Django Framework:

Introduction to Django

Django is a powerful and popular web framework written in Python. It follows the Model-View-Template (MVT) architectural pattern, which is similar to the Model-View-Controller (MVC) pattern. MVT helps in building scalable, maintainable, and efficient web applications.

Let's break down the MVT concepts in Django using real-world examples:

Model:

The Model represents the data structure of your application. It defines the database schema and handles data manipulation. For example, if you're building a blog application, your models could include classes like User, Post, and Comment.

Here's an example of a Django model:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    pub_date = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey(User,
on_delete=models.CASCADE)
```

In this example, the Post model has fields for the title, content, publication date, and an author (which is a foreign key to the User model).

View:

The View handles the logic behind rendering the data to the user. It interacts with the model to retrieve data and performs any necessary processing. It then passes the processed data to the template for rendering.

Let's continue with our blog example and define a view for displaying a list of blog posts:

```
from django.shortcuts import render
from .models import Post

def post_list(request):
    posts = Post.objects.all()
    return render(request, 'blog/post_list.html', {'posts': posts})
```


In this example, the post_list view retrieves all the Post objects from the database and passes them to the blog/post_list.html template.

Template:

The Template is responsible for rendering the data received from the view. It contains the HTML structure and presentation logic. In the blog example, we can create a template that displays the list of posts:

```
<!-- blog/post_list.html -->

<h1>Blog Posts</h1>
<ul>
{% for post in posts %}
    <li>{{ post.title }}</li>
{% endfor %}
</ul>
```

In this example, the template loops through the posts variable passed from the view and displays each post's title in an unordered list.

Overall, Django's MVT pattern allows you to separate concerns and maintain a clean structure for your web application. The model handles the data, the view handles the logic, and the template handles the presentation. This separation enhances code reusability, testability, and scalability.
Remember that this is just a brief introduction to the MVT pattern in Django, and there are many more features and concepts to explore within the framework.

Django Installation:

Here are the detailed steps to install Django:

Step 1: Prerequisites
Before installing Django, ensure that you have Python installed on your system. Django requires Python 3.6 or higher. You can check your Python version by running the following command in your terminal or command prompt:
python --version

Step 2: Create a Virtual Environment (optional but recommended)
It's recommended to create a virtual environment for your Django project. A virtual environment keeps your project dependencies isolated. To create a virtual environment, run the following command:
python -m venv myenv
Replace myenv with the desired name of your virtual environment.

Step 3: Activate the Virtual Environment

Activate the virtual environment using the appropriate command for your operating system:

- For Windows (Command Prompt):
- Copy code
- myenv\Scripts\activate
- For macOS/Linux:
- bash
- Copy code

- source myenv/bin/activate

Step 4: Install Django
Once you have activated the virtual environment, use pip (Python package installer) to install Django. Run the following command:
pip install django
This will download and install the latest stable version of Django.

Step 5: Verify the Installation
To ensure that Django is installed correctly, you can check its version using the following command:
python -m django --version

If Django is installed properly, it will display the version number.
Congratulations! You have successfully installed Django on your system.
Now you can start creating Django projects and applications.
Note: If you're using an integrated development environment (IDE) like PyCharm, it may provide a graphical interface for creating virtual environments and installing Django. You can refer to your IDE's documentation for specific instructions.
Remember to activate the virtual environment every time you work on your Django project to ensure you're using the correct Python environment.

Dynamic Web Pages

In Django, dynamic web pages are created using templates and views. Templates define the structure and presentation of the web page, while views handle the logic and data processing. Let's dive into the details with an example:

Step 1: Create a Django Project and App
First, create a new Django project by running the following command:
```
django-admin startproject myproject
```
Then, navigate into the project directory:
```
cd myproject
```
Next, create a Django app within the project:
```
python manage.py startapp myapp
```

Step 2: Define a View

In the myapp directory, open the views.py file and define a simple view.
For example, let's create a view that displays a greeting message:

```
# myapp/views.py

from django.shortcuts import render

def greeting(request):
    name = "John Doe"
    return render(request, 'myapp/greeting.html', {'name': name})
```

This view sets the name variable to "John Doe" and renders the greeting.html template.

Step 3: Create a Template

In the myapp directory, create a new directory called templates. Inside the templates directory, create another directory called myapp. This is where we'll store our templates specific to the myapp app.
Create a new file called greeting.html inside the myapp directory:

```
<!-- myapp/greeting.html -->

<!DOCTYPE html>
<html>
<head>
    <title>Greeting</title>
</head>
<body>
    <h1>Welcome, {{ name }}!</h1>
</body>
</html>
```

This template contains a simple HTML structure with a heading that displays the name variable using Django's template syntax {{ name }}.

Step 4: Define URLs and URL Configuration

In the myproject directory, open the urls.py file and define a URL pattern for the greeting view:

```
# myproject/urls.py

from django.urls import path
from myapp.views import greeting

urlpatterns = [
    path('greeting/', greeting, name='greeting'),
]
```

This maps the URL /greeting/ to the greeting view we defined earlier.

Step 5: Run the Server
Start the Django development server by running the following
`python manage.py runserver`

Step 6: Access the Web Page

Open your web browser and visit **http://localhost:8000/greeting/**. You should see the greeting message "Welcome, John Doe!" displayed on the page.

That's it! You have created a dynamic web page in Django. The view retrieves data (the name variable) and passes it to the template, which renders the web page with the dynamic content.

You can extend this example to handle more complex data, use loops, conditions, and incorporate CSS styles within your templates. Django's template language provides a powerful set of tools for building dynamic web pages.

Template System in Django

The template system in Django is a powerful tool for building dynamic web pages. It allows you to separate the HTML structure from the Python code and provides a convenient way to display data, loop through lists, perform conditionals, and more.
Let's explore the Django template system in more detail with a program example:

Step 1: Create a Django App
First, create a Django app within your project using the following
```
python manage.py startapp myapp
```

Step 2: Define a View

In the myapp/views.py file, define a view that retrieves data and passes it to the template:

```
# myapp/views.py

from django.shortcuts import render

def my_view(request):
    name = "John Doe"
    fruits = ['Apple', 'Banana', 'Orange']
    return render(request, 'myapp/my_template.html', {'name': name, 'fruits': fruits})
```

In this example, the view sets the name variable to "John Doe" and the fruits variable to a list of fruits. It then renders the **my_template.html** template, passing the name and fruits variables.

Step 3: Create a Template

In the myapp/templates directory, create a new file called my_template.html:

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

<!DOCTYPE html>
<html>
<head>
    <title>My Template</title>
</head>
<body>
    <h1>Welcome, {{ name }}!</h1>

    <ul>
    {% for fruit in fruits %}
        <li>{{ fruit }}</li>
    {% endfor %}
    </ul>
</body>
</html>
```


This template displays a heading that uses the name variable, and it loops through the fruits list to display each fruit in an unordered list.

Step 4: Define URLs and URL Configuration

In the project's main urls.py file, define a URL pattern for the my_view:

```
# myproject/urls.py

from django.urls import path
from myapp.views import my_view

urlpatterns = [
    path('myview/', my_view, name='my_view'),
]
```

This maps the URL /myview/ to the my_view function in myapp/views.py.

Step 5: Run the Server

Start the Django development server by running the following command:

```
python manage.py runserver
```

Step 6: Access the Web Page

Open your web browser and visit **http://localhost:8000/myview/**. You should see the rendered web page with the dynamic content: "Welcome, John Doe!" and a list of fruits.

Django's template system provides a variety of features, including filters for modifying data, template inheritance for creating reusable templates, and tags for more complex logic. You can also include static files like **CSS and JavaScript** within your templates.

By leveraging the Django template system, you can build flexible and dynamic web pages that separate the presentation logic from the backend code, making your codebase more maintainable and scalable.

The Administration Site of Django

The administration site in Django provides an interface for managing and interacting with the data stored in your application's database. It comes with built-in functionality for creating, updating, and deleting data records.
Let's explore the Django administration site in detail with program examples, including how to customize important admin features.

Step 1: Create a Django Superuser
Before accessing the administration site, you need to create a superuser account. Run the following command and follow the prompts:

```
python manage.py createsuperuser
```

Step 2: Register Models for Administration

To make your models accessible in the administration site, you need to register them. Open the myapp/admin.py file and define the admin configuration:

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

In this example, we register the Post model to be managed through the administration site.

Step 3: Customize the Administration Site
Django's administration site can be customized to fit your application's specific needs. Here are a few important features you can customize:

3.1. Customize Displayed Fields

By default, all fields of a registered model are displayed in the admin site. You can specify the fields you want to display by creating a custom admin class:

```
# myapp/admin.py

from django.contrib import admin
from myapp.models import Post

class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'pub_date')

admin.site.register(Post, PostAdmin)
```

In this example, we define a PostAdmin class and specify the list_display attribute to only display the title and pub_date fields.

3.2. Filter and Search

You can add filters and search functionality to the admin site to make it easier to navigate and find data. Add the list_filter and search_fields attributes to the custom admin class:

```
# myapp/admin.py

from django.contrib import admin
from myapp.models import Post

class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'pub_date')
```

```
    list_filter = ('pub_date',)
    search_fields = ('title',)

admin.site.register(Post, PostAdmin)
```

In this example, we enable filtering by the pub_date field and searching by the title field.

3.3. Customize Forms

You can customize the form used to create and update records in the administration site by creating a custom admin form:

```
# myapp/admin.py

from django.contrib import admin
from myapp.models import Post
from django import forms

class PostAdminForm(forms.ModelForm):
    # Custom form fields and behavior here

class PostAdmin(admin.ModelAdmin):
    form = PostAdminForm

admin.site.register(Post, PostAdmin)
```

In this example, we define a PostAdminForm class and specify it as the form to be used for the Post model.
Step 4: Run the Server

Start the Django development server by running the following command:

```
python manage.py runserver
```

Step 5: Access the Administration Site

Open your web browser and visit **http://localhost:8000/admin/**. Log in using the superuser credentials created earlier.
You will see the administration site with the registered models displayed. You can click on the registered models to manage and interact with the data stored in the database. The customized features, such as displayed fields, filters, and search functionality, will be available.

By customizing the administration site, you can tailor it to match your application's requirements, making it more user-friendly and efficient for managing data. Django provides various options and hooks for further customization, such as overriding templates, adding actions, and defining permissions. Refer to the Django documentation for more information on advanced customization of the administration site.

Information about Form Processing

Form processing in Django involves handling user input, validating the data, and performing CRUD (Create, Read, Update, Delete) operations on the database. Django provides a robust form handling framework that simplifies the process.
Let's explore form processing in Django in detail with CRUD program examples.

Step 1: Create a Django App

First, create a Django app within your project using the following command:

```
python manage.py startapp myapp
```

Step 2: Define a Model

In the **myapp/models.py** file, define a model that represents the data structure. For example, let's create a model for a blog post:

```
# myapp/models.py

from django.db import models

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

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

Step 3: Create a ModelForm

In Django, you can automatically generate forms from models using ModelForms. Open the myapp/forms.py file and define a ModelForm for the Post model:

```
# myapp/forms.py

from django import forms
from myapp.models import Post

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ('title', 'content')
```

In this example, we create a PostForm class that is derived from forms.ModelForm. The Meta class specifies the associated model and the fields to include in the form.

Step 4: Create Views for CRUD Operations

In the **myapp/views.py** file, create views for handling CRUD operations. For each operation, we'll define separate views:

myapp/views.py

```
from django.shortcuts import render, redirect, get_object_or_404
from myapp.models import Post
from myapp.forms import PostForm

def post_list(request):
    posts = Post.objects.all()
    return render(request, 'myapp/post_list.html', {'posts': posts})

def post_create(request):
    if request.method == 'POST':
        form = PostForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('post_list')
    else:
        form = PostForm()
    return render(request, 'myapp/post_form.html', {'form': form})

def post_update(request, pk):
    post = get_object_or_404(Post, pk=pk)
    if request.method == 'POST':
        form = PostForm(request.POST, instance=post)
        if form.is_valid():
            form.save()
            return redirect('post_list')
```

```
    else:
        form = PostForm(instance=post)
    return render(request, 'myapp/post_form.html', {'form': form})

def post_delete(request, pk):
    post = get_object_or_404(Post, pk=pk)
    if request.method == 'POST':
        post.delete()
        return redirect('post_list')
    return render(request, 'myapp/post_confirm_delete.html', {'post': post})
```

In this example, we have views for listing posts, creating a new post, updating an existing post, and deleting a post. The post_create and post_update views use the PostForm to handle the form processing.

Step 5: Create Templates

Create the necessary templates for rendering the forms and displaying the data.

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

<!DOCTYPE html>
<html>
<head>
    <title>Post List</title>
</head>
<body>
    <h1>Posts</h1>

    <ul>
```

```
    {% for post in posts %}
        <li>{{ post.title }}</li>
    {% endfor %}
    </ul>

    <a href="{% url 'post_create' %}">Create New Post</a>
</body>
</html>


<!-- myapp/templates/myapp/post_form.html -->

<!DOCTYPE html>
<html>
<head>
    <title>Post Form</title>
</head>
<body>
    <h1>Create/Edit Post</h1>

    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Save</button>
    </form>
</body>
</html>
```

Step 6: Define URLs and URL Configuration

Define URL patterns for the views in the project's main urls.py file.

Step 7: Run the Server

Start the Django development server by running the following command:
python manage.py runserver

Step 8: Access the Web Pages

Open your web browser and visit the URLs associated with the views:

- **/post_list/: Displays the list of posts.**
- **/post_create/: Displays the form to create a new post.**
- **/post_update/<pk>/: Displays the form to update an existing post.**
- **/post_delete/<pk>/: Displays the confirmation page to delete a post.**

By following this example, you'll have a complete CRUD functionality for managing posts. Users can create, read, update, and delete posts through the provided forms and views.
Keep in mind that this is a basic example, and you may need to further customize and enhance the form handling based on your specific application requirements.

Interacting with Databases: Models

Interacting with databases in Django is done through models. Models define the structure and behavior of your data, and Django's Object-Relational Mapping (ORM) handles the communication with the database.
Let's dive into the details with a program example:

Step 1: Create a Django App

First, create a Django app within your project using the following command:

```
python manage.py startapp myapp
```

Step 2: Define a Model

In the **myapp/models.py** file, define a model that represents a simple data structure. For example, let's create a model for a blog post:

```
# myapp/models.py

from django.db import models

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

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

In this example, the Post model has three fields: title, content, and pub_date. The title field is a CharField with a maximum length of 100 characters, the content field is a TextField, and the pub_date field is a DateTimeField that automatically sets the current date and time when a new post is created.

The __str__ method is defined to return a string representation of the model, which will be useful when displaying the post objects.

Step 3: Perform Migrations

Django uses migrations to handle changes to your models and keep your database schema up to date. Run the following commands to generate and apply migrations:

```
python manage.py makemigrations
python manage.py migrate
```

This will create the necessary database tables based on your model.

Step 4: Interact with the Database

To interact with the database and perform CRUD (Create, Read, Update, Delete) operations, you can use Django's ORM.
For example, let's create a new post and retrieve all posts:

```
# myapp/views.py

from django.shortcuts import render
from myapp.models import Post

def my_view(request):
```

```
    # Create a new post
    post = Post(title='First Post', content='Hello, World!')
    post.save()

    # Retrieve all posts
    posts = Post.objects.all()

    return render(request, 'myapp/my_template.html', {'posts': posts})
```

In this example, we create a new Post object with a title and content, and then save it to the database using the save() method. We retrieve all posts using the objects.all() method.

Step 5: Create a Template

Create a template file, my_template.html, in the myapp/templates/myapp directory as described in the previous example.

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

<!DOCTYPE html>
<html>
<head>
    <title>My Template</title>
</head>
<body>
    <h1>Posts</h1>

    <ul>
    {% for post in posts %}
```

```
        <li>{{ post.title }} - {{ post.content }}</li>
    {% endfor %}
    </ul>
</body>
</html>
```

This template loops through the posts queryset and displays each post's title and content in a list item.

Step 6: Define URLs and URL Configuration

Define a URL pattern for the my_view in the project's main urls.py file, similar to the previous example.

Step 7: Run the Server

Start the Django development server by running the following **python manage.py runserver**

Step 8: Access the Web Page

Open your web browser and visit `http://localhost:

About Advanced Views and urls Configuration:

Advanced views and URL configurations in Django allow you to handle more complex routing and implement advanced functionality in your web application.
Let's explore these concepts in detail with advanced project examples.

Advanced Views:

Django provides several advanced views that can be used to handle different scenarios in your web application. Here are some examples:

1. Class-Based Views (CBVs):

Class-based views provide a powerful and flexible way to structure your view logic. They are defined as classes and provide various methods for handling different HTTP methods and implementing common functionalities.

```
from django.views import View
from django.shortcuts import render

class MyView(View):
    def get(self, request):
        # Logic for handling GET requests
        return render(request, 'myapp/my_template.html')

    def post(self, request):
        # Logic for handling POST requests
        return render(request, 'myapp/my_template.html')
```

In this example, MyView is a class-based view with separate methods for handling GET and POST requests. You can override other methods like put(), delete(), etc., to handle other HTTP methods.

2.Generic Class-Based Views (GCBVs):

Django provides a set of pre-built generic class-based views that implement common functionalities like displaying lists, creating, updating, and deleting objects.

```
from django.views.generic import ListView, CreateView, UpdateView, DeleteView
from myapp.models import MyModel

class MyModelListView(ListView):
    model = MyModel
    template_name = 'myapp/mymodel_list.html'

class MyModelCreateView(CreateView):
    model = MyModel
    template_name = 'myapp/mymodel_form.html'
    fields = ['field1', 'field2']

class MyModelUpdateView(UpdateView):
    model = MyModel
    template_name = 'myapp/mymodel_form.html'
    fields = ['field1', 'field2']

class MyModelDeleteView(DeleteView):
    model = MyModel
    template_name = 'myapp/mymodel_confirm_delete.html'
    success_url = '/mymodel/list/'
```

In this example, we define generic class-based views for displaying a list of objects, creating new objects, updating existing objects, and deleting objects.

Advanced URL Configurations:
Django's URL configuration allows you to define the mapping between URL patterns and view functions or classes. Advanced URL configurations include handling parameters, named URL patterns, including URLs from other apps, and more.

1. Handling Parameters:

You can capture parameters from the URL and pass them to view functions or class-based views for further processing.

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

urlpatterns = [
    path('myview/<int:pk>/', my_view),
]
```

In this example, the URL pattern myview/<int:pk>/ captures an integer parameter (pk) from the URL. The value of pk will be passed as an argument to the my_view function.

2.Named URL Patterns:

Naming URL patterns allows you to refer to them by name in your code, making it easier to maintain and update your URLs.

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

urlpatterns = [
    path('myview/<int:pk>/', my_view, name='my_view'),
]
```

In this example, the URL pattern myview/<int:pk>/ is named as my_view. You can refer to this URL by its name using the reverse() function.

3.Including URLs from Other Apps:

You can include URLs from other apps into your project's URL configuration using the include() function.

```
from django.urls import include, path

urlpatterns = [
    # URLs from the 'myapp' app
    path('myapp/', include('myapp.urls')),
]
```

In this example, all URLs defined in the myapp.urls module will be included under the /myapp/ URL prefix.

By utilizing advanced views and URL configurations, you can build more complex and feature-rich web applications in Django. These examples provide a starting point, but remember that you can further customize and extend these concepts based on your specific project requirements.

Generic Views

Generic views in Django are pre-built class-based views that provide common functionalities for handling CRUD (Create, Read, Update, Delete) operations on models. They simplify the development process by abstracting away common patterns and reducing the amount of repetitive code.
Let's explore generic views in detail and build a CRUD project using them.

Step 1: Create a Django Project and App

First, create a new Django project and app using the following commands:

```
django-admin startproject myproject
cd myproject
python manage.py startapp myapp
```

Step 2: Define a Model

In the myapp/models.py file, define a model for your application. For example, let's create a simple model for a blog post:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
```

```
    pub_date = models.DateTimeField(auto_now_add=True)

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

In the myproject/urls.py file, define the URLs for your app, including the URLs for the generic views:

```
from django.urls import path
from myapp.views import PostListView, PostCreateView,
PostUpdateView, PostDeleteView

urlpatterns = [
    path('posts/', PostListView.as_view(), name='post_list'),
    path('posts/create/', PostCreateView.as_view(),
name='post_create'),
    path('posts/<int:pk>/update/', PostUpdateView.as_view(),
name='post_update'),
    path('posts/<int:pk>/delete/', PostDeleteView.as_view(),
name='post_delete'),
]
```


Step 4: Define Generic Views

In the myapp/views.py file, define the generic views by extending the appropriate generic view classes:

```
from django.views.generic import ListView, CreateView, UpdateView, DeleteView

from django.urls import reverse_lazy
from myapp.models import Post

class PostListView(ListView):
    model = Post
    template_name = 'myapp/post_list.html'
    context_object_name = 'posts'

class PostCreateView(CreateView):
    model = Post
    template_name = 'myapp/post_form.html'
    fields = ['title', 'content']
    success_url = reverse_lazy('post_list')

class PostUpdateView(UpdateView):
    model = Post
    template_name = 'myapp/post_form.html'
    fields = ['title', 'content']
    success_url = reverse_lazy('post_list')

class PostDeleteView(DeleteView):
    model = Post
    template_name = 'myapp/post_confirm_delete.html'
    success_url = reverse_lazy('post_list')
```

Step 5: Create Templates

Create the necessary templates for rendering the views.
myapp/templates/myapp/post_list.html:

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

<!DOCTYPE html>
<html>
<head>
    <title>Post List</title>
</head>
<body>
    <h1>Posts</h1>

    <ul>
    {% for post in posts %}
        <li>{{ post.title }}</li>
    {% endfor %}
    </ul>

    <a href="{% url 'post_create' %}">Create New Post</a>
</body>
</html>
```

myapp/templates/myapp/post_form.html:

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

<!DOCTYPE html>
<html>
<head>
```

```
    <title>Post Form</title>
</head>
<body>
    <h1>Create/Edit Post</h1>

    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Save</button>
    </form>
</body>
</html>
```

myapp/templates/myapp/post_confirm_delete.html:

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

<!DOCTYPE html>
<html>
<head>
    <title>Confirm Delete</title>
</head>
<body>
    <h1>Confirm Delete</h1>

    <p>Are you sure you want to delete "{{ object.title }}"?</p>

    <form method="post">
        {% csrf_token %}
        <button type="submit">Delete</button>
    </form>
```

```
</body>
</html>
```

Step 6: Run the Server

Start the Django development server by running the following command:

```
python manage.py runserver
```

Step 7: Access the Web Pages

Open your web browser and visit the following URLs:

- **/posts/: Displays the list of posts.**
- **/posts/create/: Displays the form to create a new post.**
- **/posts/<id>/update/: Displays the form to update an existing post.**
- **/posts/<id>/delete/: Displays the confirmation page to delete a post.**

By following these steps, you'll have a fully functional CRUD project using generic views in Django. The generic views provide the necessary functionality for listing, creating, updating, and deleting posts, reducing the amount of boilerplate code needed.
Feel free to customize the project based on your specific requirements, such as adding more fields to the model, including additional views, or applying styling to the templates.

About Extending Template Engine

Extending the template engine in Django allows you to customize and add functionality to the default template engine. You can create custom template tags, filters, and template context processors to enhance the capabilities of the template engine. Let's explore each of these components in detail with program examples.

1.Custom Template Tags:

Template tags allow you to extend the functionality of the template engine by adding custom template logic. They are written as Python functions and can perform complex operations or generate dynamic content to be rendered in templates.

Here's an example of creating a custom template tag that generates a random number:

Create a new file called myapp/templatetags/custom_tags.py:

```
from django import template
import random

register = template.Library()

@register.simple_tag
def random_number():
    return random.randint(1, 10)
```

In your template file, load the custom tags and use the random_number tag:

```
{% load custom_tags %}

<p>Random Number: {% random_number %}</p>
```

In this example, the random_number template tag generates a random number between 1 and 10 and displays it in the template.

2.Custom Template Filters:

Template filters allow you to modify the values of variables or output in templates. They can be used to format, manipulate, or filter data before rendering it in the template.
Here's an example of creating a custom template filter that truncates a string:

Create a new file called myapp/templatetags/custom_filters.py:
python

```
from django import template

register = template.Library()

@register.filter
def truncate_string(value, length):
    if len(value) > length:
        return value[:length] + '...'
    return value
```

In your template file, load the custom filters and use the truncate_string filter:

```
{% load custom_filters %}

<p>Truncated Text: {{ text|truncate_string:10 }}</p>
```

In this example, the truncate_string template filter truncates the text variable to a specified length (in this case, 10 characters) and appends ellipsis if the string is longer.

3. Custom Template Context Processors:

Template context processors allow you to add additional variables or context to the template context automatically. They are functions that take a request object as input and return a dictionary of context variables.
Here's an example of creating a custom template context processor:

- Open the myproject/settings.py file and locate the TEMPLATES setting. Add the following context processor to the context_processors list:

```
'OPTIONS': {
    'context_processors': [
        ...
        'myapp.context_processors.custom_context',
    ],
},
```

Create a new file called myapp/context_processors.py:

```
def custom_context(request):
    return {
        'app_name': 'My App',
        'user_agent': request.META.get('HTTP_USER_AGENT', ''),
    }
```

In your template file, you can directly access the context variables:
html

```
<p>App Name: {{ app_name }}</p>
<p>User Agent: {{ user_agent }}</p>
```

In this example, the custom_context template context processor adds two context variables, app_name and user_agent, to the template context. These variables can be used in any template.
By extending the template engine using custom tags, filters, and context processors, you can enhance the capabilities of Django's template engine and build more powerful and dynamic templates to suit your application's needs.

Generation of Non-HTML Content

In Django, you can generate non-HTML content using various techniques and formats. This allows you to serve different types of content such as JSON, XML, CSV, PDF, or images. Let's explore different ways to generate non-HTML content in Django with program examples.

1.Generating JSON Content:
To generate JSON content in Django, you can use the JsonResponse class or the json module.

```
from django.http import JsonResponse

def get_json_data(request):
    data = {
        'name': 'John Doe',
        'age': 30,
        'city': 'New York',
    }
    return JsonResponse(data)
```

In this example, the get_json_data view function returns a JSON response containing the data dictionary. The JsonResponse class automatically serializes the dictionary into JSON format and sets the appropriate response headers.

2. Generating XML Content:
To generate XML content in Django, you can use the HttpResponse class and manually construct the XML string using libraries such as xml.etree.ElementTree or third-party libraries like lxml.

```
from django.http import HttpResponse
import xml.etree.ElementTree as ET

def get_xml_data(request):
    root = ET.Element('person')
    name = ET.SubElement(root, 'name')
    name.text = 'John Doe'
    age = ET.SubElement(root, 'age')
    age.text = '30'

    xml_string = ET.tostring(root)

    response = HttpResponse(xml_string,
content_type='application/xml')
    response['Content-Disposition'] = 'attachment; filename="data.xml"'
    return response
```


In this example, the get_xml_data view function manually constructs an XML string using the xml.etree.ElementTree module. The XML string is then returned as an HttpResponse with the appropriate content type and headers.

3. Generating CSV Content:

To generate CSV content in Django, you can use the HttpResponse class and the csv module.

```
from django.http import HttpResponse
import csv

def get_csv_data(request):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="data.csv"'

    writer = csv.writer(response)
    writer.writerow(['Name', 'Age', 'City'])
    writer.writerow(['John Doe', 30, 'New York'])
    writer.writerow(['Jane Smith', 25, 'London'])

    return response
```


In this example, the get_csv_data view function creates an HttpResponse object with the content type set to 'text/csv' and the filename set to 'data.csv'. The csv.writer is used to write the CSV rows, and the resulting CSV data is returned as the HTTP response.

4. Generating PDF Content:
To generate PDF content in Django, you can use third-party libraries such as ReportLab or WeasyPrint.
Example (using ReportLab):

```
from django.http import HttpResponse
from reportlab.pdfgen import canvas

def get_pdf_data(request):
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="data.pdf"'

    p = canvas.Canvas(response)
    p.drawString(100, 750, "Hello, World!")
    p.showPage()
    p.save()

    return response
```

In this example, the get_pdf_data view function creates an HttpResponse object with the content type set to 'application/pdf' and the filename set to 'data.pdf'. The canvas object from ReportLab is used to generate a simple PDF with a "Hello, World!" text.
These are just a few examples of generating non-HTML content in Django. Depending on your specific requirements, you can use different techniques and formats to generate and serve the desired content types in your Django application.

Sessions, Users and Registration

Sessions, users, and registration are important components of web applications. In Django, these features are provided by the built-in authentication system, which includes session management, user authentication, and user registration. Let's explore these concepts in detail with a project example.

Project Example: User Registration and Login

Step 1: Create a Django Project

Create a new Django project by running the following command:
shell

```
django-admin startproject myproject
```

Step 2: Create a Django App

Create a new Django app within your project by running the following command:

```
cd myproject
 python manage.py startapp myapp
```

Step 3: Define the User Model

Open the myapp/models.py file and define a custom user model by extending the AbstractUser class provided by Django:

```
from django.contrib.auth.models import AbstractUser

class CustomUser(AbstractUser):
    pass
```

Step 4: Update the User Model in Settings

Open the myproject/settings.py file and update the AUTH_USER_MODEL setting to point to your custom user model:

```
AUTH_USER_MODEL = 'myapp.CustomUser'
```

Step 5: Define the Registration and Login Views

Create views for user registration and login in the myapp/views.py file:

```
from django.shortcuts import render, redirect
from django.contrib.auth.forms import UserCreationForm,
AuthenticationForm
from django.contrib.auth import login
from django.contrib import messages

def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('login')
    else:
        form = UserCreationForm()
    return render(request, 'register.html', {'form': form})
```

```
def user_login(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            user = form.get_user()
            login(request, user)
            messages.success(request,f”Login {user} succssfully”)

            return redirect('home')
    else:
        form = AuthenticationForm()
    return render(request, 'login.html', {'form': form})
```

Step 6: Create the Registration and Login Templates

Create templates for user registration (register.html) and login (login.html) in the myapp/templates directory. Here's an example of the registration template:

```
<!-- register.html -->

<!DOCTYPE html>
<html>
<head>
    <title>User Registration</title>
</head>
<body>
    <h1>User Registration</h1>

    <form method="post">
```

```
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Register</button>
    </form>
</body>
</html>
```

Step 7: Handle Logout

To handle user logout, you can create a logout view that clears the user's session:

```
from django.contrib.auth import logout
def user_logout(request):
        logout(request)
        return redirect('login')
```

Step 8: Define URLs and Views

Open the myproject/urls.py file and define the URLs and corresponding views for user registration and login:

```
from django.contrib import admin
from django.urls import path
from myapp.views import register, user_login,user_logout

urlpatterns = [
    path('admin/', admin.site.urls),
    path('register/', register, name='register'),
    path('login/', user_login, name='login'),
    path('logout/',user_logout,name='logout'),
]
```

Step 8: Run the Server

Start the Django development server by running the following command:

python manage.py runserver

Step 9: Access the Registration and Login Pages

Open your web browser and visit the following URLs:

- **/register/: Displays the user registration form.**
- **/login/: Displays the user login form.**

By following these steps, you'll have a basic user registration and login functionality in your Django project. Users can register using the registration form and login using their credentials. You can further enhance the project by adding features such as user authentication, password reset, and user profile management.

Note: This example demonstrates the basic implementation of user registration and login. In a production environment, it's important to follow security best practices, such as **validating user inputs, handling errors, and using password hashing and authentication measures**.

Sending Mail in Django

To send an email in Django using the send_mail function, you need to configure your email settings in the Django project settings file (settings.py). Here's a step-by-step guide to help you:

1. Open your Django project's settings.py file.
2. Locate the EMAIL_BACKEND setting and set it to the appropriate email backend you want to use.
3. For example, if you want to use the SMTP backend, you can set it as follows:

   ```
   EMAIL_BACKEND='django.core.mail.backends.smtp.EmailBackend'
   ```
4. Configure the SMTP settings according to your email provider. Add the following settings to your settings.py file and modify them with your email provider's details:

   ```
   EMAIL_HOST = 'your-email-host' # e.g., smtp.gmail.com
   EMAIL_PORT = 587 # The SMTP port for your email provider
   EMAIL_HOST_USER = 'your-email@example.com' # Your email address EMAIL_HOST_PASSWORD = 'your-email-password' # Your email password EMAIL_USE_TLS = True # Enable TLS encryption for security DEFAULT_FROM_EMAIL = 'your-email@example.com' # The "From" address for sent emails
   ```

   Save the changes to settings.py.
5. Now, you can use the send_mail function in your Django views or wherever you want to send an email. Here's an example of how to use it:

```
from django.shortcuts import render
from django.core.mail import send_mail
from django.template.loader import render_to_string
from django.http import HttpResponse

def send_email(request):
    subject = 'Test Email'
#     message = 'This is a test email from Django!'
    message = render_to_string('test.html')
    from_email = 'himalrawal500@gmail.com'
    recipient_list = ['sipalayainfotech01@gmail.com',
'himal8848rawal@gmail.com']
    send_mail(subject, message, from_email, recipient_list,
fail_silently=False,html_message=message)
    return HttpResponse("Successfully Sent Mail")
```

The send_mail function takes several arguments:

- Subject of the email.
- Body of the email.
- From email address.
- List of recipient email addresses.
- fail_silently parameter controls whether exceptions are raised if an error occurs while sending the email. Set it to True to suppress exceptions.

That's it! With the appropriate email settings configured and using the send_mail function, you can send emails in your Django application.

Contact Us

🏠 Nearby CCRC College Balkumari Bridge,
Koteshwor, Kathmandu Nepal
WhatsApp / Phone Call
📱 9851344071 / 9818968546
📱Phone Call:
9806393939 / 9860267997
📩 infotech@sipalaya.com
🌐 https://sipalaya.com/
🌐Google Map Link:
https://goo.gl/maps/HY8oiYmEUzQ2nVGRA

Happy Coding !!
Thank You !!