TEMPLATES

Introduction:

Django Templates are a way to separate the presentation layer from the business logic. Basically these are HTML files that contain the dynamic content of our website.

Working with App Level Templates:

App level Templates are nothing by the templates which are created in apps. And to do so create a folder called **" templates "** in any of the app.

For example, if you have an app called **blog** then you will create a folder inside this app so your template directory should look something like **blog/templates.**

Make sure while you create this **"templates"** folder, you should not make any spelling mistake, it should be spelled as it is above and in lower case.

Creating template / HTML file:

- Create a file in the **templates** folder with an extension of **'.html'** .
  Example '**home.html**'
- Write some html code in this file.

```
<!-- blog/templates/home.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Home Page</title>
</head>
<body>
    <h1>Welcome to Home Page</h1>
</body>
</html>
```

Rendering the Templates:

- After creating the template we need to render it or send it as an Http Response object.
- So to do that django provides us with an inbuilt function called as **'render()'**
- First go to your **views.py** and write a function or a class to render this template.
- Example shown below

The **`render`** function in Django is a shortcut used to combine a given template with a context dictionary and return an HttpResponse object with that rendered text. It is typically used in view functions to simplify the process of rendering templates.

Key Points:

1. **Purpose:**
    - The **`render`** function simplifies the process of generating an HttpResponse with the content of a rendered template.
    - It automatically takes care of loading the template, filling it with context data, and returning an HttpResponse.

2. **Syntax:**

```
from django.shortcuts import render

def view_function(request):
    context = {'key': 'value'}
    return render(request, 'template_name.html', context)
```

3. **Parameters**
    - **request:** The HttpRequest object.
    - **template_name:** The name of the template to be rendered.
    - **context (optional):** A dictionary containing the context data to be passed to the template.
    - **content_type (optional):** The MIME type to use for the resulting document (default is 'text/html').
    - **status (optional):** The status code for the response (default is 200).
    - **using (optional):** The name of the template engine to use for loading the template.

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

def home(request):
    return render(request, 'home.html')
```

- In the above view we have used the **render()** function which takes two mandatory arguments i.e. **request & 'template_name'.**
- This setup will render the **`home.html`** template when the **home** view is accessed.

Observative Case:

- Let's just consider two apps **app1** & **app2.**
- Now in both of these files create templates folder and then create a file ex: **index.html** in both the templates.
- Now if you try to call any of these files then the file from the first registered apps template will be called.
- Ex: If we register **app1** first in the settings.py and then **app2** then the **index.html** file which is created in then **app1's** templates will be called.

Template Inheritance:

- As we create multiple templates it can be noticed that a lot of code is being repeated in each of the template or common.
- **Template inheritance** in Django allows you to create a base/root template that contains common elements for your site, such as headers, footers, and navigation bars.
- Individual pages can then extend this base template, adding or overriding specific content blocks as needed. This approach promotes reusability and a consistent design across your site.

Key Concepts

1. **Base Template:** A template that contains common layout and structure, with designated blocks that child templates can override.
2. **Child Template:** A template that extends the base template and fills in the blocks defined by the base template.

Steps to Implement Template Inheritance

1. **Create a Base Template**
   - Create a base template **`base.html`** in a shared templates directory, such as **`templates/`**.
   - The base template serves as the skeleton for your site. It defines the common structure and includes placeholder blocks where content can be inserted.

templates > base.html > html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
    <title>Document</title>
</head>
<body>

    <nav>
        <!-- Navbar code  -->
    </nav>

    {% block content %}
    {% endblock content %}

</body>
</html>
```

- **`{% block %}`:** Defines a block that child templates can override. The name of the block (**`content`**) specifies where content will be inserted or replaced.

2. **Create Child Templates**
   - Create a child template that extends the base template, such as **`home.html`**.
   - Child templates extend the base template and define content for the placeholder blocks. The **`{% extends %}`** tag is used to inherit the base template, and **`{% block %}`** tags are used to provide content for the named blocks.

```
<!-- blog/templates/home.html -->
{% extends "base.html" %}

{% block title %}Home Page{% endblock %}

{% block content %}
    <h1>Welcome to Home Page</h1>
    <p>This is the home page of the site.</p>
{% endblock %}
```

   - **`{% extends %}`:** Indicates that this template inherits from **`base.html`**. This tag must be the first tag in the template.
   - **`{% block %}`:** Provides content for the blocks defined in the base template. The content within **`{% block content %}`** will replace the content block in the base template.

3. **Create a View to Render the Child Template**
   - Define a view to render the **`home.html`** template.

4. **Set Up URL Configuration**
   - Map a URL pattern to the **`home`** view in your app's **`urls.py`**.

Benefits of Template Inheritance

Reusability: Common layout elements are defined once in the base template and reused across multiple pages.

Maintainability: Changes to the common layout need to be made only in the base template, simplifying maintenance.

Consistency: Ensures a consistent look and feel across different pages of the site.

Context:

- **Context** in Django templates refers to the data passed from the view to the template. This data is then used to dynamically generate the HTML content of the web page. Context is typically passed as a dictionary where keys are variable names and values are the corresponding data.

```python
# blog/views.py
from django.shortcuts import render

def homepage(request):
    context = {
        'title': 'Home Page',
        'message': 'Welcome to the homepage!'
    }
    return render(request, 'blog/homepage.html', context)
```

- Context provides the dynamic data that the template uses to render the final HTML. This allows templates to display personalized or dynamic content based on the data provided by the views.
- **Passing Context:** Context is passed from views to templates using the `render` function.
- **Accessing Context:** Within the template, context variables can be accessed using double curly braces **{{ }}**.

```html
<!-- blog/templates/blog/homepage.html -->
<!DOCTYPE html>
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ message }}</h1>
    <ul>
        {% for item in items %}
            <li>{{ item }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Passing a list in context to templates:

Passing a list in context to Django templates allows you to dynamically display multiple items on a webpage. This can be useful for displaying lists of objects, such as blog posts, products, or any collection of items.

```python
# blog/views.py
from django.shortcuts import render

def home(request):
    # Define a list of items
    items = ['Item 1', 'Item 2', 'Item 3']

    # Create a context dictionary
    context = {
        'items': items
    }

    # Pass the context to the template
    return render(request, 'home.html', context)
```

Rendering the List in Template:

```
<!-- blog/templates/home.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Home Page</title>
</head>
<body>
    <h1>Welcome to Home Page</h1>
    <ul>
        {% for item in items %}
            <li>{{ item }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Using `{% for %}` Loop in Django Templates:

The **`{% for %}`** loop in Django templates is used to iterate over a sequence, such as a list, queryset, or any iterable, and render the content for each item. This allows you to dynamically generate HTML content based on the data passed from the view.

Syntax:

```
{% for item in items %}
    <!-- Render content for each item -->
    {{ item }}
{% endfor %}
```

Loop Variables

Django's **`{% for %}`** loop provides several built-in variables that are available within the loop:

- **`forloop.counter`**: The current iteration of the loop (1-indexed).
- **`forloop.counter0`**: The current iteration of the loop (0-indexed).
- **`forloop.revcounter`**: The number of iterations from the end of the loop (1-indexed).
- **`forloop.revcounter0`**: The number of iterations from the end of the loop (0-indexed).
- **`forloop.first`**: `'True'` if this is the first time through the loop.
- **`forloop.last`**: `'True'` if this is the last time through the loop.
- **`forloop.parentloop`**: For nested loops, this is the loop containing the current one.

Passing a List of Dictionaries in Context to Django Templates

When you have a list of dictionaries, you can pass it to a Django template and iterate over it using the **`{% for %}`** loop. Each dictionary can represent an object with multiple properties, such as tasks with a name, description, and status.

```python
# tasks/views.py
from django.shortcuts import render

def task_list(request):
    tasks = [
        {
            'id': 1,
            'name': 'Task 1',
            'description': 'Description for task 1'
        },
        {
            'id': 2,
            'name': 'Task 2',
            'description': 'Description for task 2'
        }
    ]

    context = {
        'tasks': tasks
    }
    return render(request, 'tasks/task_list.html', context)
```

Access the List of Dictionaries in the Template

Within the template, use a **`{% for %}`** loop to iterate over the list of dictionaries and display each task's properties.

```
<!-- tasks/templates/tasks/task_list.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Task List</title>
</head>
<body>
    <h1>Task List</h1>
    <ul>
        {% for task in tasks %}
            <li>
                <strong>{{ task.name }}</strong>: {{ task.description }} ({{ task.status }})
            </li>
        {% endfor %}
    </ul>
</body>
</html>
```

- **`{% for task in tasks %}`:** This tag starts the loop, iterating over each dictionary in the `tasks` list.
- **`{{ task.name }}`, `{{ task.description }}`, `{{ task.status }}`:** These tags access the name, description, and status properties of the current dictionary in the loop.

Using `{% include %}` Tag in Django Templates

The `{% include %}` tag in Django templates is used to include the contents of another template. This is useful for reusing common components across multiple templates, such as a navigation bar, footer, or any other shared element.

Benefits of Using `{% include %}`

1. **Reusability:** Allows you to reuse the same template code in multiple places, reducing redundancy.
2. **Maintainability:** Makes it easier to update common components since you only need to update the included template.
3. **Modularity:** Encourages a modular approach to template design, making the code cleaner and more organized.

Syntax:

```
{% include 'path/to/template.html' %}
```

Example:

Create the Navbar Template

First, create a template for the navbar. Save this template in a common location.

```
<nav>
    <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/about/">About</a></li>
        <li><a href="/contact/">Contact</a></li>
    </ul>
</nav>
```

Include the Navbar in Another Template

Now, include the **`navbar.html`** template in another template, such as the base template.

```
<!-- blog/templates/base.html -->
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My Site{% endblock %}</title>
</head>
<body>
    {% include 'partials/navbar.html' %}

    <div class="content">
        {% block content %}
        <!-- Content will be injected here by child templates -->
        {% endblock %}
    </div>
</body>
</html>
```

Dynamic URL

Making Elements Clickable in Django Templates

In web development, making elements clickable typically involves using the **<a>** (anchor) tag, which creates a hyperlink. In Django templates, you often need to dynamically generate URLs using the **{% url %}** tag.

The {% url %} Tag in Django Templates

The **{% url %}** tag in Django templates is a powerful feature that allows you to dynamically generate URLs by reversing the URL patterns defined in your **urls.py** file. This is particularly useful for ensuring that your links are always up-to-date and correct, even if the URL patterns change.

Why Use {% url %}?

- **Maintainability**: If you change a URL pattern in **urls.py**, you don't need to search and replace all instances of that URL in your templates. The **{% url %}** tag will automatically generate the correct URL.
- **Readability**: Using named URL patterns makes your templates more readable and easier to understand.
- **Error Reduction**: It reduces the chances of hard-coding errors and broken links in your templates.

Syntax:

```
{% url 'url_name' arg1 arg2 %}
```

- **`url_name`**: The name of the URL pattern as defined in your **`urls.py`** file.

Define URL Patterns with URL Names

- First, define URL patterns in your **urls.py** file, including URL names.

```python
# blog/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),           # URL name: 'home'
    path('about/', views.about, name='about'),   # URL name: 'about'
    path('contact/', views.contact, name='contact'), # URL name: 'contact'
]
```

- Define the views in your **views.py** file.

Update navbar.html to Use URL Names with {% url %} Tag:

Modify the **navbar.html** template to use URL names with the **{% url %}** tag.

```html
<nav>
    <ul>
        <li><a href="{% url 'home' %}">Home</a></li>
        <li><a href="{% url 'about' %}">About</a></li>
        <li><a href="{% url 'contact' %}">Contact</a></li>
    </ul>
</nav>
```

Path Converters:

- Path converters in Django are components of URL patterns that capture and convert parts of the URL into Python objects that can be passed as arguments to view functions.
- You can think of the path converters in Django as specifying the expected data type or format of the variable captured from the URL. While Django documentation typically refers to them as "**converters**" or "**path converters**"
- They allow Django to match specific patterns in URLs and extract dynamic data from them, such as IDs or slugs, which are then used to retrieve corresponding content from databases or perform specific actions.

Syntax:

```
<converter:variable_name>
```

Types of Path Converters in Django

1. **`<int:name>`**: Matches an integer and passes it as an integer argument to the view function.
    - Example: **`<int:pk>`** matches an integer primary key.
2. **`<str:name>`**: Matches any string without a slash (**`/`**) and passes it as a string argument to the view function.
    - Example: **`<str:st>`** matches a string st.
3. **`<slug:name>`**: Matches a slug, which is a short label containing only letters, numbers, underscores, or hyphens.
    - Example: **`<slug:post_slug>`** matches a post slug.
4. **`<uuid:name>`**: Matches a Universally Unique Identifier (UUID) and passes it as a UUID object argument to the view function.
    - Example: **`<uuid:pk>`** matches a UUID primary key.
5. **`<path:name>`**: Matches any string including slashes (**`/`**) and passes it as a string argument to the view function.
    - Example: **`<path:document_path>`** matches a document path.

URL pattern including an argument:

Consider a scenario where you have a URL pattern that includes an argument (**`id`**) for a post detail page:

Define URL Pattern:

Define your URL patterns in **urls.py**, including a pattern with an argument (**id**):

```
# blog/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
    path('post/<int:id>/', views.post_detail, name='post_detail'),
]
```

- The syntax **'post/<int:id>/'** in Django's URL patterns defines a URL route that expects a numeric integer (**<int:id>**), which is a placeholder for an actual ID value.

Create Views

Create the views in views.py. In this example, post_detail retrieves a specific post based on the id parameter:

```python
# blog/views.py
from django.shortcuts import render, get_object_or_404
from .models import Post

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

def post_detail(request, id):
    post = get_object_or_404(Post, id=id)
    return render(request, 'post_detail.html', {'post': post})
```

Use {% url %} Tag with Arguments

In your templates (**home.html** in this example), use the **{% url %}** tag to generate URLs dynamically with arguments:

```
<!-- blog/templates/home.html -->
{% extends 'base.html' %}

{% block title %}Home Page{% endblock %}

{% block content %}
<h1>Welcome to Home Page</h1>
<p>This is the home page content.</p>

<ul>
    {% for post in posts %}
        <!-- Generate URL for each post detail page -->
        <li><a href="{% url 'post_detail' post.id %}">{{ post.title }}</a></li>
    {% endfor %}
</ul>
{% endblock %}
```

Explanation

- **URL Pattern (urls.py)**:
  - The URL pattern path('post/<int:id>/', views.post_detail, name='post_detail') defines a URL that expects an integer (id) and maps it to the post_detail view.
- **View (views.py)**:
  - The post_detail view retrieves a specific post using get_object_or_404(Post, id=id) based on the id parameter passed in the URL.
- **Template (home.html)**:
  - The {% url %} tag generates a URL for each post using post.id as the argument. This dynamically creates links to each post's detail page.

Project-Level Templates in Django

Project-level templates in Django refer to templates that are shared across multiple applications within a Django project. These templates are typically stored in a centralized location and can be reused across different parts of the project.

Key Points:

- **Centralized Storage**: Project-level templates are stored in a directory typically named templates at the root level of your Django project.
- **Reusable Across Apps**: They can be reused by multiple Django applications (apps) within the same project. This promotes code reusability and consistency in the user interface across different parts of the application.
- **Settings Configuration**: To enable Django to find these project-level templates, ensure that the TEMPLATES setting in your project's settings.py file includes the directory where your project-level templates are stored.
- **Template Loading Order**: Django follows a specific order when loading templates, first looking in the app directories and then in the project-level templates directory. This ensures that app-specific templates can override project-level templates if necessary.

Configuration of Project-Level Templates in settings.py

In Django, configuring project-level templates involves specifying where Django should look for templates and how it should handle template loading. This configuration is done in the settings.py file of your Django project.

TEMPLATES Setting

The TEMPLATES setting in settings.py is a list of configurations used by Django's template engine to render templates. Within TEMPLATES, you specify options like template directories, context processors, and template engines.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            BASE_DIR / 'templates',  # Project-level templates directory
        ],
        # Other options can be configured here
    },
]
```

'DIR' Specifies a list of directories where Django should look for templates.

Typically includes the **BASE_DIR / 'templates'** to point to the project-level templates directory.

BASE_DIR: Refers to the root directory of your Django project.

Configuring project-level templates in settings.py is essential for Django to effectively manage and render templates across your project. By specifying template directories, enabling app-specific template discovery, and utilizing context processors, you streamline template handling, promote code reuse, and maintain consistency in your Django project's user interface.

Benefits:

- **Centralized Management**: Centralizing project-level templates in one directory (BASE_DIR / 'templates') promotes code organization and makes it easy to locate and update shared templates.
- **Automatic Template Discovery**: By setting APP_DIRS to True, Django automatically searches for templates in each app's templates/ directory, simplifying template loading.
- **Context Processors**: Automatically adds useful context variables to templates, reducing the need to manually pass common data to every view.

Static Files in Django

Static files in Django refer to files like CSS, JavaScript, images, and other assets that are served directly to clients without any processing by the Django server. These files are typically used for styling web pages, adding interactivity, and including media such as images or downloadable files.

Configuration in settings.py

To configure static files in Django, you need to specify certain settings in your **settings.py** file**:**

```
# settings.py

# Define the base directory for static files
STATIC_URL = '/static/'

# Define the directories where Django will look for static files
STATICFILES_DIRS = [
    BASE_DIR / 'static',  # Project-level static files directory
]

# Define the directory where collected static files will be stored during deployment
STATIC_ROOT = BASE_DIR / 'staticfiles'
```

STATIC_URL:

- Specifies the URL prefix for accessing static files. By default, it's **/static/**, meaning static files will be served from URLs like http://example.com/static/.

STATICFILES_DIRS:

- Defines a list of directories where Django will search for additional static files.
- **BASE_DIR / 'static'**: Points to the static directory located at the root (**BASE_DIR**) of your Django project, where project-specific static files are stored.

STATIC_ROOT:

- Specifies the directory where collected static files will be stored during deployment.
- It's typically set to **BASE_DIR / 'staticfiles'**, ensuring that all static files from apps and **STATICFILES_DIRS** are collected into this directory for deployment.

Serving Static Files During Development

During development (DEBUG=True in settings.py), Django's built-in development server (runserver) automatically serves static files from STATICFILES_DIRS and app-specific static directories.

Usage in Templates and Static Files:

Activating Static Files Handling

```
{% load static %}
```

- This tag must be included at the beginning of your template file where you intend to use **{% static %}** and other static files-related template tags.
- In Django, the **{% load static %}** template tag is used to load the static files template tags and filters, enabling you to easily reference static files like CSS, JavaScript, images, and other assets within your HTML templates.

Using {% static %} Template Tag

```
<link rel="stylesheet" href="{% static 'css/style.css' %}">
```

- **{% static 'css/style.css' %}** resolves to the URL of the static file located at **BASE_DIR / 'static/css/style.css'**, respecting **STATIC_URL** configuration.

Benefits

- **Modular and Reusable**: {% load static %} allows for modular inclusion of static files handling across templates, promoting code reusability and maintainability.
- **Path Resolution**: {% static %} resolves paths to static files based on STATIC_URL and STATICFILES_DIRS, ensuring correct URLs for static assets in both development and deployment.
- **Integration**: Seamlessly integrates with Django's static files configuration (STATIC_URL, STATICFILES_DIRS, STATIC_ROOT), streamlining asset management in templates.

The {% load static %} template tag is essential in Django for activating static files handling within templates. By using {% static %} alongside it, you can efficiently include CSS, JavaScript, images, and other static assets in your HTML templates, ensuring optimal performance and organization in your Django projects.