Template System Basics, Using Django Template System, Basic Template Tags and Filters, MVT Development Pattern, Template Loading, Template Inheritance, MVT Development Pattern. Configuring Databases, Defining and Implementing Models, Basic Data Access, Adding Model String Representations, Inserting/Updating data, Selecting and deleting objects, Schema Evolution

1. Explain MVT Architecture.
2. Explain Template Inheritance with an example.
3. Explain Django and Python Templates with an example.
4. Develop a simple Django app that displays an unordered list of fruits and ordered list of selected students for an event

Template System Basics:

What is a Template System?

- A template system in web development is used to generate HTML dynamically.
- Templates contain static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.
- To separate the presentation layer (HTML, CSS) from the business logic layer (Python code).
- Separation of Concerns: Keeps the business logic separate from the presentation.
- Reusability: Templates can be reused across different parts of the application.
- Maintainability: Easier to maintain and update the presentation layer without affecting the business logic.

Template Syntax

- Template Tags are enclosed within {{ }} braces like this <p>Hello, {{ user.username }}!</p>

Basic Example:

```
<html>
<head><title>Ordering notice</title></head>

<body>

<p>Dear {{ person_name }},</p>


<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>

<p>Here are the items you've ordered:</p>

<ul>
{% for item in item_list %}
<li>{{ item }}</li>
{% endfor %}
</ul>

{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% endif %}

<p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

In above Example Any text surrounded by a pair of braces (e.g., {{ person_name }}) is a variable.

Any text that's surrounded by curly braces and percent signs (e.g., {% if ordered_ warranty %}) is a template tag.

This example template contains two tags: the {% for item in item_list %} tag (a for tag) and the {% if ordered_warranty %} tag (an if tag). A for tag acts as a simple loop construct, letting you loop over each item in a sequence. An if tag, as you may expect, acts as a logical "if" statement. In this particular case, the tag checks whether the value of the ordered_warranty variable evaluates to True. If it does, the template system will display everything between the {% if ordered_warranty %} and {% endif %}. If not, the template system won't display it. The template system also supports {% else %} and other various logic statements.

Using the Template System

Step1: Setup Django App As suggested in Module 1

Step2: Creating and Rendering Templates from Files

1. Create a Template Directory

Create a directory named templates inside your app directory (e.g., myapp/templates).

2. Create a Template File

Inside the templates directory, create an HTML file, for example, home.html.

```
<!-- myapp/templates/home.html -->
<!DOCTYPE html>
<html>
<head>
   <title>{{ title }}</title>
</head>
<body>
   <h1>Hello, {{ name }}!</h1>
   <p>Welcome to {{ site_name }}.</p>
</body>
</html>
```

Step 3: Load and Render the Template in a View

In your views.py file, use Django's render function to load and render the template.

```
# myapp/views.py
from django.shortcuts import render


def home_view(request):
   context = {
      'title': 'Home',
        'name': 'John',

        'site_name': 'My Awesome Site'

    }

    return render(request, 'home.html', context)
```

Step 4: Map the View to a URL

In your urls.py file, map the view to a URL.

```
# myapp/urls.py

from django.urls import path

from .views import home_view



urlpatterns = [

    path('', home_view, name='home'), ]
```

Make sure to include your app's urls.py in the project's main urls.py file:

```
# myproject/urls.py

from django.contrib import admin

from django.urls import include, path



urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include('myapp.urls')),

]
```

Step 5: Access the URL

```
python manage.py runserver
```

Basic Template Tags and Filters

Variables

Variables in Django templates are enclosed in double curly braces {{ }}. They are used to output dynamic data.

Example

```
<p>Hello, {{ user.username }}!</p>
```

Filters

Filters are used to modify the display of variables. They are applied using a pipe | symbol.

Common Filters

- date: Formats dates.

  ```
  {{ value|date:"D d M Y" }}
  ```

- lower: Converts a string to lowercase.

  ```
  {{ name|lower }}
  ```

- length: Returns the length of a string or list.

  ```
  {{ list|length }}
  ```

- truncatewords: Truncates a string after a certain number of words.

  ```
  {{ value|truncatewords:30 }}
  ```

Tags

Tags provide arbitrary logic in the rendering process. They are enclosed in {% %} and are used for things like loops and conditional statements.

Common Tags

- {% if %}: Conditional statements.

```
{% if user.is_authenticated %}
  <p>Welcome, {{ user.username }}!</p>
{% else %}
  <p>Please log in.</p>
{% endif %}
```

- {% for %}: Loop through a list.

```
<ul>
{% for author in authors %}
  <li>{{ author.name }}</li>
{% endfor %}
</ul>
```

- {% block %}: Define a block of content in a template (used in template inheritance).

```
{% block content %}
  <p>This is the content block.</p>
{% endblock %}
```

- {% extends %}: Extend a base template (used in template inheritance).

```
{% extends "base.html" %}
```

- {% include %}: Include another template.

```
{% include "footer.html" %}
```

MVT Development Pattern

Overview

The MVT pattern in Django is a variation of the traditional MVC (Model-View-Controller) pattern. In Django:

- **Model** handles the data and database.
- **View** processes user input and returns the output.
- **Template** presents the data in an HTML format.
-

Components of MVT

1. **Model**: Represents the data structure. It is an abstraction of the database and handles the database operations.
2. **View**: Contains the business logic. It processes requests and returns responses, often using models and templates.
3. **Template**: Controls the presentation layer. It defines how the data is displayed to the user.

MVT Flow

1. **User Request**: The user sends an HTTP request to the Django application.
2. **URL Dispatcher**: The URL dispatcher maps the request to the appropriate view based on the URL pattern.
3. **View Logic**: The view processes the request, interacts with the model to fetch data, and passes the data to the template.
4. **Template Rendering**: The template renders the data into HTML and sends the response back to the user.

1. Model

The model defines the structure of the database and the data handling logic. In Django, models are defined as classes.

```
# myapp/models.py

from django.db import models

class Book(models.Model):

    title = models.CharField(max_length=100)

    author = models.CharField(max_length=100)

    published_date = models.DateField()

    def __str__(self):

        return self.title
```

2. View
The view contains the logic to handle the user request. It fetches data from the model, applies business logic, and selects a template for rendering.

```
# myapp/views.py
```

```
from django.shortcuts import render

from .models import Book

def book_list(request):

    books = Book.objects.all()

    return render(request, 'book_list.html', {'books': books})
```

In this example, the book_list view fetches all book records from the database and passes them to the book_list.html template for rendering.

3. Template

The template defines how the data is presented to the user. It uses Django's template language to dynamically display data.

```
<!-- myapp/templates/book_list.html -->

<!DOCTYPE html>

<html>

<head>

    <title>Book List</title>

</head>

<body>

    <h1>Book List</h1>

    <ul>

    {% for book in books %}

        <li>{{ book.title }} by {{ book.author }} (Published on {{ book.published_date }})</li>

    {% endfor %}

    </ul>

</body>

</html>
```

URL Configuration

The URL configuration maps URLs to the corresponding view functions.

```
# myapp/urls.py
```

```
from django.urls import path

from .views import book_list

urlpatterns = [

    path('books/', book_list, name='book_list'),]
```

Main URL Configuration:

```
# myproject/urls.py
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),
]
```

Template Loading

Template loading in Django is the process of locating and loading the template files that are used to render HTML pages. Django looks for these templates in specific directories configured in your project.

1. **Template Directories:**

   Django searches for templates in the following directories by default:

   Within each app:

   Create a directory named templates within each app directory to store app-specific templates.

2. **Template Loading**

   Django uses a template loader to find and load templates from these directories. The template loader searches for templates in the specified directories based on the order defined in the TEMPLATES setting in your project's settings.py file.

3. **Templates Setting**

```
# settings.py

import os


BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

TEMPLATES = [
  {
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [os.path.join(BASE_DIR, 'templates')],  # Directory for project-wide
templates
    'APP_DIRS': True,  # Automatically looks for templates in each app's "templates"
folder
```

```
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },]
```

Template Inheritance

Create a base template that includes the common layout and structure for your site. This template will define "blocks" that child templates can override.

- Define a base template with the overall structure of your website (header, footer, navigation).
- Create child templates that inherit from the base template and focus on specific page content.
- Child templates can override sections of the base template using block tags.
- **{% extends %}:** Used in the child template to specify the base template it inherits from.
- **{% block %}:** Defines a section in the base template that can be overridden by child templates.
- **{% endblock %}:** Marks the end of a block definition.

Example:

```
<!DOCTYPE html>
<html>
<head>
    <title>My Website</title>
</head>
<body>
    <header>
        </header>
    <nav>
        </nav>
    <main>
        {% block content %}
        {% endblock %}
    </main>
    <footer>
        </footer>
</body>
</html>
```

```
{% extends "base.html" %}

{% block content %}
<h1>About Us</h1>
<p>This is the about us page content.</p>
{% endblock %}
```

Configuring Databases

Django uses the DATABASES setting in the settings.py file to define database configurations. This setting is a dictionary where you specify the details of your database connection.

1. Default Configuration (SQLite)

When you create a new Django project, the default database configuration is set to SQLite.

Example Configuration:

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Configuring MySQL

To use MySQL, you need to install the mysqlclient library and configure the database settings.

To use this first install mysql client **pip install mysqlclient**

```
# settings.py

DATABASES = {

    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'your_db_name',
        'USER': 'your_db_user',
        'PASSWORD': 'your_db_password',
        'HOST': 'localhost',  # Set to 'localhost' for local development
        'PORT': '3306',  # Default port for MySQL
    }
}
```

After configuring your database settings, you need to run the database migrations to create the necessary tables. Using command **python manage.py migrate**

After this run development server python manage.py runserver

Defining and Implementing Models

In Django, a model is a class that inherits from django.db.models.Model and defines fields as class attributes. Each attribute represents a database field. Django automatically translates these Python classes into database tables.

1. Create a Model

Define your models by creating a class in your app's models.py file. Each class attribute represents a database field.

```
# myapp/models.py
from django.db import models


class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13, unique=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    def __str__(self):
        return self.title
```

2.Model Fields

Django provides various field types to define the data you want to store. Here are some common fields:

- CharField: For short to medium-length strings.
- TextField: For large text fields.
- IntegerField: For integer numbers.
- FloatField: For floating-point numbers.
- DecimalField: For fixed-precision decimal numbers.
- BooleanField: For Boolean values.
- DateField and DateTimeField: For dates and date-time values.
- ForeignKey: For creating a many-to-one relationship.
- ManyToManyField: For creating a many-to-many relationship.

3. String Representation

The **__str__** method defines the human-readable representation of the model. It's useful for displaying model instances in the Django admin interface and elsewhere.

4. Apply Migrations

After defining your models, you need to create and apply migrations. Migrations are Django’s way of propagating changes you make to your models (like adding a field or deleting a model) into your database schema.

```
python manage.py makemigrations

python manage.py migrate
```

5. Basic Data Access

Django's ORM (Object-Relational Mapping) provides an easy way to interact with the database. You can use Django's built-in methods to create, retrieve, update, and delete records.

Create a Record:

```
from myapp.models import Book

book = Book.objects.create(

    title="The Great Gatsby",
    author="F. Scott Fitzgerald",
    published_date="1925-04-10",
    isbn="9780743273565",
    price=10.99
)
```

Retrieve Records:

```
# Retrieve all books
books = Book.objects.all()
# Retrieve a specific book by its primary key
book = Book.objects.get(pk=1)
# Retrieve books with specific criteria
books_by_author = Book.objects.filter(author="F. Scott Fitzgerald")
```

Update a Record:

```
book = Book.objects.get(pk=1)
book.price = 12.99
book.save()
```

Delete a Record:

```
book = Book.objects.get(pk=1)
book.delete()
```

6. Adding Model String Representations

The __str__ method in the model class is used to provide a string representation of the model instances, which is useful in the Django admin site and the shell.

```
class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13, unique=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    def __str__(self):
        return f"{self.title} by {self.author}"
```

7. Inserting/Updating Data

You can insert data using the create() method or by creating an instance and calling the save() method. Updates are done by modifying the attributes and saving the instance.

Inserting Data:

```
new_book = Book(
    title="1984",
    author="George Orwell",
    published_date="1949-06-08",
    isbn="9780451524935",
    price=9.99
)
new_book.save()
```

Update data:

```
book = Book.objects.get(pk=1)
book.price = 14.99
book.save()
```

Selecting and Deleting Objects

Django provides methods for selecting and deleting records in the database.

Selecting Objects:

```
# Get all books
all_books = Book.objects.all()
# Get a single book by ID
book = Book.objects.get(id=1)
# Filter books by a field
filtered_books = Book.objects.filter(author="George Orwell")
```

Deleting Objects:

```
book = Book.objects.get(id=1)
```

book.delete()

Schema Evolution

Schema evolution refers to the ability to change the database schema over time without losing data. Django handles schema evolution through migrations.

Example Schema Changes:

- Adding a new field to a model
- Renaming a field
- Deleting a field or model

To make schema changes:

1. **Modify the Model**: Change your model definition.
2. **Create Migrations**: Run python manage.py makemigrations.
3. **Apply Migrations**: Run python manage.py migrate.