Université Assane SECK de Ziguinchor

Unité de Formation et de Recherche des Sciences et Technologies

Département d'Informatique

Développement d'applications avec Django

Licence 2 en Ingénierie Informatique

Avril 2022

©Papa Alioune CISSE

Papa-alioune.cisse@univ-zig.sn

Résumé : À la différence du Développement Front-End, qui concerne l'aspect visuel et ergonomique d'un site web, le Développement Back-End se penche sur les aspects techniques et fonctionnels du développement d'un site web dynamique. Il s'agit du développement de l'ensemble des fonctionnalités "invisibles" d'un site web (le serveur, le code applicatif, la base de données, etc.) qui sont indispensables au bon fonctionnement d'un site. Ainsi, on peut dire que les Développeurs Back End travaillent sur la partie immergée de l'iceberg, alors que les Développeurs Front End se chargent essentiellement de la partie visible.
Ce chapitre aborde le développement d'une application web dans Django à travers son architecture MVT. Il montre également comment utiliser le Scaffolding de Django pour mettre en place le back-office d'un site.

1 - Le design pattern MVT de Django

Par rapport au MVC, Django opte pour une dénomination légèrement différente :

- Modèle = Model (pas de surprise…)
- Vue = Template
- Contrôleur = View

En Django, ce qu'on appelle View est donc un contrôleur.

- **Modèle**

Le modèle interagit avec la base de données. Sa mission est de chercher dans une base de données les items correspondant à une requête et de renvoyer une réponse facilement exploitable par le programme.

Les modèles s'appuient sur un ORM (*Object Relational Mapping*, ou Mapping objet-relationnel en français). A quoi cela sert-il ?

La consultation d'une base de données relationnelle se réalise par un langage appelé SQL (*Structured Query Language*). Sa syntaxe est très différente de Python ! Par exemple, la requête suivante renverra tous les items de la table content :

```
SELECT * FROM content
```

Ceux-ci seront dans un format que vous devrez alors interpréter manuellement avant de pouvoir l'utiliser dans votre programme. Il s'agit d'une étape assez fastidieuse qui n'est pas nécessaire.

Un ORM traduit les résultats d'une requête SQL en objets Python avec lesquels vous pouvez interagir. De même, il permet d'écrire une requête SQL en Python. Un peu comme un traducteur automatique !

Par exemple, la requête suivante renverra tous les objets du modèle « Content » :

```
Content.objects.all()
```

- **Template**

Un Template est un fichier HTML qui peut recevoir des objets Python et qui est lié à une vue (nous y reviendrons). Il est placé dans le dossier « templates » (qu'il faudra créer dans chaque application). Concrètement, un template peut interpréter des variables, des objets Python et les afficher.

- **Vue**

La vue joue un rôle central dans un projet structuré en MVT : sa responsabilité est de recevoir une requête HTTP et d'y répondre de manière intelligible par le navigateur.

Concrètement, une vue reçoit une requête http (une page à laquelle un utilisateur cherche à accéder), fait les traitements nécessaires pour satisfaire la requête de l'utilisateur et renvoie une réponse http (la page HTML demandée).

Quelques traitements nécessaires pour satisfaire une requête utilisateur :

- Si une interaction avec la base de données est requise, la vue appelle un modèle et récupère les objets renvoyés par ce dernier.
- Si un gabarit est nécessaire, la vue l'appelle.

Dans un projet Django, les vues de chaque application sont regroupées dans le document ***views.py***.

Chaque vue est associée à une url. Les urls d'une application sont regroupées dans le fichier ***urls.py*** de l'application. Les urls d'un projet sont regroupées dans le fichier ***urls.py*** du projet.

2 - Les Modèles de Django

2.1 - Introduction

Comme avec beaucoup de Frameworks web, les modèles Django sont des classes qui héritent d'une classe du Framework (classe qui s'appelle "***Model"***). En faisant hériter vos modèles de cette fameuse classe, vous leur conférez, *de facto*, toutes les propriétés et méthodes relatives aux modèles (gestion de l'ORM, etc.).

2.2 - Syntaxe de déclaration d'un modèle

C'est dans le fichier « ***models.py*** » d'une application qu'il faut déclarer tous ses modèles. La syntaxe de déclaration d'un modèle est la suivante :

```
from django.db import models
class MonModel(models.Model):
```

```
    """
    La documentation du modèle.
    """
    un_champ_texte = models.CharField(max_length=150)
```

2.3 - Types de champs disponibles

En créant un modèle Django, nous allons utiliser le système d'ORM du Framework. En d'autres termes, Django propose quelques types de champs (~ d'attributs) disponibles pour utilisation dans nos modèles, qu'il se chargera de mapper automatiquement avec les champs de base de données. En voici une liste non-exhaustive :

Nom	Description
CharField	Un champ de texte, typiquement mappé vers un « VARCHAR » en BDD.
TextField	Un champ de texte long, typiquement mappé vers un « TEXT » en BDD.
ForeignKey	Une référence à un objet d'un autre modèle. En BDD, il est représenté par un entier désignant l'ID de la ligne cible.
ManyToManyField	Une référence à plusieurs objets d'un autre modèle. En BDD, il est représenté à l'aide d'une table d'association.
BooleanField	Un champ booléan (vrai ou faux).
URLField	Un champ de texte représentant une URL, typiquement mappé vers un « VARCHAR » en BDD.
SlugField	Un champ de texte représentant un slug, c'est à dire une chaîne de caractère destinée à être utilisée dans une URL.
IntegerField	Un nombre entier.
DateTimeField	Une date et une heure.

Une liste exhaustive est disponible sur la documentation officielle de Django, section Model field reference.

Chaque type de champ admet un certain nombre de paramétrages que Django appelle des options. Nous n'allons pas tous les lister, mais voici les plus utilisés :

- **null=True|False** : Permet de spécifier si le champ correspondant en BDD peut être NULL ou pas.
- **blank=True|False** : Permet de spécifier si, dans les formulaires de l'application, le champ peut être laissé vide ou pas.
- **max_length=<int>** : Permet de spécifier la longueur maximale d'un CharField (généralement, la limite est de 255 caractères)

- **default=<int|str|boolean...>** : Permet de spécifier une valeur par défaut pour le champ

Voici un exemple d'utilisation de ces options dans le cadre de la déclaration d'un modèle :

```
from django.db import models


class MonModel(models.Model):
    un_champ_texte = models.CharField(max_length=150, null=True, blank=True, default="Coucou")
```

2.4 - Un exemple de modèle

Nous déclarons ci-dessous trois modèles (à ajouter dans le fichier « models.py » de l'application « apptest »: ***Ville, Poste*** et ***TempsDeTravail***.

```
from django.db import models
class Ville(models.Model):
          code_ville = models.CharField(max_length=50, verbose_name="Code de la ville")
          nom_ville = models.CharField(max_length=150, verbose_name="Nom de la ville")
          def __str__(self):
                    return self.nom_ville


class Poste(models.Model):
          code_poste = models.CharField(max_length=50, verbose_name="Code du poste")
          nom_poste = models.CharField(max_length=150, verbose_name="Nom du poste")
          def __str__(self):
                    return self.nom_poste


class TempsDeTravail(models.Model):
          code_temps = models.CharField(max_length=50, verbose_name="Code du temps de travail")
          libelle_temps = models.CharField(max_length=150, verbose_name="Libellé du temps de travail")
          def __str__(self):
                    return self.libelle_temps
```

2.5 - Création des tables de la base de données

Nous allons utiliser la base de données MySQL en supposant qu'il est déjà installé dans vos machines. Pour cela, nous allons installer d'abord le package « mysqlclient » pour que Django puisse communiquer avec MySQL. La commande d'installation est :

```
pip install mysqlclient
```

- **Modification du fichier settings.py**

Pour préciser à Django que nous souhaitons utiliser **MySQL**, nous allons modifier le fichier ***settings.py*** de notre projet. Ouvrez le fichier, cherchez la variable ***DATABASES***, et modifiez là pour qu'elle ressemble à ceci :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'dbtest',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '3306',    }
}
```

- **makemigrations et migrate**

Django sait maintenant où écrire ses données. Nous allons maintenant lui demander de créer les tables de la base de données pour nous, à partir des informations que nous lui avons fournies dans les modèles de nos applications (en l'occurrence, de notre application, puisque notre projet n'en comporte qu'une pour le moment).

Nous allons encore une fois utiliser la commande python ***manage.py***, avec :

- D'abord, l'argument ***makemigrations*** pour recenser, s'il y'en a bien sûr, les migrations (les changements dans le modèle qui ne sont pas encore effectifs dans la base de données) :

```
python manage.py makemigrations
```

- Ensuite, l'argument ***migrate*** pour effectuer les migrations :

```
python manage.py migrate
```

Par curiosité, vous pouvez ouvrir votre base de données MySQL pour voir ce qu'elle contient !

3 - SCAFFOLDING Django

3.1 - Définition

Le SCAFFOLDING est une fonctionnalité proposée par certains Frameworks de développement, permettant de créer des pages d'administration en prenant en charge les fastidieuses interfaces et opérations CRUD : création, lecture, mise à jour et suppression d'objets.

Django fournit une interface d'administration dont on peut paramétrer et surcharger les comportements de base pour en faire un véritable back-office pour nos applications.

Un back-office, contrairement au front-office, est une partie de votre projet à laquelle les utilisateurs finaux n'ont pas accès. C'est une partie d'application, voire une application à part entière, qui permet aux administrateurs du système d'en gérer le paramétrage et les données de paramétrage.

3.2 - L'espace d'administration de Django : une application à part entière

- **Activation de l'espace d'administration Django**

Il faut rappeler que Django est modulaire et tout fonctionne par applications. C'est donc tout naturellement que le scaffolding fasse l'objet d'une application sous Django. Pour l'utiliser, il faut simplement activer l'application Django nommée "**django.contrib.admin**" en l'ajoutant dans la variable "**INSTALLED_APPS**" du fichier de configuration "**settings.py**", s'il n y est pas déjà :

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'apptest',
)
```

- **Ajout de l'url d'accès à l'espace d'administration**

Pour accéder à l'espace d'administration, il faut ajouter son url dans le "contrôleur frontal" de votre projet qui est le fichier "**urls.py**" se trouvant dans le projet "**projettest**".

Le fichier "**urls.py**" contient les routes principales du projet, c'est-à-dire, les routes vers les différentes applications composant le projet et chaque application comportera son propre fichier "urls.py" pour contenir ses "routes locales".

Dans le code suivant, il est signalé à Django de rediriger toutes les requêtes dont l'URL commence par **admin/** vers le dispatcheur d'URL de l'application **admin.site** (**admin.site.urls**). C'est dans ce dernier fichier (que nous n'aurons pas besoin de regarder) que sont définies les requêtes pour l'ajout, la modification, la suppression, etc. d'objets.

```
from django.conf.urls import path, include, url
from django.contrib import admin
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Pour accéder maintenant à l'interface d'administration, il suffit d'ajouter "/admin/" après le lien du projet : http://127.0.0.1:8000/admin/.

La fenêtre qui s'ouvre vous demande de vous connecter. Mais jusque-là, nous n'avons encore créé aucun utilisateur.

- **Création d'un "super-utilisateur" pour l'administration du Projet**

Pour créer ce super-utilisateur, il suffit de taper la commande suivante :

```
python manage.py createsuperuser
```

Django nous invite à entrer les éléments d'identification de ce super-utilisateur : Nom d'utilisateur, adresse mail et mot de passe (à répéter).

Vous pouvez maintenant retourner à la fenêtre d'administration pour vous connecter avec le super-utilisateur qui vient d'être créé.

3.3 - Administration des applications Django

Pour administrer les différentes applications, il suffit d'ajouter les "**models**" de ces applications qui comportent des données de paramétrage sur lesquelles nous souhaitons effectuer des opérations CRUD.

Pour cela, il faut signaler à Django pour quels "**models**" nous voulons disposer d'une interface d'administration. Nous allons le faire en créant un fichier « ***admin.py*** » dans les répertoires des applications.

Puisque nous n'avons pour le moment que l'application "**apptest**" qui dispose de trois modèles "**Ville**", "**Poste**" et "**TempsDeTravail**", nous allons demander à administrer ces trois modèles depuis l'espace d'administration. Pour cela, il faut se placer dans le répertoire de l'application "**apptest**", et y créer un fichier "admin.py" (s'il n'existe pas déjà) avec le code suivant :

```
from django.contrib import admin
from mon_application1.models import Ville, Poste, TempsDeTravail


# Register your models here.
@admin.register(Ville)
class VilleAdmin(admin.ModelAdmin):
```

```
            list_display = ('code_ville', 'nom_ville')
            search_fields = ('nom_ville',)
            list_filter = ('nom_ville',)

@admin.register(Poste)
class PosteAdmin(admin.ModelAdmin):
            list_display = ('code_poste', 'nom_poste')
            search_fields = ('nom_poste',)
            list_filter = ('nom_poste',)

@admin.register(TempsDeTravail)
class TempsDeTravailAdmin(admin.ModelAdmin):
            list_display = ('code_temps', 'libelle_temps')
            search_fields = ('libelle_temps',)
            list_filter = ('libelle_temps',)
```

Actualiser la fenêtre d'administration pour voir que "Ville", "Poste" et " TempsDeTravail " sont ajoutés. Vous pouvez désormais ajouter, modifier, visualiser et supprimer des villes, postes et temps de travail.

4 - Les Contrôleurs Django

4.1 - Ajout des URLs

Dans un projet Django, pour chaque application qui doit être accessible via une url, il doit figurer une « route » dans le contrôleur frontal du projet (fichier « urls.py » du projet). Dans le code suivant, il y a une route vers l'application « admin » et une vers l'application « apptest ».

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

Ce contrôleur frontal (fichier « urls.py » du projet) permet donc de spécifier les « routes globales » vers les différentes applications du projet.

Pour spécifier les « routes locales » des applications (« routes » spécifiques aux applications), il faut les spécifier dans les fichiers « urls.py » (à ajouter par le programmeur) des applications.

Un premier contenu du fichier « urls.py » de l'application « apptest» est donné :

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

Ces définitions donnent 2 « routes » accessibles à partir de « apptest» dont les urls correspondent respectivement à « 127.0.0.1 :8000/apptest/ » et « 127.0.0.1 :8000/apptest /accueil». Ces 2 urls, menant toutes les 2 à la page d'accueil de l'application, sont contrôlées par la vue nommée ici « accueil » à définir dans le fichier « views.py » de l'application qui est importé dans la 2ème ligne du code précédant. Cette vue est censée (nous le verrons dans sa définition) recevoir la requête, faire les traitements nécessaires et rendre la page HTML (le Template) appropriée.

4.2 - Définitions des vues

Une vue est une fonction définie dans un fichier « views.py » qui reçoit comme paramètres la requête à partir de laquelle elle est invoquée et d'éventuels autres paramètres (nous verrons lesquels). Les traitements effectués par la vue sont donnés dans le corps de la fonction. La vue rend enfin un Template (une page HTML) qui correspond à l'url de départ ou une redirection vers une autre vue.

Un exemple de définition de la vue « accueil » de l'application « apptest » est donné ci-dessus :

```
from django.shortcuts import render
def accueil(request):
        unObjet = "Je suis un objet envoyé à la page HTML depuis une vue"
        return render(request,'apptest/home.html', {'unObjet': unObjet})
```

Le seul traitement effectué par cette vue est la définition d'un objet « unObjet » qui est ensuite passé au Template. La dernière ligne de cette vue permet de rendre (avec la directive « render » la page HTML appelée « accueil.html » situé dans le dossier « apptest » lui-même situé dans un dossier nommé « templates » (à ajouter par le programmeur dans le répertoire de l'application). En effet, dans Django, les pages HTML (Template) doivent être ajoutées dans les dossiers « templates » et il faut aussi en informer Django dans le fichier de configuration, comme nous le verrons dans les chapitres suivants.