Objectifs de la feuille

- ORM DJANGO
- Création du modèle
- Exploration des champs du modèle
- Gestion de la base de données avec les migrations
- Shell interactif Python et API DJANGO
- Mise à jour de la vue pour afficher le modele

Représenter une entité avec un modèle

Quel que soit le but de votre application web, il est très probable qu'elle doive stocker des données. Les programmeurs commencent souvent par identifier les différentes entités au sujet desquelles nous devons stocker des données.

Pour chaque entité pour laquelle nous voulons stocker des données, nous créons un modèle pour représenter cette entité. Un modèle définit les caractéristiques que nous voulons stocker à propos d'une entité particulière.

Notre modèle de groupe pourrait donc avoir des caractéristiques. Ces caractéristiques sont également connues sous le nom de champs. Nous pouvons ensuite utiliser le modèle pour créer des objets individuels, ou instances, de ce modèle, qui ont chacun leurs propres caractéristiques uniques. On pourrait penser que le terme «modèle» n'est qu'un autre nom pour une «classe» ! Et vous auriez à moitié raison : vous créez un modèle dans DJANGO en définissant une classe Python.

En général, dans les frameworks MVC et MVT, un modèle est également capable de stocker (ou de "persister") ses données dans une base de données pour une utilisation ultérieure. Cela contraste avec les classes et objets ordinaires, dont les données existent temporairement : par exemple seulement pendant l'exécution de l'application. De même, les «caractéristiques» des classes Python sont appelées attributs, mais lorsqu'un modèle enregistre un attribut dans la base de données, il s'agit d'un champ.

C'est quoi un ORM?

Le plus dans DJANGO c'est son ORM, très puissant et très simple à utiliser. Bien sûr il ne répondra pas à tous vos besoins : passer par des requêtes SQL est inévitable mais 99,99% du temps vous avez un outil qui vous assiste vraiment.

ORM est l'acronyme anglais de Object Relational Mapping , donc mapping d'objet relationnel en français. Un ORM est une technique de programmation qui donne l' illusion de travailler avec une base de données orientée objet . Pour résumer vous ne faites plus de requêtes SQL mais vous travaillez directement avec vos objets.

L' ORM de DJANGO est très bien conçu. D'une part il vous permet de lister tous les objets nécessitant des enregistrements (on parle de modèles) dans un endroit prévu à cet effet. Si vous êtes un nouveau développeur sur un projet, vous comprenez ce projet en quelques secondes en visualisant ces fichiers. Un autre avantage de travailler avec un ORM, c'est qu'il est une couche d'abstraction qui exploite la base de données ce qui fait que le développeur ne se soucie plus du SGDBR que le projet utilise . Comme nous l'avons vu précédemment concernant les bases de données, utiliser SQLite ou MySQL implique des requêtes SQL spécifiques dans certains cas. Avec Django , il n'y a pas de différence, c'est lui qui gère de faire la requête compatible en fonction du SGBDR. La compatibilité des bases de données, c'est devenu son boulot, plus le vôtre.

Un modèle est donc pour faire simple une table de données. On lui donne un nom, des champs typés et un comportement. Travailler avec un ORM permet donc une homogénéité du code , une structure plus solide , plus simple à maintenir et est optimisé pour les requêtes les plus simples .

C'est l'un des avantages de l'utilisation d'un framework comme DJANGO : toutes les fonctionnalités de persistance des données dans une base de données ont déjà été écrites pour vous. Tout ce que vous avez à faire est de faire en sorte que votre modèle hérite de la classe `models.Model` de DJANGO. Votre modèle hérite ensuite de toutes les méthodes (comportements) nécessaires pour effectuer des opérations telles que la sélection et l'insertion de données dans une base de données.

Vous n'aurez donc pas à écrire de code qui interagit avec une base de données, mais vous devrez apprendre à interagir avec les modèles de DJANGO pour faire la même chose (ne paniquez pas, c'est plus facile ! ☺).

Création du modèle

Créons maintenant notre premier modèle. Nous allons faire simple pour commencer. 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.

Dans notre application, nous allons créer deux modèles : Produit et Element. Un produit possède un nom et un code. Un élément a un champ : une couleur. Chaque élément est associé à une produit. Ces concepts sont représentés par des classes Python. Éditez le fichier monapp/models.py de façon à ce qu'il ressemble à ceci :

```
from django.db import models

"""
Produit : nom, code, etc.
"""
class Product(models.Model):

    name = models.CharField(max_length=100)
    code = models.IntegerField()

    def __unicode__(self):
        return "{0} [{1}]".format(self.name, self.code)

"""
    Déclinaison de produit déterminée par des attributs comme la couleur, etc.
"""
class ProductItem(models.Model):
    color   = models.CharField(max_length=100)
    product = models.ForeignKey('Product', on_delete=models.CASCADE)

    def __unicode__(self):
        return "{{0}} {1} [{{2}}]".format(self.product.name, self.color, self.product.code)
```

Un modèle est donc une simple classe python qui hérite de la classe models.Model. Les champs sont définis dans la classe, on leur donne un nom et un type. Et voilà notre premier modèle.

Cela peut sembler différent des classes que vous avez construites auparavant. Les classes Python ont généralement un constructeur : la méthode __init__, où nous utilisons les arguments passés pour définir les valeurs des attributs d'instance. Avec les modèles DJANGO, les choses se font différemment. Le framework examine les champs du modèle (que nous définissons comme des attributs de classe), puis crée le constructeur pour nous. C'est un autre exemple de la «magie» du framework. Comme pour les classes Python, nous pouvons utiliser les modèles DJANGO pour créer des instances de cette classe.

Le champ des modèles DJANGO

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.

AutoField	→ incrémente automatiquement sa valeur
BinaryField	→ stocke des données binaires brutes en octets (bytes)
BooleanField	→ un champ True / False
CharField	→ un champ pour une chaine de caractères assez courte
TextField	→ pour du texte long
CommaSeparatedIntegerField	→ entiers séparés par un virgule
EmailField	→ vérifie une valeur d'adresse valide
SlugField	→ format slug (alphanumérique + tirets)
URLField	→ format URL
DateField	→ une date, instance de datetime.date en python
DateTimeField	→ une date et une heure, instance python de datetime.datetime
DecimalField	→ un nombre décimal de taille fixe, instance python de Decimal
FileField	→ un champ de fichier à téléverser
ImageField	→ idem que FileField mais vérifie qu'il s'agit d'une image
FilePathField	→ un path de fichier (paramètre *path* est obligatoire)
FloatField	→ une instance de float en python
GenericIPAddressField	→ une adresse ip valide IPV4 / IPV6
IPAddressField	→ une adresse ip textuel type 192.168.0.1
IntegerField	→ valeurs comprises entre -2147483648 à 2147483647
BigIntegerField	→ Un entier 64 bits
PositiveIntegerField	→ valeurs comprises entre 0 et 2147483647
PositiveSmallIntegerField	→ valeurs comprises entre 0 et 32767
SmallIntegerField	→ valeurs comprises entre -32768 et 32767
NullBooleanField	→ un champ booléen qui accepte le Null
TimeField	→ format heure instance de datetime.time

Chaque champ est représenté par une instance d'une classe Field – par exemple, CharField pour les champs de type caractère, et IntegerField pour les champs de type entier. Cela indique à DJANGO le type de données que contient chaque champ. Le nom de chaque instance de Field (par exemple, name, code, ou color) 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.

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. Un champ IntegerField peut aussi autoriser des paramètres facultatifs (default=0)

Finalement, notez que nous définissons une relation, en utilisant ForeignKey. Cela indique à DJANGO que chaque élément (Item) n'est relié qu'à un seul produit (Product). DJANGO propose tous les modèles classiques de relations : plusieurs-à-un, plusieurs-à-plusieurs (ManyToManyField), un-à-un (OneToOneField), mais nous y reviendrons plus tard.

Les champs des modèles sont définis dans le module suivant: django.db.models.fields. Il est donc possible de voir tous les champs existants en exécutant la commande help(django.db.models.fields)

---------- paramètres communs ----------
db_column → nom de la colonne dans la base de données
db_index → créer un index pour la colonne
default → la valeur par défaut du champ
editable → Si False le champ n'est pas éditable dans admin
help_text → texte d'aide affiché dans le formulaire
primary_key → si True devient la clé primaire
unique → si True impossible d'avoir des doublons de valeur
verbose_name → un nom plus explicite
validators → une liste de validateurs à exécuter

---------- paramètres spécifiques ----------
primary_key → renseigner la clé primaire
blank → autoriser la soumission d'un champ vide
null → autoriser d'enregistrer en base une valeur nulle
unique_for_date → unique pour une date
unique_for_month → unique pour un mois
unique_for_year → unique pour un an
choices → choix possibles

Migrations

Comme nous l'avons dit, une des caractéristiques d'un modèle est qu'il est capable de stocker ses données dans une base de données. Nous avons créé notre base de données dans le chapitre sur la configuration. Mais cette base de données ne sait encore rien de notre modèle. Et c'est là que les migrations entrent en jeu.

Si nous voulons stocker les produits dans notre base de données, nous aurons besoin d'une nouvelle table, contenant une colonne pour chaque champ que nous avons ajouté à notre modèle, ainsi qu'une colonne id pour servir de clé primaire : un identifiant unique pour chaque ligne de la table. La structure d'une base de données, en termes de tables et de colonnes, est appelée schéma.

Si nous construisions notre schéma de base de données manuellement, nous pourrions écrire une requête SQL ou utiliser une interface graphique de gestion de base de données, pour créer notre première table. Mais dans DJANGO, nous faisons les choses différemment. Nous utilisons une sous-commande de l'utilitaire de ligne de commande qui va générer des instructions pour construire la table. Et ensuite, nous utilisons une autre sous-commande pour exécuter ces instructions. Ces instructions sont appelées une migration.

Une migration est un ensemble d'instructions permettant de passer le schéma de votre base de données d'un état à un autre. Il est important de noter que ces instructions peuvent être exécutées automatiquement, comme un code.

En programmation, on parle souvent de «configuration par le code». Il s'agit d'une philosophie qui stipule que toutes les étapes nécessaires à la construction de votre application ne doivent pas être effectuées à la main, mais plutôt inscrites dans le code. Pourquoi ? Pour plusieurs raisons :

- Les étapes manuelles peuvent facilement être oubliées, mais les étapes écrites sous forme de code peuvent être stockées dans un repository afin qu'elles soient aussi sûres que tous vos autres codes sources.
- Lorsque les étapes sont conservées dans votre repository, elles peuvent être facilement partagées avec les autres membres de l'équipe.
- Les étapes écrites sous forme de code peuvent être exécutées automatiquement par votre ordinateur. Cette méthode est rapide et fiable, surtout s'il y a plusieurs étapes.

Maintenant que nous savons pourquoi les migrations sont importantes, créons-en une pour notre modèle. Exécutons une autre commande :

```
manage.py makemigrations monapp
```

En exécutant makemigrations, vous indiquez à DJANGO que vous avez effectué des changements à vos modèles (dans ce cas, vous en avez créé) 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 sur un disque.

Le résultat de la commande nous indique qu'une nouvelle migration a été enregistrée dans monapp/migrations/0001_initial.py, et que son objectif est de «Créer le modèle de produit et élément», ce qui signifie en fait que cette migration va créer deux tables dans la base de données pour notre modèle.Vous pouvez consultez la migration pour vos nouveaux modèles si vous le voulez; il s'agit du fichier monapp/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 humainement lisibles au cas où vous auriez besoin d'adapter manuellement les processus de modification de DJANGO.

L'avantage de cette commande est qu'elle analyse notre fichier models.py pour y déceler toute modification et déterminer le type de migration à générer. Maintenant que nous avons notre migration (nos instructions), nous devons exécuter ces instructions sur la base de données. 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 :

```
manage.py sqlmigrate monapp 0001
```

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 cela vous intéresse, vous pouvez aussi exécuter manage.py check. Cette commande vérifie la conformité de votre projet sans appliquer de migration et sans toucher à la base de données.

Maintenant, exécutez à nouveau la commande migrate pour créer les tables des modèles dans votre base de données :

```
manage.py migrate
```

Vous vous rappelez dans le chapitre sur l'installation quand nous avons ajouté «monapp» au INSTALLED_APPS de notre projet ? DJANGO a recherché dans chacune de ces applications installées de nouvelles migrations à exécuter, il a trouvé notre nouvelle migration et l'a «appliquée» : il a exécuté ces instructions sur la base de données.

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.

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 :

- Modifiez les modèles (dans `models.py`).
- Exécutez `manage.py makemigrations` pour créer des migrations correspondant à ces changements.
- Exécutez `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.

Exercez vous ! Modifiez votre modèle tels que :

- un produit possède un prix HT
- un produit possède une date de fabrication
- un élément possède également un code
- un produit possède un statut
- un statut est composé d'un numéro et d'un libelle (Offline, Online, Out of stock)

Et faites une ou plusieurs migrations en respectant bien les étapes et en observant les fichiers générés.

Le shell Python et l'API DJANGO

Dans cette section, nous allons écrire du code dans le shell interactif Python pour jouer avec l'API que DJANGO met gratuitement à votre disposition. Le shell de DJANGO est simplement un shell Python ordinaire qui exécute votre application DJANGO. Vous pouvez le considérer comme un endroit où vous pouvez essayer du code en temps réel : chaque fois que vous appuyez sur Entrée, la ligne de code que vous venez de taper est exécutée. Ainsi, alors que le code que vous tapez dans un module/fichier Python peut être exécuté de nombreuses fois, le code que vous tapez dans le shell DJANGO n'est exécuté qu'une seule fois, puis oublié.

Utilisons le shell pour créer quelques objets, puis enregistrons ces objets dans la base de données. Vous pouvez utiliser mes exemples, ou n'hésitez pas à ajouter vos propres objets préférés ! Pour lancer un shell Python, utilisez cette commande : `python manage.py shell`

À l'invite du shell (>>>), tapez le code suivant pour importer notre modèle :
```
>>> from monapp.models import *
```
Appuyez sur Entrée pour exécuter cette ligne.

Ensuite nous allons vérifier qu'il n'y a pas encore de produits dans notre base.
```
>>> Product.objects.all()
```

Ensuite, nous allons créer une nouvelle instance du modèle Product :
```
>>> prdt = Product()
>>> prdt.name = "ipod"
>>> prdt.code = 1234
```

Jetez un coup d'oeil à l'état actuel de l'objet en tapant simplement `prdt` puis Entrée
```
>>> prdt
```

Le shell nous dit que nous avons un objet prdt, mais l'id est None, il n'a pas encore d'identifiant. Maintenant, sauvegardons cet objet dans la base de données :
```
>>> s=Status(numero=0,libelle="Offline")
>>> s.save()
>>> prdt.status=s
>>> prdt.save()
```

La méthode `save` enregistre les données dans la base. Tant que vous ne l'avez pas appelée, aucune information n'est enregistrée. Et ensuite regardez à nouveau l'état de l'objet :
```
>>> prdt
```

Chaque fois que nous insérons un objet dans la base de données, un identifiant est ajouté automatiquement pour nous. On aurait pu tout aussi bien créer une instance de Product en utilisant la syntaxe suivante :
```
>>> Product(name="iphone", code=2468, status=s).save()
```

ou encore
```
>>> prdt=Product.objects.create(name="ipad", code=5678,status=s)
```

Notre base de données contient maintenant 3 objets Product. On peut vérifier ça comme ça :
```
>>> Product.objects.count()
```
ou encore
```
>>> Product.objects.all()
```

Pour l'instant, vous pouvez considérer qu'un QuerySet ressemble beaucoup à une liste Python.

Profitons-en pour créer une déclinaison de notre dernier produit (instance de ProductIem) qu'on associe à notre dernier produit.
```
>>> prdt_itm = ProductItem()
>>> prdt_itm.color = "blue"
>>> prdt_itm.product = prdt
>>> prdt_itm.save()
```

La variable prdt est l'instance du produit que nous avons crée précédemment. Vous ne pouvez pas directement lui donner comme valeur la primary key (clé primaire), une instance Product est attendu.

Vous pouvez récupérer des entrées de votre base de données (sous forme d'objet) via un manager que tout modèle possède et qui s'appelle objects . Nous l'avons utilisé dans les exemples précédent mais sans savoir ce qu'il est et ce qu'il est capable de faire. Vous pouvez d'ailleurs le voir dans le shell:
```
>>> Product.objects
```

Ce manager possède un grand nombre de méthode qui vous permettra de filtrer les objets en fonction de votre besoin :

- Pour afficher toutes les entrées d'un modèle vous pouvez utiliser la méthode all :
  ```
  >>> Product.objects.all()
  ```
  Le manager vous retourne une liste qui contient sous forme d'objet toutes les entrées de la base de données.
  ```
  >>> Product.objects .all()[0]
  ```
- Vous pouvez demander au manager de vous retourner un objet en utilisant la méthode get :
  ```
  >>> Product.objects.get(pk=2)
  ```
- La méthode get ne retourne qu'un seul objet, si vous voulez récupérer tous les objets ayant X conditions vous pouvez utiliser la méthode filter:
  ```
  >>> Product.objects.filter(code=1234)
  ```
- Vous pouvez effectuer des recherches plus complexes comme "Commence par":
  ```
  >>>Product.objects.filter(name__startswith="ip")
  ```
- Le concept "Contient":
  ```
  >>> Product.objects.filter(name__icontains="ipad")
  ```
  A noter que la recherche "contain" (au lieu de icontain) peut être aussi utilisé mais elle sera sensible à la casse.
- Quitter le shell : exit() ou Ctrl+D ou quit()

Petit challenge

Alors, comment faire pour que nos objets sortent de la base de données et se retrouvent dans nos pages ?

- créer une nouvelle fonction ListProducts dans votre fichier views.py
- nous avons vu comment obtenir tous les objets Product de la base de données dans le shell : Product.objects.all().
  Faites la même chose dans votre vue et stockez le résultat dans la variable prdcts
- N'oubliez pas d'importer votre modèle Product !
- La variable prdcts contient maintenant une liste de tous les produits qui peuvent être trouvés dans la base de données. Cela signifie qu'il est maintenant possible d'accéder à chacun des objets Product individuels en utilisant la notation d'index de Python, comme ceci :

```
prdcts[0] # pour le premier objet Product
prdcts[1] # pour le second, etc.
```

- Vous ferez appel à l'attribut name de l'objet en utilisant la notation par points de Python, comme nous le ferions avec n'importe quel autre objet : prdcts[0].name
- Utilisez ces techniques pour afficher les noms de vos produits dans notre page HTML. Utilisez les balises <ul> et <li> pour afficher sous forme de liste. A vous de construire la bonne chaine HTML avant de la passer en argument de HttpResponse lors du return. Pour aider à la lisibilité, vous pouvez :
  - utiliser des guillemets triples (""") pour répartir votre chaîne HTML sur plusieurs lignes ;
  - faire de cette chaîne une «f-string» (f""") afin que nous puissions injecter nos noms de produit dans la chaîne en utilisant { ... } .

- N'oubliez pas de modifier votre fichier urls.py pour associer votre nouvelle vue.

Testez !
Ajoutez des produits dans votre modèle à l'aide du shell Python.
Mettez à jour votre code, et actualisez la vue !

Refaites l'exercice pour les éléments de produits (objet ProductItem) et les statuts (si vous avez bien réussi l'exercice tout à l'heure sur l'évolution du modèle...)