Object Oriented Software Engineering (SE313)
Lab Session # 10

Objective Using Model, Template & Views (MTV) in python framework (Django)

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

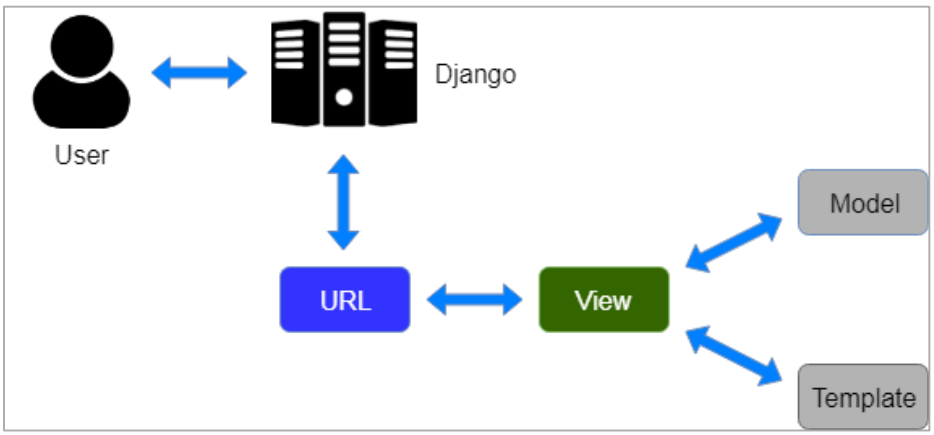
Model, Template & Views (MTV)

Django is based on MVT (Model-View-Template) architecture.

The **Template** is a presentation layer which handles User Interface part completely.

The **View** is used to execute the business logic and interact with

a **model** to carry data and renders a template.



Setting up Model

The model is going to act as the interface of your data. It is responsible for maintaining data. It is the logical data structure behind the entire application and is represented by a database (generally relational databases such as MySql, Postgres).

Database setup

To setup the database, create your first model, open up **mysite/settings.py**. It's a normal Python module with module-level variables representing Django settings.

By default, the configuration uses **SQLite**, SQLite is included in Python, so you won't need to install anything else to support your database

By default, **INSTALLED_APPS** contains the following apps, all of which come with Django:

- **django.contrib.admin** – The admin site. You'll use it shortly.
- **django.contrib.auth** – An authentication system.
- **django.contrib.contenttypes** – A framework for content types.
- **django.contrib.sessions** – A session framework.
- **django.contrib.messages** – A messaging framework.
- **django.contrib.staticfiles** – A framework for managing static files.

Some of these applications make use of at least one database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

```
...\> py manage.py migrate
```

The **migrate** command looks at the **INSTALLED_APPS** setting and creates any necessary database tables according to the database settings in your **mysite/settings.py** file and the database migrations shipped with the app

You'll see a message for each migration it applies. You will the following result

```
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying sessions.0001_initial... OK
```

Creating models

The **Django** web framework includes a default Object-Relational Mapping layer (**ORM**) that can be used to interact with application data from various relational databases such as SQLite, PostgreSQL and MySQL

This Object Relational Mapper (**ORM**) tool that translates **Python** classes to tables on relational databases and automatically

Now we'll define your models – essentially, your database layout, with additional metadata.

In our poll app, we'll create two models: **Question** and **Choice**.

- A **Question** has a question and a publication date.
- A **Choice** has two fields: the text of the choice and a vote tally. Each **Choice** is associated with a **Question**.

These concepts are represented by Python classes. Edit the **polls/models.py** file so it looks like this:

```
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
  votes = models.IntegerField(default=0)
```

Here, each model is represented by a class that subclasses **django.db.models.Model**. Each model has a number of class *variables*, each of which represents a database field in the model.

Activating models

To include the app in our project, we need to add a reference to its configuration class in the **INSTALLED_APPS** setting.

The **PollsConfig** class is in the **polls/apps.py** file, so its dotted path is **'polls.apps.PollsConfig'**. Edit the **mysite/settings.py** file and add that dotted path to the **INSTALLED_APPS** setting. It'll look like this:

```
INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Now Django knows to include the **polls** app. Let's run another command:

```
...\> py manage.py makemigrations polls
```

By running **makemigrations**, you're telling Django that you've made some changes to your models (in this case, you've made new ones)

You can read the migration for your new model if you like; it's the file **polls/migrations/0001_initial.py**

You should see something similar to the following:

```
Migrations for 'polls':
  polls/migrations/0001_initial.py:
    - Create model Choice
    - Create model Question
    - Add field question to choice
```

The **sqlmigrate** command takes migration names and returns their SQL:

```
...\> py manage.py sqlmigrate polls 0001
```

You should see something similar to the following (we've reformatted it for readability):

```
BEGIN;
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
--
-- Create model Question
--
```

If you're interested, you can also run **python manage.py check**; this checks for any problems in your project without making migrations or touching the database.

Now, run **migrate** again to create those model tables in your database:

```
...\> py manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK
```

The **migrate** command takes all the migrations that haven't been applied and runs them against your database - essentially, synchronizing the changes you made to your models with the schema in the database.

Migrations are very powerful and let you change your models over time, as you develop your project, without the need to delete your database or tables and make new ones - it specializes in upgrading your database live, without losing data.

Remember the three-step guide to making model changes:

Change your models (in **models.py**).
Run **python manage.py makemigrations**
Run **python manage.py migrate**

Now, let's hop into the interactive Python shell and play around with the free API Django gives you. To invoke the Python shell, use this command:

```
...\> py manage.py shell
```

Once you're in the shell, you can write the following commands in (InteractiveConsole)

```
>>> from polls.models import Choice, Question  # Import the model classes we just wrote.
# No questions are in the system yet.
>>> Question.objects.all()
<QuerySet []>

# Create a new Question.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
# instead of datetime.datetime.now() and it will do the right thing.

>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.
>>> q.save()

# Now it has an ID.
>>> q.id
1
```

Editing the **Question** model (in the **polls/models.py** file) and adding a **__str__()** method to both **Question** and **Choice**:

```
from django.db import models

class Question(models. Model):

    # ...

    def __str__(self):

        return self.question_text

class Choice(models.Model):

    # ...

    def __str__(self):

        return self.choice_text
```

It's important to add **__str__()** methods to your models, not only for your own convenience when dealing with the interactive prompt, but also because objects' representations are used throughout Django's automatically-generated admin.

Let's also add a custom method to this model:

```
import datetime
from django.db import models
from django.utils import timezone
class Question(models.Model):

     def was_published_recently(self):
         return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Note the addition of **import datetime** and **from django.utils import timezone**, to reference Python's standard **datetime** module and Django's time-zone-related utilities in **django.utils.timezone**,

Save these changes and start a new Python interactive shell by running **python manage.py shell** again:

```
from polls.models import Choice, Question

# Make sure our custom method worked.
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
True

# Give the Question a couple of Choices. The create call constructs a new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a question's choice) which can be accessed via the API.
>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
<QuerySet []>

# Create three choices.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>
>>> q.choice_set.count()
3
```

Writing more views

In Django, web pages and other content are delivered by views. Each view is represented by a Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that's requested (to be precise, the part of the URL after the domain name).

Now let's add a few more views to **polls/views.py**. These views are slightly different, because they take an argument:

```
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

Wire these new views into the **polls.urls** module by adding the following **path()** calls:

```
from django.urls import path

from . import views

urlpatterns = [
    # ex: /polls/
    path('', views.index, name='index'),
    # ex: /polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
    # ex: /polls/5/results/
    path('<int:question_id>/results/', views.results, name='results'),
    # ex: /polls/5/vote/
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Creating a template

So let's use Django's template system to separate the design from Python by creating a template that the view can use.

First, create a directory called **templates** in your **polls** directory. Django will look for templates in there.

our template should be at **polls/templates/polls/index.html**.

```
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{
question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Now let's update our **index** view in **polls/views.py** to use the template:

```
from django.http import HttpResponse
from django.template import loader

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponse(template.render(context, request))
```

That code loads the template called **polls/index.html** and passes it a context. The context is a dictionary mapping template variable names to Python objects.

Load the page by pointing your browser at "/polls/", and you should see a bulleted-list containing the "**What's up**" question

Raising a 404 error

Now, let's tackle the question detail view – the page that displays the question text for a given poll. Here's the view:

```
from django.http import Http404
from django.shortcuts import render

from .models import Question
# ...
def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("This Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})
```

Use the template system

Back to the **detail()** view for our poll application. Given the context variable **question**, here's what the **polls/detail.html** template might look like:

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

Student Task

Create a small app containing all concept which we discuss in lab 9 & 10