FULLSTACK DEVELOPMENT

MODULE 2 - Django Templates and Models

CONTENTS

In the previous chapter, we returned HTML directly from our Python code in example views, leading to several issues. Changing the design required altering the Python code, which is inefficient since design changes are more frequent. Additionally, writing Python and designing HTML are different disciplines often handled by separate people or departments. Combining Python and HTML in one file forces sequential work, where one person waits for the other to finish. To address these problems, we use Django's template system, which separates the design (HTML) from the logic (Python). This approach makes the code cleaner, more maintainable, and allows designers and developers to work simultaneously without interfering with each other's work. The template system enables easy design changes without affecting the underlying Python code, improving efficiency and flexibility in web development.

1. Template System Basics

A Django template is a string of text designed to separate the presentation of a document from its data. Templates define placeholders and basic logic (template tags) that dictate how the document should be displayed. While typically used for producing HTML, Django templates can generate any text-based format.

Example Template

Below is an example of a simple template for an HTML page that thanks a person for placing an order:

```
<html>
<head><title>Ordering notice</title></head>
<body>
<p>Dear {{ person_name }},</p>
<p>Thanks for placing an order from {{ company }}. It's scheduled to ship
on  <p>{{ ship_date|date:"D d M Y" }}</p>
<p>Here are the items you've ordered:</p>
<ul>
{% for item in item_list %}
<li>{{ item }}</li>
{% endfor %}
</ul>
{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% endif %}
<p>Sincerely,<br />{{ company }}</p>
</body>
</html>
```

Key Components

Variables:

Denoted by double curly braces: {{ variable_name }}

Insert the value of the given variable name.

Example: {{ person_name }}

Template Tags:

Denoted by {% tag_name %}

Perform actions or logic within the template.

Examples:

{% for item in item_list %}: Loop through each item in item_list.

{% if ordered_warranty %}: Conditional logic to check if ordered_warranty is true.

Filters:

Used to alter the display of a variable.

Attached using a pipe character (|).

Date Filter: {{ ship_date| date:"D d M Y" }} formats the ship_date variable into a readable date format.

2. **Using Django Template System**

To utilize the Django template system in your Python code, follow these two steps:

1. **Create a Template Object**:
   - Instantiate the Template class with raw template code as a string.
   - Alternatively, you can create Template objects by specifying the path to a template file on the filesystem.

2. **Render the Template**:
    - Call the render() method of the Template object with a context (a dictionary of variables).
    - This returns a fully rendered template as a string, with all variables and block tags evaluated according to the context.

Creating Template Objects

To create a Template object:

- Import the Template class from django.template.
- Instantiate the Template class with raw template code.

Example:

Python code

```
>>> from django.template import Template
>>> t = Template("My name is {{ name }}.")
>>> print(t)
<django.template.Template object at 0xb7d5f24c>
```

The output shows the Python identity of the Template object.

Error Handling:

- If there are syntax errors in the template code, a TemplateSyntaxError exception is raised.

Example:

Python code

```
>>> from django.template import Template
>>> t = Template('{% notatag %} ')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
...
django.template.TemplateSyntaxError: Invalid block tag: 'notatag'
```

Causes for TemplateSyntaxError:

- Invalid block tags
- Invalid arguments to valid block tags
- Invalid filters
- Invalid arguments to valid filters
- Invalid template syntax
- Unclosed block tags

By following these steps, you can effectively use Django's template system to separate your presentation layer from your business logic, ensuring cleaner and more maintainable code.

- ➢ **Rendering a Template**

Rendering templates in Django is a straightforward process that separates presentation from logic, making your code cleaner and more maintainable. Here's a brief overview

Process Overview:

1. **Template Creation**: Define your template with placeholders and template tags.
2. **Context Preparation**: Prepare a dictionary of variables and their values.
3. **Template Rendering**: Combine the template with the context to produce the final rendered output.

Example

1. **Import Classes**:
   - Import Template and Context from django.template:

```python
from django.template import Template, Context
```

2. **Define Raw Template**:
   - Define a multi-line string as your template:

```
raw_template = """
<p>Dear {{ person_name }},</p>
<p>Thanks for ordering {{ product }} from {{ company }}. It's scheduled
to ship on {{ ship_date|date:"F j, Y" }}.</p>
{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% endif %}
<p>Sincerely,<br />{{ company }}</p>
"""
```

3. **Create Template Object**:
   - Instantiate the Template class with the raw template:

```
t = Template(raw_template)
```

4. **Create Context**:
   - Prepare the context with the necessary data:

```
import datetime
c = Context({
    'person_name': 'John Smith',
    'product': 'Super Lawn Mower',
    'company': 'Outdoor Equipment',
    'ship_date': datetime.date(2009, 4, 2),
    'ordered_warranty': True
})
```

5. **Render Template**:
   - Render the template with the context and print the result:

```
rendered_template = t.render(c)
print(rendered_template)
```

6. **Rendered Output (html document in web browser)**

```
<p>Dear John Smith,</p>
<p>Thanks for ordering Super Lawn Mower from Outdoor Equipment. It's scheduled
to ship on April 2, 2009.</p>
<p>Your warranty information will be included in the packaging.</p>
<p>Sincerely,<br />Outdoor Equipment</p>
```

- **Context Variable Lookup**

In Django templates, the dot character (.) is a powerful tool for accessing values within complex data structures such as dictionaries, lists, and custom objects. The dot notation allows for flexible and intuitive data traversal, making it easy to render dynamic content.

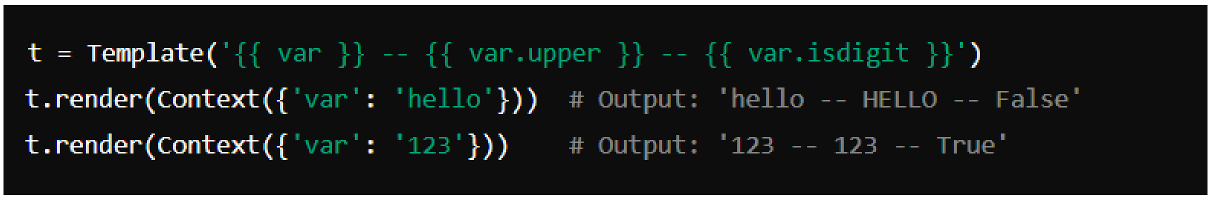
1. **Dictionary Access**:
   - Use the dot to access dictionary keys.

```
from django.template import Template, Context
person = {'name': 'Sally', 'age': '43'}
t = Template('{{ person.name }} is {{ person.age }} years old.')
c = Context({'person': person})
t.render(c)  # Output: 'Sally is 43 years old.'
```

2. **Object Attributes**:
   - Access attributes of objects using the dot notation.

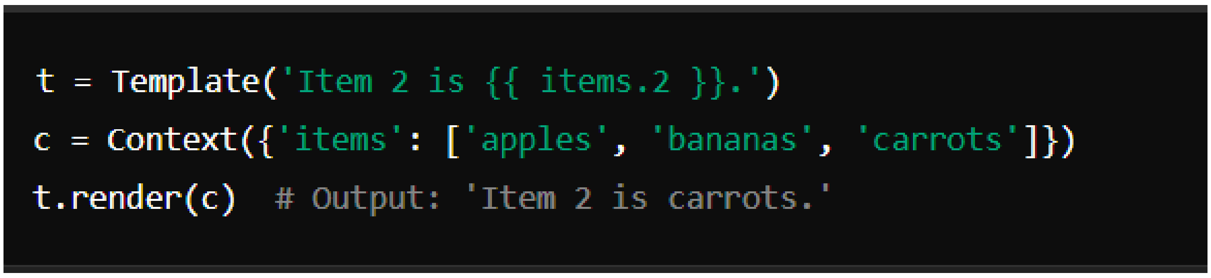
```
import datetime
d = datetime.date(1993, 5, 2)
t = Template('The month is {{ date.month }} and the year is {{ date.year }}.')
c = Context({'date': d})
t.render(c)  # Output: 'The month is 5 and the year is 1993.'
```

3. **Calling Methods**:
    - Call methods of objects without parentheses.



```
t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
t.render(Context({'var': 'hello'}))  # Output: 'hello -- HELLO -- False'
t.render(Context({'var': '123'}))    # Output: '123 -- 123 -- True'
```

4. **List Indices**:
    - Access list items by index.



```
t = Template('Item 2 is {{ items.2 }}.')
c = Context({'items': ['apples', 'bananas', 'carrots']})
t.render(c)  # Output: 'Item 2 is carrots.'
```

By understanding and leveraging these dot notation capabilities, you can create dynamic and robust Django templates that efficiently access and render complex data structures.

3. Basic Template Tags and Filters

Template Tags

Template tags in Django are special syntax constructs used within Django templates to control the rendering logic. They can perform operations such as displaying content conditionally, iterating over data, or including other templates.

Why We Use Them:

- **Dynamic Content**: To display content dynamically based on certain conditions or data from the context.

- **Logic in Templates**: To include basic logic like loops and conditional statements directly in the HTML template, keeping the view logic separate from the presentation.

Where We Use Them:

- **HTML Templates**: Template tags are used within Django HTML templates (files with extensions like .html) to control how the data passed from views is displayed.

Django Template Tags are:

1. if/else Tag:

- **Purpose**: Used to evaluate a condition and render content based on whether the condition is true or false.
- **Structure**:
  - {% if condition %}: Starts the block that is rendered if the condition is true.
  - {% else %}: (Optional) Starts the block that is rendered if the condition is false.
  - {% endif %}: Ends the conditional block.
- **Logic**: Supports logical operators such as and, or, and not to combine or negate conditions. Cannot combine and and or within the same tag, but nested if tags can be used for complex conditions.

2. for Tag:

- **Purpose**: Iterates over each item in a sequence, rendering the enclosed block for each item.
- **Structure**:
  - {% for item in sequence %}: Starts the loop, with item being the variable name for each element in the sequence.

    - {% endfor %}: Ends the loop block.
- **Features**:
    - Can loop in reverse using {% for item in sequence reversed %}.
    - Supports nested loops.
    - Provides the forloop variable within the loop with attributes such as:
        - forloop.counter: 1-based index of the current iteration.
        - forloop.counter0: 0-based index of the current iteration.
        - forloop.revcounter: 1-based index from the end of the sequence.
        - forloop.revcounter0: 0-based index from the end of the sequence.
        - forloop.first: Boolean indicating if it's the first iteration.
        - forloop.last: Boolean indicating if it's the last iteration.
        - forloop.parentloop: Reference to the parent loop's forloop object in nested loops.

3. ifequal/ifnotequal Tags:

- **Purpose**: Compares two values and conditionally renders content if they are equal or not equal.
- **Structure**:
    - {% ifequal value1 value2 %}: Starts the block that is rendered if value1 equals value2.
    - {% ifnotequal value1 value2 %}: Starts the block that is rendered if value1 does not equal value2.
    - {% else %}: (Optional) Starts the block that is rendered if the initial condition is false.
    - {% endifequal %} or {% endifnotequal %}: Ends the comparison block.

- **Limitations**: Supports comparisons with template variables, strings, integers, and decimal numbers. Does not support comparisons with other types of variables like lists or dictionaries.

4. Comments:

- **Purpose**: Adds comments to templates that are not rendered in the final output.
- **Structure**: {# comment #}
- **Limitation**: Comments cannot span multiple lines to maintain parsing performance.

Example

Urls.py

```
urlpatterns = [
    path('fordemo/',views.myfun2)
]
```

Views.py

```python
def myfun2(request):
    students = [
        {
            'name': 'deepika',
            'courses': ['python', 'java', 'Azure']
        },
        {
            'name': 'lakshmi',
            'courses': ['.net', 'sql', 'aws']
        },
        {
            'name': 'pooja',
            'courses': ['mern stack', 'cloud tech', 'agile']
        }
    ]
    context={'students':students}
    return render(request, 'home2.html',context)
```

Template (HTML file)

```html
<!DOCTYPE html>
<html>
<head>
<title>Students and Courses</title>
</head>
<body>
<h1>Student List</h1>

<!-- Basic Looping -->
    <h2>Basic Looping</h2>
    <ol>
        {% for student in students %}
            <li>{{ student.name }}</li>
        {% endfor %}
    </ol>

    <!-- Reversed Looping -->
    <h2>Reversed Looping</h2>
    <ul>
        {% for student in students reversed %}
            <li>{{ student.name }}</li>
        {% endfor %}
    </ul>

    <!-- Nested Loops -->
    <h2>Nested Loops</h2>
    <ul>
        {% for student in students %}
            <li>
                <h3>{{ student.name }}</h3>
                <ul>
                    {% for course in student.courses %}
                        <li>{{ course }}</li>

                    {% endfor %}
                </ul>
            </li>
        {% endfor %}
    </ul>




    <!-- Using forloop Variable -->
    <h2>Using forloop Variable</h2>
    <ul>
        {% for student in students %}
            <li>
                <h3>{{ forloop.counter }}. {{ student.name }}</h3>
                <p>First in loop: {{ forloop.first }}</p>
                <p>Last in loop: {{ forloop.last }}</p>
                <ul>
                    {% for course in student.courses %}
                        <li>{{ forloop.counter }}. {{ course }}</li>
                    {% endfor %}
                </ul>
            </li>
        {% endfor %}
    </ul>


    <h2>Using forloop.parentloop</h2>
    <ul>
        {% for student in students %}
            <li>
                <h3>{{ student.name }}</h3>
                <ul>
                    {% for course in student.courses %}
```

```
                        <li>
                            <!-- Accessing the parent loop's counter -->
                Student {{ forloop.parentloop.counter }} - Course {{
forloop.counter }}: {{ course }}
                        </li>
                    {% endfor %}
                </ul>
            </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Template Filters

Template filters are functions that modify the value of a variable before it is displayed. Filters can be chained to apply multiple transformations to a variable's value.

Why We Use Them:

- **Data Formatting**: To format data before displaying it, such as converting text to lowercase, formatting dates, or truncating strings.
- **Data Manipulation**: To manipulate data directly within templates without modifying the original data in views or models.

Where We Use Them:

- **Within Template Expressions**: Filters are applied within double curly braces ({{ }}) in templates.

Example

Urls.py

```
urlpatterns = [
    path('filterdemo/',views.myfun3)
]
```

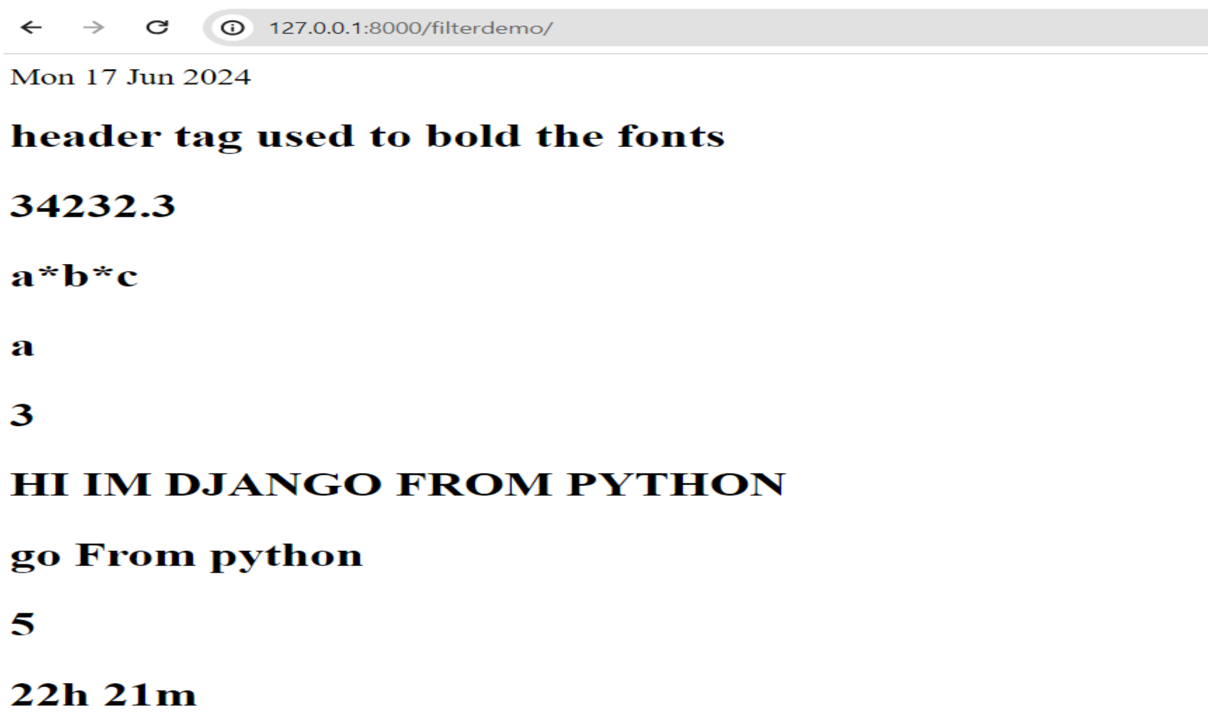
Views.py

```python
def myfun3(request):
    value=datetime.now()
    context = {
      'value':value,
       'input':'<h2> header tag used to bold the fonts <h2>',
       'fnumber':34232.34,
       'List':['a','b','c'],
       'text':'Hi Im Django From python'
    }
    return render(request, 'home3.html', context)
```

Template (HTML file)

```html
<html>
<body>
  <p>{{ value|date:"D d M Y" }}</p>
  <p>{{ input|safe}}</p>
  <p>{{ fnumber | floatformat}}</p>
  <p>{{ List | join:"*"}}</p>
  <p>{{List|first}}</p>
  <p>{{List|length}}</p>
  <p>{{text|upper}}</p>
  <p>{{text|slice:"10:"}}</p>
  <p>{{text|wordcount}}</p>
  {{ value|time:"H\h i\m" }}
</body>
</html>
```

OUTPUT


127.0.0.1:8000/filterdemo/

```
Mon 17 Jun 2024
header tag used to bold the fonts
34232.3
a*b*c
a
3
HI IM DJANGO FROM PYTHON
go From python
5
22h 21m
```

Explination

date:

- **Purpose**: Formats a date according to the given format string.
- **Usage**: {{ value|date:"D d M Y" }}
- **Details**: Converts the date into a specific format, where D is the abbreviated weekday name, d is the day of the month, M is the abbreviated month name, and Y is the year.

safe:

- **Purpose**: Marks a string as safe for HTML output, preventing it from being auto-escaped.
- **Usage**: {{ input|safe }}

- **Details**: Ensures that the input string is rendered as raw HTML, bypassing Django's automatic escaping mechanism.

floatformat:

- **Purpose**: Formats a floating-point number to a specified number of decimal places.
- **Usage**: {{ fnumber|floatformat }}
- **Details**: By default, it rounds the number to one decimal place. Additional arguments can specify the number of decimal places or if the number should be trimmed of unnecessary zeros.

join:

- **Purpose**: Joins a list into a string with a specified separator.
- **Usage**: {{ List|join:"*" }}
- **Details**: Converts the list into a single string with each element separated by the specified character(s), in this case, an asterisk (*).

first:

- **Purpose**: Returns the first item in a list.
- **Usage**: {{ List|first }}
- **Details**: Retrieves and displays the first element of the given list.

length:

- **Purpose**: Returns the length of a list, string, or any iterable.
- **Usage**: {{ List|length }}
- **Details**: Calculates and displays the number of items in the list or characters in a string.

upper:

- **Purpose**: Converts a string to uppercase.
- **Usage**: {{ text|upper }}
- **Details**: Transforms all the characters in the given string to their uppercase equivalents.

slice:

- **Purpose**: Extracts a substring using Python's list slicing syntax.
- **Usage**: {{ text |slice:"10:" }}
- **Details**: Retrieves a substring starting from the 10th character to the end of the string.

wordcount:

- **Purpose**: Counts the number of words in a string.
- **Usage**: {{ text|wordcount }}
- **Details**: Calculates and displays the total number of words in the given string.

time:

- **Purpose**: Formats a time according to the given format string.
- **Usage**: {{ value|time:"H\\h i\\m" }}
- **Details**: Converts the time into a specific format, where H is the hour, i is the minute, and the backslashes (\\) are used to escape characters, so H\h i\m will display as hours and minutes with h and m literals.

For more tags and filters follow the below link

docs.djangoproject.com › en › 5Built-in template tags and filters | Django documentation

4. Template Loading

Using Template in the view function

Current State:

The example shows a current_datetime view that displays the current date and time. However, the template logic is currently embedded within the Python code using Template and Context classes.

```
from django. http import HttpResponse
from datetime import datetime
from django. template import Template, Context



def current_datetime(request):
    now = datetime.datetime.now ()
    t = Template("<html><body>It is now {{current_date}}.
</body></html>")
    html = t. render(Context({'current_date': now}))
    return HttpResponse(html)
```

Problems:

Hardcoded Template: The template content is directly written in the code, making it difficult to manage and update separately.

Missing File Handling: The code doesn't handle the case where the template file might be missing or inaccessible.

```
def current_datetime(request):
    now = datetime.datetime.now()
    fp = open('/home/djangouser/templates/mytemplate.html')
    t = Template(fp.read())
    fp.close()
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

Repetition: If you use templates in multiple views, you'd have to repeat the template loading logic, leading to code duplication.

To address these issues, Django provides a more robust way to work with templates

Separate Template Files: Store your templates in dedicated files (e.g., .html files) within your project's directory structure.

Template Directories: Configure Django to know where to look for these template files by specifying template directories in your project's settings.

Template Loading: Use Django's template loader to load templates from the configured directories, eliminating the need for manual file handling.

This approach offers several benefits:

Maintainability: Templates are separate from code, making them easier to manage and update.

Error Handling: Django handles missing template files gracefully, raising exceptions.

Code Reusability: You can reuse the template loading logic across multiple views.

The next sections of the chapter will likely delve deeper into template loading and directory configuration in Django.

Template Loading

Django provides a convenient and powerfull way to load template with the goal of removing redundancy for both templates and template calling

1. Telling Django Where to Find Templates:

You configure Django's settings file (usually settings.py) with the TEMPLATE_DIRS setting.

This setting specifies directories where Django will look for your template files (e.g., HTML).

2. Using get_template():

Instead of manually opening and reading template files, you use the django.template.loader.get_template() function.

You provide the template's name as an argument.

get_template() searches through the directories listed in TEMPLATE_DIRS to find the matching file.

3. Advantages:

Removes redundancy in your code (no need to repeat template paths).

```
def myfun(request):
   person_data={'name':'deepa','age':'23','hobbies':['coding','music
']}
   today=date.today()
   template_path = os.path.join(BASE_DIR, 'templates',
'my_template.html')
   context = {'person': person_data,'today':today}
   return render(request, template_path)
```

Easier to manage template locations.

Django handles errors gracefully (raises TemplateDoesNotExist if the template is missing).

Example:

Set TEMPLATE_DIRS in settings.py to point to your template directory (e.g., /User/desktop/django_project/mysite/templates).

In your view function, use get_template('current_datetime.html') to get the template object.

```
def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

Error Handling:

Django's development server shows a detailed error page if a template is missing, including which directories were searched.

This helps you identify typos or incorrect directory paths.

TEMPLATE_DIRS tells Django where to look for templates.

get_template() helps you load templates efficiently and handles errors.

This approach keeps your code clean and organized.

rendor_to_response()

What it Does:

Combines several steps into one line of code:

Loading a template

Creating a context dictionary

Rendering the template with the context

Creating an HTTP response object with the rendered content

Uses

More concise and readable code compared to manual steps.

Less code to write and maintain.

How it Works:

Import render_to_response from django.shortcuts.

In your view function:

Calculate any needed data (like the current date and time in the example).

Call render_to_response with two arguments:

The name of the template to use (e.g., 'current_datetime.html').

A dictionary containing data to pass to the template (optional, defaults to an empty dictionary).

The function returns an HTTP response object ready to be sent back to the user.

Example

```
from django.shortcuts import render_to_response,render
from django.http import HttpResponse
from datetime import date

def myfun(request):
   person_data={'name':'deepa','age':'23','hobbies':['coding','music
']}
   today=date.today()
   context = {'person': person_data,'today':today}
   return render(request, 'home.html', context)
```

```
def myfun(request):
   person_data={'name':'deepa','age':'23','hobbies':['coding','music
']}
   today=date.today()
   return render_to_response('home.html', context = {'person':
person_data,'today':today}  )
```

The locals() Trick

Locals() a shortcut for passing data to templates in Django, but it's generally not recommended. Here's a breakdown:

The Problem:

- Often, you need to calculate values (like the current date) and pass them to templates using a context dictionary.
- This involves creating temporary variables and then manually adding them to the dictionary, which can be repetitive.

The locals() Trick (Not Recommended):

- Python's locals() function creates a dictionary containing all local variables in a function's scope.
- You can use locals() to avoid manually creating the context dictionary.

Example (Not Recommended):

```
def current_datetime(request):
  current_date = datetime.datetime.now()
  return render_to_response('home.html', locals())


def current_datetime(request):
     current_date = datetime.datetime.now()
     return HttpResponse(f"Current date and time: {current_date}")
```

Why It's Not Recommended:

- **Security:** The template might access unintended variables present in locals().

```
def my_view(request):
  data_to_display = datetime.datetime.now()
  hidden_variable = "This shouldn't be in the template"
  return render_to_response('my_template.html', locals())
```

- **Maintainability:** The template becomes dependent on the specific variable names used in the view.
- **Performance:** locals() creates a dictionary on every function call, which can add a small overhead.

Better Approach:

- Create the context dictionary explicitly, specifying only the variables you want to pass to the template.
- This improves security, maintainability, and performance.

The {% include %} Template Tag

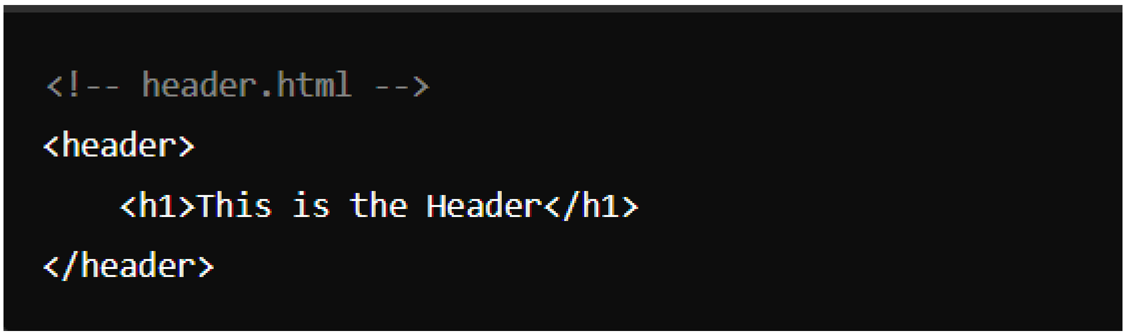
The {% include %} template tag in Django is a powerful tool used for including the contents of one template within another. This tag helps to reduce code duplication by allowing common code to be placed in a single template and included wherever needed.

- **Purpose**: To include the contents of another template within the current template.
- **Syntax**: {% include 'template_name.html' %}

Example

1. header.html

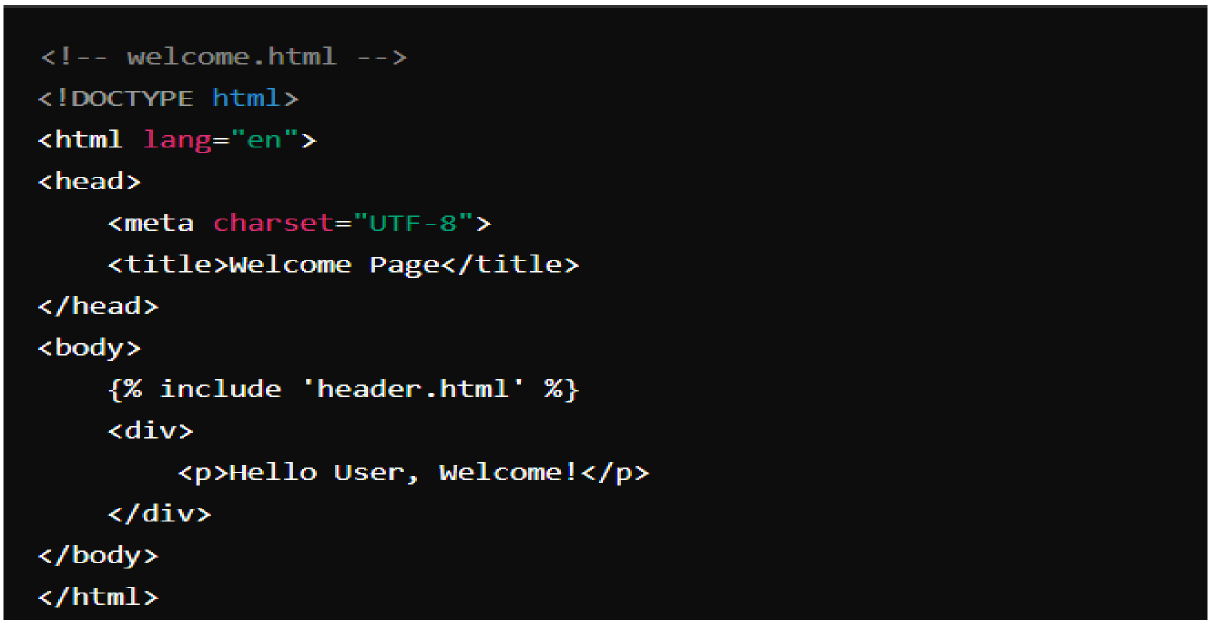
This template will contain the header content.



```
<!-- header.html -->
<header>
    <h1>This is the Header</h1>
</header>
```

2. welcome.html

This template will include the `header.html` content and also contain its own content.



```
<!-- welcome.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Welcome Page</title>
</head>
<body>
    {% include 'header.html' %}
    <div>
        <p>Hello User, Welcome!</p>
    </div>
</body>
</html>
```

Explanation

- **header.html**:
    - Contains the header content with an <h1> tag.
- **welcome.html**:
    - This is the main template.
    - It includes the content of header.html using the {% include 'header.html' %} tag.
    - It also contains a welcome message.

OUTPUT

```
This is the Header

Hello user, welcome!
```

5. Template Inheritance

Concept and Purpose

Template inheritance in Django is a mechanism that allows you to build a base "skeleton" template containing all the common parts of your site. This base template defines "blocks" that child templates can override, add to, or leave alone. This approach reduces redundancy and duplication of common page areas such as sitewide navigation, headers, footers, and other recurring elements.

How It Works

1. **Base Template**: You start by creating a base template that includes the common structure and elements of your site. This template contains {% block %} tags, which define placeholders that child templates can fill in.

    - **Example**: A base template might include common elements like the <head> section, a navigation bar, and footer content.
2. **Child Templates**: These templates extend the base template using the {% extends %} tag. They provide specific content for the placeholders defined by the {% block %} tags in the base template.
    - **Example**: A child template for a specific view or page will extend the base template and define content for specific blocks like the page title, main content area, etc.
3. **Block Tag**: The {% block %} tag in the base template specifies sections that can be overridden by child templates. The content within these tags in the base template acts as a default, which can be replaced or extended in the child templates.
4. **Extends Tag**: The {% extends %} tag in a child template indicates that it is based on a parent template. This tag must be the first tag in the child template for inheritance to work properly.

Advantages

1. **Reduces Redundancy**: By having a base template, you avoid repeating the same HTML structure in multiple templates. Changes to the common structure need to be made only in the base template.
2. **Organizes Code**: It keeps your templates organized, making it easier to manage and maintain. Each template contains only the code unique to that template.
3. **Centralizes Changes**: Sitewide changes can be made in the base template and will automatically be reflected across all pages that extend it.

Key Points

- **Block Tags**: Define sections in the base template that child templates can override.
- **Extends Tag**: Indicates that a template is based on another template.
- **Dynamic Inheritance**: The argument to {% extends %} can be a string or a variable, allowing for dynamic template inheritance.
- **Fallback Content**: If a child template does not define a block, the content from the parent template is used.
- **No Multiple Block Names**: You cannot define multiple {% block %} tags with the same name within the same template.
- **Template Loading**: The template name passed to {% extends %} is loaded using the same method as get_template(), appending to the TEMPLATE_DIRS setting.

Example

Urls.py

```
urlpatterns = [

   path('about/',views.about),
   path('home/',views.home),
   path('contact/',views.contact)
]
```

Views.py

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

Layout.html (parent template)

```
<html>
<body>
<nav>
<a href="/home/">Home</a>|
<a href="/about/">About Us</a>|
<a href="/contact/">Contact Us</a>|
</nav>

{% block content %}

{% endblock %}

<footer>
<hr>
&copy; 2024 Mycem Developed by digital india
</footer>
</body>
</html>
```

about.html

```
{% extends 'layout.html' %}

{% block content %}
<h2>we are Django developers</h2>
{% endblock %}
```

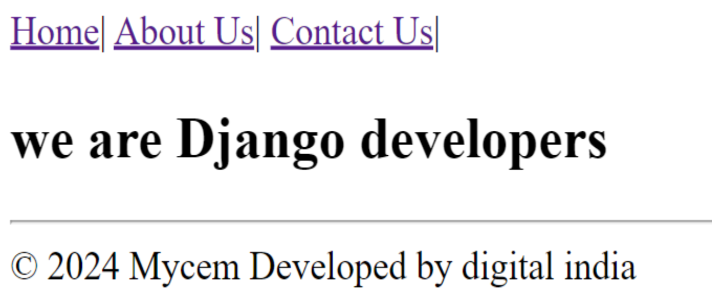
Contact.html

```
{% extends 'layout.html' %}

{% block content %}
<h2>contact us mycem.edu.in</h2>
{% endblock %}
```

Home.html

```
{% extends 'layout.html' %}

{% block content %}
<h2>This is the home page</h2>
{% endblock %}
```

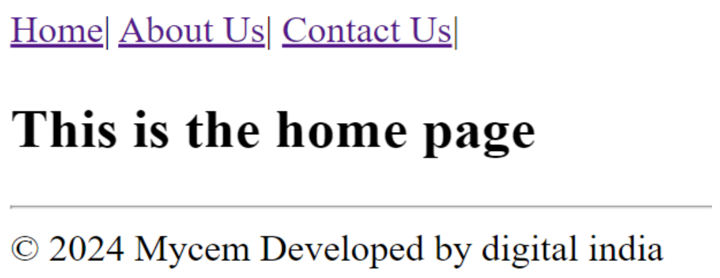
OUTPUT



Home| About Us| Contact Us|

we are Django developers

© 2024 Mycem Developed by digital india



Home| About Us| Contact Us|

This is the home page

© 2024 Mycem Developed by digital india

Home| About Us| Contact Us|

contact us mycem.edu.in

© 2024 Mycem Developed by digital india