Module 14 - DJANGO

Welcome to Module 14! This module introduces you to **Django**, a high-level Python web framework that enables rapid development of secure and maintainable websites. Django follows the "Don't Repeat Yourself" (DRY) principle and aims to make web development efficient and enjoyable.

Chapter 1: Getting Started with Django

This chapter covers the initial setup of your Django development environment.

1.1 Django Installation

Before you can start building Django projects, you need to install the Django package. It's highly recommended to do this within a virtual environment to keep your project dependencies isolated.

- **Prerequisite:** Ensure you have Python installed (preferably Python 3.8+).
- **Virtual Environment:** If you followed Module 12, you already know how to set up a virtual environment. Navigate to your project directory and activate your virtual environment.

  Bash

  ```
  # Navigate to your project directory (e.g., where you want to keep
  your Django projects)
  cd C:\Users\YourUser\PythonProjects\DjangoProjects

  # Create a virtual environment (if you haven't already)
  python -m venv django_venv

  # Activate the virtual environment
  # On Windows:
  .\django_venv\Scripts\activate
  # On macOS/Linux:
  source django_venv/bin/activate
  ```

  Expected Output: Your terminal prompt will change to include `(django_venv)`, indicating the virtual environment is active.

- **Install Django:** With your virtual environment active, use `pip` to install Django.

  Bash

  ```
  (django_venv) pip install django
  ```

  Expected Output:

  ```
  Collecting django
  ```

```
  Downloading Django-X.Y.Z-py3-none-any.whl (X.X MB)
... (other dependencies like asgiref, sqlparse, tzdata)
Successfully installed Django-X.Y.Z asgiref-X.Y.Z sqlparse-X.Y.Z
tzdata-X.Y
```

Explanation: This output confirms that Django and its necessary dependencies have been downloaded and installed into your `django_venv` virtual environment. `X.Y.Z` will be the current version numbers.

1.2 Creating a Project

A Django **project** is a collection of settings and applications that together make up a particular web application. You typically create one project for each website.

- **Command:** Use `django-admin startproject` followed by your project name.

  Bash

  ```
  (django_venv) django-admin startproject myproject .
  ```

  - `myproject`: This is the name of your Django project.
  - `.`: The dot at the end is important! It tells Django to create the project files in the current directory, rather than creating a nested `myproject/myproject` directory.

  Expected Output: There will be no direct output to the console, but a new directory structure will be created:

  ```
  .
  ├── manage.py
  └── myproject/
      ├── __init__.py
      ├── asgi.py
      ├── settings.py
      ├── urls.py  # Project-level URLs
      └── wsgi.py
  ```

  Explanation:

  - `manage.py`: A command-line utility for interacting with your Django project. You'll use this a lot.
  - `myproject/` (the inner directory): This is the actual Python package for your project.
    - `settings.py`: Contains all the configuration for your Django project.
    - `urls.py`: Declares URL patterns for your entire project.
    - `wsgi.py`, `asgi.py`: Entry points for web servers to serve your application.
- **Run Development Server:** Django comes with a lightweight web server for development. It's great for testing your application locally.

  Bash

```
(django_venv) python manage.py runserver
```

Expected Output:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work
properly until you apply the migrations for admin, auth,
contenttypes, and sessions.
Run 'python manage.py migrate' to apply them.
July 3, 2025 - 10:30:00
Django version 5.0.6, using settings 'myproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Explanation: This output indicates that the development server has started and is listening on `http://127.0.0.1:8000/`. The "unapplied migration(s)" message is normal for a new project; we'll address migrations later.

- **Action:** Open your web browser and navigate to `http://127.0.0.1:8000/`. You should see a "The install worked successfully! Congratulations!" page with a rocket icon. This confirms your project is set up correctly.
- **To stop the server:** Press `Ctrl + C` in your terminal.

- **Apply Migrations (Optional but good practice):** Migrations are Django's way of propagating changes you make to your models (your database schema) into your database. A new project comes with built-in apps that need their tables created.

  Bash

  ```
  (django_venv) python manage.py migrate
  ```

  Expected Output:

  ```
  Operations to perform:
    Apply all migrations: admin, auth, contenttypes, sessions
  Running migrations:
    Applying contenttypes.0001_initial... OK
    Applying auth.0001_initial... OK
    Applying admin.0001_initial... OK
    Applying admin.0002_logentry_add_action_flag_choices... OK
    ... (many more 'OK' lines)
  ```

  Explanation: This command creates the necessary database tables for Django's built-in authentication, administration, and session management systems.

1.3 Creating an App

A Django **app** is a web application that does something specific (e.g., a blog app, a polls app, an e-commerce app). A project can contain multiple apps, and apps can be reused in different projects.

- **Command:** Use `python manage.py startapp` followed by your app name.

  Bash

  ```
  (django_venv) python manage.py startapp myapp
  ```

  Expected Output: No direct console output, but a new directory `myapp/` will be created with the following basic structure:

  ```
  .
  ├── manage.py
  ├── myproject/
  │   └── ...
  └── myapp/
      ├── __init__.py
      ├── admin.py
      ├── apps.py
      ├── migrations/
      │   └── __init__.py
      ├── models.py
      ├── tests.py
      └── views.py
  ```

  Explanation: This creates the basic structure for your Django app. Notice that `urls.py` is **not** created by `startapp`; you will create it in the next step.

  - `views.py`: Where you'll write the logic for your web pages.
  - `models.py`: Where you'll define your database tables.
  - `admin.py`: For registering your models with the Django admin site.
  - `apps.py`: Configuration for your app.
  - `migrations/`: Stores database schema changes.
- **Register the App:** For Django to recognize your new app, you must register it in your project's `settings.py`.

  File: `myproject/settings.py` **Find:** `INSTALLED_APPS` list **Add:** `'myapp',` (or whatever you named your app)

  Python

  ```
  # myproject/settings.py

  INSTALLED_APPS = [
      'django.contrib.admin',
      'django.contrib.auth',
      'django.contrib.contenttypes',
      'django.contrib.sessions',
      'django.contrib.messages',
      'django.contrib.staticfiles',
      'myapp', # Add your app here
  ]
  ```

  Explanation: This tells Django to include your `myapp` when it loads the project.

1.4 Hello World using Django

Let's create a simple "Hello, World!" page to see how Django handles requests and responses. This involves defining a **view** and mapping a **URL** to it.

- **1. Define a View:** A view is a Python function that takes a web request and returns a web response.

  File: `myapp/views.py` **Add:**

  Python

  ```
  # myapp/views.py
  from django.http import HttpResponse

  def hello_world(request):
      return HttpResponse("Hello, Django World!")
  ```

  Explanation:

  - `from django.http import HttpResponse`: Imports the class needed to send a basic text response.
  - `def hello_world(request):`: Defines a view function. It *must* take `request` as its first argument.
  - `return HttpResponse(...)`: Creates and returns an HTTP response with the specified text.
- **2. Map a URL to the View:** You need to tell Django which URL pattern should trigger your `hello_world` view. This is done in `urls.py`.

  File: `myapp/urls.py` **CREATE THIS FILE if it doesn't exist:**

  Python

  ```
  # myapp/urls.py
  from django.urls import path
  from . import views # Import views from the current app

  urlpatterns = [
      path('hello/', views.hello_world, name='hello_world'),
  ]
  ```

  Explanation:

  - `from django.urls import path`: Imports the `path` function for defining URL patterns.
  - `from . import views`: Imports your `views.py` file from the current app directory.
  - `urlpatterns`: A list where you define your URL patterns.
  - `path('hello/', views.hello_world, name='hello_world')`: This line means:
    - If the URL path is `hello/`, then call the `hello_world` function from `views.py`.

        - `name='hello_world'` is an optional name for this URL pattern, useful for referring to it elsewhere in your project (e.g., in templates).
- **3. Include App URLs in Project URLs:** The project's main `urls.py` needs to include the URL patterns from your `myapp`.

  File: `myproject/urls.py` **Modify:**

  Python

```
# myproject/urls.py
from django.contrib import admin
from django.urls import path, include # Import include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('myapp/', include('myapp.urls')), # Include your app's URLs
]
```

  Explanation:

  - `from django.urls import include`: Imports the `include` function.
  - `path('myapp/', include('myapp.urls'))`: This line means:
    - Any URL that starts with `myapp/` should look for further URL patterns within `myapp/urls.py`.
    - So, `http://127.0.0.1:8000/myapp/hello/` will now work.
- **4. Run the Server:**

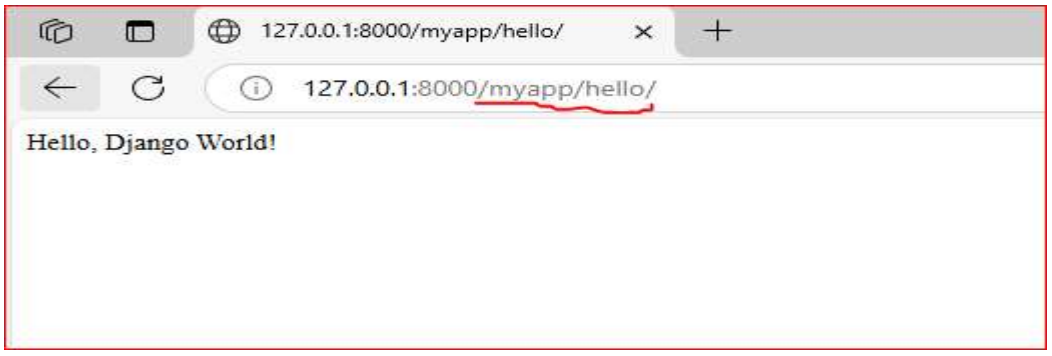
  Bash

```
(django_venv) python manage.py runserver
```

  Expected Output: The server will start, similar to before.

  - **Action:** Open your web browser and go to `http://127.0.0.1:8000/myapp/hello/`.
  - **Browser Output:** You should see a plain web page displaying:
  - `Hello, Django World!`

  Explanation: You've successfully created your first Django web page! The browser sent a request to the Django server, which routed it to your `hello_world` view, and the view returned an `HttpResponse` with the text.



Chapter 2: Django Templates and DTL

Returning plain text from `HttpResponse` is rarely enough. Websites need HTML! Django's template system allows you to generate dynamic HTML content.

2.1 Templates in Django

- **Purpose:** Django's template system allows you to separate the presentation logic (HTML) from the business logic (Python views). Templates are essentially HTML files with special placeholders and logic that Django fills in when rendering the page.
- **Location:** By convention, templates are stored in a `templates/` directory inside your app.
- **1. Create Template Directory:**

  Bash

  ```
  # From your project root (where manage.py is), ensure you are inside
  myapp/
  # For example, if you are at D:\Py_Projects\My_Project\, navigate to
  D:\Py_Projects\My_Project\myapp
  cd myapp
  mkdir templates\myapp
  # On macOS/Linux:
  # mkdir -p templates/myapp
  # Then navigate back to project root
  cd ..
  ```

  Explanation: We create `myapp/templates/myapp/`. The inner `myapp` directory is a common practice to prevent naming conflicts if another app also has a template named `index.html`, for example.

- **2. Configure Template Settings:** You need to tell Django where to find your templates.

  File: `myproject/settings.py` **Find:** `TEMPLATES` list **Modify:**

  Python

  ```
  # myproject/settings.py

  TEMPLATES = [
      {
          'BACKEND': 'django.template.backends.django.DjangoTemplates',
          'DIRS': [], # Keep this empty or add project-level template
  dirs
          'APP_DIRS': True, # THIS IS IMPORTANT: Tells Django to look
  in 'templates' directory of each app
          'OPTIONS': {
              'context_processors': [
                  'django.template.context_processors.debug',
                  'django.template.context_processors.request',
  ```

```
                'django.contrib.auth.context_processors.auth',
'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Explanation: `APP_DIRS: True` is the key setting here. It instructs Django to automatically search for a `templates/` subdirectory within each app listed in `INSTALLED_APPS`.

- **3. Create an HTML Template:**

  File: `myapp/templates/myapp/index.html` **Add:**

  HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>My First Template</title>
</head>
<body>
    <h1>Welcome to My Django App!</h1>
    <p>This content is rendered from a template.</p>
</body>
</html>
```

  Explanation: This is a standard HTML file that will be served by Django.

- **4. Modify the View to Render Template:** Instead of `HttpResponse`, use `render`.

  File: `myapp/views.py` **Modify:**

  Python

```
# myapp/views.py
from django.shortcuts import render # Import render

# Remove or comment out the old hello_world function if you want
# from django.http import HttpResponse
# def hello_world(request):
#     return HttpResponse("Hello, Django World!")

def home(request):
    return render(request, 'myapp/index.html') # Render the template
```

  Explanation:

  - `from django.shortcuts import render`: Imports the `render` shortcut function.

- `render(request, 'myapp/index.html')`: This function takes the `request` object, the path to the template (relative to the `templates/` directory and including the app's subdirectory), and an optional dictionary of context data.

- **5. Update URLs to Point to New View:**

  File: `myapp/urls.py` **Modify:**

  Python

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

urlpatterns = [
    # path('hello/', views.hello_world, name='hello_world'), # Old
URL (can be kept or removed)
    path('', views.home, name='home'), # New URL for the root of the
app
]
```

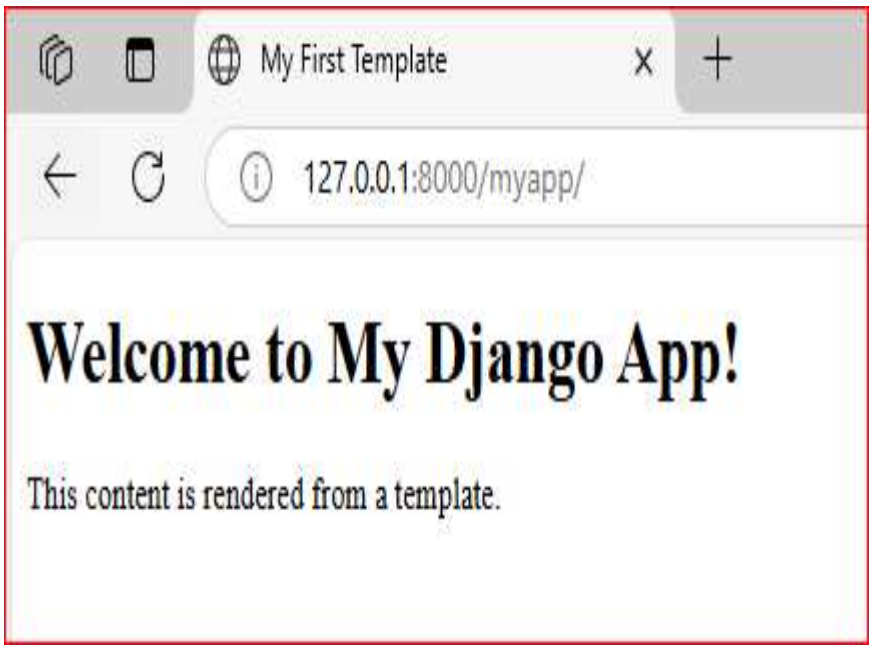
- **6. Run the Server:**

  Bash

```
(django_venv) python manage.py runserver
```

  Expected Output: Server starts.

  - **Action:** Open your web browser and go to `http://127.0.0.1:8000/myapp/`.
  - **Browser Output:** You should see a web page with the title "My First Template" and the text "Welcome to My Django App!" and "This content is rendered from a template." *Explanation:* Django successfully located and rendered your HTML template.



2.2 Django Template Language 1 (DTL - Variables and Tags)

The real power of templates comes from the Django Template Language (DTL), which allows you to insert dynamic content and control logic.

- **Variables:**
  - Syntax: `{{ variable_name }}`
  - Variables are passed from the view to the template as a dictionary (context).
  - Django replaces the variable with its actual value.
- **Tags:**
  - Syntax: `{% tag_name %}`
  - Tags execute some logic (e.g., loops, conditionals, loading static files).
- **1. Modify View to Pass Context:**

  File: `myapp/views.py` **Modify:**

  Python

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

def home(request):
    context = {
        'name': 'Alice',
        'age': 30,
        'city': 'New York',
        'is_admin': True,
        'items': ['apple', 'banana', 'cherry']
    }
    return render(request, 'myapp/index.html', context) # Pass
context dictionary
```

  Explanation: We create a `context` dictionary with various data types and pass it as the third argument to `render()`.

- **2. Modify Template to Use Variables and Tags:**

  File: `myapp/templates/myapp/index.html` **Modify:**

  HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>My First Template</title>
</head>
<body>
    <h1>Welcome, {{ name }}!</h1> {# Variable: name #}
    <p>You are {{ age }} years old and live in {{ city }}.</p> {#
Variables: age, city #}

    {% if is_admin %} {# Tag: if #}
        <p>You have administrative privileges.</p>
    {% else %}
```

```
        <p>You are a regular user.</p>
    {% endif %}

    <h2>Your Items:</h2>
    <ul>
        {% for item in items %} {# Tag: for loop #}
            <li>{{ item }}</li>
        {% endfor %}
    </ul>

    <p>Current year: {{ current_year }}</p> {# This variable is not
passed, will be empty #}
</body>
</html>
```

Explanation:

- `{{ name }}`: Displays the value of the `name` variable from the context.
- `{% if is_admin %}`: A conditional tag. The content inside `{% if %}` and `{% endif %}` will only be rendered if `is_admin` is `True`.
- `{% for item in items %}`: A loop tag. It iterates over the `items` list, and for each `item`, the content between `{% for %}` and `{% endfor %}` is repeated.

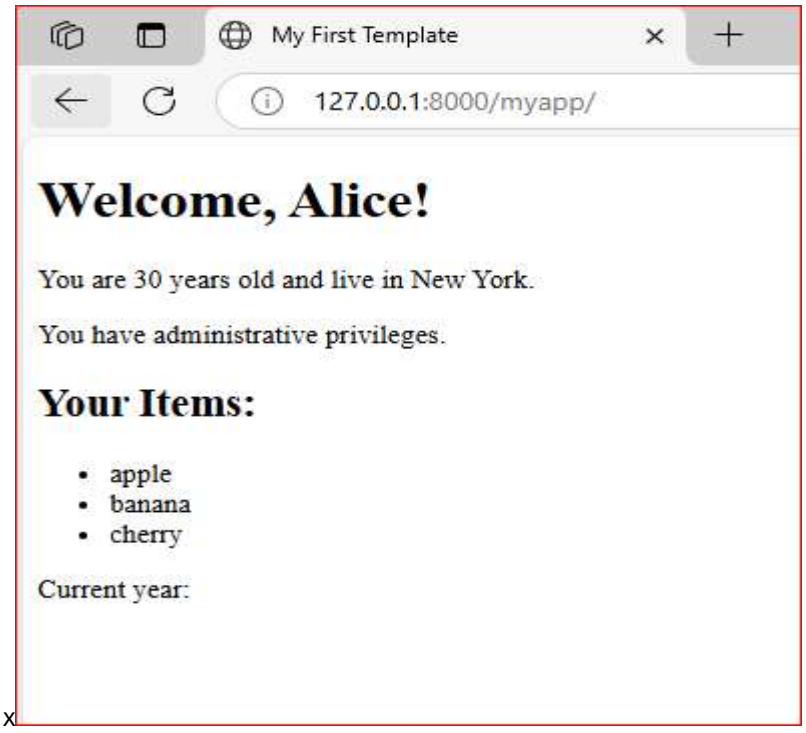
- **3. Run the Server:**

  Bash

  ```
  (django_venv) python manage.py runserver
  ```

  Expected Output: Server starts.

  - **Action:** Open your web browser and go to `http://127.0.0.1:8000/myapp/`.
  - **Browser Output:**



x

Explanation: The DTL variables are replaced with their values, the `if` block for admin is rendered, and the `for` loop correctly lists the items. `{{ current_year }}` renders as empty because it wasn't provided in the context.

2.3 Django Template Language 2 (DTL - Filters, Comments)

DTL also provides filters for transforming variable values and ways to add comments.

- **Filters:**
  - Syntax: `{{ variable|filter_name:argument }}`
  - Filters transform the display of variables. They are applied using the pipe `|` character.
- **Comments:**
  - Single-line: `{# This is a single-line comment #}`
  - Multi-line: `{% comment %} This is a multi-line comment {% endcomment %}`
- **1. Modify Template to Use Filters and Comments:**

  File: `myapp/templates/myapp/index.html` **Modify:**

  HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>My First Template</title>
</head>
<body>
    <h1>Welcome, {{ name|upper }}!</h1> {# Filter: upper - converts
to uppercase #}
    <p>You are {{ age }} years old and live in {{ city|capfirst
}}.</p> {# Filter: capfirst - capitalizes first letter #}

    {# This is a single-line comment in the template #}
    {% if is_admin %}
        <p>You have administrative privileges.</p>
    {% else %}
        <p>You are a regular user.</p>
    {% endif %}

    <h2>Your Items (Total: {{ items|length }}):</h2> {# Filter:
length - gets length of list #}
    <ul>
        {% for item in items %}
            <li>{{ item|title }}</li> {# Filter: title - capitalizes
first letter of each word #}
        {% endfor %}
    </ul>

    {% comment %}
    This is a multi-line comment.
    It will not be rendered in the final HTML output.
    {% endcomment %}

    <p>Today's date: {{ "2025-07-03"|date:"F j, Y" }}</p> {# Filter:
date - formats a date string #}
```

```
    <p>Original name: {{ name }}</p> {# Original variable value is
unaffected by filters #}
</body>
</html>
```

Explanation: We've added several filters (`upper`, `capfirst`, `length`, `title`, `date`) to demonstrate their usage. Comments are also shown.

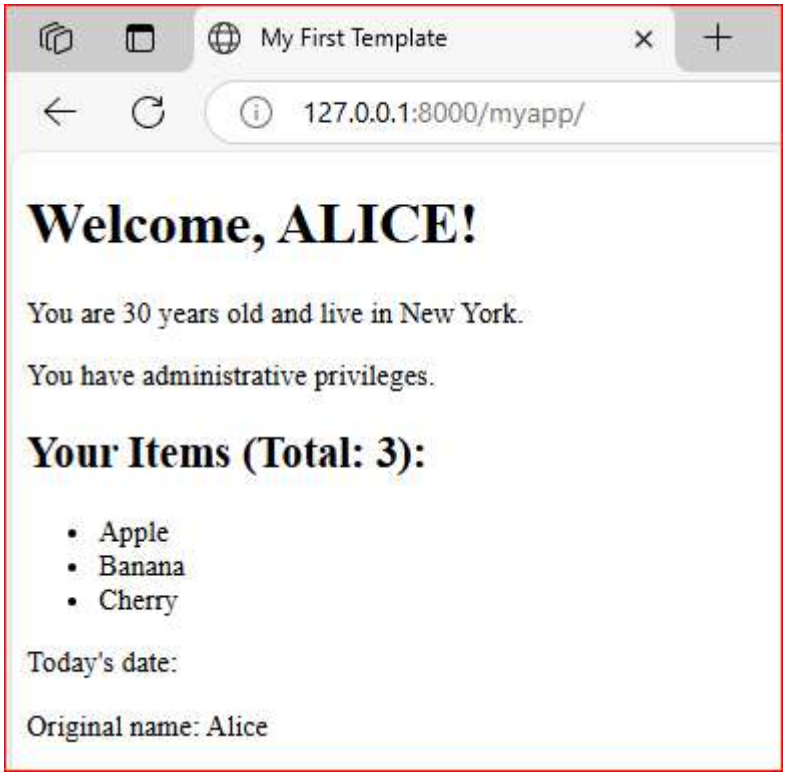
- **2. Run the Server:**

  Bash

  ```
  (django_venv) python manage.py runserver
  ```

  Expected Output: Server starts.

  - **Action:** Open your web browser and go to `http://127.0.0.1:8000/myapp/`.
  - **Browser Output:**



  Explanation: The filters have transformed the variable output as expected. Comments are completely removed from the final HTML source.

2.4 Adding Numbers (Simple Form Submission)

Let's create a simple form that takes two numbers, adds them, and displays the result. This introduces basic form handling without Django Forms.

- **1. Modify View to Handle Form Submission:**

  File: `myapp/views.py` **Modify:**

  Python

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

def home(request):
    context = {
        'name': 'Alice',
        'age': 30,
        'city': 'New York',
        'is_admin': True,
        'items': ['apple', 'banana', 'cherry'],
        'result': None # Initialize result to None
    }

    if request.method == 'POST':
        try:
            num1 = float(request.POST.get('num1'))
            num2 = float(request.POST.get('num2'))
            context['result'] = num1 + num2
        except (ValueError, TypeError):
            context['result'] = "Invalid input. Please enter
numbers."

    return render(request, 'myapp/index.html', context)
```

  Explanation:

  - `if request.method == 'POST'`: Checks if the request method is POST (meaning the form was submitted).
  - `request.POST.get('num1')`: Retrieves the value of the input field named `num1` from the POST data. `.get()` is safer than `[]` as it returns `None` if the key doesn't exist.
  - `float(...)`: Converts the string input to a float.
  - `try-except`: Handles cases where input is not a valid number.
  - `context['result']`: Stores the calculation result (or error message) in the context, which will be passed to the template.

- **2. Modify Template to Include Form:**

  File: `myapp/templates/myapp/index.html` **Modify (add this section to the `<body>`):**

  HTML

```
{# ... existing HTML and DTL ... #}

    <h2>Add Two Numbers:</h2>
    <form method="post"> {# method="post" is crucial for sending data
#}
        {% csrf_token %} {# IMPORTANT: Django's security token for
forms #}
```

```
        <label for="num1">Number 1:</label>
        <input type="text" id="num1" name="num1" required>
        <br><br>
        <label for="num2">Number 2:</label>
        <input type="text" id="num2" name="num2" required>
        <br><br>
        <button type="submit">Add</button>
    </form>

    {% if result is not None %} {# Display result only if it's set #}
        <h3>Result: {{ result }}</h3>
    {% endif %}

</body>
</html>
```

Explanation:

- `<form method="post">`: Specifies that the form data will be sent using the POST HTTP method.
- `{% csrf_token %}`: This is a vital security tag provided by Django. It protects against Cross-Site Request Forgery (CSRF) attacks. **Always include it in your forms!**
- `<input type="text" name="num1">`: Defines input fields. The `name` attribute is what Django uses to retrieve the value (`request.POST.get('num1')`).
- `<button type="submit">`: The button that submits the form.
- `{% if result is not None %}`: Conditionally displays the result only after a calculation has been performed.

- **3. Run the Server:**

  Bash

  ```
  (django_venv) python manage.py runserver
  ```

  Expected Output: Server starts.

  - **Action:** Open your web browser and go to `http://127.0.0.1:8000/myapp/`. You'll see the form.
  - **Browser Output (before submission):** The page with the existing content and a new section for "Add Two Numbers" with input fields and an "Add" button. The "Result" heading will not be visible.
  - **Browser Output (after entering 5 and 3, then clicking Add):** The page will reload, and below the form, you'll see:
  - `Result: 8.0`
  - **Browser Output (after entering "abc" and "3", then clicking Add):**
  - `Result: Invalid input. Please enter numbers.`

Explanation: The form submits data via POST to the same URL. Django processes the POST request in the view, performs the calculation, and re-renders the template with the `result` in the context.

Chapter 3: Web Fundamentals & Django Architecture

Before diving deeper, let's solidify some core web concepts and understand Django's architectural pattern.

3.1 GET vs POST (HTTP Methods)

HTTP (Hypertext Transfer Protocol) defines methods (verbs) for clients (browsers) to interact with servers. The two most common are GET and POST.

- **GET Method:**
  - **Purpose:** Used to *request* data from a specified resource. It's for retrieving information.
  - **Characteristics:**
    - Data is appended to the URL as query parameters (e.g., `example.com/search?query=django`).
    - Data is visible in the URL, browser history, and server logs.
    - Has length limitations on the URL.
    - **Idempotent:** Making the same GET request multiple times should have the same effect on the server (i.e., it should not change server state).
    - Can be bookmarked.
  - **When to use:** Retrieving static content, search queries, filtering results.
- **POST Method:**
  - **Purpose:** Used to *submit* data to be processed to a specified resource. It's for sending information to the server to create or update a resource.
  - **Characteristics:**
    - Data is sent in the body of the HTTP request, not in the URL.
    - Data is not visible in the URL or browser history (though it can be intercepted).
    - No practical length limitations on data.
    - **Not Idempotent:** Making the same POST request multiple times might have different effects (e.g., submitting a form twice might create two identical entries).
    - Cannot be bookmarked.
  - **When to use:** Submitting forms (registration, login, comments), uploading files, making changes to a database.
- **Example (from previous section):**
  - Our "Add Two Numbers" form used `method="post"`. This means when you click "Add", the numbers are sent in the request body.
  - If you were to change `method="get"`, the URL would look like `http://127.0.0.1:8000/myapp/?num1=5&num2=3` after submission.

3.2 MVT (Model-View-Template Architecture)

Django follows the **Model-View-Template (MVT)** architectural pattern. It's similar to the more common MVC (Model-View-Controller) but adapted for web frameworks.

- **Model:**
  - **What it is:** The data layer. It defines the structure of your data, how it's stored (e.g., in a database), and the business logic related to that data.
  - **In Django:** Defined in `models.py` as Python classes that inherit from `django.db.models.Model`. Each class maps to a database table. Django's ORM (Object-Relational Mapper) handles the interaction with the database.
  - **Role:** Interacts with the database (CRUD operations: Create, Read, Update, Delete).
- **View:**
  - **What it is:** The logic layer. It receives the web request, processes it (e.g., interacts with the Model to get/save data), and determines what data should be sent to the template.

    - **In Django:** Defined in `views.py` as Python functions (or classes). It's responsible for fetching data, processing user input, and deciding which template to render.
    - **Role:** Acts as the "controller" in a traditional MVC sense. It handles the request and prepares the response.
- **Template:**
    - **What it is:** The presentation layer. It defines how the data is displayed to the user (usually HTML).
    - **In Django:** HTML files (or other text formats) containing DTL syntax (`{{ variables }}`, `{% tags %}`).
    - **Role:** Renders the data provided by the View into a user-friendly format.
- **How they interact in Django:**

    1. A user's browser sends an **HTTP Request** to the Django server.
    2. Django's **URL dispatcher (`urls.py`)** receives the request and matches the URL pattern to a specific **View** function.
    3. The **View** function executes. It might interact with the **Model** (e.g., query the database for data, save new data).
    4. The **View** then prepares context data (a dictionary) and passes it to a **Template**.
    5. The **Template** renders the HTML, inserting the dynamic data from the context.
    6. The rendered HTML is sent back as an **HTTP Response** to the user's browser.

This separation of concerns makes Django applications modular, scalable, and easier to maintain.

Chapter 4: Static Files

Websites are not just dynamic content; they also need static assets like CSS stylesheets, JavaScript files, and images. Django has a robust system for managing these.

4.1 Static Files 1 (Configuring Static Files)

- **Purpose:** To serve static files efficiently during development and in production.
- **Default Location:** Django looks for a `static/` directory inside each of your apps by default.
- **Project-level Static Files:** You can also define a project-wide directory for static files.
- **1. Create Static Directory in App:**

    Bash

    ```
    # From your app directory (myapp/)
    mkdir static\myapp
    # On macOS/Linux:
    # mkdir -p static/myapp
    ```

Explanation: We create `myapp/static/myapp/`. The inner `myapp` directory is a common practice to avoid naming conflicts.

- **2. Add a Static File:**

  File: `myapp/static/myapp/style.css` **Add:**

  CSS

```
/* myapp/static/myapp/style.css */
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    color: #333;
    margin: 20px;
}

h1 {
    color: #0056b3;
}

.message {
    background-color: #d4edda;
    color: #155724;
    border: 1px solid #c3e6cb;
    padding: 10px;
    margin-bottom: 15px;
    border-radius: 5px;
}
```

  Explanation: A simple CSS file to style our pages.

- **3. Configure `settings.py` for Static Files:** Django automatically handles static files from `APP_DIRS` during development. However, it's good to know the settings.

  File: `myproject/settings.py` **Find/Add (usually at the bottom):**

  Python

```
# myproject/settings.py

# ... (other settings) ...

STATIC_URL = 'static/' # The URL prefix for static files.

# Optional: Define additional directories where Django should look
for static files
# STATICFILES_DIRS = [
#     BASE_DIR / "static_project_files", # Example for project-wide
static files
# ]

# STATIC_ROOT = BASE_DIR / "staticfiles" # Used for 'collectstatic'
in production
```

  Explanation:

- `STATIC_URL`: This is the base URL from which static files will be served. When you refer to a static file in your template using `{% static 'path/to/file.css' %}`, Django will prepend `STATIC_URL` to it.
- `STATICFILES_DIRS`: (Optional) A list of additional directories where Django will look for static files, outside of individual app `static/` folders. Useful for project-wide static assets.
- `STATIC_ROOT`: (Important for production) The absolute path to the directory where `collectstatic` will gather all static files for deployment. Not used in development server.

4.2 Static Files 2 (Using Static Files in Templates)

To use your static files (like `style.css`) in your templates, you need to load the `static` tag library and then use the `{% static %}` tag.

- **1. Modify Template to Include CSS:**

  File: `myapp/templates/myapp/index.html` **Modify (add `{% load static %}` and `<link>` tag):**

  HTML

```
{% load static %} {# Load the static files tag library at the very
top #}
<!DOCTYPE html>
<html>
<head>
    <title>My First Template</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}"> {#
Link to your CSS file #}
</head>
<body>
    <div class="message">
        This is a styled message.
    </div>
    <h1>Welcome, {{ name|upper }}!</h1>
    {# ... rest of your template content ... #}
</body>
</html>
```

  Explanation:

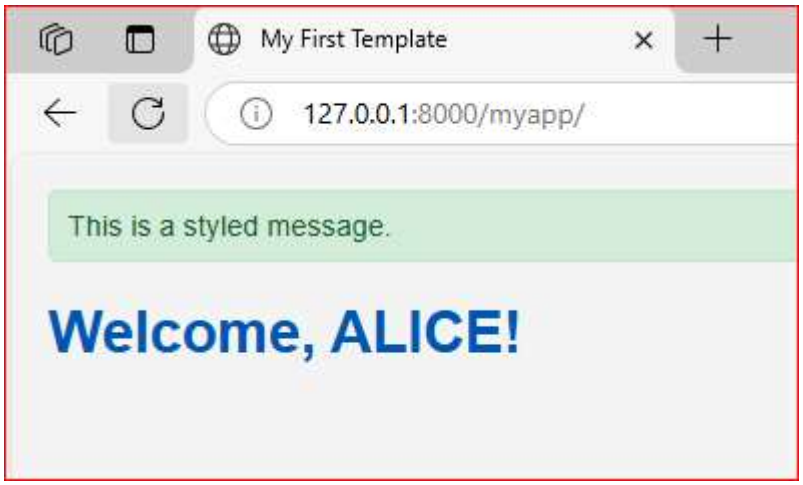
  - `{% load static %}`: This tag must be at the top of any template where you want to use the `{% static %}` tag.
  - `href="{% static 'myapp/style.css' %}"`: This DTL tag generates the correct URL for your static file. Django will look for `style.css` inside the `static/myapp/` directory of your `myapp`.
- **2. Run the Server:**

  Bash

```
(django_venv) python manage.py runserver
```

Expected Output: Server starts.

- **Action:** Open your web browser and go to `http://127.0.0.1:8000/myapp/`.
- **Browser Output:** The page should now appear with the styles defined in `style.css`. The "Welcome" heading will be blue, the body background light grey, and the "This is a styled message" text will have a light green background with green text. *Explanation:* Django's static files system correctly located and served your CSS file, and the browser applied the styles.



Chapter 5: User Authentication

Django comes with a powerful, built-in authentication system that handles user registration, login, and logout.

5.1 User Registration (Part 1: Basic Setup, User Model)

Django's authentication system uses a `User` model by default. We just need to ensure it's set up and ready.

- **Prerequisite:** You must have run `python manage.py migrate` at least once to create the necessary tables for the `auth` app. If you haven't, do it now:

  Bash

  ```
  (django_venv) python manage.py migrate
  ```

  Expected Output: Many `OK` lines for `auth` and other built-in apps.

- **Django's `User` Model:** Django provides a `User` model (`django.contrib.auth.models.User`) that handles usernames, passwords (hashed), email, first name, last name, etc. You usually don't need to create your own `User` model unless you have very specific requirements.

- **Create a Superuser (Admin User):** This allows you to access the Django admin interface, which is useful for managing users and other data.

  Bash

  ```
  (django_venv) python manage.py createsuperuser
  ```

  Expected Output:

  ```
  Username (leave blank to use 'your_username'): admin
  Email address: admin@example.com
  Password:
  Password (again):
  Superuser created successfully.
  ```

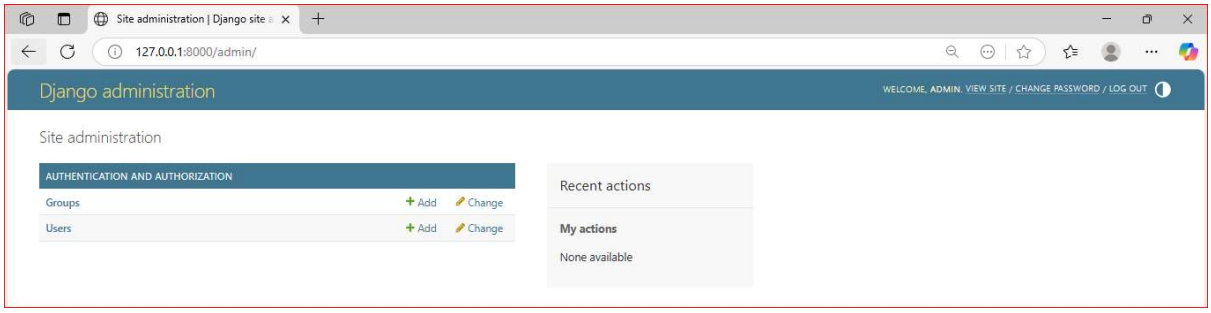
  Explanation: This command prompts you to create an administrative user for your Django project. Remember the username and password you set.

- **Access Django Admin:**

  Bash

  ```
  (django_venv) python manage.py runserver
  ```

  - **Action:** Open your browser and go to `http://127.0.0.1:8000/admin/`.
  - **Browser Output:** You'll see a login page for the Django administration site.
  - **Action:** Log in with the superuser credentials you just created.
  - **Browser Output:** You'll be redirected to the Django admin dashboard, where you can see and manage Users and Groups. *Explanation:* The admin site is automatically enabled by `django.contrib.admin` in `INSTALLED_APPS`. It provides a powerful interface for managing your database content.



5.2 User Registration (Part 2: Creating a Registration Form)

Django provides built-in forms for user authentication, including `UserCreationForm` for registration.

- **1. Create a `forms.py` in your app:**

  File: `myapp/forms.py` **Add:**

  Python

```
# myapp/forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm

class CustomUserCreationForm(UserCreationForm):
    class Meta(UserCreationForm.Meta):
        model = UserCreationForm.Meta.model # Use the default User
model
        fields = UserCreationForm.Meta.fields + ('email',) # Add
email to fields
```

Explanation:

- `from django.contrib.auth.forms import UserCreationForm`: Imports Django's built-in form for creating users.
- `CustomUserCreationForm(UserCreationForm)`: We inherit from `UserCreationForm` to easily extend it.
- `fields = UserCreationForm.Meta.fields + ('email',)`: This line tells the form to include all default fields (username, password, password confirmation) plus the `email` field.

- **2. Create a Registration View:**

  File: `myapp/views.py` **Add:**

  Python

```
# myapp/views.py
from django.shortcuts import render, redirect # Import redirect
from .forms import CustomUserCreationForm # Import your custom form

# ... (existing home view) ...

def register(request):
    if request.method == 'POST':
        form = CustomUserCreationForm(request.POST)
        if form.is_valid():
            form.save() # Save the new user to the database
            return redirect('login') # Redirect to a login page (will
create later)
    else:
        form = CustomUserCreationForm() # Create a blank form for GET
request
    return render(request, 'myapp/register.html', {'form': form})
```

  Explanation:

  - `if request.method == 'POST'`: If the form is submitted.
  - `form = CustomUserCreationForm(request.POST)`: Creates a form instance populated with the submitted data.
  - `if form.is_valid()`: Django's forms have built-in validation. This checks if all fields are valid (e.g., passwords match, username is unique).
  - `form.save()`: If valid, this method creates a new user in the database.
  - `return redirect('login')`: Redirects the user to a named URL pattern (we'll define 'login' later).

- `else: form = CustomUserCreationForm()`: For a GET request (when the page is first loaded), an empty form is created.

- **3. Create a Registration Template:**

  File: `myapp/templates/myapp/register.html` **Add:**

  HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Register</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
    <style>
        /* Basic styling for forms */
        form {
            margin: 20px;
            padding: 20px;
            border: 1px solid #ddd;
            border-radius: 8px;
            background-color: #fff;
            max-width: 400px;
        }
        form p {
            margin-bottom: 10px;
        }
        form label {
            display: block;
            margin-bottom: 5px;
            font-weight: bold;
        }
        form input[type="text"],
        form input[type="password"],
        form input[type="email"] {
            width: calc(100% - 20px);
            padding: 8px;
            margin-bottom: 10px;
            border: 1px solid #ccc;
            border-radius: 4px;
        }
        form button {
            background-color: #007bff;
            color: white;
            padding: 10px 15px;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            font-size: 16px;
        }
        form button:hover {
            background-color: #0056b3;
        }
        .errorlist {
            color: red;
            list-style-type: none;
            padding: 0;
            margin-top: -5px;
            margin-bottom: 10px;
```

```
        }
        .errorlist li {
            font-size: 0.9em;
        }
    </style>
</head>
<body>
    <h1>Register</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }} {# Renders form fields as paragraphs #}
        <button type="submit">Register</button>
    </form>
</body>
</html>
```

Explanation:

- `{{ form.as_p }}`: This is a powerful DTL feature. Django forms can render themselves. `as_p` renders each field within a `<p>` tag. Other options include `as_ul` (unordered list) and `as_table`.
- Basic CSS is included to make the form presentable.

- **4. Add URL for Registration:**

  File: `myapp/urls.py` **Modify:**

  Python

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

urlpatterns = [
    path('', views.home, name='home'),
    path('register/', views.register, name='register'), # New URL for
registration
]
```

- **5. Run the Server:**

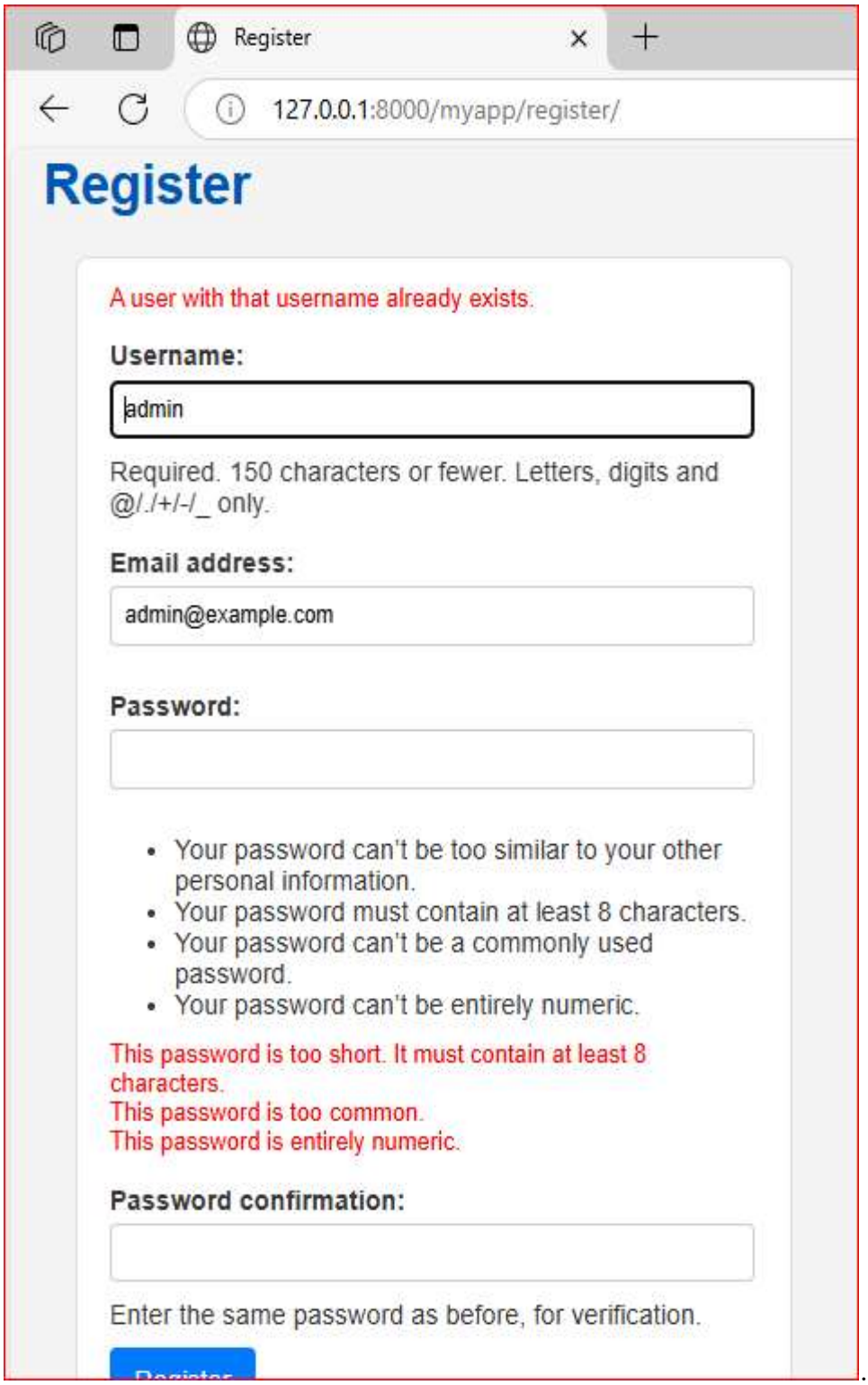
  Bash

```
(django_venv) python manage.py runserver
```

  Expected Output: Server starts.

  - **Action:** Open your browser and go to `http://127.0.0.1:8000/myapp/register/`.
  - **Browser Output:** A registration form with fields for Username, Password, Password confirmation, and Email.
  - **Action:** Try submitting with different inputs:
    - Empty fields: Error messages will appear below fields.
    - Passwords don't match: Error message.
    - Username already exists: Error message.

- Valid input: You'll be redirected to a page that says "Page not found" or similar (because we haven't created the 'login' URL yet, but the user will be created in the database). *Explanation:* Django's `UserCreationForm` handles all the HTML generation, validation, and even hashing of passwords automatically, making user registration much simpler



5.3 User Registration (Part 3: Saving User to Database)

The `form.save()` method in the view (from the previous section) is what actually creates the new user record in the database.

- **Review `form.save()`:**
  - When `form.is_valid()` is `True` for a `ModelForm` (like `UserCreationForm` which is based on Django's `User` model), `form.save()` creates a new instance of the model (in this case, a new `User` object) and saves it to the database.
  - For `UserCreationForm`, it also handles password hashing automatically.
- **Verification:**
  - After successfully submitting a valid registration form (e.g., `username: testuser`, `password: password123`, `email: test@example.com`), you can verify the user creation.
  - **Action:** Go to your Django admin site: `http://127.0.0.1:8000/admin/`. Log in with your superuser account.
  - **Browser Output:** Under the "AUTHENTICATION AND AUTHORIZATION" section, click on "Users." You should see your newly registered user (`testuser`) listed there. *Explanation:* This confirms that the `form.save()` method successfully interacted with Django's ORM to create a new record in the `auth_user` table in your database.

5.4 Login

Django's authentication system also provides built-in views and forms for user login.

- **1. Add Login View:** Django provides a built-in `LoginView` that handles the login process.

  File: `myapp/views.py` **Add:**

  Python

```
# myapp/views.py
from django.shortcuts import render, redirect
from .forms import CustomUserCreationForm
from django.contrib.auth.views import LoginView as AuthLoginView #
Import Django's built-in LoginView
from django.urls import reverse_lazy # For reverse_lazy

# ... (existing home, register views) ...

class CustomLoginView(AuthLoginView):
    template_name = 'myapp/login.html' # Specify your login template
    fields = '__all__' # Use all fields (username, password)
    redirect_authenticated_user = False # Redirect logged-in users
away from login page

    def get_success_url(self):
        return reverse_lazy('home') # Redirect to 'home' after
successful login
```

  Explanation:

- We import `LoginView` and rename it to `AuthLoginView` to avoid conflict with a potential `login` function name.
- We create a class-based view `CustomLoginView` that inherits from `AuthLoginView`. This allows us to customize its behavior (like template name and redirect URL).
- `template_name`: Points to the HTML template for the login form.
- `fields = '__all__'`: Tells the `LoginView` to use all fields from its underlying form (which is `AuthenticationForm` by default, containing username and password).
- `redirect_authenticated_user = True`: If a user who is already logged in tries to access the login page, they will be redirected to `get_success_url`.
- `get_success_url()`: Defines where the user should be redirected after a successful login. `reverse_lazy('home')` gets the URL for the `home` named pattern.

- **2. Create a Login Template:**

  File: `myapp/templates/myapp/login.html` **Add:**

  HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
    <style>
        /* Basic styling for forms (same as register.html) */
        form {
            margin: 20px;
            padding: 20px;
            border: 1px solid #ddd;
            border-radius: 8px;
            background-color: #fff;
            max-width: 400px;
        }
        form p {
            margin-bottom: 10px;
        }
        form label {
            display: block;
            margin-bottom: 5px;
            font-weight: bold;
        }
        form input[type="text"],
        form input[type="password"],
        form input[type="email"] {
            width: calc(100% - 20px);
            padding: 8px;
            margin-bottom: 10px;
            border: 1px solid #ccc;
            border-radius: 4px;
        }
        form button {
            background-color: #007bff;
            color: white;
```

```
            padding: 10px 15px;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            font-size: 16px;
        }
        form button:hover {
            background-color: #0056b3;
        }
        .errorlist {
            color: red;
            list-style-type: none;
            padding: 0;
            margin-top: -5px;
            margin-bottom: 10px;
        }
        .errorlist li {
            font-size: 0.9em;
        }
    </style>
</head>
<body>
    <h1>Login</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Login</button>
    </form>
    <p>Don't have an account? <a href="{% url 'register' %}">Register
here</a>.</p>
</body>
</html>
```

Explanation: Similar to the registration form, it renders `form.as_p` for username and password fields.

- **3. Add URL for Login:**

  File: `myapp/urls.py` **Modify:**

  Python

```
# myapp/urls.py
from django.urls import path
from . import views
# from django.contrib.auth import views as auth_views # Alternative
for direct import of built-in views

urlpatterns = [
    path('', views.home, name='home'),
    path('register/', views.register, name='register'),
    path('login/', views.CustomLoginView.as_view(), name='login'), #
New URL for login
]
```

- **4. Configure `LOGIN_REDIRECT_URL` (Login Redirect - login-2):** This setting tells Django where to redirect users after a successful login if they didn't try to access a protected page first.

File: `myproject/settings.py` **Add (usually at the bottom):**

Python

```
# myproject/settings.py

# ... (other settings) ...

LOGIN_REDIRECT_URL = '/myapp/' # Redirect to the home page of your
app after login
# Or using named URL: LOGIN_REDIRECT_URL = 'home' # If 'home' is a
top-level URL pattern
# For now, '/myapp/' works as our home is at that path.
```

Explanation: This global setting is used by Django's authentication system.

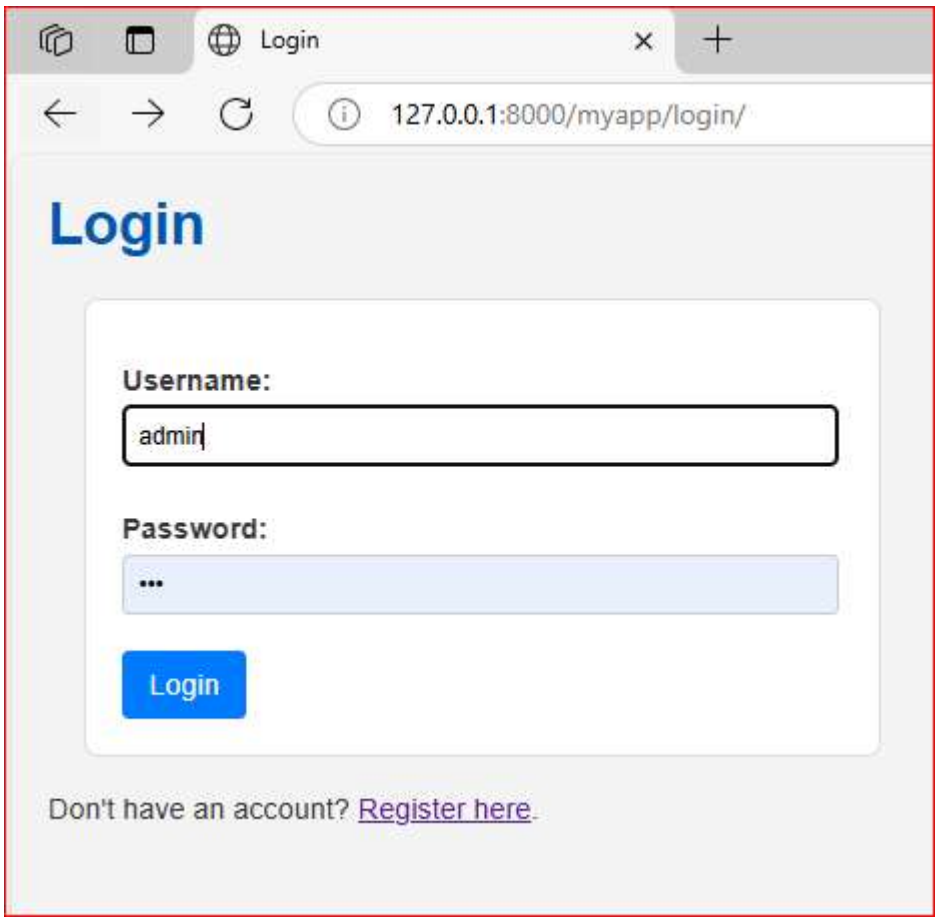
- **5. Run the Server:**

  Bash

  ```
  (django_venv) python manage.py runserver
  ```

  Expected Output: Server starts.

  - **Action:** Open your browser and go to `http://127.0.0.1:8000/myapp/login/`.
  - **Browser Output:** A login form.
  - **Action:** Try logging in with a user you registered (e.g., `testuser`, `password123`).
  - **Browser Output:** If successful, you will be redirected to `http://127.0.0.1:8000/myapp/` (your home page). *Explanation:* Django's `LoginView` handles the authentication process, sets a session cookie, and then redirects the user.



5.5 Login Redirect (login-2)

This topic was covered in the previous section (5.4 Login, step 4) by setting `LOGIN_REDIRECT_URL` in `settings.py` and `get_success_url` in `CustomLoginView`.

- **`LOGIN_REDIRECT_URL` in `settings.py`:** This is the default URL Django redirects to after a successful login.
- **`get_success_url()` in `LoginView` subclass:** If you subclass `LoginView`, you can override this method to provide dynamic redirect logic or a specific URL for that particular login view. Our `CustomLoginView` uses this to redirect to `home`.

5.6 Logout (login-3)

Logging a user out involves clearing their session data. Django provides a built-in `LogoutView`.

- **1. Add Logout View:**

  File: `myapp/views.py` **Add:**

  Python

```
# myapp/views.py
from django.shortcuts import render, redirect
```

```
from .forms import CustomUserCreationForm
from django.contrib.auth.views import LoginView as AuthLoginView,
LogoutView as AuthLogoutView # Import LogoutView
from django.urls import reverse_lazy
from django.contrib.auth.decorators import login_required # For
protecting views

# ... (existing home, register, CustomLoginView views) ...

class CustomLogoutView(AuthLogoutView):
    next_page = reverse_lazy('home') # Redirect to home page after
logout
```

Explanation: We subclass `LogoutView` and set `next_page` to redirect to our home page after logout.

- **2. Add Logout URL:**

  File: `myapp/urls.py` **Modify:**

  Python

```
# myapp/urls.py
from django.urls import path
from . import views
# from django.contrib.auth import views as auth_views

urlpatterns = [
    path('', views.home, name='home'),
    path('register/', views.register, name='register'),
    path('login/', views.CustomLoginView.as_view(), name='login'),
    path('logout/', views.CustomLogoutView.as_view(), name='logout'),
# New URL for logout
]
```

- **3. Update Home Template to Show Login/Logout Status:**

  File: `myapp/templates/myapp/index.html` **Modify (add this to the `<body>`):**

  HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>My First Template</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
</head>
<body>
    <div class="message">
        This is a styled message.
    </div>
    <h1>Welcome, {{ name|upper }}!</h1>

    {# Display login/logout links based on user status #}
    {% if user.is_authenticated %}
        <p>Hello, {{ user.username }}!</p>
```

```
        <p><a href="{% url 'logout' %}">Logout</a></p>
    {% else %}
        <p>You are not logged in.</p>
        <p><a href="{% url 'login' %}">Login</a> | <a href="{% url
'register' %}">Register</a></p>
    {% endif %}

    {# ... rest of your template content ... #}
</body>
</html>
```

Explanation:

- `{% if user.is_authenticated %}`: This is a powerful DTL feature. The `user` object is automatically available in templates (thanks to `django.contrib.auth.context_processors.auth` in `settings.py`) and allows you to check if a user is logged in.
- `user.username`: Accesses the username of the logged-in user.
- `{% url 'logout' %}`: Generates the URL for the named `logout` pattern.

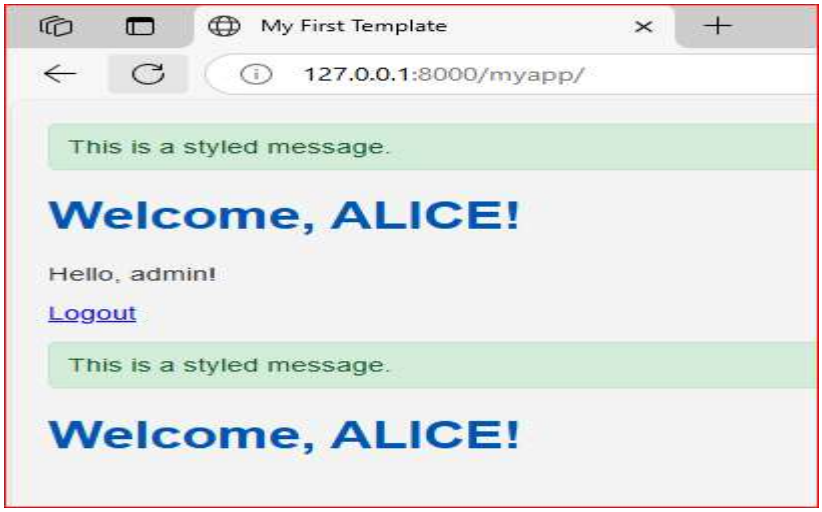
- **4. Run the Server:**

  Bash

  ```
  (django_venv) python manage.py runserver
  ```

  Expected Output: Server starts.

  - **Action:**
    1. Go to `http://127.0.0.1:8000/myapp/login/` and log in.
    2. You'll be redirected to `http://127.0.0.1:8000/myapp/`.
    3. **Browser Output:** You should now see "Hello, [your_username]!" and a "Logout" link.
    4. **Action:** Click the "Logout" link.
    5. **Browser Output:** You'll be redirected back to the home page, but now it will show "You are not logged in." and "Login | Register" links. *Explanation:* Django's authentication system correctly manages user sessions, and you can now provide a full login/logout flow.



Chapter 6: Django Forms

Django's Forms component simplifies creating, processing, and validating HTML forms.

6.1 Forms 1 (Basic Form Creation)

- **Purpose:** To define forms in Python code, handle validation, and render them easily in templates.
- **Creating a `Form` class:** Inherit from `django.forms.Form`.
- **Defining fields:** Use `forms.CharField`, `forms.IntegerField`, etc.
- **1. Create a simple contact form:**

  File: `myapp/forms.py` **Add:**

  Python

  ```
  # myapp/forms.py
  from django import forms
  from django.contrib.auth.forms import UserCreationForm

  class CustomUserCreationForm(UserCreationForm):
      class Meta(UserCreationForm.Meta):
          model = UserCreationForm.Meta.model
          fields = UserCreationForm.Meta.fields + ('email',)

  # New: Contact Form
  class ContactForm(forms.Form):
      name = forms.CharField(max_length=100, label='Your Name')
      email = forms.EmailField(label='Your Email')
      message = forms.CharField(widget=forms.Textarea, label='Your
  Message')
  ```

  Explanation:

  - `forms.CharField`: For text input. `max_length` is a validation rule.
  - `forms.EmailField`: For email input, includes email format validation.
  - `widget=forms.Textarea`: Changes the default single-line input to a multi-line text area.
- **2. Create a View for the Contact Form:**

  File: `myapp/views.py` **Add:**

  Python

  ```
  # myapp/views.py
  from django.shortcuts import render, redirect
  from .forms import CustomUserCreationForm, ContactForm # Import
  ContactForm
  from django.contrib.auth.views import LoginView as AuthLoginView,
  LogoutView as AuthLogoutView
  from django.urls import reverse_lazy
  ```

```
from django.contrib.auth.decorators import login_required
from django.contrib import messages # Import messages

# ... (existing views) ...

def contact_us(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Process the data (e.g., send email, save to DB)
            name = form.cleaned_data['name']
            email = form.cleaned.data['email']
            message_content = form.cleaned_data['message']
            print(f"Contact Form Submitted:\nName: {name}\nEmail:
{email}\nMessage: {message_content}")
            messages.success(request, 'Your message has been sent
successfully!')
            return redirect('contact_us') # Redirect to clear form
    else:
        form = ContactForm()
    return render(request, 'myapp/contact.html', {'form': form})
```

Explanation:

- `form.cleaned_data`: After `form.is_valid()` returns `True`, the validated and cleaned data is available in `form.cleaned_data` as a dictionary.
- `messages.success(request, ...)`: Django's messages framework for displaying one-time notifications to the user.

- **3. Create a Template for the Contact Form:**

  File: `myapp/templates/myapp/contact.html` **Add:**

  HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Contact Us</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
    <style>
        /* Basic styling for forms (same as register.html) */
        form {
            margin: 20px;
            padding: 20px;
            border: 1px solid #ddd;
            border-radius: 8px;
            background-color: #fff;
            max-width: 400px;
        }
        form p {
            margin-bottom: 10px;
        }
        form label {
            display: block;
            margin-bottom: 5px;
            font-weight: bold;
        }
```

```
        form input[type="text"],
        form input[type="password"],
        form input[type="email"],
        form textarea { /* Added textarea */
            width: calc(100% - 20px);
            padding: 8px;
            margin-bottom: 10px;
            border: 1px solid #ccc;
            border-radius: 4px;
        }
        form button {
            background-color: #007bff;
            color: white;
            padding: 10px 15px;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            font-size: 16px;
        }
        form button:hover {
            background-color: #0056b3;
        }
        .errorlist {
            color: red;
            list-style-type: none;
            padding: 0;
            margin-top: -5px;
            margin-bottom: 10px;
        }
        .errorlist li {
            font-size: 0.9em;
        }
        .messages { /* Styling for Django messages */
            list-style-type: none;
            padding: 0;
            margin: 20px;
        }
        .messages li {
            padding: 10px;
            border-radius: 5px;
            margin-bottom: 10px;
        }
        .messages .success {
            background-color: #d4edda;
            color: #155724;
            border: 1px solid #c3e6cb;
        }
        /* Add other message types if needed: error, warning, info */
    </style>
</head>
<body>
    <h1>Contact Us</h1>

    {% if messages %} {# Display Django messages #}
        <ul class="messages">
            {% for message in messages %}
                <li{% if message.tags %} class="{{ message.tags }}"{%
endif %}>{{ message }}</li>
            {% endfor %}
        </ul>
    {% endif %}
```

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

Explanation: Includes the `messages` loop to display success/error notifications.

- **4. Add URL for Contact Form:**

  File: `myapp/urls.py` **Modify:**

  Python

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

urlpatterns = [
    path('', views.home, name='home'),
    path('register/', views.register, name='register'),
    path('login/', views.CustomLoginView.as_view(), name='login'),
    path('logout/', views.CustomLogoutView.as_view(), name='logout'),
    path('contact/', views.contact_us, name='contact_us'), # New URL
]
```

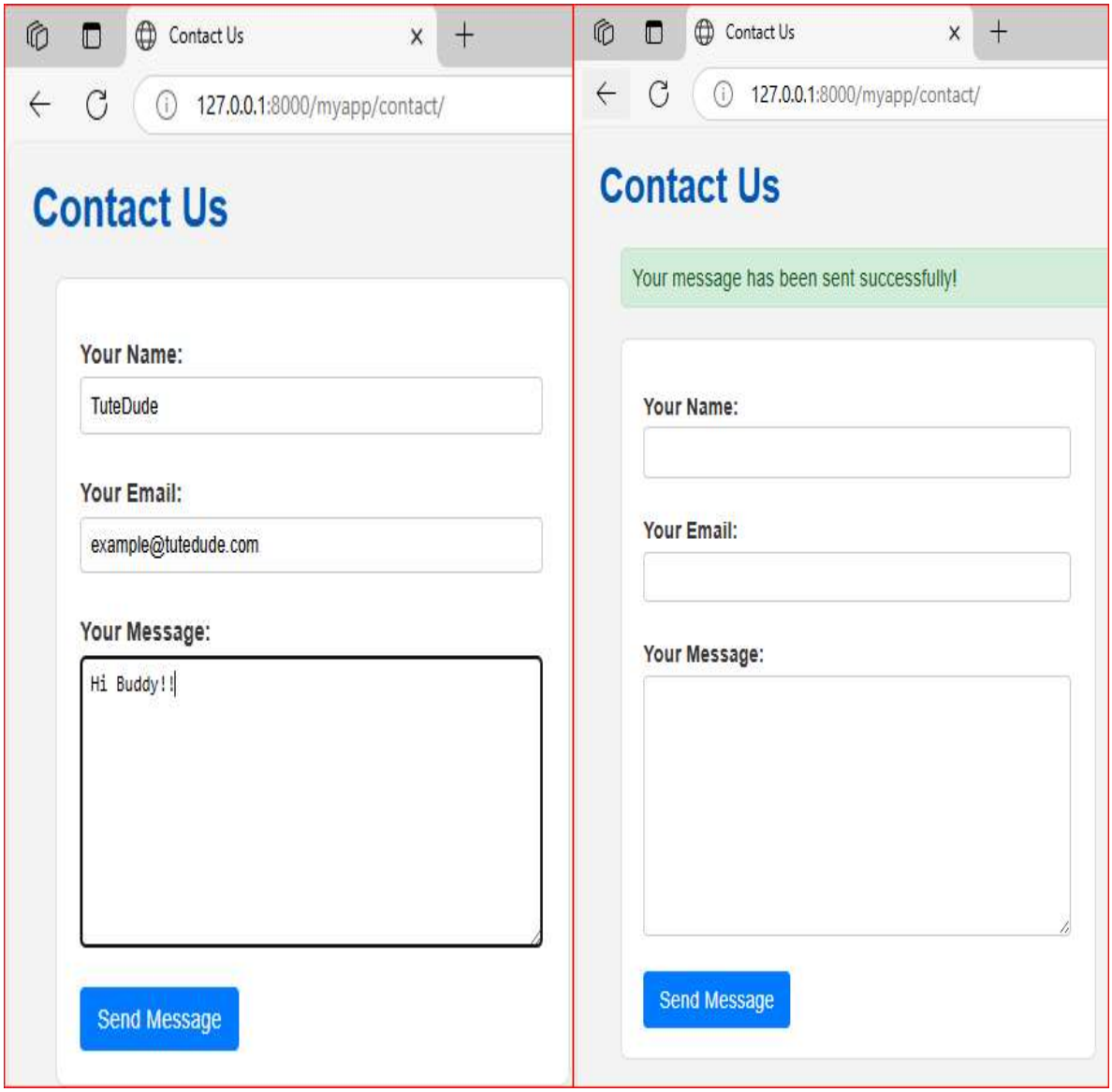
- **5. Run the Server:**

  Bash

```
(django_venv) python manage.py runserver
```

  Expected Output: Server starts.

  - **Action:** Go to `http://127.0.0.1:8000/myapp/contact/`.
  - **Browser Output:** A contact form with Name, Email, and Message fields.
  - **Action:** Fill it out and submit.
  - **Browser Output:** The page reloads, and a green "Your message has been sent successfully!" box appears at the top.
  - **Console Output:** Your terminal where the server is running will print:
  - `Contact Form Submitted:`
  - `Name: Your Name`
  - `Email: your@email.com`
  - `Message: Your message content`



Explanation: Django forms handle validation (e.g., email format) and provide cleaned data, making form processing robust.

6.2 Forms 2 (Model Forms)

- **Purpose:** Model Forms are a powerful shortcut for creating forms directly from your Django models. They automatically generate form fields based on your model's fields and handle saving data to the database.
- **Creating a `ModelForm` class:** Inherit from `django.forms.ModelForm`.
- **1. Define a Simple Model:** Let's create a `Product` model.

  File: `myapp/models.py` **Add:**

  Python

```
# myapp/models.py
```

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField(blank=True, null=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.IntegerField(default=0)
    is_available = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)

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

Explanation: Defines a `Product` model with various field types.

- **2. Make Migrations and Migrate:** Whenever you change `models.py`, you need to create and apply migrations.

  Bash

  ```
  (django_venv) python manage.py makemigrations
  ```

  Expected Output:

  ```
  Migrations for 'myapp':
    myapp\migrations\0001_initial.py
      - Create model Product
  ```

  Explanation: Django detects your new `Product` model and creates a migration file.

  Bash

  ```
  (django_venv) python manage.py migrate
  ```

  Expected Output:

  ```
  Operations to perform:
    Apply all migrations: myapp
  Running migrations:
    Applying myapp.0001_initial... OK
  ```

  Explanation: This applies the migration, creating the `myapp_product` table in your database.

- **3. Create a `ModelForm` for `Product`:**

  File: `myapp/forms.py` **Add:**

  Python

  ```
  # myapp/forms.py
  from django import forms
  from django.contrib.auth.forms import UserCreationForm
  from .models import Product # Import your Product model
  ```

```
class CustomUserCreationForm(UserCreationForm):
    class Meta(UserCreationForm.Meta):
        model = UserCreationForm.Meta.model
        fields = UserCreationForm.Meta.fields + ('email',)

class ContactForm(forms.Form):
    name = forms.CharField(max_length=100, label='Your Name')
    email = forms.EmailField(label='Your Email')
    message = forms.CharField(widget=forms.Textarea, label='Your
Message')

# New: ProductForm using ModelForm
class ProductForm(forms.ModelForm):
    class Meta:
        model = Product
        fields = ['name', 'description', 'price', 'stock',
'is_available'] # Fields to include
        # Or fields = '__all__' to include all model fields
        # Or exclude = ['created_at'] to exclude specific fields
```

Explanation:

- `class Meta:`: An inner class where you tell the `ModelForm` which `model` it's associated with and which `fields` from that model to include in the form.

- **4. Create a View to Handle Product Form:**

  File: `myapp/views.py` **Add:**

  Python

```
# myapp/views.py
from django.shortcuts import render, redirect, get_object_or_404 #
Import get_object_or_404
from .forms import CustomUserCreationForm, ContactForm, ProductForm #
Import ProductForm
from django.contrib.auth.views import LoginView as AuthLoginView,
LogoutView as AuthLogoutView
from django.urls import reverse_lazy
from django.contrib.auth.decorators import login_required
from django.contrib import messages
from .models import Product # Import Product model

# ... (existing views) ...

@login_required # Decorator to ensure user is logged in
def create_product(request):
    if request.method == 'POST':
        form = ProductForm(request.POST)
        if form.is_valid():
            form.save() # Saves the new product to the database
            messages.success(request, 'Product created
successfully!')
            return redirect('product_list') # Redirect to product
list (create later)
    else:
        form = ProductForm()
    return render(request, 'myapp/create_product.html', {'form':
form})
```

```
@login_required
def product_list(request):
    products = Product.objects.all() # Retrieve all products from the
database
    return render(request, 'myapp/product_list.html', {'products':
products})
```

Explanation:

- `@login_required`: This decorator ensures that only logged-in users can access this view. If not logged in, they are redirected to the login page.
- `form.save()`: For `ModelForm`, this method directly creates and saves a new model instance to the database.
- `Product.objects.all()`: This uses Django's ORM to fetch all `Product` objects from the database.

- **5. Create Templates for Product Forms:**

  File: `myapp/templates/myapp/create_product.html` **Add:**

  HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Create Product</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
    <style>
        /* Form styling from previous sections */
        form { margin: 20px; padding: 20px; border: 1px solid #ddd;
border-radius: 8px; background-color: #fff; max-width: 400px; }
        form p { margin-bottom: 10px; }
        form label { display: block; margin-bottom: 5px; font-weight:
bold; }
        form input[type="text"], form input[type="number"], form
input[type="checkbox"], form textarea { width: calc(100% - 20px);
padding: 8px; margin-bottom: 10px; border: 1px solid #ccc; border-
radius: 4px; }
        form input[type="checkbox"] { width: auto; margin-right: 5px;
} /* Adjust checkbox width */
        form button { background-color: #007bff; color: white;
padding: 10px 15px; border: none; border-radius: 4px; cursor:
pointer; font-size: 16px; }
        form button:hover { background-color: #0056b3; }
        .errorlist { color: red; list-style-type: none; padding: 0;
margin-top: -5px; margin-bottom: 10px; }
        .errorlist li { font-size: 0.9em; }
        .messages { list-style-type: none; padding: 0; margin: 20px;
}
        .messages li { padding: 10px; border-radius: 5px; margin-
bottom: 10px; }
        .messages .success { background-color: #d4edda; color:
#155724; border: 1px solid #c3e6cb; }
    </style>
</head>
<body>
    <h1>Create New Product</h1>
```

```
    {% if messages %}
        <ul class="messages">
            {% for message in messages %}
                <li{% if message.tags %} class="{{ message.tags }}"{%
endif %}>{{ message }}</li>
            {% endfor %}
        </ul>
    {% endif %}
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Save Product</button>
    </form>
</body>
</html>
```

File: `myapp/templates/myapp/product_list.html` **Add:**

HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Product List</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
    <style>
        table {
            width: 80%;
            border-collapse: collapse;
            margin: 20px auto;
        }
        th, td {
            border: 1px solid #ddd;
            padding: 8px;
            text-align: left;
        }
        th {
            background-color: #f2f2f2;
        }
        tr:nth-child(even) {
            background-color: #f9f9f9;
        }
    </style>
</head>
<body>
    <h1>All Products</h1>
    <p><a href="{% url 'create_product' %}">Add New Product</a></p>

    {% if products %}
        <table>
            <thead>
                <tr>
                    <th>ID</th>
                    <th>Name</th>
                    <th>Description</th>
                    <th>Price</th>
                    <th>Stock</th>
                    <th>Available</th>
                </tr>
```

```
            </thead>
            <tbody>
                {% for product in products %}
                <tr>
                    <td>{{ product.id }}</td>
                    <td>{{ product.name }}</td>
                    <td>{{ product.description|default_if_none:"N/A"
}}</td> {# default_if_none filter #}
                    <td>${{ product.price }}</td>
                    <td>{{ product.stock }}</td>
                    <td>{{ product.is_available|yesno:"Yes,No"
}}</td> {# yesno filter #}
                </tr>
                {% endfor %}
            </tbody>
        </table>
    {% else %}
        <p>No products found.</p>
    {% endif %}
</body>
</html>
```

Explanation:

- `product.id`, `product.name`: Directly access model instance attributes.
- `default_if_none:"N/A"`: DTL filter to display "N/A" if description is `None`.
- `yesno:"Yes,No"`: DTL filter to display "Yes" for `True` and "No" for `False`.

- **6. Add URLs for Products:**

  File: `myapp/urls.py` **Modify:**

  Python

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

urlpatterns = [
    path('', views.home, name='home'),
    path('register/', views.register, name='register'),
    path('login/', views.CustomLoginView.as_view(), name='login'),
    path('logout/', views.CustomLogoutView.as_view(), name='logout'),
    path('contact/', views.contact_us, name='contact_us'),
    path('products/create/', views.create_product,
name='create_product'), # New
    path('products/', views.product_list, name='product_list'), # New
]
```

- **7. Run the Server:**

  Bash

```
(django_venv) python manage.py runserver
```

  Expected Output: Server starts.

- **Action:**
    1. Go to `http://127.0.0.1:8000/myapp/login/` and log in (or register if you don't have a user).
    2. Go to `http://127.0.0.1:8000/myapp/products/create/`.
    3. **Browser Output:** A form to create a new product.
    4. **Action:** Fill out the form and submit.
    5. **Browser Output:** You'll be redirected to `http://127.0.0.1:8000/myapp/products/` (the product list page), and a "Product created successfully!" message will appear. The new product will be listed in a table.
    6. **Action:** Add more products. *Explanation:* Model Forms significantly streamline the process of creating forms that interact with your database models.

6.3 Styling Forms

- **Purpose:** To make Django-generated forms look good using CSS.
- **Methods:**
    1. **Manual CSS:** As shown in previous examples, you can write custom CSS rules to target form elements (labels, inputs, buttons, error lists). This gives you full control.
    2. **CSS Frameworks:** Integrate popular CSS frameworks like Bootstrap, Tailwind CSS, or Bulma. You'd link their CSS files in your templates and apply their classes to your form elements (often by customizing how Django renders fields).
    3. **Django Crispy Forms:** A popular third-party Django app that allows you to control the rendering behavior of your Django forms in a very elegant and DRY way. It integrates well with CSS frameworks.
- **Example (Conceptual - using existing CSS):** The `style` tags in `register.html`, `login.html`, `contact.html`, and `create_product.html` already contain basic CSS to style the forms. This is an example of **manual CSS styling**.

  HTML

```
<style>
    /* Basic styling for forms */
    form {
        margin: 20px;
        padding: 20px;
        border: 1px solid #ddd;
        border-radius: 8px;
        background-color: #fff;
        max-width: 400px;
    }
    /* ... other styles for p, label, input, button, errorlist,
messages ... */
</style>
```

  Explanation: These CSS rules target HTML elements like `form`, `label`, `input`, etc., to apply visual styles. When Django renders `{{ form.as_p }}`, it generates standard HTML `<label>` and `<input>` tags, which are then styled by your CSS.

To use a CSS framework (e.g., Bootstrap):

1. Add Bootstrap CSS link to your template's `<head>`:

   HTML

   ```
   <link
   href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/boo
   tstrap.min.css" rel="stylesheet">
   ```

2. You might then need to tell Django forms to use Bootstrap classes, which is often done with `django-crispy-forms` or by manually rendering fields with `{% for field in form %}` and adding classes.

Chapter 7: Building a Blog Application

Let's put everything together by creating a simple blog application.

7.1 Create Your Own Blog (Part 1: Models for Posts)

The core of a blog is its posts. We'll define a `Post` model.

- **1. Define the `Post` Model:**

  File: `myapp/models.py` **Add:**

  Python

  ```
  # myapp/models.py
  from django.db import models
  from django.contrib.auth.models import User # Import Django's built-
  in User model

  class Product(models.Model):
      # ... (existing Product model) ...
      pass

  # New: Blog Post Model
  class Post(models.Model):
      title = models.CharField(max_length=200, unique=True)
      slug = models.SlugField(max_length=200, unique=True) # For clean
  URLs
      author = models.ForeignKey(User, on_delete=models.CASCADE,
  related_name='blog_posts')
      content = models.TextField()
      created_on = models.DateTimeField(auto_now_add=True)
      updated_on = models.DateTimeField(auto_now=True)
      status = models.IntegerField(choices=((0, 'Draft'), (1,
  'Published')), default=0)

      class Meta:
  ```

```
        ordering = ['-created_on'] # Order posts by creation date,
newest first

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

Explanation:

- `title`, `content`: Standard text fields.
- `slug`: A short label for an item, used in URLs. `unique=True` ensures no two posts have the same slug.
- `author = models.ForeignKey(User, ...)`: This creates a many-to-one relationship. A `Post` has one `author` (a `User`), but a `User` can have many `Posts`.
  - `on_delete=models.CASCADE`: If the `User` is deleted, all their associated posts are also deleted.
  - `related_name='blog_posts'`: Allows you to access a user's posts via `user.blog_posts.all()`.
- `status`: An integer field with `choices` to define 'Draft' (0) or 'Published' (1).
- `created_on`, `updated_on`: Timestamps. `auto_now_add=True` sets the creation time once, `auto_now=True` updates it every time the object is saved.
- `class Meta: ordering = ['-created_on']`: Default ordering for queries, newest posts first.

- **2. Register Model in Admin:** This allows you to create and manage blog posts through the Django admin interface.

  File: `myapp/admin.py` **Add:**

  Python

```
# myapp/admin.py
from django.contrib import admin
from .models import Product, Post # Import Post

# Register your models here.
admin.site.register(Product)

# New: Register Post model
@admin.register(Post) # Alternative way to register using decorator
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'status', 'created_on') #
Columns to display in list view
    list_filter = ("status",) # Filter options in sidebar
    search_fields = ['title', 'content'] # Fields to search by
    prepopulated_fields = {'slug': ('title',)} # Automatically fill
slug based on title
```

  Explanation:

  - `admin.site.register(Post)`: The simplest way to register a model.
  - `@admin.register(Post)`: A decorator alternative for registering with custom `ModelAdmin` options.
  - `list_display`: Columns shown in the admin list view for posts.

- `list_filter`: Adds a sidebar filter for `status`.
- `search_fields`: Enables search functionality in the admin.
- `prepopulated_fields`: Automatically generates the `slug` from the `title` as you type in the admin.

- **3. Make Migrations and Migrate:**

  Bash

  ```
  (django_venv) python manage.py makemigrations
  ```

  Expected Output:

  ```
  Migrations for 'myapp':
    myapp\migrations\0002_post.py
      - Create model Post
  ```bash
  (django_venv) python manage.py migrate
  ```

  Expected Output:

  ```
  Operations to perform:
    Apply all migrations: myapp
  Running migrations:
    Applying myapp.0002_post... OK
  ```

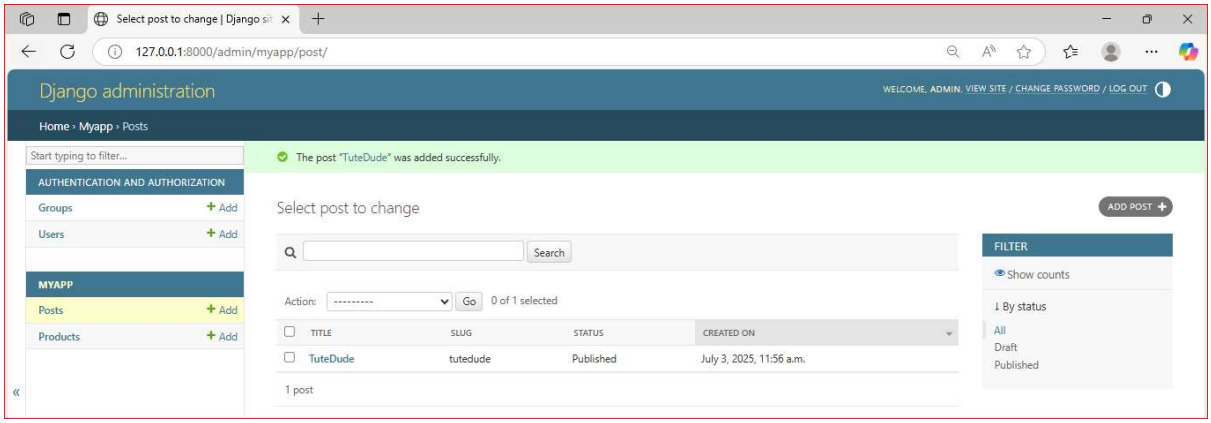
  Explanation: The `Post` table is now created in your database.

- **4. Run the Server and Verify:**

  Bash

  ```
  (django_venv) python manage.py runserver
  ```

  - **Action:** Go to `http://127.0.0.1:8000/admin/` and log in.
  - **Browser Output:** You should now see "Posts" listed under `MYAPP`.
  - **Action:** Click on "Posts" and then "Add post" to create a few sample blog posts. Make sure to set an author and status. *Explanation:* You can now manage your blog posts directly through Django's powerful admin interface.


  ```

7.2 Create Your Own Blog (Part 2: Views for Listing and Detail)

We need views to display a list of all published blog posts and a detail page for each individual post.

- **1. Create Blog List View:**

  File: `myapp/views.py` **Add:**

  Python

  ```
  # myapp/views.py
  from django.shortcuts import render, redirect, get_object_or_404
  from .forms import CustomUserCreationForm, ContactForm, ProductForm
  from django.contrib.auth.views import LoginView as AuthLoginView,
  LogoutView as AuthLogoutView
  from django.urls import reverse_lazy
  from django.contrib.auth.decorators import login_required
  from django.contrib import messages
  from .models import Product, Post # Import Post model

  # ... (existing views) ...

  def post_list(request):
      # Retrieve only published posts, ordered by creation date
      posts = Post.objects.filter(status=1).order_by('-created_on')
      return render(request, 'myapp/post_list.html', {'posts': posts})

  def post_detail(request, slug): # Takes 'slug' as an argument from
  URL
      post = get_object_or_404(Post, slug=slug, status=1) # Get post by
  slug, ensure it's published
      return render(request, 'myapp/post_detail.html', {'post': post})
  ```

  Explanation:

  - `Post.objects.filter(status=1)`: Retrieves only posts where `status` is 1 (Published).
  - `.order_by('-created_on')`: Orders the results by `created_on` in descending order (newest first).
  - `get_object_or_404(Post, slug=slug, status=1)`: This is a shortcut. It tries to get a `Post` object matching the `slug` and `status=1`. If no such object exists, it raises an `Http404` error (which Django handles by showing a "Page not found" error).
- **2. Add URLs for Blog Posts:**

  File: `myapp/urls.py` **Modify:**

  Python

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

```
urlpatterns = [
    path('', views.home, name='home'),
    path('register/', views.register, name='register'),
    path('login/', views.CustomLoginView.as_view(), name='login'),
    path('logout/', views.CustomLogoutView.as_view(), name='logout'),
    path('contact/', views.contact_us, name='contact_us'),
    path('products/create/', views.create_product,
name='create_product'),
    path('products/', views.product_list, name='product_list'),
    path('blog/', views.post_list, name='post_list'), # New: Blog
list
    path('blog/<slug:slug>/', views.post_detail, name='post_detail'),
# New: Blog detail
]
```

Explanation:

- `path('blog/<slug:slug>/', ...)`: This defines a URL pattern that captures a "slug" from the URL.
  - `<slug:slug>`: This is a path converter. It tells Django to capture the part of the URL that matches a slug pattern (letters, numbers, hyphens, underscores) and pass it as a keyword argument named `slug` to the `post_detail` view.

7.3 Create Your Own Blog (Part 3: Templates for Listing and Detail)

Finally, we create the HTML templates to display our blog posts.

- **1. Create Blog List Template:**

  File: `myapp/templates/myapp/post_list.html` **Add:**

  HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Blog Posts</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
    <style>
        .post-summary {
            background-color: #fff;
            border: 1px solid #ddd;
            padding: 15px;
            margin-bottom: 20px;
            border-radius: 8px;
        }
        .post-summary h2 {
            margin-top: 0;
            color: #007bff;
        }
        .post-summary p {
            font-size: 0.9em;
            color: #666;
        }
```

```
        .post-summary a {
            text-decoration: none;
            color: #007bff;
        }
        .post-summary a:hover {
            text-decoration: underline;
        }
    </style>
</head>
<body>
    <h1>Latest Blog Posts</h1>

    {% if posts %}
        {% for post in posts %}
            <div class="post-summary">
                <h2><a href="{% url 'post_detail' slug=post.slug
%}">{{ post.title }}</a></h2>
                <p>By {{ post.author.username }} on {{
post.created_on|date:"F j, Y" }}</p>
                <p>{{ post.content|truncatechars:200 }}</p> {#
truncatechars filter to show summary #}
                <p><a href="{% url 'post_detail' slug=post.slug
%}">Read More</a></p>
            </div>
        {% endfor %}
    {% else %}
        <p>No blog posts found.</p>
    {% endif %}
</body>
</html>
```

Explanation:

- `{% url 'post_detail' slug=post.slug %}`: Generates the URL for the detail page, passing the post's `slug` as the argument required by the URL pattern.
- `post.author.username`: Accesses the username of the related `User` object.
- `truncatechars:200`: DTL filter to limit the content displayed to 200 characters, adding "..." if truncated.

- **2. Create Blog Detail Template:**

  File: `myapp/templates/myapp/post_detail.html` **Add:**

  HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>{{ post.title }}</title>
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">
    <style>
        .post-content {
            background-color: #fff;
            border: 1px solid #ddd;
            padding: 20px;
            margin: 20px;
            border-radius: 8px;
```

```
        }
        .post-content h1 {
            color: #0056b3;
            margin-top: 0;
        }
        .post-meta {
            font-size: 0.9em;
            color: #666;
            margin-bottom: 20px;
            border-bottom: 1px solid #eee;
            padding-bottom: 10px;
        }
        .post-body {
            line-height: 1.6;
        }
    </style>
</head>
<body>
    <div class="post-content">
        <h1>{{ post.title }}</h1>
        <p class="post-meta">By {{ post.author.username }} on {{
post.created_on|date:"F j, Y, P" }}</p>
        <div class="post-body">
            {{ post.content|linebreaksbr }} {# linebreaksbr filter
converts newlines to <br> tags #}
        </div>
        <p><a href="{% url 'post_list' %}">Back to Blog List</a></p>
    </div>
</body>
</html>
```

Explanation:

- `linebreaksbr`: DTL filter that converts newlines in the `content` field into HTML `<br>` tags, preserving line breaks.

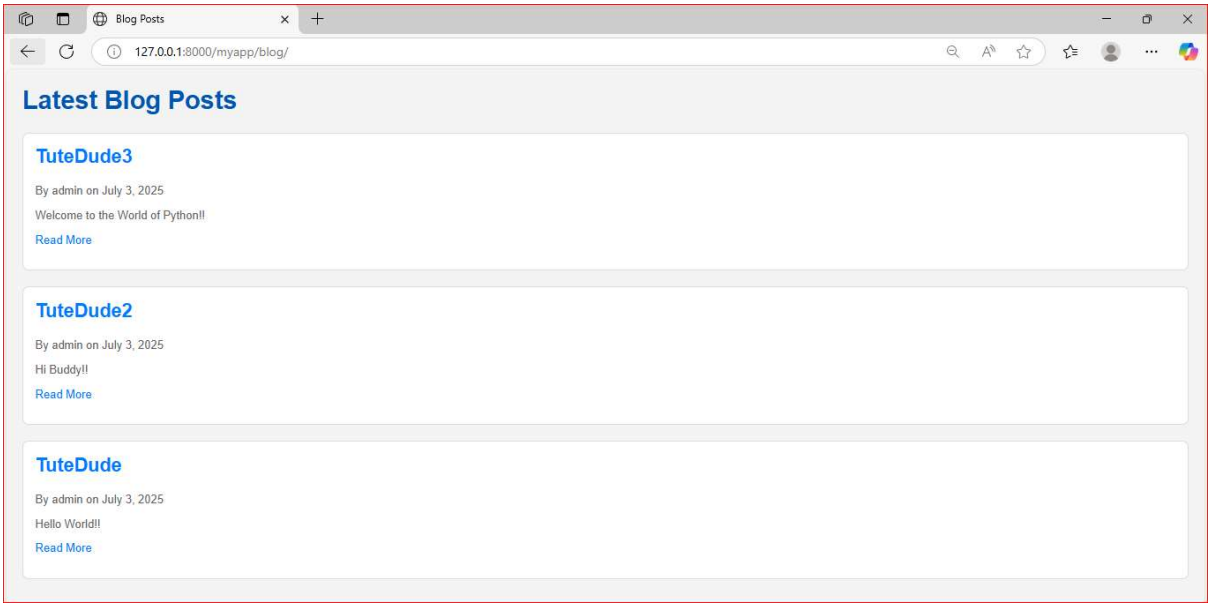
- **3. Run the Server:**

  Bash

  ```
  (django_venv) python manage.py runserver
  ```

  Expected Output: Server starts.

  - **Action:**
    1. Go to `http://127.0.0.1:8000/admin/` and log in.
    2. Go to `Posts` and create a few posts. Make sure to set their `status` to `Published`.
    3. Go to `http://127.0.0.1:8000/myapp/blog/`.
    4. **Browser Output:** A list of your published blog posts, each with a title, author, date, a truncated content summary, and a "Read More" link.
    5. **Action:** Click on a "Read More" link.
    6. **Browser Output:** You'll see the full content of that specific blog post on its detail page. *Explanation:* You've successfully built a basic blog application with list and detail views, dynamic content from the database, and clean URLs.


Blog Posts
127.0.0.1:8000/myapp/blog/
Latest Blog Posts
TuteDude3
By admin on July 3, 2025
Welcome to the World of Python!!
Read More
TuteDude2
By admin on July 3, 2025
Hi Buddy!!
Read More
TuteDude
By admin on July 3, 2025
Hello World!!
Read More

END