Workshop 11 – Project Refactoring to OOP

Before start

Important: This notebook is a *guide*.
You dont need to adhear to it perfectlly

We will cover:

1. Creating the new project folder structure (`models` , `services` , etc.)
2. Creating **entity classes** (`User` , `SecurityIncident` , `Dataset` , `ITTicket`)
3. Creating **service classes** (`DatabaseManager` , `AuthManager` , `AIAssistant`)
4. Refactoring procedural code into OOP
5. Refactoring a login flow
6. Refactoring a domain page (e.g. Cybersecurity)
7. Final clean-up & documentation checklist

---

1. Project folder structure

From the slides, the final structure should look like this:

```
multi_domain_platform/
    models/
        __init__.py
        user.py
        security_incident.py
        dataset.py
        it_ticket.py
    services/
        __init__.py
        database_manager.py
        auth_manager.py
        ai_assistant.py
    database/
        db.py
        platform.db
    pages/
        1_🔐_Login.py
        2_🛡️_Cybersecurity.py
        3_📊_Data_Science.py
        4_💻_IT_Operations.py
        5_🤖_AI_Assistant.py
    .streamlit/
```

```
        secrets.toml
    Home.py
    requirements.txt
    README.md
    .gitignore
```

Task 1.1

In your **actual project folder** (not here in the notebook):

- Create the folders: `models`, `services`, `database`, `pages`, `.streamlit`
- Add empty `__init__.py` files to `models/` and `services/` so they are packages

---

2. Entity classes in `models/`

Your **domain entities** (from the slides) are:

- `User` (authentication)
- `SecurityIncident` (cybersecurity)
- `Dataset` (data science)
- `ITTicket` (IT operations)

For each one, we will:

- Create a class in its own file inside `models/`
- Use **private attributes**
- Add an `__init__` constructor
- Add a `__str__` method
- Add 1–2 simple methods from the slides

2.1 User entity (`models/user.py`)

Task 2.1

Create a new file `models/user.py` in your project and implement the template below.

In [ ]:
```python
class User:
    """Represents a user in the Multi-Domain Intelligence Platform."""

    def __init__(self, username: str, password_hash: str, role: str):
        self.__username = username
        self.__password_hash = password_hash
        self.__role = role

    def get_username(self) -> str:
        return self.__username
    def get_role(self) -> str:
        return self.__role

    def verify_password(self, plain_password: str, hasher) -> bool:
        """Check if a plain-text password matches this user's hash.

        `hasher` is any object with a `check_password(plain, hashed)` met
        (You will inject this from your AuthManager.)
        """
        return hasher.check_password(plain_password, self.__password_hash

    def __str__(self) -> str:
        return f"User({self.__username}, role={self.__role})"
```

2.2 SecurityIncident entity (`models/security_incident.py`)

Task 2.2

Create `models/security_incident.py` and implement:

In [ ]:
```python
class SecurityIncident:
    """Represents a cybersecurity incident in the platform."""

    def __init__(self, incident_id: int, incident_type: str, severity: st
        self.__id = incident_id
        self.__incident_type = incident_type
        self.__severity = severity
        self.__status = status
        self.__description = description

    def get_id(self) -> int:
        return self.__id

    def get_severity(self) -> str:
        return self.__severity

    def get_status(self) -> str:
        return self.__status

    def get_description(self) -> str:
        return self.__description

    def update_status(self, new_status: str) -> None:
        self.__status = new_status

    def get_severity_level(self) -> int:
        """Return an integer severity level (simple example)."""
        mapping = {
            "low": 1,
            "medium": 2,
            "high": 3,
            "critical": 4,
        }
        return mapping.get(self.__severity.lower(), 0)
    def __str__(self) -> str:
        return f"Incident {self.__id} [{self.__severity.upper()}] {self._
```

2.3 Dataset entity (`models/dataset.py`)

In [ ]:
```
class Dataset:
    """Represents a data science dataset in the platform."""

    def __init__(self, dataset_id: int, name: str, size_bytes: int, rows:
        self.__id = dataset_id
        self.__name = name
        self.__size_bytes = size_bytes
        self.__rows = rows
        self.__source = source

    def calculate_size_mb(self) -> float:
        return self.__size_bytes / (1024 * 1024)

    def get_source(self) -> str:
        return self.__source

    def __str__(self) -> str:
        size_mb = self.calculate_size_mb()
        return f"Dataset {self.__id}: {self.__name} ({size_mb:.2f} MB, {s
```

2.4 ITTicket entity (`models/it_ticket.py`)

In [ ]:
```
class ITTicket:
    """Represents an IT support ticket."""

    def __init__(self, ticket_id: int, title: str, priority: str, status:
        self.__id = ticket_id
        self.__title = title
        self.__priority = priority
        self.__status = status
        self.__assigned_to = assigned_to

    def assign_to(self, staff: str) -> None:
        self.__assigned_to = staff

    def close_ticket(self) -> None:
        self.__status = "Closed"

    def get_status(self) -> str:
        return self.__status

    def __str__(self) -> str:
        return (
            f"Ticket {self.__id}: {self.__title} "
            f"[{self.__priority}] - {self.__status} (assigned to: {self._
        )
```

---

3. Service classes in `services/`

Service classes **coordinate operations** and **talk to the database / APIs**.

From the slides, we need:

- `DatabaseManager` – handles all DB connections and queries
- `AuthManager` – handles user login, registration, password hashing
- `AIAssistant` – wraps your ChatGPT / OpenAI calls

3.1 DatabaseManager (`services/database_manager.py`)

Task 3.1

Create `services/database_manager.py` with this template and then connect it to your existing SQLite database file.

In [ ]:
```python
import sqlite3
from typing import Any, Iterable


class DatabaseManager:
    """Handles SQLite database connections and queries."""

    def __init__(self, db_path: str):
        self._db_path = db_path
        self._connection: sqlite3.Connection | None = None

    def connect(self) -> None:
        if self._connection is None:
            self._connection = sqlite3.connect(self._db_path)

    def close(self) -> None:
        if self._connection is not None:
            self._connection.close()
            self._connection = None

    def execute_query(self, sql: str, params: Iterable[Any] = ()):
        """Execute a write query (INSERT, UPDATE, DELETE)."""
        if self._connection is None:
            self.connect()
        cur = self._connection.cursor()
        cur.execute(sql, tuple(params))
        self._connection.commit()
        return cur

    def fetch_one(self, sql: str, params: Iterable[Any] = ()):
        if self._connection is None:
            self.connect()
        cur = self._connection.cursor()
        cur.execute(sql, tuple(params))
        return cur.fetchone()

    def fetch_all(self, sql: str, params: Iterable[Any] = ()):
        if self._connection is None:
            self.connect()
        cur = self._connection.cursor()
        cur.execute(sql, tuple(params))
        return cur.fetchall()
```

3.2 AuthManager (`services/auth_manager.py`)

Task 3.2

Create `services/auth_manager.py` . This class will:

- Use **composition** with `DatabaseManager` (`AuthManager` *has a* `DatabaseManager`)
- Provide methods like `register_user(...)` and `login_user(...)`

> In your actual project, plug in your existing hashing library (e.g. bcrypt).

In [ ]:

```
from typing import Optional
from models.user import User
from services.database_manager import DatabaseManager
import hashlib  # simple example; replace with bcrypt in real project


class SimpleHasher:
    """Very basic hasher using SHA256 (for demo only)."""


    def hash_password(plain: str) -> str:
        return hashlib.sha256(plain.encode("utf-8")).hexdigest()


    def check_password(plain: str, hashed: str) -> bool:
        return SimpleHasher.hash_password(plain) == hashed


class AuthManager:
    """Handles user registration and login."""

    def __init__(self, db: DatabaseManager):
        self._db = db

    def register_user(self, username: str, password: str, role: str = "us
        password_hash = SimpleHasher.hash_password(password)
        self._db.execute_query(
            "INSERT INTO users (username, password_hash, role) VALUES (?,
            (username, password_hash, role),
        )

    def login_user(self, username: str, password: str) -> Optional[User]:
        row = self._db.fetch_one(
            "SELECT username, password_hash, role FROM users WHERE userna
            (username,),
        )
        if row is None:
            return None

        username_db, password_hash_db, role_db = row
        if SimpleHasher.check_password(password, password_hash_db):
            return User(username_db, password_hash_db, role_db)
        return None
```

3.3 AIAssistant (`services/ai_assistant.py`)

This class will wrap your existing OpenAI / ChatGPT logic from previous weeks. Here we just provide a simple placeholder.

In [ ]:
```python
from typing import List, Dict


class AIAssistant:
    """Simple wrapper around an AI/chat model.

    In your real project, connect this to OpenAI or another provider.
    """

    def __init__(self, system_prompt: str = "You are a helpful assistant.
        self._system_prompt = system_prompt
        self._history: List[Dict[str, str]] = []

    def set_system_prompt(self, prompt: str) :
        self._system_prompt = prompt

    def send_message(self, user_message: str) :
        """Send a message and get a (fake) response.

        Replace this body with your real API call.
        """
        self._history.append({"role": "user", "content": user_message})
        # Fake response for now:
        response = f"[AI reply to]: {user_message[:50]}"
        self._history.append({"role": "assistant", "content": response})
        return response

    def clear_history(self):
        self._history.clear()
```

---

4. Refactoring procedural code into OOP

In earlier weeks, you may have written code like this in your Streamlit pages:

```python
# Procedural style (BEFORE)
import sqlite3

conn = sqlite3.connect("database/platform.db")
cur = conn.cursor()
cur.execute("SELECT id, incident_type, severity, status,
description FROM security_incidents")
rows = cur.fetchall()

for row in rows:
    st.write(row[0], row[1], row[2], row[3])
```

We want to refactor this to use:

- `DatabaseManager`
- `SecurityIncident` objects

4.1 Example refactor (incidents list)

Task 4.1

Use this pattern in your **Cybersecurity** Streamlit page.

In [ ]:
```
from services.database_manager import DatabaseManager
from models.security_incident import SecurityIncident

db = DatabaseManager("database/platform.db")
db.connect()

rows = db.fetch_all(
    "SELECT id, incident_type, severity, status, description FROM securit

incidents: list[SecurityIncident] = []
for row in rows:
    incident = SecurityIncident(
        incident_id=row[0],
        incident_type=row[1],
        severity=row[2],
        status=row[3],
        description=row[4],
    )
    incidents.append(incident)

# Now you would pass `incidents` to your Streamlit UI
# Example (pseudo-code, not real Streamlit here):
for incident in incidents:
    print(incident)  # in Streamlit, you might use st.write(incident)
```

---

5. Refactoring the login page

The login page should now:

1. Create a `DatabaseManager` and `AuthManager`
2. On button click, call `auth_manager.login_user(username, password)`
3. If a `User` is returned, store it (or at least username/role) in the session

In [ ]:
```
# PSEUDO-CODE: What your Streamlit login page might look like,
# using AuthManager and User.

from services.database_manager import DatabaseManager
from services.auth_manager import AuthManager

db = DatabaseManager("database/platform.db")
auth = AuthManager(db)
```

```python
# In a real Streamlit page you would do:
# username = st.text_input("Username")
# password = st.text_input("Password", type="password")

username = "alice"
password = "mypassword"

user = auth.login_user(username, password)
if user is None:
    print("Login failed")  # st.error(...) in Streamlit
else:
    print("Login successful for:", user)
    # st.session_state["current_user"] = user.get_username()
    # st.session_state["current_role"] = user.get_role()
```

---

6. Refactoring a domain page (example)

Pick **one** of your domain pages (e.g. `Cybersecurity`) and:

1. Replace direct SQL calls with `DatabaseManager`
2. Wrap row data into `SecurityIncident` objects
3. Use object methods/attributes in the UI (e.g. `incident.get_severity_level()`)
4. If you use AI, pass `incident.get_description()` into your AI prompt

---

7. Checklist

This checklist is provided as **guidance only** — your final implementation may differ based on your design decisions. Use these points to help you review your work before submission:

- The project folder structure follows the recommended architecture (`models`, `services`, `database`, `pages`)
- All required domain entities are implemented as Python classes (`User`, `SecurityIncident`, `Dataset`, `ITTicket`)
- Service classes are implemented to support the core functionality (e.g. `DatabaseManager`, `AuthManager`, `AIAssistant`)
- Streamlit pages have been updated to work with class objects (rather than raw tuples or dictionaries)
- `__str__` methods are used to improve readability and debugging output
- Code includes clear **docstrings** and **comments** to explain logic and design choices
- The project **README** provides an overview of the architecture and how to run the app
- The technical report includes a UML class diagram to illustrate the OOP relationships