Gopalan College of Engineering and Management

Module-2 LECTURE NOTES

Module-2: Django Templates and Models

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.

2.1. Template System Basics:

The Django template system is a powerful tool for generating dynamic HTML content. At its core, it combines HTML markup with Django's own template language, which resembles Python syntax. This allows developers to inject dynamic data into their HTML templates without having to mix logic directly into the presentation layer.

Django's template system is designed to be both flexible and easy to use. It supports variables, tags, and filters. Variables are placeholders for dynamic data that are replaced with actual values when the template is rendered. Tags provide control flow and logic within the template, such as loops and conditionals. Filters modify the output of variables, allowing for formatting and transformation.

By separating the presentation layer from the application logic, Django promotes a clean and maintainable codebase. Templates can be reused across multiple views, making it easier to maintain consistency in the user interface.

2.2. Using Django Template System:

To use Django's template system, you first need to configure it in your Django project settings. This involves specifying the location of your template files and any additional settings related to template rendering.

Once configured, you can create HTML templates within your project's templates directory. These templates can contain both static HTML content and dynamic elements defined using Django template syntax. To render a template, you use the render() function provided by Django's django.shortcuts module. This function takes a request object, a template name, and optionally, a context dictionary containing data to be passed to the template.

For example:

```python
from django.shortcuts import render

def my_view(request):
    context = {
        'name': 'John',
        'age': 30,
    }
    return render(request, 'my_template.html', context)
```

In the above example, my_template.html is the name of the template file located in the templates directory, and context is a dictionary containing the data to be passed to the template.

2.3. Basic Template Tags and Filters:

Django provides a wide range of built-in template tags and filters that allow you to perform various operations within your templates.

Template tags, enclosed within {% %}, are used to control the flow of logic in a template. For example, you can use {% if %} tags to conditionally render content based on certain conditions, or {% for %} tags to iterate over a list of items.

Filters, indicated by the pipe character |, are used to modify the output of template variables. For example, you can use the date filter to format a date object, or the uppercase filter to convert a string to uppercase.

Combining tags and filters allows you to create dynamic and expressive templates that respond to user input and display data in a readable format.

2.4. MVT (Model-View-Template) Development Pattern:

The Model-View-Template (MVT) pattern is a design pattern commonly used in web development, particularly with Django. It is similar to the Model-View-Controller (MVC) pattern, but with a slight variation in terminology.

In the MVT pattern:

Models represent the data structure of the application. They define the fields and behavior of the data stored in the database.

Views handle the business logic of the application. They interact with models to retrieve and manipulate data, and then pass this data to templates for rendering.

Templates are responsible for rendering the user interface. They contain HTML markup with embedded Django template tags and filters to display dynamic content.

This pattern promotes a clear separation of concerns, making it easier to maintain and extend the codebase. Each component of the MVT pattern has a specific role to play, resulting in cleaner, more modular code.

2.5. Template Loading:

Django automatically loads templates from the templates directories of installed apps. When you call the render() function to render a template, Django searches through these directories to find the specified template file.

You can customize template loading by configuring the DIRS option in the TEMPLATES setting of your Django project settings. This allows you to specify additional directories where Django should look for templates.

Additionally, you can use template loaders to load templates from alternative sources, such as databases or remote storage. Django provides built-in template loaders for loading templates from files, app directories, and other sources.

By configuring template loading, you can organize your template files in a way that makes sense for your project structure, and easily reuse templates across different parts of your application.

2.6. Template Inheritance:

Template inheritance is a powerful feature of Django's template system that allows you to create a base template with common elements, and then extend it in child templates to override specific sections.

At its core, template inheritance works by defining a base template that contains one or more {% block %} tags. These block tags define sections of the template that can be overridden by child templates.

Child templates use the {% extends %} tag to specify which base template to inherit from, and then override specific blocks as needed using the {% block %} tag.

This approach promotes code reuse and maintainability by allowing you to define common layout elements in a single location, while still providing flexibility to customize the appearance of individual pages.

By using template inheritance effectively, you can create clean, modular templates that are easy to understand and maintain, even as your project grows in complexity.

2.7. Configuring Databases:

Django supports a wide range of database backends, including SQLite, PostgreSQL, MySQL, and Oracle. To configure the database for your Django project, you need to specify the connection settings in the DATABASES setting of your Django project settings.

The DATABASES setting is a dictionary where each key represents a database connection alias (e.g., 'default'), and the corresponding value is a dictionary containing the connection settings for that database backend.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydatabase',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

In the above example, we are configuring a PostgreSQL database named mydatabase with the username myuser and password mypassword. The database is hosted on localhost with port 5432.

By configuring the DATABASES setting, you can connect your Django project to the desired database backend and start working with data.

2.8. Defining and Implementing Models:

Models in Django are Python classes that define the structure and behavior of database tables. Each model class corresponds to a table in the database, and each attribute of the class represents a field in that table.

To define a model in Django, you create a Python class that inherits from django.db.models.Model and define attributes on the class to represent fields in the database. Each field is instantiated with a specific field type, such as CharField, IntegerField, DateTimeField, etc., which determines the type of data that can be stored in that field.

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    birth_date = models.DateField()
```

In the above example, we define a Person model with three fields: name, age, and birth_date, each with a corresponding field type.

Once you have defined your models, you can use Django's migration system to create the corresponding database tables based on those models.

2.9. Basic Data Access:

Django provides an Object-Relational Mapping (ORM) layer that abstracts away the details of database interaction and allows you to work with data using Python objects.

To perform basic data access in Django, you typically use model managers and querysets. Model managers are objects associated with model classes that provide methods for creating, retrieving, updating, and deleting objects. Querysets are lazy collections of database records that can be filtered, sliced, and manipulated in various ways.

For example:

```
# Creating objects
person = Person.objects.create(name='John', age=30, birth_date='1990-01-01')

# Retrieving objects
people_over_25 = Person.objects.filter(age__gt=25)

# Updating objects
person.age = 31
person.save()

# Deleting objects
person.delete()
```

In the above example, we create a Person object, retrieve people over the age of 25, update a person's age, and then delete a person from the database.

By using model managers and querysets, you can perform CRUD (Create, Read, Update, Delete) operations on your data without having to write raw SQL queries.

2.10. Adding Model String Representations:

In Django, you can define a string representation for your model instances by implementing the __str__() method on your model classes. This method returns a human-readable string that represents the object.

By default, if you don't define a __str__() method on your model class, Django will use a generic representation that includes the name of the model class and the primary key of the object. However, it's often more useful to provide a custom string representation that includes more meaningful information about the object.

For example:

```
class Person(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

    def __str__(self):
        return f'{self.name} ({self.age})'
```

In the above example, we define a __str__() method for the Person model that returns the person's name and age in a readable format.

By providing a custom string representation, you can improve the readability of your code and make it easier to work with model instances in the Django admin interface and other contexts where objects need to be displayed as strings.

2.11. Inserting/Updating Data:

In Django, you can insert or update data in the database using model instances and the save() method. When you call save() on a model instance, Django automatically generates the necessary SQL statements to insert or update the corresponding record in the database.

For example:

```
# Inserting data
person = Person(name='John', age=30)
person.save()

# Updating data
person.age = 31
person.save()
```

In the above example, we create a Person object with a name and age, and then call save() to insert the record into the database. Later, we update the person's age and call save() again to update the record in the database.

Django handles the details of database interaction for you, making it easy to work with data in your Django applications.

2.12. Selecting and Deleting Objects:

In Django, you can select objects from the database using querysets, and delete objects using either querysets or model instances.

To select objects, you use the filter() method on the model's manager, passing it a set of lookup parameters to filter the results. This returns a queryset containing the matching objects.

For example:

```
# Selecting objects
people_over_25 = Person.objects.filter(age__gt=25)

# Deleting objects
people_over_25.delete()
```

In the above example, we select all Person objects with an age greater than 25 using the filter() method, and then delete those objects using the delete() method on the resulting queryset.

Alternatively, you can delete individual objects by calling the delete() method on a model instance.

By using querysets and model instances, you can select and delete objects from the database in a simple and efficient manner.

2.13. Schema Evolution:

As your Django project evolves, you may need to make changes to your database schema to accommodate new requirements or fix issues in your data model. Django provides a migration system to manage these changes in a systematic way.

Migrations are Python files that define the operations necessary to migrate the database schema from one state to another. These operations can include creating new tables, adding or removing columns, and altering existing constraints.

To create a new migration, you use the makemigrations management command, which examines the current state of your models and generates the necessary migration files. You can then apply these migrations to the database using the migrate command.

For example:

```
$ python manage.py makemigrations myapp
$ python manage.py migrate
```

In the above example, we create migrations for the myapp app and then apply those migrations to the database.

By using migrations, you can manage changes to your database schema in a version-controlled and repeatable manner, ensuring consistency across development, staging, and production environments.

These are the detailed explanations for each of the topics you provided. Let me know if you need further clarification on any of them!