

UNIT-II

1. Introduction to Interacting with the Database Using Query Sets

Ans: Query sets are used in views to manipulate data in the database programmatically and they also allow fetching and saving data from/to the database.

2. ORM Overview

Ans: Django provides a built-in interface for admin functionalities, allowing easy management of content.

- Django's ORM provides a powerful database abstraction API.
- The Django ORM is compatible with popular databases like MySQL, PostgreSQL, SQLite, and Oracle.
- You can define the database for your project in the settings.py file.
- Django supports working with multiple databases simultaneously and allows customization through database routers.

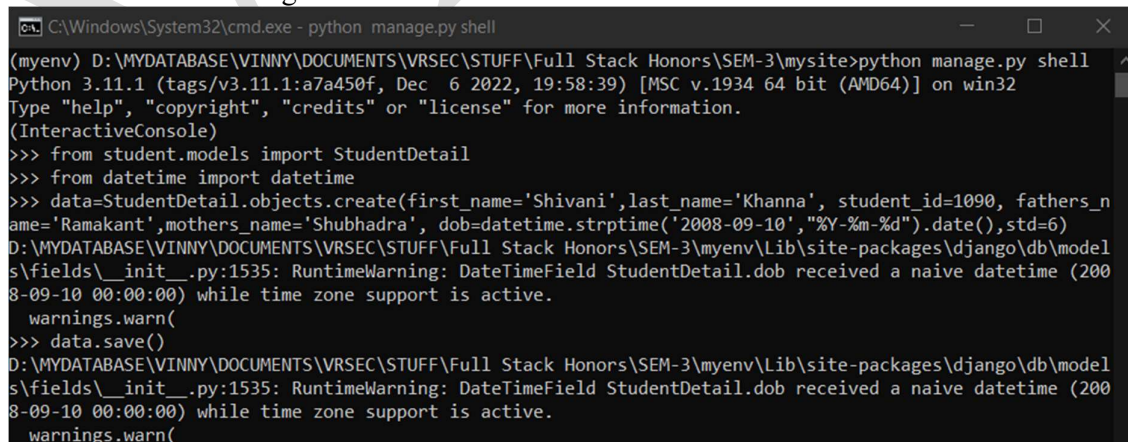
3. Query sets

Ans: QuerySet is a set of commands that need to be evaluated to interact with the database and perform database operations.

- Following are some operations on a QuerySet:
 - **Iteration:** Iterate over a QuerySet to access each record.
 - **Slicing:** Slice a QuerySet using indices.
 - **repr():** Display results in the Python interactive interpreter.
 - **len():** Get the length of the QuerySet.
 - **list():** Save QuerySet results in a list.
 - **bool():** Check if the QuerySet has any results.
- Following are some common QuerySet Methods:
 - **Filter:** Retrieve results matching specific conditions.
 - **Exclude:** Retrieve results excluding specific conditions.
 - **Latest:** Retrieve the most recent object that meets the conditions.
 - **Update:** Update records in the table and return the number of rows affected.
 - **Delete:** Delete objects from the table and return the number of deletions.
 - **Get:** Retrieve a single object matching the conditions (raises exception if multiple objects found).

4. Adding elements to your database

Ans: Execute the following commands to add data into the database



```
C:\Windows\System32\cmd.exe - python manage.py shell
(myenv) D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-3\mysite>python manage.py shell
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from student.models import StudentDetail
>>> from datetime import datetime
>>> data=StudentDetail.objects.create(first_name='Shivani',last_name='Khanna', student_id=1090, fathers_n
ame='Ramakant',mothers_name='Shubhadra', dob=datetime.strptime('2008-09-10','%Y-%m-%d').date(),std=6)
D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-3\myenv\Lib\site-packages\django\db\models
s\fields\__init__.py:1535: RuntimeWarning: DateTimeField StudentDetail.dob received a naive datetime (200
8-09-10 00:00:00) while time zone support is active.
  warnings.warn(
>>> data.save()
D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-3\myenv\Lib\site-packages\django\db\models
s\fields\__init__.py:1535: RuntimeWarning: DateTimeField StudentDetail.dob received a naive datetime (200
8-09-10 00:00:00) while time zone support is active.
  warnings.warn(
```

- If you want to see whether the object is inserted into the database, run the following commands

```
C:\Windows\System32\cmd.exe - python manage.py shell
>>> all_students_list=StudentDetail.objects.all()
>>> all_students_list
<QuerySet [<StudentDetail: StudentDetail object (1)>, <StudentDetail: StudentDetail object (2)>,
<StudentDetail: StudentDetail object (3)>, <StudentDetail: StudentDetail object (4)>, <StudentDetail: StudentDetail object (5)>]>
>>> all_students_list[4].first_name
'Shivani'
```

5. Manipulating elements of your database

Ans: Execute the following commands to fetch the details of a particular id

```
C:\Windows\System32\cmd.exe - python manage.py shell
>>> data=StudentDetail.objects.filter(student_id=1090)
>>> data
<QuerySet [<StudentDetail: StudentDetail object (5)>]>
>>> data[0].mothers_name
'Shubhadra'
>>> data[0].fathers_name
'Ramakant'
```

- Execute the following commands to modify a particular field

```
C:\Windows\System32\cmd.exe - python manage.py shell
>>> StudentDetail.objects.filter(student_id=1090).update(mothers_name='Shubadra Khanna')
1
>>> data[0].mothers_name
'Shubadra Khanna'
```

- Execute the following commands to iterate over the query set

```
C:\Windows\System32\cmd.exe - python manage.py shell
>>> for obj in data:
...     obj.fathers_name='Ramakant Khanna'
...     obj.save()
...
>>> data[0].fathers_name
'Ramakant Khanna'
```

6. Deleting elements of your database

Ans: Execute the following commands to delete a record and verify its existence

```
C:\Windows\System32\cmd.exe - python manage.py shell
>>> data=StudentDetail.objects.get(student_id=1090)
>>> data.delete()
(1, {'student.StudentDetail': 1})
>>> all_students_list=StudentDetail.objects.all()
>>> all_students_list
<QuerySet [<StudentDetail: StudentDetail object (1)>, <StudentDetail: StudentDetail object (2)>,
<StudentDetail: StudentDetail object (3)>, <StudentDetail: StudentDetail object (4)>]>
```

7. Introduction to models

Ans: Django's models and their features are crucial for storing and retrieving data in your application.

- The Django documentation on models is extensive, covering various aspects of model classes, fields, and their options.
- Understanding how model classes and their relationships (one-to-one, many-to-many, many-to-one) generate database tables is important.
- Models play a central role in the database schema design and interaction in Django.
- The **models.py** file typically includes import statements, model classes with fields, subclasses, and methods, forming the basic structure of the file.

8. Model fields

Ans: There are several types of model fields like integer, varchar, text, etc. just like the type of columns/attributes of a database table. Following are some of the model field types

- **AutoField:** Auto-generated primary key field.
- **BooleanField:** True/False field.
- **CharField:** Field for storing text.
- **DateField:** Field for storing dates.
- **DateTimeField:** Field for storing date and time.
- **DecimalField:** Field for storing decimal values.
- **FloatField:** Field for storing floating-point numbers.
- **EmailField:** Field for storing email addresses.
- **FileField:** Field for uploading files.
- **ImageField:** Field for uploading images.
- **IntegerField:** Field for storing integers.
- **SlugField:** Field for generating URL-friendly strings.
- **Relationship fields:** Fields for establishing relationships between models.
- Django's official documentation provides detailed information on all model fields.

Ex:

```
from django.db import models
class ExampleModel(models.Model):
    auto_field = models.AutoField(primary_key=True)
    boolean_field = models.BooleanField(default=False)
    char_field = models.CharField(max_length=100)
    date_field = models.DateField()
    datetime_field = models.DateTimeField()
    decimal_field = models.DecimalField(max_digits=5,
decimal_places=2)
    float_field = models.FloatField()
    email_field = models.EmailField()
    file_field = models.FileField(upload_to='files/')
    image_field = models.ImageField(upload_to='images/')
    integer_field = models.IntegerField()
    slug_field = models.SlugField()
    # Relationship field
    foreign_key = models.ForeignKey('AnotherModel',
on_delete=models.CASCADE)
class AnotherModel(models.Model):
    # Fields for AnotherModel
    pass
```

9. Meta options

Ans: Meta options in Django models provide metadata about the model class.

- Following are some commonly used meta options
 - **app_label** is used when using a model class outside the app.
 - **default_manager_name** allows specifying a custom model manager.
 - **db_table** sets the name of the table for the model.
 - **get_latest_by** determines the field used to retrieve the most recent objects.
 - **managed** controls if the model is translated into a table in the database.
 - **ordering** sets the default ordering of objects in retrieval queries.
 - **unique_together** specifies fields that must be unique together.
 - **verbose_name** sets a readable name for the model.
 - **verbose_name_plural** defines the plural name for the model on the UI.
 - There are additional Meta options available in the Django documentation.

Ex:

```
from django.db import models
class ExampleModel(models.Model):
    field1 = models.CharField(max_length=50)
    field2 = models.IntegerField()
    class Meta:
        app_label = 'myapp'
        default_manager_name = 'objects'
        db_table = 'my_table'
        get_latest_by = 'field1'
        managed = True
        order_with_respect_to = 'field2'
        ordering = ['field1', '-field2']
        unique_together = ('field1', 'field2')
        verbose_name = 'Example'
        verbose_name_plural = 'Examples'
```

10. Model methods

Ans: Model managers are classes implemented at the table level, while model methods are implemented at the row/instance level.

- Model methods are functions defined within the scope of the model class and can be used to perform various operations on individual model instances.
- Model methods are invoked using the objects/instances of the model class.
- Model methods can be used for basic tasks like object creation or for advanced purposes like generating absolute URLs.
- Model methods can be used in views, templates, or any other file where the model instance is available.

Ex:

```
from django.db import models
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    publication_year = models.IntegerField()
    def is_published(self):
        current_year = datetime.date.today().year
        return self.publication_year <= current_year
book = Book(title='Sample Book', author='John Doe',
publication_year=2020)
print(book.is_published()) # Output: True
```

11. Relationship between models

Ans: Model relationships are used to implement the relational database concept and avoid repetitive storage of data and they are of three types

- **Many-to-one** relationship is implemented using the ForeignKey field, which establishes a relationship where one object can be related to multiple objects.
- **One-to-one** relationship is implemented using the OneToOne field, which establishes a relationship where one object is related to exactly one object.
- **Many-to-many** relationship is implemented using the ManyToMany field, which establishes a relationship where multiple objects can be related to multiple objects.
- The ForeignKey field has options like `on_delete` to specify the behavior when the source object is deleted, and `limit_choices_to` to limit the choices available for the field value.
- The ManyToMany field creates an intermediate model to represent the relationship between two models.

- The OneToOne field is used to connect two tables with information about similar things that cannot be stored in a single table.
- Model relationships provide a way to link tables and manage connected data efficiently.

Ex:

```
from django.db import models
class Teacher(models.Model):
    name = models.CharField(max_length=100)
    def __str__(self):
        return self.name
class Subject(models.Model):
    name = models.CharField(max_length=100)
    def __str__(self):
        return self.name
class Student(models.Model):
    name = models.CharField(max_length=100)
    teachers = models.ManyToManyField(Teacher)
    subjects = models.ManyToManyField(Subject)
    teacher = models.ForeignKey(Teacher, on_delete=models.CASCADE)
    student_detail = models.OneToOneField('StudentDetail',
on_delete=models.CASCADE)
    def __str__(self):
        return self.name
class StudentDetail(models.Model):
    student = models.OneToOneField(Student,
on_delete=models.CASCADE)
    address = models.CharField(max_length=100)
    def __str__(self):
        return self.address
```

12. Connecting models

Ans: You can connect models from different apps or external files by importing the model class using the **from app_name.models import ModelName** syntax.

- Ensure that the app containing the model you want to import is included in the **INSTALLED_APPS** list in your project's settings.
- Once imported, you can use the imported model class just like any other model class within your app.
- This allows you to reuse model classes across different apps or separate files, promoting code organization and reusability.
- Make sure to maintain proper relationships and dependencies between the models when connecting them from different sources.

13. Introduction to Django views

Ans: Views in Django handle the logic of a web application and are invoked when a URL is matched with a corresponding url-pattern.

- Views interact with the database, perform data fetching/insertion/update, and respond with a template and context data to be rendered on the page.
- The views.py file contains import statements and view functions or classes.
- View functions accept an HTTP request and process it, returning an HTML template as a response.
- Views can be organized into classes with multiple methods and subclasses.
- Views play a crucial role in implementing the business logic of a Django project.

14. Types of views

Ans: Function-Based Views: Simple views defined as functions.

Class-Based Views (CBVs): Views defined as classes, offering more structure and reusability.

- **Built-in Class-Based Views:** Predefined class-based views provided by Django.
- **Base View:** The base class for other class-based views, providing common functionality.
- **View:** A base class for views handling HTTP requests and responses.
- **Template View:** Renders a template along with context data.
- **Redirect View:** Redirects to a specified URL.
- **Generic Views:** Predefined class-based views for common use cases.
- **List View:** Displays a list of objects from a model.
- **Detail View:** Displays details of a specific object from a model.

Here are the syntax examples:

1. Function-Based View:

```
from django.http import HttpResponse
def function_based_view(request):
    return HttpResponse("This is a Function-Based View.")
```

2. Class-Based View:

```
from django.views import View
from django.http import HttpResponse
class ClassBasedView(View):
    def get(self, request):
        return HttpResponse("This is a Class-Based View.")
```

3. Template View:

```
from django.views.generic import TemplateView
class MyTemplateView(TemplateView):
    template_name = 'my_template.html'
```

4. Redirect View:

```
from django.views.generic.base import RedirectView
class MyRedirectView(RedirectView):
    url = '/new-url/'
```

5. List View (Generic View):

```
from django.views.generic.list import ListView
from .models import MyModel
class MyListView(ListView):
    model = MyModel
    template_name = 'my_list.html'
```

6. Detail View (Generic View):

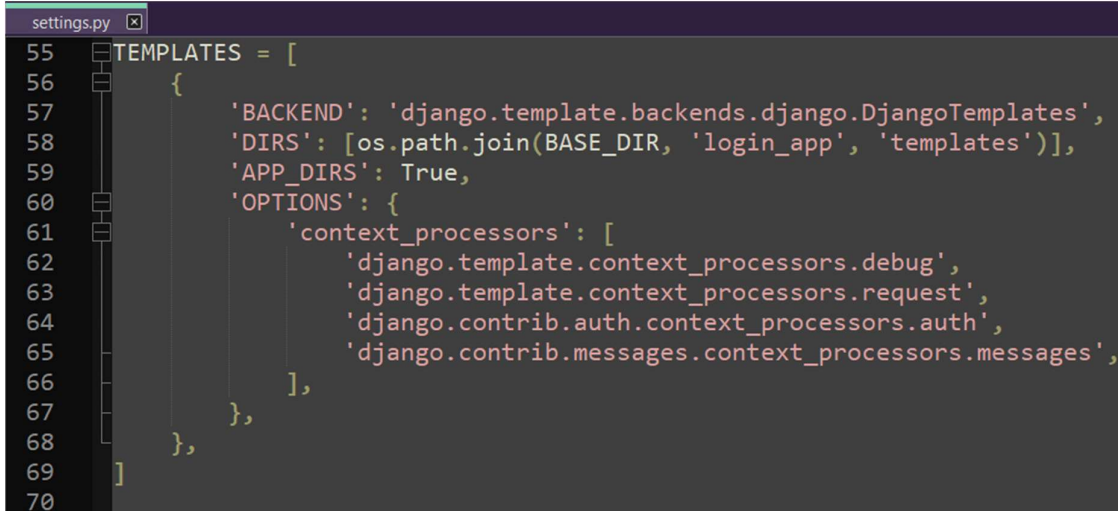
```
from django.views.generic.detail import DetailView
from .models import MyModel
class MyDetailView(DetailView):
    model = MyModel
    template_name = 'my_detail.html'
```

15. Introduction to templates

Ans: The Django Templating Language is used to create templates, which play a crucial role in incorporating dynamic data into web pages. Within these templates, any content enclosed within '{' and '}' symbols is considered Python code, forming the core of the Django Templating Language. Everything else within the templates is written in HTML.

16. Configuration

Ans: Configuration of templating engines is done in the settings.py file under the TEMPLATES tag.



```
55 TEMPLATES = [
56     {
57         'BACKEND': 'django.template.backends.django.DjangoTemplates',
58         'DIRS': [os.path.join(BASE_DIR, 'login_app', 'templates')],
59         'APP_DIRS': True,
60         'OPTIONS': {
61             'context_processors': [
62                 'django.template.context_processors.debug',
63                 'django.template.context_processors.request',
64                 'django.contrib.auth.context_processors.auth',
65                 'django.contrib.messages.context_processors.messages',
66             ],
67         },
68     },
69 ]
70
```

- The BACKEND key specifies the path to the template engine class; for Django Templating Language, it's DjangoTemplates.
- DIRS indicate where the engine should find templates, typically structured by app name.
- APP_DIRS set to true allows the engine to search for templates within individual app directories
- OPTIONS can hold advanced settings for the template backend, relevant for complex projects.

17. Template inheritance

Ans: Template inheritance allows child templates to inherit content from parent templates.

- It's useful for avoiding redundant HTML code, such as styling, headers, and footers, by inheriting common elements.
- The 'extends' tag is used to establish template inheritance.
- Child templates can override parent template content when necessary.
- If a property exists in both the child and parent templates, the child's property takes precedence.
- Inheritance creates a structure where child templates fill in data using block tags.
- Base templates serve as skeletons that child templates populate with content.

Ex:

Base.html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    <header>
      <h1>Welcome to My Website</h1>
    </header>
    <main>
      {% block content %}{% endblock %}
    </main>
    <footer>
      <p>&copy; 2023 My Website</p>
    </footer>
  </body>
</html>
```

Child.html:

```
{% extends "Base.html" %}
{% block content %}
    <h1>This is the child template content</h1>
{% endblock %}
```

18. Django Template Language

Ans: Django Templating Language is used for creating templates that incorporate dynamic data into web pages and it combines Python strings and HTML to generate dynamic HTML responses.

Syntax:

- Variables are enclosed in double curly braces, like `{{ variable_name }}`, and return values from the context passed by views.
- Tags are enclosed in curly braces with a percentage sign, like `{% tag_name %}`, and allow the execution of Python code within templates.

Using Variables:

- Variables are replaced with values from the context when the template is executed.
- Example: `{{ student.get_absolute_url }}`, `{{ student.first_name }}`.

Using Tags:

- Tags perform various actions within templates.
- Examples include `extends`, `block`, `for`, `empty`, `if`, `else`, `and`, `or`, `not`.

Filters:

- Filters modify variable values for display, such as `|date:"Y-m-d"` for date formatting.
- Filters like `default`, `default_if_none`, and `time` are available.

Comments:

- Comments are written using `{# comment #}` and won't be rendered in the HTML output.

Ex:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Django Templating Example</title>
  </head>
  <body>
    <!-- This is a comment -->
    <h1>Welcome, {{ user_name }}!</h1>
    {% if is_admin %}
      <p>You have admin privileges.</p>
    {% else %}
      <p>You do not have admin privileges.</p>
    {% endif %}
    <ul>
      {% for item in my_list %}
        <li>{{ item }}</li>
      {% empty %}
        <li>No items to display.</li>
      {% endfor %}
    </ul>
    <p>Date: {{ current_date|date:"Y-m-d" }}</p>
    <p>Default Value: {{ non_existent_variable|default:"N/A"
  }}</p>
    <p>Time: {{ current_time|time:"H:i:s" }}</p>
  </body>
</html>
```