Django Models

Models in **Django** represents the data logic in our app. Usually in a web application our data will arrive from a database. Using **Django Models** a model can represent a table in our database and a property in that model can represent a field in the table. **Django** provide us easy to use classes to synchronize between our classes and the database tables. Let's try and understand how **Django Model** classes work.

Bootstrapping our Django Project

To demonstrate **Django Models** we will start a new **Django Project** create a directory to hold your project

> mkdir django-models
> cd django-models

create a virtual environment for your project:

> virtualenv venv

activate your virtual environment

> source venv/bin/activate

install django

> pip install Django

use the **django-admin cli** to start a new **Django** project

> django-admin startproject django_models

now start a new app in your project

> cd django_models
> python manage.py startapp models_intro

Install the app by modifying the **INSTALLED_APPS** array in the project **settings.py**

INSTALLED_APPS = [

```
  'models_intro',
  'django.contrib.admin',
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.messages',
  'django.contrib.staticfiles',
]
```

verify that everything is working by launching your app:
> python manage.py runserver

you should see the it works message.

Database Configuration

Looking on our **settings.py** file you will see default database configuration that looks like this:

```
DATABASES = {
  'default': {
      'ENGINE': 'django.db.backends.sqlite3',
      'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
  }
}
```

The databases dictionary contains the configuration for our databases, right now we have only one database named as **default** which use the **sqlite** db backend and created a database called **db.sqlite3** in the root directory.
We will keep this defaul settings and use the out of the box **sqlite** but you can modify this settings to connect to other database. **Django** supports the popular databases.

Our Database

Our database will be a really simple representation of a blog database. We will have 3 tables

- **Post** table that contains our blog post
- **Author** the writer of the post
- **AuthorSettings** the writer can determine settings on his account
- **Tag** each post can contain multiple tags for filtering the articles later on

The **Post** table will contain the following fields:

- title : string

- body : string
- author : Author
- date : Date
- tags : Tag[]

Our **Author** table will contain the following:
- first_name : string
- last_name : string

Our **AuthorSettings** table will contain authors settings for their account
- is_invisible : bool
- author : Author

Our **Tag** table will contain the following:
- title : string
- posts : Post[]

In our database demo we can see that our tables are connected to each other. Let's go over the different type of connections between database tables:

Many-To-Many

Good example for **M2M** relationship between tables is with our **Tags<-->Posts** a post can have multiple tags and a tag can have multiple posts.
To define this relationship using django we will need to use the **ManyToManyField** like this:

```
class Tag(models.Model):
   title = models.CharField()

class Post(models.Model):
   tags = models.ManyToManyField(Tag)
```

Many-To-One

Good example for **Many2One** is the relationship between the tables: **Posts ←>Author** one author can write many posts.
We can use the **ForeignKey** to define this kind of relationship, for example:

```
class Author(models.Model):
   first_name = models.CharField()

class Post(models.Model):
   author = models.ForeignKey(Author)
```

OneToOne

This relationship connects a row from one table to another one but enforces just a one to one mapping between these rows meaning one row can at most be connected to one row from another table. Example of case where this relationship is useful is between our **Author←>AuthorSettings** table. An **Author** can have only a single settings row associated to him.
To define this relationship we can use **Django OneToOneField** example:

```
class AuthorSetting(models.Model):
    author = models.OneToOneField(Author)
```

Creating our Models file

We will now create the database table models in our **models_intro** application. When we created our app **Django** automatically created a **models.py** file where it expects us to place our table models there.
Modify that file to look like this:

```
from django.db import models

class Author(models.Model):
  first_name = models.CharField(max_length=100, blank=False, null=False)
  last_name = models.CharField(max_length=100, blank=True, null=True, default=None)

  def __unicode__(self):
      return self.first_name


class AuthorSettings(models.Model):
  author = models.OneToOneField(Author, blank=False, null=False)
  is_invisible = models.BooleanField(default=False)

  def __unicode__(self):
      return "%s settings" % self.author.first_name

class Tag(models.Model):
  title = models.CharField(max_length=100, blank=False, null=False, unique=True)

  def __unicode__(self):
```

```
    return self.title

class Post(models.Model):
  tags = models.ManyToManyField(Tag)
  author = models.ForeignKey(Author)
  title = models.CharField(max_length=200, blank=False, null=False)
  body = models.TextField(blank=True, null=True)
  date = models.DateField()

  def __unicode__(self):
    return self.title
```

for each table we have in our database there is a corresponding class that inherits from **models.Model**
The properties of those classes represent the fields in each table.
Django helps us easily define the connections between our tables and we no longer need to use **SQL** queries directly and we can use those classes to perform action on our database.
Notice that we also provided a **__unicode__** override for those classes and this will help when we want to print a string representation of our class instance for debugging use or for django admin use.

Creating our tables in the database

We created the model classes but the tables in the database are not automatically created, we have to make sure to sync the database every time we are changing our models field table structure.
No more complex migration script, Django easily takes care for the database migrations.
To create the migration for the database according to the models we created we just need to tell django to create the migration by running:
> python manage.py makemigrations

after running the command you should see that the **migrations** package in your app **models_intro** now has a file called **0001_initial.py**
the numbered scripts in that package holds the migration scripts that django automatically created to turn the databse structure according the the **Model** classes we created in the **models.py** file.
Now if we run those scripts by order our database will migrate to the correct structure, and this action will work on different databases not just the default **sqlite** that we are using.

Now to run our migration script you need to type in the terminal:
> python manage.py migrate

Django will run all the migrations in your app and the apps that are installed via the **INSTALLED_APPS** array located at the **settings.py** file.

Viewing our database

If we look now on our project you will see a file is created for us with the database called **db.sqlite3**. There are many sqlite managers you can use to view that database, we recommend to use an sqlite plugin for firefox called **Sqlite Manager** which you can download in this link
If you look at the tables that were created for us, you will notice that our app tables are prefixed with our app name, for example: **models_intro_author**.
Also there are some table with our app prefix that exists and we didn't create like **models_intro_post_tags**
Django automatically created tables we need for our different connections between the tables, our **Post** and **Tag** tables are connected with **M2M** so we need a table to help us with this connection.

Creating rows in our tables

Let's try and use the model classes we created earlier to manipulate our database. To do this we will open a python shell with **manage.py** which will ensure that we run the shell after our settings.py is run and in the context of our project files.
In the terminal type:
> python manage.py shell

We can now use in the shell along with the model classes we created to perform actions on our database.
Let's first create a new row in the **Author** table. In the python shell run the following:

```
>>> from models_intro.models import *
>>> author1 = Author(first_name='John', last_name='Doe')
>>> author1.save()
```

check your database again in the authors table you should see the new item we just created.
let's try and modify the author we just created. Type the following to change the first name and last name of that author:

```
>>> author1.first_name = 'foo'
>>> author1.last_name = 'bar'
>>> author1.save()
```

check your database and you should see the row is updated. Notice that the row was updated only when you called the **save** method.
Let's try to get all the authors we have so far, in the interpreter type:

>>> Author.objects.all()

Let's try and delete the row we just created, type the following in the terminal:

>>> author1.delete()

now let's try and bulk create an array of authors:

```
>>> authors = []
>>> authors.append(Author(first_name='user1', last_name='user1'))
>>> authors.append(Author(first_name='user2', last_name='user2'))
>>> authors.append(Author(first_name='user3', last_name='user3'))
>>> authors.append(Author(first_name='user4', last_name='user4'))
```

You can refresh your database and verify that your rows are not created yet. Now to bulk create the rows type:

>>> Author.objects.bulk_create(authors)

check your database again and you should see that the rows of the authors are created.

Let's try and query the database to return all the rows with first_name is set to user1

>>> Author.objects.filter(first_name='user1')

the filter command returns an iterable list of **Author** that match the query, that iterable list is object of type **QuerySet**. You can also get a single item by calling the **get** but be aware that an exception can be raised if no argument match the query while **filter** will just return an empty **QuerySet.** You can pass the same arguments in **get** like **filter** only if there are multiple objects that match your query then **get** will raise an error.
Also note the **QuerySet** are lazy so you can construct them and only when you want to access the data the query to the database will be made.

Let's try and query the rows for **Author** with first_name set to user1 or user2.

>>> Author.objects.filter(first_name__in=['user1', 'user2'])

we saw above that we are not limited to a default exact match we can also use double underscore and specify the type of comparison, in this case we are placing the **in** to determine if the value is one of the above.
Another way we can **OR** query our Author table is by using a **Django Q** object.
With the **Q** object we can create complex queries on our database.
An example of using a **Q** object:

```
>>> from django.db.models import Q
>>> q = Q(first_name='user1') | Q(first_name='user2')
>>> Author.objects.filter(q)
```

So the **filter** or **get** methods can also get a **Q** object and with the **Q** object we can create complex queries and use operators on the **Q** objects to create a more complex query. In this example we used the **or** perator **|** between two **Q** objects.

You can also update a queryset and the changes will be to all the items in the queryset.
For example we can change the first name of the query above by running:

```
>>> Author.objects.filter(q).update(first_name='changed both users')
```

Now let's try and add rows to other tables and create connection. Let's create a new settings for our authors:

```
>>> settings1 = AuthorSettings.objects.create(is_invisible=True,
author=Author.objects.get(id=2))
```

Instead of creating an instance of the class **AuthorSettings** and then call the **save** method of that instance, we can just call **AuthorSettings.objects.create** which will create an instance and save it to the database.

you can access the user from the user settings instance

```
>>> print settings1.author.first_name
```

Let's create settings for all the authors we have:

```
>>> settings2 = AuthorSettings.objects.create(is_invisible=True,
author=Author.objects.get(id=3))
>>> settings3 = AuthorSettings.objects.create(is_invisible=False,
author=Author.objects.get(id=4))
>>> settings4 = AuthorSettings.objects.create(is_invisible=False,
author=Author.objects.get(id=5))
```

notice that if you accidently create a settings for a user that has a settings already then an error will be displayed.

We can query our tables based on related fields:

>>> AuthorSettings.objects.filter(author__gte=3)

this query will grab all the settings of the authors with **PK** greater or equal to 3.

You can also query from **Author** to **AuthorSettings** :

>>> Author.objects.get(authorsettings=3)

Let's try and examine the other tables and relationships that we have.
We will now create posts and attach them to our users, recall that **Author←>Post** is one to many relationship:

```
>>> import datetime
>>> Post.objects.create(author=Author.objects.get(first_name='user3'), body='1st post', title='numero uno', date=datetime.datetime.today())
>>> Post.objects.create(author=Author.objects.get(first_name='user3'), body='2nd post', title='another post from user3', date=datetime.datetime.today())
>>> Post.objects.create(author=Author.objects.get(first_name='user3'), body='3rd post', title='another post from user3', date=datetime.datetime.today())
```

we created here 3 posts for user3. We can query our posts based on the user to find all the posts from user4 (we should get an empty array):

>>> Post.objects.filter(author=5)

we can also get the posts of the user from the user object:

```
>>> user3 = Author.objects.get(first_name='user3')
>>> user3.post_set.all()
```

Django easily let's us filter by fields or relationships, construct advanced queries by using the **Q** object and manage our database queries more efficiently.
The next step will be to create a view that will create a new **Author**.

Creating Create Author View

We will now create a view with a form to create a new author:

We will start by connecting the **urls.py** with the urls of our app.
Modify **urls.py** like this:

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

urlpatterns = [
  url(r'^admin/', admin.site.urls),
  url(r'^', include(models_intro.urls))
]
```

We are including in the root url the urls from our app, so we now need to create the **urls.py** file in our **models_intro** package.
Create the file to look like this:

```
from django.conf.urls import url
from models_intro import views

urlpatterns = [
  url(r'^$', views.create_author)
]
```

We create a single url in our app that we will implement it shortly to display the form of the author.
Modify the **views.py** in the **models_intro** package to look like this:

```
from django.shortcuts import render
from django.http import HttpResponse
from django.forms import modelform_factory
from models_intro.models import Author

def create_author(request):
  AuthorForm = modelform_factory(Author, fields=('first_name', 'last_name'))
  if request.method == 'POST':
      form = AuthorForm(data=request.POST)
      form.save()
      return HttpResponse(content='successfully saved')
  return render(
      request=request,
      template_name='models_intro/create-author.html',
      context={'form': AuthorForm()}
  )
```

Django can automatically create a form for us based on our model. We are using the **modelform_factory** to create our form based on the **first_name** and **last_name** of the author. We are also dealing with the submit of the form in this view and in case this is a post method, meaning the user submitted the form we will use the form to create a new instance of our author.
If it's not a post we will render the template of the view and pass the form we created.
We will talk more about **Django Forms** on the next lessons.

Let's create the template for our view, create a folder in the **models_intro** called **templates** and create a folder called **models_intro** in the templates. It's common to create another directory with the app name so template names won't collide. Inside that folder create the file **create-author.html** with the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  <form method="post">
     {% csrf_token %}
     {{ form }}
     <button type="submit">
        Submit
     </button>
  </form>
</body>
</html>
```

in our template we are placing the **form** we passed from the view inside a **form** tag that sends a post request to the same url we are in.

Launch your app and try to submit the form we just created and you should see that a new **Author** is now created for us.

Summary

With **Django** models we can easily attach class that correspond to tables in our database, **Django** keeps our classes in sync with the tables by allowing us to easily create **migration** scripts after we change our classes. We can also easily manage relations between our tables and create forms for our models.