

Django

Avant-propos

Ce document a été finalisé tard dans la nuit et comporte probablement de nombreuses coquilles. Nous vous recommandons de postposer l'impression de celui-ci.

Objectifs et sources

Les objectifs de ces premiers laboratoires sont multiples.

- Apprendre les fondements de Python :
 - Différentes structures élémentaires ;
 - Les classes ;
 - Les modules...
- Découvrir un framework web Python qui partage de nombreuses similitudes avec la suite logiciel Odoo dont vous apprendrez à créer une application.

Ce cours est basé sur les sources suivantes :

- Le [tutoriel Django](#)
- La [documentation Django](#)
- [Django for professionals](#)

Installation

Pour procéder à l'installation, nous vous conseillons

- d'installer `python 3.8` ;
- installer `pip` qui est un gestionnaire de package pour `python` ;
- et d'ensuite lancer l'installation de Django avec la commande `python -m pip install Django`.

Optionnellement, vous pouvez jeter un oeil à [venv](#) ; ou encore à [Docker](#).

Légende

Il s'agit d'un cours avant tout, parfois vous verrez différents emoji. Ceux-ci permettent d'attirer votre attention de différentes manières.

Emoji	Signification
🐍	Parenthèse Python
☕	Lien Java
★	Il s'agit d'un petit exercice
🐇	Suivez moi les yeux fermés !
👉 new	Indique l'ajout d'un morceau de code
⚠️	Source d'erreur
😎	Lecture supplémentaire optionnelle, mais vivement recommandé
📄	Note un peu plus personnelle

Introduction

Dans ce cours, vous allez apprendre à créer un site web permettant de gérer les tâches de différents développeurs.

La première chose à faire est de créer un projet Django.

```
$ django-admin startproject mproject
```

📄 Sur les ordinateurs de l'esi, utilisez la commande `py -m django startproject mproject`.

Cette commande a créé un répertoire dans votre répertoire courant. Dans ce dossier, se trouve les fichiers suivants :

```
mproject/  
  manage.py  
  mproject/  
    __init__.py  
    settings.py  
    urls.py  
    asgi.py  
    wsgi.py
```

Voici quelques explications :

- Le premier répertoire racine `mproject/` est un contenant pour votre projet. Son nom n'a pas d'importance pour Django ; vous pouvez le renommer comme vous voulez.
- `manage.py` : un utilitaire en ligne de commande qui vous permet d'interagir avec ce projet Django de différentes façons. Vous trouverez toutes les informations nécessaires sur `manage.py` dans `django-admin` et `manage.py`.
- **Le sous-répertoire `mproject/`** correspond au paquet Python effectif de votre projet. C'est le nom du paquet Python que vous devrez utiliser pour importer ce qu'il contient (par ex. : `mproject.urls`).
- `mproject/__init__.py` : un fichier vide qui indique à Python que ce répertoire doit être considéré comme un paquet. Si vous êtes débutant en Python, lisez les informations sur les paquets (en) dans la documentation officielle de Python.
- `mproject/settings.py` : réglages et configuration de ce projet Django. Ce cours de Django vous apprendra tout sur le fonctionnement des réglages (enfin presque tout).

- **mproject/urls.py** : les déclarations des URL de ce projet Django, une sorte de « table des matières » de votre site Django. Vous pouvez en lire plus sur les URL dans Distribution des URL.
- **mproject/asgi.py** : un point d'entrée pour les serveurs Web compatibles aSGI pour déployer votre projet. Voir [Comment déployer avec ASGI pour plus de détails](#).
- **mproject/wsgi.py** : un point d'entrée pour les serveurs Web compatibles WSGI pour déployer votre projet. Voir [Comment déployer avec WSGI pour plus de détails](#).

Serveur de développement

Django est fourni avec un serveur de développement qui permet de simplifier le développement avant la mise en production. Parmi ces simplifications, on peut noter que le serveur sert les fichiers statiques.

Vous pouvez le tester en lançant la commande

```
$ python manage.py runserver
```

Vous devriez alors voir le message suivant :

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are
applied.
Run 'python manage.py migrate' to apply them.

septembre 02, 2020 - 15:50:53
Django version 3.1, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Note : Vous pouvez ignorer les avertissements liés à la migration de la base de donnée.

Maintenant que le serveur tourne, allez à l'adresse <http://127.0.0.1:8000> avec votre navigateur Web. Vous verrez une page avec le message « Félicitations ! » ainsi qu'une fusée qui décolle. Ça marche !

Recharge automatique du serveur

Le serveur de développement recharge automatiquement le code Python lors de chaque requête si nécessaire. Vous ne devez pas redémarrer le serveur pour que les changements de code soient pris en compte. Cependant, certaines actions comme l'ajout de fichiers ne provoquent pas de redémarrage, il est donc nécessaire de redémarrer manuellement le serveur dans ces cas.

Ma première vue Django

Avant d'écrire une vue à proprement parler, nous allons la faire au sein d'une nouvelle application intitulée **developer**.

Création de l'application developer

Quelle est la différence entre un projet et une application ? Une application est une application Web qui fait quelque chose – par exemple un système de blog, une base de données publique ou une petite application de sondage. Un projet est un ensemble de réglages et d'applications pour un site Web particulier. Un projet peut contenir plusieurs applications. Une application peut apparaître dans plusieurs projets.

Pour créer votre application, assurez-vous d'être dans le même répertoire que `manage.py` et saisissez la commande :

```
$ python manage.py startapp developer
```

Cela va créer un répertoire `developer`, qui est structuré de la façon suivante :

```
developer/
  __init__.py
  admin.py
  apps.py
  migrations/
    __init__.py
  models.py
  tests.py
  views.py
```

Écriture d'une première vue

Écrivons la première vue. Ouvrez le fichier `developer/views.py` et placez-y le code Python suivant :

```
developer/views.py
```

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You're at the developers index.")
```

Parenthèse python 🐍

Dans cette vue, nous découvrons plusieurs éléments python.

- "`from django.http import HttpResponse`" permet d'importer la classe `HttpResponse` du module `django.http`.
- "`def index(request):`" permet de définir une fonction en python. Dans le jargon Django, nous appellerons cela *une fonction de vue*.

C'est la vue Django la plus simple possible. Pour appeler cette vue, il s'agit de l'associer à une URL, et pour cela nous avons besoin d'un *URLconf*.

Pour créer un *URLconf* dans le répertoire `developer`, créez un fichier nommé `urls.py`. Votre répertoire d'application devrait maintenant ressembler à ceci :

```
developer/  
  __init__.py  
  admin.py  
  apps.py  
  migrations/  
    __init__.py  
  models.py  
  tests.py  
  urls.py  
  views.py
```

Dans le fichier `developer/urls.py`, insérez le code suivant :

`developer/urls.py`

```
from django.urls import path  
  
from . import views  
  
urlpatterns = [  
    path('', views.index, name='index'),  
]
```

Parenthèse python 🧐

1. Nous importons la fonction `path` du module `django.url`.
2. Nous importons les éléments de notre répertoire `views` (c'est-à-dire notre fonction vue `index` écrite précédemment).
3. Nous assignons à la variable `urlpatterns` une `liste` de chemin (ici un seul chemin).
Notez qu'en Python, il est de bonne pratique de toujours terminer par une virgule, même si la liste n'est constitué que d'un seul élément.

L'étape suivante est de faire pointer la configuration d'URL racine vers le module `developer.urls`. Dans `mproject/urls.py`, ajoutez une importation `django.urls.include` et insérez un appel à `include()` dans la liste `urlpatterns`, ce qui donnera :

`mproject/urls.py`

```
from django.contrib import admin  
from django.urls import include, path  
  
urlpatterns = [  
    path('developer/', include('developer.urls')),  
    path('admin/', admin.site.urls),  
]
```

L'idée derrière `include()` est de faciliter la connexion d'URL. Comme l'application de développeur possède son propre `URLconf` (`developer/urls.py`), ses URL peuvent être injectées sous « `/developer/` », sous « `/dev/` » ou sous « `/content/dev/` » ou tout autre chemin racine sans que cela change quoi que ce soit au fonctionnement de l'application.

Vous avez maintenant relié une vue index dans la configuration d'URL. Vérifiez qu'elle fonctionne avec la commande suivante :

```
$ python manage.py runserver
```

Ouvrez <http://localhost:8000/developer/> dans votre navigateur et vous devriez voir le texte « Hello, world. You're at the developers index. » qui a été défini dans la vue index.

Paramètres de la fonction path

La fonction `path()` reçoit quatre paramètres, dont deux sont obligatoires : **route** et **view**, et deux facultatifs : **kwargs** et **name**. À ce stade, il est intéressant d'examiner le rôle de chacun de ces paramètres.

route

route est une chaîne contenant un motif d'URL. Lorsqu'il traite une requête, Django commence par le premier motif dans `urlpatterns` puis continue de parcourir la liste en comparant l'URL reçue avec chaque motif jusqu'à ce qu'il en trouve un qui correspond.

Les motifs ne cherchent pas dans les paramètres GET et POST, ni dans le nom de domaine. Par exemple, dans une requête vers <https://www.example.com/myapp/>, l'URLconf va chercher `myapp/`. Dans une requête vers <https://www.example.com/myapp/?page=3>, l'URLconf va aussi chercher `myapp/`.

view

Lorsque Django trouve un motif correspondant, il appelle la fonction de vue spécifiée, avec un objet `HttpRequest` comme premier paramètre et toutes les valeurs « capturées » par la route sous forme de paramètres nommés. Nous montrerons cela par un exemple un peu plus loin.

kwargs

Des paramètres nommés arbitraires peuvent être transmis dans un *dictionnaire* vers la vue cible.

Parenthèse Python 🐍

Un dictionnaire est une structure de donnée élémentaire de Python. Elle fonctionne sur le principe de clé-valeur. Nous y reviendrons !

name

Le nommage des URL permet de les référencer de manière non ambiguë depuis d'autres portions de code Django, **en particulier depuis les gabarits**. Cette fonctionnalité puissante permet d'effectuer des changements globaux dans les modèles d'URL de votre projet en ne modifiant qu'un seul fichier.

Mon premier modèle

Nous allons maintenant définir les modèles – essentiellement, le schéma de base de données, avec quelques métadonnées supplémentaires.

Philosophie de django

Un modèle est la source d'information unique et définitive pour vos données. Il contient les champs essentiels et le comportement attendu des données que vous stockerez. Django respecte la philosophie DRY (**Don't Repeat Yourself**, « ne vous répétez pas »). Le but est de définir le modèle des données à un seul endroit, et ensuite de dériver automatiquement ce qui est nécessaire à partir de celui-ci.

Ceci inclut les migrations. En effet, les migrations sont entièrement dérivées du fichier des modèles et ne sont au fond qu'un historique que Django peut parcourir pour mettre à jour le schéma de la base de données pour qu'il corresponde aux modèles actuels.

Migration et base de données

Lorsqu'on exécute la commande `python manage.py migrate`, cela provoque une migration du modèle. Nous n'avons pas encore défini de modèle, mais certains éléments existent déjà ! Ainsi, après avoir saisi la commande donnée, un fichier `db.sqlite3` apparaît.

Par défaut, Django utilise une base de donnée sqlite. Cela peut facilement être changé en modifiant le fichier `settings`. C'est ce que nous ferons un peu plus tard. Voici la configuration actuelle :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Parenthèse Python 🐍

DATABASES est initialisé avec un dictionnaire. Dans ce dictionnaire, il n'y a qu'un seul élément dont la clé est 'default' et la valeur est un autre dictionnaire.

La valeur de `default` est donc un dictionnaire contenant deux valeurs.

Notons ici que `BASE_DIR` est un objet de type `path`. L'opérateur `/` permet de concaténer un chemin.

Exercice Python 🐍 ★

Quelles sont les résultats cachés par des points d'interrogation des instructions suivantes ?

```
>>> trigrammes = {'jlc': 'Jonathan Lechien', 'sdr': 'Sébastien Drobisz'}
>>> trigrammes['jlc']
?
# On peut mettre d'autres types comme clé-valeur. On peut même varier dans
un même dictionnaire.
>>> mon_dico = {3: 'trois', 'trois': 3}
>>> mon_dico[3]
?
>>> mon_dico['trois']
?
```

Module sqlite pour Visual Studio Code ou VSCodium

Nous allons vérifier le contenu de la base de donnée. Pour cela, nous allons utiliser le module `sqlite` pour Visual Studio Code.

📖 Remarquez que vous pouvez aussi utiliser la version libre de ce logiciel, installé sur les machines de l'école : VSCodium.

1. Allez dans l'onglet module afin d'installer `sqlite`.
2. Une fois ceci fait, faite un clic droit sur le fichier : `open database`.
3. En apparence, rien n'a changé, mais en y regardant de plus près, un onglet sqlite manager est apparu tout en bas sous l'arborescence de fichiers. En cliquant dessus, vous aurez un

aperçu du schéma du site.

4. Vous pouvez réaliser des requêtes afin de voir le contenu de ces tables. Vérifiez le contenu de la table `auth_permission` avant d'aller plus loin. ★

Développons notre premier modèle

Nous allons maintenant développer notre premier modèle. Dans le fichier `developer/models.py`, copiez le contenu ci-dessous.

```
from django.db import models

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

class Task(models.Model):
    title = models.CharField(max_length=100, unique=True)
    description = models.TextField()
    assignee = models.ForeignKey(Developer, related_name="tasks",
on_delete=models.CASCADE, null=True, verbose_name="assignee")
```

- Ici, chaque modèle est représenté par une classe qui hérite de `django.db.models.Model`. Chaque modèle possède des variables de classe, chacune d'entre elles représentant un champ de la base de données pour ce modèle. (🧐 En python, l'héritage se fait en mettant le modèle parent entre parenthèse. En ☕ java, il se fait avec le mot clé `extends`.)
- Chaque champ est représenté par une instance d'une classe `Field` – par exemple, `CharField` pour les champs de type caractère, et `DateTimeField` pour les champs date et heure. Cela indique à Django le type de données que contient chaque champ.
- Le nom de chaque instance de `Field` (par exemple, `first_name` ou `title`) est le nom du champ en interne. Vous l'utiliserez dans votre code Python et votre base de données l'utilisera comme nom de colonne.
- Vous pouvez utiliser le premier paramètre de position (facultatif) d'un `Field` pour donner un nom plus lisible au champ. C'est utilisé par le système d'inspection de Django, et aussi pour la documentation. Si ce paramètre est absent, Django utilisera le nom du champ interne. Dans l'exemple, nous n'avons défini qu'un seul nom, pour `first_name` (en réalité, le nom donné automatiquement par Django est le même... 🤔). Parfois, le premier champ est pris par un autre paramètre. Dans ce cas, il est malgré tout possible d'assigner une valeur grâce à `verbose_name` (voir `assignee`).
- Certaines classes `Field` possèdent des paramètres obligatoires. La classe `CharField`, par exemple, a besoin d'un attribut `max_length`. Ce n'est pas seulement utilisé dans le schéma de base de la base de données, mais également pour valider les champs, comme nous allons voir prochainement.
- Finalement, notez que nous définissons une relation, en utilisant `ForeignKey` (plusieurs-à-un). Cela indique à Django que chaque tâche (`Task`) n'est relié qu'à un seul développeur. Django propose tous les modèles classiques de relations : plusieurs-à-un, plusieurs-à-plusieurs, un-à-un.

Pour plus d'information sur les champs :

- [Options des champs](#) 🕶️ ;
- [Type des champs](#) 🕶️ ;
- [plusieurs-à-plusieurs](#) 🕶️ ;
- [plusieurs-à-un](#) 🕶️ ;
- [un-à-un](#) 🕶️.

Activation du modèle et migrations

Ce petit morceau de code décrivant les modèles fournit beaucoup d'informations à Django. Cela lui permet de :

- créer un schéma de base de données (instructions `CREATE TABLE`) pour cette application.
- Créer une API Python d'accès aux bases de données pour accéder aux objets `Developer` et `Task`.

Essayons de migrer les changements 🐱. La migration se fait grâce à la commande

```
$ python manage.py makemigrations.
```

```
python manage.py makemigrations
No changes detected
```

Rien ne s'est passé, en réalité, il faut d'abord "installer" l'application developer.

Installation de l'application developer

Pour inclure l'application dans notre projet, nous avons besoin d'ajouter une référence à sa classe de configuration dans le réglage `INSTALLED_APPS` présent dans le fichier `settings.py`. La classe `DeveloperConfig` se trouve dans le fichier `developer/apps.py`, ce qui signifie que son chemin pointé est `developer.apps.DeveloperConfig`. Modifiez le fichier `mproject/settings.py` et ajoutez ce chemin pointé au réglage `INSTALLED_APPS`. Il doit ressembler à ceci :

```
mproject/settings.py

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

    # My apps 🐱 new
    'developer.apps.DeveloperConfig', # 🐱 new
```

Commande makemigrations

Maintenant que c'est fait, nous pouvons relancer la commande `python manage.py makemigrations`.

Vous devriez avoir quelque chose de similaire à ceci :

```
Migrations for 'developer':
  developer\migrations\0001_initial.py
    - Create model Developer
    - Create model Task
```

En exécutant `makemigrations`, vous indiquez à Django que vous avez effectué des changements à vos modèles (dans le cas présent, vous avez créé un nouveau modèle) et que vous aimeriez que ces changements soient stockés sous forme de migration.

Les migrations sont le moyen utilisé par Django pour stocker les modifications de vos modèles (et donc de votre schéma de base de données), il s'agit de fichiers présent sur votre disque. Vous pouvez consultez la migration pour vos nouveaux modèles si vous le voulez ; il s'agit du fichier `developer/migrations/0001_initial.py` ★. Soyez sans crainte, vous n'êtes pas censé les lire chaque fois que Django en crée, mais ils sont conçus pour être lisibles facilement par un humain au cas où vous auriez besoin d'adapter manuellement les processus de modification de Django.

Commande sqlmigrate

Il existe une commande qui exécute les migrations et gère automatiquement votre schéma de base de données, elle s'appelle `migrate`. Nous y viendrons bientôt, mais tout d'abord, voyons les instructions SQL que la migration produit. La commande `sqlmigrate` accepte des noms de migrations et affiche le code SQL correspondant :

```
$ python manage.py sqlmigrate developer 0001
```

Vous devriez voir quelque chose de similaire à ceci (remis en forme par souci de lisibilité) :

```
BEGIN;
--
-- Create model Developer
--
CREATE TABLE "developer_developer" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "first_name" varchar(200) NOT NULL,
    "last_name" varchar(200) NOT NULL);
--
-- Create model Task
--
CREATE TABLE "developer_task" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(100) NOT NULL UNIQUE,
    "description" text NOT NULL,
    "assignee_id" integer NOT NULL
        REFERENCES "developer_developer" ("id")
        DEFERRABLE INITIALLY DEFERRED);

CREATE INDEX "developer_task_assignee_id_497c1e11"
ON "developer_task" ("assignee_id");

COMMIT;
```

Notez les points suivants :

- Ce que vous verrez dépendra de la base de données que vous utilisez. L'exemple ci-dessus est généré pour SQLite.
- Les noms de tables sont générés automatiquement en combinant le nom de l'application (developer) et le nom du modèle en minuscules – developer et task (vous pouvez modifier ce comportement).
- Des clés primaires (ID) sont ajoutées automatiquement (vous pouvez modifier ceci également).
- Par convention, Django ajoute "_id" au nom de champ de la clé étrangère (et oui, vous pouvez aussi changer ça).
- La relation de clé étrangère est rendue explicite par une contrainte FOREIGN KEY.
- Ce que vous voyez est adapté à la base de données que vous utilisez. Ainsi, des champs spécifiques à celle-ci comme `auto_increment` (MySQL), `serial` (PostgreSQL) ou `integer`

`primary key autoincrement` (SQLite) sont gérés pour vous automatiquement. Tout comme pour les guillemets autour des noms de champs (simples ou doubles).

- La **commande `sqlmigrate` n'exécute pas réellement la migration** dans votre base de données - elle se contente de l'afficher à l'écran de façon à vous permettre de voir le code SQL que Django pense nécessaire. C'est utile pour savoir ce que Django s'apprête à faire ou si vous avez des administrateurs de base de données qui exigent des scripts SQL pour faire les modifications.

★ Si vous vérifiez la base de donnée, vous ne verrez aucun changement. En effet, vous n'avez pas encore appliqué la migration.

Commande `migrate`

Appliquons maintenant notre migration. Saisissez la commande :

```
$ python manage.py migrate
```

La commande `migrate` sélectionne toutes les migrations qui n'ont pas été appliquées (Django garde la trace des migrations appliquées en utilisant une table spéciale dans la base de données : `django_migrations`) puis les exécute dans la base de données, ce qui consiste essentiellement à synchroniser les changements des modèles avec le schéma de la base de données.

★ Vérifiez à nouveau la base de donnée. Quelle(s) table(s) a(ont) été ajoutée(s) ?

Résumé des commandes `makemigrations` et `migrate`

Les migrations sont très puissantes et permettent de gérer les changements de modèles dans le temps, au cours du développement d'un projet, sans devoir supprimer la base de données ou ses tables et en refaire de nouvelles. Une migration s'attache à mettre à jour la base de données en live, sans perte de données. Nous les aborderons plus en détails dans une partie ultérieure de ce didacticiel, mais pour l'instant, reprenez le guide en trois étapes pour effectuer des modifications aux modèles :

1. Modifiez les modèles (dans `models.py`).
2. Exécutez `python manage.py makemigrations` pour créer des migrations correspondant à ces changements.
3. Exécutez `python manage.py migrate` pour appliquer ces modifications à la base de données.

La raison de séparer les commandes pour créer et appliquer les migrations est que celles-ci vont être ajoutées dans votre système de gestion de versions et qu'elles seront livrées avec l'application ; elles ne font pas que faciliter le développement, elles sont également exploitables par d'autres développeurs ou en production.

Pensez à utiliser un gestionnaire de versions

Cela pourrait être utile pour vous de revenir en arrière dans ce tutoriel et dans votre code. Pensez donc à versionner votre code aussi souvent que vous le jugerez nécessaire.

Interface de programmation (API)

Maintenant, utilisons un shell interactif Python pour jouer avec l'API que Django met à votre disposition. Pour lancer un shell Python, utilisez cette commande :

```
$ python manage.py shell
```

Nous utilisons celle-ci au lieu de simplement taper « python », parce que `manage.py` définit la variable d'environnement `DJANGO_SETTINGS_MODULE`, qui indique à Django le chemin d'importation Python vers votre fichier `developer/settings.py`.

📖 Notez que cette API est utilisée plus loin pour la manipulation des modèles. La maîtrise de celle-ci est donc essentiel pour le développement d'application Web avec Django ! ⚠️

Une fois dans le shell, explorez [l'api de base de donnée](#) 😎.

```
>>> from developer.models import Developer, Task
>>> Developer.objects.all()
<QuerySet []>
```

On obtient un *QuerySet* en utilisant le *Manager* du modèle. Chaque modèle a au moins un Manager ; il s'appelle *objects* par défaut.

- [QuerySet](#) 😎
- [Manager](#) 😎

Ici, le *QuerySet* est vide puisque aucun élément n'a été créé.

```
>>> jlc = Developer(first_name='Jonahtan', last_name='Lechien')
```

Nous venons de créer un nouveau développeur. Vérifiez que celui-ci a bien été créé dans la base de donnée ! 🐰★

Vous vous êtes peut-être fait avoir. Quoiqu'il en soit, vous avez pu vérifier qu'il n'y a aucun nouvel enregistrement. Il est nécessaire de le sauvegarder pour que celui-ci soit enregistré en base de donnée..

```
>>> jlc.save()
```

Il est possible de créer un nouvel enregistrement en passant par un manager, il n'est alors pas nécessaire de le sauvegarder. Essayez ! ★

```
>>> sdr = Developer.objects.create(first_name='Sébastien',
last_name='Drobisz')
```

Continuons d'explorer

```
>>> jlc.id
1
>>> jlc.first_name
'Jonahtan'
>>> jlc.last_name
'Lechien'
>>> jlc.first_name = 'Jonathan'
>>> jlc.save()
>>> Developer.objects.all()
<QuerySet [<Developer: Developer object (1)>, <Developer: Developer object (2)>]>
```

Une seconde. `<Developer: Developer object (1)>` n'est pas une représentation très utile de cet objet. On va arranger cela en éditant le modèle `Developer` (dans le fichier `developer/models.py`) et en ajoutant une méthode `__str__()` à `Developer` et à `Task`:

```

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

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

class Task(models.Model):
    title = models.CharField(max_length=100, unique=True)
    description = models.TextField()
    assignee = models.ForeignKey(Developer, related_name="tasks",
on_delete=models.CASCADE, null=True, verbose_name="assignee")

    def __str__(self): ➡ new
        return f"{self.title} ({self.description})" ➡ new

```

Parenthèses python 🐍

- Vous l'aurez probablement compris, `__str__()` est à Python 🐍 ce que `toString()` est à Java ☕.
- Avant, on pouvait faire `'%s %s' % (self.first_name, self.last_name)` 🤖 pour formater du texte. Ensuite, les choses se sont améliorées et on pouvait faire `'{}'.format(self.first_name, self.last_name)` 🤖. Tout ça, c'était avant, maintenant (depuis python 3.6), on utilise la notation `f"{self.first_name} {self.last_name}"` 👍.

Vous pouvez relancer le shell maintenant.

```

>>> Developer.objects.all()
<QuerySet [ <Developer: Jonahtan Lechien>, <Developer: Sébastien Drobisz>]>

```

Continuons sur notre lancée

```

>>> Developer.objects.filter(id=1)
<QuerySet [ <Developer: Jonahtan Lechien>]>
>>> Developer.objects.filter(first_name__startswith='S')
<QuerySet [ <Developer: Sébastien Drobisz>]>
>>> Developer.objects.get(pk=1)
<Developer: Jonahtan Lechien>
>>> faire_cours_django = Task.objects.create(title='cours django',
description='Faire le cours de django (avec un peu de python)')
>>> faire_cours_django.assignee = sdr
>>> faire_cours_django.save()
>>> jlc.tasks.create(title='cours Odoo', description='Faire le cours sur
Odoo')

```

Si vous avez lu le tuto [ici](#) vous avez pu remarquer que nous utilisons `tasks` plutôt que `task_set`. Cela nous est possible puisque nous avons défini le paramètre `relative_name` dans notre modèle `Task`.

```
>>> jlc.tasks.all()
<QuerySet [ <Task: cours Odoo (Faire le cours sur Odoo)>]>
>>> jlc_task = jlc.tasks.all()[0]
>>> jlc_task.title
'cours Odoo'
>>> jlc.tasks.count()
1
>>> jlc_task.delete()
(1, {'developer.Task': 1})
>>> jlc.tasks.count()
0
```

Toujours plus d'information 🕶 :

- [Recherche dans les champs.](#)
- [Référence des objets liés.](#)

Vues et templates introduction

Une vue est un « type » de page Web dans votre application Django qui sert généralement à une fonction précise et possède un gabarit spécifique. Par exemple, dans une application de blog, vous pouvez avoir les vues suivantes :

- La page d'accueil du blog – affiche quelques-uns des derniers billets.
- La page de « détail » d'un billet – lien permanent vers un seul billet.
- La page d'archives pour une année – affiche tous les mois contenant des billets pour une année donnée.
- La page d'archives pour un mois – affiche tous les jours contenant des billets pour un mois donné.
- La page d'archives pour un jour – affiche tous les billets pour un jour donné.
- Action de commentaire – gère l'écriture de commentaires sur un billet donné.

Dans notre application, nous possédons plusieurs vues. Parmi celles-ci :

- une page d'accueil ;
- une page qui liste les développeurs ;
- une page qui donne le détail des développeurs - c'est-à-dire le nom, prénom ainsi que toutes ses tâches ;
- une page pour l'ensemble des tâches...

Dans Django, les pages Web et les autres contenus sont générés par des vues. Chaque vue est représentée par une fonction Python (ou une méthode dans le cas des vues basées sur des classes). Django choisit une vue en examinant l'URL demandée (pour être précis, la partie de l'URL après le nom de domaine).

Un modèle d'URL est la forme générale d'une URL ; par exemple : `/archive/<année>/<mois>/`.

Pour passer de l'URL à la vue, Django utilise ce qu'on appelle des configurations d'URL (URLconf). Une configuration d'URL associe des motifs d'URL à des vues.

Vive les gabarits (templates)

La vue que nous avons écrite jusqu'à maintenant est très sommaire, une page web ne ressemble en rien à cela.

De plus, il y a un problème : l'allure de la page est codée en dur dans la vue. Si vous voulez changer le style de la page, vous devrez modifier votre code Python. Nous allons donc utiliser le système de gabarits de Django pour séparer le style du code Python en créant un gabarit que la vue pourra utiliser.

Tout d'abord, créez un répertoire nommé `templates` dans votre répertoire `developer`. C'est là que Django recherche les gabarits.

Le paramètre `TEMPLATES` de votre projet indique comment Django va charger et produire les gabarits. Le fichier de réglages par défaut configure un moteur DjangoTemplates dont l'option `APP_DIRS` est définie à `True`. Par convention, DjangoTemplates recherche un sous-répertoire « `templates` » dans chaque application figurant dans `INSTALLED_APPS`. (Allez vérifier la présence de cette option dans le fichier `project/settings.py` ★)

Dans le répertoire `templates` que vous venez de créer, créez un autre répertoire nommé `developer` dans lequel vous placez un nouveau fichier `index.html`. Autrement dit, le chemin de votre gabarit doit être `developer/templates/developer/index.html`. Conformément au fonctionnement du chargeur de gabarit `app_directories` (cf. explication ci-dessus), vous pouvez désigner ce gabarit dans Django par `developer/index.html`.

Espace de noms des gabarits ⚠

Il serait aussi possible de placer directement nos gabarits dans `developer/templates` (plutôt que dans un sous-répertoire `developer`), mais ce serait une mauvaise idée. Django choisit le premier gabarit qu'il trouve pour un nom donné et dans le cas où vous avez un gabarit de même nom dans une autre application, Django ne fera pas la différence. Il faut pouvoir indiquer à Django le bon gabarit, et la meilleure manière de faire cela est d'utiliser des espaces de noms. C'est-à-dire que nous plaçons ces gabarits dans un autre répertoire portant le nom de l'application.

Lisez et insérez ce code dans le gabarit `developer/index.html`

`developer/index.html`

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, > initial-scale=1.0">
  <title>MProject</title>
</head>
<body>
  <h1>MProject</h1>
  <h2>Liste des développeurs</h2>

  {% if developers %}
  <ul>
    {% for dev in developers %}
    <li>{{ dev.first_name }}</li>
    {% endfor %}
  </ul>
  {% else %}
  <p><strong>Il n'y a aucun développeur enregistré !</strong></p>
  {% endif %}
</body>
</html>
```

Mettez à jours la vue afin de permettre le rendu de ce gabarit.

```
from django.shortcuts import render
# from django.http import HttpResponse ➡ old

from .models import Developer ➡ new

def index(request):
    # return HttpResponse("Hello, world. You're at the developers index.") ➡ old
    context = { ➡ new
        'developers': Developer.objects.all() ➡ new
    } ➡ new

    return render(request, 'developer/index.html', context) ➡ new
```

Ce code charge le gabarit appelé `developer/index.html` et lui fournit un contexte. Ce contexte est un dictionnaire qui fait correspondre des objets Python (valeurs) à des noms de variables de gabarit (clés).

Chargez la page en appelant l'URL « `/developer/` » dans votre navigateur et vous devriez voir une liste à puces contenant une liste de développeurs.

Exercices ★

- Supprimez chacun des développeurs et vérifiez que le message "Il n'y a aucune développeur enregistré !" soit bien affiché.
- Rajoutez ensuite au moins deux développeurs.

Une deuxième vue

Nous allons ajouter une deuxième vue qui va nous permettre d'afficher le détail des informations que l'on a sur les développeurs. Nous allons devoir compléter les étapes suivantes :

1. Ajout d'une url qui pointe vers la nouvelle vue
2. Ajout d'une vue
3. Ajout d'un nouveau template.

`developer/url.py`

```
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:developer_id>', views.detail, name='detail'), ➡ new
]
```

`developer/view.py`


```
def index(request):
    context = {
        'developers': Developer.objects.all(),
    }

    return render(request, 'developer/index.html', context)

def detail(request, developer_id): ➡new
    developer = Developer.objects.get(pk=developer_id) ➡new
    return render(request, 'developer/detail.html', {'developer': developer})
➡new
```

developer/templates/developer/detail.html

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>MProject</title>
</head>
<body>
    <h1>MProject</h1>
    <h2>Détail de {{ developer.first_name }}</h2>

    <p><Strong>Prénom : {{ developer.first_name }}</Strong></p>
    <p><Strong>Nom de famille : {{ developer.last_name }}</Strong></p>
</body>
</html>
```

Ouvrez votre navigateur à l'adresse « `/developer/3/` ». La méthode `detail()` sera exécutée et affichera le développeur fourni dans l'URL.

📖 Nous vous suggérons ici d'utiliser la valeur 3 pour l'adresse. Cette valeur devrait correspondre à l'id du développeur que vous avez recréé après avoir supprimé, comme demandé, les deux développeurs "sdr" et "jlc". Si vous avez un doute, vous pouvez aller dans le `shell` et lancer la commande `[dev.id for dev in Developer.objects.all()]` après avoir importé la classe `Developer`. Cette commande va vous retourner la liste des ids présents dans la base de donnée. (Ce code n'a rien de magique, il s'agit de la constitution d'une liste sur base d'un parcours des développeurs disponibles dans la BDD.)

Lorsque quelqu'un demande une page de votre site Web, par exemple « `/developer/3/` », Django charge le module Python `mproject.urls` parce qu'il est mentionné dans le réglage `ROOT_URLCONF`. Il trouve la variable nommée `urlpatterns` et parcourt les motifs dans l'ordre. Après avoir trouvé la correspondance `'developer/'`, il retire le texte correspondant ("`developer/`") et passe le texte restant – `"3/"` – à la configuration d'URL "`developer.urls`" pour la suite du traitement. Là, c'est `<int:developer_id>/` qui correspond ce qui aboutit à un appel à la vue `detail()` comme ceci :

```
detail(request=<HttpRequest object>, developer_id=3)
```

La partie `developer_id=3` vient de `<int:developer_id>`. En utilisant des chevrons, cela « capture » une partie de l'URL l'envoie en tant que paramètre nommé à la fonction de vue ; la partie `:developer_id` de la chaîne définit le nom qui va être utilisé pour identifier le motif trouvé, et la partie `<int:` est un convertisseur qui détermine ce à quoi les motifs doivent correspondre dans cette partie du chemin d'URL.

Pour plus d'info sur la distribution d'url, cela se passe [ici](#). 🕶

Erreur 404

Si vous avez bien suivi ce cours jusqu'à maintenant, vous ne devriez pas avoir rencontré d'erreur. Si vous vous rendez sur l'adresse `localhost:8000/developer/42` vous serez redirigé vers une page d'erreur Django. En effet, à moins d'avoir créé beaucoup de développeurs, le développeur possédant l'id 42 n'existe pas.

Nous pouvons corriger cela en utilisant la fonction `get_object_or_404`.

```
from django.shortcuts import render, get_object_or_404 ➡new
from django.http import HttpResponse

from .models import Developer

def index(request):
    context = {
        'developers': Developer.objects.all(),
    }

    return render(request, 'developer/index.html', context)

def detail(request, developer_id):
    #developer = Developer.objects.get(pk=developer_id) ➡old
    developer = get_object_or_404(Developer, pk=developer_id) ➡new
    return render(request, 'developer/detail.html', {'developer': developer})
```

La fonction `get_object_or_404()` prend un modèle Django comme premier paramètre et un nombre arbitraire de paramètres mots-clés, qu'il transmet à la méthode `get()` du gestionnaire du modèle. Elle lève une exception `Http404` si l'objet n'existe pas.

Lien entre les vues

Pour passer d'une vue à l'autre, nous allons naturellement utiliser des liens `html` (`<a>`). Revenons dans la vue `index` et plus précisément dans le template et ajoutons ces liens.

`index.html`

```
# ...
{% if developers %}
<ul>
    {% for dev in developers %}
    {#{<li>{{ dev.first_name }}</li>#} ➡ ceci est commenté !
    <li><a href='/developer/{{ dev.id }}'>{{ dev.first_name }}</a></li> ➡
new
    {% endfor %}
</ul>
{% else %}
    <p><strong>Il n'y a aucune développeur enregistré !</strong>/p>
{% endif %}
# ...
</body>
```

Vous pouvez maintenant essayer d'aller sur l'index de votre site et suivre les liens qui sont créés !

`{% url %}`

Le problème de cette approche codée en dur et fortement couplée est qu'il devient fastidieux de modifier les URL dans des projets qui ont beaucoup de gabarits. Cependant, comme vous avez défini le paramètre « name » dans les fonctions `path()` du module `developer.urls`, vous pouvez supprimer la dépendance en chemins d'URL spécifiques définis dans les configurations d'URL en utilisant la balise de gabarit `{% url %}` :

```
<li><a href="{% url 'detail' dev.id %}">{{ dev.first_name }}</a></li>
```

Le principe de ce fonctionnement est que l'URL est recherchée dans les définitions du module `developer.urls`. Ci-dessous, vous pouvez voir exactement où le nom d'URL de « detail » est défini :

```
path('<int:developer_id>/', views.detail, name='detail'),
```

Si vous souhaitez modifier l'URL de détail des développeurs, par exemple sur le modèle `developer/specifics/12/`, il suffit de faire la modification dans `developer/urls.py`. Il n'est pas nécessaire de modifier un nombre potentiellement grand de gabarit.

`developer/urls.py`

```
path('specifics/<int:developer_id>/', views.detail, > name='detail'), #➡
ajout de specifics
```

Espaces de noms et noms d'URL

Le projet ne contient actuellement qu'une seule application, `developer`. Plus tard, une autre application va se greffer à notre projet. Comment Django arrive-t-il à différencier les noms d'URL entre elles ? Par exemple, l'application `developer` possède une vue `detail` et il se peut tout à fait qu'une autre application du même projet en possède aussi une. Comment peut-on indiquer à Django quelle vue d'application il doit appeler pour une URL lors de l'utilisation de la balise de gabarit `{% url %}` ?

La réponse est donnée par l'ajout d'espaces de noms à votre configuration d'URL. Dans le fichier `developer/urls.py`, ajoutez une variable `app_name` pour définir l'espace de nom de l'application :

developer/url.py

```
app_name = 'developer' ➡ new
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:developer_id>', views.detail, name='detail'),
]
```

Modifiez maintenant les liens du gabarit `developer/index.html` pour qu'elle pointe vers la vue « detail » à l'espace de nom correspondant.

developer/index.html

```
#...
{% if developers %}
<ul>
    {% for dev in developers %}
    {#{<li>{{ dev.first_name }}</li>#}
    <li><a href="{% url 'developer:detail' dev.id %}">{{ dev.first_name }}</a>
</li> <!-- ➡ ajout de "developer:" -->
    {% endfor %}
</ul>
#...
```

Héritage de template

On vous a annoncé plus tôt que DRY est la devise de Django. Pourtant, si l'on regarde les templates, nous pouvons voir énormément de redondance. Nous allons améliorer cela et mettre un peu de style dans notre site.

Template du projet


La première chose que nous allons faire, est de définir un gabarit de base pour le projet.


Pour cela, nous devons d'abord modifier le fichier `settings.py`

Ajoutez le code suivant :

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'], ➡ new
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]
```

Ce bout de code permet d'ajouter une liste de chemin dans lequel Django peut trouver un gabarit. Le chemin que l'on définit ici est un dossier qui se nomme `templates` et qui se trouve dans le dossier de base. Créez ce dossier et ajoutez un fichier nommé `_base.html`.

 Le nom de ce fichier est totalement arbitraire, toutefois, nous précédon le nom par un underscore puisque ce fichier est destiné à être étendu. La communauté de Django vous remerciera de respecter cette convention.

 Petit rappel, `BASE_DIR / 'templates'` désigne la concaténation d'un chemin (`BASE_DIR`) qui est défini dans le fichier `settings.py` avec `templates`.

Gabarit de base

Dans le fichier `_base.html`, placez-y le code suivant

`BASE_DIR/templates/_base.html`

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" > content="width=device-width, > initial-scale=1.0">
  <title>{% block title %}MProject{% endblock > title %}</title>
</head>
<body>
  <h1>MProject</h1>
  <h2>Un site web de gestion de tâches</h2>

  {% block content %}
  {% endblock content %}
</body>
</html>
```

Les balises `{% block %}` permettent de définir des blocs qui peuvent être surchargés par les gabarits enfants.

 Notez qu'il n'est pas obligatoire de nommer le bloc fermé (`{% endblock content %}`), mais nous préférons par soucis de lisibilité.

Extension du gabarit

Modifier maintenant le code du gabarit `detail.html`

`developer/detail.html`

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

{% block title %} Détail - {{ developer.> first_name }} {{
developer.last_name }} {% > endblock title %}
{% block content %}
  <p><Strong>Prénom : {{ developer.first_name > }}</Strong></p>
  <p><Strong>Nom de famille : {{ developer.> last_name }}</Strong></p>
{% endblock content %}
```

Ainsi que le fichier `index.html`

`developer/index.html`

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

{% block title %} Liste des développeurs {% endblock title %}

{% block content %}
{% if developers %}
<ul>
  {% for dev in developers %}
  {#{<li>{{ dev.first_name }}</li>#}
  <li><a href="{% url 'developer:detail' dev.id %}"> {{ dev.first_name }}
</a></li>
  {% endfor %}
</ul>
{% else %}
<p><strong>Il n'y a aucune développeur enregistré !</strong></p>
{% endif %}
{% endblock content %}
```

Les gabarits ont été modifiés afin que les morceaux de code soient placés correctement au sein des blocs `title` et `content` que nous avons défini dans le gabarit de base. Nous avons ajouté la balise `{% extends '_base.html' %}` dans les deux derniers gabarit afin de notifier à Django qu'ils héritent du gabarit de base.

Vérifiez que votre site fonctionne toujours aussi bien que avant. ★

Pour plus d'information sur les balises, lisez cette [page](#).

Un peu de style avec bootstrap 4 et font-awesome

Maintenant que nous avons un gabarit de base, nous allons lui ajouter un peu de style.

Dans le head du fichier `_base.html`, nous ajoutons les liens vers un CDN `bootstrap4` ainsi que vers `fontawesome` pour agrémenter notre projet de quelques logos.

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js">
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js">
</script>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js">
</script>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/4.7.0/css/font-awesome.min.css">
```

Ajoutons un peu de forme pour l'entête du site

```
<div class="p-3 bg-primary">
  <h1 class="display-1">MProject</h1>
  <h2>Un site web de gestion de tâches</h2>
</div>
```

Ajoutons un menu

```

<nav class="navbar navbar-expand-sm bg-primary navbar-dark border-top border-white">
  <ul class="navbar-nav">
    <li id="nav-home" class="nav-item">
      <a class="nav-link" href="#"><i class="fa fa-home"></i></a>
    </li>
    <li id="nav-dev" class="nav-item active">
      <a class="nav-link" href="{% url 'developer:index' %}">Developers</a>
    </li>
    <li id="nav-task" class="nav-item">
      <a class="nav-link" href="#">Tasks</a>
    </li>
  </ul>
</nav>

```

Nous n'avons pas ajouté de fonctionnalité, mais notre site est maintenant un peu plus habillé. Passez de la vue `détail` à la vue `index` et profiter du site mis en style.

Question ★

Quel bout de code permet de revenir à la liste des développeurs lorsque je clique sur le menu `Developers` ?

Ajout d'un développeur

Pour le moment, nous devons passer par le `shell` afin de pouvoir ajouter un développeur. Ce n'est pas très pratique.

Nous allons ajouter au sein de l'index de développeur, la possibilité d'ajouter un développeur.

Ainsi, sur la page de l'index, nous aurons la liste des développeurs ainsi qu'un formulaire permettant d'en ajouter un.

Ajout du formulaire

Ajoutez ce morceau de code dans le gabarit `index.html`

`index.html`

```

#...
{% else %}
  <p><strong>Il n'y a aucune développeur enregistré !</strong></p>
{% endif %}

<form action="{% url 'developer:create' %}" method="post"> ➡ new
  {% csrf_token %} ➡ new

  <label for="first_name">First name</label> ➡ new
  <input type="text" name="first_name" required> ➡ new
  <label for="last_name">Last name</label> ➡ new
  <input type="text" name="last_name" required> ➡ new
  <button type="submit">Create</button> ➡ new
</form> ➡ new
{% endblock content %}

```

Un résumé rapide :

- Ce gabarit affiche maintenant un formulaire avec deux champs texte et un bouton de création. Notez que dans ce formulaire, nous avons nommé les deux entrées `first_name` et `last_name`. Ce sont les concepts de base des formulaires HTML.
- Nous avons défini `{% url 'developer:create' %}` comme attribut action du formulaire, et nous avons précisé `method="post"`. L'utilisation de `method="post"` (par opposition à `method="get"`) est très importante, puisque le fait de valider ce formulaire va entraîner des modifications de données sur le serveur. À chaque fois qu'un formulaire modifie des données sur le serveur, vous devez utiliser `method="post"`. Cela ne concerne pas uniquement Django ; c'est une bonne pratique à adopter en tant que développeur Web.
- Comme nous créons un formulaire POST (qui modifie potentiellement des données), il faut se préoccuper des attaques inter-sites. Heureusement, vous ne devez pas réfléchir trop longtemps car Django offre un moyen pratique à utiliser pour s'en protéger. En bref, tous les formulaires POST destinés à des URL internes doivent utiliser la balise de gabarit `{% csrf_token %}`.

Url et vue pour la création de développeur

Maintenant, nous allons créer une vue Django qui récupère les données envoyées pour nous permettre de les exploiter. D'abord, nous devons ajouter un chemin vers cette nouvelle vue.

`developer/urls.py`

```
app_name = 'developer'
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:developer_id>', views.detail, name='detail'),
    path('create', views.create, name='create'),
]
```

👉 new

et maintenant, ajoutons la vue

`developer/views.py`

```
#...
from django.http import HttpResponse, HttpResponseRedirect 👉 ajout
HttpResponseRedirect
from django.urls import reverse 👉 new
#...
def create(request): 👉 new
    Developer.objects.create( 👉 new
        first_name=request.POST['first_name'], 👉 new
        last_name = request.POST['last_name'] 👉 new
    ) 👉 new
    # Toujours renvoyer une HttpResponseRedirect après avoir géré correctement
    # les données de la requête POST. Cela empêche les données d'être postée
    # deux
    # fois si l'utilisateur clique sur le bouton précédent.
    return HttpResponseRedirect(reverse('developer:index')) 👉 new
```

Ce code contient quelques points encore non abordés dans ce tutoriel :

- `request.POST` est un objet similaire à un dictionnaire qui vous permet d'accéder aux données envoyées par leurs clés. Dans ce cas, `request.POST['first_name']` et `request.POST['last_name']` renvoient le prénom et nom du développeur sous forme d'une chaîne de caractères. Les valeurs dans `request.POST` sont toujours des chaînes de

caractères. Pensez donc à réaliser une transformation si le type de votre entrée n'est pas de nature `string`.

Parenthèse Python 🐍

En Python vous pouvez convertir une chaîne de caractère en un entier grâce à la fonction `int()`. Par exemple : `int("42")`.

- Notez que Django dispose aussi de `request.GET` pour accéder aux données GET de la même manière – mais nous utilisons explicitement `request.POST` dans notre code, pour s'assurer que les données ne sont modifiées que par des requêtes POST.
- Après la création d'un développeur, le code renvoie une `HttpResponseRedirect` plutôt qu'une `HttpResponse` normale. `HttpResponseRedirect` prend un seul paramètre : l'URL vers laquelle l'utilisateur va être redirigé (voir le point suivant pour la manière de construire cette URL dans ce cas).
- Comme le commentaire Python l'indique, vous devez systématiquement renvoyer une `HttpResponseRedirect` après avoir correctement traité les données POST. Ceci n'est pas valable uniquement avec Django, c'est une bonne pratique du développement Web.
- Dans cet exemple, nous utilisons la fonction `reverse()` dans le constructeur de `HttpResponseRedirect`. Cette fonction nous évite de coder en dur une URL dans une vue. On lui donne en paramètre la vue vers laquelle nous voulons rediriger ainsi que la partie variable de l'URL qui pointe vers cette vue. Dans ce cas, en utilisant l'URLconf défini précédemment, l'appel de la fonction `reverse()` va renvoyer la chaîne de caractères `developer/`. Cette URL de redirection va ensuite appeler la vue `index` pour afficher la liste des développeurs.

Les classes formulaires

Nous allons simplifier les étapes de créations de formulaire grâce aux classes formulaires.

Dans le dossier `Developer`, ajoutez un fichier `forms.py`. Dans celui-ci ajoutez le code suivant :

Créez un formulaire

`developer/forms.py`

```
from django import forms

from .models import Developer

class DeveloperForm(forms.Form):
    first_name = forms.CharField(label="First name", max_length=100)
    last_name = forms.CharField(max_length=100)
```

Nous définissons ainsi une nouvelle classe `DeveloperForm`. Celles-ci possède les deux mêmes champs que le modèle associé.

Ajoutez le formulaire au gabarit

Nous allons maintenant modifier le gabarit afin que celui-ci affiche le formulaire. Enlevez tout ce qui a trait aux champs et ajoutez `{{ form }}`.

 Vous pouvez mettre le formulaire en forme de différente façon.

- `{{ form.as_table }}`
- `{{ form.as_p }}`

- `{{ form.as_ul }}`

developer/index.html

```
<form action="{% url 'developer:create' %}" method="post">
    {% csrf_token %}

    <!--<label for="first_name">First name</label>
    <input type="text" name="first_name" required>
    <label for="last_name">Last name</label>
    <input type="text" name="last_name" required>-->
    {{ form }}
    <button type="submit">Create</button>
</form>
```

Envoyez le formulaire au gabarit

Le gabarit n'est évidemment pas en mesure de deviner quel formulaire il doit afficher. Il est de la responsabilité de la vue d'ajouter le formulaire au contexte.

developer/views.py

```
#...
from .forms import DeveloperForm

def index(request):
    context = {
        'developers': Developer.objects.all(),
        'form': DeveloperForm,          ➡ new
    }

    return render(request, 'developer/index.html', context)
#...
```

Valider le formulaire

Nous allons maintenant utiliser ce formulaire afin d'obtenir les données saisies par l'utilisateur.

developer/views.py

```
#...
def create(request):
    form = DeveloperForm(request.POST)          ➡ new

    if form.is_valid():                          ➡ new
        Developer.objects.create(               ➡ new
            first_name=form.cleaned_data['first_name'], ➡ new
            last_name=form.cleaned_data['last_name']    ➡ new
        )                                             ➡ new
        # Toujours renvoyer une HttpResponseRedirect après avoir géré correctement
        # les données de la requête POST. Cela empêche les données d'être postées
        # deux fois si l'utilisateur clique sur le bouton précédent.
        return HttpResponseRedirect(reverse('developer:index'))
#...
```

Notez que nous n'utilisons plus l'instruction `request.POST['xxx']` pour récupérer la donnée associée à un champ, mais `form.cleaned_data['first_name']`. Cela a plusieurs impacts.

1. Il est nécessaire de demander la validité (`is_valid` du formulaire avant d'obtenir une donnée *nettoyée*.)
2. Une donnée nettoyée n'est pas nécessairement un string. Ainsi, pour un champ de type `IntegerField`, la donnée retournée sera de type entier.

Un formulaire DRY

Vous l'avez peut-être remarqué, mais dans le modèle, les champs avaient une longueur de 200 caractères. Dans le formulaire de 100. Ce type d'incohérence apparaît lorsque nous oublions le principe DRY.

Django a prévu une meilleure manière de procéder afin de créer un formulaire sur base d'un modèle.

developer/forms.py

```
from django import forms

from .models import Developer

class DeveloperForm(forms.ModelForm):
    # first_name = forms.CharField(label="First name", max_length=100)
    old
    # last_name = forms.CharField(label='Last name', max_length=100)
    old
    class Meta:
        model = Developer
        fields = ['first_name', 'last_name']
```

Et voilà, nous avons un formulaire basé sur le modèle `Developer`. Et surtout, nous respectons le principe DRY !

Vues génériques

En Django, il y a les vues fonctions que nous avons déjà vues, mais il y a aussi les vues génériques (vues classes). Ces dernières, plus récentes, ont pour objectif de simplifier davantage la création de vue.

De base, ces vues sont utilisées pour les cas classiques du développement Web : récupérer les données depuis la base de données suivant un paramètre contenu dans l'URL, charger un gabarit et renvoyer le gabarit interprété.

Les vues génériques permettent l'abstraction de pratiques communes, à un tel point que vous n'avez pas à écrire de code Python pour écrire une application.

Nous allons convertir notre application de gestion de projet pour qu'elle utilise le système de vues génériques. Nous pourrons ainsi supprimer une partie de notre code. Nous avons quelques pas à faire pour effectuer cette conversion. Nous allons :

- Convertir l'URLconf.
- Supprimer quelques anciennes vues désormais inutiles.
- Introduire de nouvelles vues basées sur les vues génériques de Django.

DevDetailView

Vue DevDetailView

Dans le fichier `developer.views.py`, crée la classe `DevDetailView` et supprimez la fonction `detail()`.

`developer.view.py`

```
class DevDetailView(DetailView):
    model = Developer
    template_name = 'developer/detail.html'

# def detail(request, developer_id):
#     #developer = Developer.objects.get(pk=developer_id)
#     developer = get_object_or_404(Developer, pk=developer_id)
#     return render(request, 'developer/detail.html', {'developer':
developer})
```

- Nous utilisons ici la vues générique : `DetailView`. Cette vue permet l'abstraction des concepts affichés une page détaillée pour un type particulier d'objet (ici `Developer`).
- Par défaut, la vue générique `DetailView` utilise un gabarit appelé `<nom app>/<nom modèle>_detail.html`. Dans notre cas, elle utiliserait le gabarit `"developer/developer_detail.html"`. L'attribut `template_name` est utilisé pour signifier à Django d'utiliser un nom de gabarit spécifique plutôt que le nom de gabarit par défaut. Dans notre cas, nous avons choisi de renommer le template, mais cela n'était pas obligatoire. En revanche, cela le devient si vous devez afficher de deux manières différentes un même modèle.
- Dans les parties précédentes de ce tutoriel, le template `detail.html` a été renseigné avec un contexte qui contenait la variable de contexte `developer`. Pour `DetailView`, la variable `developer` est fournie automatiquement ; comme nous utilisons un modèle nommé `Developer`, Django sait donner un nom approprié à la variable de contexte.

La vue générique `DetailView` s'attend à ce que la clé primaire capturée dans l'URL s'appelle "pk", nous allons donc changer `developer_id` en `pk` pour la vue générique.

URL DevDetailView

Nous l'avons vu, `path` prend en deuxième paramètre une fonction vue. La transformation de la vue générique vers une vue fonction se fait grâce à l'appel à la méthode `as_view()`.

`developer.urls.py`

```
from .views import DevDetailView
#...
urlpatterns = [
    path('', views.index, name='index'),
    #path('<int:developer_id>', views.detail, name='detail'),    ➡ old
    path('<int:pk>', DevDetailView.as_view(), name='detail'),    ➡ new
    path('create', views.create, name='create'),
]
```

⚠ Pourquoi y a-t-il les parenthèses après `DevDetailView.as_view()` et pas après `views.detail` ?

Le deuxième paramètre de `path` est une fonction qui sera appelée lorsque l'url coïncidera avec le premier paramètre. Ainsi, dans le cas d'une vue fonction, nous donnons en paramètre la fonction `detail`. Dans le cas d'une vue générique, nous donnons en paramètre le retour de la fonction `as_view()` qui est une fonction !

IndexView

Cela va se compliquer un petit peu. En effet, nous avons une liste des développeurs un peu particulière puisqu'elle est suivie d'un formulaire de création. Nous allons procéder par étape afin de rendre l'implémentation de la classe générique aussi claire que possible.

Vue IndexView

Commençons par créer notre classe comme si nous n'avions pas de formulaire.

`developer.view.py`

```
from django.views.generic import DetailView, ListView ➡ On ajoute ListView

#...

class IndexView(ListView):                                ➡ new
    model = Developer                                     ➡ new
    template_name = "developer/index.html"                ➡ new
    context_object_name = 'developers'                     ➡ new
```

- Nous créons une nouvelle classe qui hérite de `ListView`.
- Nous indiquons que la vue est faite pour le modèle `Developer`.
- À l'instar d'une `DetailView` un nom de template est généré automatiquement. Dans le cas d'une `ListView`, le nom généré automatiquement est le suivant : `<nom_app>/<nom_modèle><suffixe_gabarit>.html`. Étant donné que le suffixe par défaut est `_list`, nous aurions pour notre modèle : `developer/developer_list.html`. Nous modifions ce nom de template afin qu'il corresponde à ce qu'il y a déjà dans le gabarit (ce changement pourrait aussi se faire au niveau du gabarit).
- Nous modifions également le nom de la variable du contexte qui contient la liste des développeur. Par défaut, celle-ci aurait pour nom : `developer_list`.

Ajout du formulaire à IndexView

Nous avançons, mais nous n'avons pas ajouté notre formulaire dans le contexte. Pour cela, il est nécessaire de réécrire la méthode `get_context_data()`.

```

from django.views.generic import DetailView, ListView

#...

class IndexView(ListView):
    model = Developer
    template_name = "developer/index.html"
    context_object_name = 'developers'

    def get_context_data(self, **kwargs):
        context = super(IndexView, self).get_context_data(**kwargs)
        context['form'] = DeveloperForm
        return context

```

👉 new
👉 new
👉 new
👉 new

Parenthèse Python 🐍

En ☕ Java, nous écrivons `super.getContextData(...)` afin d'appeler la fonction de la classe mère. En python, il est nécessaire de donner la classe en premier paramètre et `self` en second paramètre.

Ainsi,

1. Nous chargeons dans la variable `context` le contexte tel qu'il était défini précédemment, c'est-à-dire contenant la variable `developers`.
2. Ensuite, nous ajoutons une clé `form` et le formulaire à utiliser `DeveloperForm`. Nous retournons ensuite le contexte définit au sein de la variable `context`.

URL IndexView

Il est maintenant temps d'associer une url à notre nouvelle classe vue. Rien de plus simple, vous avez fait quelque chose de très similaire avec `DevDetailView`.

```

developer.urls.py

from .views import DevDetailView
#...
urlpatterns = [
    #path('', views.index, name='index'),           👉 old
    path('', IndexView.as_view(), name='index'),    👉 new
    path('<int:pk>', DevDetailView.as_view(), name='detail'),
    path('create', views.create, name='create'),
]

```

Vue générique et Mixin

C'est bien gentil tout ça, mais tout cela m'a l'air bien compliqué et je ne sais pas où vous avez été cherché l'information. 🤔

En réalité, tout cela est relativement simple. Surtout quand on sait où chercher l'information.

Les vues génériques sont basées sur le principe de Mixin. Wikipédia le définit assez simplement ce principe de la manière suivante

Concept de Mixin

"En programmation orientée objet, un mixin ou une classe mixin est une classe destinée à être composée par héritage multiple avec une autre classe pour lui apporter des fonctionnalités. C'est un cas de réutilisation d'implémentation. Chaque mixin représente un

service qu'il est possible de greffer aux classes héritières. "

[Wikipédia](#)

Maintenant que vous savez ce que c'est, il vous suffit de savoir quelle fonctionnalité est greffée à votre classe. Vous trouverez ces informations dans la documentation. Par exemple, pour `DetailView` vous trouverez les Mixin utilisés et donc les fonctionnalités [ici](#).

Quelques exemples :

- Comment je sais que le template par défaut pour une classe qui hérite de `ListView` est `<nom_app>/<nom_modèle><suffixe_gabarit>.html` ? Cela est ajouté grâce au [MultipleObjectTemplateResponseMixin](#) dont hérite la classe `ListView`.
- Et pour `get_context_data()` ? Alors là c'est dans [ContextMixin](#)

Nous vous recommandons également d'aller jeter un coup d'oeil de temps à autre dans le code des vues génériques. Il vous apprendra beaucoup sur leur fonctionnement. Il se trouve [ici](#).

Modal, include et Crispy

Le modal

On peut imaginer que pour une utilisation normale de l'application, l'ajout d'un développeur se fait de manière occasionnelle.

Nous allons donc vous proposer de mettre ce formulaire au sein d'un *Modal*.

Un modal est une sorte de boîte de dialogue qui est affichée devant la page courante lorsqu'un évènement survient ou que l'utilisateur en fait la demande.

`developer/index.html`

```
<!--<form action="{% url 'developer:create' %}" method="post">      ➡old
    {% csrf_token %}      ➡old
    {{ form }}      ➡old
    <button type="submit">Create</button>      ➡old
</form>      ➡old
-->

<!-- Ajout d'un bouton pour faire apparaître la boîte de dialogue ➡ début
du nouveau block -->
<button type="button" class="btn btn-primary" data-toggle="modal" data-
target="#add-dev-modal">Add user</button>

<!-- Ajout du modal contenant le formulaire -->
<div class="modal fade " id="add-dev-modal">
    <div class="modal-dialog modal-dialog-centered">
        <div class="modal-content">
            <div class="modal-header">
                <h4 class="modal-title">New developer</h4>
                <button type="button" class="close" data-dismiss="modal"><i
class="fa fa-close"></i></button>
            </div>
            <div class="modal-body">
                <form action="{% url 'developer:create' %}" method="post">
                    {% csrf_token %}
                    {{ form }}
                </div>
```

```

        <button class="btn btn-primary"
type="submit">Créer</button>
    </div>
</form>
</div>
</div>
</div>
</div> <!-- 🖱️ fin de l'ajout -->

```

Modal et respect du Dry

Il est fort probable que l'ajout d'un utilisateur puisse se faire à partir de plusieurs fenêtre. Et même si cela ne se produit pas dans ce projet, il reste de bonne pratique de le supposer.

Nous allons donc extraire celui-ci afin de pouvoir le réutiliser plus tard, dans un autre template, si besoin en est.

Dans le dossier `developper/templates`, ajoutez un nouveau template `_create_dev_modal.html`. Comme discuté précédemment, le nom de notre nouveau template commence par un `_`. En effet, celui-ci ne sera jamais utilisé indépendamment d'un autre template.

Copiez-y tout le code que vous venez d'ajouter dans le fichier `developper/index.html`.

`developper/_create_dev_modal.html`

```

<!-- Ajout d'un bouton pour faire apparaître la boîte de dialogue -->
<button type="button" class="btn btn-primary" data-toggle="modal" data-
target="#add-dev-modal">Add user</button>

<!-- Ajout du modal contenant le formulaire -->
<div class="modal fade " id="add-dev-modal">
    <div class="modal-dialog modal-dialog-centered">
        <div class="modal-content">
            <div class="modal-header">
                <h4 class="modal-title">New developer</h4>
                <button type="button" class="close" data-dismiss="modal"><i
class="fa fa-close"></i></button>
            </div>
            <div class="modal-body">
                <form action="{% url 'developer:create' %}" method="post">
                    {% csrf_token %}
                    {{ form }}
                    <div>
                        <button class="btn btn-primary"
type="submit">Créer</button>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>

```

Dans `developper/index.html`, remplacez tout ce code par l'inclusion du fichier `developper/_create_dev_modal.html`. L'inclusion se fait grâce à la balise `{% include '<nom du fichier>' %}`.

`developper/index.html`


```
#...
</ul>
{% else %}
    <p><strong>Il n'y a aucune développeur enregistré !</strong>/p>
{% endif %}

<!-- bloc modal --> 🖱️ old
{% include 'developer/_create_dev_modal.html' %} 🖱️ new
{% endblock content %}
```

Crispy

Si vous avez bien suivi le tutoriel jusqu'à maintenant, vous avez peut-être choisi d'utiliser `{{ form.as_p }}` ou autre pour avoir un formulaire un peu plus joli.

Dans Django, il est possible d'ajouter facilement des apps externes. Nous allons illustrer cela par l'ajout d'une app nommée Crispy. Elle permet de rendre un peu plus joli les formulaires.

1. Installez le module. Pour cela, saisissez la commande `python -m pip install django-crispy-forms`
2. Ajoutez django-crispy-forms aux applications installées

settings.py

```
INSTALLED_APPS = [
    #...
    'django.contrib.staticfiles',

    #My apps
    'developer.apps.DeveloperConfig',

    #Third-party app 🖱️ new
    'crispy_forms', 🖱️ new
]
```

3. Configuré le pack à utiliser en ajoutant la variable `CRISPY_TEMPLATE_PACK` au fichier settings.py.

```
# CRISPY FORM CONFIGURATION
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

4. Modifiez `{{ form }}` ou `{{ form.as_qqc }}` par `{{ form|crispy }}` et enfin, chargez le tag crispy dans votre template formulaire. Cela se fait grâce à la balise `{% load %}`.

developer/_create_dev_modal.html

```
{% load crispy_forms_tags %} 🖱️ new

<button type="button" class="btn btn-primary" data-toggle="modal" data-target="#add-dev-modal">Add user</button>

#...

    <!-- {{ form.as_p }} --> 🖱️ old
    {{ form|crispy }} 🖱️ new

#...
```

Vous venez de terminer de rendre votre formulaire propre et réutilisable.

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 instanciation.

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

1. `python manage.py makemigrations`
2. `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">
    new
    {% csrf_token %}
    new
    <button type="submit" class="close"><i class="fa fa-trash"></i>
  </button> new
  </form>
  new
  <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 `developeurs`. 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 créée, 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`.

Gestion des utilisateurs

Vous voici presque à la fin de ce très long exercice. Nous allons maintenant gérer les utilisateurs.

Dans cette section, nous allons voir

- la gestion de la page d'administration ;
- la gestion de l'authentification ;
- la gestion des permissions.

Commençons avec la page d'administration et le super utilisateur.

Page d'administration et super utilisateur

Vous vous êtes sûrement demandé à quoi servait les fichiers `admin.py` ? Celui-ci permet de configurer la page d'administration de notre projet. Nous aurions pu le décrire plus tôt, mais vous auriez trop vite délaissé le `shell`.

Nous allons donc commencer par créer un super utilisateur. Saisissez la commande suivante et remplissez le formulaire.

```
$ python manage.py createsuperuser
```

Redémarrez le serveur et rendez-vous sur la page `localhost:8000/admin` ; connectez-vous et explorez un peu.

Maintenant que vous avez découvert la page d'administration, nous allons configurer les fichiers `admin.py` de application `developer` et `task`.

Modifiez les fichiers `admin.py` avec les codes respectifs ci-dessous.

`developer/admin.py`

```
from .models import Developer

admin.site.register(Developer)
```

`task/admin.py`

```
from .models import Task

admin.site.register(Task)
```

Retournez sur la page administration et vérifiez l'ajout de l'administration des développeurs et des tâches.

À noter ici :

- Le formulaire est généré automatiquement à partir des modèles `Developer` et `Task`.
- Les différents types de champs du modèle (`DateTimeField`, `CharField`) correspondent au composant graphique d'entrée HTML approprié. Chaque type de champ sait comment s'afficher dans l'interface d'administration de Django.

La partie inférieure de la page vous propose une série d'opérations :

- Enregistrer – Enregistre les modifications et retourne à la page liste pour modification de ce type d'objet.
- Enregistrer et continuer les modifications – Enregistre les modifications et recharge la page d'administration de cet objet.
- Enregistrer et ajouter un nouveau – Enregistre les modifications et charge un nouveau formulaire vierge pour ce type d'objet.
- Supprimer – Affiche une page de confirmation de la suppression.

(Enfin, ça c'est si vous avez configuré votre page en français. Si vous ne l'avez pas fait, vous pouvez le faire. Cela se passe dans le fichier `settings.py`. Modifiez le champ `LANGUAGE_CODE = 'fr'`. Vous trouverez l'ensemble des langues supportées [ici](#).)

Configuration de la page admin.

Nous n'allons pas rentrer dans les détails ici, juste vous proposer deux modifications appropriées.

La première est de rendre visible les tâches dans la vue développeur.

Modifiez le code comme ceci :

```
class TaskInline(admin.TabularInline):
    model = Task
    extra = 1

class DeveloperAdmin(admin.ModelAdmin):
    inlines = [TaskInline]

admin.site.register(Developer, DeveloperAdmin)
```

Enfin, lorsque vous affichez la liste des développeurs ou des tâches, vous voyez ce qui a été défini dans la méthode `__str__()`. C'est bien, mais on peut faire mieux.

Modifier les fichiers afin de détailler les champs que vous souhaitez lister.

task/admin.py

```
from .models import Task

class TaskAdmin(admin.ModelAdmin):
    list_display = ('title', 'description')

admin.site.register(Task, TaskAdmin)
```

et

developer/admin.py

```
class DeveloperAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'is_free')
    inlines = [TaskInline]
```

Si vous êtes attentif, vous avez remarqué que `is_free` n'est pas un champ, mais une méthode. Cela fonctionne aussi.

Vous pouvez également améliorer l'affichage en ajoutant quelques attributs à votre méthode

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}"

is_free.boolean = True 📌new
is_free.short_description = 'Is free' 📌new
```

⚠ Vous n'avez pas besoin de réaliser une migration pour cette étape, même si vous touchez au modèle.

Nous avons modifié le minimum de la page d'administration, mais vous pouvez configurer davantage votre page d'administration. Voici un peu de lecture

- [Tutoriel admin](#) 🧐
- [Doc admin](#) 🧐
- [Action admin](#) 🧐

Faire des développeurs, des users

Nous allons maintenant modifier le modèle développeur afin que ceux-ci soient des utilisateurs.

Si vous cherchez par vous-même de la documentation, vous risquez de tomber sur l'ancienne approche qui consistait à réaliser un lien OneToOne qui était appelé "modèle de profile". Ça c'était avant, nous allons voir la technique la plus simple parmi les deux techniques les plus récentes.

En effet, il existe deux possibilités. Soit étendre le modèle `AbstractUser`, soit étendre le modèle `AbstractBaseUser`. Nous allons plutôt choisir la première possibilité, car même si le second permet beaucoup plus de flexibilité, la première méthode est plus simple et permet un changement postérieur si les besoins changent.

Nous allons suivre ces 4 étapes :

1. Créer un modèle "CustomUser".
2. Mettre à jour le fichier `settings.py`.
3. Adapter `UserCreationForm` et `UserChangeForm`.
4. Ajouter le nouveau modèle à `admin.py`.

⚠ Avant de continuer, supprimer toute migration de votre db

1. `python manage.py migrate zero`
2. suppression des fichiers de migration

Configuration du modèle d'utilisateur

developer/models



```
from django.contrib.auth.models import AbstractUser 📌new
from django.db import models

#class Developer(models.Model): 📌old
class Developer(AbstractUser): 📌new
```

Nous allons maintenant ajouter un paramètre `AUTH_USER_MODEL` au bas de notre fichier de configuration afin de demander à notre projet d'utiliser notre modèle plutôt que le modèle d'utilisateur par défaut.

settings.py

```
# CRISPY FORM CONFIGURATION
CRISPY_TEMPLATE_PACK = 'bootstrap4'

# AUHT CONFIGURATION 
AUTH_USER_MODEL = 'developer.Developer' 
```

Formulaire pour le modèle d'utilisateur

On ne va pas y aller par 4 chemins, ici nous allons remplacer notre formulaire, modifier la classe dont elle hérite, et ajouter un formulaire pour permettre la modification de notre utilisateur.

developer/forms.py

```
from django.contrib.auth import get_user_model
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from django import forms
from .models import Developer

class DeveloperForm(UserCreationForm):
    class Meta:
        model = get_user_model()
        fields = ('first_name', 'last_name', 'username', 'email',)

class DeveloperChangeForm(UserChangeForm):
    class Meta:
        model = get_user_model()
        fields = ('first_name', 'last_name', 'username', 'email')
```



Modification Admin de l'utilisateur


Nous allons maintenant modifier le fichier `developer/admin.py` afin de modifier les formulaires de création et de modification.

Pour cela, il est nécessaire de modifier les champs `add_form` et `form` hérité de la classe `UserAdmin`.

La méthode `get_user_model` permet d'importer le modèle d'utilisateur précédemment configuré dans le fichier `settings.py`.

developer/admin.py

```
from django.contrib.auth import get_user_model 
from django.contrib.auth.admin import UserAdmin 
from django.contrib import admin

from .forms import DeveloperForm, DeveloperChangeForm 
from .models import Developer
from task.models import Task
```



```

class TaskInline(admin.TabularInline):
    model = Task
    extra = 1

#class DeveloperAdmin(admin.ModelAdmin):
class DeveloperAdmin(UserAdmin):
    add_form = DeveloperForm
    form = DeveloperChangeForm
    model = get_user_model()
    list_display = ('first_name', 'last_name', 'username', 'is_free')
    'username'
    inlines = [TaskInline]

```

Adaptation pour la commande createsuperuser

Enfin, un super utilisateur est un utilisateur comme les autres. Il est donc important d'ajouter le prénom et le nom d'un super utilisateur lors de sa création (appel à `createsuperuser`).

developer/models.py

```

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

    REQUIRED_FIELDS=['first_name', 'last_name']
    def is_free(self):
        return self.tasks.count() == 0
#...

```

Faites la migration et testez la création d'un nouveau super utilisateur. Vous devriez avoir le nom et prénom qui fait dorénavant partie des données requises.

Correction du formulaire de création de développeur

Si vous essayez de créer un développeur, vous verrez que de nombreux champs se sont ajoutés au formulaire. Essayez ! ★ Ceci est tout à fait normal puisqu'un développeur est un utilisateur. Les champs username ; email ; password... sont ainsi demandés. Nous allons donner un autre formulaire afin de créer un développeur avec les données minimales à leur gestion.

Ajoutez un formulaire simplifié lié directement au modèle développeur.

developer/forms.py

```

#...
class ShortDeveloperForm(forms.ModelForm):
    class Meta:
        model = Developer
        fields = ['first_name', 'last_name', 'username']
#...

```

Et enfin, n'oubliez pas que ce formulaire est traité lors de la création d'un développeur et envoyé lorsque la vue `index` des développeurs est demandée. Modifiez la vue afin que le champs `username` soit également considéré.

developer/views.py

```
#...
#from .forms import DeveloperForm
from .forms import ShortDeveloperForm
#...
class IndexView(ListView):
    model = Developer
    template_name = "developer/index.html"
    context_object_name = 'developers'

    def get_context_data(self, **kwargs):
        context = super(IndexView, self).get_context_data(**kwargs)
        #context['form'] = DeveloperForm           🐞old
        context['form'] = ShortDeveloperForm       🐞new

#...

def create(request):
    form = ShortDeveloperForm(request.POST)

    if form.is_valid():
        Developer.objects.create(
            first_name=form.cleaned_data['first_name'],
            last_name=form.cleaned_data['last_name'],
            username=form.cleaned_data['username'], 🐞new
        )
#...
```

Testez la création d'un développeur au sein de votre projet (sans passer par l'interface d'administration). Vérifiez la création de l'utilisateur. Si vous voulez, vous pouvez ajouter un mot de passe à ce nouveau développeur.

D'un point de vue utilisateur, votre projet fonctionne comme avant (exception faite du champ `username`). Mais nous pouvons maintenant passer à la gestion de la connexion et des permissions.



Dans la vue, nous avons laissé la création d'un développeur grâce à la fonction `Developer.objects.create(...)`. Si vous souhaitez créer un utilisateur avec tous les champs requis, il est alors nécessaire d'utiliser la fonction `Developer.objects.create_user(...)`.

Demande d'authentification

Maintenant que nous avons la capacité de créer des utilisateurs. Nous allons limiter l'accès au site aux utilisateurs connectés et limiter l'accès à certaines fonctionnalités aux utilisateurs qui en ont le droit.

Voici donc la suite du programme :

1. Nous allons ajouter une page d'accueil qui va afficher les tâches de l'utilisateur connecté et permettre à l'utilisateur de se connecter s'il ne l'est pas.
2. Nous allons empêcher l'accès aux vues index des applications `developer` et `task` aux utilisateurs non connecté.
3. Nous allons ajouter des permissions au niveau de la gestion des tâches afin que certains utilisateurs ne puissent voir que celles qui sont attribuées à travers la description des développeurs, mais pas au travers de la liste des tâches.

Une app pour les gouverner toutes

Pour le moment, `localhost:8000` ne mène vers aucune vue. C'est-à-dire que nous n'avons pas de page d'accueil. Nous allons créer cette page d'accueil. Celle-ci va être placée dans une application.


Créez l'application `home`, ajoutez celle-ci dans le fichier `settings.py`.

Ajoutez une vue dans cette nouvelle application qui va pointer vers un template qui aura pour simple rôle de dire bonjour.

`home/views.py`

```
from django.views.generic import TemplateView

class HomeView(TemplateView):
    template_name = "home/index.html"
```

 `TemplateView` est une vue très basique qui a pour seul devoir d'afficher un gabarit donné.

Ajoutez le template associé à cette vue

`home/templates/home/index.html`

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

{% block content %}
<h1>Coucou vous !</h1>
{% endblock content%}
```

Ajoutez les urls de cette application à celle du projet (comme vous avez fait pour les deux autres applications).

`mproject/urls.py`

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('developer/', include('developer.urls')),
    path('task/', include('task.urls')),
    path('', include('home.urls')), 🐞 new
]
```

Et enfin, ajoutez un chemin dans le fichier `urls.py` de votre applicaton vers cette nouvelle vue.

`home/urls.py`

```
from django.urls import path

from . import views

urlpatterns = [
    path('', views.HomeView.as_view(), name='home'),
]
```

Lancez le serveur et testez ! 🌟

Personnalisation de l'accueil

Actuellement, le message est assez impersonnel. Modifiez le template afin qu'il corresponde au code ci-dessous.

home/templates/home/index.html

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

{% block content %}
{% if user.is_authenticated %}
    <h1>Coucou {{ user.first_name }}</h1>
{% else %}
    <h1>Coucou toi !</h1>
{% endif %}
{% endblock content %}
```

Relancez le serveur si vous l'avez arrêté et testé à nouveau. Si vous avez le message "Coucou toi" alors rendez-vous dans la page admin pour vous y connecter. Si vous avez comme message "Coucou <votre prénom>" alors rendez-vous sur la page admin pour vous déconnecter.

Remarquez que la variable de context `user` est automatiquement ajouté par Django dans tout affichage de gabarit.

★ Le lien "home" (🏠) ne fonctionne pas. Corrigez-le.

Affichage des tâches

Modifiez le template afin d'afficher les tâches, le nom et le prénom de l'utilisateur connecté.

Exemple de code

home/template/home/index.html

```
#...
{% if user.is_authenticated %}
<div class="p-1 m-3 bg-light">
    <h1>
        {{ user.first_name }} {{ user.last_name }}
    </h1>
</div>

<div class="container-sm">
    {% if user.tasks.all|length %}
        <ul class="list-group fluid">
            {% for task in user.tasks.all %}
                <li class="list-group-item">
                    <strong>{{ task.title }}</strong>
                    {{ task.description }}
                </li>
            {% endfor %}
        </ul>
    {% else %}
        <alert class="alert alert-warning">No Tasks</alert>
    {% endif %}
</div>
{% else %}
    <h1>Coucou toi !</h1>
```

```
#...
```

Ajoutez une tâche à votre utilisateur pour vérifier le bon fonctionnement. S'il y en avait déjà une, supprimez la.

📖 Si vous avez été attentif jusqu'à maintenant, et si vous avez bien fait tout ce qui est demandé nous nous attendons à une question équivalente à celle-ci : "Je pensais que seuls les superutilisateurs peuvent se connecter. L'utilisateur que nous avons fait précédemment ne peut pas se connecter". En effet, seuls un utilisateur ("du staff") peut se connecter à notre page admin. Mais tout utilisateur peut se connecter à notre site. Dans les fait, c'est vrai si on lui en donne l'occasion. C'est ce que nous allons faire dans la prochaine section.

Urls d'authentications

Connection

La première chose à faire est de définir les urls qui vont mener aux vues d'authentification.

```
mproject/urls.py
```

```
urlpatterns = [
    path('', include('home.urls')),
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')), ➡ new
    path('developer/', include('developer.urls')),
    path('task/', include('task.urls')),
```

Et ajoutez un lien pour se connecter.

```
templates/home/index.html
```

```
{% else %}
<h1>Coucou toi !</h1>
<a href={% url 'login' %}>Log in</a> ➡ new
{% endif %}
```

📖 L'url "login" fait partie des urls incluse à l'étape précédente.

🐰 Cliquez pour vous connecter ! Et prenez 5 minutes pour lire l'erreur obtenue. Cette erreur est simple, le template permettant de se connecter n'a pas été trouvé. Django s'attend en effet à le trouver dans le répertoire `registration` et celui-ci doit s'appeler `login.html`.

Ajoutez donc le fichier suivant dans le dossier `templates` de votre projet avec l'arborescence attendue.

```
registration/login.html
```

```
{% extends "_base.html" %}
{% load crispy_forms_tags %}
{% block title %}GProject - login{% endblock title %}

{% block content %}
<div class="container-sm p-3 mt-2 bg-light text-primary">
<h2>Log in</h2>
<form method="post">
```

```
{% csrf_token %}
{{ form|crispy }}
<button type="submit" class="btn btn-dark">Log in</button>
</form>
</div>
{% endblock content %}
```

(Ce gabarit devrait vous sembler naturel maintenant.)

Essayez à nouveau de vous connecter avec les identifiants d'un développeur créé dans votre projet (en dehors de la page admin) et pour lequel vous avez défini un mot de passe ★.

Une nouvelle fois, vous rencontrez une page d'erreur. Moins évidente cette fois. Pas de panique toutefois ! Après vous êtes connecté, Django cherche une page de profil qui n'existe pas. Et nous n'allons pas la créer ! Plutôt que de faire cela, nous allons rediriger l'utilisateur qui s'est connecté vers la page d'accueil de notre site.

Dans le fichier `settings.py`, ajoutez cette ligne à la fin.

```
LOGIN_REDIRECT_URL = 'home'
```

★ Essayez et profitez 😊

Déconnexion

Nous allons gérer la déconnexion maintenant, rassurez vous, le plus dur est fait. Pour cela nous devons

1. Ajouter un lien pour nous déconnecter
2. Rediriger l'utilisateur vers la page d'accueil.

```
templates/home/index.html
```

```
{% if user.is_authenticated %}
<div class="p-1 m-3 bg-light">
  <p class="float-right"><a href="{% url 'logout' %}"><i class="fa fa-sign-
out"></i></a></p> 🗑 new
  <h1>
    {{ user.first_name }} {{ user.last_name }}
  </h1>
</div>
```

Et on redirige en ajoutant cette variable à notre fichier `settings.py` :

```
LOGOUT_REDIRECT_URL = 'home'
```

★ Testez !

Limiter l'accès aux utilisateurs connectés

Nous allons empêcher l'accès aux utilisateurs non connectés aux pages d'index des développeurs et des tâches ainsi qu'à la page de détail d'un développeur.

Grâce aux mixins, c'est sans doute l'une des étapes les plus simples. Il suffit que les vues héritent du mixin `LoginRequiredMixin` pour que cela fonctionne comme souhaité.

```
developer/views.py
```

```
from django.contrib.auth.mixins import LoginRequiredMixin ➡ new
#...

class IndexView(LoginRequiredMixin, ListView):
#...

class DevDetailView(LoginRequiredMixin, DetailView):
#...
```

task/views.py

```
from django.contrib.auth.mixins import LoginRequiredMixin ➡ new
#...
class IndexView(LoginRequiredMixin, generic.ListView):
```

Remarques :

- Si l'utilisateur n'est pas connecté, alors il sera redirigé automatiquement vers la page de login.
- On place le mixin `LoginRequiredMixin` avant les autres mixins. En effet, les mixins sont chargés dans l'ordre donné. Si l'utilisateur n'est pas connecté, il est inutile de charger les autres mixins.

Permissions

Nous voici dans la dernière ligne droite. Nous allons ajouter le droit à certains utilisateurs de gérer les tâches.

Si vous avez prêté attention à la page admin des utilisateurs (développeurs), vous avez remarqué qu'il est possible d'ajouter des permissions prédéfinies. Nous allons créer notre propre permission et l'utiliser.

Cela se fait au niveau des modèles.

Modifiez le modèle `Task` comme ceci

task/models.py

```
class Task(models.Model):
#...
assignee = models.ForeignKey(Developer, related_name="tasks",
on_delete=models.CASCADE, null=True)

class Meta:
    permissions = [
        ('task_management', 'Can create, assign and delete tasks'),
    ]
```

Réalisez une migration. ★ Dans quelle table trouvez vous les différentes permissions ?

Utilisons à nouveau les mixins pour ajouter cette fonctionnalité. Celui qui nous intéresse est `PermissionRequiredMixin`.

Modifiez la vue index des tâches.

task/views.py

```

from django.contrib.auth.mixins import LoginRequiredMixin,
PermissionRequiredMixin ➡ ajout de PermissionRequiredMixin
#...
class IndexView(LoginRequiredMixin, PermissionRequiredMixin,
generic.ListView):
#...
permission_required = 'task.task_management' ➡ new

```

Notez qu'il est nécessaire d'ajouter le champ `permission_required` avec la permission qui est nécessaire.

📖 Encore une fois, prêtez attention à l'ordre de vos mixins !

Créez un utilisateur (non super utilisateur). Ajoutez un mot de passe et accordez lui cette permission.

Créez un autre utilisateur (non super utilisateur). Ne lui accordez pas cette permission.

Vérifiez le bon fonctionnement de votre projet.

Il reste à empêcher d'accéder au bouton d'ajout de tâche dans le détail d'un développeur. Pour cela rien de plus simple, il suffit d'utiliser la variable de contexte `perms`

`task/_create_task_modal.html`

```

{% load crispy_forms_tags %}
{% if perms.task.task_management %}
#...
{% endif %}

```

et le tour est joué !

Solutions

Suppression développeur

`developer/views.py`

```

def delete(request, developer_id):
    get_object_or_404(Developer, pk=developer_id).delete()

    return HttpResponseRedirect(reverse('developer:index'))

```

`developer/urls.py`

```

path('<int:developer_id>/delete', views.delete, name='delete'),

```

Il faut aussi modifier le modèle afin que la suppression d'un développeur n'implique pas la suppression de la tâche.

`developer/models.py`


```
assignee = models.ForeignKey(
    Developer,
    related_name="tasks",
    on_delete=models.SET_NULL,
    null=True,
    verbose_name="assignee")
```

⚠ Attention, il est nécessaire de réaliser une migration !

App Task - réponse

1. Création de l'app task `python manage.py startapp task`.
2. On ajoute cette application aux applications installées (dans `settings.py`).
3. On copie le modèle dans l'application task
4. On ajoute l'import de `Developer` : `from developer.models import Developer`.
5. On réinitialise la DB `python manage.py migrate developer zero`
6. On supprime l'historique de migration (fichier migrations)
7. On lance la commande `python manage.py makemigrations`
8. On réalise la migration `python manage.py migrate`

Ajout de la liste des tâches - réponse

1. Commençons par écrire un nouveau gabarit `task/index.html` au sein du dossier `templates` que l'on ajoute à l'application

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

{% block title %}MProject - tâches{% endblock title %}

{% block content %}
{% if tasks %}
<ul class="list-group">
  {% for task in tasks %}
    <li class="list-group-item">
      {{ task.title }}

      {% if task.assignee %}
        <span>(assignée à {{ task.assignee.first_name }})</span>
      {% else %}
        <span>(non assignée)</span>
      {% endif %}
    </li>
  {% endfor %}
</ul>
{% else %}
<div class="container m-4">
  <alert class="alert alert-warning">No Tasks</alert>
</div>
{% endif %}
{% endblock content %}
```

2. Ajoutons une nouvelle vue dans `task.views.py`

```

from django.shortcuts import render
from django.views import generic

from .models import Task

class IndexView(generic.ListView):
    model = Task
    template_name = "task/index.html"
    context_object_name = 'tasks'

```

3. Ajoutons une nouvelle route vers cette vue.

- Dans l'application `task`, ajoutons le dossier `urls.py`

```

from django.urls import path

from .views import IndexView

app_name = 'task'
urlpatterns = [
    path('', IndexView.as_view(), name='index'),
]

```

- Faisons un lien vers ce urlpatterns dans celui du projet (`mproject/urls.py`)

```

path('task/', include('task.urls')),

```

4. Ajoutons un lien vers cette vue dans le template de base (`BASE_DIR/templates/_base.html`)

```

<li id="nav-task" class="nav-item">
  <a class="nav-link" href="{% url 'task:index' %}">Tasks</a>
</li>

```

Activer le bon lien - solution

`tempaltes/developer/_base.html`

```

{% extends "_base.html" %}

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

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

```

`tempaltes/developer/index.html`

```

{% extends "_base.html" %}
{% extends "developer/_base.html" %}

```

👉old

👉new

Suppression d'une tâche - solution

On ajoute une vue qui permet de supprimer une tâches.

task/views.py

```
#...
def delete(request, task_id):
    task = get_object_or_404(Task, pk=task_id)
    task.delete()

    return HttpResponseRedirect(reverse('task:index'))
```

On ajoute un chemin vers cette vue.

task/views.py

```
urlpatterns = [
    path('', IndexView.as_view(), name='index'),
    path('<int:task_id>/delete', views.delete, name='delete'), ➡ new
]
```

Création d'une tâche - solution

On ajoute un formulaire permettant la création d'une tâche

task/forms.py

```
from django import forms
from developer.models import Developer

class TaskForm(forms.Form):
    title = forms.CharField(label='Title', max_length=100)
    description = forms.CharField(label='Description', widget=forms.Textarea)
    assignee = forms.ModelChoiceField(queryset=Developer.objects.all(),
    required=False)
```

Ou mieux, avec un `ModelForm` 😊

task/forms.py

```
from django import forms
from .models import Task

class TaskForm(forms.ModelForm):
    class Meta:
        model = Task
        fields = ['title', 'description', 'assignee']
```

On ajoute le formulaire dans la vue d'index ; et tant qu'on est dans les vues, on ajoute une vue qui va nous permettre de valider et traiter le formulaire.

task/views.py

```
from .forms import TaskForm ➡ new
```

```

class IndexView(generic.ListView):
    model = Task
    template_name = "task/index.html"
    context_object_name = 'tasks'

    def get_context_data(self, **kwargs):
        context = super(IndexView, self).get_context_data(**kwargs)
        context['form'] = TaskForm
        return context
#...
def create(request):
    form = TaskForm(request.POST)

    if form.is_valid():
        Task.objects.create(
            title=form.cleaned_data['title'],
            description=form.cleaned_data['description'],
            assignee=form.cleaned_data['assignee'])

    return HttpResponseRedirect(reverse('task:index'))

```

👉 new
👉 new
👉 new
👉 new

On modifie le gabarit, ou plutôt, on crée un gabari que le va importer 😊

`templates/task/_create_task_modal.html

```

{% load crispy_forms_tags %}

<button type="button" class="btn btn-primary" data-toggle="modal" > data-
target="#add-dev-modal">Ajouter une tâche</button>

<div class="modal fade " id="add-dev-modal">
  <div class="modal-dialog modal-dialog-centered">
    <div class="modal-content">
      <div class="modal-header">
        <h4 class="modal-title">Nouvelle tâche</h4>
        <button type="button" class="close" data-dismiss="modal"><i
class="fa fa-close"></i></button>
      </div>
      <div class="modal-body">
        <form action="{% url 'task:create' %}" method="post">
          {% csrf_token %}
          {{ form|crispy }}
          <div>
            <button class="btn btn-primary"
type="submit">Créer</button>
          </div>
        </form>
      </div>
    </div>
  </div>
</div>

```

On inclut ce gabari dans le gabari d'index des tâches.

`templates/task/index.html

```

<div class="container m-4">
  <alert class="alert alert-warning">No Tasks</alert>
</div>
{% endif %}
{% include 'task/_create_task_modal.html' %}
{% endblock content %}

```

On ajoute un chemin vers la vue de création de tâche.

task/urls.py

```

app_name = 'task'
urlpatterns = [
    path('', IndexView.as_view(), name='index'),
    path('<int:task_id>/delete', views.delete, name='delete'),
    path('create', views.create, name="create"),
]

```

On fait une petite prière au dieu du copy/past 🙏

Et on lance le serveur pour tester.

Dans le détail d'un développeur - solution

On inclut le formulaire

developpeur/detail.html

```

#...
    </div>
    {% endif %}
    {% include 'task/_create_task_modal.html' %} 🙏new
</div>
{% endblock content %}

```

Dans la vue détail, on ajoute le formulaire dans le contexte.

developpeur/views.py

```

from task.forms import TaskForm
🙏new
#...
class DevDetailView(DetailView):
    model = Developer
    template_name = 'developer/detail.html'

    def get_context_data(self, **kwargs):
        🙏new
        context = super(DetailView, self).get_context_data(**kwargs)
        🙏new
        form = TaskForm(
            🙏new
            initial={
                🙏new
                'assignee': get_object_or_404(Developer, pk=self.kwargs['pk'])
            }
        )
        🙏new

```

```

    })
    new
    form.fields['assignee'].disabled = True
    new
    context['form'] = form
    new
    return context
    new

```

On réactive le champs désactivé lors de l'envoi du formulaire. Sinon la donnée n'est pas envoyée.

task/_create_task_modal.html

```

<div class="modal-body">
  <form
    id="create-form"
    onsubmit="$('#create-form *').removeAttr('disabled');"
    action="{% url 'task:create' %}" method="post">
    {% csrf_token %}
    {{ form|crispy }}
    <button class="btn btn-primary" type="submit">Créer</button>
  </form>
</div>

```

Annexes

Activer le bon lien app_name

Une autre solution pour activer le bon lien consiste à utiliser la variable ``request.resolver_match.app_name`` afin de récupérer l'application courante.

- `app_name` représente l'espace nom de l'application pour lequel il y a eu correspondance avec l'URL.

`BASE_DIR/_base.html

```
<nav class="navbar navbar-expand-sm bg-primary navbar-dark border-top
border-white">
  <ul class="navbar-nav">
    <li id="nav-home" class="nav-item {% if
request.resolver_match.app_name == "" %}active{% endif %}">
      <a class="nav-link" href="{% url 'home' %}"><i class="fa fa-
home"></i></a>
    </li>
    <li id="nav-dev" class="nav-item {% if
request.resolver_match.app_name == "developer" %} active {% endif %}">
      <a class="nav-link" href="{% url 'developer:index'
%}">Developers</a>
    </li>
    <li id="nav-task" class="nav-item {% if
request.resolver_match.app_name == "task" %} active {% endif %}">
      <a class="nav-link" href="{% url 'task:index' %}">Tasks</a>
    </li>
  </ul>
</nav>
```

LiveServerTestCase et selenium

Voir [ce lien](#)
