApp/forms.py        26      7    73%   25-31
monApp/models.py       28      3    89%   25, 32, 38
monApp/views.py       158     96    39%   11-15, 19, 23, 27-28, 33-35, 40-51, 55-57, 62-63, 68-77, 82-84, 89-97, 101-102, 107-109, 114-126, 130-132, 137-138, 143-154, 159-161,
76, 180-191, 197-198, 201
-------------------------------------------------
TOTAL                279    147    47%
```

C'est à vous de jouer

Il vous reste à tester Livre , User et la fonction load_user( ) dans le modèle. En rajoutant un livre et un user dans votre base de données de test dans la fonction testapp( ) du fichier conftest.py, vous pouvez écrire des tests unitaires dans deux fichiers test_models_livre.py et test_models_user.py dans le répertoire tests/unit/.

Tests fonctionnels des vues et des urls de type GET

Le modèle est couvert à 100% à présent. Occupons-nous maintenant des routes. Créez le répertoire functional dans tests/ , puis créez un fichier __init__.py (vide) et test_routes_auteur.py avec le contenu suivant :

```
def test_auteurs_liste(client):      #client est la fixture définie dans conftest.py
    response = client.get('/auteurs/')
    assert response.status_code == 200
    assert b'Victor Hugo' in response.data
```

Pour ce test, nous nous servons de client défini dans conftest.py et l'on vérifie que la vue s'affiche bien (réponse HTTP = 200) et que cela concerne bien notre premier élément de notre base de test.

Certaines de nos vues nécessitent que l'utilisateur soit authentifié. Il faut donc tester la redirection vers la page de login :

```
def test_auteur_update_before_login(client):
    response = client.get('/auteurs/1/update/', follow_redirects=True)
    assert b"Login" in response.data # vérifier redirection vers page Login
```

Ce n'est bien sur pas suffisant ! Il faut également tester si la vue s'affiche bien une fois l'utilisateur authentifié. Nous avons besoin d'une méthode login( ) qui simule la connexion. Elle prend en paramètre le login, le mot de passe, et l'url de destination post connexion :

```
def login(client, username, password, next_path):
    return client.post(   "/login/",
                          data={"Login": username,"Password": password, "next":next_path} ,
                          follow_redirects=True)

def test_auteur_update_after_login(client,testapp):
    with app.app_context():
        # user non connecté
        response=client.get('/auteurs/1/update/', follow_redirects=False)
        # Redirection vers la page de login
        assert response.status_code == 302
        # vérification redirection vers page Login
        assert "/login/?next=%2Fauteurs%2F1%2Fupdate%2F" in response.headers["Location"]
        # simulation connexion user
        response=login(client, "CDAL", "AIGRE", "/auteurs/1/update/")
        # Page update après connexion
        assert response.status_code == 200
        assert b"Modification de l'auteur Victor Hugo" in response.data
```

C'est à vous de jouer

Il vous reste à tester :

- la vue/url /auteurs/1/view/ qui ne demande pas de connection
- les vues/urls /auteurs/1/delete et /auteur/ (pour la création) avant et après authentification

Tests fonctionnels des formulaires et des urls de type POST

Jusqu'à présent, nous avons faire le tour des vues basées sur de méthodes de type GET. Il nous faut tester les trois vues basées sur les méthodes POST, celles évoquées lors de la soumission des différents formulaires de modification, création ou suppression. Pour cela, nous allons créer un nouveau fichier test_forms_auteur.py.

Nous allons nous pencher sur la vue /save/auteur/. Nous allons créer un second auteur dans notre base de données de test pour commencer. Aussi, nous devons nous authentifier pour pouvoir modifier ce nouvel auteur en BDD ; nous ferons appel à la méthode login() précédente Puis nous soumettrons le formulaire via la méthode POST /save/auteur/ en modifiant le nom de l'auteur. Ensuite, nous vérifierons que nous avons bien été redirigés vers la vue de consultation de nouvel auteur /auteurs/<idA>/view/ avec le nom modifié. Enfin nous vérifierons en BDD que le nom a bien changé. Ce qui nous donne ceci :

```
from monApp.models import Auteur
from monApp import db
from monApp.tests.functional.test_routes_auteur import login

def test_auteur_save_success(client, testapp):
    # Créer un auteur dans la base de données
    with testapp.app_context():
        auteur = Auteur(Nom="Ancien Nom")
        db.session.add(auteur)
        db.session.commit()
        idA = auteur.idA

        # simulation connexion user et soumission du formulaire
        response=login(client, "CDAL", "AIGRE", "/auteur/save/")
        response = client.post("/auteur/save/",
        data={"idA": idA,"Nom": "Alexandre Dumas"},
        follow_redirects=True)

        # Vérifier que la redirection a eu lieu vers /auteurs/<idA>/view/ et que le contenu
    # est correct
        assert response.status_code == 200
        assert f"/auteurs/{idA}/view/" in response.request.path
        assert b"Alexandre Dumas" in response.data # contenu de la page vue

        # Vérifier que la base a été mise à jour
        with testapp.app_context():
            auteur = Auteur.query.get(idA)
            assert auteur.Nom == "Alexandre Dumas"
```

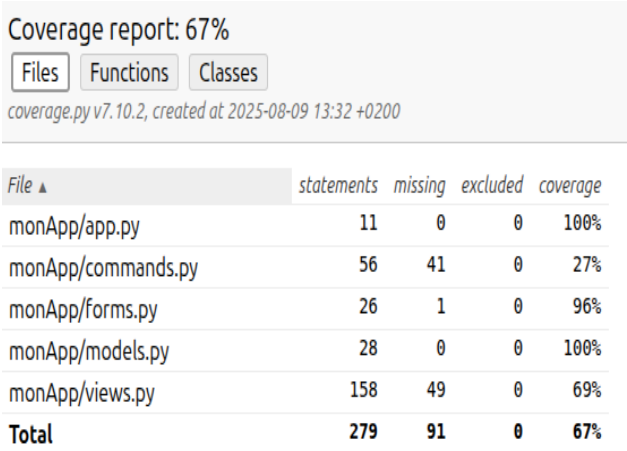
C'est à vous de jouer

Fort de cette expérience, essayez de faire de même avec les méthodes POST qui nécessitent d'être authentifié, à savoir :

- /auteur/insert/ pour la création
- /auteur/erase/ pour la suppression

C'est à vous de jouer

Voyez par vous-même, il nous reste un peu de travail…


Coverage report: 67%

Files | Functions | Classes

coverage.py v7.10.2, created at 2025-08-09 13:32 +0200

File ▲	statements	missing	excluded	coverage
monApp/app.py	11	0	0	100%
monApp/commands.py	56	41	0	27%
monApp/forms.py	26	1	0	96%
monApp/models.py	28	0	0	100%
monApp/views.py	158	49	0	69%
Total	**279**	**91**	**0**	**67%**

Essayez d'atteindre plus de 90 % de couverture de tests en travaillant sur les tests fonctionnels des routes GET et POST pour les livres.Et si cela ne suffit pas, vous pouvez tester la vue index, about, contact et logout.

Pour ma part, à la fin de ce TP, je suis à 97 %.