

# Django - TD 3

---

## Les tests

Avant de continuer, nous allons procéder à la rédaction de tests.

Si vous vous posez la question de ce qu'est des tests automatisés, quelle est leur utilité ou encore quelle stratégie mettre en oeuvre pour élaborer les tests. Alors commencez par lire l'introduction de [ce tutoriel](#).

## Tests du modèles

Avant de commencer les tests, nous allons ajouter un brin de matière à tester.

Dans `developer.models.py`, ajoutez une méthode qui permet de vérifier si un développeur est libre de toute tâche.

`developer.models.py`

–

```
class Developer(models.Model):
    first_name = models.CharField("first name", max_length=200)
    last_name = models.CharField(max_length=200)

    def is_free(self):
        return self.tasks.count() == 0

    def __str__(self):
        return f"{self.first_name} {self.last_name}"
```

Et maintenant, lançons nous dans les tests.

Un endroit conventionnel pour placer les tests d'une application est le fichier `tests.py` dans le répertoire de l'application. Cependant, le système de test va automatiquement trouver les tests dans tout fichier dont le nom commence par `test`.

Placez ce qui suit dans une classe `DeveloperModelTests` qui hérite de la classe `TestCase` déjà importée dans le fichier `tests.py` de l'application `developer`:

```
def test_is_free_with_no_tasks(self):
    """
    is_free() returns True for developer with no
    tasks.
    """

    dev = Developer.objects.create(first_name="Sébastien",
last_name="Drobisz")
    self.assertIs(dev.is_free(), True)

def test_is_free_with_one_tasks(self):
    """
    is_free() returns False for developer with at least one
    tasks.
    """

    dev = Developer.objects.create(first_name="Sébastien",
last_name="Drobisz")
    dev.tasks.create(title="cours Django", description="Faire
le cours sur Django")
    self.assertIs(dev.is_free(), False)
```

Nous venons ici de créer une sous-classe de `django.test.TestCase` contenant

- une première méthode qui crée une instance `Developer` avec des données quelconques. Nous vérifions ensuite le résultat de `is_free()` qui devrait valoir `True`.
- Une seconde méthode qui crée une même instance de `Developer`. Nous lui assignons cette fois-ci la tâche d'écrire le cours sur Django. Enfin, nous vérifions le résultat de la méthode `is_free()` qui devrait cette fois-ci valoir `False`.

Lançons les tests

```
$python manage.py test developer
```

Voici ce qui s'est passé :

- La commande `manage.py test developer` a cherché des tests dans l'application `developer` ;
- elle a trouvé une sous-classe de `django.test.TestCase` ;
- elle a créé une base de données spéciale uniquement pour les tests ⚠ ;
- elle a recherché des méthodes de test, celles dont le nom commence par `test` ;
- dans `test_is_free_with_no_tasks`, elle a créé une instance de `Developer` ;
- et à l'aide de la méthode `assertIs()`, elle a pu vérifier son bon fonctionnement.

Si le test avait échoué, (vous pouvez essayer), le test nous indique alors le nom du test qui a échoué ainsi que la ligne à laquelle l'échec s'est produit.

## "Before"

Vous pouvez initialiser certains éléments avant de réaliser les tests. Cela évite de réaliser plusieurs fois la même instantiation.

Cela se fait grâce à la méthode `setUp()`.

```
def setUp(self):
    Developer.objects.create(first_name="Sébastien",
last_name="Drobisz")

    #...
    #dev = Developer.objects.create(first_name="Sébastien",
last_name="Drobisz")
    dev = Developer.objects.get(first_name="Sébastien")
    self.assertIs(dev.is_free(), True)

    #...
    #dev = Developer.objects.create(first_name="Sébastien",
last_name="Drobisz")
    dev = Developer.objects.get(first_name="Sébastien")
```

Pour plus d'informations sur la configuration des tests, vous pouvez lire [ce lien](#).

## Tests de la vue

Django fournit un Client de test pour simuler l'interaction d'un utilisateur avec le code au niveau des vues. On peut l'utiliser dans `tests.py` ou même dans le shell.


## Tests de la vue dans le shell

Nous commencerons encore une fois par le shell, **où nous devons faire quelques opérations qui ne seront pas nécessaires dans `tests.py`**. La première est de configurer l'environnement de test dans le shell:


### Mise en place de l'environnement de test

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

`setup_test_environment()` installe un moteur de rendu de gabarit qui va nous permettre d'examiner certains attributs supplémentaires des réponses, tels que `response.context` qui n'est normalement pas disponible. Notez que cette méthode ne crée pas de base de données de test, ce qui signifie que ce qui suit va être appliqué à la base de données existante et que par conséquent, le résultat peut légèrement différer en fonction des développeurs que vous avez déjà créés.

 Si vous obtenez une erreur étrange du style "Invalid HTTP\_HOST header: 'testserver'. You may need to add 'testserver' to ALLOWED\_HOSTS.". Alors vous avez probablement oublié la mise en place de l'environnement de test

### Import d'un client de test

Ensuite, il est nécessaire d'importer la classe Client de test (  plus loin dans `tests.py`, nous utiliserons la classe `django.test.TestCase` qui apporte son propre client, ce qui évitera cette étape)

```
>>> from django.test import Client
>>> client = Client()
```

Ceci fait, nous pouvons demander au client de faire certaines tâches pour nous :

```

>>> response = client.get('/')
Not Found: /
# Si on y regarde de plus près...
>>> response.status_code
404 # la page n'a pas été trouvée.
from django.urls import reverse
>>> response = client.get(reverse('developer:index'))
>>> response.status_code
200
>>> response.content
#Le code Html de la page est affiché...
>>> response.context ['developers']
<QuerySet [<Developer:.....>]

```

## Tests automatiques de la vue

### Tests de IndexView

```

class DeveloperIndexViewTests(TestCase):
    def test_no_developers(self):
        """
        If no developers exist, an appropriate message is displayed.
        """
        response = self.client.get(reverse('developer:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "Il n'y a aucune développeur
enregistré !")
        self.assertQuerysetEqual(response.context['developers'], [])

```

Dans le premier test, nous vérifions que la page existe bel et bien, qu'un message indiquant l'absence de développeur est affiché et que la variable `developer` du context est vide.



Les tests sont fait dans un ordre logique permettant de déterminer directement la source de l'erreur !

Dans le second test, nous vérifions que le prénom du développeur est bien affiché.

```

def test_one_developer(self):
    """
    A developer is displayed on the index page.
    """
    dev = Developer.objects.create(
        first_name="Jonathan",
        last_name="Lechien")
    response = self.client.get(reverse('developer:index'))
    self.assertEqual(response.status_code, 200)
    self.assertQuerysetEqual(response.context['developers'],
        [f'<Developer: {dev.first_name} {dev.last_name}>'])
    self.assertContains(response, dev.first_name)

```

## Tests de DevDetailView()

Nous allons faire deux tests afin de vérifier la vue d'un développeur.

1. Un premier qui permet de vérifier que le nom et le prénom d'un développeur est bien affiché
2. Un deuxième qui permet de vérifier qu'une page 404 est proposée si le développeur recherché n'existe pas.

```

class DevDetailView(TestCase):
    def test_existing_developer(self):
        """
        The detail view of a developer displays the developer's
        text.
        """
        dev = Developer.objects.create(
            first_name="Jonathan",
            last_name="Lechien")
        url = reverse('developer:detail', args=(dev.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.context['developer'], dev)
        self.assertContains(response, dev.first_name)
        self.assertContains(response, dev.last_name)

    def test_non_existing_developer(self):
        """
        The detail view of a non existing developer should return
        404 status_code response.
        """

```

```
url = reverse('developer:detail', args=(1,))
response = self.client.get(url)
self.assertEqual(response.status_code, 404)
```

## Postgresql

Et si nous changions le SGBD afin d'utiliser Postgresql ?

Vous n'avez pas grand chose à faire.

1. Installer `postgresql`
2. Créer une DB (par exemple `mproject`): `CREATE DATABASE mproject;`
3. Créer un rôle (dans `psql`): `create role <le rôle> login password '<le mdp>';`
4. Donner les droits nécessaires à ce nouveau rôle (dans `psql`): `grant all privileges on database mproject to <le rôle>;`
5. Installer `psycopg2`: `python -m pip install django psycopg2.`
6. Configurer l'utilisation de la db dans `settings.py`

```
#'default': {
#    'ENGINE': 'django.db.backends.sqlite3',
#    'NAME': BASE_DIR / 'db.sqlite3',
#},
'default': {
    'ENGINE': 'django.db.backends.postgresql_psycopg2',
    'NAME': 'mproject',
    'USER': '<le rôle>',
    'PASSWORD': '<le mdp>',
    'HOST': 'localhost',
    'PORT': '',
}
```

Et voilà, tout est fait !

Enfin, n'oubliez pas de migrer 😊 ( ★quelle commande allez vous utiliser pour réaliser la migration ? )

# Migrations

Lorsque nous sommes au début du développement d'un logiciel, nous pouvons échapper aux problèmes liés aux migrations. Nous allons cependant voir le minimum afin de gérer les quelques conflits que nous pourrions rencontrer. Si la gestion des migrations vous intéresse, vous trouverez de quoi satisfaire votre curiosité [ici](#).

Imaginons que nous souhaitons ajouter un nouveau champ `username` à notre modèle `Developer`.

`developer/models.py`

```
class Developer(models.Model):
    first_name = models.CharField("first name", max_length=200)
    last_name = models.CharField(max_length=200)
    user_name = models.CharField(max_length=50) ➡ new
    #...
```

Saisissez la commande `python manage.py makemigrations` qui va vérifier les changements et générer un nouveau fichier permettant la migration.

Vous devriez avoir ce message :

```
You are trying to add a non-nullable field 'user_name' to developer
without a default; we can't do that (the database needs something
to populate existing rows).
Please select a fix:
  1) Provide a one-off default now (will be set on all existing rows
with a null value for this column)
  2) Quit, and let me add a default in models.py
Select an option:
```

La raison est simple. Des données sont potentiellement présentes dans la db et nous ne pouvons pas supposer qu'il n'y en a pas (imaginez s'il y a plusieurs instances du site). Django vous demande donc ce que vous voulez faire pour le champs `username` puisque celui-ci s'ajoute aux enregistrements présents et que celui-ci est obligatoire (si si puisque nous n'avons pas dit le contraire).


La procédure que nous allons vous soumettre est un peu radicale, mais nous sommes aux prémices du développement de notre projet. C'est donc satisfaisant comme cela !

## 1. Réinitialiser la db.



- Si vous utilisez toujours sqlite, alors supprimez le fichier `db.sqlite3` (ou mieux, faite la configuration nécessaire à l'utilisation de Postgresql).
  - Si vous utilisez postgresql comme demandé, nous allons plutôt défaire toutes les migrations réalisées. Lancez la commande : `$ python manage.py migrate developer zero`.
2. Supprimez les fichiers présents dans le dossier `migrations`. Ceux-ci ont généralement la forme : `0001_...`
  3. Relancez la procédure complète de migration
    - a. `python manage.py makemigrations`
    - b. `python manage.py migrate`

Pour vous entraîner, supprimez le champs `user_name` que vous venez de créer, cela va nous gêner par la suite et cela vous permet de vous entraîner avec la procédure ! ★

 Certains diront qu'il est également possible de supprimer la db et de la recréer. Ceux-ci n'ont pas tort, mais pensez à supprimer les fichiers de migration malgré tout !

## Préparons la suite

### Enjolivons la page d'index des développeurs

#### Cas aucun développeur

En principe, si vous vous êtes juste contenté de suivre le tutoriel jusqu'à maintenant, vous devriez avoir le message `Il n'y a aucun développeur enregistré` d'affiché.

Nous allons le mettre légèrement en forme.

`developer/index.html`

```
<div class="container m-4">
  <alert class="alert alert-warning">Il n'y a aucun développeur
  enregistré</alert>
</div>
```

Vérifiez le résultat.

## Cas au moins 1 développeur

Maintenant ajoutez au moins 2 développeurs.

Modifiez le rendu de ceux-ci.

`developer.index.html`

```
{% if developers %}
    <div class="container-sm 1-3 d-flex flex-wrap border">
        {% for dev in developers %}
            <div class="card bg-primary m-2 p-1 rounded-lg"
style="width:300px">
                <div class="card-title">
                    {{ dev.first_name }} {{ dev.last_name }}
                </div>
                <div class="card-body">
                    {{ dev.tasks.all|length }}
tâche{{developer.tasks.all|length|pluralize}}
                </div>
                <div class="card-footer">
                    <a href="{% url 'developer:detail' dev.id %}"
class="btn btn-outline-light">Détails</a>
                </div>
            </div>
        {% endfor %}
    </div>
{% else %}
```

Remarquez la ligne

```
{{ dev.tasks.all|length }} tâche{{dev.tasks.all|length|pluralize}}
```

Celle-ci permet d'afficher le nombre de tâches (`{{ dev.tasks.all|length }}`) et d'ajouter un 's' à `tâche` s'il y en a plusieurs ou 0 (`tâche{{developer.tasks.all|length|pluralize}}`).

## Enjolivons la page détails des développeurs

developer/detail.html

```
{% block content %}
  <div class="jumbotron" style="height:150px">
    <h1>{{ developer.first_name }} {{ developer.last_name }}
  </h1>
    <p>{{ developer.tasks.all|length }}
    tâche{{developer.tasks.all|length|pluralize}}
    assignée{{developer.tasks.all|length|pluralize}}.</p>
  </div>
{% endblock content %}
```

C'est terminé, ajoutez une tâche à un développeur afin de vérifier que le nombre de tâches assignées est bien amandé.

Profitons-en pour afficher les tâches d'un développeur dans la vue détail.

developer/detail.html

```
<div class="container-sm">
  {% if not developer.is_free %}
    <ul class="list-group">
      {% for task in developer.tasks.all %}
        <li class="list-group-item">
          <strong>{{ task.title }}!</strong> {{
task.description }}
        </li>
      {% endfor %}
    </ul>
  {% else %}
    <div class="alert alert-danger">
      Aucune tâche n'est assignée à {{
developer.first_name }}.
    </div>
  {% endif %}
</div>
{% endblock content %}
```

# Et si on supprimait les développeurs ?

## ★ Exercice

Ajoutez le code ci-dessous

`developer/detail.html`

```
<div class="jumbotron" style="height:150px">
  <form action="{% url 'developer:delete' developer.id %}"
  method="POST">
    {% csrf_token %}
    <button type="submit" class="close"><i class="fa fa-trash"></i></button>
  </form>
  <h1>{{ developer.first_name }} {{ developer.last_name }}</h1>
  <p>{{ developer.tasks.all|length }}
  tâche{{developer.tasks.all|length|pluralize}}
  assignée{{developer.tasks.all|length|pluralize}}.</p>
</div>
```

Ce code a pour objectif d'ajouter une petite corbeille près du nom d'un développeur afin de permettre sa suppression.

1. Inspirez vous des vues existantes afin de permettre la suppression d'un développeur. Lors de la suppression, redirigez vers l'index des développeurs.
2. Ajoutez l'url adéquate
3. Testez

Que se passe-t-il pour les tâches qui étaient assignées ? Amendez le code afin qu'une tâche assignée à un développeur ne soit plus supprimée.

## Gestion des tâches

Nous allons maintenant ajouter la gestion des tâches. Attention, nous allons installer les bases ensemble, vous ferez le reste seul.

## App Task

Actuellement, les tâches (Task) sont dans le modèle `developer`. Nous avons choisi cela afin de rentrer plus rapidement dans la matière.

★ Commencez par créer et ajouter une nouvelle application `task` et ensuite, déplacez le modèle de `Task` dans cette nouvelle application.

## Ajout d'une nouvelle vue index

### Ajout de la liste des tâches

★ Inspirez vous de ce qui a déjà été fait pour ajouter une nouvelle vue qui permet uniquement d'afficher la liste de toutes les tâches. Si un développeur est assigné à une tâche, son nom doit apparaître à côté. Sinon, il doit être indiqué qu'elle n'est pas assignée.

### Activer le bon lien

Si vous avez prêté attention, le lien activé dans le menu de navigation reste celui des développeurs. C'est normal puisque nous ne l'avons pas changé.

Pour faire cela, nous allons utiliser un gabarit de base pour les applications, mais avant, ajoutons un bloc au gabarit de base du projet.

```
BASE_DIR/templates/_base.html
```

```
#...
{% block content %}
{% endblock content %}

<script>
    {% block menu-script %}
        $("#nav-home").addClass('active')
        $("#nav-dev").removeClass('active')
        $("#nav-task").removeClass('active')
    {% endblock menu-script %}
</script>
</body>
```

Dans ce bloc, nous ajoutons un peu de JQuery nous permettant d'activer le bon lien. Par défaut, c'est naturellement le lien home qui doit être activé.

Dans le dossier `templates/task` ajoutez maintenant un nouveau gabarit nommé `_base.html`

Celui-ci doit étendre le template du projet et activer le bon lien.

```
tempaltes/task/_base.html
```

```
{% extends "_base.html" %}

{% block title %}GProject - Gestion des tâches{% endblock
title %}

{% block menu-script %}
    $("#nav-home").removeClass('active')
    $("#nav-dev").removeClass('active')
    $("#nav-task").addClass('active')
{% endblock menu-script %}
```

Et enfin, le gabarit `templates/task/index.html` doit hériter de ce nouveau template.

Attention, puisqu'il y a

maintenant deux gabarits nommé `_base.html`, il faut bien indiquer l'application.

```
tempaltes/task/index.html
```

```
{% extends "_base.html" %}      ➡old
{% extends "task/_base.html" %} ➡new
```

Voilà, c'est terminé, mais dans ce processus, vous pourrez remarquer que le lien ne s'active plus lorsque nous cliquons sur l'onglet `developers`. C'est normal... Il faut également ajoutez un gabarit de base à cette application... Faites-le ! N'oubliez pas qu'il y a deux vues à adapter! ★

## Suppression d'une tâche

Modifiez le code `templates/task/index.html` en ajoutant ces quelques lignes :

```
tempaltes/task/index.html
```

```
#...
<li class="list-group-item">
  <form action="{% url 'task:delete' task.id %}" method="post">
    {% csrf_token %}
    <button class="close" type="submit"><i class="fa fa-
trash"></i></button>
  </form>
#...
```


Ce bout de code permet d'ajouter une corbeille pour supprimer une tâche.

★ Ajoutez le code nécessaire afin d'ajouter la fonctionnalité de suppression de tâches.

## Création d'une tâche

### Dans la page d'index des tâches

★ On vous donne un petit coup de pouce et le reste est dans vos mains.

 Précédemment, vous avez créé un formulaire pour la création de développeur. Vous allez devoir faire la même chose ici. Le champ `assignee` pourra vous poser problème. Le voici

```
assignee =
forms.ModelChoiceField(queryset=Developer.objects.all(),
required=False)
```

Vous trouverez davantage de doc sur le `ModelChoiceField` [ici](#).

Si vous utilisez l'héritage de `FormModel`, ce sera encore plus facile.

### Dans le détail d'un développeur

★ Ajoutez la possibilité de créer une tâche dans la vue détail d'un développeur. Lorsqu'une tâche sera créé, l'utilisateur sera redirigé vers l'index des tâches. Ce n'est pas optimal, mais nous ferons avec.

- Il serait agréable que le formulaire soit pré-rempli au niveau du développeur assigné. Lisez la documentation des formulaires (paramètre `initial`).

- Il serait aussi bien de ne pas exposer l'utilisateur à une erreur possible. Désactivez le champ pour que celui-ci ne soit pas modifiable. Attention, un champ désactivé n'est pas envoyé dans les données `POST`.