Chapitre 3
Modèles

1. Introduction aux modèles

1.1 Django et modèle de données

Django est un framework de développement d'applications conçues au-dessus d'une base de données relationnelle. Chacun de ces mots revêt une importance capitale. Django est fait pour gérer la partie technique et vous offrir des facilités pour créer rapidement une application. Il s'appuie sur une base de données relationnelle. Ce type de base de données est le plus répandu, elles existent depuis très longtemps, sont très stables et très performantes. On peut citer PostgreSQL, appréciée pour sa performance et sa robustesse, ou MariaDB, appréciée pour sa simplicité d'utilisation et sa popularité. Toutes deux sont extrêmement fiables.

Bien que Django soit polyvalent et qu'il puisse fonctionner au-dessus de MongoDB, par exemple, une base de données orientée document, il est vraiment taillé et efficace pour travailler avec une base de données relationnelle.

Développer une application consiste alors simplement à gérer son modèle de données et à l'utiliser pour gérer des fonctionnalités ou des affichages.

Le fait de concevoir le modèle de données est la toute première des préoccupations du développeur Django, loin devant le fait de savoir utiliser ces données et de devoir les rendre dans divers formats.

1.2 Qu'est-ce qu'un modèle ?

En modélisation de base de données, on parle d'entité et de relation. Une entité peut être par exemple, un produit, une commande, une livraison ou encore une personne, un rôle ou une mission. La relation permet de lier des entités. On peut décider qu'une personne a un et un seul rôle, qu'une personne a 0 à plusieurs missions, mais qu'une mission est remplie par une et une seule personne, qu'une commande a un à plusieurs produits, mais qu'un produit peut se trouver dans 0 à n commandes.

C'est en définissant précisément ces règles que l'on pourra modéliser comment les données pourront vivre, comment elles seront validées et comment elles seront restituées.

Chaque entité peut avoir plusieurs champs pour la décrire. Chaque champ d'une entité doit concerner l'entité elle-même. Par exemple, une personne a un nom, un prénom, une date de naissance… On peut, pour chaque champ, décider s'il est obligatoire, s'il a des limites (par exemple 50 lettres maximum pour le prénom, une date de naissance comprise entre 1950 et 2000, par exemple), etc.

Chaque association peut également avoir des données propres. Par exemple, l'association entre la commande et le produit peut porter le nombre de produits commandés. Mais elle peut aussi porter le prix, si l'on décide que le prix peut changer d'une commande à l'autre de manière aléatoire.

On pourra sinon décider d'utiliser une entité liste de prix pour gérer des prix différents pour des types de clients différents ou encore que le prix est une propriété du produit et qu'il est toujours fixe pour un produit donné.

Là encore, pour tous ces choix, il n'y a pas de bonne réponse de premier abord. Il faut comprendre comment les choses fonctionnent dans le monde réel, là où l'application sera utilisée, pour modéliser un modèle de données qui permet de représenter la réalité.

Autant comprendre les outils de Django est relativement aisé, cela demande juste de la pratique, autant le fait de concevoir le modèle de données en faisant en sorte qu'il soit réellement adapté au besoin est une tâche très importante sur laquelle il faudra passer énormément de temps.

1.2.1 Exemple du tutoriel de Django

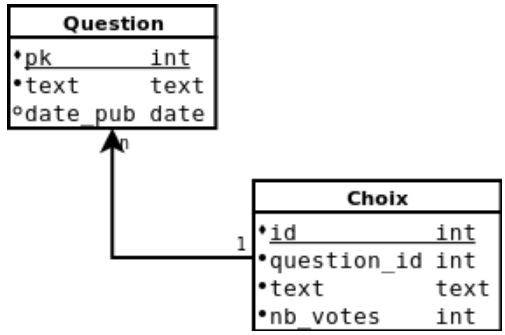
Par exemple, le tutoriel de Django (qui est excellent et qu'il est recommandé de faire au moins une fois si l'on débute) a pour but de décrire un sondage. Un sondage est une liste de questions et chaque question peut avoir plusieurs choix possibles.

Nous avons donc deux modèles : le modèle **question** et le modèle **choix**.

Dans le modèle **question**, il va simplement se trouver un texte contenant la question, puis toute une série de propriétés qui peuvent être rajoutées et qui sont purement liées à la question. Dans l'exemple du tutoriel, une date de publication.

Dans le modèle **choix**, on a le texte décrivant le choix, mais aussi le lien vers le modèle **question**, parce que ce choix est une des réponses possibles à la question. En effet, chaque choix est lié à une et une seule question, mais une question peut techniquement avoir zéro à plusieurs choix (et fonctionnellement au moins deux).

On a aussi un champ vote permettant de compter le nombre de votes pour le choix en question.



Exemple de schéma de la base de données possible pour le tutoriel Django

Il serait possible de s'interdire d'avoir deux questions identiques en ajoutant une contrainte d'unicité sur le champ `text`. Il serait aussi possible de s'interdire d'avoir deux choix identiques pour une même question en ajoutant une contrainte d'unicité sur le couple de champs `(question_id, text)`.

Selon les besoins liés à l'application, il aurait été possible d'imaginer quelque chose de plus complexe, par exemple de gérer les votes dans un modèle à part, pour tracer les votes, les lier à un compte utilisateur et interdire de voter deux fois ou plus simplement pour sauvegarder l'instant du vote et ainsi se permettre d'avoir des statistiques sur les évolutions des votes dans le temps.

L'idée à retenir est que chaque détail d'un objet doit être décrit par son modèle et que les fonctionnalités à réaliser vont déterminer le nombre de modèles à créer, leur complexité et les champs à y mettre.

1.2.2 Exemple pour une bibliothèque de livres

Prenons un autre exemple : nous souhaitons gérer une bibliothèque de livres. On pourrait imaginer tout gérer avec un seul modèle **livre**, qui contiendrait le titre du livre, le nom de son auteur, mais aussi des champs texte pour noter son format (poche, bande dessinée…) ou encore le genre (policier, fantastique…).

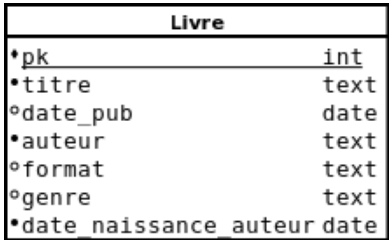
Si l'on fait cela, lorsque l'on saisira les données, il faudra imaginer un formulaire avec seulement des champs textes. Si l'on veut répondre à la question : « Quels sont les livres écrits par tel auteur ? » ou « Quels sont les livres de tel genre ? », c'est possible, mais ce n'est pas structuré.

En effet, le fait de répéter le nom du genre, par exemple, à chaque entrée est problématique, car cela fait beaucoup de fois la même information et on prend le risque d'une erreur de saisie.

De plus, lorsque l'on débute, on est souvent tenté de mettre toutes les informations au même endroit. Par exemple, on pourrait rajouter un champ `date_naissance_auteur` dans le modèle livre pour avoir aussi cette information, car on souhaite l'afficher sur la page présentant le livre.

C'est une mauvaise idée, car cette information appartient à l'auteur et non au livre lui-même. Il ne faut donc surtout pas hésiter à avoir plus de modèles, mais chacun ayant sa cohérence de données, plutôt que de vouloir faire peu de modèles mais très fournis.

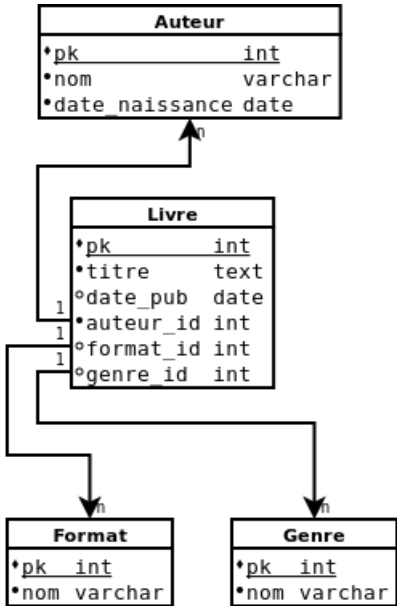
En effet, avec ce type de modèles, on multiplie des informations identiques et on a potentiellement deux dates de naissance différentes pour un même auteur.



Exemple de modèle mal conçu

Avoir un modèle spécifique pour gérer les données spécifiques à l'auteur permet d'éviter ces écueils et c'est quelque chose d'important à faire même si le modèle ne contient rien d'autre que le nom de l'auteur.

Permettre d'avoir un modèle pour gérer les formats et un autre pour les genres permettra de prédéfinir ces données et on imagine alors que lors de la saisie d'un livre, ce ne sera plus un champ libre que l'on aura, mais une liste déroulante, retirant toute ambiguïté et tout risque d'erreur de saisie.



Exemple de modèle mieux conçu

1.3 Notion d'app Django

Dans la sémantique de Django, le projet est l'ensemble du code que vous allez créer. Il contient le package Python du projet que nous avons détaillé, dans lequel on trouve un package portant le même nom que le projet et qui contient entre autres le fichier settings.py. Le projet contient également le répertoire static dans lequel vont se retrouver tous les fichiers image, les feuilles de style ou le JavaScript que vous servirez.

Le code en tant que tel va se trouver dans des applications (ou app Django). Django vous propose donc de diviser votre code en plusieurs applications pour en faciliter la maintenance. À vous de diviser de manière pertinente et de tenter de ne pas avoir d'application de taille trop élevée.

Pour créer une nouvelle application, il faut, dans le terminal, ouvrir une console bash dans le conteneur Docker de Django :

```
$ make bash
```

Une fois dans le conteneur, on peut taper la commande suivante :

```
$ poetry run python ./manage.py --help
```

Cela permet de visualiser les différentes actions que l'on peut faire à partir du point d'entrée de Django. En ce qui nous concerne, on veut utiliser `startapp`. Voici comment voir les options disponibles de cette commande :

```
$ poetry run python ./manage.py startapp --help
```

Pour notre part, nous allons utiliser sa forme la plus basique :

```
$ poetry run python ./manage.py startapp app
```

Nous venons donc de créer une application nommée **app** qui contient plusieurs éléments. Par ordre chronologique pour nous, le premier élément est le fichier models.py, dans lequel nous allons écrire nos modèles. Ce sera l'objet de ce chapitre, mais également celui du chapitre Notions avancées sur les modèles, pour approfondir.

Nous avons ensuite le fichier admin.py dans lequel nous allons déclarer notre interface d'administration. Nous aborderons ceci dans les chapitres CRUD et Gestion avancée de l'interface d'administration.

Le fichier views.py contiendra toutes les vues, abordées dans les chapitres Introduction aux vues et Notions avancées sur les vues ainsi que les points d'accès API, abordés dans le chapitre Rest et Swagger.

Enfin le fichier tests.py qui contiendra tous les tests unitaires, abordés dans le chapitre Tests.

Il existe aussi un fichier apps.py qui permet de paramétrer les métadonnées de l'app Django. Ce dernier peut être personnalisé pour personnaliser l'affichage du nom de l'app et le rendre traductible.

Le fichier ressemblera alors à cela :

```
from django.apps import AppConfig

from django.utils.translation import gettext_lazy as gettext


class AppConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'app'
    verbose_name = gettext("My app")
```

Pour terminer, on peut signaler le répertoire migrations qui va contenir toutes les instructions à donner à la base de données pour passer d'un état à un autre (et ainsi suivre les modifications que l'on fera au fur et à mesure sur le modèle, en créant, modifiant ou supprimant des tables ou des colonnes, par exemple, pour faire correspondre la base de données au modèle).

Pour terminer notre travail, il faut maintenant modifier la configuration du projet pour rajouter l'application que nous venons de créer. Pour cela, il faut ouvrir le fichier settings.py et identifier la constante `INSTALLED_APPS` pour y rajouter ceci :

```
INSTALLED_APPS = [
    'app.apps.AppConfig',
    'django.contrib.admin',
    ...
]
```

Pour visualiser le code relatif à ce chapitre : git diff v1.0.6..v1.1.0.

1.4 Création d'un modèle

Pour créer un modèle, il faut se rendre dans le fichier models.py et créer une classe qui va hériter du modèle de base de Django.

Dans l'exemple fil rouge de ce livre, nous allons gérer des jeux.

```
class Game(models.Model):
    pass
```