**WEEK 6**

**Pattern Usage Report for University Social Media App**

This report outlines the design patterns applied in the university social media app, including their explanations, usage, benefits, trade-offs, and code samples with justifications. The patterns used are Observer, Singleton, and Factory, integrated into the app’s Django-based architecture to enhance modularity, scalability, and maintainability.

**1. Observer Pattern**

**Explanation**

The Observer pattern is a behavioral design pattern where an object (subject) maintains a list of dependents (observers) and notifies them of state changes. It enables a one-to-many dependency, allowing multiple objects to react to changes in the subject.

**How and Where It Is Used**

The Observer pattern is used in the real-time notification system. When a user posts, comments, or sends a direct message, the system notifies relevant users (e.g., post author, tagged users). Django Channels and Redis facilitate this by managing WebSocket connections, where the NotificationSubject (subject) tracks user actions, and NotificationObserver (observer) instances handle updates for connected clients.

* **Location**: Notification module (notifications/consumers.py and notifications/models.py).
* **Usage**: When a user performs an action (e.g., commenting), the NotificationSubject triggers an update, and observers (user sessions) receive notifications via WebSockets.

**Benefits**

* **Decoupling**: Subjects and observers are loosely coupled, allowing independent modification.
* **Scalability**: Easily add new observers (e.g., new notification types) without altering the subject.
* **Real-Time Updates**: Supports dynamic, real-time notifications for user interactions.

**Trade-offs**

* **Complexity**: Managing WebSocket connections and observer lists increases implementation complexity.
* **Performance**: Notifying many observers can strain resources, especially with high user activity.
* **Memory Leaks**: Improper observer deregistration may lead to memory issues.

**Code Sample and Justification**

The following code in notifications/consumers.py demonstrates the Observer pattern for real-time notifications using Django Channels. The NotificationConsumer acts as an observer, listening for updates from the subject (handled by Redis pub/sub).

from channels.generic.websocket import AsyncWebsocketConsumer

import json

class NotificationConsumer(AsyncWebsocketConsumer):

async def connect(self):

self.user = self.scope["user"]

self.group\_name = f"user\_{self.user.id}"

await self.channel\_layer.group\_add(self.group\_name, self.channel\_name)

await self.accept()

async def disconnect(self, close\_code):

await self.channel\_layer.group\_discard(self.group\_name, self.channel\_name)

async def notify(self, event):

message = event["message"]

await self.send(text\_data=json.dumps({"message": message}))

**Justification**: This code implements the Observer pattern by allowing user sessions to subscribe to notifications via WebSocket groups. When a subject (e.g., a post creation event) publishes a message to the group, all connected observers receive it. This decouples the notification logic from the action trigger, enabling scalable real-time updates.

**2. Singleton Pattern**

**Explanation**

The Singleton pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to it. It is useful for managing shared resources, such as database connections.

**How and Where It Is Used**

The Singleton pattern is applied to manage the SQLite database connection in the Django app. A DatabaseConnection class ensures a single connection instance is reused across the application, optimizing resource usage.

* **Location**: Database utility module (core/utils/database.py).
* **Usage**: The DatabaseConnection class initializes a single SQLite connection, accessed by all database operations (e.g., user queries, post retrieval).

**Benefits**

* **Resource Efficiency**: Prevents multiple database connections, reducing memory and overhead.
* **Consistency**: Ensures all parts of the app use the same connection, avoiding conflicts.
* **Global Access**: Simplifies access to the database connection across modules.

**Trade-offs**

* **Global State**: Can lead to hidden dependencies, making testing and debugging harder.
* **Scalability Limits**: A single connection may bottleneck in high-concurrency scenarios.
* **Thread Safety**: Requires careful handling in multi-threaded environments like Django.

**Code Sample and Justification**

The following code in core/utils/database.py implements the Singleton pattern for the SQLite database connection.

import sqlite3

class DatabaseConnection:

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super(DatabaseConnection, cls).\_\_new\_\_(cls)

cls.\_instance.connection = sqlite3.connect("db.sqlite3", check\_same\_thread=False)

return cls.\_instance

def get\_connection(self):

return self.connection

**Justification**: This code ensures only one SQLite connection is created, reused across the app. The \_\_new\_\_ method checks for an existing instance, creating it only once. This reduces resource usage and ensures consistent database access, critical for a student-focused app with moderate traffic.

**3. Factory Pattern**

**Explanation**

The Factory pattern is a creational design pattern that encapsulates object creation, allowing a factory class to produce objects based on input parameters. It promotes flexibility and loose coupling by delegating object instantiation to subclasses or methods.

**How and Where It Is Used**

The Factory pattern is used to create user profiles based on user roles (e.g., Student, Professor, Admin). A ProfileFactory class determines the appropriate profile type during user registration, ensuring the correct profile subclass is instantiated.

* **Location**: User management module (users/factories.py and users/models.py).
* **Usage**: When a user registers, the ProfileFactory creates a profile (e.g., StudentProfile, ProfessorProfile) based on the selected role, associating it with the user account.

**Benefits**

* **Flexibility**: Easily extend to support new profile types without modifying existing code.
* **Encapsulation**: Hides object creation logic, simplifying client code.
* **Maintainability**: Centralizes profile creation, making updates easier.

**Trade-offs**

* **Complexity**: Adds an extra layer of abstraction, increasing code complexity.
* **Overhead**: May be overkill for simple systems with few profile types.
* **Maintenance**: Requires updates to the factory when adding new profile types.

**Code Sample and Justification**

The following code in users/factories.py implements the Factory pattern for user profile creation.

from users.models import StudentProfile, ProfessorProfile, AdminProfile

class ProfileFactory:

@staticmethod

def create\_profile(user, role):

if role == "student":

return StudentProfile(user=user, major="Undecided")

elif role == "professor":

return ProfessorProfile(user=user, department="Unknown")

elif role == "admin":

return AdminProfile(user=user, permissions="full")

else:

raise ValueError("Invalid role")

**Justification**: This code centralizes profile creation, allowing the app to instantiate the correct profile type based on the user’s role. It decouples profile creation from the registration logic, making it easy to add new roles (e.g., AlumniProfile) in the future. This is particularly useful for a university app with diverse user types.

**Conclusion**

The Observer, Singleton, and Factory patterns enhance the university social media app’s functionality and maintainability. The Observer pattern enables real-time notifications, the Singleton pattern optimizes database resource usage, and the Factory pattern streamlines user profile creation. While each pattern introduces some complexity, their benefits in scalability, flexibility, and decoupling align with the app’s goals of supporting academic collaboration and community engagement.