Competency Based Learning Material (CBLM)

Web Application Development with Python

Level-4

Module: Using Django Model

Code: CBLM- OU-ICT-WADP-03-L4-V1

National Skills Development Authority
Chief Advisor's Office
Government of the People's Republic of Bangladesh

Copyright

National Skills Development Authority
Chief Advisor's Office
Level10-11, Biniyog Bhaban,
E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.
Emailec@nsda.gov.bd
Websitewww.nsda.gov.bd.
National Skills Portalhttp:\\skillsportal.gov.bd

This Competency Based Learning Materials (CBLM) on "**Using Django Model**" under the **Web Application Development with Python , Level-4** qualification is developed based on the national competency standard approved by National Skills Development Authority (NSDA)

This document is to be used as a key reference point by the competency-based learning materials developers, teachers/trainers/assessors as a base on which to build instructional activities.

National Skills Development Authority (NSDA) is the owner of this document. Other interested parties must obtain written permission from NSDA for reproduction of information in any manner, in whole or in part, of this Competency Standard, in English or other language.

It serves as the document for providing training consistent with the requirements of industry in order to meet the qualification of individuals who graduated through the established standard via competency-based assessment for a relevant job.

This document has been developed by NSDA in association with industry representatives, academia, related specialist, trainer, and related employee. Public and private institutions may use the information contained in this CBLM for activities benefitting Bangladesh.

Approved by the Authority………. meeting held on ....

How to use this Competency Based Learning Material (CBLM)

The module contains training materials and activities for you to complete. These activities may be completed as part of structured classroom activities or you may be required you to work at your own pace. These activities will ask you to complete associated learning and practice activities in order to gain knowledge and skills you need to achieve the learning outcomes.

1. Review the **Learning Activity** page to understand the sequence of learning activities you will undergo. This page will serve as your road map towards the achievement of competence.

2. Read the **Information sheet s.** This will give you an understanding of the jobs or tasks you are going to learn how to do. Once you have finished reading the **Information sheet s** complete the questions in the **Self-Check.**

3. **Self-**Checks are found after each **Information sheet** . **Self-Checks** are designed to help you know how you are progressing. If you are unable to answer the questions in the **Self-Check** you will need to re-read the relevant **Information sheet** . Once you have completed all the questions check your answers by reading the relevant **Answer Keys** found at the end of this module.

4. Next move on to the **Job Sheets. Job Sheets** provide detailed information about *how to do the job* you are being trained in. Some **Job Sheets** will also have a series of **Activity Sheets**. These sheets have been designed to introduce you to the job step by step. This is where you will apply the new knowledge you gained by reading the Information sheet s. This is your opportunity to practise the job. You may need to practise the job or activity several times before you become competent.

5. Specification **sheets**, specifying the details of the job to be performed will be provided where appropriate.

6. A review of competency is provided on the last page to help remind if all the required assessment criteria have been met. This record is for your own information and guidance and is not an official record of competency

When working though this Module always be aware of your safety and the safety of others in the training room. Should you require assistance or clarification please consult your trainer or facilitator.

When you have satisfactorily completed all the Jobs and/or Activities outlined in this module, an assessment event will be scheduled to assess if you have achieved competency in the specified learning outcomes. You will then be ready to move onto the next Unit of Competency or Module

Table of Contents

Copyright ii

How to use this Competency Based Learning Material (CBLM).................... vi

Module Content 1

Learning Outcome 1: Create model 2

Learning Experience 1: Create model 3
Information sheet 1: Create model.................... 4
Self-Check Sheet 1: Create model 13
Answer Key 1 : Create model.................... 15
Job Sheet-1: Create a model with the possibility of being substituted with a different implementation in the future.................... 16
Specification Sheet-1: Create a model with the possibility of being substituted with a different implementation in the future.................... 17

Learning Outcome 2: Implement Django model Queries and Managers 18

Learning Experience 2: Implement Django model Queries and Managers 19
Information sheet 2: Implement Django model Queries and Managers 20
Self-Check Sheet 2 : Implement Django model Queries and Managers.................... 38
Answer Key 2 : Implement Django model Queries and Managers 40
Job Sheet-2: Perform Create, Read, Update, and Delete (CRUD) operations on individual records in a Django model. 41
Specification Sheet-2: Perform Create, Read, Update, and Delete (CRUD) operations on individual records in a Django model.................... 42

Learning Outcome 3 : Create class-based view 43

Learning Experience 3 : Create class-based view.................... 44
Information sheet 3: Create class-based view.................... 45
Self-Check Sheet 3: Create class-based view 81
Answer Key 3: Create class-based view.................... 82
Job Sheet-3: Create a web view using a Django class-based view (CreateView) to handle form submission and creation of new model records. 83

Reference 85

Review of Competency.................... 86

Development of CBLM 87

Module Content

Unit of Competency	Use Django Mode
Unit Code	OU-ICT-WADP-01-L4-V1
Module Title	**Using Django Mode**
Module Descriptor	This module covers the knowledge, skills and attitudes required to use Django Model It includes the task of creating model, implementing Django model Queries and Managers and creating class-based view
Nominal Hours	60 Hours
Lerning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs 1. Create model 2. Implement Django model Queries and Managers 3. Create class-based view

Assessment Criteria

1. Model class is defined with proper model data type and relationships
2. Model migration is performed
3. CRUD single records in Django Models are performed using shell and Admin
4. CRUD multiple records in Django Models are performed using shell and admin
5. CRUD relationship records across Django models are performed
6. Model queries are performed
7. Custom and multiple model managers are created
8. Model records with class-based view are created
9. CRUD with class-based view is performed
10. Mixin is applied with class-based view

Learning Outcome 1: Create model

Assessment Criteria	1. Model class is defined with proper model data type and relationships 2. Model migration is performed
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Model data type a. Binary b. Boolean c. Date/time d. Number e. Text 2. Model class 3. Model migration
Activities/job/Task	1. Create a model with the possibility of being substituted with a different implementation in the future.
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 1: Create model

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Create model'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Define Tasks of the Project"	2. Read Information sheet 1: Create model 3. Answer Self-check 1: Create model 4. Check your answer with Answer key 1: Create model
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-1: Create a model with the possibility of being substituted with a different implementation in the future.

Information sheet 1: Create model

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

1.1. Model class is defined with proper model data type and relationships
1.2. Model migration is performed

1.1. Model class with proper model data type and relationships

In Django, **model classes** are used to define the structure of your database tables and the relationships between them. The fields in a model represent the columns of the table, and each instance of a model represents a row in the database. Django uses an ORM (Object-Relational Mapping) to automatically translate between Python objects and database records.
Here's a guide on how to define model classes with proper field types and relationships in Django:

A. Basic Model Field Types

Django provides a wide range of field types to define the data you want to store in your models. Here are some common field types and their uses:

- CharField: A string field for short text, e.g., names, titles.
- TextField: A field for long text, e.g., descriptions.
- IntegerField: An integer field for storing whole numbers.
- FloatField: A field for storing floating-point numbers.
- DecimalField: A field for storing fixed-precision decimal numbers.
- DateField: A field for storing dates.
- DateTimeField: A field for storing date and time.
- BooleanField: A field for storing True/False values.
- EmailField: A field for storing email addresses.
- URLField: A field for storing URLs.
- FileField: A field for storing files.
- ImageField: A field for storing image files.
- ForeignKey: A field for creating a many-to-one relationship with another model.
- ManyToManyField: A field for creating a many-to-many relationship with another model.
- OneToOneField: A field for creating a one-to-one relationship with another model.

B. Basic Model Example

Below is an example of a basic model with different types of fields:

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=200)  # A short text field (e.g., name of the product)
    description = models.TextField()  # A long text field (e.g., detailed description)
    price = models.DecimalField(max_digits=10, decimal_places=2)  # A decimal field for price with 2 decimal places
    stock_quantity = models.IntegerField()  # An integer field for the stock quantity
    available = models.BooleanField(default=True)  # A boolean field to track availability
    created_at = models.DateTimeField(auto_now_add=True)  # DateTime field to store when the record was created
    updated_at = models.DateTimeField(auto_now=True)  # DateTime field to store when the record was last updated

    def __str__(self):
        return self.name
```

In this example:

- name: A CharField to store a short string (product name).
- description: A TextField for a longer description.
- price: A DecimalField to store the price with a precision of 10 digits and 2 decimal places.
- stock_quantity: An IntegerField to store the stock amount.
- available: A BooleanField to store availability (True or False).
- created_at and updated_at: DateTimeField to store creation and update timestamps.

C. Defining Relationships Between Models

Django allows you to define relationships between models using **ForeignKey**, **ManyToManyField**, and **OneToOneField**.

One-to-Many Relationship (ForeignKey)

In a one-to-many relationship, one record in one model can be associated with multiple records in another model. For example, a **Category** model can have multiple **Product** models.

```
class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Product(models.Model):
    name = models.CharField(max_length=200)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)  # One-to-many relationship
    price = models.DecimalField(max_digits=10, decimal_places=2)

    def __str__(self):
        return self.name
```

In this example:

- Product has a ForeignKey to Category. This creates a **one-to-many** relationship where one category can have multiple products.
- on_delete=models.CASCADE specifies that when a Category is deleted, all related Product entries will also be deleted.

Many-to-Many Relationship (ManyToManyField)

In a many-to-many relationship, multiple records in one model can be associated with multiple records in another model. For example, a **Product** can have multiple **Tags**, and a **Tag** can be associated with multiple products.

```
class Tag(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Product(models.Model):
    name = models.CharField(max_length=200)
    tags = models.ManyToManyField(Tag)  # Many-to-many relationship

    def __str__(self):
        return self.name
```

In this example:

- Product has a ManyToManyField to Tag. This creates a **many-to-many** relationship, meaning that a product can have multiple tags, and a tag can be associated with multiple products.

One-to-One Relationship (OneToOneField)

In a one-to-one relationship, each record in one model is associated with exactly one record in another model. For example, a **Profile** model could be associated with a single **User** model.

```python
from django.contrib.auth.models import User

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)  # One-to-one relationship
    birth_date = models.DateField()
    phone_number = models.CharField(max_length=20)

    def __str__(self):
        return f"Profile of {self.user.username}"
```

In this example:

- Profile has a OneToOneField to User, creating a **one-to-one** relationship where each user has exactly one profile.

D. Using Custom Managers for Model Queries

Sometimes, you may need to define custom query methods for your models. You can achieve this by creating a custom manager.

```python
class ProductManager(models.Manager):
    def in_stock(self):
        return self.filter(stock_quantity__gt=0)  # Custom query to get products that are in stock

class Product(models.Model):
    name = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock_quantity = models.IntegerField()

    objects = ProductManager()  # Using the custom manager

    def __str__(self):
        return self.name
```

In this example, ProductManager is a custom manager that defines the in_stock() method, which filters products that have a stock quantity greater than zero.

E. Model Meta Options

Django allows you to define additional metadata options inside the Meta class of a model, such as ordering, verbose names, and table names.

```
class Product(models.Model):
    name = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=10, decimal_places=2)

    class Meta:
        ordering = ['name']  # Default ordering by 'name'
        verbose_name = 'Product'  # Singular form name
        verbose_name_plural = 'Products'  # Plural form name

    def __str__(self):
        return self.name
```

In this example:

- ordering: Specifies the default ordering of objects returned by the database.
- verbose_name and verbose_name_plural: Provide more human-readable names for the model.

F. Model Inheritance

Django supports model inheritance, allowing you to create a hierarchy of models.

Abstract Base Classes

You can create an abstract base class that contains common fields for other models to inherit from.

```
class CommonInfo(models.Model):
    name = models.CharField(max_length=200)
    description = models.TextField()

    class Meta:
        abstract = True  # Make this model abstract

class Product(CommonInfo):
    price = models.DecimalField(max_digits=10, decimal_places=2)

class Category(CommonInfo):
    pass
```

In this example, both Product and Category inherit from the abstract CommonInfo model, which defines shared fields like name and description.

1.2. Model migration

In Django, migrations are a way to propagate changes you make to your models (such as creating a new model, altering an existing model, or deleting models) into the database schema. Django provides a powerful migration system that allows you to manage database changes easily. Migrations track changes to models and create the necessary database schema updates.

What is a Migration?

- A migration is a file that Django creates whenever you change your models (for example, by adding a field, changing a field type, or deleting a model).
- Migrations ensure that your database schema is in sync with your Django models.

Steps for Model Migration

There are three basic steps involved in Django migrations:

1. **Create a Migration File**
2. **Apply the Migration**
3. **Verify the Migration**

1. Create a Migration File

After making changes to your Django model (e.g., adding a new field or modifying an existing field), you need to create a migration file. This migration file will describe the changes to your database schema.

Example:

Suppose you have the following model for a blog application:

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

class Blog(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

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

Now, let's say we want to add a new field called author_name to the Blog model.

```
# models.py (Updated)
class Blog(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author_name = models.CharField(max_length=100)  # New field added
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

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

To create a migration file for this change, you run the following command:

```
python manage.py makemigrations
```

Output:

```
Migrations for 'blog':
  blog/migrations/0002_auto_20250214_1532.py
    - Add field author_name to blog
```

.

This command generates a migration file inside the blog/migrations/ directory (e.g., 0002_auto_20250214_1532.py). This file contains the necessary instructions to update the database schema.

2. Apply the Migration

Once the migration file has been created, you need to apply it to update your database schema. You can do this by running the following command:

```
python manage.py migrate
```

Output:

```
Operations to perform:
  Apply all migrations: blog
Running migrations:
  Applying blog.0002_auto_20250214_1532... OK
```

This command applies the migration, and Django updates the database schema based on the changes in the migration file.

3. Verify the Migration

After applying the migration, you can verify that the changes have been applied by checking the database. For example, you can access the Django shell or use a database management tool to check if the author_name field has been added to the Blog model.

You can also check the status of migrations with this command:

```
python manage.py showmigrations
```

Output:

```
blog
 [X] 0001_initial
 [X] 0002_auto_20250214_1532
```

This output indicates that both migrations (0001_initial and 0002_auto_20250214_1532) have been applied.

Additional Migration Commands

1. **Rolling Back a Migration (Undoing Changes)**: You can roll back a migration to a previous state by using the migrate command with the migration name:
2. python manage.py migrate blog 0001

   This command will roll back the migration to the state defined in migration 0001.

3. **Making Migrations for a Specific App**: If you want to create migrations only for a specific app, use the following command:
4. python manage.py makemigrations blog
5. **Squashing Migrations**: If you have many migrations and want to combine them into a single migration to clean up your migration history, you can use the squashmigrations command:
6. python manage.py squashmigrations blog 0001 0005

   This will combine migrations from 0001 to 0005 into a single migration file.

Example Workflow: Full Model Migration Example

1. Initial Model Creation:

```
# models.py
class Blog(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

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

1. Run makemigrations:
2. python manage.py makemigrations
3. Apply the migration:
4. python manage.py migrate

2. Modify the Model (Add a Field):

```
# models.py (Updated)
class Blog(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author_name = models.CharField(max_length=100)  # New field added
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

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

1. Run makemigrations:
2. python manage.py makemigrations
3. Apply the migration:
4. python manage.py migrate

Self-Check Sheet 1: Create model

Q1. What is the primary purpose of defining models in a Django web application?

Answer:

- A. To define URL patterns (for routing requests)
- B. To define the structure of stored data, including field types and relationships
- C. To implement custom login views
- D. To design the website's layout

Q2. Which of the following is NOT a common type of field used in Django models?

Answer:

- A. CharField (for storing text)
- B. DateField (for storing dates)
- **C. CustomField (there's no built-in field type like this)**
- D. ForeignKey (for one-to-many relationships)

Q3. What does the `related_name` argument do when used with a `ForeignKey` field?

Answer:

- A. It specifies the field on the related model that the foreign key refers to.
- B. It allows you to set a default value for the foreign key field.
- **C. It specifies the name used to access the related model's objects from the current model.**
- D. It determines whether the foreign key field can be null.

Q4. What is the difference between a `ForeignKey` and a `ManyToManyField` in Django models?

Answer:

- A. There is no difference, they both achieve the same relationship.
- B. A `ForeignKey` is used for one-to-one relationships, while a `ManyToManyField` is used for many-to-many relationships.
- C. A `ForeignKey` allows a model to have zero or one related object, while a `ManyToManyField` always requires at least one related object.
- D. A `ForeignKey` represents a one-to-many relationship, while a `ManyToManyField` represents a many-to-many relationship.

Q5. What is the advantage of using separate models for different data entities in your Django application?

Answer:

- A. It makes the code more complex and harder to maintain.
- B. It allows for easier data manipulation and retrieval through queries.
- C. It improves the visual design of the website.
- D. It reduces the number of database tables required.

Answer Key 1 : Create model

Q1. What is the primary purpose of defining models in a Django web application?

Answer: B. To define the structure of stored data, including field types and relationships

Q2. Which of the following is NOT a common type of field used in Django models?

Answer: C. CustomField (there's no built-in field type like this)

Q3. What does the `related_name` argument do when used with a `ForeignKey` field?

Answer: C. It specifies the name used to access the related model's objects from the current model.

Q4. What is the difference between a `ForeignKey` and a `ManyToManyField` in Django models?

Answer: D. A `ForeignKey` represents a one-to-many relationship, while a `ManyToManyField` represents a many-to-many relationship.

Q5. What is the advantage of using separate models for different data entities in your Django application?

Answer: B. It allows for easier data manipulation and retrieval through queries.

Job Sheet-1: Create a model with the possibility of being substituted with a different implementation in the future.

UoC Cover

OU-ICT-WADP-03- L4-V1: Use Django Model

Working Procedure / Steps

1. Create a new Python file in your Django app's `models.py` directory.
2. Define your model class inheriting from `models.Model`.
3. Include necessary fields using appropriate Django field types (e.g., `CharField`, `IntegerField`).
4. Consider using abstract base classes for common functionalities across mulTip:le models.
5. **Use `settings.AUTH_USER_MODEL` as a Reference:**
    a. If your model represents a user, consider using Django's built-in mechanism for swapping the user model.
    b. In your project's `settings.py` file, set the `AUTH_USER_MODEL` setting to the path of your custom user model class (e.g., `'your_app.CustomUser'`).
6. **Abstract Base Class (Optional):**
    a. Define an abstract base class inheriting from `models.Model`.
    b. Move common fields and functionalities to this base class.
    c. Inherit from the abstract base class in your specific model implementation.
7. Run `python manage.py makemigrations your_app_name` to generate migration files for your model changes.
8. This creates migration files that track the database schema changes associated with your model.

Specification Sheet-1: Create a model with the possibility of being substituted with a different implementation in the future.

Technical Requirements

- **Operating System:** Windows 10 or 11 (or equivalent Linux/macOS)
- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later
- **Database:** PostgreSQL or MySQL (or other supported database)
- **Web Server:** Apache or Nginx (or other compatible web server)
- **Text Editor/IDE:** Visual Studio Code, PyCharm, or Sublime Text

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Internet Connection:** A reliable internet connection for downloading and installing software, accessing online resources, and deploying the application.
- **Version Control System:** Git for managing project code and collaboration.

Additional Requirements

- **Development Environment:** A virtual environment is recommended to isolate project dependencies.
- **Deployment Platform:** A hosting platform (e.g., Heroku, AWS, DigitalOcean) for deploying the application.
- **Database Management Tools:** Tools like pgAdmin or phpMyAdmin for managing databases.
- **Browser:** A modern web browser (e.g., Chrome, Firefox, Edge) for testing and development.
- **Text Editor:** A code editor or IDE for writing and editing Python code.

Learning Outcome 2: Implement Django model Queries and Managers

Assessment Criteria	1. CRUD single records in Django Models are performed using shell and Admin 2. CRUD mulTip:le records in Django Models are performed using shell and admin 3. CRUD relationship records across Django models are performed 4. Model queries are performed 5. Custom and mulTip:le model managers are created
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. CRUD single records in Django Models 2. CRUD mulTip:le records in Django Models 3. CRUD relationship records across Django models 4. Model queries a. By SQL Key word b. Raw method c. Python DB API 5. Custom and mulTip:le model managers
Activities/job/Task	1. Perform Create, Read, Update, and Delete (CRUD) operations on individual records in a Django model.
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 2: Implement Django model Queries and Managers

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Implement Django model Queries and Managers'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Define Tasks of the Project"	2. Read Information sheet 2: Implement Django model Queries and Managers 3. Answer Self-check 2: Implement Django model Queries and Managers 4. Check your answer with Answer key 2: Implement Django model Queries and Managers
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-2: Perform Create, Read, Update, and Delete (CRUD) operations on individual records in a Django model.

Information sheet 2: Implement Django model Queries and Managers

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

2.1. CRUD single records in Django Models are performed using shell and Admin
2.2. CRUD mulTip:le records in Django Models are performed using shell and admin
2.3. CRUD relationship records across Django models are performed
2.4. Model queries
2.5. Custom and mulTip:le model managers

2.1. CRUD single records in Django Models using shell and Admin

In Django, **CRUD operations** (Create, Read, Update, Delete) are essential tasks for interacting with your models. Below is a guide on how to perform CRUD operations on a single record in Django, both using the **Django shell** and the **Django Admin** interface.

A. CRUD Operations in Django using the Django Shell

a. Start the Django Shell

To begin interacting with your Django models in the shell, use the following command:

```
python manage.py shell
```

Once inside the shell, you can import your models and perform CRUD operations.

b. Create (C)

To create a new record, simply instantiate the model and call the save() method.

Example:

```python
from myapp.models import Product  # Import the Product model

# Create a new product record
product = Product(name="Laptop", description="A powerful laptop", price=999.99, stock_quantity=10)
product.save()  # Save the record to the database
```

c. Read (R)

To read or retrieve a record from the database, you can use methods like objects.get(), objects.all(), or objects.filter().

Example 1: **Get a single record by primary key (ID)**

```python
from myapp.models import Product  # Import the Product model

# Create a new product record
product = Product(name="Laptop", description="A powerful laptop", price=999.99, stock_quantity=10)
product.save()  # Save the record to the database
```

Example 2: **Get all records**

```python
# Retrieve a product by its primary key (ID)
product = Product.objects.get(id=1)
print(product.name)  # Output the product name
```

Example 3: **Filter records**

```python
# Retrieve products where stock_quantity is greater than 5
products = Product.objects.filter(stock_quantity__gt=5)
for product in products:
    print(product.name)
```

d. Update (U)

To update a record, first retrieve it, modify the fields, and then call save() again to save the changes.

Example:

```
# Retrieve a product by its primary key
product = Product.objects.get(id=1)

# Update the price and stock quantity
product.price = 899.99
product.stock_quantity = 5
product.save()  # Save the updated record to the database
```

e. Delete (D)

To delete a record, first retrieve it, and then call the delete() method on the instance.

Example:

```
# Retrieve the product by its primary key
product = Product.objects.get(id=1)

# Delete the product
product.delete()  # This will delete the record from the database
```

B. CRUD Operations in Django using the Admin Interface

Django comes with a built-in admin interface that allows you to perform CRUD operations easily on your models.

a. Set up the Admin Interface

1. **Register your model** in the admin.py file of your app.

For example, if you have a Product model:

```
# myapp/admin.py
from django.contrib import admin
from .models import Product

admin.site.register(Product)  # Register the Product model with the admin site
```

2. **Create a superuser** to access the admin interface:

```
python manage.py createsuperuser
```

Follow the prompts to create the superuser account (username, email, and password).

b. Access the Django Admin Interface

Start the development server:

python manage.py runserver

Now, go to the Django admin interface by navigating to http://127.0.0.1:8000/admin/ in your browser. Log in with the superuser credentials you just created.

c. Performing CRUD Operations in Admin

Once logged in to the Django Admin interface, you will see a list of registered models (e.g., Product).

1. **Create** a new record:
    - Click on the **Product** model in the admin dashboard.
    - Click **Add Product**.
    - Fill in the form with the required fields (e.g., name, description, price, stock quantity).
    - Click **Save** to create the new product.
2. **Read** records:
    - After clicking on the **Product** model, you'll see a list of all products in the database. You can filter, search, or paginate through the list.
3. **Update** a record:
    - Click on the product you want to edit from the list.
    - Modify the values in the form fields (e.g., update the price or stock quantity).
    - Click **Save** to apply the changes.
4. **Delete** a record:
    - Click on the product you want to delete from the list.
    - Scroll to the bottom of the page and click **Delete**.
    - Confirm the deletion by clicking **Yes, I'm sure**.

Example Admin Operations

1. **Creating a New Product**:
    - In the admin interface, you might add a new product like this:
        - **Name**: "Smartphone"
        - **Description**: "A modern smartphone"
        - **Price**: 499.99
        - **Stock Quantity**: 20
    - After saving, the product will be added to the database.
2. **Editing an Existing Product**:

- If you need to update the price of a product, click on the product from the admin list, change the **Price** field, and click **Save**.

3. **Deleting a Product**:
   - If you want to delete a product, click on it, scroll to the bottom, click **Delete**, and confirm by clicking **Yes, I'm sure**.

Customizing the Admin Interface

Django's admin interface can be customized to make it more user-friendly and functional for your needs. Here's an example of how you can customize the Product model in the admin interface:

```
# myapp/admin.py
from django.contrib import admin
from .models import Product

class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price', 'stock_quantity')  # Display these fields in the list view
    search_fields = ('name',)  # Enable search functionality for the 'name' field
    list_filter = ('available',)  # Filter by availability (True/False)

admin.site.register(Product, ProductAdmin)  # Register the customized ProductAdmin
```

This customization will:

- Display the name, price, and stock_quantity in the list view.
- Allow searching by the product name.
- Provide a filter for the available field (whether the product is in stock or not).

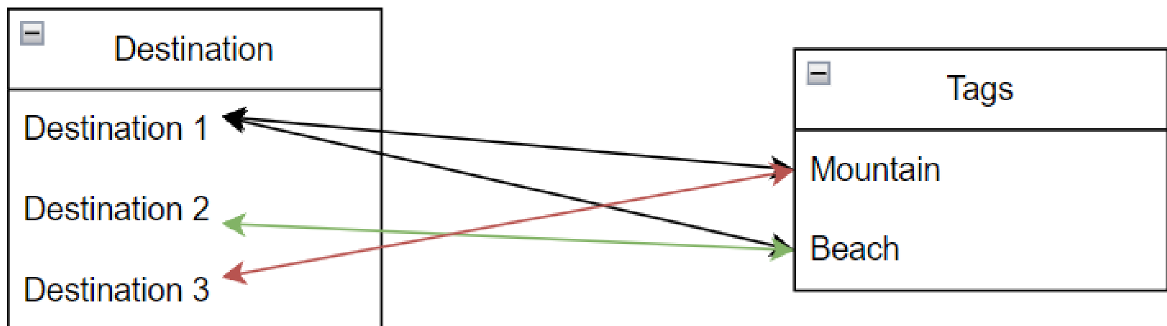
2.2. CRUD mulTip:le records in Django Models using shell and admin

In this section you'll learn how to work with mulTip:le records in Django models. Although the process is just as easy as working with single model records, working with mulTip:le records can require mulTip:le database calls, as well as caching techniques and bulk operations, all of which need to be taken into account to minimize execution times.

```
chamcham@anonymous: ~/todoproject/todoproject
>>> incomplete_tasks = Task.objects.filter(completed=False)
>>> incomplete_tasks.update(completed=True)
2
>>> all_tasks = Task.objects.all()
>>>
>>> for task in all_tasks:
...     print(f'Title: {task.title}, Completed: {task.completed}')
...
Title: Complete Assignment, Completed: True
Title: Another Task, Completed: True
>>>
```

2.3. CRUD relationship records across Django models

CRUD stands for Create, Read, Update, and Delete. These are the basic operations you can perform on data in a database. In Django, these operations can be performed on models, which represent the database tables. When working with related models (models that have foreign keys or many-to-many relationships), you can perform CRUD operations across these relationships as well.



Example: Using Related Models in Django

Let's assume we have two models: Author and Book. An author can have multiple books, so we'll define a **one-to-many relationship** using a foreign key.

```python
# models.py
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    bio = models.TextField()

    def __str__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    published_date = models.DateField()

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

In this example:

- The Author model has a name and bio field.
- The Book model has a title, author (a foreign key that links to the Author model), and a published_date.

A. Create (CRUD) Records in Related Models

To create records, you will create an instance of the Author and then associate books with that author.

 # Creating an Author and Book record

author = Author.objects.create(name="J.K. Rowling", bio="Famous author of the Harry Potter series.")

book1 = Book.objects.create(title="Harry Potter and the Sorcerer's Stone", author=author, published_date="1997-06-26")

book2 = Book.objects.create(title="Harry Potter and the Chamber of Secrets", author=author, published_date="1998-07-02")

Here:

- We first create an Author record.
- Then, we create two Book records associated with the Author.

B. Read (CRUD) Relationship Records

To retrieve related records, Django provides powerful querying capabilities. You can access related records using the reverse relationship.

```
# Retrieve all books by an author
author = Author.objects.get(name="J.K. Rowling")
books_by_author = author.book_set.all()  # Reverse relationship using book_set
for book in books_by_author:
    print(book.title)
```

This code retrieves all the books written by "J.K. Rowling" and prints the titles.

C. Update (CRUD) Relationship Records

Updating related records can be done by modifying the attributes of related objects.

```
# Update a Book record
book = Book.objects.get(title="Harry Potter and the Sorcerer's Stone")
book.title = "Harry Potter and the Philosopher's Stone"  # Update the title
book.save()  # Save changes to the database
```

This code changes the title of the book "Harry Potter and the Sorcerer's Stone" to "Harry Potter and the Philosopher's Stone".

4. Delete (CRUD) Relationship Records

You can delete related records, and Django will automatically handle the relationships, depending on the on_delete behavior defined in the model.

```
# Delete a Book record
book_to_delete = Book.objects.get(title="Harry Potter and the Sorcerer's Stone")
book_to_delete.delete()
```

This will delete the book "Harry Potter and the Sorcerer's Stone" from the database.

2.4. Model Queries

Django provides a powerful and flexible query API to retrieve, update, and delete records. You can filter records based on field values, perform aggregation, and even perform joins between related models.

A. Basic Queries:

1. **Retrieve All Records:** To retrieve all records from a model, use the all() method:

```
authors = Author.objects.all()
for author in authors:
    print(author.name)
```

2. **Filtering Records:** To filter records based on certain conditions, use the filter() method.

```
# Get all books published after 1997
books = Book.objects.filter(published_date__year__gt=1997)
for book in books:
    print(book.title)
```

Here, published_date__year__gt=1997 is used to filter books that were published after 1997.

3. **Getting a Single Record:** To retrieve a single record, use the get() method. This will return the object or raise an exception if no matching object is found.

```
author = Author.objects.get(name="J.K. Rowling")
print(author.bio)
```

4. **Excluding Records:** Use the exclude() method to exclude records that match a certain condition.

```
# Get all books except those published in 1997
books = Book.objects.exclude(published_date__year=1997)
for book in books:
    print(book.title)
```

5. **Ordering Records:** Use order_by() to order the queryset by one or more fields.

```
# Get the first 2 books ordered by publication date
books = Book.objects.all()[:2]
for book in books:
    print(book.title)
```

6. **Limiting the Number of Records:** Use [:n] to limit the number of records returned.

```
# Get the first 2 books ordered by publication date
books = Book.objects.all()[:2]
for book in books:
    print(book.title)
```

7. **Aggregation:** Django provides aggregation methods such as Count, Sum, Max, Min, and Avg to perform calculations on querysets.

```
from django.db.models import Count
# Get the number of books each author has written
authors_with_books = Author.objects.annotate(num_books=Count('book'))
for author in authors_with_books:
    print(f'{author.name} has written {author.num_books} books.')
```

B. Related Model Queries

When working with related models, Django allows you to follow relationships to retrieve related records. Let's use our Author and Book example to demonstrate related queries.

Get all books by a particular author:

```
author = Author.objects.get(name="J.K. Rowling")
books_by_author = author.book_set.all()
for book in books_by_author:
    print(book.title)
```

Perform a join-like operation to get authors who have written more than one book:

```
authors_with_multiple_books =
Author.objects.annotate(num_books=Count('book')).filter(num_books__gt=1)

for author in authors_with_multiple_books:

   print(author.name)
```

This code uses the Count aggregation to count the number of books each author has written and filters those who have written more than one.

2.5. Custom and mulTip:le model managers

As you've learned throughout the examples presented in this and the previous chapter, a Django model's objects reference or default model manager, offers an extensive array of functionalities to execute database operations.

Under most circumstances, Django models don't require any modifications to their default model manager or objects reference. However, there can be circumstances where the need arises to customize a Django model's default model manager or inclusively create mulTip:le model managers.

Custom and mulTip:le model managers

One of the main reasons to create custom Django model managers is to add custom manager methods, to make the execution of recurring queries on a model easier.

For example, running queries such as Item.objects.filter(menu__name='Sandwiches') or Item.objects.filter(menu__name='Salads') is simple, but if you start writing these same queries over and over, the process can become tiresome and error prone. This is particularly true for raw SQL queries, which take more time to write and have a higher degree of complexity.

A custom manager method allows you to write a query once as part of a model, and later invoke the custom manager method -- just like other model manager methods (e.g. all(), filter(), exclude()) -- to trigger the query. Example illustrates a

custom model manager class with a custom method, including a model that uses it, in addition to various model manager calls.

Example: Django custom model manager with custom manager methods

```
from django.db import models

# Create custom model manager
class ItemMenuManager(models.Manager):
    def salad_items(self):
        return self.filter(menu__name='Salads')

    def sandwich_items(self):
        return self.filter(menu__name='Sandwiches')

# Option 1) Override default model manager
class Item(models.Model):
    menu = models.ForeignKey(Menu, on_delete=models.CASCADE)
    name = models.CharField(max_length=30)
    ...
    objects = ItemMenuManager()

# Queries on default custom model manager
Item.objects.all()
Item.objects.salad_items()
Item.objects.sandwich_items()

# Option 2) Create new model manager field and leave default model manager as is
    menu = models.ForeignKey(Menu, on_delete=models.CASCADE)
    name = models.CharField(max_length=30)
    ...
    objects = models.Manager()
    menumgr = ItemMenuManager()

# Queries on default and custom model managers
Item.objects.all()
Item.menumgr.salad_items()
Item.menumgr.sandwich_items()
# ERROR Item.objects.salad_items()
# 'Manager' object has no attribute 'salad_items'
# ERROR Item.objects.sandwich_items()
# 'Manager' object has no attribute 'sandwich_items'
Item.menumgr.all()
```

The first class is the ItemMenuManager that functions as a custom model manager. Notice how this class inherits its behavior from the models.Manager class, which is what gives it model manager behavior. Next,

the ItemMenuManager class declares two methods that return QuerySet results. Notice how the class methods reference self -- representing the model class instance -- and call standard model methods to trigger database queries.

It's worth mentioning custom model managers don't necessarily need to use native model queries or return QuerySet data structures, custom model managers can equally contain any logic (e.g. Python DB API calls) or return any data structure (e.g. Python tuples).

Once you have a custom model manager there are two options to assign it to a model class. The first option illustrated consists of overriding a model's default model manager objects and explicitly assigning it a custom model manager. Once this is done, you can use the same objects reference to call the custom model manager methods. In addition, notice that even when overriding the default model manager objects, a model continues to have access to the built-in model manager methods (e.g. all()) because the custom model inherits its behavior from the parent models.Manager class.

Next, Example illustrates the second option to integrate a custom model manager. This option consists of adding a new model field to reference a custom manager and leave the default manager objects as is. In this case, the custom model manager methods become accessible through the new field reference (e.g. Item.menumgr.salad_items()) and the objects reference continues to work with its default behavior.

Warning If you don't define a default model manager in a multi-manager model, Django choose the first manager declared in the model. This can have unexpected behaviors in model operations that can't explicitly chose model managers (e.g. dumpdata) unlike queries that can use dot-notation to choose a model manager.

Custom model managers and QuerySet classes with methods

Model managers are closely tied to methods that return QuerySet data structures. As you've seen, nearly all methods chained to the default model manager objects (e.g. all(), filter()) generate QuerySet data structures. When you create custom model managers, it's possible to override the default behavior for these QuerySet methods, as well as create your own custom QuerySet classes and methods.

One of the most important QuerySet methods in model managers is the get_queryset() method, used to define a model's initial QuerySet or what's returned by a model manager's all() method. In custom model managers, the get_queryset() method is particularly important because it let's you filter the initial QuerySet depending on the purpose of a model manager.

Example: Django custom model managers with custom get_queryset() method

```
class SanDiegoStoreManager(models.Manager):
    def get_queryset(self):
        return super(SanDiegoStoreManager, self).get_queryset().filter(city='San Diego')

class LosAngelesStoreManager(models.Manager):
    def get_queryset(self):
        return super(LosAngelesStoreManager, self).get_queryset().filter(city='Los Angeles')

class Store(models.Model):
    name = models.CharField(max_length=30)
    ...
    objects = models.Manager()
    sandiego = SanDiegoStoreManager()
    losangeles = LosAngelesStoreManager()

# Call default manager all() query, backed by get_queryset() method
Store.objects.all()

# Call sandiego manager all(), backed by get_queryset() method
Store.sandiego.all()

# Call losangeles manager all(), backed by get_queryset() method
Store.losangeles.all()
```

The first two classes represent custom model managers, however, notice that unlike the custom model manager, both
the SanDiegoStoreManager and LosAngelesStoreManager classes define
the get_queryset() method. In both cases, the get_queryset() method returns
a QuerySet generated by calling the parent model manager get_queryset() method (i.e. all()) -- via the super() method -- and applying an additional filter() to the parent depending on the purpose of the custom model manager (e.g. get stores by city).

Once the custom managers are defined, Example declares the custom model managers as separate fields in the Store model class. Finally, you can see calls made to each of the model managers using the all() method, which return the appropriate filtered results depending on the logic of the
backing get_queryset() method.

An alternative to mulTip:le custom model managers, is to create a single custom manager and rely on a custom QuerySet class and methods to execute the same logic.

Example: Django custom model manager with custom QuerySet class and methods

```
class StoreQuerySet(models.QuerySet):
    def sandiego(self):
        return self.filter(city='San Diego')
    def losangeles(self):
        return self.filter(city='Los Angeles')

class StoreManager(models.Manager):
    def get_queryset(self):
        return StoreQuerySet(self.model, using=self._db)
    def sandiego(self):
        return self.get_queryset().sandiego()
    def losangeles(self):
        return self.get_queryset().losangeles()

class Store(models.Model):
    name = models.CharField(max_length=30)
    ...
    objects = models.Manager()
    shops = StoreManager()
Store.shops.all()
Store.shops.sandiego()
Store.shops.losangeles()
```

The StoreQuerySet class is a custom QuerySet class that defines the sandiego() and losangeles() methods, both of which apply additional filters to its base QuerySet. Once you have a QuerySet class, it's necessary to associate it with a custom model manager. You can see the StoreManager class represents a custom model manager, which defines its get_queryset() method to set its initial data through the custom StoreQuerySet class.

Next, notice how the custom model manager StoreManager class defines the additional sandiego() and losangeles() methods, which are hooked up to call the methods by the same name in the custom StoreQuerySet class.

Finally, the custom model manager StoreManager is set up as the shops field in the Store model class, where you can observe how calls are made via the shops reference to trigger the query methods backed by the custom StoreQuerySet class.

As helpful as the technique, is to cut down on the amount model managers, if you look carefully at Example, there's still a fair amount of redundancy declaring similar named methods for both a custom model manager and a custom QuerySet class.

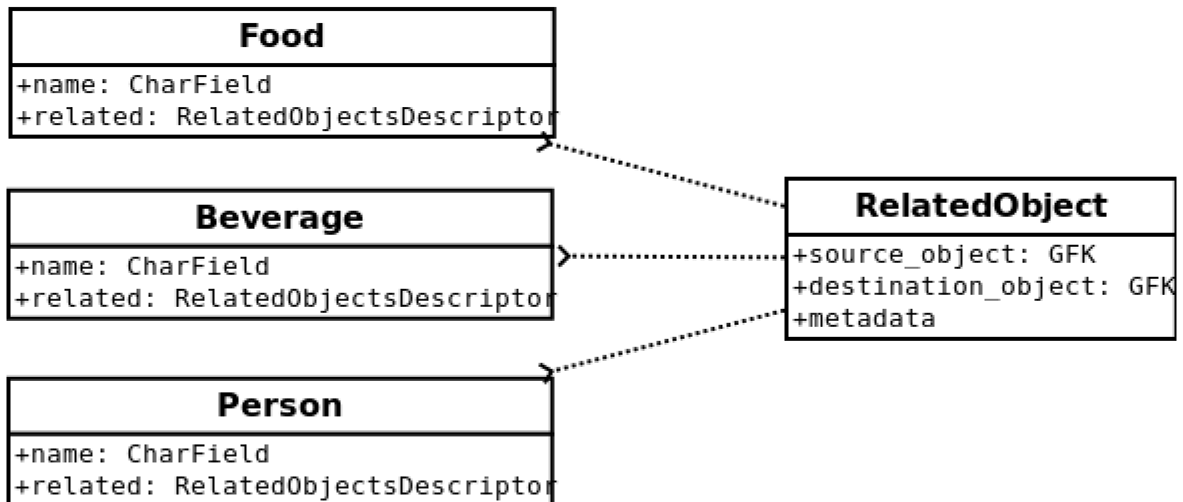
To cut down on redundant methods when using custom model managers and custom QuerySet classes, the latter type of class offers the as_manager() method to automatically convert a QuerySet class into a custom model manager.

Example: Django custom model manager with custom QuerySet converted to manager

```
class StoreQuerySet(models.QuerySet):
    def sandiego(self):
        return self.filter(city='San Diego')
    def losangeles(self):
        return self.filter(city='Los Angeles')

class Store(models.Model):
    name = models.CharField(max_length=30)
    ...
    objects = models.Manager()
    shops =  StoreQuerySet.as_manager()

Store.shops.all()
Store.shops.sandiego()
Store.shops.losangeles()
```

Custom reverse model managers for related models

Earlier in the CRUD relationship records section, you learned how models that have relationships between one another, use reverse queries or _set syntax to execute operations from the model that doesn't have the relationship definition.



These reverse operations are executed by a model manager dubbed RelatedManager which is a subclass of a model's default manager. This means all reverse queries or _set syntax calls are based on the objects model manager reference or whatever default model manager is used by a model.

If you configure a default model manager on a model, then all the reverse operations on a model will automatically use this same manager. However, it's possible to define a custom model manager exclusively for reverse operations, while ignoring the default model manager. This technique consists of explicitly declaring a model manager as part of the reverse operation.

Example: Django custom model manager for reverse query operations

```
from django.db import models

class Item(models.Model):
    ...
    objects = models.Manager()  # Default manager for direct queries
    reverseitems = CustomReverseManagerForItems() # Custom Manager for reverse queries

# Get Menu record named Breakfast
breakfast_menu = Menu.objects.get(name='Breakfast')

# Fetch all Item records in the Menu, using Item custom model manager for reverse queries
breakfast_menu.item_set(manager='reverseitems').all()
# Call on_sale_items() custom manager method in CustomReverseManagerForItems
breakfast_menu.item_set(manager='reverseitems').on_sale_items()
```

Example: first declares the Item model with its default objects model manager and a custom model manager assigned to the reverseitems field. Next, a query is made to get a Menu record, followed by various queries to get the Menu record's related Item records via the reverse _set syntax.

However, notice how the reverse query operation with _set syntax, uses the manager argument to indicate which model manager to use for reverse operations, in this case, the reverseitems model manager is used to execute the queries, instead of the default objects model manager.

Self-Check Sheet 2 : Implement Django model Queries and Managers

1. **In the context of Django models, what does "mulTiplicity" refer to?**

Answer:

- A. The total number of model fields
- B. The number of relationships between models (one-to-one, one-to-many, many-to-many)
- C. A unique identifier for each model instance
- D. The maximum length allowed for character fields

2. **When designing models for a library website, what should be considered before jumping into coding?**

Answer:

- A. The choice of website layout
- B. The relationships between different data objects (e.g., Information sheet and Author)
- C. The specific CSS styles needed
- D. The database engine to be used (consideration during project settings)

3. **What is the UML diagram in the passage primarily used for?**

Answer:

- A. Defining the website's user interface
- B. Visualizing the relationships between different models (boxes) and their multiplicities
- C. Specifying the exact field names and types
- D. Illustrating the website's layout and navigation

4. **What does the Django model `Information sheet Instance:status` exemplify?**

Answer:

- A. A recommended approach for defining selection list options
- B. A model field with pre-defined choices (since it's hardcoded)
- C. An Example: of a `ForeignKey` relationship
- D. An illustration of data that is not modeled due to limited options and no expectation of change

5. **What does the `verbose_name` argument within a model class do?**

Answer:

- A. Defines the maximum length allowed for the field
- B. Specifies the default value for a field
- C. Provides a human-readable name for the model (singular and plural)
- D. Determines if the field is allowed to be blank in forms

Answer Key 2 : Implement Django model Queries and Managers

1. **In the context of Django models, what does "mulTiplicity" refer to?**
Answer:
    - B. The number of relationships between models (one-to-one, one-to-many, many-to-many)

2. **When designing models for a library website, what should be considered before jumping into coding?**
Answer:
    - B. The relationships between different data objects (e.g., Information sheet and Author)

3. **What is the UML diagram in the passage primarily used for?**
Answer:
    - B. Visualizing the relationships between different models (boxes) and their multiplicities

4. **What does the Django model `Information sheet Instance:status` exemplify?**
Answer:
    - D. An illustration of data that is not modeled due to limited options and no expectation of change

5. **What does the `verbose_name` argument within a model class do?**
Answer:
    - C. Provides a human-readable name for the model (singular and plural)

Job Sheet-2: Perform Create, Read, Update, and Delete (CRUD) operations on individual records in a Django model.

UoC Cover

OU-ICT-WADP-03- L4-V1: Use Django Model

Working Procedure / Steps

1. Create a Single Record
2. Read a Single Record
3. Update a Single Record
4. Delete a Single Record

Specification Sheet-2: Perform Create, Read, Update, and Delete (CRUD) operations on individual records in a Django model.

Technical Requirements

- **Python:** Python 3.7 or later
- **Django:** Django 3.2 or later
- **Database:** PostgreSQL or MySQL (or other supported database)
- **Text Editor/IDE:** Visual Studio Code, PyCharm, or Sublime Text

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Internet Connection:** A reliable internet connection for downloading and installing software, accessing online resources, and deploying the application.
- **Development Environment:** A virtual environment is recommended to isolate project dependencies.

Learning Outcome 3 : Create class-based view

Assessment Criteria	1. Django Template Syntaxes are applied 2. Built-In Django Filters are applied
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Django Template Syntaxes 2. Django Templates are configuration 3. Template Search Paths 4. Built-In Django Filters 5. Dates 6. Strings 7. Lists 8. Numbers
Activities/job/Task	1. Create a web view using a Django class-based view (CreateView) to handle form submission and creation of new model records.
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 3 : Create class-based view

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Create class-based view'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Define Tasks of the Project"	2. Read Information sheet 3: Create class-based view 3. Answer Self-check 3: Create class-based view 4. Check your answer with Answer key 3: Create class-based view
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet Job Sheet-3: Create a web view using a Django class-based view (CreateView) to handle form submission and creation of new model records.

Information sheet 3: Create class-based view

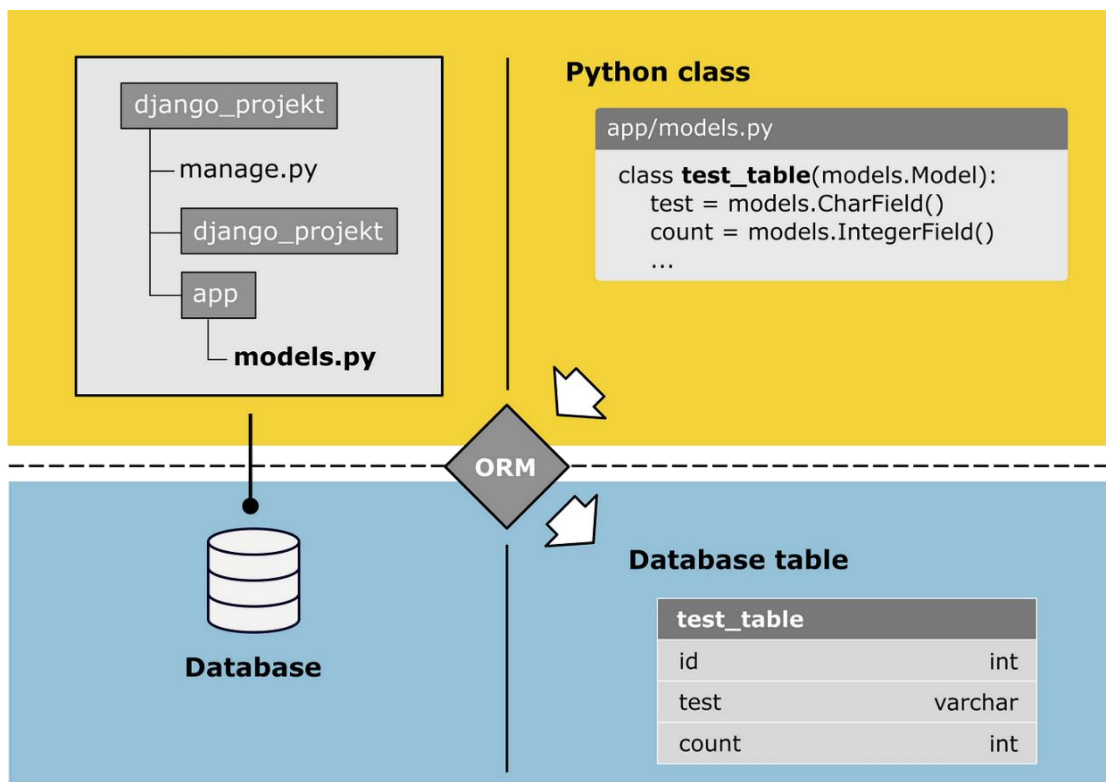
Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

3.1. Model records with class-based view are created
3.2. CRUD with class-based view is performed
3.3. Mixin is applied with class-based view

3.1. Model class with proper model data type and relationships

Previously, you learned how class-based views allow you to create views that operate with object oriented programming (OOP) principles (e.g. encapsulation, polymorphism and inheritance) leading to greater re-usability and shorter implementation times.



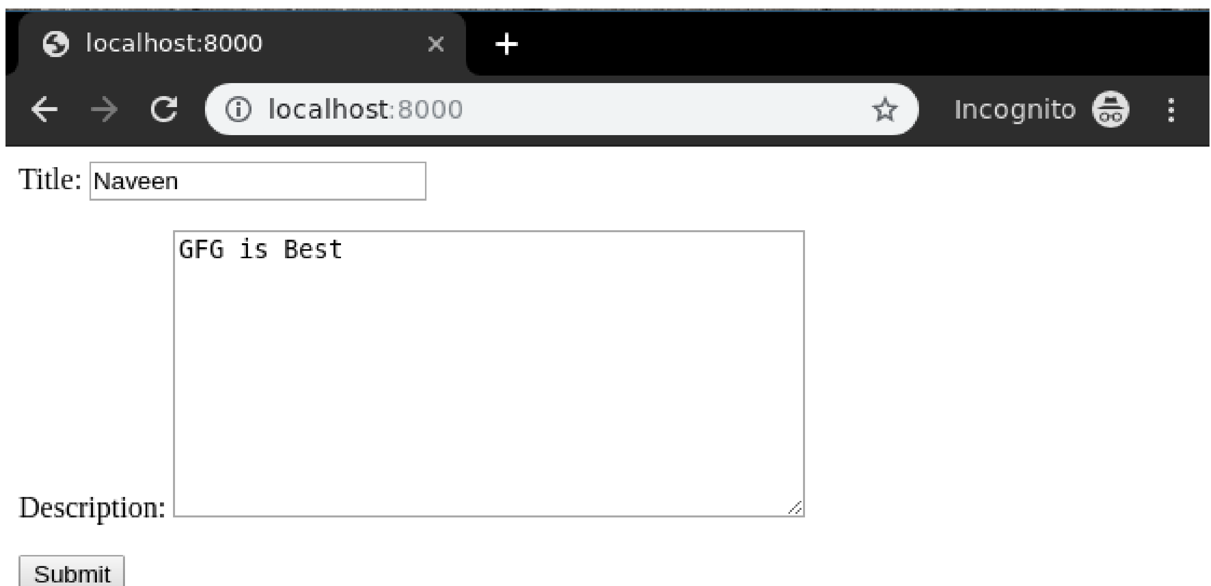
Unlike standard Django views -- explored in the early sections of chapter 2 -- which allow open ended logic to process a request and generate a response, class-based views with models encapsulate the logic performed against Django models in a more modular way.

For example, the logical patterns to create, read, update and delete model instances in a standard view method, generally follow a very consistent workflowget input data from a url or form, execute CRUD operation on the model and send the response to a template.

In the spirit of Django's DRY (Don't repeat yourself) principle, class-based views with models offer a way to cut down on the boilerplate code used in standard view methods and use class fields and methods to define the workflow used for Django model CRUD operations.

Create model records with the class-based view CreateView

As you've learned up to this point, the creation of Django model instances in real-life projects comes accompanied by a series of constructs that can includemodel forms, GET/POST request processing and the use of templates, among other things.



The Django CreateView class-based view is specifically designed to cut-down on the amount of boilerplate code needed to perform the creation of a model record.

Example: Django class-based view with CreateView to create model records

```
# views.py
from django.views.generic.edit import CreateView
from .models import Item, ItemForm
from django.core.urlresolvers import reverse_lazy
```

```
class ItemCreation(CreateView):
    model = Item
    form_class = ItemForm
    success_url = reverse_lazy('items:index')

# models.py
from django import forms
from django.db import models

class Menu(models.Model):
    name = models.CharField(max_length=30)

class Item(models.Model):
    menu = models.ForeignKey(Menu, on_delete=models.CASCADE)
    name = models.CharField(max_length=30)
    description = models.CharField(max_length=100)

class ItemForm(forms.ModelForm)
    class Meta
        model = Item
        fields = '__all__'
        widgets = {
            'description'forms.Textarea(),
        }

# urls.py
from django.conf.urls import url
from coffeehouse.items import views as items_views

urlpatterns = [
    url(r'^new/$', items_views.ItemCreation.as_view(), name='new'),
]

# templates/items/item_form.html
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="btn btn-primary">Create</button>
    </form>
```

The first part illustrates the ItemCreation class-based view that inherits its behavior from the CreateView class. Notice this view lacks a request reference, processing logic or return statement, all of which were common in the standard view methods. So what is the ItemCreation class-based view actually doing ?

Because you know beforehand you want to create a model record, the CreateView class -- used by the ItemCreation class-based view -- supports all the necessary boilerplate logic and requires a minimum set of code to fulfill its model record creation logic.

The ItemCreation class-based view, example uses the model field to tell Django to create Item model records. In addition, the form_class field specifies the ItemForm form -- also declared in example which is used to create model records. In addition, the success_url field is used to return control to the item:index url when model record creation is successful.

Why use reverse_lazy in class-based views, instead of reverse ?

Due to the simultaneous import order of models, views and urls in class-based views, using the standard reverse() method to resolve url names can result in the error*django.core.exceptions.ImproperlyConfiguredThe included URLconf '------' does not appear to have any patterns in it. If you see valid patterns in the file then the issue is probably caused by a circular import*.

The reverse_lazy method ensures any reverse url name resolution is attempted only after all models, views and urls have been properly imported, therefore it's the common choice in the context of class-based views.

You may still be left wondering, where are the save() and is_valid() methods for the ItemCreation class-based view if its creating model records ? By default there aren't any. Because you *just* want to create a model record, the parent CreateView class takes care of this supporting logic.

The next part contains the urls.py file with the hook to set up the class-based view into the application. In this case, the ItemCreation class-based view is set to run on the new/ url and is declared as part of the url() method by using the class-based view as_view() method -- Note: this url set up technique is identical for all class-based views.

So what happens when a user visits the new/ url ? Control is sent to the ItemCreation class-based view. And because this view's purpose is to create an Item model record, the class-based view looks for a template to render the form, under the TEMPLATES directory path of a project following the convention <app_name>/<model_name>_form.html.

In the last part of Example: , you can see the template templates/items/item_form.html, where templates represents a TEMPLATES directory path value, items the app name and item the model name defined for the class-based view. In addition, notice the contents of the item_form.html template use a standard form layout, which you can adjust like any Django form.

So where is the POST form handler and error handling ? By default, there isn't any either. Once a user gets an unbound form illustrated at the bottom of Example, the form processing and validation is taken care of behind the scenes by the ItemCreation class-based view. If the form contains errors, the template is re-rendered -- like any other form -- using the same template 3-9. If the form data is valid, form processing is deemed successful and the class-based view creates an Item model record -- like a model form -- redirecting control to the item:index url defined in the class-based view success_url field.

As you can see in this example, a class-based view that inherits its behavior from the CreateView class, cuts-down the boilerplate code needed to a create model record.

3.2. Apply Mixin with class-based view

In Django, **Mixins** are a way to add reusable chunks of code to class-based views (CBVs). Mixins allow you to reuse code across multiple views, keeping your views DRY (Don't Repeat Yourself).

A Mixin is typically a class that provides specific methods or functionality and is designed to be inherited by other classes. When applied to a class-based view, Mixins provide additional functionality without modifying the original class itself.

A. Understanding Mixins

In Django, mixins are used to implement common functionality that can be shared between different views. For example, you might use a mixin to provide permissions checking, context data, or queryset filtering.

B. Common Django Mixins

Some of the most commonly used mixins in Django include:

- **LoginRequiredMixin**: Ensures that only authenticated users can access a view.
- **PermissionRequiredMixin**: Ensures that only users with specific permissions can access a view.
- **ContextMixin**: Adds extra context to the response, often used to add common context to multiple views.
- **MultipleObjectMixin**: Helps in dealing with multiple objects for views like list views.

C. Example of Creating and Using Mixins

Let’s go through an example where we create a mixin and apply it to a class-based view.

a. Create a Custom Mixin

Let's say we want to create a mixin that checks if a user is the owner of an object (e.g., a Product), and only allows the owner to edit or delete the product.

```python
from django.http import Http404
from django.contrib.auth.mixins import LoginRequiredMixin
from .models import Product

class IsOwnerMixin:
    """
    Mixin to ensure that the logged-in user is the owner of the object.
    """
    def get_object(self, queryset=None):
        obj = super().get_object(queryset)
        if obj.owner != self.request.user:
            raise Http404("You do not have permission to access this object.")
        return obj
```

In this example:

- **IsOwnerMixin** checks if the owner of a Product is the currently logged-in user (based on the request.user).
- If the user is not the owner, it raises an Http404 exception.

b. Use the Mixin in a Class-Based View

Now, let's apply this IsOwnerMixin to a class-based view, such as a UpdateView, to allow only the owner of a Product to update it.

```python
from django.views.generic import UpdateView
from django.urls import reverse_lazy
from .models import Product
from .mixins import IsOwnerMixin

class ProductUpdateView(LoginRequiredMixin, IsOwnerMixin, UpdateView):
    model = Product
    fields = ['name', 'description', 'price']
    template_name = 'product_update.html'
    success_url = reverse_lazy('product_list')  # Redirect after successful update

    def get_queryset(self):
        # Optionally override the queryset to filter products
        return Product.objects.filter(owner=self.request.user)
```

In this example:

- **ProductUpdateView** uses both LoginRequiredMixin and IsOwnerMixin.
    - LoginRequiredMixin: Ensures that only authenticated users can access the view.

    - IsOwnerMixin: Ensures that only the owner of the Product can update it.
- **get_queryset()** filters the products, making sure the user can only update their own products.

c. Use the Mixin in a Delete View

We can apply the same IsOwnerMixin to a delete view:

```
from django.views.generic import DeleteView
from django.urls import reverse_lazy
from .models import Product
from .mixins import IsOwnerMixin

class ProductDeleteView(LoginRequiredMixin, IsOwnerMixin, DeleteView):
    model = Product
    template_name = 'product_confirm_delete.html'
    success_url = reverse_lazy('product_list')  # Redirect after successful deletion

    def get_queryset(self):
        # Optionally override the queryset to filter products
        return Product.objects.filter(owner=self.request.user)
```

This is a **delete view** that ensures only the owner of a Product can delete it.

D. Understanding the Order of Mixins

The order in which mixins are inherited in the class-based view is important. In the example above, we used:

```
class ProductUpdateView(LoginRequiredMixin, IsOwnerMixin, UpdateView):
```

The LoginRequiredMixin comes first to ensure that the user is authenticated before checking ownership with IsOwnerMixin.

Django processes the mixins and their methods in the order they are listed in the inheritance hierarchy. The method resolution order (MRO) determines the order in which mixins and methods are called.

E. Example of Built-In Mixins in Django

Django provides several built-in mixins that you can use directly. Some common ones include:

- **LoginRequiredMixin**: Ensures the user is authenticated.
- **PermissionRequiredMixin**: Ensures the user has specific permissions.
- **ContextMixin**: Allows adding custom context to the response.

Example of using **LoginRequiredMixin** and **PermissionRequiredMixin**:

```
from django.contrib.auth.mixins import LoginRequiredMixin, PermissionRequiredMixin
from django.views.generic import ListView
from .models import Product

class ProductListView(LoginRequiredMixin, PermissionRequiredMixin, ListView):
    model = Product
    template_name = 'product_list.html'
    context_object_name = 'products'
    permission_required = 'myapp.view_product'  # Specify the required permission

    def get_queryset(self):
        return Product.objects.all()
```

In this example:

- **LoginRequiredMixin** ensures that only authenticated users can access this view.
- **PermissionRequiredMixin** ensures that the user has the view_product permission.

F. Chaining Multiple Mixins

You can combine multiple mixins in your class-based views to achieve the desired behavior. Django class-based views use a **left-to-right, top-to-bottom** approach when processing mixins and their methods.

Example:

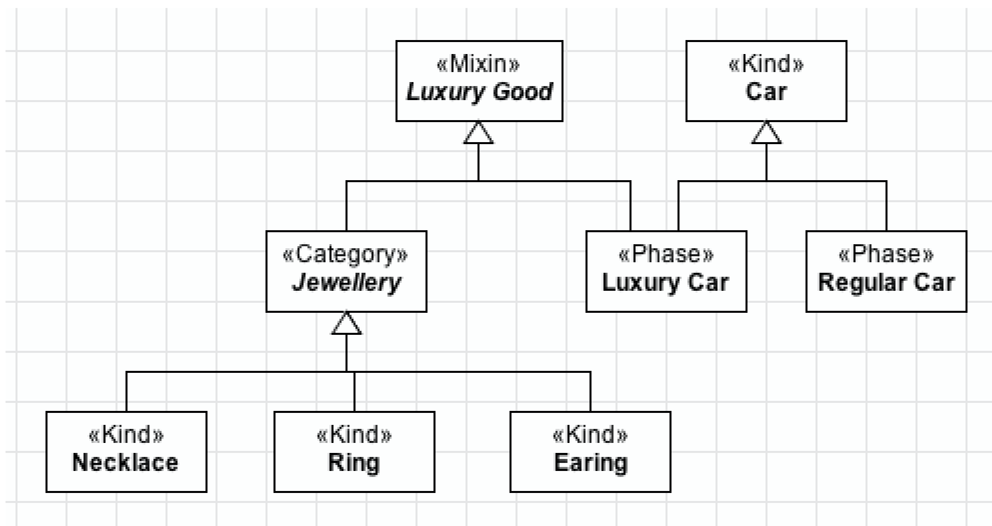
```
class MyView(LoginRequiredMixin, PermissionRequiredMixin, SomeOtherMixin, DetailView):
    # Your view code here
```

This ensures that:

1. The user is authenticated (LoginRequiredMixin).
2. The user has the required permission (PermissionRequiredMixin).
3. Any other functionality from SomeOtherMixin is applied.

3.3. Mixin with class-based view

Previously, you learned how class-based views allow you to create views that operate with object oriented programming (OOP) principles (e.g. encapsulation, polymorphism and inheritance) leading to greater re-usability and shorter implementation times. Now that you know how Django models work, we can address class-based views that integrate with models described in the last part of table.



Unlike standard Django views -- explored in the early sections which allow open ended logic to process a request and generate a response, class-based views with models encapsulate the logic performed against Django models in a more modular way.

For example, the logical patterns to create, read, update and delete model instances in a standard view method, generally follow a very consistent workflowget input data from a url or form, execute CRUD operation on the model and send the response to a template.

In the spirit of Django's DRY (Don't repeat yourself) principle, class-based views with models offer a way to cut down on the boilerplate code used in standard view methods and use class fields and methods to define the workflow used for Django model CRUD operations.

a. Create model records with the class-based view CreateView

Create View refers to a view (logic) to create an instance of a table in the database. Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

- Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

Class based views are simpler and efficient to manage than function-based views. A function based view with tons of lines of code can be converted into a class based views with few lines only. This is where Object Oriented Programming comes into impact.

```
# import the standard Django Model

# from built-in library

from django.db import models



# declare a new model with a name "GeeksModel"

class GeeksModel(models.Model):



    # fields of the model

    title = models.CharField(max_length = 200)

    description = models.TextField()



    # renames the instances of the model

    # with their title name

    def __str__(self):

        return self.title
```

After creating this model, we need to run two commands in order to create Database for the same.

```
Python manage.py makemigrations

Python manage.py migrate
```

Class Based Views automatically setup everything from A to Z. One just needs to specify which model to create View for and the fields. Then Class based CreateView will automatically try to find a template in app_name/modelname_form.html. In our case it is geeks/templates/geeks/geeksmodel_form.html. Let's create our class based view. In geeks/views.py,

```
from django.views.generic.edit import CreateView
from .models import GeeksModel

class GeeksCreate(CreateView):

    # specify the model for create view
    model = GeeksModel

    # specify the fields to be displayed

    fields = ['title', 'description']
```

Now create a url path to map the view. In geeks/urls.py,

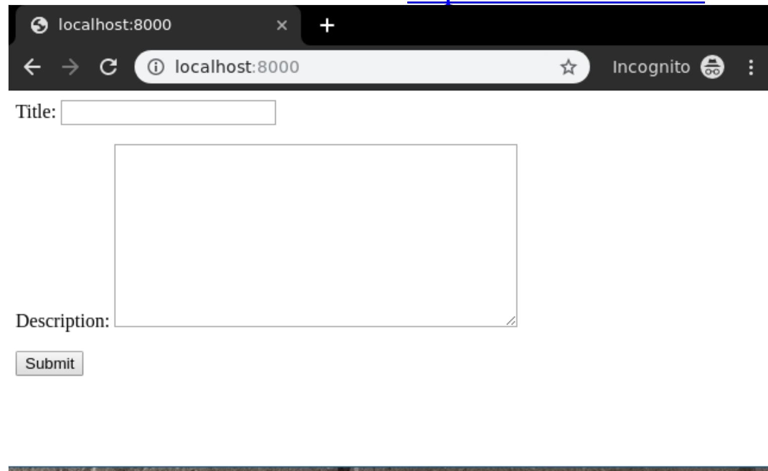
```
from django.urls import path

# importing views from views..py
from .views import GeeksCreate
urlpatterns = [
    path('', GeeksCreate.as_view() ),
]
```

Create a template in templates/geeks/geeksmodel_form.html,

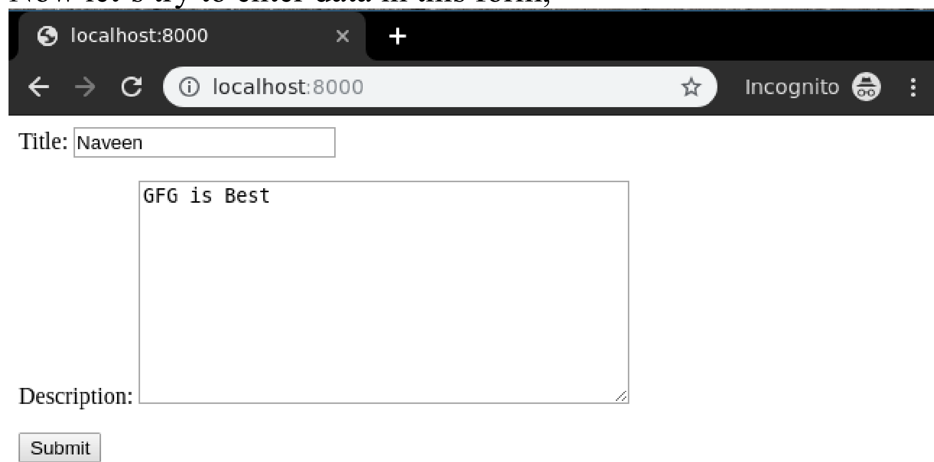
- html

```
<form method="POST" enctype="multipart/form-data">

    <!-- Security token -->
    {% csrf_token %}

    <!-- Using the formset -->
    {{ form.as_p }}

    <input type="submit" value="Submit">
</form>
```

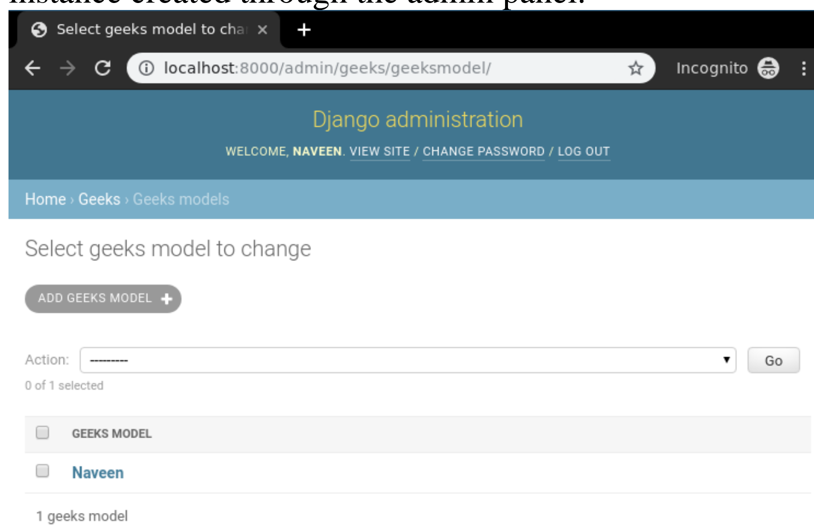
Let's check what is there on http://localhost:8000/



Now let's try to enter data in this form,



Bingo.! Create view is working and we can verify it using the instance created through the admin panel.



This way one can create view for a model in Django.

b. Why use reverse_lazy in class-based views, instead of reverse ?

Due to the simultaneous import order of models, views and urls in class-based views, using the standard reverse() method to resolve url names can result in the error*django.core.exceptions.ImproperlyConfiguredThe included URLconf '------' does not appear to have any patterns in it. If you see valid patterns in the file then the issue is probably caused by a circular import*.

The reverse_lazy method ensures any reverse url name resolution is attempted only after all models, views and urls have been properly imported, therefore it's the common choice in the context of class-based views.

You may still be left wondering, where are the save() and is_valid() methods for the ItemCreation class-based view if its creating model records ? By default there aren't any. Because you *just* want to create a model record, the parent CreateView class takes care of this supporting logic.

The next part contains the urls.py file with the hook to set up the class-based view into the application. In this case, the ItemCreation class-based view is set to run on the new/ url and is declared as part of the url() method by using the class-based view as_view() method -- Note: this url set up technique is identical for all class-based views .

So what happens when a user visits the new/ url ? Control is sent to the ItemCreation class-based view. And because this view's purpose is to create an Item model record, the class-based view looks for a template to render the form, under the TEMPLATES directory path of a project following the convention <app_name>/<model_name>_form.html.

In the last part of Example, you can see the template templates/items/item_form.html, where templates represents a TEMPLATES directory path value, items the app name and item the model name defined for the class-based view. In addition, notice the contents of the item_form.html template use a standard form layout, which you can adjust like any Django form.

So where is the POST form handler and error handling? By default, there isn't any either. Once a user gets an unbound form illustrated at the bottom of Example, the form processing and validation is taken care of behind the scenes by the ItemCreation class-based view. If the form contains errors, the template is re-rendered -- like any other form --

using the same template. If the form data is valid, form processing is deemed successful and the class-based view creates an Item model record -- like a model form -- redirecting control to the item:index url defined in the class-based view success_url field.

As you can see in this example, a class-based view that inherits its behavior from the CreateView class, cuts-down the boilerplate code needed to a create model record.

CreateView fields and methods

While at first glance a class-based view that inherits its behavior from the CreateView class can appear to be inflexible, it's possible to override its default behaviors like any Django construct.

As it turns out, the CreateView class inherits its behavior from many other Django class-based views, which is what gives
the CreateView class -- and its implementing child classes like the class-based view - its behind-the-scene powers. The CreateView class inherits its behavior from the following class-based view classes:
django.views.generic.detail.SingleObjectTemplateResponseMixin
django.views.generic.base.TemplateResponseMixin
django.views.generic.edit.BaseCreateView
django.views.generic.edit.ModelFormMixin
django.views.generic.edit.FormMixin
django.views.generic.detail.SingleObjectMixin
django.views.generic.edit.ProcessFormView
django.views.generic.base.View

So why are these classes even important ? Because they provide the default behavior for all CreateView class-based views. The scant fields declared in the ItemCreation class-based view are only three fields
for CreateView class-based views. It's possible to declare over a dozen more fields and methods -- that belong to this past list of classes -- to provide behaviors, such asusing another template other
than <app_name>/<model_name>_form.html; specifying the content type for the response (e.g. text/csv); custom methods to run when a model form is valid or invalid; as well as declaring custom methods to manually execute GET and POST workflow logic.

Note: A CreateView class-based view inherits many fields and methods from its parent classes. The following options are the most common ones, for an exhaustive list consult each of the CreateView parent classes.

c. Basic CreateView optionsmodel, form_class and success_url fields

As you've already seen, the essential logic fulfilled by
a CreateView class-based view is to create a model record using a form, which in turn requires a basic set of parameters. First, the model field is basic to the whole operation, because a CreateView class-based view must know beforehand which type of model record to create. Second, the form_class is also a basic parameter, because a user must be presented with a form to capture the data to create the model record.

Finally, because successfully creating a model record entail notifying a user about the action and moving away from the form page,
the success_url field is also a basic part of a CreateView class-based view to indicate where to redirect a user after a model record is created.

Customize template name, MIME type and context template_name and content_type fields & get_context_data() method

Sometimes relying on the CreateView class-based view template naming convention <app_name>/<model_name>_form.html is unfeasible, either because you have a pre-existing template to reuse or simply because you don't like the default convention. You can declare the template_name field as part of a CreateView class-based view to override this convention. Similarly, it's also possible to override the default MIME type used by a class-based view response -- if the template contains something other than text/html (e.g. text/csv) -- using the content_type field as part of a CreateView class-based view.

In addition, you can alter the context data passed to a class-based view template by defining an implementation of
the get_context_data() method. This last process is common when you need to pass additional data to a template -- besides the form reference -- or change the actual form reference to another name.

Example: Django class-based view with CreateView with template_name, content_type and get_context_data()

```
# views.py
from django.views.generic.edit import CreateView
from .models import Item, ItemForm, Menu

class ItemCreation(CreateView):
    template_name = "items/item_form.html"
    context_type = "text/html"
    model = Item
    form_class = ItemForm
```

```
    success_url = reverse_lazy('items:index')
    def get_context_data(self,**kwargs):
        kwargs['special_context_variable'] = 'My special context
variable!!!'
        context = super(ItemCreation, self).get_context_data(**kwargs)
        return context
```

You can see the template_name and content_type fields are declared as part of the class-based view. In this particular case, both fields values are assigned their default values -- making them redundant -- for simplicity, but you can adjust accordingly to your needs.

The get_context_data() method 3-10 first adds the custom special_context_variable key to make it available to the class-based view template (i.e. items/item_form.html). Next, a call is made to the parent class's get_context_data() method (i.e. CreateView) to run its context set up logic, which consists of setting up the form reference which is also used in the template. Finally, the context reference with all the template context values is returned by the get_context_data() method for use inside the template.

As you can see, by simply adding fields and overriding methods in a CreateView class-based view, you can easily start changing its default behavior.

d. Customize form initialization and validationinitial field, get_initial(), get_form(), form_valid() and form_invalid() methods

The initial field on a CreateView class-based view works just like a standard form's initial argument to specify default values for an unbound form. For example, declaring the field initial = {'size':'L'} on a CreateView class-based view sets its form's size field to L.

Form initialization can sometimes require more complex requirements than a single line statement, in which a CreateView class-based view also offers the get_initial() method. The get_initial() method functions like the __init__ method used in standard forms -- in that you can introduce open-ended logic to set up default values -- but is intended solely to set up initial values and to return a dictionary of values -- unlike the __init__ method where you can introduce other actions besides default form values (e.g. change widgets, add validation).

Example: Django class-based view with CreateView with initial and get_initial()

```
# views.py
from django.views.generic.edit import CreateView
```

```
from .models import Item, ItemForm, Menu

class ItemCreation(CreateView):
    initial = {'size':'L'}
    model = Item
    form_class = ItemForm
    success_url = reverse_lazy('items:index')
    def get_initial(self):
        initial_base = super(ItemCreation, self).get_initial()
        initial_base['menu'] = Menu.objects.get(id=1)
        return initial_base
```

The first step set the initial field to set the class-based view's form size field to L. Next, the get_initial() method is declared to add another default value to a class-based view form.

Inside the the get_initial() method, the first step is to call the parent class's get_initial() method (i.e. CreateView) to run its initial set up logic, this ensures the class's initial field value (i.e.{'size':'L'}) is taken into account to as part of the initial value. Next, the initial_base reference is updated to set the form's menu field to the Menu record with id=1. Finally, the get_initial() method returns a dictionary containing both the initial field value set by the class-based view and the custom form value set in its body.

The get_form() method of a CreateView class-based view is designed to tap into the full initialization process of a form and not just on setting its default values like the initial field and get_initial() method. This makes get_form() method suited to perform broader form initialization tasks like setting form widgets and validation initialization, like its done in the __init__ method in standard forms. Inclusively, it's possible to use the get_form() method to specify default form values and forgo the use of initial and get_initial() altogether.

Example: Django class-based view with CreateView with get_form()

```
# views.py
from django.views.generic.edit import CreateView
from .models import Item, ItemForm, Menu

class ItemCreation(CreateView):
    initial = {'size':'L'}
    model = Item
    form_class = ItemForm
    success_url = reverse_lazy('items:index')
    def get_form(self):
```

```
        form = super(ItemCreation, self).get_form()
        initial_base = self.get_initial()
        initial_base['menu'] = Menu.objects.get(id=1)
        form.initial = initial_base
        form.fields['name'].widget = forms.widgets.Textarea()
        return form
```

The first step in the get_form() method is to call the parent class's get_form() method (i.e. CreateView) to get the base form. Next, a call is made to the class-based view's get_initial() method to get its initial form value, as well as set up a default value for the form's menu field using a model query..

Next, the form initialization dictionary is assigned to the base form using the standard initial form reference. Finally, before returning an instance of the form class, a custom widget is set on the form's name field, overriding the default widget of the form name field.

In addition to initialization tasks, the form validation process for a CreateView class-based view can also be customized. The form_valid() and form_invalid() methods are used to access the points at which a class-based view form is deemed successful or erroneous, respectively.

Example: Django class-based view with CreateView with form_valid() and form_invalid()

```
# views.py
from django.views.generic.edit import CreateView
from django.http import HttpResponseRedirect
from django.contrib import messages
from .models import Item, ItemForm, Menu

class ItemCreation(CreateView):
    initial = {'size':'L'}
    model = Item
    form_class = ItemForm
    success_url = reverse_lazy('items:index')
    def form_valid(self,form):
        super(ItemCreation,self).form_valid(form)
        # Add action to valid form phase
        messages.success(self.request, 'Item created successfully!')
        return HttpResponseRedirect(self.get_success_url())
    def form_invalid(self,form):
        # Add action to invalid form phase
```

return self.render_to_response(self.get_context_data(form=form))

As you can see, the form_valid() method gains access to the form instance via its form input argument, which in turn allows you to perform actions on the form when it passes its validation phase. In this case, no action is taken on the form itself to simplify things, but an additional piece of logic is added to illustrate the customization process.

The first step in the form_valid() is a call to the parent class's form_valid() method (i.e. CreateView) to run its form validation set up logic, this ensures the base class's form validation is run first to verify any form rule violations (e.g. add errors to the form). If a call to the parent's form_valid() method detects a rule violation, then the class's form_valid() method short-circuits and falls back to the form_invalid() method. If the parent's form_valid() method passes, the logic continues to add a Django message framework success message to present to a user.

Finally, because you're handling a valid form workflow, the form_valid() method must explicitly redirect a user to a location to finish its work. In this case, a standard Django HttpResponseRedirect method is used with the class-based view's get_success_url() method, the last of which gets the class-based view success_url field value.

The form_invalid() method does nothing in particular to handle an invalid form. It simply does the minimum amount work -- by means of class-based view methods -- which is to return control to the same template location and add the context that contains the form with errors to present to a user.

As you can see, the benefit of the form_valid() and form_invalid() methods in a CreateView class-based view is they allow you to customize the form validation workflow, without the need to modify other parts of a CreateView workflow (e.g. saving the form data to the database).

e. Customize view method workflowget() and post() methods

Beside the previous customization options for a CreateView class-based view, it's also possible to get absolute control over the workflow done by a class-based view with either the get() or post() methods. The get() method is used to tap into the HTTP GET workflow associated with a class-based view, where as the post() method is used to tap into the HTTP POST workflow associated with a class-based view.

Although the use of the get() or post() methods can offer some of the greatest flexibility to a class-based view, they also require to explicitly declare the initialization, validation and redirect sequences of a class-based view.

Still, sometimes the appeal of class-based views is so great, the use of the get() or post() methods is warranted. Example illustrates an Example: that uses the get() and post() methods in a CreateView class-based view.

Example: Django class-based view with CreateView with get() and post()

```
# views.py
from django.views.generic.edit import CreateView
from django.shortcuts import render
from django.contrib import messages

class ItemCreation(CreateView):
   initial = {'size':'L'}
   model = Item
   form_class = ItemForm
   success_url = reverse_lazy('items:index')
   template_name = "items/item_form.html"
   def get(self,request, *args, **kwargs):
      form = super(ItemCreation, self).get_form()
      # Set initial values and custom widget
      initial_base = self.get_initial()
      initial_base['menu'] = Menu.objects.get(id=1)
      form.initial = initial_base
      form.fields['name'].widget = forms.widgets.Textarea()
      # return response using standard render() method
      return render(request,self.template_name,
                {'form':form,
                 'special_context_variable':'My special context variable!!!'})

   def post(self,request,*args, **kwargs):
      form = self.get_form()
      # Verify form is valid
      if form.is_valid():
         # Call parent form_valid to create model record object
         super(ItemCreation,self).form_valid(form)
         # Add custom success message
         messages.success(request, 'Item created successfully!')
         # Redirect to success page
```

```
        return HttpResponseRedirect(self.get_success_url())
    # Form is invalid
    # Set object to None, since class-based view expects model record object
    self.object = None
    # Return class-based view form_invalid to generate form with errors
    return self.form_invalid(form)
```

The first step in the get() method is to create an unbound form using the parent class's get_form() method (i.e. CreateView). Because the get() method gives you full control over the workflow, it would be equally valid to create an unbound form using standard form syntax (e.g. form = ItemForm()), but since it's a class-based view, the Example: leverages a class-based view construct. Once the unbound form is created, a series of initial values and a custom widget is set on the form, just like it's done in the previous class-based view examples.

Once the unbound form is ready, notice the return statement of the get() method uses the standard render() method used in regular view methods. In this case, the render() method redirects control to the class-based view template and sets the template context with the unbound form and an additional special_context_variable variable to use in the template.

Next, the post() method is tasked with processing the form with user data. The first step in the post() method is to get a bound form instance using the class-based view get_form() class. Similarly to the get() method, it would be equally valid to create a bound form using standard form syntax (e.g. form = ItemForm(request.POST)).

With a bound form instance, a check is made to verify if the user provided form data is valid, just like it's done with standard forms (e.g. form.is_valid()). If the form data is valid, a call is made to the parent class's is_valid() method (i.e. CreateView), which ensures the core logic of a class-based view is executed when a form is valid (e.g. saving the form data to a database). Here it's possible once again to use any standard model construct, but calling the parent class's is_valid() method is easier to execute this routine logic to create a model object record. Once the routine validation logic is complete, a success message is added to present to an end user and a redirect is made to the class-based view's success url. If the form data is invalid, the class-based view's object field is set to None -- since class-based views expect to handle an object record instance in POST processing -- and control is returned using the class-based form_invalid() method which takes care of the underlying details vs. using the render() method to create a standard view method response.

As you can now understand from this Example, a CreateView class-based view can be just as flexible as a standard view method. It's simply a matter of knowing and understanding the different fields and methods supported by a CreateView class-based view. Of course, if you feel the custom logic for a CreateView class-based view becomes to unwieldy, you can always fall back to a standard view method .

Similar to the process of creating model records, the process to read model records also follows a near identical process for all modelscreate a query to get model record(s) and then use a template to display the model record(s). The Django ListView and DetailView class-based views are specifically designed to cut-down on the amount of boilerplate code needed to display a list of Django model records and a single Django model record, respectively.

The ListView class-based view can quickly set up a query for a list of model records and display them in a template.

Example: Django class-based view with ListView to read list of records

```
# views.py
from django.views.generic.list import ListView
from .models import Item

class ItemList(ListView):
   model = Item

# urls.py
from django.conf.urls import url
from coffeehouse.items import views as items_views

urlpatterns = [
   url(r'^$',items_views.ItemList.as_view(),name="index"),
]

# templates/items/item_list.html
 {% regroup object_list by menu as item_menu_list %}
 {% for menu_section in item_menu_list %}
  <li>{{ menu_section.grouper }}
   <ul>
      {% for item in menu_section.list %}
      <li>{{item.name|title}}</li>
      {% endfor %}
   </ul>
   </li>
```

{% endfor %}

The first definition is the ItemList class which inherits its behavior from the ListView class-based view class. The model field in the ItemList class set to Item, tells Django to generate a list of all Item model records (e.g. Item.objects.all())

Next, the ListView class-based view is hooked up to a root url regular exression -- r'^$' -- using the as_view() method, the last of which is available on all class-based views and was also used in the past section to set up a a CreateView class-based view.

Finally, the last part illustrates the template item_list.html which generates a loop over the object_list reference, the last of which is the default context variable used by a ListView class-based view that contains the model record list.

Recapping, the most important default behaviors of a ListView class-based view are:

- The list of records is made from all records of the model defined in the model option.
- The template to render the list of records uses the convention <app_name>/<model_name>_list.html under the TEMPLATES directory path of a project.
- The context variable passed to a template (i.e. the one containing the records) is named object_list.

As you can see in this example, a ListView class-based view cuts-down the boilerplate code needed to present a list of model records to one field.

The DetailView class-based view is another record reading construct, designed to quickly set up a single record query and present the results in a template.

Example: Django class-based view with DetailView to read model record

```
# views.py
from django.views.generic. import DetailView
from .models import Item

class ItemDetail(DetailView):
   model = Item

# urls.py
from django.conf.urls import url
from coffeehouse.items import views as items_views
```

```
urlpatterns = [

url(r'^(?P<pk>\d+)/$',items_views.ItemDetail.as_view(),name="detail"),
]

# templates/items/item_detail.html
<h4> {{item.name|title}}</h4>
<p>{{item.description}}</p>
<p>${{item.price}}</p>
<p>For {{item.get_size_display}} sizeOnly {{item.calories}} calories
{% if item.drink %}
and {{item.drink.caffeine}} mg of caffeine.</p>
{% endif %}
</p>
```

The first definition is the ItemDetail class which inherits its behavior from the DetailView class-based view class. The model field in the ItemDetail class is set to Item, which tells Django to get an Item model record. Unlike the ListView class-based view class that reads all model records, a DetailView class-based view class must always limit its model query to a single record, which is where the class-based view url definition comes into play.

The DetailView class-based view is hooked up to a root url regular exression -- r'^$' -- using the as_view() method -- just like other class-based views -- but notice the url definition contains the (?P<pk>\d+) url parameter, which gets passed to the class-based view to delimit the model query to a single record.

For example, if a request is made on the url /items/1/, the 1 is assigned to the pk parameter, which is passed to the classed-based view, to build the model query Item.objects.get(pk=1) that gets the Item model record with pk=1 -- Note: the pk field stands for primary key, which is generally equivalent to the id field.

In this manner, as the url requests made on the DetailView class-based view change (e.g. /items/2/, /items/3/), so does the backing query made for a model a record, and with it the record returned to be displayed in the template.

Finally, the last part illustrates the template item_detail.html which outputs various fields of the item reference that represents the model record. In this case, the item reference is used because the default context variable used by a DetailView class-based view for a model record is the name of the model itself.

Recapping, the most important default behaviors of a DetailView class-based view are:

- The model record is determined based on the model option and the url pk parameter that delimits the query to a single record based on the model's primary key.
- The template to render the record uses the convention <app_name>/<model_name>_detail.html under the TEMPLATES directory path of a project.
- The context variable passed to a template (i.e. the one containing the record) is named after the model of the class-based view. (e.g. if model=Item, the context variable is named item).

As you can see in this example, a class-based view that inherits its behavior from the DetailView class, cuts-down the boilerplate code needed to present a single model record.

f. ListView fields and methods

Similar to the CreateView class-based view presented earlier, it's possible to override many of default behaviors of a ListView class-based view.

As it turns out, the ListView class-based view inherits its behavior from many other Django class-based views, presented in the following list:

django.views.generic.list.MulTip:leObjectTemplateResponseMixin

django.views.generic.base.TemplateResponseMixin

django.views.generic.list.BaseListView

django.views.generic.list.MulTip:leObjectMixin

django.views.generic.base.View

Note: A ListView class-based view inherits many fields and methods from its parent classes. The following options are the most common ones, for an exhaustive list consult each of the ListView parent classes.

g. Basic ListView optionmodel field

As you saw , the essential logic fulfilled by a ListView class-based view is to create a list of records from a model. Therefore the model field option is essential to indicate on which model to create a record list.

The next sections describe how to customize a ListView class-based view, including how to delimit and generate mulTip:le pages for a list of records, as well as how to override other default behaviors.

Customize template context reference namecontext_object_name

, you can see the template of the ListView class-based view uses the rather unfriendly context variable named object_list. This is the default behavior, but to help template editors, it's possible to use a different context variable to contain the list of records.

The context_object_name field is used to define a custom context variable name to manipulate the list of records inside a template (e.g. context_object_name = 'item_list').

h. Customize record listqueryset and ordering fields & pagination behavior

By default, a ListView class-based view generates a list for all records that belong to a model. While this behavior is reasonable, it can also be necessary to create a ListView class-based view that returns a more limited criteria, in which case it's necessary to delimit the backing model query. The queryset field is used to define a custom query to generate a list of records.

Another helpful option to customize a record list is to specify the field order in which to generate the record list. Similar to the standard order_by() method used in model queries, a ListView class-based view can specify the ordering field to define the sort order of a record list.

Example: Django class-based view with ListView to reduce record list with queryset

```
# views.py
from django.views.generic.list import ListView
from .models import Item

class ItemList(ListView):
    model = Item
    queryset = Item.objects.filter(menu__id=1)
    ordering = ['name']
```

As you can see, the queryset field is assigned a standard model query to produce Item records with a menu id=1. In this manner, the resulting record list passed by ListView the class-based view to the template only

contains Item records that match this criteria. In addition, Example, also makes use of the ordering = ['name'] field, which in this case ensures the query to generate the record list is sorted by the Item model's name field.

Although the queryset option is helpful to delimit the size of the record list presented by a ListView class-based view, sometimes it's necessary to deal with a large record list and still be able to delimit the amount of results displayed in a template. For such scenarios, you can use pagination to split out a large record list over mulTip:le pages.

Since pagination by definition depends on the use of mulTip:le pages, it forces you to adjust not only a ListView class-based view definition, but also the url structure and template used by the class-based view to support mulTip:le pages.

Example: Django class-based view with ListView to read list of records with pagination

```
# views.py
from django.views.generic.list import ListView
from .models import Item

class ItemList(ListView):
    model = Item
    paginate_by = 5

# urls.py
from django.conf.urls import url
from coffeehouse.items import views as items_views

urlpatterns = [
    url(r'^$',items_views.ItemList.as_view(),name="index"),

url(r'^page/(?P<page>\d+)/$',items_views.ItemList.as_view(),name="
page"),
]

# templates/items/item_list.html
  {% regroup object_list by menu as item_menu_list %}
  {% for menu_section in item_menu_list %}
   <li>{{ menu_section.grouper }}
    <ul>
       {% for item in menu_section.list %}
       <li>{{item.name|title}}</li>
       {% endfor %}
```

```
   </ul>
   </li>
 {% endfor %}
{% if is_paginated %}
   {{page_obj}}
{% endif %}
```

The relevant pagination logic is in bold. First off, the paginate_by = 5 is added to the class-based view definition, which tells Django to limit the record list to five records per page. Since the ListView class-based view will require a page number to display records beyond the first five records of query -- since pages are limited 5 -- the natural place to obtain a page number is through a url (e.g. /page/1/ to create a list with the first five records of a query, /page/2/ to create a list with the records six through ten, etc).

Next, you can see a new url definition with the (?P<page>\d+) parameter hooked up the ListView class-based view. This url definition allows a request that matches the regular expression pattern (e.g. /page/1/,/page/2/) to pass the url page parameter value to the class-based view. When a ListView class-based view detects a url page parameter value with the presence of the paginate_by option, it adjusts the query for the record list to return the appropriate set of records to a template based on the page (e.g. /page/1/ generates a list of records from one to five of the overall query, /page/2/ generates a list of records from six to ten of the overall query, etc).

Finally, the last section represents the backing template for a ListView class-based view that uses pagination. The {% is_paginated %} tag and {{page_obj}} context reference are used let users know on which page of the overall record list they're on.

i. DetailView fields and methods

Like other class-based views, a DetailView class-based view can also be created with custom fields and methods to override their default behaviors.

As it turns out, the DetailView class inherits its behavior from many other Django class-based views, which are described in the following list:
django.views.generic.detail.SingleObjectTemplateResponseMixin
django.views.generic.base.TemplateResponseMixin
django.views.generic.detail.BaseDetailView
django.views.generic.detail.SingleObjectMixin

django.views.generic.base.View

Note: A DetailView class-based view inherits many fields and methods from its parent classes. The following options are the most common ones, for an exhaustive list consult each of the DetailView parent classes.

j. Basic DetailView optionsmodel field and url with pk parameter

As you saw, the essential logic fulfilled by a DetailView class-based view is to get a model record and display it in a template. Therefore the model field is one of the essential pieces to this type of class-based view and must always by provided.

In order to get a specific model record, a DetailView class-based view must also define a url parameter which helps it select a model record. By default, this url parameter must be named pk and its value is used to perform a query on the model value using the pk field, which is generally equivalent to a model's id field.

Example: Django class-based view with DetailView and slug_field option

```
# views.py
from django.views.generic import DetailView
from .models import Item

class ItemDetail(DetailView):
   model = Item
   slug_field = 'name__iexact'

# urls.py
from django.conf.urls import url
from coffeehouse.items import views as items_views
urlpatterns = [

url(r'^(?P<slug>\w+)/$',items_views.ItemDetail.as_view(),name="deta
il"),
]
```

The first important aspect is the url definition has a url parameter named slug that matches any word pattern due to the w+ regular expression. This means urls such as /items/espresso/ and /items/latte/ match this url, unlike the

prior DetailView class-based url definition that uses the pk parameter to match numeric values with d+ regular expression.

When a DetailView class-based view receives a url parameter named slug, it signals the class-based view that the model record query should done on a model field other thank pk. Because this slug value can be ambiguous (e.g. it can represent one of several model field values, such as name, cost or description), it's necessary to qualify which model field the slug value represents, which is the purpose of slug_field field.

In the case of Example, the slug_field field is set to name__iexact, which tells the DetailView class-based view to perform a case-insensitive model query with the slug value on the Item model name field. The case insensitive query is provided by the __iexact lookup and is important in case the database record values are mixed-case (e.g. the url /items/capuccino/ would match an Item name record capuccino, Capuccino or CAPUCCINO. Without case inventivenesses, the Item name record would need to be exactly capuccino to match the url).

Although not presented in the sample, another option available for a DetailView class-based view is the slug_url_kwarg option, which has a similar to the pk_url_kwarg option described earlier.

k. Update model records with the class-based view UpateView

When you update a Django model record, the typical process involvesfetching the model record you want to update from a database, presenting the model data in a form so a user can make changes and finally saving the changes back to the database. The Django UpdateView class-based view is specifically designed to cut-down on the amount of boilerplate code needed to update a Django model record.

Due to its functionality, the UpdateView class-based view operates like a combination of the CreateView -- which creates a model record through a form -- and the DetailView -- which displays a model record.

Example: Django class-based view with UpdateView to edit a record

```
# views.py
from django.views.generic import UpdateView
from .models import Item

class ItemUpdate(UpdateView):
   model = Item
   form_class = ItemForm
   success_url = reverse_lazy('items:index')

# urls.py
from django.conf.urls import url
from coffeehouse.items import views as items_views

urlpatterns = [
    url(r'^edit/(?P<pk>\d+)/$',          items_views.ItemUpdate.as_view(),
name='edit'),
]

# templates/items/item_form.html
<form method="post">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit" class="btn btn-primary">
        {% if object == None%}Create{% else %}Update{% endif %}
      </button>
</form>
```

The first definition is the ItemUpdate class which inherits its behavior from the UpdateView class-based view class. The model field in the ItemUpate class sets a value to Item, which tells Django to update Item model records. In addition, because you're dealing with an update operation, the Item model record must be presented in a form -- which is the purpose of the from_class option -- as well as define a success url to redirect a user if an update is successful, which is the purpose of the success_url option.

Next is the url definition that hooks up the UpdateView class-based view class. Because an UpdateView class-based view must present a specific model record to edit, it relies on a url parameter to limit a query to a single record. In this case, notice the url contains the pk url parameter that gets passed to the class-based view -- just like it's done with a DetailView class-based view.

l. UpdateView fields and methods

Like other class-based views, UpdateView class-based views can also be created with custom fields and methods to override their default behaviors.

As it turns out, the UpdateView class inherits its behavior from many other Django class-based views, presented in the following list:
django.views.generic.detail.SingleObjectTemplateResponseMixin
django.views.generic.base.TemplateResponseMixin
django.views.generic.edit.BaseUpdateView
django.views.generic.edit.ModelFormMixin
django.views.generic.edit.FormMixin
django.views.generic.detail.SingleObjectMixin
django.views.generic.edit.ProcessFormView
django.views.generic.base.View

Basic UpdateView optionsmodel, form_class and success_url fields & url with pk parameter

As you saw , the essential logic fulfilled by an UpdateView class-based view is to get a model record and display it in a form for editing. Therefore the model field is one of the essential pieces for this type of class-based view and must always by provided. In addition, because a record is updated via form, it's also necessary to specify a form through the form_class option, as well as a success url through the success_url option for when an update is successful.

In order to get a specific model record, an UpdateView class-based view must also define a url parameter which helps it select a model record to edit. By default, this url parameter must be named pk and its value is used to perform a query on the model value using the pk field, which is generally equivalent to a model's id field.

m. Delete records with the class-bases view DeleteView

When you delete a Django model record, besides executing the actual delete operation, it's generally a good practice to present a confirmation page to a user. The Django DeleteView class-based view is specifically designed to cut-down on the amount of boilerplate code needed to delete a model record, while presenting a confirmation page.

Example: Django class-based view with DeleteView to delete record

```
# views.py
from django.views.generic.edit import DeleteView
from .models import Item

class ItemDelete(UpdateView):
    model = Item
    success_url = reverse_lazy('items:index')

# urls.py
from django.conf.urls import url
from coffeehouse.items import views as items_views

urlpatterns = [
    url(r'^delete/(?P<pk>\d+)/$', items_views.ItemDelete.as_view(), name='delete'),
]

# templates/items/item_confirm_delete.html
  <form method="post">
      {% csrf_token %}
      Do you really want to delete "{{ object }}"?
      <button class="btn btn-primary" type="submit">Yes, remove it!</button>
  </form>
```

The first definition is the ItemDelete class which inherits its behavior from the DeleteView class-based view class. The model field in the ItemDelete class sets a value to Item, which tells Django to delete Item model records. In addition, because you'll want users to confirm the delete operation, the success_url option must also be declared to specify where to take users after the delete operation is successful.

Next is the url definition that hooks up the DeleteView class-based view class. Because a DeleteView class-based view must be told which model record to delete, it relies on a url parameter to provide this query delimiter. In this case, notice the url contains the pk url parameter that gets passed to the class-based view -- just like it's done with a DetailView class-based view.

For example, if a request is made for the url /items/delete/1/, 1 is passed as the pk argument to the DeleteView class-based view, which in turn performs the query Item.objects.get(pk=1) to prepare the record for deletion.

Finally, the last part illustrates the item_confirm_delete.html template which is used for the user-facing deletion sequence. Notice this template contains a pseudo-form (i.e no form fields) with a question and submit button. The reason for this pseudo-form is
the item_confirm_delete.html template has a dual function.

n. DeleteView fields and methods

Like other class-based views, a DeleteView class-based view can also be created with custom fields and methods to override their default behaviors.

As it turns out, the DeleteView class inherits its behavior from many other Django class-based views, presented in the following list:
django.views.generic.detail.SingleObjectTemplateResponseMixin
django.views.generic.base.TemplateResponseMixin
django.views.generic.edit.BaseDeleteView
django.views.generic.edit.DeletionMixin
django.views.generic.detail.BaseDetailView
django.views.generic.detail.SingleObjectMixin
django.views.generic.base.View

o. Basic DeleteView optionsmodel and success_url fields & url with pk parameter

As you saw , the essential logic fulfilled by a DeleteView class-based view is to get a model record and delete it while presenting a confirmation page. Therefore the model field is one of the essential pieces to this type of class-based view and must always by provided. In addition, because a record is set to be deleted, it's also necessary to specify a success url through the success_url option to re-direct a user after the record is deleted.

In order to get a specific model record, a DeleteView class-based view must also define a url parameter which helps it select a model record to delete. By default, this url parameter must be named pk and its value is used to perform a query on the model value using the pk field, which is generally equivalent to a model's id field.

p. Class-based views with mixins

Although all class-based views that operate on models typically follow the same workflow to create, read, update and delete model records, it's fair to say that after seeing the previous class-based view sections, you'll often find yourself adjusting the default behavior of class-based views.

While it can be perfectly reasonable to adjust the methods and fields of a class-based view a couple of times, it can get tiresome if you need to do it over and over to obtain the same functionality across dozens or hundreds of class-based views.

Nowhere is this more evident than when you're forced to define a get() or post() method in a class-based view, to include some functionality that's not supported in a class-based view (e.g. CreateView, ListView, DetailView, UpdateView, DeleteView), which requires typing in a a long winded workflow which can turn out to be repetitive if done mulTip:le times. To cut down on repetitive customizations in the context of class-based views, you can use *mixins*.

For starters, you've already used mixins in the context of class-based views, even if you didn't realize it. If you looked closely at the class-based view classes that provide the behaviors to the model class-based views described in previous sections, you may have noticed many include the term *mixin* in their name (e.g. ModelFormMixin, FormMixin, SingleObjectMixin).

A software mixin is a construct that allows *inheritance-like* behavior between classes, noting the emphasis on *inheritance-like*. When you use class inheritance, a parent-child class relationship is often described with the 'is a' term (e.g. if a Drink class inherits its behavior from an Item class, a Drink *is an* Item). A mixin class on the other hand, allows a class to adopt the behaviors of a mixin class without the 'is a' behavior.

In other words, a mixin class is a way to add functionality to classes, with the mixin class serving as a re-usable component. For example, you can have a mixin Checkout class used by a Store class and OnlineStore class, which allows the functionality of the mixin class to be reused in any other class. Notice that for mixin classes, the inheritance 'is a' behavior doesn't apply, you can't say a Store *is a* Checkout or an OnlineStore *is a* Checkout, it's more of 'uses a' behavior. Therefore although mixin classes -- semantically speaking -- are used to inherit behaviors, technically speaking they don't use inheritance as it's commonly known in software engineering, hence the use of the term *inheritance-like*.

Example: Django class-based view with CreateView and mixin class

```
# views.py
from django.views.generic.edit import CreateView
from django.contrib.messages.views import SuccessMessageMixin
from .models import Item, ItemForm
```

```
class ItemCreation(SuccessMessageMixin,CreateView):
    model = Item
    form_class = ItemForm
    success_url = reverse_lazy('items:index')
    success_message = "Item %(name)s created successfully"
```

Self-Check Sheet 3: Create class-based view

1. What is the primary function of the CreateView class-based view in Django?

Answer:

- A. To display a detail view of a model instance
- B. To simplify the creation of new model records
- C. To handle form validation for editing model instances
- D. To provide a generic search interface for model data

2. Which of the following fields is essential for a CreateView class-based view?

Answer:

- A. success_url (specifies redirect after creation)
- B. get_context_data (customizes template context)
- C. form_class (defines the form used for data entry)
- D. template_name (specifies the template used for rendering)

3. How does the get_initial method in a CreateView class work?

Answer:

- A. It validates the data entered in the form.
- B. It sets default values for the form fields.
- C. It handles redirecting the user after successful creation.
- D. It retrieves existing data from the database.

4. What is the purpose of the form_valid method in a CreateView class?

Answer:

- A. It renders the form template with any errors.
- B. It performs actions after the form data is successfully validated.
- C. It initializes the form instance with default values.
- D. It checks if the user has submitted the form.

5. When would you use the get or post methods in a CreateView class-based view?

Answer:

- A. To define the success URL after creation.
- B. To customize the form template for data entry.
- **C.** To gain complete control over the GET and POST request workflows.
- D. To validate and save the form data to the database.

Answer Key 3: Create class-based view

1. What is the primary function of the CreateView class-based view in Django?

Answer: B. To simplify the creation of new model records

2. Which of the following fields is essential for a CreateView class-based view?

Answer: D. template_name (specifies the template used for rendering)

3. How does the get_initial method in a CreateView class work?

Answer: B. It sets default values for the form fields.

4. What is the purpose of the form_valid method in a CreateView class?

Answer: B. It performs actions after the form data is successfully validated.

5. When would you use the get or post methods in a CreateView class-based view?

Answer: C. To gain complete control over the GET and POST request workflows.

Job Sheet-3: Create a web view using a Django class-based view (CreateView) to handle form submission and creation of new model records.

UoC Cover

OU-ICT-WADP-03- L4-V1: Use Django Model

Working Procedure / Steps

1. Define the Model:
   - Create a Django model class representing the data you want to manage (e.g., Product, BlogPost).
   - Specify fields with appropriate data types (e.g., name - CharField, description - TextField).
2. **Create the Form:**
   - Create a Django form class that inherits from ModelForm.
   - Specify the model and define fields to be included in the form.
   - Add validation rules and widgets for user input if necessary.
3. **Create the Class-Based View:**
   - Define a class inheriting from CreateView.
   - Set the model attribute to the model class.
   - Set the form_class attribute to the form class.
   - Define the success_url attribute to specify the URL to redirect to after successful creation.
   - Optionally, override methods for:
     - get_form_kwargs: Customize form instance arguments before rendering.
     - form_valid: Process valid form data and save the model instance.
4. **Integrate the View into your URL Configuration:**
   - In your project's urls.py, map a URL pattern to the view class.
5. **Create a Template:**
   - Create an HTML template for the form (e.g., product_form.html).
   - Use Django template tags to render the form and handle user input.

Specification Sheet-3: Create a web view using a Django class-based view (CreateView) to handle form submission and creation of new model records.

Technical Requirements

- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later
- **Text Editor/IDE:** Visual Studio Code, PyCharm, Sublime Text (or any preferred code editor)

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Internet Connection:** A reliable internet connection for downloading and installing software, accessing online resources, and deploying the application (if applicable).
- **Development Environment:** A virtual environment is recommended to isolate project dependencies.
- **Web Server (Optional):** If deploying your application, you'll need a web server like Apache or Nginx.

Reference

1. **Flask Web Development: Developing Python Web Applications**
   Author: Miguel Grinberg
   Year: 2018 (2nd Edition)
2. **Django for Beginners: Build Awesome Websites with Python**
   Author: William S. Vincent
   Year: 2021 (2nd Edition)
3. **Two Scoops of Django: Best Practices for the Django Web Framework**
   Authors: Daniel J. Greenfeld and Audrey Roy Greenfeld
   Year: 2020 (3rd Edition)

Review of Competency

Below is yourself assessment rating for module "Use Django Model"

Assessment of performance Criteria	Yes	No
Model class is defined with proper model data type and relationships	☐	☐
Model migration is performed	☐	☐
CRUD single records in Django Models are performed using shell and Admin	☐	☐
CRUD multiple records in Django Models are performed using shell and admin	☐	☐
3. CRUD relationship records across Django models are performed	☐	☐
Model queries are performed	☐	☐
Custom and multiple model managers are created	☐	☐
Model records with class-based view are created	☐	☐
CRUD with class-based view is performed	☐	☐
Mixin is applied with class-based view	☐	☐

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of 'Use Django Model' (Occupation: Web Application Development with Python , Level-4) for National Skills Certificate is developed by NSDA with the assistance of SAMAHAR Consultants Ltd.in the month of June, 2024 under the contract number of package SD-9C dated 15th January 2024.

SL No.	Name and Address	Designation	Contact Number
1	Khan Mohammad Mahmud Hasan	Writer	Cell: 01714087897 Email: kmmhasan@gmail.com
2	A K M Mashuqur Rahman Mazumder	Editor	Cell: 01676323576 Email : mashuq.odelltech@odell.com.bd
3	Khan Mohammad Mahmud Hasan	Co-Ordinator	Cell: 01714087897 Email: kmmhasan@gmail.com
4	Md. Saif Uddin	Reviewer	Cell:01723004419 Email: engrbd.saif@gmail.com