User Authentication in Flask:

Authentication vs Authorization

1. **Authentication**:
    - Authentication is the process of verifying the identity of a user.
    - The goal is to ensure that the user is who they say they are.
    - Typically, this involves the user providing credentials (like a username/email and password) which are checked against a stored set of data (e.g., a database).
2. **Authorization**:
    - Authorization is the process of granting or denying access to resources or actions based on the authenticated user's roles or permissions.
    - Once a user is authenticated, authorization determines what they can do within the application (e.g., view a specific page, edit data, or delete something).

How to Implement Authentication in Flask

- Flask provides a simple way to implement authentication using several tools and extensions, such as **Flask-Login** for session management and **Flask-Bcrypt** for password hashing.

Steps to Implement Authentication:

1. **Install Necessary Extensions**:
    - `Flask-Login`: Manages user sessions.
    - `Flask-Bcrypt`: Used for securely hashing passwords.

    ```
    pip install flask
    pip install flask-sqlalchemy
    pip install flask-login
    pip install flask-bcrypt
    ```

2. **Set Up Flask-Login**:
    - `Flask-Login` handles user session management, including login and logout.
    - It stores user session data in cookies for the duration of the session.

- Flask-Login provides a simplified way of managing users, which includes easily logging in and out users, as well as restricting certain pages to authenticated users.
- Manages user authentication, session handling, and login/logout functionality.

Example:

app.py:

```
from flask import Flask, render_template, redirect, url_for, request,
flash
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt
from flask_login import LoginManager, UserMixin, login_user,
login_required, logout_user, current_user
import os


app = Flask(__name__)

basedir = os.path.abspath(os.path.dirname(__file__))




# Configure the database
app.config['SQLALCHEMY_DATABASE_URI'] =
"sqlite:///"+os.path.join(basedir,"app.db")

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# app.secret_key = 'your_secret_key'

app.config['SECRET_KEY'] = 'your_secret_key'

db = SQLAlchemy(app)
bcrypt = Bcrypt(app) #Enables password hashing.
login_manager = LoginManager()#Initializes the login system.
login_manager.init_app(app)    #Explicitly binds the LoginManager to the
Flask app
```

```python
login_manager.login_view = 'login' #Redirects unauthorized users to the
login page.


class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(150), nullable=False)
    email = db.Column(db.String(150), unique=True, nullable=False)
    password_hash = db.Column(db.String(256), nullable=False)
    mobile = db.Column(db.String(15), nullable=False)
    role = db.Column(db.String(50), nullable=False, default='user')

    def set_password(self, password):
        self.password_hash = bcrypt.generate_password_hash(
            password)


    def check_password(self, password):
        return bcrypt.check_password_hash(self.password_hash, password)


@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))


with app.app_context():
    db.create_all()
```

- Here We need to specify a secret key, which can be any random string of characters, and is necessary as Flask-Login requires it to sign session cookies for protection against data tampering. Next, we need to initialize the *LoginManager* class from Flask-Login, to be able to log in and out users.

  .

1. **bcrypt:**

- The `bcrypt` is a password-hashing library used for securely hashing passwords before storing them in a database. It is widely used for creating hashes that are computationally expensive to generate, making them harder to crack using brute force attacks.
- **Password Hashing**: When you use `bcrypt` to hash passwords, it applies a process known as **key stretching** (repeated hashing) to make the hash more secure. This means that the hash is not easily reverse-engineered.

  Setting a Password:

  ```
  def set_password(self, password):
          self.password_hash = bcrypt.generate_password_hash(
              password)
  ```

  - The `generate_password_hash()` function takes a password and returns a hashed version of the password.

  Checking a Password:

```
def check_password(self, password):
    return bcrypt.check_password_hash(self.password_hash, password)
```

  - The `check_password_hash()` function compares the stored hash with the hash generated from the entered password.

2. **UserMixin**:

- `UserMixin` is a class from `Flask-Login` library that provides default implementations of the required user authentication methods, such as `is_authenticated`, `is_active`, `is_anonymous`, and `get_id()`.

- It provides the following default behaviors:
    - `is_authenticated()`: Returns `True` if the user is authenticated.
    - `is_active()`: Returns `True` if the user is active.
    - `is_anonymous()`: Returns `True` if the user is anonymous (not logged in).
    - `get_id()`: Returns the unique identifier of the user, typically their `user_id`.

Example:

```
class User(db.Model, UserMixin):
```

3. **Flask-Login and LoginManager:**

- `Flask-Login` is an extension that helps manage user sessions for Flask applications. It provides the tools needed to handle user login/logout and authentication.
- LoginManager: This is the core of Flask-Login, managing the session and handling user loading.

Initialization:

```
login_manager = LoginManager()
login_manager.init_app(app)
```

- The `LoginManager()` is initialized and linked to your Flask app using `init_app(app)`.

Login View:

- This tells `Flask-Login` which view(function) should be called when a user is not logged in.

```
login_manager.login_view = 'login'
```

- It **sets the default route** where unauthenticated users will be redirected when they try to access a `@login_required` protected page.

- `"login"` refers to the **function name** (view function) of your login route in the **app.py** file.

4. **User Loader:**

- Flask-Login uses the `user_loader` callback to load a user object from a database or other storage. It is responsible for fetching the user based on their `user_id`.

```
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

- This function takes the `user_id` (which is stored in the session) when the user logged in for the first time, and queries the database to return the corresponding `User` object.

- Flask-Login keeps track of the logged-in user by storing their user ID in the session.
- However, Flask-Login doesn't know how to retrieve a user from the database based on this stored ID.
- The `@login_manager.user_loader` decorator defines a function that tells Flask-Login how to load a user object when needed.
- Flask-Login provides a `current_user` object, which represents the currently logged-in user.
- When you access `current_user`, Flask-Login calls `load_user(user_id)` internally.

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

```python
@app.route("/register", methods=["GET", "POST"])
def register():
    if request.method == "POST":
        name = request.form.get("name")
        email = request.form.get("email")
        password = request.form.get("password")
        confirm_password = request.form.get("confirm_password")
        mobile = request.form.get("mobile")
        role = request.form.get("role")

        # Check if passwords match
        if password != confirm_password:
            flash("Passwords do not match!", "danger")
            return redirect(url_for("register"))

        # Check if the email already exists
        if User.query.filter_by(email=email).first():
            flash("Email already exists!", "danger")
            return redirect(url_for("register"))

        new_user = User(name=name, email=email, mobile=mobile, role=role)
        new_user.set_password(password)
        db.session.add(new_user)
        db.session.commit()

        flash("Registration successful! Please log in.", "success")
        return redirect(url_for("login"))

    return render_template('register.html')


@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        email = request.form.get("email")
        password = request.form.get("password")
        role = request.form.get("role")

        user = User.query.filter_by(email=email, role=role).first()
        if user and user.check_password(password):
            login_user(user)
            flash("Login successful!", "success")
            return redirect(url_for("dashboard"))
        else:
            flash("Invalid credentials!", "danger")

    return render_template("login.html")


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


@app.route("/logout")
@login_required
def logout():
    logout_user()
    flash("Logged out successfully!", "info")
    return redirect(url_for("login"))


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


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

5. **login_user:**

- `login_user()` is a function provided by `Flask-Login` to log a user in. It takes the user object as an argument and stores the user's information in the session, effectively logging them in.

  Example:

  ```
  login_user(user)
  ```

- This function should be called after successfully verifying a user's credentials (e.g., email and password). It manages the user session and redirects the user to a protected page.

6. **login_required:**
   - `login_required` is a decorator provided by `Flask-Login` that ensures the user is authenticated before they can access a specific route. If the user is not logged in, they will be redirected to the login page.

   Example:

   ```
   @app.route("/dashboard")
   @login_required
   def dashboard():
       return render_template("dashboard.html")
   ```

7. **logout_user:**
   - `logout_user()` is a function provided by `Flask-Login` to log the user out. It removes the user's information from the session, effectively ending the session.

   Example:

   ```
   @app.route("/logout")
   @login_required
   ```

```
def logout():
    logout_user()
    flash("Logged out successfully!", "info")
    return redirect(url_for("login"))
```

- Calling `logout_user()` will log the user out and redirect them to a different page (e.g., the login page) after a successful logout.

8. **current_user:**
    - `current_user` is a proxy provided by `Flask-Login` that allows you to access the currently logged-in user. It represents the user object for the currently authenticated user.

      Example:

      ```
      <h2>Profile of {{ current_user.name }}</h2>
      <p>Email: {{ current_user.email }}</p>
      <p>Role: {{ current_user.role }}</p>
      ```

Final Application:

Folder Structure:

FlaskAuthenticationApp:

```
|
|--app.py
|--app.db
|
|--templates/
        |
```

```
|--base.html
|--index.html
|--register.html
|--login.html
|--dashboard.html
|--profile.html
```

app.py:

```
from flask import Flask, render_template, redirect, request, url_for,
flash
from flask_sqlalchemy import SQLAlchemy
import os
from flask_login import LoginManager, UserMixin, login_user, logout_user,
login_required, current_user
from flask_bcrypt import Bcrypt


basedir = os.path.abspath(os.path.dirname(__file__))

app = Flask(__name__)

-
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///" + \
    os.path.join(basedir, "app.db")
app.config["SQLALCHEMY_TRACK_MODIFICATION"] = False

app.config["SECRET_KEY"] = "Your secret key"


db = SQLAlchemy(app)

bcrypt = Bcrypt(app)
login_manager = LoginManager()
```

```python
login_manager.init_app(app)

login_manager.login_view = "login"


class User(db.Model, UserMixin):

    __tablename__ = "user"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(100), nullable=False, unique=True)
    password_hash = db.Column(db.String(100), nullable=False)
    mobile = db.Column(db.String(15), nullable=False)
    role = db.Column(db.String(50), nullable=False, default="user")

    def set_password(self, password):
        self.password_hash = bcrypt.generate_password_hash(password)

    def check_password(self, password):
        return bcrypt.check_password_hash(self.password_hash, password)


@login_manager.user_loader
def load_user(user_id):
    return db.session.get(User, int(user_id))


with app.app_context():
    db.create_all()


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


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


@app.route("/login", methods=["GET", "POST"])
def login():

    if request.method == "POST":
        email = request.form.get("email")
        password = request.form.get("password")
        role = request.form.get("role")

        user = User.query.filter_by(email=email, role=role).first()
        if user and user.check_password(password):
            login_user(user)
            flash("Login successful!", "success")
            return redirect(url_for("dashboard"))
        else:
            flash("Invalid credentials!", "danger")

    return render_template("login.html")


@app.route("/register", methods=["GET", "POST"])
def register():

    if request.method == "POST":
        name = request.form.get("name")
        email = request.form.get("email")
        password = request.form.get("password")
        confirm_password = request.form.get("confirm_password")
        mobile = request.form.get("mobile")

        # Check if passwords match
        if password != confirm_password:
            flash("Passwords do not match!", "danger")
            return redirect(url_for("register"))

        # Check if the email already exists
        if User.query.filter_by(email=email).first():
            flash("Email already exists!", "danger")
            return redirect(url_for("register"))

        new_user = User(name=name, email=email, mobile=mobile)
        new_user.set_password(password)
        db.session.add(new_user)
        db.session.commit()

        flash("Registration successful! Please log in.", "success")
        return redirect(url_for("login"))

    return render_template("/register.html")


@app.route("/logout")
@login_required
def logout():
    logout_user()
    flash("Logged out successfully!", "info")
    return redirect(url_for("login"))


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


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

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_block %} {% endblock %}</title>


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

integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6h
W+ALEwIH" crossorigin="anonymous">


    <style>
        body {
            margin: 0;
            padding: 0;
        }


        header {
            height: 10vh;
        }


        main {
            height: 80vh;
        }


        footer {
            height: 10vh;
        }
    </style>

</head>


<body>
```

```
    <header class="bg-info ">

        <ul class="nav justify-content-end">
            <li class="nav-item">
                <a class="nav-link " aria-current="page" href="#">Flask
Auth App</a>
            </li>

            {% if current_user.is_authenticated %}

            <li class="nav-item">
                <a class="nav-link"
href="{{url_for('dashboard')}}">Dashboard</a>
            </li>
            <li class="nav-item">
                <a class="nav-link"
href="{{url_for('logout')}}">Logout</a>
            </li>

            {% else %}

            <li class="nav-item">
                <a class="nav-link"
href="{{url_for('register')}}">Register</a>
            </li>

            <li class="nav-item">
                <a class="nav-link" href="{{url_for('login')}}">Login</a>
            </li>

            {% endif %}
        </ul>


    </header>


    <!-- Page Specific content -->
    <main class="bg-success overflow-auto">

        <!-- Flash Message Display -->

        {% for category,messages in
get_flashed_messages(with_categories=True) %}

        <div class="alert alert-{{category}} alert-dismissible fade show"
role="alert">
            {{messages}}
            <button type="button" class="btn-close"
data-bs-dismiss="alert" aria-label="Close"></button>
        </div>


        {% endfor %}


        {% block main_block %}


        {% endblock %}


    </main>


    <!-- Footer -->
    <footer class="bg-danger">
        <p class="text-center">&copy; 2025 Flask Auth System</p>
    </footer>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle
.min.js"

integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN
7N6jIeHz"
        crossorigin="anonymous"></script>
</body>


</html>
```

index.html:

```
<!-- index.html -->
{% extends 'base.html' %}

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

{% block main_block %}
<div class="text-center">
    <h1>Welcome to Flask Authentication System</h1>
    <p>Please login or register to continue.</p>
</div>
{% endblock %}
```

register.html:

```
<!-- register.html -->
{% extends 'base.html' %}

{% block title_block %}Registeration Page{% endblock %}


{% block main_block %}
```

```
<div class="container">
    <h2>Register</h2>
    <form action="{{url_for('register')}}" method="POST">


        <div class="mb-3">
            <label for="name">Enter Full Name:</label>
            <input type="text" name="name" id="name" class="form-control"
placeholder="Full Name" required>
        </div>


        <div class="mb-3">
            <label for="email">Enter Username:</label>
            <input type="email" name="email" id="email"
class="form-control" placeholder="Email" required>
        </div>


        <div class="mb-3">
            <label for="password">Enter Password:</label>
            <input type="password" name="password" id="password"
class="form-control" placeholder="Password" required>
        </div>


        <div class="mb-3">
            <label for="confirm_password">Confirm Password:</label>
            <input type="password" name="confirm_password"
id="confirm_password" class="form-control"
                placeholder="Confirm Password" required>
        </div>


        <div class="mb-3">
            <label for="mobile">Enter Mobile Number:</label>
            <input type="text" name="mobile" id="mobile"
class="form-control" placeholder="Mobile Number" required><br>
```

```
        </div>


        <input type="submit" class="btn btn-primary" value="Register">
    </form>


    <p>Already registered? <a href="{{ url_for('login') }}">Login
here</a></p>


</div>
{% endblock %}
```

login.html:

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

{% block title_block %}Login Page{% endblock %}

{% block main_block %}
<h2>Login</h2>
<div class="container">
    <form action="{{url_for('login')}}" method="POST">


        <div class="mb-3">
            <label for="username">Enter Username:</label>
            <input type="email" name="email" id="username"
class="form-control" placeholder="Email" required>
        </div>


        <div class="mb-3">
            <label for="password">Enter Password:</label>
            <input type="password" name="password" id="password"
class="form-control" placeholder="Password" required>
```

```
        </div>


        <div class="mb-3">
            <label for="role">Select Role:</label>
            <select name="role" id="role" class="form-control" required>
                <option value="">Choose Role</option>
                <option value="user">User</option>
                <option value="admin">Admin</option>
            </select>
        </div>


        <div class="mb-3">
            <input type="submit" class="btn btn-primary" value="Login">
        </div>
    </form>
    <p>New User? <a href="{{ url_for('register') }}">Register
first</a></p>
</div>


{% endblock %}
```

dashboard.html:

```
<!-- dashboard.html -->
{% extends 'base.html' %}

{% block title_block %}Dashboard{% endblock %}

{% block main_block %}
<div class="text-center">
    <h2>Welcome, {{ current_user.name }}!</h2>
    <p>Your role: <strong>{{ current_user.role }}</strong></p>
    <hr>
    <a href="{{ url_for('profile') }}" class="btn btn-primary">View
Profile</a>
```

```
    <a href="{{ url_for('logout') }}" class="btn btn-danger">Logout</a>
</div>
{% endblock %}
```

profile.html:

```
<!-- profile.html -->
{% extends 'base.html' %}

{% block title_block %}Profile Page{% endblock %}

{% block main_block %}
<h2>Profile of {{ current_user.name }}</h2>
<hr>
<div class="container">
    <p><strong>Name:</strong> {{ current_user.name }}</p>
    <p><strong>Email:</strong> {{ current_user.email }}</p>
    <p><strong>Mobile:</strong> {{ current_user.mobile }}</p>
    <p><strong>Role:</strong> {{ current_user.role }}</p>
</div>
<hr>
<a href="{{ url_for('dashboard') }}" class="btn btn-secondary">Back to
Dashboard</a>
{% endblock %}
```

Note: To Create a User with Role Admin define the following code inside the app.py file of the above application:

```
with app.app_context():
    db.create_all()

    # Check if an admin user already exists
    if not User.query.filter_by(role="admin").first():
        admin_user = User(name="Admin", email="admin@gmail.com",
mobile="1234567890", role="admin")
        admin_user.set_password("admin123")  # Set a default password
        db.session.add(admin_user)
        db.session.commit()
        print("Admin user created with email: admin@gmail.com and
password: admin123")
```

Implementing Role based authorization:

- Role-based authorization is a security mechanism that controls access to different parts of a Flask application based on a user's role. This ensures that only authorized users can access certain routes or perform specific actions.

Use Case

For example, in a web application:

- **Admin** users can access the admin panel and manage users.
- **Regular** users can access their profiles and perform limited actions.
- **Guests** may only view public content without logging in.

Restricting the access to certain routes based on the user role:

- Steps to use the role based authentication inside the above application:

Step1: Creating a Custom `admin_required` Decorator inside the **app.py** file

```
from functools import wraps

def admin_required(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if current_user.role != 'admin':
            flash("Access denied!", "danger")
            return redirect(url_for('dashboard'))
        return func(*args, **kwargs)
    return wrapper
```

Explanation:

- **Importing the required modules:**
    - Before defining the `admin_required` decorator, you need to import `wraps` from the `functools` module:

      ```
      from functools import wraps
      ```

    - `wraps(func)` ensures that the decorated function retains its original name, docstring, and attributes.
- **Defining the `admin_required` Decorator:**
    - This function acts as a **decorator** that will wrap other route functions.

      ```
          def admin_required(func):
      ```

    - It takes a function (`func`) as an argument, which represents the protected view (e.g., an admin dashboard).
- **Creating the Inner Wrapper Function:**

  ```
  @wraps(func)#Preserves the metadata of the original function
  def wrapper(*args, **kwargs):
  ```

    - The `wrapper` function is the actual function that gets executed instead of the original function.
    - `@wraps(func)` ensures that `func` retains its original properties.

- **Checking the User Role:**

```
 if current_user.role != 'admin':
```

    - current_user is provided by Flask-Login, representing the currently logged-in user.
    - It checks if the logged-in user's role is not "admin".

- **Denying Access for Non-Admin Users:**

```
 flash("Access denied!", "danger")
return redirect(url_for('dashboard'))
```

    - If the user is not an admin, a flash message ("Access denied!") is displayed.
    - The user is redirected to the dashboard instead of being allowed to access the protected page.

- **Executing the Original Function for Admins:**

```
 return func(*args, **kwargs)
```

    - If the user is an admin, the original function (func) is executed normally.

- **Returning the Wrapper Function:**

```
return wrapper
```

    - The wrapper function is returned, effectively replacing the original function with the decorated one.

Step2: Use this decorator on any Flask route that should be restricted to admins only:

- Inside the app.py define one route which should be restricted to admin only:

```
@app.route('/admin')
@login_required
@admin_required
def admin():
    return "Welcome to the admin panel!"
```

Step 3: To dynamically show/hide menu links based on roles, modify the navigation bar (`base.html`):

```
<header class="bg-info ">

        <ul class="nav justify-content-end">

            <li class="nav-item">

      <a class="nav-link " aria-current="page" href="#">Flask Auth App</a>

            </li>

            {% if current_user.is_authenticated %}

            <li class="nav-item">

        <a class="nav-link" href="{{url_for('dashboard')}}">Dashboard</a>

            </li>

            <li class="nav-item">

            <a class="nav-link" href="{{url_for('logout')}}">Logout</a>

            </li>


            {% if current_user.role == "admin" %}

            <li class="nav-item">

                <a class="nav-link" href="{{url_for('admin')}}">Admin</a>
            </li>

            {% endif %}

            {% else %}

            <li class="nav-item">

         <a class="nav-link" href="{{url_for('register')}}">Register</a>

            </li>

            <li class="nav-item">

                <a class="nav-link" href="{{url_for('login')}}">Login</a>

            </li>

            {% endif %}

        </ul>

    </header>
```

Student Task: Combine the above FlaskAuthenticationApp with the Product Management Application such a way that the Product home page should be accessible only after the successful login and product delete can be done only by the admin.