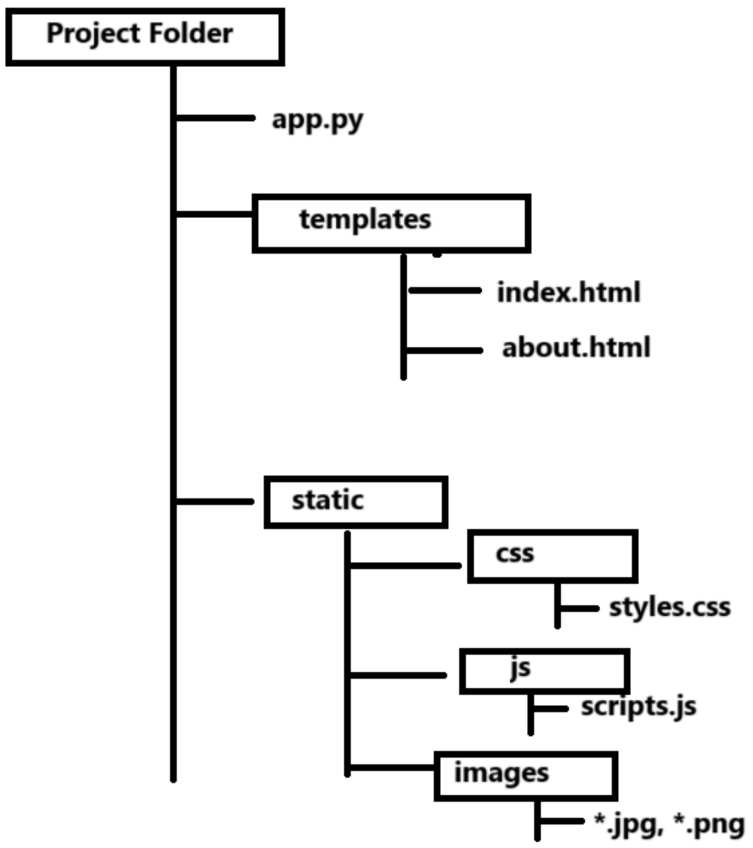
## Working with Templates inside the flask Application:

- So far we have only returned back HTML manually through a Python string.
- In real time we will want to connect a view function to render HTML templates(Already created HTML files for our webpages).
- Flask will automatically look for HTML templates(HTML files) inside the **templates** folder of our application.
- We can render templates simply by importing the **render_template()** function from flask and returning a **.html** file from our view function.

### Why Use Templates?

1. **Separation of Concerns**: Templates allow you to keep your HTML and Python code separate(separate the presentation layer from the application logic, making your application easier to maintain.
2. **Reusability**: Templates can be reused across multiple pages, reducing duplication.
3. **Dynamic Content**: Templates enable you to insert dynamic data into HTML files.

## Folder Structure for using Templates in Flask Application:



**Example:**

**step1:** Create a folder called **templates** inside the project_folder

**step2:** Create an **index.html** file inside this **templates** folder as follows:

**index.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>

    <h1>Welcome to Chitkara</h1>

</body>

</html>
```

**step3:** import the render_template() function and using this function render the above created html page as follows.

**app.py:**

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

if __name__ == '__main__':
    app.run(debug=True)
```

**step4:** launch the application and type the following inside the URL bar of browser:

http://127.0.0.1:5000/

Lets try to keep an image inside our application by creating a **static folder** and under this create an **images** folder. (refer the above folder structure)

- modify the above **index.html** file as follows:

**index.html:**

```
<!DOCTYPE html>
```

```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>

    <h1>Welcome to Chitkara</h1>

    <img src="./static/images/sh1.jpg" height="200px" width="200px">

</body>

</html>
```

**Note: We should not navigate from one html to another html file directly, That navigation should happen through the Flask view function only.**

**Example2:**

**Project Folder Structure**

```
Flask_App2
        |--app.py
        |--templates
                |--index.html
                |--contact.html
                |--about.html
```

app.py:

```python
from flask import Flask, render_template

app = Flask(__name__)


@app.route("/")

def home():

    return render_template("index.html")



@app.route("/contact")

def contact():

    return render_template("contact.html")



@app.route("/about")

def about():

    return render_template("about.html")



if __name__ == "__main__":

    app.run(debug=True)
```

index.html:

```
<body>
    <h1>Welcome to Chitkara</h1>
    <h1>This is the home page</h1>
    <a href="/about">About Page</a>
    <a href="/contact">Contact Page</a>
</body>
```

about.html:

```
<body>
    <h1>This is About page</h1>
    <a href="/">Back to Home Page</a>
</body>
```

contact.html:

```
<body>
    <h1>This is the contact page</h1>
    <a href="/">Back to Home Page</a>
</body>
```

## Template Variables:

- Using the **render_template()** function we can directly render an HTML file with our flask web app.
- But we haven't leveraged the power of Python at all yet!.
- We want a way to be able to use Python code in our app, changing and updating variables and logic. and then send that information to the HTML **template.**
- We can use the **Jinja template engine** to do this.
- **Jinja** templating will let us directly insert variables from our Python code to the HTML file.
- The syntax for inserting a variable is:
  - {{ variable_name }}
- We can pass in Python strings, lists, dictionaries, and more into the templates.
- We can set parameters (of our choice) inside the **render_template()** function and then use the {{ }} syntax to insert them in the template.

**Example:**

**app.py:**

```
from flask import Flask, render_template

app = Flask(__name__)


@app.route("/")

def index():

    some_variable = "Chitkara"

    return render_template("index.html",
my_variable=some_variable)
```

```python
if __name__ == '__main__':
    app.run(debug=True)
```

index.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>
    <h1>Welcome to {{ my_variable }}</h1>
</body>
</html>
```

Example2:

app.py:

```python
@app.route("/")
def index():
    name = "Ram"
    letters = list(name)
    student = {"roll": 10, "name": "Rahul", "marks": 800 }
```

```
        return render_template("index.html", name=name,
        letters=letters, student = student)
```

index.html:

```
<body>

    <h1>Welcome {{ name }}</h1>

    <h2>List is {{ letters }}</h2>

    <h2>Using Expression 1: {{ letters[0] }}</h2>

    <h2>Using Expression 2: {{ letters[:2] }}</h2>

    <h2>Using Expression 3: {{10 + 20}}</h2>

    <h2>Getting dictionary: {{ student }}</h2>

    <h2>Getting dictionary value: {{ student[“name”] }}</h2>

</body>
```

## Template Control Flow:

- With **Jinja** templating we can also have access to the control flow syntax in our template such as **for loop**, and **if-statements**.
- The syntax is as follows:
    - {% %}
- Imagine we passed a list variable to the HTML using a template variable.
- Instead of displaying the entire list at once, we can display each item in the Python list as a bulleted HTML list.

Syntax:

```
 <ul>
```

```
        {% for item in mylist %}

            <li>{{item}}</li>

        {% endfor %}

    </ul>
```

Example:

app.py

```
@app.route("/")
def index():
    cities = ["Delhi", "mumbai", "Chennai", "Kolkata"]
    return render_template("index.html", cities=cities)
```

index.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>

    <h1>Welcome to Chitkara</h1>

    <ul>

        {% for city in cities%}
```

```
            <li>{{city}}</li>

        {% endfor %}

    </ul>

</body>



</html>
```

## Using Conditional Statements:

Syntax:

```
            {% if condition %}

                        <p>If condition matches</p>

            {% elif other_condition %}

                        <p>Some other condition matches</p>

            {% else %}

                        <p>None of the condition matches</p>

           {% endif %}
```

Example:

index.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>

    <h1>Welcome to Chitkara</h1>

    {% if "Mumbai" in cities %}

                        <p>Mumbai is the Financial Capital in India</p>

    {% elif "Delhi" in cities %}

                        <p>Delhi is the Capital of India</p>

    {% else %}

                        <p>No Capital found in the Cities</p>

    {% endif %}

</body>

</html>
```

Example2:

app.py

```
@app.route("/")
 def index():
    user_logged_in = False
    return render_template("index.html", user_logged_in=user_logged_in)
```

index.html:

```
<body>
    <h1>Welcome to Chitkara</h1>


            {% if user_logged_in %}
                        <h2>Welcome Back, You are now logged In!</h2>
            {% else %}
                        <h2>Please Log in!</h2>
            {% endif %}
</body>
```

### Example3: Printing a table

**app.py:**

```
@app.route("/")
def index():
    num = 5
    return render_template("index.html", num=num)
```

index.html:

```
<body>
```

```
    <h1>Welcome to Chitkara</h1>

    <h1>Multiplication Table for {{ num }}</h1>



    {% for i in range(1, 11) %}

    <p>{{ num }} * {{ i }} = {{ num * i }}</p>

    {% endfor %}



</body>
```

## Exercise:

For the following list of Products, display them inside a table

```
 products = [

        {"id": 1, "name": "Pen", "price": 50},

        {"id": 2, "name": "Pencil", "price": 80},

        {"id": 3, "name": "Sharper", "price": 90},

        {"id": 4, "name": "Eraser", "price": 12}

    ]
```

Solution:

app.py

```
@app.route("/")
def index():
    products = [

        {"id": 1, "name": "Pen", "price": 50},
        {"id": 2, "name": "Pencil", "price": 80},
        {"id": 3, "name": "Sharper", "price": 90},
        {"id": 4, "name": "Eraser", "price": 12}
    ]

    return render_template("index.html", products=products)
```

index.html:

```
<body>

    <h1>Welcome to Chitkara</h1>

    <h1 style="text-align: center;">Product List</h1>

    <table class="table">

        <thead>

            <tr>

                <th>ID</th>

                <th>Name</th>

                <th>Price (INR)</th>

            </tr>

        </thead>

        <tbody>

            {% for product in products %}

            <tr>

                <td>{{ product.id }}</td>
```

```
          <td>{{ product.name }}</td>

          <td>{{ product.price }}</td>

        </tr>

        {% endfor %}

      </tbody>

    </table>

</body>

</html>
```

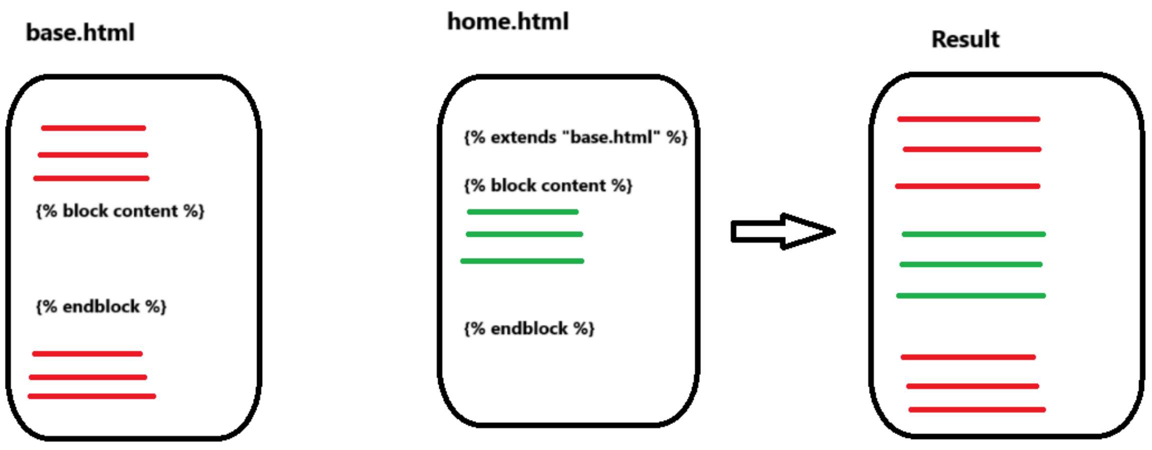
## Template Inheritance in Flask:

- Template inheritance is a powerful feature in Flask that allows you to create a consistent layout for your web application. It works similarly to the inheritance concept in object-oriented programming, where a child template inherits from a parent template.
- **Template inheritance** in Flask allows you to create a base structure (or layout) for your application and reuse it across multiple templates. This reduces code duplication, ensures consistency, and makes your application easier to maintain.
- Usually pages across a web application shares a lots of features (e.g. header, navigation bar, footer, etc)

## How Template Inheritance Works

1. **Base Template**:
    - Define a base template (e.g., `base.html`) with common elements like headers, footers, and navigation menus.

    - Use **block tags** (`{% block block_name %}{% endblock %}`) to define sections that child templates can override or extend. This Sections of the template that can be replaced or filled by child templates.
2. **Child Templates**:
    - Use the `{% extends "base.html" %}` directive to inherit the structure of the base template.
    - Override or add content to specific blocks in the base template.



## Why Use Template Inheritance?

1. **Consistency**: Ensures that common elements like headers, footers, and navigation menus are consistent across all pages.
2. **Reusability**: Allows you to define shared structures in a single template, reducing code duplication.
3. **Maintainability**: Updating shared elements in one place automatically reflects on all pages that inherit from the base template.

**Example:**

### folder structure:

```
project_folder
    ├── app.py
    ├── templates
```

```
│           ├── base.html
│           ├── home.html
│           ├── about.html
│           ├── contact.html
```

app.py:

```python
from flask import Flask, render_template

app = Flask(__name__)


@app.route('/')
def home():
    return render_template('home.html')

@app.route('/about')
def about():
    return render_template('about.html')

@app.route("/contact")
def contact():
    return render_template("contact.html")



if __name__ == '__main__':
    app.run(debug=True)
```

**<u>base.html:</u>**

- The base template contains the overall structure, including a header, footer, and placeholders (blocks) for dynamic content.

```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>

    <style>

        main {

            height: 40vh;

        }

    </style>

</head>


<body bgcolor="pink">

    <header>

        <h1>Welcome to My Flask App</h1>

        <nav>

            <ul>

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

                <li><a href="/about">About</a></li>

                <li><a href="/contact">Contact</a></li>

            </ul>

        </nav>
    </header>



    <main style="background-color: antiquewhite;">
        {% block content %}

        {% endblock %}
    </main>

    <footer>
        <p>&copy; 2025 Flask App</p>
    </footer>
</body>
</html>
```

**Key Points:**

- {% block title %} and {% block content %} are placeholders that child templates can override.
- The header and footer are common across all pages.

**Child Template: home.html:**

- The home page extends the base template and adds specific content blocks.

```
{% extends "base.html" %}

{% block content %}

    <h2>Home Page</h2>

    <p>Welcome to the home page of our Flask application!</p>

{% endblock %}
```

**Child Template: about.html:**

- The about page also extends the base template and customizes the content blocks.

```
{% extends "base.html" %}

{% block content %}

    <h2>About Us</h2>

    <p>This Flask application is an example of template inheritance About Us page.</p>

{% endblock %}
```

**Child Template: contact.html:**

- The about page also extends the base template and customizes the content blocks.

```
{% extends "base.html" %}

{% block content %}

    <h2>Contact Page</h2>

    <p>This Flask application is an example of template inheritance. Contact Page </p>
```

{% endblock %}

## Example2: Using Bootstrap:

app.py:

- From previous example

base.html:

```
<!DOCTYPE html>

<html lang="en">


<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width,
initial-scale=1.0">

    <title>{% block title %}Flask Bootstrap App{% endblock %}</title>

    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css/bootstrap.m
in.css" rel="stylesheet">

    <link rel="stylesheet" href="{{ url_for('static',
filename='css/styles.css') }}">

</head>


<body>

    <!-- Navigation Bar -->
```

```html
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">

    <div class="container">

        <a class="navbar-brand" href="/">My Flask App</a>


        <div class="collapse navbar-collapse" id="navbarNav">

            <ul class="navbar-nav ms-auto">

                <li class="nav-item">

                    <a class="nav-link" href="/">Home</a>

                </li>

                <li class="nav-item">

                    <a class="nav-link" href="/about">About</a>

                </li>

                <li class="nav-item">

                    <a class="nav-link" href="/contact">Contact</a>

                </li>


            </ul>

        </div>

    </div>

</nav>


<!-- Main Content -->

<div class="container my-5">

    {% block content %}{% endblock %}

</div>
```

```html
    <!-- Footer -->

    <footer class="bg-dark text-white text-center py-3">

        <p>&copy; 2025 My Flask App</p>

    </footer>


</body>


</html>
```

home.html:

```html
{% extends "base.html" %}


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


{% block content %}

<div class="text-center">

    <h1>Welcome to My Flask App</h1>

    <p class="lead">This is the home page. Explore the website
using the navigation bar.</p>

    <a href="/about" class="btn btn-primary">Learn More</a>

</div>

{% endblock %}
```

about.html:

```
{% extends "base.html" %}


{% block title %}About{% endblock %}


{% block content %}

<div class="text-center">

    <h1>About Us</h1>

    <p class="lead">This Flask application demonstrates the use of
template inheritance with Bootstrap.</p>

    <a href="/" class="btn btn-secondary">Go Back</a>

</div>

{% endblock %}
```

<u>contact.html:</u>

```
{% extends "base.html" %}

{% block title %}Contact{% endblock %}

{% block content %}

<div class="text-center">

    <h1>Contact Page</h1>

    <p class="lead">This Flask application demonstrates the use of
template inheritance with Bootstrap.</p>

    <a href="/" class="btn btn-secondary">Go Back</a>

</div>

{% endblock %}
```

**Note:** We can define multiple blocks inside any section with different names, and also we can access a parent template variable to its child template or variables passed from the child can be accessible from its parent also.

**Example:**

**folder structure:**

```
project_folder
        |--app.py
        |--templates
                |--index.html (base html)
                |--contact.html
                |--about.html
```

app.py:

```
from flask import Flask, render_template

app = Flask(__name__)


@app.route('/')

def index():

    return render_template("index.html")


@app.route('/about')

def about():

    return render_template('about.html', page_name="About")


@app.route("/contact")
```

```
    def contact():

        return render_template("contact.html", page_name="Contact")



    if __name__ == '__main__':

        app.run(debug=True)
```

<u>index.html:</u>

```
<!DOCTYPE html>

<html lang="en">



<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width,
initial-scale=1.0">

    <title>{{ page_name }}</title>

    <style>

        html,

        body {

            margin: 0;

            padding: 0;

            height: 100vh;

            display: flex;

            flex-direction: column;

        }
```

```css
header {

    height: 10vh;

    background-color: antiquewhite;

    text-align: center;

    display: flex;

    align-items: center;

    justify-content: space-between;

    padding: 0 10px;

}


main {

    height: 80vh;

    background-color: aquamarine;

    display: flex;

    align-items: center;

    justify-content: center;

}


footer {

    height: 10vh;

    background-color: skyblue;

    display: flex;

    align-items: center;

    justify-content: center;

}
```

```
        nav ul {
            list-style-type: none;
            display: flex;
            gap: 15px;
        }



        nav ul li a {
            text-decoration: none;
            color: black;
            font-weight: bold;
        }

        nav ul li a:hover {
            text-decoration: underline;
        }
    </style>

    {% block style %}
    {% endblock %}
</head>

<body>
    <header>
```

```
        <h1>Welcome to Chitkara</h1>

        <nav>

            <ul>

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

                <li><a href="/about">About</a></li>

                <li><a href="/contact">Contact</a></li>

            </ul>

        </nav>

    </header>



    <main>

        {% block main %}

        {% endblock %}

    </main>



    <footer>

        <h2>&copy; 2025 Flask Application</h2>

    </footer>

</body>



</html>
```

contact.html

```
      {% extends "index.html" %}
```

```
{% block title %}Contact{% endblock %}

{% block style %}
<style>
    main {
        background-color: pink;
    }
</style>
{% endblock %}

{% block main %}
<h1>This is the Contact Page</h1>

{% endblock %}
```

<u>about.html:</u>

```
{% extends "index.html" %}

{% block title %}About{% endblock %}

{% block style %}
<style>
    main {
        background-color: green;
    }
```

```
</style>

{% endblock %}


{% block main %}

<h1>This is the About Page </h1>


{% endblock %}
```

## Using filters inside the Flask Application:

- Using filters we can quickly change/edit a variable passed to a template.
- In Flask templates, filters are a way to transform the output of variables before rendering them in the HTML. Filters are a feature of Jinja2, the templating engine used by Flask. They are applied to variables using the pipe `|` symbol.
- Syntax:
  - `{{variable | filter}}`
- Example:
  - `{{ name }}    # simran`
  - `{{name | capitalize}}  #Simran`

### Common Use Cases for Filters

1. Formatting text (e.g., converting to uppercase or lowercase).
2. Manipulating lists, strings, or numbers.
3. Formatting dates and numbers.

## Built-In Jinja2 Filters

Here are some commonly used filters:

- `lower`: Converts a string to lowercase.
- `upper`: Converts a string to uppercase.
- `capitalize`: Capitalizes the first letter of a string.
- `title`: Capitalizes the first letter of each word in a string.
- `length`: Returns the length of a string, list, or dictionary.
- `default`: Provides a default value if the variable is undefined.
- `replace`: Replaces parts of a string with another string.
- `join`: Joins a list into a single string with a specified delimiter.
- `truncate`: Truncates a string to a specified length.

### Example of Using Filters:

**app.py:**

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    data = {
        "name": 'flask application',
        "fruits": ['apple', 'banana', 'cherry'],
        "price": 1250.50,
    }
    return render_template('index.html', data=data)
```

```python
if __name__ == '__main__':
    app.run(debug=True)
```

index.html:

```html
<!DOCTYPE html>

<html lang="en">


<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width,
initial-scale=1.0">

    <title>Using Filters</title>

</head>


<body>

    <h1>Filters in Flask Templates</h1>

    <p>Original Name: {{ data.name }}</p>

    <p>Uppercase Name: {{ data.name | upper }}</p>

    <p>Capitalized Name: {{ data.name | capitalize }}</p>

    <p>Title Case Name: {{ data.name | title }}</p>


    <hr>


    <p>Fruits: {{ data.fruits | join(', ') }}</p>
    <p>Number of Fruits: {{ data.fruits | length }}</p>


    <hr>


    <p>Price: {{ data.price }}</p>

    <p>Formatted Price: {{ "%.2f" | format(data.price) }}</p>


    <hr>


    <p>Default Value Example: {{ data.undefined_variable | default('No
Value Provided') }}</p>

</body>


</html>
```

## Custom Filters:

- You can also create your own filters in Flask.
- Example: Custom Filter to Reverse a String
    - Inside **app.py** file

    ```
    @app.template_filter("reverse")
    def reverse_filter(s):
        return s[::-1]
    ```

- Use in **index.html**:
    - `<p>Reversed Name: {{ data.name | reverse }}</p>`

### Exercises

1. Use the `truncate` filter to limit a description to 50 characters.
2. Create a custom filter to check if a number is even or odd.
3. Apply the `replace` filter to replace spaces in a string with hyphens (`-`).

## URL links with templates:

- `url_for()` is one of the most powerful and convenient functions in Flask, enabling dynamic URL generation. It helps avoid hardcoding URLs in templates and Python code, making your application more flexible, maintainable, and scalable.

### Key Features of `url_for()`

1. **Dynamic URL Generation:**
   - Generates URLs dynamically by using the name of the view function.
   - This helps in generating URLs without having to hardcode the paths, which makes your code maintainable as the routes evolve.
2. **Handles Parameters:**
   - Supports dynamic route parameters (e.g., `/<username>`), which means URLs can be constructed with variables or dynamic segments.
   - It also works with query strings like `/user?name=John`.
3. **Easily Integrates with Static Files:**

    - Helps generate URLs for static resources such as CSS, JavaScript, images, etc., through `url_for('static', filename='...')`.
4. **Error Prevention:**
    - Avoids hardcoding of URL paths, ensuring they will work even if route names change. It ensures that if a route changes, the URL generation will still work without further code modification.

---

### Basic Syntax:

```
url_for(endpoint, **values)
```

- **`endpoint`**: The name of the view function (string).
- **`values`**: Key-value pairs used for URL parameters or query strings (for dynamic parts in routes).

### Example 1: Generating URLs for Routes

- In this example, `url_for()` is used to generate the URL for the routes dynamically based on their view function names.

**app.py:**

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/contact')
def contact():
    return render_template("contact.html")

if __name__ == '__main__':
    app.run(debug=True)
```

**index.html:**

```
<body>

    <h1>Welcome to Chitkara</h1>

    <ul>

        <li><a href="{{ url_for('home') }}">Home</a></li>

        <li><a href="{{ url_for('contact') }}">Contact</a></li>

    </ul>

</body>

</html>
```

- **The above url_for function will generate the dynamic routes as follows:**
    - `<a href="/">Home</a>`
    - `<a href="/contact">Contact</a>`

**contact.html:**

```
<body>

    <h1>This is the Contact.html file</h1>

</body>
```

**Example 2:** Sending the request parameter using **url_for()** function

- In this example, the `url_for()` function generates URLs that include dynamic route parameters.

<u>**app.py:**</u>

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/contact')
def contact():
    return render_template("contact.html")

@app.route('/user/<name>')
def user_profile(name):
    return render_template("user.html", username=name)

if __name__ == '__main__':
    app.run(debug=True)
```

<u>**user.html:**</u>

```
<body>

    <h1>Welcome {{username}}</h1>

</body>



</html>
```

**index.html:**

```html
<body>

  <h1>Welcome to Chitkara</h1>

  <ul>

    <li><a href="{{ url_for('home') }}">Home</a></li>

    <li><a href="{{ url_for('contact') }}">Contact</a></li>

   <li><a href="{{ url_for('user_profile', name='Rahul') }}">User Profile</a></li>

  </ul>

</body>
```

- This Generates:
  - `<a href="/user/Rahul">User Profile</a>`
- This dynamic URL is created using the view function name (user_profile) and the name parameter.

**Example 3: Using url_for() in Redirects:**

- In Flask, you can use `url_for()` for redirecting to another route. This is useful to create clean, maintainable redirects.

**app.py:**

```python
from flask import Flask, render_template, url_for, redirect
```

```
app = Flask(__name__)

@app.route('/')
def home():
    #return redirect('/new_home')
    return redirect(url_for('new_route'))

@app.route('/new_home')
def new_route():
      # url = url_for("new_route", _external=True)

      # return render_template("index.html", url=url)

      return render_template("index.html")

if __name__ == '__main__':
    app.run(debug=True)
```

- This will redirect users from the `/` route to the `/new_home` route dynamically.

<u>index.html:</u>

```
<body>

    <h1>Welcome to Chitkara</h1>

</body>
```

**Example 4: Linking Static Files using url_for()**

- Flask's `url_for()` can also be used to link static files such as CSS, JavaScript, and images. This ensures that static files are linked properly and will be correctly served by Flask.

- Use url_for('static', filename='path/to/file') to link static files.

```
<link rel="stylesheet" href="{{ url_for('static', filename='css/styles.css') }}">

<img src="{{ url_for('static', filename='images/sh1.jpg') }}" alt="Logo" height="200px"
width="200px">
```

### Advantages of `url_for()`

1. **Avoids Hardcoding of URLs:**
   - Ensures that your application remains maintainable as the route paths may change over time. You won't need to manually update URLs in multiple places.
2. **Supports Subdomains and Configurations:**
   - `url_for()` respects Flask configurations such as `SCRIPT_NAME` or subdomains, ensuring URLs are generated based on your app's configuration.
3. **Prevents Errors from Changed Routes:**
   - If you change a route or its parameters, `url_for()` will automatically adapt and generate the correct URL for that route.
4. **Dynamic URL Parameters:**
   - Easily manage dynamic URL segments like user profiles or product details, improving flexibility in how URLs are structured.

## Additional Notes about the url_for() function:

- **Query Parameters**: If your route includes query parameters, you can pass them in url_for() like this:

**Example:**

**url_for('user_profile', name='Rahul', age=25)**

- This will generate a URL like /user/Rahul?age=25.
- Changing the templates and static folder name

```
app = Flask(__name__, static_folder="assets",template_folder="htmls")
```

## Handling Errors in Flask: @app.errorhandler

- In Flask, you can define custom error pages for different HTTP status codes using the @app.errorhandler decorator. This allows you to create user-friendly pages when certain errors occur (like a 404 error for "Page Not Found" or a 500 error for "Internal Server Error"). It helps improve the user experience and prevents showing default error pages generated by the web server.

**Syntax of @app.errorhandler:**

```
@app.errorhandler(error_code)
def handle_error(error):
    return render_template('error_page.html'), error_code
```

- **error_code**: The HTTP status code you want to handle (like 404, 500, etc.).
- **handle_error**: The function that will handle the error and return a custom response.
- **render_template('error_page.html')**: Returns a custom HTML template for the error page.
- **error_code** (optional): The error status code you want to return (e.g., 404, 500).

## Example: Handling 404 Error (Page Not Found)

**app.py:**

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
   return render_template("index.html")


@app.errorhandler(404)
def page_not_found(e):
   print(e)  # It will print the entire error message
   return render_template("404.html"), 404


if __name__ == '__main__':
   app.run(debug=True)
```

<u>index.html:</u>

```
<body>
   <h1>Welcome to Chitkara</h1>
</body>
```

<u>404.html:</u>

```html
<body>

    <h1>Oops! Page Not Found (404)</h1>

    <p>Sorry, the page you're looking for does not exist.</p>

    <p><a href="{{ url_for('index') }}">Go back to Home</a></p>


</body>
```

## Other Common Error Codes and Their Handling:

1. **Handling 500 Error (Internal Server Error):**
    - This will handle server errors and display a custom 500.html page.

```python
@app.errorhandler(500)
def internal_error(e):
    return render_template("500.html"), 500
```

2. **Handling 403 Error (Forbidden):**
    - This will handle cases where the user doesn't have permission to access a particular resource.

```python
@app.errorhandler(403)
def forbidden_error(e):
    return render_template("403.html"), 403
```

## Error Handler Flow:

1. User visits a URL that does not exist (e.g., /nonexistent-page).

2. Flask detects the 404 error.
3. The @app.errorhandler(404) decorator catches the error and triggers the page_not_found function.
4. The function returns a custom 404 error page with a message like "Page Not Found."