

# Django - TD 1

---

## 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 3.9+` ;
- de vérifier quel alias pointe vers cette version : `python -V` ou `python3 -V`
- 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`.

### *Mise en pratique au local 601*









*Une version de python est installée sur chacune des machines. Il faut cependant installer pour votre utilisateur courant le module Django. Il vous suffit d'utiliser la commande d'installation suivante qui spécifie que le module doit être installé dans votre répertoire utilisateur :*

```
python -m pip install Django --user
```

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 `python -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](#).

## Outils de développement

Vous êtes libre d'utiliser l'outil que vous préférez pour vos développements. Sur les ordinateurs de l'esi, nous mettons PyCharm à votre disposition.

Un des intérêts de cet outil est la gestion native des environnements virtuels : [venv](#).

## 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 :

`develper/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
```

Sur mon Mac : <http://127.0.0.1:8000/developer/>

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 ?

Ou taper ça ? Dans "python3 manage.py shell"

```
>>> 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']  
?
```

## 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'introspection 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`). exemple : `first_name = models.CharField(verbose_name="patate", max_length=200)`
- 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 présents sur votre disque. Vous pouvez consulter 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, retenez 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 [Git](#), par exemple



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.

# Liste des développeurs

- Jonathan
- Sébastien

## 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 !

## Configurer les chemins via `{% 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>
#...
```