Objectifs de la feuille

- Tester les modèles
- Tester les formulaires
- Tester les URLs
- Tester les vues
- Taux de couverture de test

Tester les modèles

Tester les modèles dans Django consiste généralement à vérifier que les objets peuvent être créés, modifiés, supprimés, et que les méthodes personnalisées du modèle, si vous en avez, fonctionnent comme prévu. Les tests unitaires de DJANGO utilisent le module `unittest` de la bibliothèque Python standard. Ce module définit les tests selon une approche basée sur des classes.

Les tests ayant besoin d'une base de données (à savoir les tests avec des modèles) n'utilisent pas la «vraie» base de données (de production). Des bases de données distinctes et vierges sont créées tout exprès pour les tests. Que les tests passent ou échouent, les bases de données de test sont supprimées dès que tous les tests ont été exécutés.

Pour tester un modèle, vous pouvez utiliser la classe `TestCase` de DJANGO, qui vous permet d'interagir avec une base de données pendant les tests. Cela inclut la création et la manipulation d'objets du modèle. Une classe héritant de `TestCase`, elle-même héritant de `unittest`, lance chaque test dans une transaction pour garantir l'isolation.

Si des tests dépendent de l'accès à une base de données pour créer ou interroger des modèles, les classes de test doivent être des sous-classes de `django.test.TestCase` plutôt que de `unittest.TestCase`. En utilisant `unittest.TestCase`, on allège les tests en évitant l'étape d'envelopper chaque test dans une transaction et de réinitialiser la base de données. Mais si les tests concernés interagissent avec la base de données, leur comportement variera en fonction de l'ordre dans lequel ils seront lancés par le lanceur de tests. Cela peut conduire certains tests unitaires à réussir lorsqu'il sont exécutés individuellement, mais à échouer lorsqu'ils font partie d'une suite.

Le gabarit par défaut de `startapp` crée un fichier `tests.py` dans la nouvelle application. Cela convient bien quand il n'y a que quelques tests. Mais dès que la suite de tests grandit, il devient préférable de restructurer ce fichier en un module Python afin de pouvoir séparer les tests en sous-modules tels que `test_models.py`, `test_views.py`, `test_forms.py`, etc. Vous êtes libre de choisir le découpage structurel qui vous convient.

¨Pour commencer, nous allons appeler notre premier fichier tests_ProductAttributeValue_Model.py, autrement dit nous allons tester le modèle de ProductAttributeValue.
Notre classe ProductAttributeValueModelTest héritera de TestCase.

```
from django.test import TestCase
from monapp.models import ProductAttribute, ProductAttributeValue

class ProductAttributeValueModelTest(TestCase):
```

Une première méthode setUp( ) permet de créer les objets nécessaires pour chaque test. Dans cet exemple, un attribut de produit (ProductAttribute) et une valeur d'attribut (ProductAttributeValue) sont créés.

```
    def setUp(self):
        # Créer un attribut produit à utiliser dans les tests
        self.attribute = ProductAttribute.objects.create(name="Couleur")

        # Créer une valeur pour cet attribut
        self.value = ProductAttributeValue.objects.create(value="Vert",
            product_attribute=self.attribute, position=1)
```

Le test test_product_attribute_value_creation( ) vérifie que la création de l'objet ProductAttributeValue se passe correctement. Il s'assure que la valeur de l'attribut, le lien avec ProductAttribute, et la position sont bien enregistrés.

```
    def test_product_attribute_value_creation(self):
    """
    Tester si une ProductAttributeValue est bien créée
    """
        self.assertEqual(self.value.value, "Vert")
        self.assertEqual(self.value.product_attribute.name, "Couleur")
        self.assertEqual(self.value.position, 1)
```

Le test test_string_representation vérifie que la méthode __str__() du modèle renvoie bien le format correct (comme vous l'avez défini dans votre modèle).

```
    def test_string_representation(self):
    """
    Tester la méthode __str__ du modèle ProductAttributeValue
    """
        self.assertEqual(str(self.value), "Vert [Couleur]")
```

Le test test_update_product_attribute_value( ) vérifie la capacité à mettre à jour un objet ProductAttributeValue en changeant la valeur et en s'assurant que la modification est bien enregistrée en base de données.

```
def test_update_product_attribute_value(self):
"""
Tester la mise à jour d'une ProductAttributeValue
"""
        self.value.value = "Orange"
        self.value.save()

        # Récupérer l'objet mis à jour
        updated_value = ProductAttributeValue.objects.get(id=self.value.id)
        self.assertEqual(updated_value.value, "Orange")
```

Le test test_delete_product_attribute_value( ) vérifie que l'objet peut être supprimé, et qu'il n'est plus présent dans la base de données après la suppression.

```
def test_delete_product_attribute_value(self):
"""
Tester la suppression d'une ProductAttributeValue
"""
        self.value.delete()
        self.assertEqual(ProductAttributeValue.objects.count(), 0)
```

Notre premier fichier sera positionné dans le répertoire monapp/tests/Model/. Vous aurez besoin d'un fichier __init__.py dans le sous-dossier pour que DJANGO reconnaisse les modules. Pour exécuter les tests, vous pouvez utiliser la commande suivante dans votre terminal à la racine de votre projet DJANGO :

```
./manage.py test monapp.tests.Model.tests_ProductAttributeValue_Model
```

```
Found 4 test(s).
Creating test database for alias 'default'...
System check identified some issues:

WARNINGS:
monapp.ProductItem.attributes: (fields.W340) null has no effect on ManyToManyField.

System check identified 1 issue (0 silenced).
....
----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK
Destroying test database for alias 'default'...
```

Lorsque vous lancez vos tests, le comportement par défaut de l'utilitaire de test est de rechercher toutes les classes de cas de test dans tous les fichiers dont le nom commence par test, puis de construire automatiquement une suite de tests et d'exécuter cette suite.

Tester les formulaires

Tester les formulaires (forms) dans DJANGO est une bonne pratique pour s'assurer que les champs sont correctement validés et que le formulaire fonctionne comme prévu dans différentes situations. Nous allons donc tester le formulaire ProductAttributeValueForm. Vous pouvez organiser vos tests en créant un fichier tests_ProductAttributeValue_Form.py dans le répertoire monapp/tests/Form/ que vous créerez. Là encore, vous aurez besoin d'un fichier __init__.py

Voici le début de la classe ProductAttributeValueFormTest :

```
from django.test import TestCase
from monapp.forms import ProductAttributeValueForm
from monapp.models import ProductAttribute, ProductAttributeValue

class ProductAttributeValueFormTest(TestCase):

    def setUp(self):
        self.attribute = ProductAttribute.objects.create(name="Couleur")
```

Voici comment vous pourriez écrire des tests pour ce formulaire en vous assurant qu'il fonctionne comme prévu avec toutes les validations nécessaires.

La méthode test_form_valid_data( ) vérifie que le formulaire est valide avec des données correctes.

```
def test_form_valid_data(self):
    """
    Tester que le formulaire est valide avec des données correctes
    """
    form = ProductAttributeValueForm ( data = {  'value': 'Cyan',
                                                 'product_attribute': self.attribute.id,
                                                 'position': 1} )
    self.assertTrue(form.is_valid()) # Le formulaire doit être valide
```

La méthode test_form_invalid_data( ) vérifie que le formulaire est invalide si le champ value est manquant.

```
def test_form_invalid_data(self):
    """
    Tester que le formulaire est invalide si 'value' est manquant
    """
    form = ProductAttributeValueForm ( data = {  'product_attribute': self.attribute.id,
                                                 'position': 1 } )
    self.assertFalse(form.is_valid()) # Le formulaire ne doit pas être valide
    self.assertIn('value', form.errors) # Le champ 'value' doit contenir une erreur
```

La méthode test_form_invalid_product_attribute( ) vérifie que le formulaire est invalide si le champ product_attribute est manquant.

```
def test_form_invalid_product_attribute(self):
    """
    Tester que le formulaire est invalide si 'product_attribute' est manquant
    """
    form = ProductAttributeValueForm ( data = {  'value': 'Cyan', 'position': 1 } )
    # Le formulaire ne doit pas être valide
    self.assertFalse(form.is_valid())
    # Le champ 'product_attribute' doit contenir une erreur
    self.assertIn('product_attribute', form.errors)
```

La méthode test_form_optional_position( ) vérifie que le formulaire est valide même si le champ position est omis.

```
def test_form_optional_position(self):
    """
    Tester que le formulaire est valide même sans la position (champ facultatif)
    """
    form = ProductAttributeValueForm ( data = {  'value': 'Vert',
                                                 'product_attribute': self.attribute.id,
                                                 'position': None } )
    # Le formulaire doit être valide même sans la position
    self.assertTrue(form.is_valid())
```

La méthode test_form_save( ) vérifie que le formulaire peut être enregistré avec des données valides et que les données sauvegardées correspondent à celles fournies.

```
def test_form_save(self):
    """
    Tester que le formulaire peut être enregistré avec des données valides
    """
    form = ProductAttributeValueForm ( data = {  'value': 'Bleu',
                                                 'product_attribute': self.attribute.id,
                                                 'position': 2 } )
    self.assertTrue(form.is_valid())
    product_attribute_value = form.save()
    self.assertEqual(product_attribute_value.value, 'Bleu')
    self.assertEqual(product_attribute_value.product_attribute, self.attribute)
    self.assertEqual(product_attribute_value.position, 2)
```

Exécutez vos tests ./manage.py test monapp.tests.Form.tests_ProductAttributeValue_Form et admirez le travail !

Tester les URLs

Tester les URLs dans un projet DJANGO permet de s'assurer que chaque URL renvoie la vue correcte avec le bon statut HTTP et que les routes fonctionnent comme prévu. Ces tests garantissent que vos utilisateurs sont dirigés vers la bonne page lorsqu'ils accèdent à une URL donnée. Commencez par créer un fichier dédié pour tester les URLs, par exemple tests_ProductAttributeValue_Urls.py dans votre répertoire tests/Url/. Comme vous l'auriez compris au regard du nom du fichier, nous allons tester les Urls liées à ProductAttributeValue.

DJANGO offre des utilitaires pour tester les URLs via la méthode reverse( ) pour obtenir une URL à partir du nom de la vue, ou directement en testant les chemins d'URL. Tester les URLs avec reverse( ) permet de s'assurer que les noms d'URLs fonctionnent correctement et renvoient la vue correcte.

```
from django.test import SimpleTestCase
from django.urls import reverse, resolve
from monapp.views import ProductAttributeValueCreateView, ProductAttributeValueListView

class ProductAttributeValueTestUrls(SimpleTestCase):

    def test_create_view_url_is_resolved(self):
    """
    Tester que l'URL de la création de ProductAttributeValue renvoie la bonne vue
    """
        url = reverse('value-add')
        self.assertEqual(resolve(url).func.view_class, ProductAttributeValueCreateView)

    def test_list_view_url_is_resolved(self):
    """
    Tester que l'URL de la liste de ProductAttributeValue renvoie la bonne vue
    """
        url = reverse('value-list')
        self.assertEqual(resolve(url).func.view_class, ProductAttributeValueListView)
```

La méthode reverse('value-add') permet de générer l'URL associée au nom value-add (celui que vous avez défini dans votre fichier urls.py). La méthode resolve(url).func.view_class permet de résoudre l'URL et de vérifier que la vue associée est correcte (ici, ProductAttributeValueCreateView).

Vous pouvez tester directement si l'URL renvoie le bon statut des réponses HTTP (par exemple, 200 OK pour une page accessible). La méthode self.client.get( ) ci-dessous envoie une requête HTTP GET à l'URL générée par reverse( ). Vous vérifiez ensuite que le code de statut est 200, indiquant que la page s'affiche correctement.

```
class ProductAttributeValueTestUrlResponses(TestCase):

    def setUp(self):
        self.user = User.objects.create_user(username='testuser', password='secret')
        self.client.login(username='testuser', password='secret')

    def test_create_view_status_code(self):
    """
    Tester que l'URL de création renvoie un statut 200 (OK)
    """
        response = self.client.get(reverse('value-add'))
        self.assertEqual(response.status_code, 200)

    def test_list_view_status_code(self):
    """
    Tester que l'URL de la liste renvoie un statut 200 (OK)
    """
        response = self.client.get(reverse('value-list'))
        self.assertEqual(response.status_code, 200)
```

Si votre vue ou URL nécessite des paramètres (comme un id par exemple), vous pouvez inclure ces paramètres dans vos tests. Testons par exemple notre vue pour afficher les détails d'un ProductAttributeValue avec un paramètre id.

```
class ProductAttributeValueTestUrlResponsesWithParameters(TestCase):

    def setUp(self):
        self.attribute = ProductAttribute.objects.create(name="Couleur")
        self.value = ProductAttributeValue.objects.create(  value="Rouge",
                                                            product_attribute=self.attribute)

    def test_detail_view_status_code(self):
    """
        Tester que l'URL des détails renvoie un statut 200 pour un ID valide
    """
        url = reverse('value-detail', args=[self.value.id])
        response = self.client.get(url)
        self.assertEqual(response.status_code, 200)

    def test_detail_view_status_code_invalid_id(self):
    """
        Tester que l'URL des détails renvoie un statut 404 pour un ID invalide
    """
        url = reverse('value-detail', args=[9999]) # ID non existant
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)
```

La méthode reverse('value-detail', args=[self.value.id]) génère une URL avec un paramètre id pour la vue de détail. Avec le paramètre args=[self.value.id], vous passez l'id de l'objet que vous voulez tester. Dans le second test, on passe un id qui n'existe pas pour s'assurer que l'URL renvoie un statut 404.

Certaines vues peuvent rediriger vers une autre page (par exemple, après une création ou une mise à jour réussie), et vous pouvez tester si ces redirections fonctionnent correctement.

```
class ProductAttributeValueTestUrlRedirect(TestCase):

    def setUp(self):
        self.user = User.objects.create_user(username='testuser', password='secret')
        self.client.login(username='testuser', password='secret')

        # Créer un attribut produit de test pour l'utiliser dans le formulaire
        self.product_attribute = ProductAttribute.objects.create(name="Couleur")

    def test_redirect_after_creation(self):
    """
    Tester qu'après la création d'un ProductAttributeValue, l'utilisateur est redirigé correctement
    """
        response = self.client.post(reverse('value-add'), {
                                                    'value': 'Bleu',
                                                    'product_attribute': self.product_attribute.id,
                                                    'position': 2 } )
        # Statut 302 = redirection
        self.assertEqual(response.status_code, 302)
        # Redirection vers la vue de detail
        self.assertRedirects(response, '/monapp/value/1')
```

Voilà nous avons fait le tour des tests d'URLs. Il n'y a pas tous les tests possibles. Ce n'est qu'un exemple. Dans un seul et même fichier, j'ai volontairement crée 4 classes différentes pour :

- tester simplement les URLs avec reverse( )
- tester le statut des réponses HTTP
- tester les URLs avec des paramètres
- tester les redirections

Exécutez vos tests avec la commande :

```
./manage.py test monapp.tests.Url.tests_ProductAttributeValue_Urls
```

Tester les vues

Pour tester vos vues DJANGO, vous pouvez écrire des tests pour vérifier que vos vues fonctionnent correctement. Il est important de tester plusieurs aspects, notamment que :

- Les vues renvoient les bons templates.
- Les requêtes HTTP (GET, POST) fonctionnent comme attendu.
- Les redirections et les erreurs sont correctement gérées.

Là encore, au sein d'un seul et unique fichier tests_ProductAttributeValue_CRUD_Views.py, nous allons tester quelques vues au travers de plusieurs classes de test. Ce fichier se trouvera dans le répertoire tests/View/ que vous allez créer.
Pour la création d'un ProductAttributeValue, vous pouvez tester la fonctionnalité GET (affichage du formulaire) et POST (enregistrement de données).

```
class ProductAttributeValueCreateViewTest(TestCase):

    def setUp(self):
        self.user = User.objects.create_user(username='testuser', password='secret')
        self.client.login(username='testuser', password='secret')
        # Créer un attribut produit de test pour l'utiliser dans le formulaire
        self.product_attribute = ProductAttribute.objects.create(name="Couleur")

    def test_create_view_get(self):
    """
    Tester que la vue de création renvoie le bon template et s'affiche correctement
    """
        response = self.client.get(reverse('value-add')) # Utilisation du nom de l'URL
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'monapp/new_value.html')

    def test_create_view_post_valid(self):
    """
    Tester que la vue de création crée un nouvel objet lorsque les données sont valides
    """
        data = { 'value': 'Violet', 'product_attribute': self.product_attribute.id, 'position': 1 }

        response = self.client.post(reverse('value-add'), data)
        # Vérifie la redirection après la création
        self.assertEqual(response.status_code, 302)
        # Vérifie qu'un objet a été créé
        self.assertEqual(ProductAttributeValue.objects.count( ), 1)
        # Vérifie la valeur de l'objet créé
        self.assertEqual(ProductAttributeValue.objects.first( ).value, 'Violet')
```

La vue de détail (DetailView) doit afficher correctement les informations d'un ProductAttributeValue

```
class ProductAttributeValueDetailViewTest(TestCase):

    def setUp(self):
        self.product_attribute = ProductAttribute.objects.create(name="Couleur")
        self.product_attribute_value = ProductAttributeValue.objects.create(
        value='Violet', product_attribute=self.product_attribute, position=1)

    def test_detail_view(self):
    """
    Tester que la vue de détail renvoie le bon template et affiche les bonnes données
    """
        response = self.client.get(reverse('value-detail', args=[self.product_attribute_value.id]))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'monapp/detail_value.html')

        # Vérifie que le nom de la valeur est affiché
        self.assertContains(response, 'Violet')

        # Vérifie que l'attribut associé est affiché
        self.assertContains(response, 'Couleur')
```

Pour les tests de mise à jour, vous devez vérifier à la fois le chargement du formulaire (GET) et la soumission des données mises à jour (POST).

```
class ProductAttributeValueUpdateViewTest(TestCase):

    def setUp(self):
        self.user = User.objects.create_user(username='testuser', password='secret')
        self.client.login(username='testuser', password='secret')

        self.product_attribute = ProductAttribute.objects.create(name="Couleur")
        self.product_attribute_value = ProductAttributeValue.objects.create(
            value='Violet', product_attribute=self.product_attribute, position=1)

    def test_update_view_get(self):
    """
    Tester que la vue de mise à jour s'affiche correctement
    """
        response = self.client.get(reverse('value-update', args=[self.product_attribute_value.id]))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'monapp/update_value.html')

    def test_update_view_post_valid(self):
    """
    Tester que la vue met à jour l'objet lorsque les données sont valides
    """
        data = { 'value': 'Jaune', 'product_attribute': self.product_attribute.id, 'position': 2 }

        response = self.client.post(reverse('value-update',
            args=[self.product_attribute_value.id]), data)

        # Redirection après la mise à jour
            self.assertEqual(response.status_code, 302)

        # Recharger l'objet depuis la base de données
        self.product_attribute_value.refresh_from_db()

        # Vérifier la mise à jour
        self.assertEqual(self.product_attribute_value.value, 'Jaune')

        # Vérifier la mise à jour de la position
        self.assertEqual(self.product_attribute_value.position, 2)
```

La vue de suppression doit permettre de supprimer un ProductAttributeValue et rediriger l'utilisateur après la suppression.

```
class ProductAttributeValueDeleteViewTest(TestCase):

    def setUp(self):
        self.user = User.objects.create_user(username='testuser', password='secret')
        self.client.login(username='testuser', password='secret')

        self.product_attribute = ProductAttribute.objects.create(name="Couleur")
        self.product_attribute_value = ProductAttributeValue.objects.create(
            value='Rouge', product_attribute=self.product_attribute, position=1)

    def test_delete_view_get(self):
    """
    Tester que la vue de suppression s'affiche correctement
    """
        response = self.client.get(reverse('value-delete', args=[self.product_attribute_value.id]))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'monapp/delete_value.html')

    def test_delete_view_post(self):
    """
    Tester que l'objet est supprimé lorsque le formulaire de suppression est soumis
    """
        response = self.client.post(reverse('value-delete', args=[self.product_attribute_value.id]))

        # Redirection après suppression
        self.assertEqual(response.status_code, 302)

        # Vérifier que l'objet est supprimé
        self.assertEqual(ProductAttributeValue.objects.count(), 0)
```

Exécutez vos tests avec la commande :

```
./manage.py test monapp.tests.View.tests_ProductAttributeValue_CRUD_Views
```

Il manque ici les tests pour la vue ProductAttributeValueListView mais je suis sûr que vous êtes capable de le faire sans moi.

Taux de couverture de test

Le taux de couverture de test (ou test coverage) est un indicateur qui mesure le pourcentage de votre code source qui est exécuté lors de l'exécution de vos tests. En d'autres termes, il vous permet de savoir dans quelle mesure votre code est testé, ce qui peut aider à identifier les parties du code qui ne sont pas couvertes par des tests.

Dans DJANGO (et plus largement en Python), l'outil le plus utilisé pour mesurer la couverture de test est coverage.py. Il permet de suivre l'exécution des tests et de produire des rapports sur le code exécuté. Si vous ne l'avez pas encore, vous devez installer coverage avec pip : `pip install coverage`

Pour exécuter vos tests tout en mesurant la couverture, utilisez la commande suivante :

```
coverage run --source='monapp' manage.py test
```

Le paramètre `--source='monapp'` spécifie le répertoire de votre application DJANGO à couvrir. La commande `manage.py test` lance vos tests unitaires DJANGO comme d'habitude.

Une fois les tests exécutés, vous pouvez générer un rapport de couverture simple dans le terminal avec la commande suivante : `coverage report`
Cela affichera un rapport dans le terminal, vous indiquant le pourcentage de lignes de code couvertes par les tests.

```
Name                                                                    Stmts   Miss  Cover
-------------------------------------------------------------------------------------------
monapp/__init__.py                                                          0      0   100%
monapp/admin.py                                                            38      8    79%
monapp/apps.py                                                              4      0   100%
monapp/forms.py                                                            22      0   100%
monapp/migrations/0001_initial.py                                           6      0   100%
monapp/migrations/__init__.py                                               0      0   100%
monapp/models.py                                                           45      3    93%
monapp/tests/Form/__init__.py                                               0      0   100%
monapp/tests/Form/tests_ProductAttributeValue_Form.py                      27      0   100%
monapp/tests/Model/__init__.py                                              0      0   100%
monapp/tests/Model/tests_ProductAttributeValue_Model.py                    20      0   100%
monapp/tests/Url/__init__.py                                                0      0   100%
monapp/tests/Url/tests_ProductAttributeValue_Urls.py                       44      0   100%
monapp/tests/View/__init__.py                                               0      0   100%
monapp/tests/View/tests_ProductAttributeValue_CRUD_Views.py                61      0   100%
monapp/tests/__init__.py                                                    0      0   100%
monapp/urls.py                                                              5      0   100%
monapp/views.py                                                           252     87    65%
-------------------------------------------------------------------------------------------
TOTAL                                                                     524     98    81%
```

Pour obtenir un rapport plus détaillé et visuellement organisé, vous pouvez générer un rapport HTML avec la commande suivante : coverage html
Cette commande créera un dossier htmlcov/ contenant un fichier HTML index.html. Ouvrez-le dans un navigateur pour voir le rapport sous forme de tableau, indiquant quelles parties de votre code sont couvertes et lesquelles ne le sont pas.

Coverage report: 81%

coverage.py v7.4.1, created at 2024-09-24 08:46 +0200

Module	statements	missing	excluded	coverage
monapp/__init__.py	0	0	0	100%
monapp/admin.py	38	8	0	79%
monapp/apps.py	4	0	0	100%
monapp/forms.py	22	0	0	100%
monapp/migrations/0001_initial.py	6	0	0	100%
monapp/migrations/__init__.py	0	0	0	100%
monapp/models.py	45	3	0	93%
monapp/tests/Form/__init__.py	0	0	0	100%
monapp/tests/Form/tests_ProductAttributeValue_Form.py	27	0	0	100%
monapp/tests/Model/__init__.py	0	0	0	100%
monapp/tests/Model/tests_ProductAttributeValue_Model.py	20	0	0	100%
monapp/tests/Url/__init__.py	0	0	0	100%
monapp/tests/Url/tests_ProductAttributeValue_Urls.py	44	0	0	100%
monapp/tests/View/__init__.py	0	0	0	100%
monapp/tests/View/tests_ProductAttributeValue_CRUD_Views.py	61	0	0	100%
monapp/tests/__init__.py	0	0	0	100%
monapp/urls.py	5	0	0	100%
monapp/views.py	252	87	0	65%
Total	**524**	**98**	**0**	**81%**

coverage.py v7.4.1, created at 2024-09-24 08:46 +0200

100% de couverture signifie que chaque ligne de code est exécutée au moins une fois lors de l'exécution des tests, mais cela ne garantit pas que tous les cas d'usage et toutes les conditions sont bien testés. Un taux de couverture inférieur à 100% indique qu'il y a des parties de votre code qui ne sont jamais exécutées par vos tests. Cela pourrait révéler des branches de code non testées, comme des blocs if, else, ou des exceptions.

Dans notre cas, le taux de couverture global est de 81%, et vous pouvez voir les fichiers qui ne sont pas complètement couverts. 80%-90% de couverture est souvent considéré comme un bon objectif. 100% de couverture n'est pas toujours nécessaire et peut être difficile à atteindre dans certains cas. L'important est de s'assurer que les fonctionnalités critiques et la logique métier sont bien couvertes.

La couverture ne garantit pas la qualité : Un code couvert à 100% peut toujours contenir des bugs. La couverture ne teste pas la justesse des résultats. Même avec une couverture élevée, il est essentiel de tester des scénarios pertinents, y compris des cas limites, des erreurs et des exceptions. Le taux de couverture est un bon indicateur pour savoir quelles parties du code sont testées, mais il ne remplace pas l'importance de créer des tests bien conçus et pertinents. Une couverture élevée aide à maintenir la qualité et la robustesse de votre application, mais assurez-vous que vos tests vérifient réellement les comportements attendus.

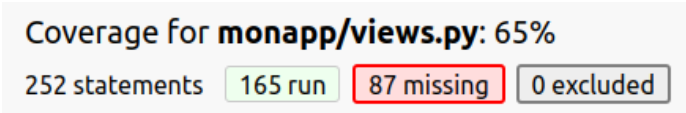
Pour exclure certains fichiers ou répertoires du calcul du taux de couverture lors de l'utilisation de coverage.py, vous pouvez configurer l'outil pour ignorer certains fichiers ou parties du code. Vous pouvez créer un fichier de configuration .coveragerc à la racine de votre projet pour y inclure vos règles d'exclusion. Cela vous permet de centraliser les paramètres de coverage.

.coveragerc
```
[run]
source = monapp
omit =
    */migrations/*
    */settings/*
    */wsgi.py
    */asgi.py
    *__init__.py

[report]
# Pour afficher uniquement les fichiers qui ne sont pas à 100%
show_missing = True

[html]
directory = htmlcov
```

Si vous relancer, vous obtiendrez directement ceci :

```
Name                                                                    Stmts   Miss  Cover
-------------------------------------------------------------------------------------------
monapp/admin.py                                                            38      8    79%
monapp/apps.py                                                              4      0   100%
monapp/forms.py                                                            22      0   100%
monapp/models.py                                                           45      3    93%
monapp/tests/Form/tests_ProductAttributeValue_Form.py                      27      0   100%
monapp/tests/Model/tests_ProductAttributeValue_Model.py                    20      0   100%
monapp/tests/Url/tests_ProductAttributeValue_Urls.py                       44      0   100%
monapp/tests/View/tests_ProductAttributeValue_CRUD_Views.py                61      0   100%
monapp/urls.py                                                              5      0   100%
monapp/views.py                                                           252     87    65%
```

Dans notre cas, l'effort doit être fait sur le fichier views.py dont le taux de couverture est de 65%. Dans le rapport HTML, si vous cliquez sur le lien monapp/views.py, vous allez apercevoir les endroits qui ne sont pas testés (en rouge) :



Grosso modo, les vues de création, mise à jour, suppression et d'affichage des autres éléments de notre modèle. C'est à vous de jouer !! Vous pouvez créer autant de fichiers de test que vous voulez ! Il suffit de bien les nommer et de bien les ranger dans votre arborescence. N'oubliez le modèle, le formulaire et les urls en plus de vues manquantes pour Product, ProductItem et ProductAttribute...