VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANA SANGAMA, BELGAVI-590018, KARNATAKA

FULLSTACK DEVELOPMENT

(AS PER CBCS SCHEME 2021)

SUB CODE: 21CS62

PREPARED BY:

DEEPA B (ASST.PROF DEPT OF CSE, KNSIT)

INDHUMATHI R (ASST.PROF DEPT OF DS (CSE), KNSIT)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

K.N.S INSTITUTE OF TECHNOLOGY

HEGDE-NAGAR, KOGILU ROAD,

THIRUMENAHALLI, YELAHANKA,

BANGALORE-560064

Module-2: Django Templates and Models

Program 1

Develop a Django app that displays current date and time in server views.py

views.py

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.

def dis_datetime(request):
    import datetime
    x = datetime.datetime.today()
    return HttpResponse(x)
```

urls.py

```
from django.contrib import admin
from django.urls import path, include

from datetimeapp import views as vd
from initial import views as vi

urlpatterns = [
    #path('',include('initial.urls')),
    path('admin/', admin.site.urls),
    path('wl/',vi.home),

    path('dt/',vd.dis_datetime), #an url to open datatime views.py
]
```

Program 2

Develop a Django app that displays date and time four hours ahead and four hours before as an offsetof current date and time in server.

views.py

```
from django.shortcuts import render from
django.http import HttpResponse
from datetime import datetime, timedelta
# Create your views here.
def datetime_display(request):
    # Get current date and time
    current_datetime = datetime.now()
    # Calculate offsets
    four_hours_before = current_datetime - timedelta(hours=4)
    four_hours_ahead = current_datetime + timedelta(hours=4)

    # Prepare the response content
    response_content = (

        f"Current Date and Time: {current_datetime}\n"
        f"Four Hours Before: {four_hours_before}\n"
        f"Four Hours Ahead: {four_hours_ahead}\n"

    )
    # Return the response
    return HttpResponse(response_content, content_type='text/plain')
```

urls.py

```
from django.contrib import admin
from django.urls import path, include
from datetimeapp import views as vd
from initial import views as vi
urlpatterns = [
    #path('',include('initial.urls')),
    #path('admin/', admin.site.urls),
    #path('wl/',vi.home),
    #path('dt/',vd.dis_datetime),
    path('dt2/',vd.datetime_display),
]
```

Template System Basics

In Django, the template system is a powerful tool for generating dynamic HTML content.

1.Creating Templates:

Templates are HTML files with Django template language (DTL) syntax. You store them in a directory named templates within your app directory. Django will automatically find templates in this directory.

2.Template Syntax:

Django's template language uses special syntax enclosed within {% %} and {{ }} delimiters.

- {% ... %}: Used for template tags, which control the logic and flow of the template (e.g., loops, conditionals).
- {{ ... }}: Used for template variables, which insert dynamic content into the template.
- {# ... #}: Used for comments in templates.

3. Rendering Templates:

Views in Django render templates using the render() function. This function loads a template, passes data to it (optional), and returns an HTTP response containing the rendered HTML.

```
from django.shortcuts import render
def my_view(request):
    context = {'name': 'Alice'}
    return render(request, 'myapp/my_template.html', context)
```

4.Template Inheritance:

Django supports template inheritance, allowing you to create a base template with common elements and extend it in other templates.

```
<!-- base.html -->
<html>
<head>
    <title>{% block title %}My Site{% endblock %}</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>

<!-- child.html -->
{% extends 'base.html' %}
{% block title %}My Custom Title{% endblock %}
{% block content %}
<h1>Hello, {{ name }}!</h1>
{% endblock %}
```

5.Template Tags and Filters:

Django provides built-in template tags and filters for performing common operations in templates, such as looping over lists, formatting dates, and filtering querysets.

The Django template system is flexible and easy to use, allowing you to create dynamic and reusable HTML content for your web applications.

Using Django Template System

Using the Django template system involves several steps:

1. Create Templates:

Create HTML files with DTL (Django Template Language) syntax. Place them in the templates directory within your app directory. For example, create a file named index.html:

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}My Site{% endblock %}</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

2. Render Templates in Views:

In your views (in views.py), import the render function from django.shortcuts and use it to render the template:

```
# views.py
from django.shortcuts import render
def index(request):
    context = {'name': 'Alice'}
    return render(request, 'index.html', context)
```

3. Use Template Variables:

Pass data to the template by defining a context dictionary and passing it as the third argument to the render() function. Access the data in the template using template variables enclosed in {{ }}:

```
# views.py
def index(request):
    context = {'name': 'Alice'}
    return render(request, 'index.html', context)
```

In the template:

```
<!-- index.html -->
<h1>Hello, {{ name }}!</h1>
```

4.Template Inheritance:

Use template inheritance to create a base template with common elements and extend it in other templates. Define blocks in the base template using {% block %} tags, and override them in child templates:

```
<!-- base.html -->
<html>
```

```
<head>
   <title>{% block title %}My Site{% endblock %}</title>
</head>
<body>
   {% block content %}{% endblock %}
</body>
</html>

<!-- child.html -->
{% extends 'base.html' %}

{% block title %}My Custom Title{% endblock %}

{% block content %}
<h1>Hello, {{ name }}!</h1>
{% endblock %}
```

This is a basic overview of using the Django template system. It allows you to create dynamic HTML content for your web applications by combining HTML with Django's template language syntax

TEMPLATES

Django provides a convenient way to generate dynamic HTML pages by using its template system.

A template consists of static parts of the desired HTML output as well as some special syntax describing howdynamic content will be inserted.

Django's template language (DTL) is designed to strike a balance between power and ease. It's designedto feel comfortable to those used to working with HTML

It is possible to added additional styles to my webpage like heading, image, and so on. For example

```
def home(request):
    return HttpResponse("<h1>Welcome to my homepage</h1>")
```

EXAMPLE

create a new folder "Webpages" and create a new HTML file.

Index1.html

```
<h1> Using Django Templates </h1>
<p> Django's template language is designed to strike a balance between power and ease.
It's designed to feel comfortable to those used to working with HTML. </p>
```

Now open settings.py file and go to templates and change

```
'DIRS': [os.path.join(BASE_DIR,'Webpages')],
```

Next step to call the HTML page in views.py in main app

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
def home(request):
    return render(request,'index1.html') #render is used to combine static
```

and dynmaic webpage, index1.htmlis a html file name

urls.py

```
from django.urls import path

from . import views

urlpatterns = [path('',views.home, name="home")]
```

Render function - This function takes three parameters

Request – The initial request

The path to the template- This is the path relative to the TEMPLATE_DIRS option in the project settings.py variables.

Dictionary of parameters – A dictionary that contains all variables needed in the template. This variable can be created or we can use locals() to pass all local variable declared in the view.

Filters

They help you modify variables at display time. Filters structure looks like the following:

{{var|filters}}.

Example

{{string|lower}} − Converts the string to lowercase.

{{string|escape|linebreaks}} − Escapes string contents, then converts line breaks to tags.

{{string|truncatewords:80}} − This filter will truncate the string, so we can see only first 80 words

views.py

```
from django.shortcuts import render
from datetime import datetime
def filters_ex(request):
    current_date = datetime.now().date()
    current_time = datetime.now().time()
    username = "django"
    email = "jdjango@gmail.com"
    context = {
        'current_date': current_date,
        'current_time': current_time,
        'username': username,
        'email': email,
    }
    return render(request, 'filter_ex.html', context)
```

filter_ex.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Filters Template</title>
```

```
</head>
<body>
   <h1> Filters example</h1>
   <p>Today's date is: {{ current_date|date:"Y-m-d" }}</p>
   <p>Current time is: {{ current_time|time:"H:i:s" }}</p>
   <p>Your username is: {{ username|upper }}</p>
   <p>Your email is: {{ email|lower }}</p>
</body>
</html>
```

urls.py

```
from django.urls import path
from . import views
urlpatterns = [
   path('', views.fliters_ex, name='fliters_ex'),

]
```

urls.py - Main project

```
from django.contrib import admin
from django.urls import path, include
```

```
urlpatterns = [
    path('flt/', include('filters.urls')),
]
```

Tags

Tags lets you perform the following operations: if condition, for loop, template inheritance and more.

For - Loop over each item in an array.

if, **elif**, and **else -** Evaluates a variable, and if that variable is "true" the contents of the block are displayed.

Develop a simple Django app that displays an unordered list of fruits and ordered list of selected students for an event

Student_tag.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Event Details</title>
</head>
<body>
  <h1>Fruits</h1>
  <ul>
```

```
        {% for fruit in fruits %}
            <li>{{ fruit }}</li>
        {% endfor %}
    </ul>
    <h1>Selected Students</h1>
    <ol>
        {% for student in students %}
            <li>{% if student.score > 80 %}{{ student.name }}{% endif %}</li>
        {%endfor%}
    </ol>
</body>
</html>
```

Athlete_list.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Athletes List</title>
</head>
<body>
  <h1>List of Athletes</h1>
  <ul>
```

```
    {% for athlete in athlete_list %}
    <li>{{ athlete }}</li>
    {% endfor %}
  </ul>
  {% if athlete_list %}
    <p>Number of athletes: {{ athlete_list|length }}</p>
  {% elif athlete_in_locker_room_list %}
    <p>Athletes should be out of the locker room soon!</p>
  {% else %}
    <p>No athletes.</p>
  {% endif %}
</body>
</html>
```

views.py

```
from django.shortcuts import render
def event_details(request):
  fruits = ['Apple', 'Banana', 'Orange', 'Mango']
```

```
    # Sample student
    data with scores
    students = [
        {'name': 'Harish', 'score': 85},
        {'name': 'Briana', 'score': 70},
        {'name': 'John', 'score': 95},
        {'name': 'Arul', 'score': 60},
    ]
    return render(request, 'student_tag.html', {'fruits': fruits, 'students': students})
def athlete_l(request):
    context = {
        'athlete_list': ['Rajwinder Kaur', 'Chitra Soman', 'Manjeet Kaur '],
        'athlete_in_locker_room_list': []
    }
    return render(request, 'athlete_list.html', context)
```

urls.py

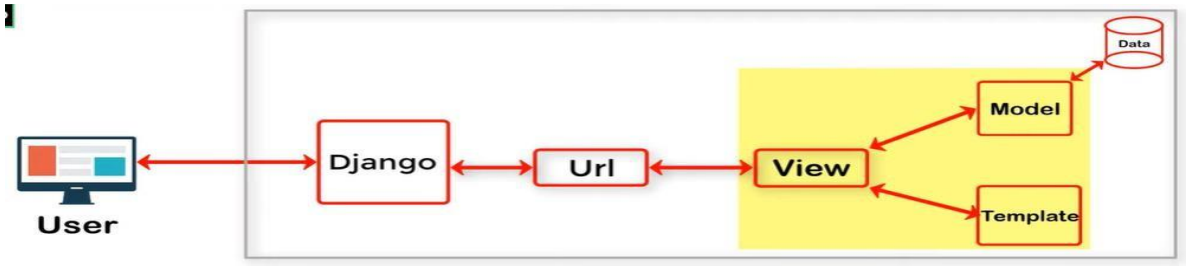
```
from django.urls import path
from . import views

urlpatterns = [
```

```
    path('events/', views.event_details,
    name='event_details'),path('athlete/',
    views.athlete_l, name='athlete_l'),
]
```

urls.py- Main Project

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('tags/', include('Lab_temp_1.urls')),
]
```

MVT Development Pattern



The MVT (Model View Template) is a software design pattern. It is a collection of three important components: Model View and Template.

MVT Structure has the following three parts –

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

View: The View is the user interface — what you see in your browser when you render a website. It isrepresented by HTML/CSS/Javascript and Jinja files.

Template: A template consists of static parts of the desired HTML output as well as some special syntaxdescribing how dynamic content will be inserted.

Working:

1. Request Handling:

• When a user interacts with the application, such as submitting a form or clicking a link, their action triggers an HTTP request to the server.

• The framework's URL dispatcher maps the incoming request to a specific view function or class based on the URL pattern defined in the application's URL configuration.

2. View Processing:

• Upon receiving the request, the corresponding view function or class is invoked. This view function or class is responsible for processing the request and generating an appropriate response.

• The view interacts with the Model layer to retrieve or modify data as necessary. This may involve querying the database, performing calculations, or invoking other business logic.

• Once the required data has been obtained, the view prepares it for presentation and passes it to the Template layer for rendering.

3. Template Rendering:

• The Template layer receives the data from the view and uses it to render the HTML markup for the response.

- Templates typically contain placeholders (template tags) that are replaced with dynamic content generated by the view. These placeholders can include variables, loops, conditionals, and other logic provided by the template language.
- After rendering the HTML content, the template engine combines it with any static assets (CSS, JavaScript, images, etc.) to form the complete HTTP response.

4. Response Delivery:

- Finally, the fully-rendered HTTP response is sent back to the client's browser, where it is displayed to the user.
- If the response includes any client-side interactivity (e.g., JavaScript functionality), the browser may execute scripts to handle user interactions and update the DOM dynamically.

Throughout this process, the Model layer handles data management and business logic, the View layer manages request processing and response generation, and the Template layer controls the presentation and layout of the user interface. By separating these concerns, the MVT pattern promotes code organization, maintainability, and scalability in web applications.

Template Inheritance

Template inheritance allows you to build a base "skeleton" template that contains all the common elementsof your site and defines **blocks** that child templates can override.

EXAMPLE

views.py

```
from django.shortcuts import render

def news_article(request, article_id):
```

```
    # Add logic to fetch a specific news article by its ID
  article = {'title': 'News Article', 'content': 'Content of the article with ID
  {}'.format(article_id)}
    return render(request, 'inh_news_article.html', {'article': article})

  def news(request):

    # Add logic to fetch and pass news articles to the template

    context = {

      'articles': [

        {'id': 1, 'title': 'News Article 1', 'content': 'Content of Article 1'},

        {'id': 2, 'title': 'News Article 2', 'content': 'Content of Article 2'},

      ]

    }

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

urls.py

```
  from django.urls import path

  from . import views

  urlpatterns = [

    path('news/', views.news, name='news'),

    path('news/<int:article_id>/', views.news_article, name='news_article'),

  ]
```

Inh_news.html

```
{% extends "inh_base.html" %}
{% block title %}News - {{ block.super }}{% endblock %}

{% block sidebar %}
  {{ block.super }}
  <ul>
    <li><a href="/news/">Latest News</a></li>
    <li><a href="/news/archive/">News Archive</a></li>
  </ul>
{% endblock %}

{% block content %}
{% for article in articles %}
  <h2>{{ article.title }}</h2>
  <p>{{ article.content }}</p>
{% endfor %}
{% endblock %}
```

Inh_news_article.html

```
{% extends "inh_base.html" %}
{% block title %}{{ article.title }} - {{ block.super }}{% endblock %}
{% block content %}
  <h2>{{ article.title }}</h2>
```

```
    <p>{{ article.content }}</p>

{% endblock %}
```

Inh_base.html

```
<!DOCTYPE html>

<html lang="en">

<head>

   <title>{% block title %}My amazing site{% endblock %}</title>

</head>

<body>

   <div id="sidebar">

      {% block sidebar %}

      <ul>

         <li><a href="/">Home</a></li>

         <li><a href="/blog/">Blog</a></li>

      </ul>

      {% endblock %}

   </div>

   <div id="content">
```

```
        {% block content %}{% endblock %}
      </div>
    </body>
    </html>
```

CONFIGURING THE DATABASE:

Configuring a database in Django involves setting up your project to connect to a database like MySQL. This allows Django to store and retrieve data for your application. You'll need to install the database software, create a database, and then configure Django to use it.

Real-World Example: Setting Up MySQL with Django

1. Download and Install MySQL:

-Install MySQL

- Download and install MySQL Workbench and MySQL Server.

2. Create a Database in MySQL:

- Open MySQL Command Line Client or MySQL Workbench.

- Log in with your MySQL root password.

- Create a new database by running the following command

"CREATE DATABASE mydatabase;"

3. Install MySQL Client Library in Django:

- Install the MySQL client library for Python using pip:

"*pip install mysqlclient*"

4. Configure Django to Use MySQL:

- Open your Django project's `settings.py` file.

- Modify the `DATABASES` setting to use MySQL:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'mydatabase',
        'USER': 'root',
        'PASSWORD': 'your_password',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

5. Run Migrations:

- Apply Django's migrations to set up your database tables

" python manage.py migrate"

DEFINING AND IMPLEMENTING MODELS:

Models in Django are Python classes that represent database tables. Each attribute of the class represents a field in the database table. Models allow you to define the structure of your data, interact with the database, and perform queries.

Example: Blog Application

Let's create a simple blog application where we have posts with titles, content, and timestamps.

1. Define Models:

- Models are defined in the `models.py` file of your Django app.

Example:

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


class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    date_posted = models.DateTimeField(auto_now_add=True)


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

>>Explanation:

- `Post` class: Represents a blog post.

- `title`: A character field for the post title, limited to 100 characters.

- `content`: A text field for the post content.

- `date_posted`: A date-time field that automatically sets the current date and time when a post is created.

- `__str__`: A method that returns the title of the post, which is helpful when dealing with the admin interface or shell.

2. Apply Migrations:

- Migrations are Django's way of propagating changes you make to your models into your database schema.

python manage.py makemigrations

python manage.py migrate

>>Explanation:

- `makemigrations`: Creates migration files based on the changes in `models.py`.

- `migrate`: Applies the migrations to the database, creating the necessary tables.

3. Use the Models:

- You can now interact with the `Post` model in your views, shell, or admin interface.

Example:

views.py

```
from django.shortcuts import render
from .models import Post
def home(request):
    posts = Post.objects.all()
    return render(request, 'blog/home.html', {'posts': posts})
```

>>Explanation:

- `home` view: Fetches all posts from the database and passes them to the `home.html` template.

4. Admin Interface:

- Register your models with the Django admin to manage them via the admin panel.

Example:

```
# admin.py
from django.contrib import admin
from .models import Post
admin.site.register(Post)
```

>>Explanation:

- Registers the `Post` model with the admin site, so you can add, edit, and delete posts via the admin interface.

Summary:

1. Define Models: Create model classes in `models.py` that represent your database tables.
2. Apply Migrations: Run migration commands to create the corresponding tables in the database.
3. Use Models: Interact with your models in views, templates, and the admin interface.
4. Admin Interface: Register models in `admin.py` to manage them through Django's built-in admin panel.

BASIC DATA ACCESS:

In Django, you can perform basic data access operations using its built-in ORM (Object-Relational Mapping) system. Here's a basic overview:

1. Creating Objects: To create a new object, you define a model class in models.py, then instantiate and save it using the ORM:

```
from myapp.models import MyModel

new_object = MyModel(field1=value1, field2=value2)

new_object.save()
```

2. Reading Objects: To retrieve objects from the database, you can use the objects manager provided by Django:

```
from myapp.models import MyModel

all_objects = MyModel.objects.all()

specific_object = MyModel.objects.get(id=1)
```

3. Updating Objects: To update an existing object, you retrieve it, modify its attributes, and call save():

```
specific_object = MyModel.objects.get(id=1)

specific_object.field1 = new_value

specific_object.save()
```

4. Deleting Objects: To delete an object, you retrieve it and call the delete() method:

```
specific_object = MyModel.objects.get(id=1)

specific_object.delete()
```

These are the basic operations. Django's ORM provides much more functionality for complex data access patterns, including filtering, querying related objects, and aggregations.

ADDING MODEL STRING REPRESENTATION:

It is helpful for displaying meaningful information about objects in the admin interface and debugging.

1. Importing Django's Model:

from django.db import models

This line imports the models module from Django's database abstraction layer.

2. Defining a Model:

class MyModel(models.Model):

This line defines a Django model named MyModel. The model inherits from models. Model, indicating that it's a Django model.

3. Adding Fields:

name = models.CharField(max_length=100)

age = models.IntegerField()

These lines define two fields for the MyModel class. name is a character field (CharField) with a maximum length of 100 characters, and age is an integer field (IntegerField).

4. Defining the _str_ Method:

```
def _str_(self):
    return f"{self.name} (Age: {self.age})"
```

The _str_ method is a special method in Python classes that defines how the object should be represented as a string. In this case, it returns a string that includes the name and age attributes of the object.

5. Customizing the String Representation:

In the *_str_* method, self.name and *self.age* refer to the name and age attributes of the MyModel object. By including them in the returned string, we create a custom string representation for each MyModel object.

By defining the *_str_* method in this way, whenever you try to display a MyModel object as a string (for example, in the Django admin interface), it will show the name and age of the object, making it easier to identify and work with the objects.

INSERTING/UPDATING, SELECTING AND DELETING OBJECTS:

1. **Inserting Data**:

To insert new data into your database using Django models, follow these steps:

- Create a Model Instance: First, create an instance of your model class. Populate the instance with the desired data.
- Save the Instance: After creating the instance, call the save() method on it to save the data to the database. Django will automatically generate an SQL INSERT statement to insert the data into the appropriate table.

Example:

```
from myapp.models import YourModel


# Create a new instance
instance = YourModel(name='Example Name')
```

```
# Save the instance to the database
instance.save()
```

This will insert a new row into the database table associated with the YourModel class.

2. Updating Data:

To update existing data in your database using Django models, follow these steps:

•Retrieve the Instance: First, retrieve the instance you want to update from the database. You can use methods like get(), filter(), or first() to retrieve the instance based on certain criteria.

•Update the Instance: Once you have the instance, modify its attributes as needed to reflect the changes you want to make.

•Save the Instance: Finally, call the save() method on the instance to save the changes to the database. Django will generate an SQL UPDATE statement to update the corresponding row in the table.

Example:

```
# Retrieve an instance from the database
instance = YourModel.objects.get(id=1)

# Update its attributes
instance.name = 'New Name'
```

```
# Save the changes to the database
instance.save()
```

This will update the name attribute of the instance with the new value ‘New Name’ in the database.

2.1. Bulk Updating: If you need to update multiple instances at once, you can use the update() method on a queryset. This will generate a single SQL UPDATE statement to update all matching rows in the table.

Example:

```
YourModel.objects.filter(some_filter_condition).update(name='New Name')
```

These are the basic steps for inserting and updating data in Django models. It’s essential to understand these concepts to effectively work with databases in Django applications.

3. Selecting Objects:

Django provides a powerful ORM (Object-Relational Mapping) that allows you to interact with your database using Python objects. You can retrieve objects from the database using querysets, which are a representation of a database query.

- Retrieve all objects: Use the all() method on the model’s manager to retrieve all objects of a particular model.

Example:

```
from myapp.models import YourModel

all_objects = YourModel.objects.all()
```

- Filtering objects: Use the filter() method to retrieve objects that match certain criteria.

Example:

```
filtered_objects = YourModel.objects.filter(some_field=some_value)
```

- Retrieve a single object: Use the get() method to retrieve a single object based on its primary key or other unique field.

Example:

```
single_object = YourModel.objects.get(id=1)
```

You can also use other queryset methods like exclude(), order_by(), distinct(), etc., to further refine your querysets.

4. Deleting Objects:

Once you have selected the objects you want to delete, you can use the delete() method to remove them from the database.

•Delete based on criteria: Use the delete() method directly on a queryset to delete all objects that match certain criteria.

Example:

```
YourModel.objects.filter(some_field=some_value).delete()
```

- Delete a single object: Use the delete() method on an individual object to remove it from the database.

Example:

single_object.delete()

When you call delete(), Django generates and executes the necessary SQL DELETE statement to remove the selected objects from the database.

Remember to handle deletion with caution, especially when using filtering criteria, to avoid unintentionally removing important data from your database. Django's ORM provides safeguards to prevent accidental data loss, such as requiring explicit confirmation for bulk deletions.

SCHEMA EVOLUTION:

Schema evolution refers to the process of modifying the structure of a database schema over time, typically to accommodate changes in application requirements or to optimize performance. In the context of Django or any other ORM (Object-Relational Mapping) framework, schema evolution involves making changes to the models that define the structure of the database tables.

Here's how schema evolution works in Django:

1. **Model Changes:**

When you need to modify the schema of your database, you make corresponding changes to your Django models. This could involve adding new fields, removing existing fields, modifying field types, adding new models, etc.

Example:

```
# Before modification

class MyModel(models.Model):

    name = models.CharField(max_length=100)

    age = models.IntegerField()

# After modification (adding a new field)

class MyModel(models.Model):

    name = models.CharField(max_length=100)

    age = models.IntegerField()

    email = models.EmailField()
```

2. Migration Files:

After modifying your models, you need to generate migration files using the make migrations command. These migration files contain Python code that represents the changes you've made to your models.

Example:

```
python manage.py make migrations
```

This command examines the current state of your models and creates migration files in the migrations directory of each app in your Django project.

3. Applying Migrations:

Once you've generated the migration files, you need to apply them to your database using the migrate command. This command executes the migration files and updates the schema of your database accordingly.

Example:

python manage.py migrate

Django maintains a table in the database to track which migrations have been applied. It only applies migrations that have not been applied previously.

4. Database Schema Update:

After applying migrations, Django updates the schema of your database to reflect the changes you made to your models. This could involve creating new tables, adding or removing columns, modifying column types, etc.

Django's migration framework handles various types of schema changes, including adding and removing fields, changing field types, creating and deleting tables, and even data migrations (migrating data between different schema versions).

By following this process, you can evolve the schema of your database over time in a controlled and systematic manner, ensuring that your database structure remains in sync with your application's requirements. Django's migration framework automates much of the process, making it easier to manage schema changes as your application evolves.