Imports & Setup

from __future__ import annotations

- Enables **future annotations** in Python → allows type hints for classes/functions that are defined later in the file.

import hashlib

- Used for **hashing passwords** with algorithms like SHA-256.

import logging

- Built-in Python library for **logging information, warnings, and errors**.

import os

- Provides functions to **work with files, paths, and directories**.

import pymysql

- Python connector to work with **MySQL databases**.

from contextlib import contextmanager

- Used to create **context managers** (e.g., safely opening and closing database connections).

from dataclasses import dataclass

- Provides the @dataclass decorator to define simple **data models** without writing boilerplate code.

from datetime import datetime

- Used for **date and time** handling.

from enum import Enum

- Allows defining **enumerations** (e.g., User types like Artist/Customer, Payment methods).

from pathlib import Path

- Object-oriented way to work with **file system paths**.

from typing import Any, Dict, List, Optional, Union

- Provides **type hints** (for clarity and debugging).

- Any → any type.
- Dict → dictionary.
- List → list.
- Optional → value can be None.
- Union → multiple allowed types.

import re

- Python's **Regular Expressions** library. Used for validating email, date formats, etc.

import uuid

- Used to generate **unique IDs** (e.g., for transaction IDs).

Logging Setup

logging.basicConfig(level=logging.INFO)

- Configures logging system at **INFO level** (logs info, warnings, and errors).

logger = logging.getLogger(__name__)

- Creates a logger object named after the current module (this file).
- Used throughout the code to log activity.

---

2. Database Configuration

```
DB_CONFIG = {
   'host': 'localhost',
   'user': 'root',
   'password': 'root',  # Update this with your MySQL password
   'database': 'brush_and_soul',
   'charset': 'utf8mb4',
   'autocommit': True
}
```

- Configuration dictionary for **MySQL database**:
    - host: database location (localhost).
    - user: MySQL username (default is root).
    - password: MySQL password (here "root").
    - database: name of your database (brush_and_soul).
    - charset: character encoding (utf8mb4 supports emojis).
    - autocommit: saves changes automatically after each SQL query.

```
UPLOADS_DIR = "uploads"
```

- Defines the default **directory for file uploads** (images, videos, etc.).

Enums & Data Models

User Type Enum

```
class UserType(Enum):
    """User type enumeration"""
    ARTIST = "artist"
    CUSTOMER = "customer"
```

- class UserType(Enum): Defines an **enumeration** for user roles.
- ARTIST = "artist": Represents an artist user.
- CUSTOMER = "customer": Represents a customer user.
- 🔑 Benefit: safer than using plain strings, avoids typos.

Payment Method Enum

```
class PaymentMethod(Enum):
    """Payment methods enumeration - ENHANCED"""
    CREDIT_CARD = "Credit Card"
    DEBIT_CARD = "Debit Card"
    UPI = "UPI (PhonePe/GPay/Paytm)"
```

NET_BANKING = "Net Banking"

CASH_ON_DELIVERY = "Cash on Delivery"

DIGITAL_WALLET = "Digital Wallet"

- Defines available **payment options**.
- Each attribute corresponds to a payment type.
- Example usage: PaymentMethod.UPI ensures consistent values across app.

User Data Model

@dataclass

class User:

"""User data model - Bio and Website NOT included here"""

user_id: int

username: str

email: str

password_hash: str

user_type: UserType

created_at: Optional[datetime] = None

- @dataclass: Automatically creates __init__, __repr__, etc.
- user_id: Unique identifier for each user.
- username: The user's chosen name.
- email: The user's email address.
- password_hash: The hashed password (not plain text).
- user_type: Enum (artist or customer).
- created_at: Optional → when the account was created.

Payment Information Data Model

```
@dataclass
class PaymentInfo:
    """Payment information structure - NO JSON usage"""
    method: Union[str, PaymentMethod]
    amount: float
    transaction_id: str = ""
    status: str = "pending"
    timestamp: str = ""
    details: Dict[str, Any] = None
```

- Represents **payment details** for an order.
- method: Can be a string or a PaymentMethod.
- amount: Payment amount.
- transaction_id: Unique payment ID (generated).
- status: Default is "pending" (could be "success" or "failed").
- timestamp: When payment was made.
- details: Dictionary for extra info (like gateway response).

```
    def __post_init__(self):
        if self.details is None:
            self.details = {}

        if isinstance(self.method, PaymentMethod):
            self.method = self.method.value
```

- __post_init__: Runs automatically after dataclass init.
- Ensures details is always a dictionary.
- Converts PaymentMethod enum into a **string** for DB storage.

Shipping Information Data Model

```
@dataclass
class ShippingInfo:
    """Shipping address information"""
    full_name: str = ""
    address_line1: str = ""
    address_line2: str = ""
    city: str = ""
    state: str = ""
    pincode: str = ""
    phone: str = ""
    email: str = ""
```

- Represents a **shipping address**.
- Default values are empty strings.
- Stores recipient name, address, city, state, postal code, phone, and email.

4. Date Formatting Functions

Function: format_date_to_ddmmyyyy

```
def format_date_to_ddmmyyyy(date_input: Union[str, datetime, None]) -> str:
    """
    Convert various date formats to dd-mm-yyyy format
    Handles: YYYYMMDDHHMMSS, datetime objects, ISO dates, etc.
    """
```

- Defines a function that takes many possible date formats (str, datetime, or None) and returns them in **dd-mm-yyyy** format.
- Docstring explains the use: it handles raw strings, datetime objects, ISO format, etc.

```
    try:
```

```
if not date_input:
    return ""
```

- If no date is provided (e.g., None or empty string), return an empty string.

```
if isinstance(date_input, str) and re.match(r'^\d{2}-\d{2}-\d{4}$', date_input):
    return date_input
```

- If the input is already in the correct format (dd-mm-yyyy), just return it.

```
if isinstance(date_input, str) and len(date_input) >= 8:
    if len(date_input) >= 14:
        date_part = date_input[:8]
    else:
        date_part = date_input
```

- Handles raw string formats like YYYYMMDDHHMMSS.
- If timestamp length is ≥ 14, extract first 8 characters (YYYYMMDD).

```
    if len(date_part) == 8 and date_part.isdigit():
        dt = datetime.strptime(date_part, "%Y%m%d")
        return dt.strftime("%d-%m-%Y")
```

- If extracted string is exactly 8 digits (YYYYMMDD), convert it into dd-mm-yyyy.

```
if isinstance(date_input, datetime):
    return date_input.strftime("%d-%m-%Y")
```

- If the input is a datetime object, directly format it into dd-mm-yyyy.

```
if hasattr(date_input, 'strftime'):
    return date_input.strftime("%d-%m-%Y")
```

- Covers any other objects that support .strftime() (like date objects).

```
if isinstance(date_input, str):
    try:
        dt = datetime.fromisoformat(str(date_input).replace('Z', '+00:00'))
```

```
        return dt.strftime("%d-%m-%Y")
    except ValueError:
        pass
```

- If input is an **ISO date string** (e.g., 2025-08-06 or with Z timezone), parse and convert it.

```
common_formats = [
    "%Y-%m-%d",
    "%Y/%m/%d",
    "%d/%m/%Y",
    "%d-%m-%Y",
    "%Y%m%d",
    "%d.%m.%Y",
    "%Y-%m-%d %H:%M:%S",
    "%d/%m/%Y %H:%M:%S"
]
```

- List of common formats the function supports:
    - 2025-08-06
    - 2025/08/06
    - 06/08/2025
    - 06-08-2025
    - 20250806
    - 06.08.2025
    - 2025-08-06 22:00:00
    - 06/08/2025 22:00:00

```
for fmt in common_formats:
    try:
```

```
            dt = datetime.strptime(str(date_input), fmt)
            return dt.strftime("%d-%m-%Y")
        except ValueError:
            continue
```

- Loops through all formats.
- If one works, converts it into dd-mm-yyyy.
- If parsing fails, move to next format.

```
    return str(date_input)
```

- If no conversion works, just return the input as string.

```
  except Exception as e:
    logger.error(f"Error formatting date: {e}")
    return str(date_input) if date_input else ""
```

- If something goes wrong, log the error and return the original input.

Function: get_current_date_ddmmyyyy

```
def get_current_date_ddmmyyyy() -> str:
  """Get current date in dd-mm-yyyy format"""
  return datetime.now().strftime("%d-%m-%Y")
```

- Returns **today's date** in dd-mm-yyyy.

Function: get_current_datetime_ddmmyyyy

```
def get_current_datetime_ddmmyyyy() -> str:
  """Get current datetime in dd-mm-yyyy HH:MM format"""
  return datetime.now().strftime("%d-%m-%Y %H:%M")
```

- Returns **current date and time** in dd-mm-yyyy HH:MM.

Function: format_order_date

```
def format_order_date(order_date: Any) -> str:
    """Format order date specifically for display"""
    return format_date_to_ddmmyyyy(order_date)
```

- A wrapper around format_date_to_ddmmyyyy for orders.
- Keeps code **clean and consistent**.

Function: format_timestamp_to_ddmmyyyy

```
def format_timestamp_to_ddmmyyyy(timestamp: str) -> str:
    """Convert YYYYMMDDHHMMSS timestamp to dd-mm-yyyy format"""
```

- Converts timestamp strings into dd-mm-yyyy.

```
    try:
        if not timestamp or len(timestamp) < 8:
            return ""
```

- If invalid timestamp, return empty string.

```
        date_part = timestamp[:8]
        if len(date_part) == 8 and date_part.isdigit():
            year = date_part[:4]
            month = date_part[4:6]
            day = date_part[6:8]
            return f"{day}-{month}-{year}"
```

- Extracts first 8 chars → YYYYMMDD.
- Breaks it into year, month, day → returns dd-mm-yyyy.

```
        return timestamp
```

- If parsing fails, return as is.

```
    except Exception as e:
```

logger.error(f"Error formatting timestamp: {e}")

return str(timestamp) if timestamp else ""

- Logs any errors and returns safe value.

✅ That covers all the **Date Formatting functions**.
Now your app can handle almost any date input and display it consistently as **dd-mm-yyyy**.

5. Payment Details Serialization

Function: serialize_payment_details

def serialize_payment_details(details: Dict[str, Any]) -> str:

"""Convert payment details dict to a serialized string"""

- Defines a function that converts a Python **dictionary** of payment details into a **string**.
- Example:
- {"method": "UPI", "amount": 500}

→

method=UPI|amount=500

try:

if not details:

return ""

- If no details are provided (empty dict), return an empty string.

items = []

- Initialize a list to collect key=value pairs.

for key, value in details.items():

if value is None:

value = ""

items.append(f"{key}={value}")

- Loop through each key-value pair in dictionary.
- If a value is None, replace it with empty string (to avoid errors).
- Append each pair as key=value into the list.

return "|".join(items)

- Join all key=value pairs with the | separator.
- Example:
  ["method=UPI", "amount=500", "status=success"] →
  "method=UPI|amount=500|status=success"

except Exception as e:

logger.error(f"Error serializing payment details: {e}")

return ""

- If something goes wrong, log the error and return empty string.

Function: deserialize_payment_details

def deserialize_payment_details(details_str: str) -> Dict[str, Any]:

"""Convert serialized string back to dict"""

- Defines function to convert the **serialized string** back into a **dictionary**.

try:

if not details_str:

return {}

- If input string is empty, return an empty dictionary.

details = {}

- Create an empty dictionary to store results.

for item in details_str.split("|"):

if "=" in item:

key, value = item.split("=", 1)

details[key] = value

- Split string by | into ["method=UPI", "amount=500"].
- For each item, split again by =.
- Assign key=value to dictionary.
- Example: "method=UPI|amount=500" → {"method": "UPI", "amount": "500"}

return details

- Return the final dictionary.

except Exception as e:

logger.error(f"Error deserializing payment details: {e}")

return {}

- If any error occurs (e.g., malformed string), log it and return empty dict.

🔑 **Why This Approach?**

- Instead of storing JSON in DB, it stores details as a simple string.
- Advantage: works with plain SQL tables (no need for JSON columns).
- Example usage:
    - **Serialize before saving in DB**
    - serialize_payment_details({"method": "UPI", "status": "success"})
    - # Output: "method=UPI|status=success"
    - **Deserialize after fetching from DB**
    - deserialize_payment_details("method=UPI|status=success")
    - # Output: {"method": "UPI", "status": "success"}

✅ That covers **Payment Serialization & Deserialization**.
This ensures payment info is **stored safely** in DB and **retrievable in dictionary form**.

6. Database Manager

Class Definition

```
class DatabaseManager:
  """Handles all database operations for Brush and Soul - ENHANCED"""
```

- Defines the DatabaseManager class.
- Purpose: manage all MySQL interactions (connection, queries, table creation).

Initialization

```
  def __init__(self, config: Dict[str, Any]):
    self.config = config
    self.ensure_database_exists()
    self.create_tables()
```

- __init__: Constructor method.
- config: Takes DB configuration (like DB_CONFIG).
- ensure_database_exists(): Checks if database exists, creates if not.
- create_tables(): Creates all required tables if missing.

Database Connection (Context Manager)

```
  @contextmanager
  def get_connection(self):
    conn = None
    try:
      conn = pymysql.connect(**self.config)
      yield conn
    finally:
      if conn:
        conn.close()
```

- Defines get_connection as a **context manager** using @contextmanager.
- Opens a DB connection with pymysql.connect.
- Yields connection for query execution.
- Ensures it's **always closed** after use (prevents leaks).

Usage example:

```
with db_manager.get_connection() as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
```

Ensure Database Exists

```
    def ensure_database_exists(self):
        config_copy = self.config.copy()
        db_name = config_copy.pop('database')
```

- Make a copy of config.
- Extract database name (brush_and_soul).

```
        conn = pymysql.connect(**config_copy)
        cursor = conn.cursor()
        cursor.execute(f"CREATE DATABASE IF NOT EXISTS {db_name} CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci")
        conn.close()
```

- Connects to MySQL **without database name**.
- Runs CREATE DATABASE IF NOT EXISTS ... to ensure DB exists.
- Closes connection.

Create Tables

```
    def create_tables(self):
```

```
with self.get_connection() as conn:
    cursor = conn.cursor()
```

- Opens a DB connection.
- Gets cursor for SQL execution.

Users Table

```
cursor.execute("""
    CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        username VARCHAR(255),
        email VARCHAR(255),
        password_hash TEXT,
        user_type ENUM('artist','customer'),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
""")
```

- Creates users table with:
    - id: auto-increment primary key.
    - username, email, password_hash.
    - user_type: must be 'artist' or 'customer'.
    - created_at: timestamp of account creation.

Artworks Table

```
cursor.execute("""
    CREATE TABLE IF NOT EXISTS artworks (
        id INT AUTO_INCREMENT PRIMARY KEY,
```

```
        title VARCHAR(255),
        description TEXT,
        filename TEXT,
        price FLOAT,
        artist_username VARCHAR(255),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
    """)
```

- Stores artworks uploaded by artists.
- Includes title, description, filename (image file), price, artist_username.

Blogs Table

```
    cursor.execute("""
      CREATE TABLE IF NOT EXISTS blogs (
        id INT AUTO_INCREMENT PRIMARY KEY,
        title VARCHAR(255),
        content TEXT,
        author_username VARCHAR(255),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )
    """)
```

- Blog posts written by users.

Materials Table

```
    cursor.execute("""
      CREATE TABLE IF NOT EXISTS materials (
```

```
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    description TEXT,
    file_path TEXT,
    uploader_username VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  )
""")
```

- Stores learning materials uploaded by users.

Tutorials Table

```
cursor.execute("""
  CREATE TABLE IF NOT EXISTS tutorials (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    description TEXT,
    video_path TEXT,
    uploader_username VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  )
""")
```

- Tutorials uploaded by artists (e.g., videos).

Portfolios Table

```
cursor.execute("""
  CREATE TABLE IF NOT EXISTS portfolios (
```

```
    id INT AUTO_INCREMENT PRIMARY KEY,
    artist_username VARCHAR(255),
    bio TEXT,
    website TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  )
"""")
```

- Portfolio information for artists (bio + website).

Orders Table

```
cursor.execute("""
  CREATE TABLE IF NOT EXISTS orders (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255),
    subtotal FLOAT,
    tax FLOAT,
    shipping_cost FLOAT,
    total FLOAT,
    status VARCHAR(50),
    payment_method VARCHAR(50),
    payment_status VARCHAR(50),
    transaction_id VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    shipping_info TEXT,
    payment_details TEXT
  )
```

"""")

- Stores **orders** placed by customers.
- Includes cost breakdown, payment info, shipping info.

Order Items Table

```
cursor.execute("""
  CREATE TABLE IF NOT EXISTS order_items (
    id INT AUTO_INCREMENT PRIMARY KEY,
    order_id INT,
    artwork_id INT,
    quantity INT,
    price FLOAT,
    FOREIGN KEY (order_id) REFERENCES orders(id),
    FOREIGN KEY (artwork_id) REFERENCES artworks(id)
  )
""")
```

- Stores individual items for each order.
- Links order → artwork.

Cart Table

```
cursor.execute("""
  CREATE TABLE IF NOT EXISTS cart (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255),
    artwork_id INT,
    quantity INT,
```

```
        FOREIGN KEY (artwork_id) REFERENCES artworks(id)
    )
    """)
```

- Stores items temporarily before checkout.

Payment Transactions Table

```
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS payment_transactions (
            id INT AUTO_INCREMENT PRIMARY KEY,
            order_id INT,
            amount FLOAT,
            method VARCHAR(50),
            status VARCHAR(50),
            transaction_id VARCHAR(255),
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (order_id) REFERENCES orders(id)
        )
    """)
```

- Stores payment transaction history.

✅ That finishes the **DatabaseManager class and all table creation logic**.
This is the foundation of your backend — it ensures all tables exist when the app runs.

Password Hashing

```
def hash_password(password: str) -> str:
    """Hash password using SHA-256"""
    return hashlib.sha256(password.encode()).hexdigest()
```

- Defines function hash_password.
- Takes plain password as input (string).
- password.encode(): converts to bytes.
- hashlib.sha256(...).hexdigest(): generates a secure SHA-256 hash in hexadecimal.
- 🔑 Important: **Passwords are never stored as plain text** — only hashes.

Register User

def register_user(db: DatabaseManager, username: str, email: str, password: str, user_type: str) -> bool:

"""Register new user"""

- Defines register_user function.
- Inputs: database object, username, email, plain password, user type (artist or customer).
- Returns True if registration successful, else False.

try:

password_hash = hash_password(password)

- Hash the password before storing.

with db.get_connection() as conn:

cursor = conn.cursor()

cursor.execute(

"INSERT INTO users (username, email, password_hash, user_type) VALUES (%s, %s, %s, %s)",

(username, email, password_hash, user_type)

)

return True

- Open DB connection.
- Insert new user into users table.

- Store username, email, **hashed password**, and user_type.
- Return True on success.

except Exception as e:

logger.error(f"Error registering user: {e}")

return False

- If any error occurs (e.g., duplicate username), log it and return False.

Verify Login

def verify_login(db: DatabaseManager, username: str, password: str) -> Optional[User]:

"""Verify login credentials"""

- Function to check if given username and password are valid.
- Returns a User object if login successful, otherwise None.

try:

password_hash = hash_password(password)

- Hash the provided password (same as stored hash).

with db.get_connection() as conn:

cursor = conn.cursor()

cursor.execute("SELECT id, username, email, password_hash, user_type, created_at FROM users WHERE username=%s", (username,))

row = cursor.fetchone()

- Fetch user data from DB by username.
- Get first matching row.

if row and row[3] == password_hash:

return User(

user_id=row[0],

username=row[1],

```
            email=row[2],
            password_hash=row[3],
            user_type=UserType(row[4]),
            created_at=row[5]
        )
```

- If user exists **and password hash matches**:
    - Return a User dataclass object filled with DB values.

```
    return None
```

- If no match, return None.

```
except Exception as e:
    logger.error(f"Error verifying login: {e}")
    return None
```

- On any error, log and return None.

Update Password

```
def update_password(db: DatabaseManager, username: str, new_password: str) -> bool:
    """Update user's password"""
```

- Function to update user’s password.

```
    try:
        password_hash = hash_password(new_password)
```

- Hash new password before storing.

```
        with db.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(
                "UPDATE users SET password_hash=%s WHERE username=%s",
                (password_hash, username)
```

)

return True

- Update DB row for given username with new hash.
- Return True if successful.

except Exception as e:

logger.error(f"Error updating password: {e}")

return False

- On error (e.g., user not found), log and return False.

🔑 **Key Takeaways**

- **Passwords are always hashed** → safe storage.
- **Register** → insert user with hashed password.
- **Login** → compare hash of input password with stored hash.
- **Update password** → replace old hash with new hash.

✅ That finishes the **Authentication Section**.
Now your app can handle **sign up, login, and password reset** securely.

8. CRUD Operations

Artworks

Save Artwork

def save_artwork(db: DatabaseManager, title: str, description: str, filename: str, price: float, artist_username: str) -> bool:

"""Save artwork details"""

- Function to save an artwork to the database.
- Parameters: title, description, filename (image path), price, artist_username.

```
    try:
        with db.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(
                "INSERT INTO artworks (title, description, filename, price, artist_username) VALUES (%s, %s, %s, %s, %s)",
                (title, description, filename, price, artist_username)
            )
        return True
```

- Inserts artwork record into artworks table.
- Returns True if successful.

```
    except Exception as e:
        logger.error(f"Error saving artwork: {e}")
        return False
```

- Logs error and returns False on failure.

Load Artworks

```
def load_artworks(db: DatabaseManager) -> List[Dict[str, Any]]:
    """Load all artworks"""
```

- Returns a list of all artworks.

```
    try:
        with db.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute("SELECT id, title, description, filename, price, artist_username, created_at FROM artworks ORDER BY created_at DESC")
            rows = cursor.fetchall()
```

- Selects all artworks, ordered by newest first.

```
    return [
      {
        "id": row[0],
        "title": row[1],
        "description": row[2],
        "filename": row[3],
        "price": row[4],
        "artist_username": row[5],
        "created_at": format_date_to_ddmmyyyy(row[6])
      }
      for row in rows
    ]
```

- Converts each row into a dictionary.
- Formats created_at date into dd-mm-yyyy.

```
  except Exception as e:
    logger.error(f"Error loading artworks: {e}")
    return []
```

- Returns empty list if error occurs.

Delete Artwork

```
def delete_artwork(db: DatabaseManager, artwork_id: int) -> bool:
  """Delete artwork by ID"""
```

- Deletes artwork by ID.

```
  try:
    with db.get_connection() as conn:
      cursor = conn.cursor()
```

cursor.execute("DELETE FROM artworks WHERE id=%s", (artwork_id,))

return True

- Executes DELETE query.
- Returns True on success.

except Exception as e:

logger.error(f"Error deleting artwork: {e}")

return False

- Logs error and returns False.

Blogs

def save_blog(db: DatabaseManager, title: str, content: str, author_username: str) -> bool:

"""Save blog"""

- Inserts blog post into DB.

def load_blogs(db: DatabaseManager) -> List[Dict[str, Any]]:

"""Load blogs"""

- Retrieves all blogs in descending order of creation.

def delete_blog(db: DatabaseManager, blog_id: int) -> bool:

"""Delete blog by ID"""

- Deletes a blog entry.

Materials

def save_material(db: DatabaseManager, title: str, description: str, file_path: str, uploader_username: str) -> bool:

"""Save learning material"""

- Inserts learning material record.

def load_materials(db: DatabaseManager) -> List[Dict[str, Any]]:

"""Load materials"""

- Fetches all materials.

def delete_material(db: DatabaseManager, material_id: int) -> bool:

"""Delete material"""

- Deletes material entry.

Tutorials

def save_tutorial(db: DatabaseManager, title: str, description: str, video_path: str, uploader_username: str) -> bool:

"""Save tutorial"""

- Inserts tutorial record.

def load_tutorials(db: DatabaseManager) -> List[Dict[str, Any]]:

"""Load tutorials"""

- Fetches all tutorials.

def delete_tutorial(db: DatabaseManager, tutorial_id: int) -> bool:

"""Delete tutorial"""

- Deletes tutorial.

Portfolios

def save_portfolio(db: DatabaseManager, artist_username: str, bio: str, website: str) -> bool:

"""Save artist portfolio"""

- Inserts portfolio entry.

def load_portfolio(db: DatabaseManager, artist_username: str) -> Optional[Dict[str, Any]]:

"""Load portfolio for a specific artist"""

- Fetches portfolio for a specific artist.

def update_portfolio(db: DatabaseManager, artist_username: str, bio: str, website: str) -> bool:

"""Update portfolio"""

- Updates portfolio info.

def delete_portfolio(db: DatabaseManager, portfolio_id: int) -> bool:

"""Delete portfolio"""

- Deletes portfolio entry.

🔑 **Key Takeaways**

- Every content type (Artworks, Blogs, Materials, Tutorials, Portfolios) follows the same CRUD structure:
  - **Save (Create)**
  - **Load (Read)**
  - **Update (only for Portfolio)**
  - **Delete**
- Data is always returned as dictionaries → makes integration with Streamlit easier.

✅ That covers **CRUD Operations**.
Next major part is **Cart and Order Management** (shopping cart, placing orders, tracking payments).

9. Cart Functions

Add to Cart

def add_to_cart(db: DatabaseManager, username: str, artwork_id: int, quantity: int) -> bool:

"""Add item to cart"""

- Function to add an artwork to a user's cart.

try:

```
    with db.get_connection() as conn:
        cursor = conn.cursor()
        cursor.execute("INSERT INTO cart (username, artwork_id, quantity) VALUES (%s, %s, %s)",
                       (username, artwork_id, quantity))
    return True
```

- Inserts entry into cart table.
- Returns True if successful.

```
  except Exception as e:
    logger.error(f"Error adding to cart: {e}")
    return False
```

- Logs and returns False on error.

Load Cart

```
def load_cart(db: DatabaseManager, username: str) -> List[Dict[str, Any]]:
  """Load cart items for a user"""
```

- Fetches all items in a user's cart.

```
  try:
    with db.get_connection() as conn:
      cursor = conn.cursor()
      cursor.execute("""
        SELECT c.id, c.artwork_id, c.quantity, a.title, a.price, a.artist_username
        FROM cart c
        JOIN artworks a ON c.artwork_id = a.id
        WHERE c.username=%s
      """, (username,))
```

```
        rows = cursor.fetchall()
```

- Joins cart with artworks to show full artwork info.
- Returns results as list of rows.

```
        return [
            {"id": row[0], "artwork_id": row[1], "quantity": row[2],
             "title": row[3], "price": row[4], "artist_username": row[5]}
            for row in rows
        ]
```

- Converts rows into list of dictionaries.

```
    except Exception as e:
        logger.error(f"Error loading cart: {e}")
        return []
```

- On error → empty list.

Remove from Cart

```
def remove_from_cart(db: DatabaseManager, cart_id: int) -> bool:
    """Remove item from cart"""
```

- Deletes an item from cart by ID.

```
    try:
        with db.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute("DELETE FROM cart WHERE id=%s", (cart_id,))
        return True
```

- Executes delete query.
- Returns True on success.

```
    except Exception as e:
```

logger.error(f"Error removing from cart: {e}")

return False

- Logs error and returns False.

---

10. Orders & Checkout

Place Order

def place_order(db: DatabaseManager, username: str, subtotal: float, tax: float, shipping_cost: float,

total: float, payment_method: str, shipping_info: ShippingInfo,

payment_info: PaymentInfo) -> Optional[int]:

"""Place an order and return order ID"""

- Function to create a new order.
- Takes:
    - subtotal, tax, shipping_cost, total
    - payment_method (UPI, Credit Card, etc.)
    - shipping_info (address)
    - payment_info (payment details like transaction ID)

try:

with db.get_connection() as conn:

cursor = conn.cursor()

- Start DB transaction.

cursor.execute(

"INSERT INTO orders (username, subtotal, tax, shipping_cost, total, status, payment_method, payment_status, transaction_id, shipping_info, payment_details) VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)",

(username, subtotal, tax, shipping_cost, total, "pending", payment_method, payment_info.status,

payment_info.transaction_id, str(shipping_info.__dict__),

serialize_payment_details(payment_info.details))

)

- Insert new row into orders table.
- Save all details including:
    - status starts as "pending".
    - Shipping info stored as string.
    - Payment details serialized.

order_id = cursor.lastrowid

- Get the auto-generated order_id.

cursor.execute("SELECT artwork_id, quantity FROM cart WHERE username=%s", (username,))

cart_items = cursor.fetchall()

- Fetch all items currently in user's cart.

for item in cart_items:

cursor.execute(

"INSERT INTO order_items (order_id, artwork_id, quantity, price) SELECT %s, a.id, %s, a.price FROM artworks a WHERE a.id=%s",

(order_id, item[1], item[0])

)

- For each cart item:
    - Insert into order_items table.
    - Copies price from artworks table.

cursor.execute("DELETE FROM cart WHERE username=%s", (username,))

- Empty the user's cart after placing order.

return order_id

- Return the new order ID.

except Exception as e:

logger.error(f"Error placing order: {e}")

return None

- If error, log and return None.

Load Orders

def load_orders(db: DatabaseManager, username: str) -> List[Dict[str, Any]]:

"""Load orders for a user"""

- Loads all orders of a specific user.

try:

with db.get_connection() as conn:

cursor = conn.cursor()

cursor.execute("SELECT id, subtotal, tax, shipping_cost, total, status, payment_method, payment_status, transaction_id, created_at, shipping_info, payment_details FROM orders WHERE username=%s ORDER BY created_at DESC", (username,))

rows = cursor.fetchall()

- Fetch all orders, latest first.

return [

{"id": row[0], "subtotal": row[1], "tax": row[2], "shipping_cost": row[3], "total": row[4],

"status": row[5], "payment_method": row[6], "payment_status": row[7], "transaction_id": row[8],

"created_at": format_date_to_ddmmyyyy(row[9]), "shipping_info": row[10],

"payment_details": deserialize_payment_details(row[11])}

for row in rows

]

- Converts rows into list of dicts.
- Formats dates.
- Deserializes payment details back into dictionary.

except Exception as e:

logger.error(f"Error loading orders: {e}")

return []

- Returns empty list if error.

Update Order Status

def update_order_status(db: DatabaseManager, order_id: int, new_status: str, payment_status: Optional[str] = None) -> bool:

"""Update order status"""

- Updates status of an order (e.g., from pending → shipped).

try:

with db.get_connection() as conn:

cursor = conn.cursor()

if payment_status:

cursor.execute("UPDATE orders SET status=%s, payment_status=%s WHERE id=%s",

(new_status, payment_status, order_id))

else:

cursor.execute("UPDATE orders SET status=%s WHERE id=%s", (new_status, order_id))

return True

- If payment_status is provided, update both.
- Else update only status.

except Exception as e:

logger.error(f"Error updating order status: {e}")

return False

- Logs error and returns False.

✅ That covers **Cart and Orders** functionality.
So now your app supports:

- Adding/removing items from cart
- Checkout and placing orders
- Storing payment & shipping info
- Tracking order status

1. Payment Transactions

Record Payment Transaction

def record_payment_transaction(db: DatabaseManager, order_id: int, payment_info: PaymentInfo) -> bool:

"""Record payment transaction in database"""

- Defines a function to **log a payment transaction** in the DB.
- Parameters:
  - order_id: the order the payment belongs to.
  - payment_info: contains details like amount, method, transaction ID, and status.

---

try:

with db.get_connection() as conn:

cursor = conn.cursor()

- Opens DB connection and prepares a cursor.

```
cursor.execute(
    "INSERT INTO payment_transactions (order_id, amount, method, status, transaction_id) VALUES (%s,%s,%s,%s,%s)",
    (order_id, payment_info.amount, payment_info.method,
     payment_info.status, payment_info.transaction_id)
)
```

- Inserts a new row into the payment_transactions table.
- Stores:
  - order_id (links to orders)
  - amount (payment amount)
  - method (UPI, card, wallet, etc.)
  - status (pending, success, failed)
  - transaction_id (unique identifier from payment system)

```
return True
```

- Returns True if the insert succeeds.

```
except Exception as e:
    logger.error(f"Error recording payment transaction: {e}")
    return False
```

- If something goes wrong (DB error, bad data), logs the error and returns False.

🔑 Key Points

- **Why separate payments table?**
  - The orders table stores overall order details.

    - The payment_transactions table stores **individual payment attempts** (success or fail).
    - This separation makes it easier to **audit payments**, handle retries, and check histories.

- **Example usage:**
- payment_info = PaymentInfo(
- method="UPI",
- amount=500,
- transaction_id="TXN12345",
- status="success"
-)
- record_payment_transaction(db, order_id=10, payment_info=payment_info)

→ Creates a row in payment_transactions table.

APP.PY CODES

Imports

import streamlit as st

- Imports the Streamlit library and aliases it as st so you can call UI APIs like st.button, st.image, etc.

from typing import List, Dict, Any, Optional

- Brings in typing hints:
    - List, Dict for container types.
    - Any for “any type”.
    - Optional for a value that can be None.

from dataclasses import dataclass

- Imports the @dataclass decorator to auto-generate init/repr/equality methods for simple data containers.

from abc import ABC, abstractmethod

- ABC for abstract base classes.
- @abstractmethod to mark methods that child classes **must** implement.

App configuration (immutable settings)

@dataclass(frozen=True)

class AppConfig:

"""Application configuration with immutable settings"""

PAGE_TITLE: str = "Brush and Soul"

PAGE_ICON: str = " 🎨 "

LAYOUT: str = "wide"

PRIMARY_COLOR: str = "#8B4513"

SECONDARY_COLOR: str = "#A0522D"

ACCENT_COLOR: str = "#5C4033"

LIGHT_COLOR: str = "#F8F4E8"

DARK_COLOR: str = "#343434"

Explanation

- @dataclass(frozen=True) → after creation, fields can't be changed (safe constants).
- Holds page settings and your color palette:
    - PAGE_TITLE, PAGE_ICON, LAYOUT → passed to st.set_page_config.
    - Color hex codes are later injected into CSS as variables.

Data models

Artwork

```
@dataclass
class Artwork:
    """Artwork model with comprehensive details"""
    img_path: str
    title: str
    artist: str
    description: str
    materials: str
    state: str
    style: str
    price: str
    key: str
```

Explanation

- Defines one artwork's data: image path, meta (title/artist/style/state), pricing, a unique key used for toggling detail visibility.

```
    def to_display_dict(self) -> Dict[str, str]:
        """Convert artwork to display dictionary"""
        return {
            "Artist": self.artist,
            "Title": self.title,
            "Description": self.description,
            "Materials": self.materials,
            "State": self.state,
            "Style": self.style,
            "Price": self.price
        }
```

- Helper method to format fields for neat display (e.g., Markdown list).

NavigationItem

@dataclass

class NavigationItem:

"""Navigation item model"""

label: str

page_path: str

- One nav button: text label + page path to switch to.

FeatureSection

@dataclass

class FeatureSection:

"""Feature section model"""

title: str

features: List[str]

expanded: bool = True

- Represents an expander box with a title, bullet features, and default open/closed state.

SearchResult

@dataclass

class SearchResult:

"""Search result model for unified display"""

category: str

title: str

```
    subtitle: str
    description: str
    link: str = ""
    image_path: str = ""
```

Explanation

- Unified shape for all search results (artworks/materials/blogs/tutorials/portfolios).
- category → emoji + type (e.g., “ 🎨 Artwork”).
- title / subtitle → main+secondary text.
- description → short preview string.
- link → page to open via st.switch_page.
- image_path → optional thumbnail.

Abstract base classes

```
class BaseUIComponent(ABC):
    """Abstract base class for UI components"""

    @abstractmethod
    def render(self) -> None:
        """Render the component"""
        pass
```

- Contract: all UI components must implement render().

```
class BaseThemeProvider(ABC):
    """Abstract base class for theme providers"""

    @abstractmethod
    def get_css(self) -> str:
```

"""Get CSS styling"""

pass

- Contract: a theme provider must return CSS as a string.

Universal search (core search bar + results)

class UniversalSearchComponent(BaseUIComponent):

"""Universal search component with corrected view button UI"""

- Component that:
    - Shows the search bar.
    - Queries different data categories.
    - Renders grouped results with “View” buttons.

def __init__(self, artworks: List[Artwork]):

self.artworks = artworks

self._initialize_session_state()

self._init_database_connection()

- Saves provided artworks (not strictly used for DB search; useful as a fallback or to match UI).
- Initializes st.session_state keys for search.
- Tries to wire DB helpers from utils.py.

def _init_database_connection(self):

"""Initialize database connection"""

try:

from utils import (

get_all_artworks, get_all_materials, get_all_blogs,

get_all_tutorials, get_artists_with_content

)

```
        self.get_all_artworks = get_all_artworks
        self.get_all_materials = get_all_materials
        self.get_all_blogs = get_all_blogs
        self.get_all_tutorials = get_all_tutorials
        self.get_artists_with_content = get_artists_with_content
        self.db_available = True
```

- Tries to import DB accessor functions from utils.
- If success: attaches them to self and marks db_available=True.

```
    except ImportError as e:
        st.warning(f"Database connection not available: {e}")
        self.db_available = False
        # Set dummy functions to prevent errors
        self.get_all_artworks = lambda: []
        self.get_all_materials = lambda: []
        self.get_all_blogs = lambda: []
        self.get_all_tutorials = lambda: []
        self.get_artists_with_content = lambda: []
```

- If utils not present, warn in UI and stub all getters to empty lists to avoid crashes.

```
def _initialize_session_state(self) -> None:
    """Initialize session state for search"""
    if 'search_query' not in st.session_state:
        st.session_state.search_query = ""
    if 'search_results' not in st.session_state:
        st.session_state.search_results = []
```

- Ensures the session has default keys for query text and results.

Search helpers (per category)

Artworks

```
def _search_artworks(self, query: str) -> List[SearchResult]:
    """Search artworks from database"""
    results = []
    if not self.db_available:
        return results
```

- Returns empty if DB isn't available.

```
    try:
        artworks = self.get_all_artworks()
        query_lower = query.lower()
```

- Pulls all artworks from DB, lowercases the query for case-insensitive matching.

```
        for artwork in artworks:
            title = str(artwork.get('title', '')).lower()
            description = str(artwork.get('description', '')).lower()
            materials = str(artwork.get('materials', '')).lower()
            style = str(artwork.get('style', '')).lower()
            state = str(artwork.get('state', '')).lower()
```

- Extracts searchable fields from each artwork (with defaults), all lowercased.

```
            artist_name = (artwork.get('artist') or
                           artwork.get('username') or
                           artwork.get('creator') or
                           artwork.get('author') or
                           'Unknown Artist')
            artist = str(artist_name).lower()
```

- Flexible fallback for artist/creator fields.

if (query_lower in title or

query_lower in artist or

query_lower in description or

query_lower in materials or

query_lower in style or

query_lower in state):

- If the query appears in any field → it's a match.

results.append(SearchResult(

category="🎨 Artwork",

title=artwork.get('title', 'Unknown'),

subtitle=f"by {artist_name}",

description=f"{str(artwork.get('description', ''))[:100]}... | {artwork.get('style', '')} from {artwork.get('state', '')} | ₹{artwork.get('price', '0')}",

image_path=artwork.get('image_path', artwork.get('image', '')),

link="pages/05_Artworks.py"

))

- Builds a SearchResult with category, title, subtitle, a compact info description, thumbnail path, and the page to open.

except Exception as e:

st.error(f"Error searching artworks: {e}")

return results

- Shows any exceptions in UI and returns matches.

Materials / Blogs / Portfolios / Tutorials

- Each _search_* function follows the **same pattern**:
    - Early return if db_available is False.

- Get list via corresponding getter.
- Lowercase query and fields.
- Match on a few fields.
- Append a tailored SearchResult with:
  - category (🛒 , 📝 , 👤 , 🎓),
  - subtitle (e.g., seller/author/creator),
  - description (short preview + meta),
  - link to its page.

(The internal field names differ slightly per type, with safe fallbacks like “Unknown Seller”.)

Execute full search

```
def _perform_search(self, query: str) -> List[SearchResult]:
    """Perform comprehensive search across all categories"""
    if not query or len(query.strip()) < 2:
        return []
```

- Ignore empty/very short queries.

```
    if not self.db_available:
        st.warning("Database connection not available. Please check your utils.py configuration.")
        return []
```

- Warns and aborts when DB is missing.

```
    all_results = []

    try:
        all_results.extend(self._search_artworks(query))
        all_results.extend(self._search_materials(query))
```

```
            all_results.extend(self._search_blogs(query))
            all_results.extend(self._search_portfolios(query))
            all_results.extend(self._search_tutorials(query))
        except Exception as e:
            st.error(f"Error performing search: {e}")

        return all_results
```

- Aggregates results across all categories.
- Any unexpected error is surfaced in UI.

Render search results

```
    def _render_search_results(self, results: List[SearchResult]) -> None:
        """Render search results with corrected view button styling"""
        if not results:
            st.info("🔍 No results found. Try searching for artist names, artwork titles, materials, or techniques.")
            return
```

- Shows an info message when there’s nothing to show.

```
        st.markdown(f"### 🔍 Search Results ({len(results)} found)")
```

- Summary header.

```
        # Group results by category
        categories = {}
        for result in results:
            if result.category not in categories:
                categories[result.category] = []
            categories[result.category].append(result)
```

- Groups results (Artworks/Blogs/etc.) to render them under section headers.

```
# Display results by category with corrected view buttons
for category, items in categories.items():
    st.markdown(f"#### {category} ({len(items)} items)")
```

- Prints a subheader per category.

```
    for i, item in enumerate(items):
        with st.container():
            st.markdown(
                f"""
                <div class="search-result-card">
                    <h4 style="color: var(--accent); margin-bottom: 5px;">{item.title}</h4>
                    <p style="color: var(--primary); font-weight: 600; margin-bottom: 8px;">{item.subtitle}</p>
                    <p style="color: var(--dark); margin-bottom: 10px;">{item.description}</p>
                </div>
                """,
                unsafe_allow_html=True
            )
```

- For each result:
    - Wraps in a container (keeps layout tidy).
    - Injects custom HTML block styled by the CSS (title/subtitle/desc).

```
            if item.image_path and item.image_path.strip():
                try:
                    col1, col2 = st.columns([1, 2])
                    with col1:
                        st.image(item.image_path, width=150)
```

with col2:

View button with CSS wrapper for styling

st.markdown(' ', unsafe_allow_html=True)

- If there’s an image:
    - Split row into two columns.
    - Left: image thumbnail.
    - Right: “View” button container (CSS class changes style).

button_labels = {

" 🎨 Artwork": "View Artwork",

" 🛒 Materials": "View Materials",

" 📝 Blog": "View Blog",

" 👤 Portfolio": "View Portfolio",

" 🎓 Tutorial": "View Tutorial"

}

button_label = button_labels.get(item.category, "View Details")

unique_key = f"view_btn_{item.category.replace(' ', '_')}_{i}_{hash(item.title + item.subtitle)}"

- Human-friendly label per category.
- Unique Streamlit key to avoid widget clashes.

if st.button(

f" 🔍 {button_label}",

key=unique_key,

help=f"Navigate to {button_label.lower()}"

):

with st.spinner(f"Opening {button_label.lower()}..."):

st.switch_page(item.link)

st.markdown(' ', unsafe_allow_html=True)

- Renders the button. On click:
  - Shows spinner.
  - Navigates to the result's page.

except:

Fallback without image

...

- If any error happens with the two-column layout, fall back to a single "View" button.

else:

No image case

...

- If there's no image, render only the "View" button.

st.markdown("---")

- Divider between results.

Render the search bar

def render(self) -> None:

"""Render the search component"""

st.markdown("### 🔍 Universal Search")

st.markdown("*Search for artists, artworks, materials, blogs, portfolios, and tutorials*")

- Component title and short hint.

Search input with clean styling

col1, col2 = st.columns([4, 1])

```
with col1:
    search_query = st.text_input(
        "Search artworks, materials, blogs, portfolios, and tutorials...",
        placeholder="Try: artist name, artwork title, material type...",
        label_visibility="collapsed"
    )

with col2:
    search_button = st.button("🔍 Search", use_container_width=True)
```

- Two columns: left for input, right for button.
- Label is collapsed to keep the UI minimal (placeholder carries the hint).

```
# Show database status
if not self.db_available:
    st.error("⚠️ Database connection not available. Search functionality is limited.")
```

- Warns if DB is missing.

```
# Perform search on input change or button click
if search_query or search_button:
    if search_query != st.session_state.get('last_search_query', ''):
        st.session_state.last_search_query = search_query
        st.session_state.search_results = self._perform_search(search_query)

    # Display results
    self._render_search_results(st.session_state.search_results)
```

- Triggers search on text change or button click.
- Caches the last query in session state to avoid re-query loops.

- Renders results from session state.

Theme provider (CSS)

```
class ModernThemeProvider(BaseThemeProvider):
  """Modern theme provider with corrected view button styling matching image 2"""


  def __init__(self, config: AppConfig):
    self.config = config
```

- Stores AppConfig to interpolate colors into CSS.

```
  def get_css(self) -> str:
    """Generate CSS styling with corrected view button colors matching image 2"""
    return f"""
    <style>
    :root {{
      --primary: {self.config.PRIMARY_COLOR};
      --secondary: {self.config.SECONDARY_COLOR};
      --accent: {self.config.ACCENT_COLOR};
      --light: {self.config.LIGHT_COLOR};
      --dark: {self.config.DARK_COLOR};
      --glass-bg: rgba(255, 255, 255, 0.95);
      --shadow: 0 6px 12px rgba(139, 69, 19, 0.15);
      --hover-shadow: 0 8px 16px rgba(139, 69, 19, 0.25);
      --gradient-primary: linear-gradient(145deg, {self.config.PRIMARY_COLOR}, {self.config.SECONDARY_COLOR});
      --gradient-accent: linear-gradient(135deg, {self.config.ACCENT_COLOR}, {self.config.PRIMARY_COLOR});
```

```
    /* View button colors matching image 2 */
    --view-button-bg: #007BFF;
    --view-button-hover: #0056b3;
    --view-button-active: #004085;
}}
...
</style>
"""
```

Explanation (high-level so it's readable)

- Defines CSS variables from your theme: colors, gradients, shadows.
- Styles:
    - .stApp background and base font.
    - .view-button-container .stButton > button → dedicated styling for "View" buttons (normal/hover/active).
    - .stColumns tweaks to keep nav buttons in a single line and responsive spacing.
    - Navigation buttons, generic search button styles.
    - Fancy card styles (.card-3d, .search-result-card).
    - Input aesthetics (rounded, focus ring).
    - Headings/typography, divider, images (rounded + hover scale).
    - Footer container look.
    - Spinner color.
    - Media queries for small screens to shrink fonts/paddings.

(If you want, I can annotate each CSS selector one-by-one too.)

Navigation bar

class NavigationComponent(BaseUIComponent):

"""Navigation bar component with single-line layout"""

def __init__(self, navigation_items: List[NavigationItem]):

self.navigation_items = navigation_items

- Holds a list of nav entries to render.

def render(self) -> None:

"""Render navigation bar with single-line layout"""

Create exactly 8 columns for 8 buttons

nav_cols = st.columns(8)

- Creates 8 equal columns (one per button).

Short button labels

ultra_short_labels = {

"Artwork": "Art",

"Blog": "Blog",

"Material": "Materials",

"Tutorial": "Learn",

"Portfolio": "Portfolio",

"Order": "Cart",

"Register": "Register",

"Login": "Login"

}

- Map long labels to shorter UI-friendly text.

for i, (col, nav_item) in enumerate(zip(nav_cols, self.navigation_items)):

with col:

button_key = f"nav_{nav_item.label}_{i}"

button_text = ultra_short_labels.get(nav_item.label, nav_item.label[:4])

```
        if st.button(button_text, key=button_key, use_container_width=True):
            st.switch_page(nav_item.page_path)
```

- For each column:
    - Render a button with a unique key.
    - On click → navigate to the target page.

Artwork gallery

```
class ArtworkGalleryComponent(BaseUIComponent):
    """Artwork gallery component"""

    def __init__(self, artworks: List[Artwork]):
        self.artworks = artworks
        self._initialize_session_state()
```

- Keeps the provided artworks and sets initial “detail open/closed” flags.

```
    def _initialize_session_state(self) -> None:
        """Initialize session state for artwork details"""
        if 'show_artwork_detail' not in st.session_state:
            st.session_state['show_artwork_detail'] = {
                artwork.key: False for artwork in self.artworks
            }
```

- For each artwork key, store a boolean “show details” flag in session state.

```
    def _toggle_artwork_detail(self, artwork_key: str) -> None:
        """Toggle artwork detail visibility"""
        current_state = st.session_state['show_artwork_detail'].get(artwork_key, False)
        st.session_state['show_artwork_detail'][artwork_key] = not current_state
```

- Flips the visibility on button click.

```
def _render_artwork_card(self, artwork: Artwork, column) -> None:
    """Render individual artwork card"""
    with column:
        try:
            st.image(artwork.img_path, use_container_width=True)
        except:
            st.write("🖼️ Image not found")
```

- Tries to show the artwork image (handles missing path).

```
        if st.button("View Details", key=f"view_{artwork.key}"):
            self._toggle_artwork_detail(artwork.key)
```

- “View Details” toggles the detail section.

```
        if st.session_state['show_artwork_detail'][artwork.key]:
            self._render_artwork_details(artwork)
```

- Conditionally render the detail block.

```
def _render_artwork_details(self, artwork: Artwork) -> None:
    """Render artwork details"""
    details = artwork.to_display_dict()
    detail_text = "  \n".join([f"**{key}:** {value}" for key, value in details.items()])
    st.markdown(detail_text, unsafe_allow_html=False)
```

- Formats details as Markdown bullets (line breaks with \n).

```
def render(self) -> None:
    """Render artwork gallery"""
    if self.artworks:
        cols = st.columns(min(len(self.artworks), 3))
        for i, artwork in enumerate(self.artworks):
```

```
            col_index = i % len(cols)
            self._render_artwork_card(artwork, cols[col_index])
```

- Makes up to 3 columns for the gallery.
- Distributes artworks across columns (modulus indexing).

Feature sections (expanders)

```
class FeatureSectionComponent(BaseUIComponent):
  """Feature section component"""

  def __init__(self, sections: List[FeatureSection]):
    self.sections = sections
```

- Keeps the sections to render.

```
  def render(self) -> None:
    """Render feature sections"""
    feat_cols = st.columns(len(self.sections))

    for col, section in zip(feat_cols, self.sections):
      with col:
        with st.expander(f"**{section.title}**", expanded=section.expanded):
          for feature in section.features:
            st.write(f"- {feature}")
```

- Creates one column per section.
- Inside each column, shows an st.expander and lists bullet points.

Factories (to create ready-made data)

```
class ArtworkFactory:
```

```
    """Factory for creating artwork instances"""

    @staticmethod
    def create_featured_artworks() -> List[Artwork]:
        """Create featured artworks collection"""
        artworks_data = [
            { ... }, { ... }, { ... }
        ]

        return [Artwork(**artwork_data) for artwork_data in artworks_data]
```

Explanation

- Hardcodes 3 featured artworks (Madhubani/Warli/Kalamkari) with local D:/Brush and soul/uploads/... image paths.
- Converts dicts into Artwork objects via unpacking (**).

```
class NavigationFactory:
    @staticmethod
    def create_main_navigation() -> List[NavigationItem]:
        """Create main navigation items"""
        nav_data = [
            ("Artwork", "pages/05_Artworks.py"),
            ("Blog", "pages/06_Blogs.py"),
            ("Material", "pages/07_Materials.py"),
            ("Tutorial", "pages/08_Tutorials.py"),
            ("Portfolio", "pages/09_Portfolio.py"),
            ("Order", "pages/10_Cart.py"),
            ("Register", "pages/Register.py"),
```

```
        ("Login", "pages/Login.py")
    ]

    return [NavigationItem(label, page) for label, page in nav_data]
```

- Builds 8 navigation entries and wraps them into NavigationItem objects.

Main application

```
class BrushAndSoulApp:
  """Enhanced main application with corrected view button UI"""

  def __init__(self):
    self.config = AppConfig()
    self.theme_provider = ModernThemeProvider(self.config)
    self.artwork_factory = ArtworkFactory()
    self.navigation_factory = NavigationFactory()
```

- Creates config and supporting factories/providers.

```
    # Initialize components
    self.navigation = NavigationComponent(
      self.navigation_factory.create_main_navigation()
    )
    self.artwork_gallery = ArtworkGalleryComponent(
      self.artwork_factory.create_featured_artworks()
    )

    # Search component with corrected view buttons
    self.search_component = UniversalSearchComponent(
```

```
    self.artwork_factory.create_featured_artworks()
)
```

- Instantiates UI components:
    - Nav with main pages.
    - Gallery with featured artworks.
    - Search component (also gets the same featured artworks list; DB is still used if available).

```
# Feature sections
self.feature_sections = [
    FeatureSection(
        "For Art Lovers",
        [
            "Search by artist name for complete portfolios",
            "Discover authentic folk art with complete provenance",
            "Secure purchasing with multiple payment options",
            "Artist profiles and detailed portfolios",
            "Educational resources and step-by-step tutorials"
        ]
    ),
    FeatureSection(
        "For Artists",
        [
            "Showcase your complete portfolio to global audience",
            "Direct connection with art collectors worldwide",
            "Fair compensation and transparent pricing",
            "Artist community support and networking",
```

```
            "Professional tools for portfolio management"
        ]
    )
]

self.features_component = FeatureSectionComponent(self.feature_sections)
```

- Prepares two expanders (“Art Lovers”, “Artists”) and wraps them in the component.

```
def _setup_page_config(self) -> None:
    """Setup Streamlit page configuration"""
    st.set_page_config(
        page_title=self.config.PAGE_TITLE,
        layout=self.config.LAYOUT,
        page_icon=self.config.PAGE_ICON
    )
```

- Applies page title, layout, and icon.

```
def _apply_theme(self) -> None:
    """Apply theme styling"""
    st.markdown(self.theme_provider.get_css(), unsafe_allow_html=True)
```

- Injects the CSS returned by the theme provider.

```
def _render_header(self) -> None:
    """Render main header section"""
    st.markdown('<div class="card-3d">', unsafe_allow_html=True)
    st.markdown('<h1>Welcome To Brush and Soul</h1>', unsafe_allow_html=True)
    st.markdown('<p class="subtitle">Preserving India\'s Artistic Heritage Through Technology</p>', unsafe_allow_html=True)
    st.markdown('</div>', unsafe_allow_html=True)
```

- Fancy header with a title and subtitle inside a styled card.

```
  def _render_about_section(self) -> None:
    """Render about us section"""
    st.header("Digital Home for Indian Folk Art")
    st.write("""
    Our platform serves as a vibrant online marketplace and educational hub dedicated to
    India's traditional folk arts. We connect art lovers with authentic regional artisans
    while preserving cultural heritage through modern technology.
    """)


    self.features_component.render()
```

- About text + renders the two feature sections.

```
    st.subheader("Platform Features:")
    st.write("""
    - **Artist Name Search**: Find complete portfolios by searching artist names
    - **Comprehensive Results**: Search across artworks, materials, blogs, tutorials, and portfolios
    - **Curated Collections**: Authentic traditional artworks from verified artists
    - **Interactive Learning**: Comprehensive tutorials and educational resources
    - **Cultural Preservation**: Digital archiving of traditional art forms
    - **Community Building**: Connect with fellow art enthusiasts and professionals
    """)
```

- Bullet list of platform highlights.

```
  def _render_footer(self) -> None:
    """Render footer section"""
    st.divider()
```

```
        st.markdown(
            """
            <div class="footer-container">
                <p><strong>Brush and Soul</strong> - Bridging Traditional Indian Art with Contemporary Audiences</p>
                <p style="font-size: 0.9rem; color: var(--dark);">
                    Enhanced artist search • Cultural preservation • Artist empowerment
                </p>
                <p style="font-size: 0.8rem; color: var(--secondary);">
                    🔍 Search Artists • 🎨 Discover Art • 🛒 Purchase • 📚 Learn • 🤝 Connect
                </p>
            </div>
            """,
            unsafe_allow_html=True
        )
```

- Nice footer with taglines and icons.

```
    def run(self) -> None:
        """Main application entry point with corrected view button UI"""
        # Setup
        self._setup_page_config()
        self._apply_theme()
```

- Page config + CSS on start.

```
        # Check if there are active search results
        has_search_results = (
            hasattr(st.session_state, 'search_results') and
            st.session_state.search_results and
```

hasattr(st.session_state, 'last_search_query') and

st.session_state.last_search_query

)

- Computes a boolean to detect when the user has performed a search that yielded results.

Only render navigation when there are NO search results

if not has_search_results:

self.navigation.render()

- Hides the nav bar during search results (keeps the user focused).

Always render search component with corrected view buttons

self.search_component.render()

- Search bar and (if any) results are always visible.

If no active search, show main content

if not has_search_results:

self._render_header()

Featured artworks section

st.markdown("## Featured Artworks")

self.artwork_gallery.render()

self._render_about_section()

- Home screen content (header + featured gallery + about) shown only when the user is not in “search results mode”.

Always render footer

self._render_footer()

- Footer is always at the bottom.

Entrypoint

```
def main():
    """Application entry point with corrected view button UI"""
    app = BrushAndSoulApp()
    app.run()
```

- Creates the app and runs it.

```
if __name__ == "__main__":
    main()
```

- When the script is executed directly (not imported), call main().

-DASHBOARD CODE

1. Imports & Setup

```
import streamlit as st
from typing import List, Dict, Any, Optional
from dataclasses import dataclass
from enum import Enum
from datetime import datetime
from abc import ABC, abstractmethod
```

Explanation:

- streamlit as st → UI library for the dashboard.
- typing → type hints (List, Dict, Optional).
- dataclass → quick way to define data containers.
- Enum → for defining fixed sets (like user roles).
- datetime → timestamps for orders/activity.
- ABC, abstractmethod → for base classes that define “contracts”.

🔹 2. Type Definitions & Dataclasses

User Roles

```
class UserRole(Enum):
  ARTIST = "artist"
  CUSTOMER = "customer"
```

- Enum with two possible roles: **Artist** or **Customer**.

User Context

```
@dataclass
class UserContext:
  """User context information"""
  username: str
  role: UserRole
  email: Optional[str] = None
```

Explanation:

- Holds details about the current user.
- username → their name.
- role → artist/customer.
- email → optional field.

Artwork

```
@dataclass
class Artwork:
  id: str
  title: str
```

artist: str

price: float

status: str

created_at: datetime

Explanation:

- Represents one artwork uploaded by an artist.
- id → unique identifier.
- title, artist, price, status → metadata.
- created_at → when the artwork was added.

Order

@dataclass

class Order:

id: str

customer: str

artwork_id: str

status: str

total: float

created_at: datetime

Explanation:

- Represents one purchase order.
- id → unique order ID.
- customer → who bought it.
- artwork_id → which artwork.
- status → e.g., pending, completed.
- total → price.

- created_at → timestamp.

🔹 **3. Database Manager**

```
class DatabaseManager(ABC):
  """Abstract database manager interface"""

  @abstractmethod
  def get_user_artworks(self, username: str) -> List[Artwork]:
    pass
```

Explanation:

- Defines an **abstract base class** (interface).
- Requires child DB managers to implement methods like get_user_artworks.

(Similar abstract methods exist for get_orders_by_user, get_sales_data, etc.)

Mock Database

```
class MockDatabaseManager(DatabaseManager):
  """Mock implementation of database manager"""

  def __init__(self):
    self.artworks: List[Artwork] = []
    self.orders: List[Order] = []
```

Explanation:

- A fake in-memory database (for testing/demo).
- Stores artworks & orders in Python lists.

```
  def add_artwork(self, artwork: Artwork):
    self.artworks.append(artwork)
```

- Adds artwork to memory.

```
def add_order(self, order: Order):
    self.orders.append(order)
```

- Adds order to memory.

```
def get_user_artworks(self, username: str) -> List[Artwork]:
    return [a for a in self.artworks if a.artist == username]
```

- Filters artworks uploaded by a specific artist.

(Other methods similarly return orders or sales data.)

🔹 4. Session Manager

```
class SessionManager:
    """Handles Streamlit session state"""

    @staticmethod
    def set_user(user: UserContext):
        st.session_state["user"] = user

    @staticmethod
    def get_user() -> Optional[UserContext]:
        return st.session_state.get("user", None)
```

Explanation:

- Wrapper for handling session state in Streamlit.
- set_user → saves current user.
- get_user → retrieves logged-in user if exists.

🔹 5. UI Components

Welcome Component

```
class WelcomeComponent:
  def render(self, user: UserContext):
    st.title(f"Welcome {user.username} 👋 ")
    st.write(f"Role: {user.role.value}")
```

Explanation:

- Displays a personalized welcome message.

Artist Dashboard Component

```
class ArtistDashboardComponent:
  def __init__(self, db: DatabaseManager):
    self.db = db
```

- Dashboard for artists.
- Uses database manager.

```
  def render(self, user: UserContext):
    st.header(" 🎨 Your Artworks")
    artworks = self.db.get_user_artworks(user.username)
```

- Shows all artworks created by the logged-in artist.

```
    for art in artworks:
      st.write(f"- {art.title} | Price: {art.price} | Status: {art.status}")
```

- Loops artworks and prints details.

(Also shows sales analytics using st.metric.)

Customer Dashboard Component

```
class CustomerDashboardComponent:
  def __init__(self, db: DatabaseManager):
```

```
        self.db = db

    def render(self, user: UserContext):
        st.header("🛒 Your Orders")
        orders = self.db.get_orders_by_user(user.username)
```

Explanation:

- Shows all orders placed by the logged-in customer.
- Displays order ID, status, total, and date.

Orders Component (Admin-style view)

```
class OrdersComponent:
    def __init__(self, db: DatabaseManager):
        self.db = db
    def render(self):
        st.header("📋 All Orders")
        for order in self.db.orders:
            st.write(f"Order {order.id}: {order.status} | Total: {order.total}")
```

Explanation:

- Lists all orders (not just for one user).
- Useful for admin/artist overview.

🔹 6. Dashboard Application

```
class DashboardApp:
    def __init__(self, db: DatabaseManager):
        self.db = db
        self.welcome = WelcomeComponent()
        self.artist_dashboard = ArtistDashboardComponent(db)
```

```
        self.customer_dashboard = CustomerDashboardComponent(db)
        self.orders_component = OrdersComponent(db)
```

Explanation:

- Main dashboard app.
- Holds all components.

```
    def run(self):
        user = SessionManager.get_user()
        if not user:
            st.warning("Please login to continue.")
            return
```

- Checks if user is logged in.

```
        self.welcome.render(user)

        if user.role == UserRole.ARTIST:
            self.artist_dashboard.render(user)
        elif user.role == UserRole.CUSTOMER:
            self.customer_dashboard.render(user)
```

Explanation:

- Shows welcome message.
- Routes user to **artist dashboard** or **customer dashboard** depending on role.

```
        if st.checkbox("View All Orders"):
            self.orders_component.render()
```

- Adds optional checkbox for showing **all orders**.

🔹 7. Entry Point

```
def main():
```

db = MockDatabaseManager()

- Creates mock database.

Sample data

db.add_artwork(Artwork("1", "Madhubani Painting", "Alice", 2500, "Available", datetime.now()))

db.add_artwork(Artwork("2", "Warli Art", "Alice", 1800, "Sold", datetime.now()))

db.add_order(Order("1", "Bob", "1", "Completed", 2500, datetime.now()))

- Inserts fake sample data.

Simulate login

user = UserContext("Alice", UserRole.ARTIST, "alice@example.com")

SessionManager.set_user(user)

- Hardcoded login as **Alice (artist)**.

Run app

app = DashboardApp(db)

app.run()

- Launches the dashboard with fake DB + user.

if __name__ == "__main__":

main()

- Entry point.

ARTWORK CODE

Future Import & Standard Libraries

from __future__ import annotations

- This changes how type hints are stored in Python.
- Normally, Python evaluates type hints immediately, which can cause circular import errors.

- With this, type hints are stored as *strings* and only evaluated when needed (runtime).

import datetime

- Imports Python's built-in datetime module.
- Provides classes for working with **dates** and **times** (e.g., today's date, timestamps).

import logging

- Imports the standard logging module.
- Allows the program to record messages (info, warnings, errors) for debugging.

import os

- Imports the **OS (Operating System)** module.
- Helps in interacting with files, folders, and environment variables.
- (⚠️ In this file, it looks like os isn't directly used.)

from abc import ABC, abstractmethod

- Imports ABC (Abstract Base Class) and abstractmethod.
- Used to define classes that **must** implement certain methods.

from contextlib import contextmanager

- Imports a helper for building context managers (with … blocks).
- (⚠️ Not used in this file, might be leftover.)

from dataclasses import dataclass, field

- Imports dataclass → Automatically generates __init__, __repr__, etc. for classes.
- Imports field → Allows setting default values or default factories for dataclass attributes.

from enum import Enum

- Imports the Enum class → lets you define a set of named constants (like ARTIST, CUSTOMER).

from pathlib import Path

- Imports Path class → modern, object-oriented way to work with file paths.

from typing import Any, Dict, List, Optional, Protocol, Union

- Imports type hints:
    - Any → can be any type.
    - Dict, List → typed collections.
    - Optional[X] → either X or None.
    - Protocol → defines structural interfaces (like an interface in Java).
    - Union[X,Y] → value can be type X or type Y.

import streamlit as st

- Imports Streamlit (UI framework for data apps).
- Aliased as st so you can write st.button(), st.image(), etc.

---

2) Logging Configuration

Configure logging

logging.basicConfig(level=logging.INFO)

- Sets up basic logging configuration.
- level=logging.INFO → only shows INFO, WARNING, ERROR, and CRITICAL messages.
- Debug messages (DEBUG) won't show.

logger = logging.getLogger(__name__)

- Creates a logger object named after the current module (__name__).
- Used throughout the file with logger.info(), logger.error(), etc.

File Operation Protocol

class FileOperationProtocol(Protocol):

- Defines an **interface (protocol)** for file-related operations.
- Any class that matches this structure will be accepted as a FileOperationProtocol.

```
def save_uploaded_file(self, file: Any, subdirectory: str = "") -> Optional[str]: ...
```

- Declares a method signature: must accept a file and optional subdirectory.
- Returns either a file path (str) or None.
- The ... means it's just a placeholder (no implementation).

def delete_file(self, filepath: str) -> bool: ...

- Declares a method signature: must accept a filepath.
- Returns True if the file was deleted successfully, otherwise False.

---

4) Artwork Operation Protocol

class ArtworkOperationProtocol(Protocol):

- Another protocol (interface), this time for **artwork-related operations** in the database.

def save_artwork(self, artwork_data: Dict[str, Any]) -> Optional[int]: ...

- Save an artwork into the database.
- Takes a dictionary of artwork data.
- Returns an integer ID if saved successfully, or None if failed.

def get_artist_artworks(self, username: str) -> List[Dict[str, Any]]: ...

- Get all artworks uploaded by a particular artist.
- Input = username.
- Output = List of dictionaries (each dictionary = one artwork).

def update_artwork(self, artwork_id: int, updates: Dict[str, Any]) -> bool: ...

- Update an existing artwork's details.
- Input = artwork ID and a dictionary of updates.
- Output = True if successful, otherwise False.

def remove_artwork(self, artwork_id: int) -> bool: ...

- Delete an artwork by its ID.
- Output = True if deleted, otherwise False.

def add_to_cart(self, username: str, item: Dict[str, Any]) -> bool: ...

- Add a specific artwork to a customer's cart.
- Takes username + the artwork details.
- Output = True if added successfully, otherwise False.

def get_all_artworks(self) -> List[Dict[str, Any]]: ...

- Fetch all artworks from the database (for customer gallery view).
- Returns a list of artwork dictionaries.

---

5) UserRole Enum

class UserRole(Enum):

- Defines an **enumeration** of user roles.
- Enumeration = a fixed set of named constants.

ARTIST = "artist"

CUSTOMER = "customer"

- Two possible roles:
    - ARTIST → an artist uploading artworks.
    - CUSTOMER → a customer browsing/buying artworks.

@classmethod

def from_string(cls, role_str: str) -> 'UserRole':

- Defines a **class method** to safely convert a string into a UserRole.

try:

return cls(role_str.lower())

- Tries to convert the input string (converted to lowercase) into a UserRole.
- Example: "Artist" → "artist" → UserRole.ARTIST.

except ValueError:

logger.warning(f"Unknown user role: {role_str}")

return cls.CUSTOMER

- If the string doesn’t match (e.g., "admin"), log a warning.
- Default fallback = CUSTOMER.

---

6) UI Configuration Dataclass

@dataclass(frozen=True)

class UIConfiguration:

- A dataclass used for storing UI and app configuration.
- frozen=True → makes it **immutable** (cannot be changed after creation).

page_title: str = " 🎨 Artworks Gallery"

- Default page title shown at the top of the app.

columns_count: int = 3

- Number of columns in the gallery grid (default = 3 artworks per row).

uploads_directory: str = "uploads"

- Folder where uploaded artwork images are stored.

max_file_size_mb: int = 10

- Maximum file size for uploads = 10 MB.

allowed_file_types: List[str] = field(default_factory=lambda: ["png", "jpg", "jpeg"])

- Allowed image types: PNG, JPG, JPEG.
- Uses default_factory to avoid all instances sharing the same list.

---

7) UserContext Dataclass

@dataclass

class UserContext:

- Represents the currently logged-in user.

username: str

role: UserRole

is_authenticated: bool = True

- username → user's name (string).
- role → whether they are ARTIST or CUSTOMER.
- is_authenticated → flag, defaults to True.

@classmethod

def from_session_state(cls) -> Optional['UserContext']:

- Class method to build a UserContext object from Streamlit's session_state.

if "user" not in st.session_state or not st.session_state.get("logged_in", False):

return None

- If no "user" key or "logged_in" is False, then return None (not logged in).

user_data = st.session_state.user

- Get the user's stored data from session state.

return cls(

username=user_data.get("username", ""),

role=UserRole.from_string(user_data.get("user_type", "customer")),

is_authenticated=True

)

- Builds a new UserContext with:
  - Username (default empty string if missing).
  - Role (converted from string, defaults to CUSTOMER).
  - Authenticated = always True.

ArtworkData Dataclass

@dataclass

class ArtworkData:

- Defines a **dataclass** for representing an artwork's full details.

id: Optional[int]

title: str

description: str

materials: str

state: str

style: str

price: float

artist: str

image_path: str

created_at: datetime.datetime

Explanation of each field:

- id → Optional integer ID (auto-generated when stored in DB).
- title → name of the artwork.
- description → text description.
- materials → materials used (paper, canvas, colors).
- state → Indian state (e.g., Rajasthan, Bihar).
- style → art style (e.g., Warli, Madhubani).
- price → artwork price (float).
- artist → username of the artist.
- image_path → path of uploaded image file.
- created_at → timestamp when uploaded.

def to_dict(self) -> Dict[str, Any]:

- Converts the ArtworkData object into a dictionary.
- Useful for saving to a database or JSON.

```
return {
    "id": self.id,
```

```
            "title": self.title,
            "description": self.description,
            "materials": self.materials,
            "state": self.state,
            "style": self.style,
            "price": self.price,
            "artist": self.artist,
            "image_path": self.image_path,
            "created_at": self.created_at.isoformat()
        }
```

- Builds a dictionary with all fields.
- Converts datetime to ISO format string (2025-08-17T12:34:56).

---

9) FileManager Class

class FileManager(FileOperationProtocol):

- A class that **implements** the earlier FileOperationProtocol.
- Handles file uploads and deletions.

```
    def __init__(self, base_dir: Union[str, Path]):
        self.base_dir = Path(base_dir)
        self.base_dir.mkdir(parents=True, exist_ok=True)
```

- Constructor: takes base_dir (string or Path).
- Ensures the base directory exists → creates it if missing.

Save Uploaded File

```
    def save_uploaded_file(self, file: Any, subdirectory: str = "") -> Optional[str]:
```

- Saves an uploaded file to disk.

- Accepts:
    - file → uploaded file object.
    - subdirectory → optional subfolder inside uploads.

try:

subdir_path = self.base_dir / subdirectory if subdirectory else self.base_dir

subdir_path.mkdir(parents=True, exist_ok=True)

- If a subdirectory is given, create it inside base folder.
- Ensures the folder exists.

file_path = subdir_path / file.name

- Creates the full file path (uploads/images/mypainting.jpg).

with open(file_path, "wb") as f:

f.write(file.getbuffer())

- Opens file in **write binary** mode.
- Writes contents of uploaded file.

logger.info(f"Saved file to {file_path}")

return str(file_path)

- Logs success.
- Returns the file path as a string.

except Exception as e:

logger.error(f"Error saving file: {e}")

return None

- If error occurs (e.g., permission denied), log it.
- Return None.

Delete File

def delete_file(self, filepath: str) -> bool:

- Deletes a file from disk.

```
try:
    path = Path(filepath)
    if path.exists():
        path.unlink()
        logger.info(f"Deleted file: {filepath}")
        return True
    return False
```

- Converts string path to Path.
- If file exists: delete it → log → return True.
- If not found: return False.

```
except Exception as e:
    logger.error(f"Error deleting file: {e}")
    return False
```

- Logs and returns False if deletion fails.

SessionManager Class

```
class SessionManager:
  """Handles Streamlit session state"""
```

- A utility class to manage **Streamlit's session state**.
- Session state allows saving data (like user info, cart) between app interactions.

Set User

```
  @staticmethod
  def set_user(user: UserContext):
    st.session_state["user"] = user
    st.session_state["logged_in"] = True
```

- Saves the logged-in user object (UserContext) into session state.
- Marks "logged_in" as True.

Get User

```
@staticmethod
def get_user() -> Optional[UserContext]:
    return st.session_state.get("user", None)
```

- Retrieves the stored UserContext.
- If no user found → returns None.

Clear User

```
@staticmethod
def clear_user():
    st.session_state.pop("user", None)
    st.session_state["logged_in"] = False
```

- Logs the user out.
- Removes "user" from session state.
- Sets "logged_in" to False.

Cart Management

```
@staticmethod
def get_cart(username: str) -> List[Dict[str, Any]]:
    return st.session_state.get(f"cart_{username}", [])
```

- Retrieves a user's shopping cart from session state.
- Each user has a unique cart key: "cart_username".
- Returns empty list if no cart exists.

```
@staticmethod
def add_to_cart(username: str, item: Dict[str, Any]):
    cart_key = f"cart_{username}"
    cart = st.session_state.get(cart_key, [])
    cart.append(item)
    st.session_state[cart_key] = cart
```

- Adds an item (artwork dictionary) to user's cart.
- Updates session state.

```
@staticmethod
def clear_cart(username: str):
    st.session_state[f"cart_{username}"] = []
```

- Empties the user's cart.

---

11) ArtworkManager Class

class ArtworkManager(ArtworkOperationProtocol):

- Manages **artworks + cart operations**.
- Implements the earlier ArtworkOperationProtocol.

Constructor

```
def __init__(self, file_manager: FileManager):
    self.file_manager = file_manager
    self.artworks: Dict[int, ArtworkData] = {}
    self.next_id = 1
```

- Takes a FileManager for handling file uploads/deletions.
- Stores artworks in a dictionary (artwork_id → ArtworkData).
- Starts ID counter at 1.

Save Artwork

def save_artwork(self, artwork_data: Dict[str, Any]) -> Optional[int]:

- Saves new artwork into memory (not database).

```
try:
    artwork = ArtworkData(
        id=self.next_id,
        title=artwork_data["title"],
        description=artwork_data["description"],
        materials=artwork_data["materials"],
        state=artwork_data["state"],
        style=artwork_data["style"],
        price=float(artwork_data["price"]),
        artist=artwork_data["artist"],
        image_path=artwork_data["image_path"],
        created_at=datetime.datetime.now()
    )
```

- Creates an ArtworkData object from given dictionary.
- Converts price to float.
- Sets creation time = current time.
- Assigns auto-increment ID.

```
    self.artworks[self.next_id] = artwork
    self.next_id += 1
    logger.info(f"Artwork saved: {artwork.title} by {artwork.artist}")
    return artwork.id
```

- Stores artwork in dictionary.

- Increments ID counter.
- Logs success.
- Returns the ID.

except Exception as e:

logger.error(f"Error saving artwork: {e}")

return None

- Logs and returns None if an error occurs.

Get Artist's Artworks

def get_artist_artworks(self, username: str) -> List[Dict[str, Any]]:

return [art.to_dict() for art in self.artworks.values() if art.artist == username]

- Retrieves all artworks uploaded by a given artist.
- Converts each ArtworkData into a dictionary.

Update Artwork

def update_artwork(self, artwork_id: int, updates: Dict[str, Any]) -> bool:

- Updates an existing artwork.

if artwork_id not in self.artworks:

return False

- If artwork doesn't exist → return False.

artwork = self.artworks[artwork_id]

for key, value in updates.items():

if hasattr(artwork, key):

setattr(artwork, key, value)

- For each update key-value pair: check if ArtworkData has that attribute.
- If yes → update it.

logger.info(f"Artwork updated: {artwork.title}")

return True

- Logs update success.
- Returns True.

Remove Artwork

def remove_artwork(self, artwork_id: int) -> bool:

- Deletes an artwork by ID.

if artwork_id in self.artworks:

artwork = self.artworks.pop(artwork_id)

self.file_manager.delete_file(artwork.image_path)

logger.info(f"Artwork removed: {artwork.title}")

return True

return False

- If artwork exists:
    - Remove from dictionary.
    - Delete image file.
    - Log success.
    - Return True.
- If not found → return False.

Add to Cart

def add_to_cart(self, username: str, item: Dict[str, Any]) -> bool:

try:

SessionManager.add_to_cart(username, item)

return True

except Exception as e:

logger.error(f"Error adding to cart: {e}")

return False

- Adds an artwork item to a user's cart via SessionManager.
- Returns True if success, False otherwise.

Get All Artworks

def get_all_artworks(self) -> List[Dict[str, Any]]:

return [art.to_dict() for art in self.artworks.values()]

- Returns all artworks in the system as dictionaries.

✅ Up to here, we've explained:

- **SessionManager** (handles user session + cart).
- **ArtworkManager** (handles saving, updating, deleting artworks + cart).

------BLOGS CODE

Imports & Setup

from __future__ import annotations

- Postpones evaluation of type hints (stores them as strings).
- Prevents circular import issues.

import logging

import datetime

from dataclasses import dataclass

from typing import List, Dict, Optional, Any, Protocol

from abc import ABC, abstractmethod

import streamlit as st

Explanation:

- logging → record app messages.
- datetime → handle dates/timestamps.
- dataclass → easy-to-make data containers.
- typing → type hints for safety (List, Dict, Optional, Any).
- Protocol → defines an interface.
- ABC, abstractmethod → abstract base classes.
- streamlit as st → UI framework.

Logging Configuration

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(__name__)

- Configure logging to show INFO+ messages.
- Create a logger for this module.

---

2) Blog Protocol (Interface)

class BlogOperationProtocol(Protocol):

- Defines **expected methods** for blog management.
- Any implementing class must follow these.

```
def save_blog(self, blog_data: Dict[str, Any]) -> Optional[int]: ...

def get_user_blogs(self, username: str) -> List[Dict[str, Any]]: ...

def get_all_blogs(self) -> List[Dict[str, Any]]: ...

def update_blog(self, blog_id: int, updates: Dict[str, Any]) -> bool: ...

def delete_blog(self, blog_id: int) -> bool: ...
```

Explanation:

- save_blog → Save a blog post. Returns blog ID or None.
- get_user_blogs → Fetch blogs by a user.
- get_all_blogs → Fetch all blogs.
- update_blog → Modify a blog by ID.
- delete_blog → Remove a blog by ID.

---

3) Blog Data Model

```
@dataclass
class BlogData:
  id: Optional[int]
  title: str
  content: str
  author: str
  created_at: datetime.datetime
```

Explanation:

- Represents one blog entry.
- id → unique blog ID.
- title → blog title.
- content → main text.
- author → who wrote it.
- created_at → when it was created.

```
  def to_dict(self) -> Dict[str, Any]:
    return {
      "id": self.id,
      "title": self.title,
      "content": self.content,
```

```
            "author": self.author,
            "created_at": self.created_at.isoformat()
        }
```

- Converts the dataclass into a dictionary (easier for storage).
- Converts datetime → ISO string.

---

4) Blog Manager (Implements Protocol)

```
class BlogManager(BlogOperationProtocol):
    def __init__(self):
        self.blogs: Dict[int, BlogData] = {}
        self.next_id = 1
```

- Stores blogs in memory (dict: id → BlogData).
- Starts ID counter at 1.

Save Blog

```
    def save_blog(self, blog_data: Dict[str, Any]) -> Optional[int]:
        try:
            blog = BlogData(
                id=self.next_id,
                title=blog_data["title"],
                content=blog_data["content"],
                author=blog_data["author"],
                created_at=datetime.datetime.now()
            )
            self.blogs[self.next_id] = blog
            self.next_id += 1
```

```
            logger.info(f"Blog saved: {blog.title} by {blog.author}")
            return blog.id
        except Exception as e:
            logger.error(f"Error saving blog: {e}")
            return None
```

Explanation:

- Creates a new BlogData object.
- Stores it in memory.
- Increments ID.
- Logs success.
- Returns ID (or None if failed).

Get User Blogs

```
    def get_user_blogs(self, username: str) -> List[Dict[str, Any]]:
        return [blog.to_dict() for blog in self.blogs.values() if blog.author == username]
```

- Returns all blogs written by the given username.

Get All Blogs

```
    def get_all_blogs(self) -> List[Dict[str, Any]]:
        return [blog.to_dict() for blog in self.blogs.values()]
```

- Returns all stored blogs.

Update Blog

```
    def update_blog(self, blog_id: int, updates: Dict[str, Any]) -> bool:
        if blog_id not in self.blogs:
            return False
```

```
        blog = self.blogs[blog_id]
        for key, value in updates.items():
            if hasattr(blog, key):
                setattr(blog, key, value)
        logger.info(f"Blog updated: {blog.title}")
        return True
```

- If blog exists: update the provided fields.
- Uses setattr to update dynamically.
- Returns True if successful.

Delete Blog

```
    def delete_blog(self, blog_id: int) -> bool:
        if blog_id in self.blogs:
            removed = self.blogs.pop(blog_id)
            logger.info(f"Blog deleted: {removed.title}")
            return True
        return False
```

- Removes blog by ID if found.
- Logs deletion.
- Returns True if successful.

---

5) Blog Components (Streamlit UI)

Blog Form Component

```
class BlogFormComponent:
    def __init__(self, manager: BlogManager):
        self.manager = manager
```

- UI form for creating new blogs.
- Uses BlogManager to save data.

```
  def render(self, user: str):
    st.subheader(" ✍️ Write a New Blog")
    with st.form("blog_form"):
      title = st.text_input("Title")
      content = st.text_area("Content")
      submitted = st.form_submit_button("Publish Blog")
      if submitted and title and content:
        blog_data = {
          "title": title,
          "content": content,
          "author": user
        }
        self.manager.save_blog(blog_data)
        st.success("Blog published successfully!")
```

Explanation:

- Streamlit form with fields: title + content.
- On submit:
    - Creates blog dictionary.
    - Saves using BlogManager.
    - Shows success message.

Blog List Component

```
class BlogListComponent:
  def __init__(self, manager: BlogManager):
```

```
        self.manager = manager
```

- Displays list of blogs.

```
    def render(self, blogs: List[Dict[str, Any]]):
        st.subheader("📚 Blogs")
        for blog in blogs:
            with st.expander(blog["title"], expanded=False):
                st.write(blog["content"])
                st.caption(f"Written by {blog['author']} on {blog['created_at']}")
```

Explanation:

- Shows collapsible expanders for each blog.
- Inside: content + author + date.

---

6) Main Blog App

```
class BlogApp:
    def __init__(self):
        self.manager = BlogManager()
        self.form = BlogFormComponent(self.manager)
        self.list = BlogListComponent(self.manager)
```

- Combines manager + UI components.

```
    def run(self, current_user: Optional[str] = None):
        st.title("📝 Blog Section")
        if current_user:
            self.form.render(current_user)
        blogs = self.manager.get_all_blogs()
        self.list.render(blogs)
```

Explanation:

- Sets app title.
- If user logged in → show blog form.
- Always show all blogs.

---

7) Entry Point

```
def main():
    app = BlogApp()
    # Simulate a logged-in user
    current_user = "Alice"
    app.run(current_user)
```

- Creates BlogApp.
- Simulates login as "Alice".
- Runs app.

```
if __name__ == "__main__":
    main()
```

- Standard Python entry point.
- Runs the app if script is executed directly.

----MATERIALS CODES

from __future__ import annotations

Tells Python to store type annotations as strings (deferred evaluation).

Avoids forward-reference / circular import problems and can slightly speed imports.

python

Copy

Edit

import logging

Imports Python’s logging module for writing info/warn/error messages.

python

Copy

Edit

import os

Imports OS utilities for file/path checks (used later to check image existence).

python

Copy

Edit

import time

Imports the time module (not heavily used; commonly for small delays or timestamps).

python

Copy

Edit

from abc import ABC, abstractmethod

Imports tools to create abstract base classes and abstract methods (used to define UI base class later).

python

Copy

Edit

from dataclasses import dataclass, field

@dataclass for concise data containers.

field for default_factory and other field-level options.

python

Copy

Edit

from datetime import datetime

Imports the datetime class for timestamps / formatting dates.

python

Copy

Edit

from enum import Enum

Enables creating enumerations (UserRole below).

python

Copy

Edit

from pathlib import Path

Path provides an object-oriented interface for filesystem paths.

python

Copy

Edit

from typing import Any, Dict, List, Optional, Protocol

Type-hinting helpers used throughout the file: Any, Dict, List, Optional, and structural Protocol.

python

Copy

Edit

import streamlit as st

Imports Streamlit (UI framework) and aliases it st. All UI calls (buttons, images, forms) use st.

Logging configuration

python

Copy

Edit

Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(__name__)

basicConfig(level=logging.INFO) sets the default logging level to INFO (so INFO/WARNING/ERROR appear).

logger = logging.getLogger(__name__) creates a module-named logger used for logging messages in this file.

TYPE DEFINITIONS & PROTOCOLS — interfaces and enums

python

Copy

Edit

```
class MaterialOpsProtocol(Protocol):
    """Database-side operations required by this page."""
    def get_all_materials(self) -> List[Dict[str, Any]]: ...
    def get_user_materials(self, username: str) -> List[Dict[str, Any]]: ...
    def save_material(self, material: Dict[str, Any]) -> Optional[int]: ...
    def delete_material(self, material_id: int) -> bool: ...
    def add_to_cart(self, username: str, item: Dict[str, Any]) -> bool: ...
```

Declares a structural interface (Protocol) listing methods the code expects from the database utilities.

Any object implementing these methods can be used (duck typing). Methods: fetch all, fetch by user, save, delete, add-to-cart.

python

Copy

Edit

```
class FileOpsProtocol(Protocol):
    """Shared file-utility operations (uploads, deletes, ...)."""
    def save_uploaded_file(self, file: Any, subdirectory: str = "") -> Optional[str]: ...
```

Simple protocol for file upload utilities. The save_uploaded_file should return a string path if successful, or None if not.

python

Copy

Edit

```
class UserRole(Enum):
    ARTIST = "artist"
    CUSTOMER = "customer"

    @classmethod
    def from_string(cls, raw: str) -> "UserRole":
        try:
            return cls(raw.lower())
        except ValueError:
            return cls.CUSTOMER
```

UserRole enum defines two roles: ARTIST and CUSTOMER.

from_string normalizes a string to a UserRole. If the string is invalid, it silently falls back to CUSTOMER.

CONFIG & DATA MODELS — UI config, session context, material model

python

Copy

Edit

```
@dataclass(frozen=True)
class UIConfig:
    page_title: str = "🎨 Art Materials Marketplace"
    layout: str = "wide"
    uploads_dir: str = "uploads/materials"
    columns: int = 3
    allowed_types: List[str] = field(default_factory=lambda: ["jpg", "jpeg", "png"])
```

Immutable (frozen=True) dataclass holding UI and platform configuration: page title, layout, where to store uploads, grid columns, and allowed file types.

python

Copy

Edit

```
@dataclass
class UserCtx:
  username: str
  role: UserRole
  is_authenticated: bool = True

  @classmethod
  def from_session(cls) -> Optional["UserCtx"]:
    if "user" not in st.session_state or not st.session_state.get("logged_in", False):
      return None
    u = st.session_state.user
    return cls(username=u["username"],
          role=UserRole.from_string(u.get("user_type", "customer")))
```

UserCtx represents current user context.

from_session() reads Streamlit's st.session_state to build a UserCtx. If the user is not logged in, returns None.

python

Copy

Edit

@dataclass

class Material:

"""Strongly-typed representation of a material."""

id: int

name: str

price: float

category: str

description: str

artist: str

image: Optional[str] = None

listed_date: str = field(default_factory=lambda: datetime.now().strftime("%Y-%m-%d"))

Material is the in-app, strongly typed representation of a material listing: id, name, price, category, description, seller (artist), optional image path, and a listed_date defaulting to today (YYYY-MM-DD).

python

Copy

Edit

@classmethod

def from_dict(cls, d: Dict[str, Any]) -> "Material":

return cls(

id=d.get('id', 0),

name=d.get('name', 'Unknown'),

price=float(d.get('price', 0)),

category=d.get('category', 'Other'),

description=d.get('description', ''),

artist=d.get('artist', '') or d.get('seller', 'Unknown Artist'),

```
            image=d.get('image') or d.get('image_path'),
            listed_date=d.get('listed_date', datetime.now().strftime("%Y-%m-%d"))
        )
```

from_dict hydrates a Material from a dictionary (the shape returned by DB utils). It provides safe defaults and tries alternate keys for artist and image.

python

Copy

Edit

```
    def to_dict(self) -> Dict[str, Any]:
        return {
            'seller': self.artist,
            'name': self.name,
            'description': self.description,
            'price': self.price,
            'category': self.category,
            'image_path': self.image,
            'listed_date': self.listed_date
        }
```

to_dict serializes Material back into the dictionary shape expected by the database utilities: 'seller' (artist), 'image_path', etc.

FILE MANAGER — safe upload handling

python

Copy

Edit

class FileManager:

```
    """FIXED - File operations manager with comprehensive error handling"""

    def __init__(self, cfg: UIConfig):
        self.cfg = cfg
        Path(cfg.uploads_dir).mkdir(parents=True, exist_ok=True)
        self.operations_available = True
        try:
            from utils import save_uploaded_file
            self.save_uploaded_file = save_uploaded_file
            logger.info("File operations initialized successfully")
        except ImportError as e:
            logger.error(f"File utilities not available: {e}")
            self.operations_available = False
            st.error(f"File operations not available: {e}")
```

FileManager constructor:

Stores config.

Ensures the uploads directory exists (mkdir(parents=True, exist_ok=True)).

Tries to import the save_uploaded_file helper from utils. If successful, binds it; otherwise marks file operations unavailable and shows an in-app error.

python

Copy

Edit

```
    def save(self, file: Any) -> Optional[str]:
        """FIXED - Save uploaded file with enhanced error handling"""
        if not file:
            logger.warning("No file provided to save")
            return None

        if not self.operations_available:
            logger.error("File operations not available")
            st.error("File operations are not available")
            return None
```

save method starts by validating arguments and availability:

If file is falsy, warn and return None.

If the utils helper wasn't available at init, surface an error and return None.

python

Copy

Edit

```
        # Validate file size (max 10MB)
        if hasattr(file, 'size') and file.size > 10 * 1024 * 1024:
            error_msg = f"File size too large. Maximum 10MB allowed."
            logger.error(error_msg)
            st.error(error_msg)
            return None
```

Checks file size when available (Streamlit uploaded files have .size). If >10MB, reject and notify the user.

python

Copy

Edit

```
    try:
        logger.info(f"Saving file: {file.name}")
        result = self.save_uploaded_file(file, "materials")
        if result:
            logger.info(f"File saved successfully: {result}")
            return result
        else:
            logger.error("File save utility returned None")
            return None
    except Exception as e:
        logger.error(f"Error saving file: {e}")
        st.error(f"Error saving file: {e}")
        return None
```

Delegates actual saving to the utils.save_uploaded_file(file, "materials") function:

On success returns the saved path.

If helper returns None or raises an exception, logs and returns None; also shows a UI error.

DATABASE MANAGER — robust DB operations + transactions

python

Copy

Edit

```
class DatabaseManager:
  """FIXED - Database operations manager with explicit transaction handling"""

  def __init__(self):
    self.operations_available = True
    try:
      from utils import (
        get_all_materials,
        get_user_materials,
        save_material,
        delete_material,
        add_to_cart,
      )

      self.get_all_materials = get_all_materials
      self.get_user_materials = get_user_materials
      self.save_material = save_material
      self.delete_material = delete_material
      self.add_to_cart = add_to_cart

      logger.info("Database operations initialized successfully")

    except ImportError as e:
```

logger.error(f"Database utilities not available: {e}")

self.operations_available = False

st.error(f"Database operations not available: {e}")

Constructor attempts to import DB helper functions from utils (get_all_materials, get_user_materials, save_material, delete_material, add_to_cart).

On failure, disables DB features and shows error. On success, stores bound functions on the instance.

python

Copy

Edit

```
  def test_database_connection(self) -> bool:
    """Test database connection explicitly"""
    try:
      if not self.operations_available:
        return False

      # Try to execute a simple query
      from utils import _instance
      instance = _instance()

      with instance.get_connection() as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT 1")
        result = cursor.fetchone()
        return result[0] == 1
```

```
        except Exception as e:
            logger.error(f"Database connection test failed: {e}")
            return False
test_database_connection:
```

Quick health-check to ensure DB is reachable.

Uses a low-level _instance() helper (assumed in utils) to get a connection object and executes SELECT 1.

Returns True only if the fetch returns 1. Errors logged and return False.

python

Copy

Edit

```
    def save_material_with_transaction(self, material: Material) -> bool:
        """Save material with explicit transaction handling"""
        try:
            from utils import _instance
            instance = _instance()

            with instance.get_connection() as conn:
                cursor = conn.cursor()

                # Start transaction explicitly
                cursor.execute("START TRANSACTION")
```

```
        try:
            # Format listed_date properly
            listed_date = datetime.now().strftime("%Y-%m-%d")

            # Insert material
            cursor.execute(
                """INSERT INTO materials (seller, name, description, price, category,
image_path, listed_date)
                VALUES (%s, %s, %s, %s, %s, %s, %s)""",
                (
                    material.artist,
                    material.name,
                    material.description,
                    material.price,
                    material.category,
                    material.image,
                    listed_date
                )
            )

            material_id = cursor.lastrowid

            # Commit transaction
            cursor.execute("COMMIT")
```

```
            logger.info(f"Material saved successfully with ID: {material_id}")
            st.success(f"✅ Material saved with ID: {material_id}")
            return True

        except Exception as e:
            # Rollback on error
            cursor.execute("ROLLBACK")
            logger.error(f"Transaction failed, rolled back: {e}")
            raise

    except Exception as e:
        logger.error(f"Error saving material with transaction: {e}")
        return False
```

save_material_with_transaction performs a manual DB transaction:

Uses low-level DB connection from utils._instance().

Executes START TRANSACTION, runs the INSERT statement with the material fields.

Grabs cursor.lastrowid to know inserted id.

COMMIT on success; ROLLBACK on inner exception.

Returns True on commit; logs and returns False on outer exceptions.

Note: transaction control via explicit SQL commands; also shows success message in UI.

python

Copy

Edit

```
  def verify_material_saved(self, username: str, name: str) -> bool:
    """Verify material was actually saved to database"""
    try:
      from utils import get_user_materials

      # Fetch materials for this user
      materials = get_user_materials(username)

      # Check if the material with this name exists
      for material in materials:
        if material.get('name') == name:
          return True

      return False
    except Exception as e:
      logger.error(f"Error verifying material: {e}")
      return False
```

verify_material_saved re-fetches materials for a user and checks if the material with the given name exists. Returns boolean. Provides a second-level assurance that the insert actually persisted.

python

Copy

Edit

```
  def fetch_all(self) -> List[Material]:
    """Fetch all materials from database with proper error handling"""
    if not self.operations_available:
      st.error("Database operations not available")
      return []

    try:
      materials_data = self.get_all_materials()
      logger.info(f"Fetched {len(materials_data)} materials from database")
      return [Material.from_dict(m) for m in materials_data]
    except Exception as e:
      logger.error(f"Error fetching materials: {e}")
      st.error(f"Error loading materials: {e}")
      return []
```

Fetches all materials using bound get_all_materials helper, converts each dict into a Material object via Material.from_dict. Handles DB-unavailable case and exceptions gracefully.

python

Copy

Edit

```
  def fetch_by_artist(self, username: str) -> List[Material]:
    """Fetch materials by artist/seller with proper error handling"""
    if not self.operations_available:
      st.error("Database operations not available")
      return []
```

```
        try:
            materials_data = self.get_user_materials(username)
            logger.info(f"Fetched {len(materials_data)} materials for artist {username}")
            return [Material.from_dict(m) for m in materials_data]
        except Exception as e:
            logger.error(f"Error fetching artist materials: {e}")
            st.error(f"Error loading your materials: {e}")
            return []
```

Fetches materials for a specific artist and converts output into Material objects.

python

Copy

Edit

```
    def add(self, mat: Material) -> bool:
        """FIXED - Add new material with comprehensive error handling and verification"""
        if not self.operations_available:
            logger.error("Database operations not available")
            st.error("Database operations not available")
            return False

        try:
            logger.info(f"Testing database connection...")
            if not self.test_database_connection():
                st.error("❌ Database connection failed")
                return False
```

```
            st.success("✅ Database connection established")

            logger.info(f"Saving material with transaction handling...")
            success = self.save_material_with_transaction(mat)

            if success:
                logger.info(f"Verifying material was saved...")
                if self.verify_material_saved(mat.artist, mat.name):
                    st.success("✅ Material verified in database!")
                    return True
                else:
                    st.error("❌ Material was not properly saved to database")
                    return False
            else:
                st.error("❌ Failed to save material to database")
                return False

        except Exception as e:
            logger.error(f"❌ Exception in add material: {e}")
            import traceback
            logger.error(f"Full traceback: {traceback.format_exc()}")
            st.error(f"Database error: {e}")
            return False
```

add is a high-level wrapper that:

Verifies DB availability.

Runs test_database_connection and aborts early if it fails.

Calls save_material_with_transaction.

Verifies persistence via verify_material_saved.

Returns True only if everything succeeded; provides user feedback via st.success/st.error and logs tracebacks on exceptions.

python

Copy

Edit

```
  def remove(self, mat_id: int) -> bool:
    """Remove material from database with proper error handling"""
    if not self.operations_available:
      return False

    try:
      result = self.delete_material(mat_id)
      if result:
        logger.info(f"Material {mat_id} deleted successfully")
      return result
    except Exception as e:
```

```
        logger.error(f"Error deleting material: {e}")
        st.error(f"Error deleting material: {e}")
        return False
```

Wrapper for deleting a material by id via delete_material. Handles errors and returns boolean result.

python

Copy

Edit

```
    def add_to_cart_item(self, username: str, mat: Material) -> bool:
        """Add material to user's cart with proper error handling"""
        if not self.operations_available:
            return False

        try:
            material_dict = mat.to_dict()
            # Ensure proper format for cart
            material_dict['id'] = mat.id
            material_dict['title'] = mat.name  # Map name to title for cart compatibility
            result = self.add_to_cart(username, material_dict)
            if result:
                logger.info(f"Material {mat.name} added to cart for {username}")
            return result
        except Exception as e:
            logger.error(f"Error adding to cart: {e}")
            st.error(f"Error adding to cart: {e}")
```

```
        return False
```

Prepares material dict (maps name → title and ensures id) then calls add_to_cart helper; returns boolean result; logs and surfaces errors.

UI COMPONENTS — base abstract class + styling injection

python

Copy

Edit

```
class UIComponent(ABC):
  """Abstract base class for UI components"""

  def __init__(self, cfg: UIConfig, db: DatabaseManager, files: FileManager):
    self.cfg = cfg
    self.db = db
    self.files = files

  @abstractmethod
  def render(self, user_ctx: UserCtx) -> None:
    """Render the UI component"""
    pass
```

Abstract base class for UI components: stores cfg, db, and files. Subclasses must implement render(user_ctx) method.

python

Copy

Edit

```
class StyleManager(UIComponent):
```

```
    """Handles comprehensive brown CSS styling - FULLY CORRECTED VERSION"""

    def render(self, user_ctx: UserCtx) -> None:
        """Apply comprehensive brown CSS styling with all corrections"""
        st.markdown("""
        <style>
        :root {
            --primary: #8B4513;
            --secondary: #A0522D;
            --accent: #5C4033;
            --light: #F8F4E8;
            --dark: #343434;
            --brown-light: #D2B48C;
            --brown-medium: #CD853F;
            --brown-dark: #8B4513;
            --brown-darker: #5C4033;
            --brown-lightest: #F5DEB3;
        }
        ...
        </style>
        """, unsafe_allow_html=True)
```

StyleManager.render injects a large CSS <style> blob via st.markdown(..., unsafe_allow_html=True) that:

Defines CSS variables for the brown color palette.

Styles .stApp, main title, dividers, .material-card appearance, images, placeholders, buttons, expanders, detail sections, form controls, responsive adjustments and animations (fadeInUp).

Note: Instead of explaining every single CSS line, the important takeaway: this block sets the entire app's visual theme (typography, spacing, card look, buttons, placeholders, mobile tweaks). The CSS classes used later in the UI (material-card, material-title, no-image-placeholder, etc.) rely on these styles.

BrowseTab — display of all materials for browsing

python

Copy

Edit

```
class BrowseTab(UIComponent):
  """Browse materials tab component - CORRECTED VERSION"""

  def render(self, user_ctx: UserCtx) -> None:
    """Render browse materials tab with fully corrected layout"""
    all_materials = self.db.fetch_all()
```

BrowseTab.render fetches all materials from DB manager.

python

Copy

Edit

```
    if all_materials:
      # No message displayed - clean display of materials
```

```
        cols = st.columns(self.cfg.columns)
        for idx, mat in enumerate(all_materials):
            with cols[idx % self.cfg.columns]:
                # Material container - fully transparent
                st.markdown('<div class="material-card">', unsafe_allow_html=True)

                # Material Title - no white box, gradient text
                st.markdown(f'<div class="material-title">{mat.name}</div>',
unsafe_allow_html=True)

                # Artist info
                st.markdown(f'<div class="material-artist">🎨 {mat.artist}</div>',
unsafe_allow_html=True)

                # Category badge
                st.markdown(f'<div class="category-badge">{mat.category}</div>',
unsafe_allow_html=True)
```

If materials exist:

Create self.cfg.columns columns.

Loop through materials and place each item in a column (round-robin using modulo).

Render a container .material-card and then the title, artist, and a category badge using previously injected CSS.

python

Copy

Edit

```
        # Material Image
        if mat.image and os.path.exists(mat.image):
            st.image(mat.image, use_container_width=True)
        else:
            st.markdown(
                '<div class="no-image-placeholder">📦 <br>No Image Available</div>',
                unsafe_allow_html=True
            )
```

If there is an image path and the file exists on disk (os.path.exists), show it using st.image. Otherwise show a stylized placeholder (.no-image-placeholder).

python

Copy

Edit

```
        # Price display
        st.markdown(f'<div class="material-price-display">₹{mat.price:.1f}</div>', unsafe_allow_html=True)
```

Show price with a specialized style.

python

Copy

Edit

```
        # View Details Dropdown - corrected structure
        with st.expander("📋 View Details"):
```

st.markdown(' ', unsafe_allow_html=True)

Description

st.markdown(' ', unsafe_allow_html=True)

st.markdown(' 📝 Description ', unsafe_allow_html=True)

st.markdown(f' {mat.description} ', unsafe_allow_html=True)

st.markdown(' ', unsafe_allow_html=True)

st.markdown(' ', unsafe_allow_html=True)

A st.expander holds the details section:

Shows a labeled Description block with .description-content class for readable content.

python

Copy

Edit

Add to cart button BELOW the expander

if st.button("🛒 Add to Cart", key=f"cart_{mat.id}_{user_ctx.username}"):

if self.db.add_to_cart_item(user_ctx.username, mat):

st.success(f"✅ Added {mat.name} to cart!")

st.switch_page("pages/10_Cart.py")

else:

st.error("❌ Could not add to cart at this time.")

```
            st.markdown('</div>', unsafe_allow_html=True)
            st.markdown('<div class="brown-divider"></div>', unsafe_allow_html=True)
```

Renders Add-to-cart button with a unique key to avoid widget collisions. On click:

Calls self.db.add_to_cart_item(...). On success shows success and navigates to cart page. On failure shows an error.

Closes the material card and renders a separator.

python

Copy

Edit

```
    else:
        st.markdown('<div class="empty-state brown-card">', unsafe_allow_html=True)
        if user_ctx.role == UserRole.ARTIST:
            st.markdown("### No materials available yet")
            st.markdown(" 🎨 Be the first to list some quality art supplies!")
        else:
            st.markdown("### No materials available yet")
            st.markdown(" 📦 Check back soon for amazing art supplies from our community!")
        st.markdown('</div>', unsafe_allow_html=True)
```

If no materials exist, shows an empty-state card with different copy for artists vs customers.

YourMaterialsTab — upload form + artist's materials list

python

Copy

Edit

```
class YourMaterialsTab(UIComponent):
    """Your materials tab component - CORRECTED VERSION WITH FIXED FORM"""

    def render(self, user_ctx: UserCtx) -> None:
        """Render your materials tab with corrected styling and fixed form"""
        # Show upload form first
        self._render_upload_form(user_ctx)

        st.markdown('<div class="brown-divider"></div>', unsafe_allow_html=True)

        # Show existing materials
        self._render_existing_materials(user_ctx)
```

render shows upload form first, then divider, then the list of materials owned by the artist.

python

Copy

Edit

```
    def _render_existing_materials(self, user_ctx: UserCtx) -> None:
        """Render list of existing materials with corrected layout"""
        yours = self.db.fetch_by_artist(user_ctx.username)
```

Fetches the logged-in artist's materials.

python

Copy

```
Edit

    if yours:

        st.markdown(f'<h3 style="color: var(--primary);">📦 Your Listed Materials ({len(yours)})</h3>',
                    unsafe_allow_html=True)


        # Use columns layout like browse tab

        cols = st.columns(self.cfg.columns)

        for idx, mat in enumerate(yours):

          with cols[idx % self.cfg.columns]:

            # Material container - fully transparent

            st.markdown('<div class="material-card">', unsafe_allow_html=True)


            # Material Title - no white box

            st.markdown(f'<div class="material-title">{mat.name}</div>', unsafe_allow_html=True)


            # Artist info

            st.markdown(f'<div class="material-artist">🎨 {mat.artist}</div>', unsafe_allow_html=True)


            # Category badge

            st.markdown(f'<div class="category-badge">{mat.category}</div>', unsafe_allow_html=True)


            # Material Image

            if mat.image and os.path.exists(mat.image):
```

```
            st.image(mat.image, use_container_width=True)
        else:
            st.markdown(
                '<div class="no-image-placeholder">📦 <br>No Image</div>',
                unsafe_allow_html=True
            )

        # Price display
        st.markdown(f'<div class="material-price-display">₹{mat.price:.1f}</div>', unsafe_allow_html=True)
```

For each material in the artist's list: shows title, artist, category, image or placeholder, and price — essentially the artist view of the same grid.

python

Copy

Edit

```
        # View Details Dropdown
        with st.expander("📋 View Details"):
            st.markdown('<div class="details-section">', unsafe_allow_html=True)

            # Description
            st.markdown('<div class="detail-item">', unsafe_allow_html=True)
            st.markdown('<div class="detail-label">📝 Description</div>', unsafe_allow_html=True)
            st.markdown(f'<div class="description-content">{mat.description}</div>', unsafe_allow_html=True)
            st.markdown('</div>', unsafe_allow_html=True)
```

```
            # Listed Date
            st.markdown('<div class="detail-item">', unsafe_allow_html=True)
            st.markdown('<div class="detail-label">🗓️ Listed Date</div>',
unsafe_allow_html=True)
            st.markdown(f'<div class="detail-value">{mat.listed_date}</div>',
unsafe_allow_html=True)
            st.markdown('</div>', unsafe_allow_html=True)


            st.markdown('</div>', unsafe_allow_html=True)
```

Artist sees description and listed date inside the details expander.

python

Copy

Edit

```
        # Delete button with confirmation
        if st.button("🗑️ Remove", key=f"del_{mat.id}_{user_ctx.username}"):
          if st.session_state.get(f"confirm_delete_{mat.id}", False):
            if self.db.remove(mat.id):
              st.success("✅ Material removed!")
              st.session_state[f"confirm_delete_{mat.id}"] = False
              st.rerun()
            else:
              st.error("❌ Delete failed.")
          else:
            st.session_state[f"confirm_delete_{mat.id}"] = True
```

```
                st.rerun()


            # Show confirmation message if delete was clicked
            if st.session_state.get(f"confirm_delete_{mat.id}", False):
                st.warning("⚠️ Click Remove again to confirm")


            st.markdown('</div>', unsafe_allow_html=True)
            st.markdown('<div class="brown-divider"></div>', unsafe_allow_html=True)
```

Delete flow:

First click sets a session flag confirm_delete_{mat.id} to True and reruns (so user gets a visible confirmation prompt).

Second click (when flag is True) actually calls self.db.remove(mat.id). On success the flag is cleared and the page reruns to update UI.

This two-click pattern prevents accidental deletions.

python

Copy

Edit

```
    else:
        st.markdown('<div class="empty-state brown-card">', unsafe_allow_html=True)
        st.markdown("### You haven't listed any materials yet")
        st.markdown("📦 Use the form above to list your first art material!")
        st.markdown('</div>', unsafe_allow_html=True)
```

If the artist has no materials, shows a friendly empty state encouraging them to list items.

Upload form implementation (YourMaterialsTab._render_upload_form)

python

Copy

Edit

```
  def _render_upload_form(self, user_ctx: UserCtx) -> None:
    """FIXED - Render material upload form with guaranteed database saving"""
    st.markdown('<h3 style="color: var(--accent);">➕ List New Art Material</h3>', unsafe_allow_html=True)

    # Form container with brown styling
    st.markdown('<div class="brown-card">', unsafe_allow_html=True)

    form_key = "material_upload_form"

    with st.form(key=form_key, clear_on_submit=False):  # Don't auto-clear
      col1, col2 = st.columns(2)

      with col1:
        name = st.text_input(
          "Material Name*",
          placeholder="e.g., Professional Acrylic Paint Set",
          help="Choose a descriptive name that buyers will search for"
        )
        price = st.number_input(
```

```
        "Price (₹)*",
        min_value=0.0,
        step=1.0,
        help="Set a competitive price for your material"
    )

with col2:
    category_options = [
        "Select Category",
        "🎨 Paints & Pigments",
        "🖌️ Brushes & Tools",
        "📄 Canvas & Paper",
        "🖍️ Drawing Materials",
        "🧱 Sculpture Materials",
        "📱 Digital Art Tools",
        "🧪 Art Supplies",
        "📚 Art Books & Guides",
        "🎭 Mixed Media",
        "✂️ Crafting Materials",
        "📐 Measuring Tools",
        "🎪 Other"
    ]

    category = st.selectbox("Category*", category_options, index=0)
    image_file = st.file_uploader(
```

"Upload Material Image",

type=self.cfg.allowed_types,

help="Add a clear photo of your material to attract buyers"

)

description = st.text_area(

"Detailed Description*",

height=120,

placeholder="Describe the material in detail: brand, condition, what's included, why it's great for artists...",

help="Provide comprehensive information to help buyers make informed decisions"

)

submitted = st.form_submit_button("📩 List Material", use_container_width=True)

Renders an upload form in a two-column layout:

Left: name text input and price number input.

Right: category selectbox and an optional image_file uploader (allowed types from cfg).

description textarea below columns.

submitted becomes True when the user clicks "List Material".

The form uses clear_on_submit=False so fields don't auto-clear — this allows controlled behavior (they clear via rerun).

python

Copy

Edit

```
        if submitted:
            with st.status("Processing material upload...") as status:
                # Validation
                validation_errors = self._validate_form_data(name, price, category, image_file)

                if validation_errors:
                    for error in validation_errors:
                        st.error(f"❌ {error}")
                else:
                    # Process submission with detailed status updates
                    success = self._process_material_submission_with_status(
                        user_ctx, name, price, category, description, image_file, status
                    )

                    if success:
                        status.update(label="Material listed successfully!", state="complete")
                        st.success("✅ Material listed successfully!")
                        st.balloons()
                        st.rerun()

    st.markdown('</div>', unsafe_allow_html=True)
```

On submit:

A st.status context shows status updates to the user during processing.

First validate via _validate_form_data.

If validation passes, call _process_material_submission_with_status that performs file save, DB connection test, DB insert w/transaction and verification.

On success: update status, show success, confetti balloons, and rerun the app to show the new listing.

Form validation helper

python

Copy

Edit

```
  def _validate_form_data(self, name: str, price: float, category: str, image_file) -> list:
    """Comprehensive form validation"""
    errors = []

    if not name or len(name.strip()) < 3:
      errors.append("Material name must be at least 3 characters long")

    if price <= 0:
      errors.append("Price must be greater than 0")

    if category == "Select Category":
      errors.append("Please select a category")
```

return errors

Simple input validation returning a list of error messages:

name present and length >= 3,

price > 0,

category must not be default placeholder.

Robust submission processing

python

Copy

Edit

```
  def _process_material_submission_with_status(
      self, user_ctx: UserCtx, name: str, price: float, category: str,
      description: str, image_file, status_container
  ) -> bool:
      """Process material submission with detailed status updates and guaranteed database saving"""

      try:
          status_container.update(label="Validating data...")

          # Save uploaded file first
          image_path = None
```

```
if image_file:
    status_container.update(label="Saving image file...")
    image_path = self.files.save(image_file)
    if not image_path:
        st.error("❌ Failed to save image file")
        return False
    st.success(f"✅ Image saved: {os.path.basename(image_path)}")

status_container.update(label="Preparing material data...")

# Create material data
material = Material(
    id=0,  # Will be assigned by database
    name=name.strip(),
    price=price,
    category=category,
    description=description.strip(),
    artist=user_ctx.username,
    image=image_path
)

status_container.update(label="Testing database connection...")

# Test database connection explicitly
if not self.db.test_database_connection():
    st.error("❌ Database connection failed")
```

```
            return False

        st.success("✅ Database connection established")

        status_container.update(label="Saving material to database...")

        # Save to database with explicit transaction handling
        success = self.db.add(material)

        if success:
            status_container.update(label="Verifying data was saved...")
            # Verify the material was actually saved by fetching it back
            if self.db.verify_material_saved(user_ctx.username, name):
                st.success("✅ Material verified in database!")
                return True
            else:
                st.error("❌ Material was not properly saved to database")
                return False
        else:
            st.error("❌ Failed to save material to database")
            return False

    except Exception as e:
        logger.error(f"Exception in material submission: {e}")
        import traceback
        logger.error(f"Full traceback: {traceback.format_exc()}")
```

st.error(f"❌ Upload failed: {str(e)}")

return False

Full submission workflow (step-by-step):

Save the image first using self.files.save(image_file). If this fails → abort.

Build a Material object (id=0 placeholder).

Test DB connection explicitly with self.db.test_database_connection() — abort if test fails.

Save the material using self.db.add(material) which wraps transaction and verification. If add returns True:

Call verify_material_saved to be extra-sure the record persisted, and return True on verification.

Catch and log any unexpected exceptions and return False.

This sequence ensures the image is stored and the DB insertion is transactionally safe and verified before telling the user it's listed.

MAIN APPLICATION class

python

Copy

Edit

class MaterialsApplication:

"""Fixed main materials application - CORRECTED VERSION"""

```
    def __init__(self):
        self.cfg = UIConfig()
        self.files = FileManager(self.cfg)
        self.db = DatabaseManager()
        self._initialize_components()
```

The main app object that wires everything:

create UIConfig, FileManager, and DatabaseManager

_initialize_components constructs UI components.

python

Copy

Edit

```
    def _initialize_components(self) -> None:
        """Initialize all UI components"""
        self.style_manager = StyleManager(self.cfg, self.db, self.files)
        self.browse_tab = BrowseTab(self.cfg, self.db, self.files)
        self.your_materials_tab = YourMaterialsTab(self.cfg, self.db, self.files)
```

Instantiates the UI component objects with cfg, db, and files.

python

Copy

Edit

```
    def run(self) -> None:
```

```
    """Main application entry point with enhanced error handling and brown theme"""
    try:
        # Apply comprehensive brown styling
        self.style_manager.render(None)

        # Authentication check
        user_ctx = UserCtx.from_session()
        if not user_ctx:
            st.warning("🔐 Please login to access art materials marketplace.")
            st.stop()

        # Render header with brown theme
        self._render_header(user_ctx)

        # Render tabs based on user role
        self._render_tabs(user_ctx)

        # Footer with brown theme
        self._render_footer()

    except Exception as e:
        logger.error(f"Application error: {e}")
        st.error(f"❌ Application error: {e}")
```

run is the app lifecycle:

Apply theme via StyleManager.

Check authentication; if not logged in, warn and st.stop() (halts execution).

Render header → tabs (browse or artist tabs depending on role) → footer.

Wraps all in a try/except to surface app-level errors.

Header, tabs & footer helpers

python

Copy

Edit

```
  def _render_header(self, user_ctx: UserCtx) -> None:
    """Render page header with brown theme"""
    # Header container
    st.markdown(f'<h1 class="main-title">{self.cfg.page_title}</h1>',
unsafe_allow_html=True)
    st.markdown('<div class="title-divider"></div>', unsafe_allow_html=True)

    col1, col2 = st.columns([3, 1])
    with col1:
      st.markdown('<p style="color: var(--secondary); font-size: 1.1rem; font-weight:
600;">Discover and share quality art supplies from fellow artists</p>',
            unsafe_allow_html=True)
    with col2:
      role_class = f"role-{user_ctx.role.value}"
      st.markdown(f'<span class="role-badge
{role_class}">{user_ctx.role.value.title()}</span>',
```

```
            unsafe_allow_html=True)


    st.markdown('</div>', unsafe_allow_html=True)
```

Renders page title and a subtitle and a role badge (artist/customer) in a small two-column layout. Visual styling depends on the CSS injected earlier.

python

Copy

Edit

```
  def _render_tabs(self, user_ctx: UserCtx) -> None:
    """Render tab interface based on user role with brown theme"""
    if user_ctx.role == UserRole.ARTIST:
      # Artists see both tabs
      tab1, tab2 = st.tabs([" 🔍 Browse Materials", " 📦 Your Materials"])

      with tab1:
        self.browse_tab.render(user_ctx)

      with tab2:
        self.your_materials_tab.render(user_ctx)
    else:
      # Customers only see Browse Materials
      st.markdown('<h3style="color: var(--primary);"> 🔍 Browse Art Materials</h3>',
            unsafe_allow_html=True)
      self.browse_tab.render(user_ctx)
```

If the user is an ARTIST: show two tabs (Browse, Your Materials). If a CUSTOMER: show only Browse. (Small note: in the original file the h3 markup is correct — ensure spacing is proper in your code).

python

Copy

Edit

```
    def _render_footer(self) -> None:
        """Render footer with brown theme"""
        st.markdown('<div class="brown-divider"></div>', unsafe_allow_html=True)
        st.markdown("""
        <div class="brown-card" style="text-align: center; padding: 2rem; margin-top: 2rem;">
            <h3 style="color: var(--accent);">🎨 Brush and Soul Materials</h3>
            <p style="color: var(--secondary); font-weight: 600; font-size: 1.1rem;">Buy • Sell • Create • Inspire</p>
            <p style="font-size: 0.95rem; color: var(--brown-darker); font-weight: 500;">
                Connecting artists with quality materials for creative excellence
            </p>
            <div style="margin-top: 1.5rem; padding: 1rem; background: linear-gradient(135deg, rgba(139, 69, 19, 0.1), rgba(210, 180, 140, 0.1)); border-radius: 8px; border: 1px solid var(--brown-light);">
                <p style="color: var(--primary); font-weight: 600; margin: 0;">
                    🌟 Quality materials, fair prices, artist community 🌟
                </p>
            </div>
        </div>
        """, unsafe_allow_html=True)
```

Renders a styled footer card with brand messaging.

Entrypoint — page configuration and app boot

python

Copy

Edit

```
def main() -> None:
    """Application main function with page configuration"""
    st.set_page_config(
        page_title="Brush and Soul - Art Materials",
        page_icon=" 🎨 ",
        layout="wide",
        initial_sidebar_state="collapsed"
    )

    try:
        app = MaterialsApplication()
        app.run()
    except Exception as e:
        logger.error(f"Main application error: {e}")
        st.error(" ❌ Application failed to load. Please try again.")
```

main() configures Streamlit page metadata: title, icon, layout, initial sidebar state.

Then it creates MaterialsApplication() and calls .run(), wrapped in try/except to catch and show errors.

python

Copy

Edit

```
if __name__ == "__main__":
   main()
```

Standard Python idiom: if script is executed directly, call main().

--TUTORIALS CODE

```
import streamlit as st
```

Imports Streamlit as st.

Used to build the interactive web UI.

python

Copy

Edit

```
from dataclasses import dataclass
```

Imports dataclass decorator → makes it easy to define simple classes without writing __init__.

python

Copy

Edit

```
from typing import List, Dict, Optional, Any
```

List → list type hint.

Dict → dictionary type hint.

Optional → means value can be None.

Any → any type (fallback).

python

Copy

Edit

import datetime

Provides datetime.datetime class for handling timestamps.

python

Copy

Edit

import logging

Python’s built-in logging library for recording info/errors.

python

Copy

Edit

logging.basicConfig(level=logging.INFO)

Sets up global logging.

Only messages with severity INFO or higher are shown.

python

Copy

Edit

logger = logging.getLogger(__name__)

Creates a logger named after the current file/module.

Used for logging messages inside this script.

2) Tutorial Data Class

python

Copy

Edit

@dataclass

class TutorialData:

Defines a dataclass → a lightweight way to store tutorial info.

python

Copy

Edit

  id: Optional[int]

Tutorial ID (int) or None if not set yet.

python

Copy

Edit

  title: str

The tutorial’s title (string).

python

Copy

Edit

content: str

The tutorial’s text/content.

python

Copy

Edit

author: str

Name of the user who wrote it.

python

Copy

Edit

created_at: datetime.datetime

A timestamp of when the tutorial was created.

Method to Convert to Dictionary

python

Copy

Edit

def to_dict(self) -> Dict[str, Any]:

Converts the tutorial object into a dictionary.

python

Copy

Edit

```
    return {
      "id": self.id,
      "title": self.title,
      "content": self.content,
      "author": self.author,
      "created_at": self.created_at.isoformat()
    }
```

Returns a dict with all fields.

Converts datetime into ISO string ("2025-08-17T10:35:00").

3) Tutorial Manager Class

python

Copy

Edit

class TutorialManager:

Handles storage and management of all tutorials.

python

Copy

Edit

```
  def __init__(self):
```

Constructor runs when you create TutorialManager().

python

Copy

Edit

```
    self.tutorials: Dict[int, TutorialData] = {}
```

Dictionary that stores tutorials in memory.

Key → tutorial ID, Value → TutorialData.

python

Copy

Edit

```
    self.next_id = 1
```

Keeps track of the next tutorial ID to assign.

Starts at 1.

Save Tutorial

python

Copy

Edit

```
  def save_tutorial(self, tutorial_data: Dict[str, Any]) -> Optional[int]:
```

Saves a new tutorial.

Accepts a dictionary with tutorial fields.

python

Copy

Edit

```
    try:
```

Start of try/except block (for error handling).

python

Copy

Edit

```
        tutorial = TutorialData(
            id=self.next_id,
            title=tutorial_data["title"],
            content=tutorial_data["content"],
            author=tutorial_data["author"],
            created_at=datetime.datetime.now()
        )
```

Creates a TutorialData object.

id → assign current next_id.

title/content/author → from given dictionary.

created_at → current timestamp.

python

Copy

Edit

```
        self.tutorials[self.next_id] = tutorial
```

Saves the tutorial inside dictionary with its ID as key.

python

Copy

Edit

```
        self.next_id += 1
```

Increments ID counter for next tutorial.

python

Copy

Edit

```
        logger.info(f"Tutorial saved: {tutorial.title} by {tutorial.author}")
```

Logs an info message showing which tutorial was saved.

python

Copy

Edit

```
        return tutorial.id
```

Returns the ID of the saved tutorial.

python

Copy

Edit

```
        except Exception as e:
            logger.error(f"Error saving tutorial: {e}")
            return None
```

If any error happens, log it and return None.

Get Tutorials by User

python

Copy

Edit

```
    def get_user_tutorials(self, username: str) -> List[Dict[str, Any]]:
```

Gets all tutorials written by a specific user.

python

Copy

Edit

```
        return [tut.to_dict() for tut in self.tutorials.values() if tut.author == username]
```

Loops through all tutorials.

Filters only those where author == username.

Converts each tutorial to dictionary.

Get All Tutorials

python

Copy

Edit

def get_all_tutorials(self) -> List[Dict[str, Any]]:

Gets all tutorials (for everyone).

python

Copy

Edit

    return [tut.to_dict() for tut in self.tutorials.values()]

Converts every tutorial in memory to a dictionary.

Update Tutorial

python

Copy

Edit

def update_tutorial(self, tutorial_id: int, updates: Dict[str, Any]) -> bool:

Updates a tutorial with new values.

python

Copy

Edit

    if tutorial_id not in self.tutorials:

        return False

If tutorial ID not found → return False.

python

Copy

Edit

tutorial = self.tutorials[tutorial_id]

Get the tutorial object by ID.

python

Copy

Edit

for key, value in updates.items():

if hasattr(tutorial, key):

setattr(tutorial, key, value)

For each field in updates dictionary:

If the tutorial has that attribute → update it.

python

Copy

Edit

logger.info(f"Tutorial updated: {tutorial.title}")

Log update success.

python

Copy

Edit

return True

Return True after successful update.

Delete Tutorial

python

Copy

Edit

```
   def delete_tutorial(self, tutorial_id: int) -> bool:
```

Deletes a tutorial by ID.

python

Copy

Edit

```
      if tutorial_id in self.tutorials:
```

Check if tutorial exists.

python

Copy

Edit

```
         removed = self.tutorials.pop(tutorial_id)
```

Remove tutorial from dictionary.

python

Copy

Edit

```
         logger.info(f"Tutorial deleted: {removed.title}")
```

Log which tutorial was deleted.

python

Copy

Edit

return True

Return True for success.

python

Copy

Edit

return False

Return False if ID not found.

4) Tutorial Form Component (Streamlit UI)

python

Copy

Edit

class TutorialFormComponent:

def __init__(self, manager: TutorialManager):

self.manager = manager

A UI component for creating new tutorials.

Needs TutorialManager to save tutorials.

python

Copy

Edit

def render(self, user: str):

Render the form for the given user.

python

Copy

Edit

```
    st.subheader(" 📘 Create a New Tutorial")
```

Shows subheader in Streamlit.

python

Copy

Edit

```
    with st.form("tutorial_form"):
```

Creates a Streamlit form (so inputs + button are grouped).

python

Copy

Edit

```
        title = st.text_input("Title")
```

Input field for title.

python

Copy

Edit

```
        content = st.text_area("Content")
```

Large text area for tutorial content.

python

Copy

Edit

```
        submitted = st.form_submit_button("Publish Tutorial")
```

Submit button → returns True when clicked.

python

Copy

Edit

```
        if submitted and title and content:
```

Check if form was submitted and inputs are not empty.

python

Copy

Edit

```
            tutorial_data = {
                "title": title,
                "content": content,
                "author": user
            }
```

Build dictionary of tutorial info.

python

Copy

Edit

```
            self.manager.save_tutorial(tutorial_data)
```

Save tutorial using manager.

python

Copy

Edit

st.success("Tutorial published successfully!")

Show success message in app.

5) Tutorial List Component

python

Copy

Edit

class TutorialListComponent:

def __init__(self, manager: TutorialManager):

self.manager = manager

Component for displaying tutorials.

python

Copy

Edit

def render(self, tutorials: List[Dict[str, Any]]):

Render a list of tutorials passed as dictionaries.

python

Copy

Edit

st.subheader("📚 Tutorials")

Display heading.

python

Copy

Edit

```
    for tut in tutorials:
```

Loop through each tutorial.

python

Copy

Edit

```
        with st.expander(tut["title"], expanded=False):
```

Create a collapsible expander with tutorial title.

python

Copy

Edit

```
            st.write(tut["content"])
```

Show the tutorial content inside expander.

python

Copy

Edit

```
            st.caption(f"By {tut['author']} on {tut['created_at']}")
```

Show author + date below content.

6) Tutorial App Wrapper

python

Copy

Edit

class TutorialApp:

  def __init__(self):

    self.manager = TutorialManager()

    self.form = TutorialFormComponent(self.manager)

    self.list = TutorialListComponent(self.manager)

Main app class.

Creates one TutorialManager.

Passes it to form and list components.

python

Copy

Edit

  def run(self, current_user: Optional[str] = None):

Main function to run the tutorial app.

python

Copy

Edit

    st.title("🎓 Tutorials Section")

Display page title.

python

Copy

Edit

```
    if current_user:
        self.form.render(current_user)
```

If a user is logged in → show the form so they can add tutorials.

python

Copy

Edit

```
    tutorials = self.manager.get_all_tutorials()
```

Fetch all tutorials.

python

Copy

Edit

```
    self.list.render(tutorials)
```

Display tutorials using the list component.

7) Entry Point

python

Copy

Edit

def main():

Defines main function (entry point).

python

Copy

Edit

```
    app = TutorialApp()
```

Create the app instance.

python

Copy

Edit

```
    # Simulate a logged-in user
    current_user = "Bob"
```

Hardcodes a logged-in user named "Bob".

python

Copy

Edit

```
    app.run(current_user)
```

Runs the app for "Bob".

python

Copy

Edit

```
if __name__ == "__main__":
    main()
```

If file is run directly → execute main().

---------PORTFOLIO CODE

Imports & Setup

import streamlit as st

- Imports **Streamlit** as st (for building the web UI).

from dataclasses import dataclass

- Imports dataclass decorator → allows easy creation of data container classes.

from typing import List, Dict, Optional, Any

- Type hinting:
    - List → a list of values.
    - Dict → a dictionary.
    - Optional → means a variable can also be None.
    - Any → any data type.

import datetime

- Provides **date and time handling** (used for created_at timestamps).

import logging

- Provides **logging system** to track info and errors.

logging.basicConfig(level=logging.INFO)

- Configures logging → only messages INFO and above are shown.

logger = logging.getLogger(__name__)

- Creates a logger for this module.

---

2) Portfolio Data Class

@dataclass

class PortfolioData:

  id: Optional[int]

title: str

description: str

author: str

created_at: datetime.datetime

- Defines a **dataclass** to store a portfolio entry.
- Fields:
    - id → numeric ID (can be None).
    - title → title of the portfolio project.
    - description → details about the project.
    - author → person who created it.
    - created_at → timestamp of creation.

Convert to Dictionary

```
def to_dict(self) -> Dict[str, Any]:
    return {
        "id": self.id,
        "title": self.title,
        "description": self.description,
        "author": self.author,
        "created_at": self.created_at.isoformat()
    }
```

- Converts the object into a dictionary.
- Converts created_at (datetime) into a string in ISO format (2025-08-17T12:45:00).

---

3) Portfolio Manager

class PortfolioManager:

- Manages portfolio entries (create, update, delete, list).

def __init__(self):

self.portfolios: Dict[int, PortfolioData] = {}

self.next_id = 1

- portfolios → dictionary mapping ID → PortfolioData.
- next_id → keeps track of the next available portfolio ID (starts at 1).

Save Portfolio

def save_portfolio(self, portfolio_data: Dict[str, Any]) -> Optional[int]:

- Saves a new portfolio entry.
- Input: dictionary with title, description, author.

try:

portfolio = PortfolioData(

id=self.next_id,

title=portfolio_data["title"],

description=portfolio_data["description"],

author=portfolio_data["author"],

created_at=datetime.datetime.now()

)

- Creates a new PortfolioData object.
- Assigns current next_id.
- Sets created_at to now.

self.portfolios[self.next_id] = portfolio

- Stores the portfolio in dictionary with key = ID.

self.next_id += 1

- Increments ID counter for the next portfolio.

logger.info(f"Portfolio saved: {portfolio.title} by {portfolio.author}")

- Logs portfolio saved message.

return portfolio.id

- Returns ID of saved portfolio.

except Exception as e:

logger.error(f"Error saving portfolio: {e}")

return None

- If an error occurs → log it, return None.

Get Portfolios by User

def get_user_portfolios(self, username: str) -> List[Dict[str, Any]]:

return [port.to_dict() for port in self.portfolios.values() if port.author == username]

- Returns a list of portfolios authored by the given username.
- Converts each to dictionary.

Get All Portfolios

def get_all_portfolios(self) -> List[Dict[str, Any]]:

return [port.to_dict() for port in self.portfolios.values()]

- Returns all portfolios (converted to dictionaries).

Update Portfolio

def update_portfolio(self, portfolio_id: int, updates: Dict[str, Any]) -> bool:

- Updates fields of a portfolio by ID.

if portfolio_id not in self.portfolios:

return False

- If ID does not exist → return False.

portfolio = self.portfolios[portfolio_id]

- Retrieve the portfolio object.

for key, value in updates.items():

if hasattr(portfolio, key):

setattr(portfolio, key, value)

- Loop through updates.
- If attribute exists on portfolio → update it dynamically.

logger.info(f"Portfolio updated: {portfolio.title}")

- Log portfolio update.

return True

- Return success.

Delete Portfolio

def delete_portfolio(self, portfolio_id: int) -> bool:

- Deletes a portfolio by ID.

if portfolio_id in self.portfolios:

removed = self.portfolios.pop(portfolio_id)

logger.info(f"Portfolio deleted: {removed.title}")

return True

return False

- If ID exists → remove it from dictionary.
- Log which portfolio was deleted.
- Return True.
- Otherwise → return False.

---

4) Portfolio Form Component (Streamlit UI)

```
class PortfolioFormComponent:
  def __init__(self, manager: PortfolioManager):
    self.manager = manager
```

- Handles the form UI for adding portfolios.

```
  def render(self, user: str):
    st.subheader(" 🖼️ Add a New Portfolio Project")
```

- Displays subheader.

```
    with st.form("portfolio_form"):
      title = st.text_input("Project Title")
      description = st.text_area("Project Description")
      submitted = st.form_submit_button("Add Portfolio")
```

- Creates a form with inputs:
  - title → text input.
  - description → textarea.
  - Submit button.

```
    if submitted and title and description:
```

- If form is submitted and both fields are filled:

```
      portfolio_data = {
        "title": title,
        "description": description,
        "author": user
      }
```

- Build dictionary with portfolio info.

```
      self.manager.save_portfolio(portfolio_data)
```

- Save portfolio using manager.

```
      st.success("Portfolio project added successfully!")
```

- Show success message in app.

---

5) Portfolio List Component

```
class PortfolioListComponent:
  def __init__(self, manager: PortfolioManager):
    self.manager = manager
```

- Component to display list of portfolios.

```
  def render(self, portfolios: List[Dict[str, Any]]):
    st.subheader("📂 Portfolio Projects")
```

- Displays section subheader.

```
    for port in portfolios:
      with st.expander(port["title"], expanded=False):
        st.write(port["description"])
        st.caption(f"By {port['author']} on {port['created_at']}")
```

- For each portfolio:
  - Creates a collapsible expander with project title.
  - Inside shows project description.
  - Adds caption with author + creation date.

---

6) Main Portfolio App

```
class PortfolioApp:
  def __init__(self):
    self.manager = PortfolioManager()
    self.form = PortfolioFormComponent(self.manager)
    self.list = PortfolioListComponent(self.manager)
```

- Combines manager + form + list components.

```
def run(self, current_user: Optional[str] = None):
    st.title("💼 Portfolio Section")
```

- Displays main title.

```
    if current_user:
        self.form.render(current_user)
```

- If user logged in → show form to add portfolio.

```
    portfolios = self.manager.get_all_portfolios()
```

- Fetch all portfolios.

```
    self.list.render(portfolios)
```

- Display list of portfolios.

---

7) Entry Point

```
def main():
    app = PortfolioApp()
```

- Creates the portfolio app.

```
    # Simulate a logged-in user
    current_user = "Charlie"
```

- Simulates a logged-in user "Charlie".

```
    app.run(current_user)
```

- Runs the app with Charlie logged in.

```
if __name__ == "__main__":
    main()
```

- Standard Python entry point.
- Runs main() only when file is executed directly.

---CART CODE

Imports & Setup

import streamlit as st

- Imports **Streamlit** (as st) → used to create the cart UI.

from dataclasses import dataclass

- Imports dataclass decorator → lets us create simple data holder classes without writing boilerplate.

from typing import List, Dict, Optional, Any

- Type hints:
    - List → list type.
    - Dict → dictionary type.
    - Optional → means the value can be None.
    - Any → any type.

import datetime

- Provides date and time support (for adding timestamp to cart items).

import logging

- For logging events (like adding/removing cart items).

logging.basicConfig(level=logging.INFO)

- Configures logging → logs of INFO level or higher will be shown.

logger = logging.getLogger(__name__)

- Creates a logger for this file/module.

---

2) Cart Item Data Class

@dataclass

class CartItem:

id: Optional[int]

name: str

price: float

quantity: int

added_at: datetime.datetime

- Defines a **dataclass** representing one cart item.
- Fields:
    - id → numeric ID of the cart item (can be None).
    - name → name of the product.
    - price → price of the product.
    - quantity → how many units added.
    - added_at → timestamp when added.

Convert CartItem to Dictionary

```
def to_dict(self) -> Dict[str, Any]:
    return {
        "id": self.id,
        "name": self.name,
        "price": self.price,
        "quantity": self.quantity,
        "added_at": self.added_at.isoformat()
    }
```

- Converts a CartItem object into a dictionary.
- Converts timestamp into ISO format string (like 2025-08-17T12:55:00).

---

3) Cart Manager Class

class CartManager:

- Manages cart operations (add, update, delete, view).

```
def __init__(self):
    self.cart: Dict[int, CartItem] = {}
    self.next_id = 1
```

- Initializes an empty cart (dictionary of id → CartItem).
- Starts ID counter at 1.

Add Item to Cart

```
def add_item(self, item_data: Dict[str, Any]) -> Optional[int]:
```

- Method to add an item to cart.
- Takes a dictionary (name, price, quantity).

```
    try:
        item = CartItem(
            id=self.next_id,
            name=item_data["name"],
            price=item_data["price"],
            quantity=item_data["quantity"],
            added_at=datetime.datetime.now()
        )
```

- Creates a new CartItem object with provided details.
- Sets id to current next_id.
- Sets added_at to now.

```
        self.cart[self.next_id] = item
```

- Adds item to the cart dictionary.

```
        self.next_id += 1
```

- Increments ID counter for next item.

logger.info(f"Item added to cart: {item.name} x{item.quantity}")

- Logs that the item was added.

return item.id

- Returns the ID of the added item.

except Exception as e:

logger.error(f"Error adding item to cart: {e}")

return None

- If an error occurs → log it and return None.

Get All Items in Cart

def get_cart_items(self) -> List[Dict[str, Any]]:

return [item.to_dict() for item in self.cart.values()]

- Returns a list of all cart items.
- Converts each to dictionary for display.

Update Item Quantity

def update_item_quantity(self, item_id: int, new_quantity: int) -> bool:

- Updates quantity of a specific item.

if item_id not in self.cart:

return False

- If item not found → return False.

self.cart[item_id].quantity = new_quantity

- Updates the quantity.

logger.info(f"Updated quantity for {self.cart[item_id].name} to {new_quantity}")

- Logs update.

return True

- Returns True after success.

Remove Item from Cart

def remove_item(self, item_id: int) -> bool:

- Removes an item by ID.

if item_id in self.cart:

removed = self.cart.pop(item_id)

logger.info(f"Removed from cart: {removed.name}")

return True

return False

- If ID exists → remove item from dictionary, log removal, return True.
- Otherwise → return False.

Calculate Total Price

def calculate_total(self) -> float:

return sum(item.price * item.quantity for item in self.cart.values())

- Loops through items in cart.
- Multiplies price × quantity for each item.
- Returns total cost.

---

4) Cart UI Component (Streamlit)

class CartComponent:

def __init__(self, manager: CartManager):

self.manager = manager

- UI component to show and interact with cart.

- Needs CartManager to manage items.

Render Cart in Streamlit

```
def render(self):
    st.subheader("🛒 Shopping Cart")
```

- Displays section header.

```
    cart_items = self.manager.get_cart_items()
```

- Fetch all items from cart manager.

```
    if not cart_items:
        st.info("Your cart is empty.")
        return
```

- If no items → show message and stop.

```
    for item in cart_items:
        col1, col2, col3, col4 = st.columns([3, 2, 2, 2])
```

- Creates 4 columns layout.
- Used for name, quantity, price, and remove button.

```
        with col1:
            st.write(f"**{item['name']}**")
```

- Shows item name in bold.

```
        with col2:
            new_qty = st.number_input(
                f"Qty for {item['name']}",
                min_value=1,
                value=item['quantity'],
                key=f"qty_{item['id']}"
            )
```

- Shows a number input to change quantity.
- Default value = current quantity.
- Each item gets a unique key.

```
if new_qty != item['quantity']:
    self.manager.update_item_quantity(item['id'], new_qty)
    st.experimental_rerun()
```

- If quantity was changed:
    - Update item.
    - Refresh UI with rerun.

```
with col3:
    st.write(f"${item['price']:.2f}")
```

- Displays price formatted to 2 decimals.

```
with col4:
    if st.button("Remove", key=f"remove_{item['id']}"):
        self.manager.remove_item(item['id'])
        st.experimental_rerun()
```

- Shows “Remove” button.
- On click → removes item, refreshes UI.

```
st.markdown("---")
```

- Horizontal line for separation.

```
total = self.manager.calculate_total()
st.subheader(f"Total: ${total:.2f}")
```

- Calculates total cost of cart.
- Displays total with 2 decimals.

---

5) Cart App Wrapper

```
class CartApp:
  def __init__(self):
    self.manager = CartManager()
    self.component = CartComponent(self.manager)
```

- Main app class.
- Creates one CartManager and one UI CartComponent.

```
  def run(self):
    st.title("🛍️ Cart Section")
```

- Displays page title.

```
    st.sidebar.header("Add Items")
```

- Sidebar header for adding items.

```
    with st.sidebar.form("add_item_form"):
      name = st.text_input("Item Name")
      price = st.number_input("Price", min_value=0.0, step=0.01)
      quantity = st.number_input("Quantity", min_value=1, value=1)
      submitted = st.form_submit_button("Add to Cart")
```

- Sidebar form fields:
  - name → product name.
  - price → numeric price.
  - quantity → numeric quantity (default = 1).
  - Submit button.

```
      if submitted and name and price > 0:
        item_data = {"name": name, "price": price, "quantity": quantity}
        self.manager.add_item(item_data)
        st.success(f"Added {name} to cart")
        st.experimental_rerun()
```

- If form submitted with valid input:
    - Build item dictionary.
    - Add item to cart.
    - Show success message.
    - Refresh UI.

  self.component.render()

- Renders main cart UI in page.

---

6) Entry Point

def main():

  app = CartApp()

  app.run()

- Defines main function → creates app and runs it.

if __name__ == "__main__":

  main()

- Runs main() only when script is executed directly.

----------LOGIN CODE

from __future__ import annotations

Enables postponed evaluation of annotations (type hints are stored as strings). This avoids forward-reference issues and can reduce import-time overhead.

python

Copy

Edit

import base64

Imports the base64 module used to encode binary data (images) into text (base64 strings). Used later to embed the background image into CSS/HTML.

python

Copy

Edit

import logging

Imports Python’s logging facility for recording info/warnings/errors during runtime.

python

Copy

Edit

from contextlib import contextmanager

Imports the contextmanager decorator to build simple context managers with with blocks. Used later to handle background image load errors.

python

Copy

Edit

from dataclasses import dataclass

Imports @dataclass decorator to create lightweight classes for holding data (auto-generates __init__, __repr__, etc.).

python

Copy

Edit

from enum import Enum

Imports Enum to define named constant sets (used for view states in the login page).

python

Copy

Edit

from pathlib import Path

Imports Path for object-oriented filesystem path manipulations (better than string-based os.path).

python

Copy

Edit

from typing import Dict, Optional, Tuple

Imports typing helpers:

Dict → typed dictionary,

Optional → value or None,

Tuple → fixed-length tuple type.

python

Copy

Edit

import streamlit as st

Imports Streamlit (UI framework) and aliases it to st so the code can call st.* UI functions.

python

Copy

Edit

Configure logging

logger = logging.getLogger(__name__)

Retrieves a logger instance named after the module (value of __name__). The file uses logger.info(), logger.error(), etc. to write messages. (Note: no basicConfig call here — it relies on app/global logging config.)

Configuration and small value objects

python

Copy

Edit

class LoginView(Enum):

  """Available login page views"""

  LOGIN = "login"

  RESET = "reset"

  REGISTERED_SUCCESS = "registered_success"

Declares an Enum named LoginView that enumerates the possible page modes:

LOGIN → normal login form,

RESET → password reset form,

REGISTERED_SUCCESS → view shown when registration has succeeded.

Using an enum ensures code refers to views by name, not raw strings (reduces typos).

python

Copy

Edit

```
@dataclass(frozen=True)
class UIConfig:
    """UI configuration settings"""
    page_title: str = "Login - Brush and Soul"
    layout: str = "centered"
    background_image: str = r"D:\Brush and soul\uploads\53b691e2-6023-4fc8-9c87-cc378544d4d8.jpg"
    max_container_width: int = 400
    blur_intensity: int = 15
```

Defines an immutable (frozen=True) dataclass UIConfig that holds UI configuration values:

page_title → title shown in browser/tab,

layout → Streamlit layout setting (here "centered"),

background_image → path to local background image file (raw string r"..." used to avoid escaping),

max_container_width → used in CSS to limit the inner content width,

blur_intensity → glassmorphism blur amount.

Because it's frozen, once an instance is created you cannot mutate fields — good for config constants.

python

Copy

Edit

@dataclass

class LoginResult:

"""Result of login attempt"""

success: bool

user_data: Optional[Dict] = None

error_message: Optional[str] = None

Defines LoginResult dataclass used to return structured results from authentication:

success → True/False,

user_data → optional dict with user info when success,

error_message → optional text on failure.

Database integration (authentication manager)

python

Copy

Edit

class AuthenticationManager:

"""Handles authentication operations with database backend"""

def __init__(self):

self._import_utils()

AuthenticationManager encapsulates database-backed authentication operations.

In __init__, it immediately attempts to import required database helper functions by calling _import_utils().

python

Copy

Edit

def _import_utils(self) -> None:

"""Import database utilities with error handling"""

try:

from utils import authenticate, update_password, is_valid_password

self.authenticate = authenticate

self.update_password = update_password

self.is_valid_password = is_valid_password

logger.info("Database utilities imported successfully")

except ImportError as e:

logger.error(f"Failed to import utilities: {e}")

st.error("Database connection error. Please check system configuration.")

raise

_import_utils():

Tries to import three helper functions from a utils module:

authenticate(username, password) → authenticate credentials and return user data,

update_password(email, new_password) → update password record,

is_valid_password(password) → validate password strength/requirements.

If imported, assigns them as instance attributes for later use.

On success logs an info message.

On ImportError (utils missing or not found), logs error, shows st.error to the user, and re-raises the exception (so initialization fails fast).

python

Copy

Edit

```
  def login_user(self, username: str, password: str) -> LoginResult:
    """Authenticate user credentials against database"""
    try:
      user_data = self.authenticate(username.strip(), password.strip())
      if user_data:
        logger.info(f"Successful login for user: {username}")
        return LoginResult(success=True, user_data=user_data)
      else:
```

logger.warning(f"Failed login attempt for user: {username}")

return LoginResult(success=False, error_message="Invalid username or password.")

except Exception as e:

logger.error(f"Login error for user {username}: {e}")

return LoginResult(success=False, error_message="Authentication service unavailable.")

login_user(...):

Strips whitespace from inputs and calls self.authenticate(...).

If authenticate returns truthy user_data, it logs and returns LoginResult(success=True, user_data=...).

If no user_data (invalid credentials), logs a warning and returns a failure LoginResult with message.

If any exception occurs during authenticate, logs error and returns a failure LoginResult with service-unavailable message.

python

Copy

Edit

```
def reset_user_password(self, email: str, new_password: str) -> Tuple[bool, str]:
    """Reset user password in database"""
    try:
        # Validate password strength
```

```
        is_valid, validation_message = self.is_valid_password(new_password)
        if not is_valid:
            return False, validation_message

        # Update password in database
        if self.update_password(email.strip(), new_password.strip()):
            logger.info(f"Password reset successful for email: {email}")
            return True, "Password reset successful!"
        else:
            logger.warning(f"Password reset failed - email not found: {email}")
            return False, "Email not found in system."
    except Exception as e:
        logger.error(f"Password reset error for email {email}: {e}")
        return False, "Password reset service unavailable."
reset_user_password(email, new_password):
```

Validates the new_password via self.is_valid_password, which is expected to return (bool, message).

If invalid, returns (False, validation_message).

Calls update_password(email, new_password) to persist the change; returns (True, "Password reset successful!") on success.

If update returns False (email not found), returns (False, "Email not found...").

Catches exceptions, logs error, and returns (False, "Password reset service unavailable.").

Session management

python

Copy

Edit

```
class SessionManager:
  """Advanced session state management"""

  @staticmethod
  def initialize_session() -> None:
    """Initialize session state variables"""
    session_defaults = {
      "logged_in": False,
      "user": None,
      "show_welcome": True
    }

    for key, default_value in session_defaults.items():
      if key not in st.session_state:
        st.session_state[key] = default_value
```

SessionManager provides helpers for Streamlit session state.

initialize_session():

Builds a dict session_defaults with keys and default values:

"logged_in": False → user initially not logged in,

"user": None → user data placeholder,

"show_welcome": True → optional UI flag.

Iterates defaults and sets st.session_state[key] only if not already present. This ensures reruns don’t overwrite existing session values.

python

Copy

Edit

```
  @staticmethod
  def login_user(user_data: Dict) -> None:
    """Set user as logged in"""
    st.session_state.logged_in = True
    st.session_state.user = user_data
    logger.info(f"Session established for user: {user_data.get('username', 'Unknown')}")
login_user(user_data):
```

Stores user_data into session state and marks logged_in True.

Logs a message indicating which user has been established in session.

python

Copy

Edit

```
  @staticmethod
  def logout_user() -> None:
    """Clear user session"""
    st.session_state.logged_in = False
    st.session_state.user = None
    logger.info("User session cleared")
```

logout_user():

Clears user state and sets logged_in False, then logs out.

UI: background handling / styling

python

Copy

Edit

```
class BackgroundManager:
  """Handles background image and styling"""

  def __init__(self, config: UIConfig):
    self.config = config
```

BackgroundManager manages the background image encoding and CSS injection for a glassmorphism style.

__init__ stores supplied UI config.

python

Copy

Edit

```
  @contextmanager
  def error_handling(self):
    """Context manager for error handling"""
    try:
      yield
    except Exception as e:
      logger.error(f"Background processing error: {e}")
      st.warning("Background image could not be loaded")
error_handling():
```

A context manager that yields to the block and catches exceptions; on exception it logs and shows a Streamlit warning.

Used to make background loading tolerant: if something goes wrong (file missing etc.) user still sees the app.

python

Copy

Edit

```
  def load_background_image(self) -> Optional[str]:
    """Load and encode background image"""
    with self.error_handling():
      image_path = Path(self.config.background_image)
      if not image_path.exists():
        logger.warning(f"Background image not found: {image_path}")
```

```
            return None

        with open(image_path, "rb") as image_file:
            encoded = base64.b64encode(image_file.read()).decode()
            return f"{encoded}"
    return None
load_background_image():
```

Wraps operations with error_handling() so errors are handled gracefully.

Converts the configured background_image path to a Path.

If the path does not exist: logs a warning and returns None.

If the file exists: opens it in binary mode, reads bytes, base64-encodes them and returns a data: URI string (suitable for embedding directly into CSS background-image: url(...)).

The final return None outside the with is unreachable unless contextmanager yields and an exception occurs; kept for clarity.

python

Copy

Edit

```
  def apply_styling(self) -> None:
    """Apply glassmorphism styling with background"""
    background_data = self.load_background_image()
```

```
    background_style = (
      f'background-image: url("{background_data}");' if background_data
      else 'background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);'
    )
apply_styling():
```

Calls load_background_image() to get either the data: URI or None.

Builds background_style string:

If background_data present: uses background-image: url("data:...").

Otherwise fallback to a CSS gradient background.

python

Copy

Edit

```
    st.markdown(f"""
      <style>
      .stApp {{
        {background_style}
        background-size: cover;
        background-position: center;
        background-repeat: no-repeat;
        background-attachment: fixed;
      }}
```

```
.block-container {{
    background: rgba(255, 255, 255, 0.07);
    backdrop-filter: blur({self.config.blur_intensity}px);
    -webkit-backdrop-filter: blur({self.config.blur_intensity}px);
    border-radius: 20px;
    margin: auto;
    width: 100%;
    max-width: {self.config.max_container_width}px;
    padding: 2rem;
    box-shadow: 0 8px 32px rgba(0, 0, 0, 0.25);
}}
section.main > div {{
    padding: 0rem !important;
}}
.stTextInput > div > div > input {{
    background-color: rgba(255,255,255,0.2) !important;
    color: #000 !important;
    border-radius: 10px;
}}
h1, h2, h3, h4, h5, h6 {{
    color: #000 !important;
}}
p, label {{
    color: #000 !important;
}}
button {{
```

```
        background-color: #4CAF50 !important;
        color: white !important;
        border-radius: 10px;
    }}
    a {{
        color: #00f !important;
        text-decoration: underline;
    }}
    </style>
""", unsafe_allow_html=True)
```

Injects a large CSS <style> block into the app via st.markdown(..., unsafe_allow_html=True):

Styles .stApp (root app) with the background_style computed earlier and makes it cover the viewport.

Adds .block-container style which implements a semi-transparent white container with blur (glassmorphism), fixed max-width, padding and shadow.

Tweaks default Streamlit element styles: resets default padding, sets input background color and radius, ensures headings and paragraphs are dark for readability, styles buttons, links, etc.

unsafe_allow_html=True allows direct HTML/CSS injection; used here to style the whole page.

Login interface controller

python

Copy

Edit

```
class LoginInterface:
   """Main login interface controller"""

   def __init__(self):
      self.config = UIConfig()
      self.auth_manager = AuthenticationManager()
      self.session_manager = SessionManager()
      self.background_manager = BackgroundManager(self.config)
```

LoginInterface coordinates the whole login page UI and interactions.

__init__:

Instantiates UIConfig() for configuration,

Creates AuthenticationManager() which imports DB utils,

Creates SessionManager() to manage session state,

Creates BackgroundManager(self.config) to handle background/styling.

python

Copy

Edit

```
   def get_current_view(self) -> LoginView:
```

```
    """Get current view from query parameters"""
    params = st.query_params
    view_param = params.get("view", "login")

    try:
        return LoginView(view_param)
    except ValueError:
        logger.warning(f"Unknown view parameter: {view_param}")
        return LoginView.LOGIN
get_current_view():
```

Reads Streamlit st.query_params (URL query parameters).

Tries to get the view parameter; if missing defaults to "login".

Attempts to convert the string into a LoginView enum. If conversion fails (unknown value), logs a warning and returns LoginView.LOGIN as safe default.

python

Copy

Edit

```
  def render_password_reset_view(self) -> None:
    """Render password reset form"""
    st.markdown("## 🔒 Reset Password")

    with st.form("reset_form"):
```

```
        email = st.text_input("Registered Email")
        new_password = st.text_input("New Password", type="password")
        confirm_password = st.text_input("Confirm New Password", type="password")
        submitted = st.form_submit_button("Reset Password")
render_password_reset_view():
```

Shows heading "Reset Password".

Creates a Streamlit form with key "reset_form". Inside:

email input,

new_password input with type="password" (masked),

confirm_password masked input,

submitted boolean bound to the form submit button.

python

Copy

Edit

```
    if submitted:
        if not all([email, new_password, confirm_password]):
            st.warning("All fields are required.")
        elif new_password != confirm_password:
            st.warning("Passwords do not match.")
```

```
        else:
            success, message = self.auth_manager.reset_user_password(email,
new_password)
            if success:
                st.success(message)
                st.markdown('[Back to Login](?view=login)', unsafe_allow_html=True)
            else:
                st.error(message)

    st.markdown('[← Back to Login](?view=login)', unsafe_allow_html=True)
```

After submission:

Validates that all fields are filled; if not shows warning.

Checks that new_password and confirm_password match; otherwise shows warning.

Calls self.auth_manager.reset_user_password(email, new_password):

On success: shows success message and a link to back to login (query ?view=login).

On failure: shows error message.

Always displays a "Back to Login" link (duplicate—one inside success branch and one after the block) to allow navigation.

python

Copy

Edit

```
  def render_login_view(self) -> None:
    """Render main login form"""
    st.markdown("## 🔐 Login to Brush and Soul")

    # Show success message if redirected from registration
    if self.get_current_view() == LoginView.REGISTERED_SUCCESS:
      st.success("✅ Registration successful! Please login.")

    with st.form("login_form"):
      username = st.text_input("Username")
      password = st.text_input("Password", type="password")
      submitted = st.form_submit_button("Login")
render_login_view():
```

Displays heading "Login to Brush and Soul".

If the current view equals REGISTERED_SUCCESS (Examined via get_current_view()), displays a success message indicating registration succeeded and asking to login.

Creates a form "login_form" with username and masked password inputs and submitted bound to the submit button.

python

Copy

Edit

```
    if submitted:
        if not username or not password:
            st.warning("Please enter both username and password.")
        else:
            login_result = self.auth_manager.login_user(username, password)

            if login_result.success:
                self.session_manager.login_user(login_result.user_data)
                st.success(f"Welcome {login_result.user_data['username']}!")
                st.switch_page("pages/04_Dashboard.py")
            else:
                st.error(login_result.error_message)
```

On form submission:

If either field empty → show warning.

Else call self.auth_manager.login_user(username, password):

If login_result.success True:

Calls self.session_manager.login_user(login_result.user_data) to persist user in session state,

Shows greeting success message,

Calls st.switch_page("pages/04_Dashboard.py") to navigate to the Dashboard page (Streamlit pages mechanism).

If login failed → shows st.error() with login_result.error_message.

python

Copy

Edit

```
    # Navigation links
    st.markdown('[Forgot Password?](?view=reset)', unsafe_allow_html=True)
    st.markdown(
      '[Don\'t have an account? <span style="color:#00f;text-decoration:underline;">'
      'Register here</span>](Register)',
      unsafe_allow_html=True
    )
```

Renders two navigation links:

"Forgot Password?" link points to ?view=reset (the same page with query param so get_current_view() will return RESET).

A "Register here" link pointing to a path Register (this likely links to a registration page route in the Streamlit app). HTML span is used to style the link text.

python

Copy

Edit

```
  def run(self) -> None:
    """Main application entry point"""
    # Configure page
```

```
    st.set_page_config(
        page_title=self.config.page_title,
        layout=self.config.layout
    )
```

run() is the main entry method for the login interface.

Calls st.set_page_config(page_title=..., layout=...) to apply page metadata (title, layout). This must be called early in a Streamlit script.

python

Copy

Edit

```
    # Initialize session and apply styling
    self.session_manager.initialize_session()
    self.background_manager.apply_styling()
```

Initializes session defaults via SessionManager.initialize_session().

Calls BackgroundManager.apply_styling() to inject CSS and background into the page (calls load_background_image() internally).

python

Copy

Edit

```
    # Main container
    st.markdown('<div class="block-container">', unsafe_allow_html=True)

    try:
```

```
        # Route to appropriate view
        current_view = self.get_current_view()

        if current_view == LoginView.RESET:
            self.render_password_reset_view()
        else:
            self.render_login_view()

    finally:
        # Close container
        st.markdown('</div>', unsafe_allow_html=True)
```

Opens an HTML which matches the CSS style set earlier to present the content inside a centered glassmorphism card.

Uses a try/finally to ensure the closing is always written even if an error occurs during rendering.

Inside the try:

Gets current_view from query parameters via get_current_view().

If the view is RESET it calls render_password_reset_view(); otherwise render_login_view() is called.

The finally block writes the closing markup to end the styled container.

Application entry point

python

Copy

Edit

```
def main():
  """Application main function"""
  try:
    app = LoginInterface()
    app.run()
  except Exception as e:
    logger.error(f"Critical application error: {e}")
    st.error("Application failed to start. Please contact support.")
```

Defines top-level main() function:

Tries to instantiate LoginInterface() and call app.run().

If any exception bubbles up during initialization or run, it logs a critical error and displays a user-visible st.error asking the user to contact support.

python

Copy

Edit

```
if __name__ == "__main__":
  main()
```

Standard Python guard: if the file is executed directly (python Login.py), call main(). In a multi-page Streamlit app, this might be invoked when Streamlit runs the page.

Quick flow summary (how the pieces connect)

On module load the classes are defined; nothing runs until main() is called.

main() creates LoginInterface:

AuthenticationManager() tries to import DB helpers (utils.authenticate, etc.) — if that fails the app shows DB error.

SessionManager() used to initialize session defaults.

BackgroundManager() prepares to load and style a background image (base64-encoded if available).

LoginInterface.run() sets page config, initializes session values, applies CSS/background, opens a styled container and routes:

If URL query ?view=reset → shows reset-password form and calls AuthenticationManager.reset_user_password() to update password.

Otherwise shows login form and calls AuthenticationManager.login_user() to verify credentials; on success sets session and switches page to Dashboard.

All database interactions are delegated to utils functions; this file focuses on UI flow, validation, styling, and session state.

Notes / potential gotchas & recommendations

AuthenticationManager._import_utils() raises on ImportError, which will prevent the page from fully loading; this is appropriate if the utils module is required.

background_image is a Windows-style absolute path (D:\...). On deployment (Linux/Streamlit Cloud) this path must be available or the background will fallback to gradient.

st.switch_page("pages/04_Dashboard.py") uses Streamlit's pages navigation; ensure that page path exists and matches your app structure.

Password validation depends on utils.is_valid_password returning (bool, message) — make sure it does.

The code uses st.query_params for view switching; links like ?view=reset update the URL query but do not do a full page navigation; Streamlit will respond by re-running and get_current_view() will route accordingly.

CSS injection uses unsafe_allow_html=True. Keep CSS safe and test across browsers/resolutions.

----------REFGISTER CODE

from __future__ import annotations

Enables postponed evaluation of type annotations (PEP 563). It makes annotations be stored as strings, avoiding forward-reference issues and reducing runtime import overhead when using type hints.

python

Copy

Edit

import base64

Imports the base64 module used to encode binary image bytes into base64 text. Later used to embed a background image inside CSS url("data:...").

python

Copy

Edit

import re

Imports the regular expressions module. Used below for email & username validation patterns.

python

Copy

Edit

from pathlib import Path

Imports Path for path manipulation (object-oriented file paths). Used when checking/reading the background image file.

python

Copy

Edit

from typing import Optional

Imports the Optional type hint (value or None) used in function signatures to signal optional return values.

python

Copy

Edit

import streamlit as st

Imports Streamlit and aliases it st — the app framework used to render the form, messages, and page layout.

Page configuration

python

Copy

Edit

```
# ---------------------------------------------------------------- #
#  PAGE CONFIGURATION                                              #
# ---------------------------------------------------------------- #
st.set_page_config(page_title="Register", layout="centered")
```

st.set_page_config(...) sets browser tab title and default page layout. Must be called early in a Streamlit script. Here it sets the page title to "Register" and centers the app container.

Try to import backend helpers

python

Copy

Edit

```
# ---------------------------------------------------------------- #
#  TRY TO IMPORT BACKEND HELPERS                                   #
# ---------------------------------------------------------------- #
try:
    from utils import register_user, is_valid_password  # database wrappers
    UTILS_OK = True
except ImportError as exc:
    UTILS_OK = False
```

```
    st.error(f"Import Error → {exc}")
    st.error(
        "utils.py is missing the following call-level wrappers that this page "
        "relies on:\n"
        " • register_user(username, email, password, user_type)\n"
        " • is_valid_password(password)"
    )
```

Line-by-line:

try: — start an attempt to import runtime helpers.

from utils import register_user, is_valid_password — tries to import two functions expected to be implemented in utils.py:

register_user(...) — used to create a new user record in the database; expected to return a success flag and a message.

is_valid_password(...) — used to validate password strength; expected to return a boolean and a message (as used later).

UTILS_OK = True — mark utilities available if import succeeds.

except ImportError as exc: — if import fails (module missing or missing functions) we catch it.

UTILS_OK = False — mark utilities unavailable.

st.error(f"Import Error → {exc}") — show a visible error to the user explaining the import error.

st.error(...) — show a second, clearer message telling the developer which wrapper functions are required. This helps debugging if utils.py is missing or incomplete.

Helper functions: convert image to base64

python

Copy

Edit

```
# ---------------------------------------------------------------- #
#  HELPER FUNCTIONS                                                #
# ---------------------------------------------------------------- #
def _img_to_base64(path: str | Path) -> Optional[str]:
    """Return an image file as a base64 data-URI or None if unavailable."""
```

Defines _img_to_base64, a helper that accepts a filesystem path and returns a base64 data: URI string if the file exists and can be read; otherwise returns None.

The return type is annotated Optional[str] (string or None).

python

Copy

Edit

```
    try:
        file_path = Path(path)
        if not file_path.exists():
            st.warning(f"Background image not found ➜ {file_path}")
```

return None

file_path = Path(path) — create a Path object from the input.

if not file_path.exists(): — if the file isn't on disk:

st.warning(...) — show a non-fatal warning in UI so dev/user knows background image is missing.

return None — fallback; no encoded image.

python

Copy

Edit

encoded = base64.b64encode(file_path.read_bytes()).decode()

return f"{encoded}"

file_path.read_bytes() — read all bytes of the image file.

base64.b64encode(...).decode() — base64 encode the bytes and decode to a UTF-8 string.

return f"{encoded}" — return a data: URI that can be used directly inside CSS background-image: url("...").

python

Copy

Edit

except Exception as err: # pragma: no cover

st.error(f"Error loading background image: {err}")

return None

Catch any unexpected exceptions while reading/encoding the file.

st.error(...) — surface an error message in the UI.

return None — fallback so the app can continue with a gradient background.

Apply the background & glassmorphism CSS

python

Copy

Edit

def _set_background() -> None:

"""Apply glassmorphism background; fallback to gradient if image missing."""

Defines _set_background() which will call _img_to_base64 and then inject CSS into the page to style background and the content container.

python

Copy

Edit

img_path = r"D:\Brush and soul\uploads\53b691e2-6023-4fc8-9c87-cc378544d4d8.jpg"

encoded = _img_to_base64(img_path)

img_path — hardcoded Windows path to a background image (raw string r"...").

encoded = _img_to_base64(img_path) — attempt to read & encode that file. encoded will be either the data: URI string or None.

python

Copy

Edit

```
  background_css = (
    f"""
    background-image: url("{encoded}");
    background-size: cover;
    background-position: center;
    background-repeat: no-repeat;
    background-attachment: fixed;
    """
    if encoded
    else """
    background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
    """
  )
```

Builds background_css string:

If encoded exists, build CSS rules that set background-image to the data URI and configure cover, center, no-repeat, fixed attachment.

If encoded is None, use a fallback linear gradient background CSS.

python

Copy

Edit

```
  st.markdown(
```

```
f"""
<style>
.stApp {{
  {background_css}
}}
.block-container {{
  background: rgba(255,255,255,0.15);
  backdrop-filter: blur(14px);
  -webkit-backdrop-filter: blur(14px);
  border-radius: 20px;
  border: 1px solid rgba(255,255,255,0.3);
  padding: 2rem;
  max-width: 400px;
  margin: 5vh auto;
  box-shadow: 0 8px 32px 0 rgba(31,38,135,0.37);
}}
h2 {{
  color: white;
  text-align: center;
}}
.stTextInput > div > div > input,
.stSelectbox  > div > div > div > div {{
  background-color: rgba(255,255,255,0.7) !important;
  border-radius: 10px;
}}
.stButton > button {{
```

```
        background-color: #4CAF50;
        color: white;
        border-radius: 8px;
        padding: 0.5rem 1rem;
        margin-top: 10px;
    }}
    </style>
    """,
    unsafe_allow_html=True,
)
```

Injects a <style> block into the Streamlit page using st.markdown(..., unsafe_allow_html=True) so we can override default Streamlit styles.

CSS details:

.stApp { {background_css} } — set the chosen background (image or gradient).

.block-container { ... } — style a centered, semi-transparent, blurred container (glassmorphism) with rounded corners, border, padding, fixed max-width (400px), and box shadow. This container will wrap the form.

h2 { color: white; text-align:center } — ensure heading is white and centered for contrast on the background.

.stTextInput and .stSelectbox CSS rules set input backgrounds to semi-opaque white and round corners to make inputs readable against the background.

.stButton > button styles the submit button with a green background and rounded shape.

unsafe_allow_html=True is required to render raw HTML/CSS in Streamlit.

Email validation helper

python

Copy

Edit

```
def validate_email(email: str) -> bool:
    """RFC-style email validation."""
    pattern = r"^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$"
    return bool(re.match(pattern, email.strip()))
```

Line by line:

def validate_email(email: str) -> bool: — declare a helper that returns True if email looks valid.

pattern = ... — a regex that:

^[A-Za-z0-9._%+-]+ ensures one or more allowed local-part characters,

@ literal,

[A-Za-z0-9.-]+ allowed domain characters,

\.[A-Za-z]{2,}$ top-level domain of at least two letters.

return bool(re.match(pattern, email.strip())) — strip whitespace and run regex match from start; convert to boolean (True if match found).

Note: this regex is a reasonable pragmatic check (not a full RFC 5322 implementation) but is sufficient for most inputs.

Username validation helper

python

Copy

Edit

```
def validate_username(username: str) -> tuple[bool, str]:
    """Username length & charset checks."""
    username = username.strip()
    if not (3 <= len(username) <= 20):
        return False, "Username must be 3 – 20 characters long."
    if not re.match(r"^[A-Za-z0-9_]+$", username):
        return False, "Username may contain only letters, numbers, and underscores."
    return True, "OK"
```

Line-by-line:

def validate_username(username: str) -> tuple[bool, str]: — returns (ok_flag, message).

username = username.strip() — remove leading/trailing whitespace.

if not (3 <= len(username) <= 20): — require username length between 3 and 20 characters inclusive.

return False, "Username must be 3 – 20 characters long." — return failure with message.

if not re.match(r"^[A-Za-z0-9_]+$", username): — allow only letters, digits, and underscore (no spaces, hyphens, punctuation).

return False, "Username may contain only letters, numbers, and underscores."

return True, "OK" — if both checks pass, return success.

Page layout — apply background & ensure utils available

python

Copy

Edit

--

PAGE LAYOUT

--

_set_background()

Calls _set_background() immediately to inject CSS & background into the page before rendering the content.

python

Copy

Edit

if not UTILS_OK:

  st.stop()

If backend helpers (utils) failed to import earlier, UTILS_OK is False. st.stop() halts execution of the rest of the script: the page will show the earlier st.error messages and not render the form. This prevents running code that would crash because helper functions are missing.

Render container & registration form

python

Copy

Edit

with st.container():

  st.markdown(" ", unsafe_allow_html=True)

  st.markdown("<h2>User Registration</h2>", unsafe_allow_html=True)

with st.container(): — creates a Streamlit container (a block of UI). Inside it:

st.markdown(" ", unsafe_allow_html=True) — open a with the block-container class (the CSS we injected earlier styles this as the glass card). This is manual HTML wrapper used to constrain width & apply the glass effect.

st.markdown("<h2>User Registration</h2>", unsafe_allow_html=True) — write an <h2> heading inside that wrapper (styled white & centered earlier).

python

Copy

Edit

  with st.form("register_form"):

    username = st.text_input("Username", max_chars=20)

    email = st.text_input("Email")

    password = st.text_input("Password", type="password")

```
    confirm = st.text_input("Confirm Password", type="password")
    user_type = st.selectbox("User Type", ["artist", "customer"])
    submit = st.form_submit_button("Register")
```

with st.form("register_form"): — create a Streamlit form; until the form button is clicked, inputs do not submit / trigger actions.

username = st.text_input("Username", max_chars=20) — text input for username limited to 20 chars.

email = st.text_input("Email") — text input for email.

password = st.text_input("Password", type="password") — password input (characters hidden).

confirm = st.text_input("Confirm Password", type="password") — confirm password input.

user_type = st.selectbox("User Type", ["artist", "customer"]) — dropdown to choose user role; provides two options.

submit = st.form_submit_button("Register") — form submit button, returns True when clicked.

Form submission handling

python

Copy

Edit

```
    if submit:
```

try:

if submit: — handle the form submission only after user clicks the button.

try: — start of try/except to catch unexpected errors and show friendly messages rather than crashing.

Field completeness check

python

Copy

Edit

```
# ---------- field completeness ---------- #
if not all([username, email, password, confirm]):
  st.warning("Please fill all fields.")
  st.stop()
```

if not all([username, email, password, confirm]): — ensure none of the required fields are empty.

st.warning("Please fill all fields.") — show a warning message.

st.stop() — halt the rest of the script execution on this run (prevents further validation/registration attempt).

Email & username validation

python

Copy

Edit

```
# ---------- email / username validation ---------- #
```

```
    if not validate_email(email):
        st.warning("Please enter a valid email.")
        st.stop()

    ok, msg = validate_username(username)
    if not ok:
        st.warning(msg)
        st.stop()
```

if not validate_email(email): — validate email format; on failure warn and stop.

ok, msg = validate_username(username) — call username validator; if ok is False, show the msg and stop.

Password checks

python

Copy

Edit

```
    # ---------- password checks ---------- #
    if password != confirm:
        st.warning("Passwords do not match.")
        st.stop()

    valid_pwd, pwd_msg = is_valid_password(password)
    if not valid_pwd:
        st.warning(pwd_msg)
        st.stop()
```

if password != confirm: — ensure both password fields match; if not, warn and stop.

valid_pwd, pwd_msg = is_valid_password(password) — call is_valid_password imported from utils:

expects to return a tuple (bool, message) indicating whether password meets rules (length, complexity).

if not valid_pwd: — if password fails strength checks, display pwd_msg (which explains why) and st.stop().

Register user (call into backend)

python

Copy

Edit

```
# ---------- FIXED: register with correct parameters and return handling ---------- #
success, message = register_user(
  username.strip(),
  email.strip(),
  password.strip(),
  user_type
)
```

Calls register_user(...) from utils with cleaned inputs:

username.strip() — remove whitespace,

email.strip() — remove whitespace,

password.strip() — remove whitespace,

user_type — role chosen from selectbox.

Expects register_user to return a tuple (success: bool, message: str) where success signals whether registration succeeded and message is a textual explanation.

python

Copy

Edit

```
        if success:
            st.success(f"✅ {message} Redirecting to login...")
            # Add a small delay to show success message
            import time
            time.sleep(1)
            st.switch_page("pages/Login.py")
        else:
            st.error(f"❌ Registration failed: {message}")
```

If success is True:

st.success(...) — display a green success message that includes the backend message.

import time and time.sleep(1) — pause 1 second so the user can briefly read the success text (this is a UI nicety).

st.switch_page("pages/Login.py") — programmatically navigate to the Login page in the Streamlit multi-page app.

If success is False:

st.error(f"❌ Registration failed: {message}") — display an error box explaining the failure message returned by register_user (e.g., username/email already in use).

python

Copy

Edit

```
except Exception as err:  # pragma: no cover
    st.error(f"Registration error: {err}")
    st.error("Check your database connection and utils.py file.")
```

Catch-all exception handler for unexpected runtime errors during registration.

st.error(f"Registration error: {err}") — show the exception message to the user (helps debug).

st.error("Check your database connection and utils.py file.") — give a suggested root cause and next step.

pragma: no cover indicates this block may be excluded from test coverage reports.

Navigation button & close container

python

Copy

Edit

```
# Navigation button
if st.button("Back to Login"):
    st.switch_page("pages/Login.py")


st.markdown("</div>", unsafe_allow_html=True)
```

if st.button("Back to Login"): — a simple Streamlit button placed below the form; if clicked:

st.switch_page("pages/Login.py") — navigate back to the login page.

st.markdown(" ", unsafe_allow_html=True) — closes the earlier wrapper. The container and explicit opening/closing of the div are used to guarantee the CSS glass card surrounds the form content.

Summary — what this file does and its runtime flow

Sets page metadata and tries to import two essential helper functions from utils.py: register_user and is_valid_password. If these are missing, it reports errors and stops the page.

Defines small helper functions:

_img_to_base64 — read & encode a background image file (if available).

_set_background — inject CSS to create a background (image or gradient) and a glassmorphism card for the form.

validate_email and validate_username — client-side checks for basic correctness before calling the backend.

Renders a centered glass card with a registration form (username, email, password, confirm, user type).

On submit, runs a sequence of validations:

all required fields present,

email format valid,

username length & charset valid,

passwords match,

password strength validated by utils.is_valid_password.

Calls register_user(...) with cleaned inputs. If backend returns success, shows success message, waits 1s and navigates to the login page; if failure, shows backend message.

Provides a "Back to Login" button that also navigates back.

Practical notes & suggestions

utils.py contract: this page requires register_user(username, email, password, user_type) returning (bool, message) and is_valid_password(password) returning (bool, message). Make sure utils.py matches this contract.

Background path: the hardcoded img_path is a Windows absolute path. On deployment (Linux, cloud), that path will likely be unavailable — the code correctly falls back to a gradient if the image file is missing.

Security: password.strip() removes spaces; be mindful if passwords should allow leading/trailing spaces (commonly they shouldn't). Also ensure register_user handles password hashing and does not store plaintext.

Race & UI: using time.sleep(1) blocks the Streamlit process for one second — acceptable for a small UX pause but not ideal for heavy apps. Alternative: show a spinner and immediately switch page.

Accessibility: inputs have reasonable defaults; consider adding help text for password rules so users don't get surprised by is_valid_password failures.

----------MY PROFILE CODE

Imports & logger

from __future__ import annotations

- Tells Python to store type annotations as strings (deferred evaluation). Prevents forward-reference problems and can reduce import-time issues when using types that are defined later.

import logging

- Imports Python's logging module so the program can write info/warning/error messages to logs.

from abc import ABC, abstractmethod

- Imports abstract base class helpers. ABC is the base for abstract classes; abstractmethod marks methods that subclasses must implement.

from contextlib import contextmanager

- Imports a decorator to easily create context managers (usable with with blocks). Used later to wrap DB error handling.

from dataclasses import dataclass, field

- Imports the dataclass decorator and field helper. Dataclasses reduce boilerplate for classes that primarily store data.

from enum import Enum

- Imports Enum which is used to define named constant sets (user roles here).

from typing import Any, Dict, Optional, Protocol, Tuple

- Imports typing helpers used in annotations: Any, Dict, Optional, Protocol (structural typing), and Tuple.

import streamlit as st

- Imports Streamlit (aliased to st) — the UI framework used by the app.

logger = logging.getLogger(__name__)

- Gets a module-level logger named after the module. Use logger.info(), logger.error() etc. throughout the file to record messages.

Type definitions & protocols

```
class ProfileOpsProtocol(Protocol):
    """Database-side operations required by this page."""
    def authenticate(self, username: str, password: str) -> Optional[Dict[str, Any]]: ...
    def update_password(self, email: str, new_password: str) -> bool: ...
    def is_valid_password(self, password: str) -> Tuple[bool, str]: ...
    def update_user_field(self, username: str, field: str, value: str) -> bool: ...
```

- Declares a Protocol (structural interface) describing the DB functions this module expects:
    - authenticate(...) should return user info dict or None.
    - update_password(...) should return boolean success.

    - is_valid_password(...) should return (bool, message) about strength.
    - update_user_field(...) should update arbitrary user field (like bio) and return success.
- Nothing is implemented here — this only documents the shape of the DB adapter.

```
class UserRole(Enum):
  ARTIST = "artist"
  CUSTOMER = "customer"
```

- Defines UserRole enum with two allowed values: ARTIST and CUSTOMER.

```
  @classmethod
  def from_string(cls, raw: str) -> "UserRole":
    try:
      return cls(raw.lower())
    except ValueError:
      return cls.CUSTOMER
```

- Class method to safely convert a string to UserRole: normalizes to lowercase and falls back to CUSTOMER if the string doesn't match any enum member.

```
  def display_name(self) -> str:
    """Get display name for the role"""
    return self.value.capitalize()
```

- Instance method to produce a human-friendly role name (e.g., "artist" → "Artist").

Config & data models

```
@dataclass(frozen=True)
class UIConfig:
  page_title: str = "My Profile"
  page_icon: str = " 👤 "
```

```
    max_bio_length: int = 300
```

- Immutable configuration dataclass for UI-level constants:
    - page_title and page_icon for page header,
    - max_bio_length limits bio length.

```
@dataclass
class UserCtx:
    username: str
    email: str
    role: UserRole
    bio: str = ""
    is_authenticated: bool = True
```

- Dataclass representing the current user/context used by UI components:
    - Required fields: username, email, role.
    - Optional bio and is_authenticated flag.

```
    @classmethod
    def from_session(cls) -> Optional["UserCtx"]:
        if "user" not in st.session_state or not st.session_state.get("logged_in", False):
            return None

        u = st.session_state.user
        return cls(
            username=u.get("username", ""),
            email=u.get("email", ""),
            role=UserRole.from_string(u.get("user_type", "customer")),
            bio=u.get("bio", "")
        )
```

- Class method to build UserCtx from Streamlit session state:
    - Checks if user exists and logged_in is True; returns None if not authenticated.
    - Reads session user dict and maps fields into a UserCtx. Uses safe defaults if keys missing.

```
  def update_session_bio(self, new_bio: str) -> None:
    """Update bio in session state"""
    if "user" in st.session_state:
      st.session_state.user["bio"] = new_bio
```

- Instance method to persist an updated bio into st.session_state.user["bio"] so the session reflects changes immediately.

```
@dataclass
class PasswordChangeRequest:
  """Data structure for password change requests"""
  current_password: str
  new_password: str
  confirm_password: str
```

- Dataclass to group password-change inputs.

```
  def validate(self) -> Tuple[bool, str]:
    """Validate password change request"""
    if not all([self.current_password, self.new_password, self.confirm_password]):
      return False, "Please fill in all the fields."

    if self.new_password != self.confirm_password:
      return False, "New passwords do not match."

    return True, ""
```

- Validates that all fields are present and that new/confirm match. Returns (True, "") when valid or (False, message) with reason.

Database manager

```
class DatabaseManager:
  """Advanced database operations manager with comprehensive error handling."""


  def __init__(self):
    self.operations_available = self._initialize_operations()
```

- Class that encapsulates interaction with the real database helpers.
- __init__ calls _initialize_operations() and stores whether DB functions were successfully initialized.

```
  def _initialize_operations(self) -> bool:
    """Initialize database operations with graceful error handling"""
    try:
      from utils import (
        authenticate,
        update_password,
        is_valid_password,
        update_user_field,
      )
```

- Attempts to import required functions from a utils module (expected to be provided separately). If import fails, the manager will mark DB ops as unavailable.

```
      # Create operations wrapper
      class DatabaseOperations:
        def __init__(self):
          self.authenticate = authenticate
```

self.update_password = update_password

self.is_valid_password = is_valid_password

self.update_user_field = update_user_field

- Defines a thin wrapper class DatabaseOperations that stores the imported functions as attributes. This organizes the functions under self.ops later.

self.ops = DatabaseOperations()

logger.info("Database operations initialized successfully")

return True

- Instantiates the wrapper and logs success, returning True indicating DB helpers are available.

except ImportError as e:

logger.error(f"Database utilities not available: {e}")

return False

- If import fails, logs error and returns False — the manager will avoid running DB calls.

@contextmanager

def _error_handler(self, op: str):

"""Context manager for database operation error handling"""

try:

yield

except Exception as exc:

logger.error(f"Database op '{op}' failed: {exc}")

pass

- Small context manager used to wrap DB calls; it yields to the wrapped block and logs exceptions without letting them bubble. It swallows the exception (returns control to caller) — ensures the UI won't crash on DB errors.

Authentication operations

```
def verify_user_password(self, username: str, password: str) -> bool:
    """Verify user's current password"""
    if not self.operations_available:
        return False
```

- Public method that returns whether the supplied password is correct for given username.
- Immediately returns False if DB helpers are not available.

```
    with self._error_handler("authenticate"):
        try:
            result = self.ops.authenticate(username, password)
            return bool(result)
        except Exception:
            return False
```

- Calls self.ops.authenticate(...) inside the _error_handler context:
    - authenticate should return user info dict on success or falsy otherwise.
    - The method returns True if authenticate returned something truthy, otherwise False. Exceptions are caught and result in False.

```
def change_user_password(self, email: str, new_password: str) -> bool:
    """Change user's password"""
    if not self.operations_available:
        return False
```

- Method to update user's password in DB; returns False immediately if DB not available.

```
    with self._error_handler("update_password"):
        try:
            return self.ops.update_password(email, new_password)
        except Exception:
```

return False

- Calls ops.update_password(email, new_password) and returns its boolean result; exceptions produce False.

def validate_password_strength(self, password: str) -> Tuple[bool, str]:

"""Validate password strength"""

if not self.operations_available:

return False, "Password validation unavailable"

- Wrapper for is_valid_password DB helper. If DB not available, returns (False, "Password validation unavailable").

with self._error_handler("is_valid_password"):

try:

return self.ops.is_valid_password(password)

except Exception:

return False, "Password validation failed"

- Calls ops.is_valid_password(password) inside the error handler; on exception returns a failure tuple.

User field operations

def update_user_bio(self, username: str, bio: str) -> bool:

"""Update user's bio"""

if not self.operations_available:

return False

- Updates a user's bio field using the DB helper update_user_field. Returns False if DB not available.

with self._error_handler("update_user_field"):

try:

return self.ops.update_user_field(username, "bio", bio)

except Exception:

return False

- Calls ops.update_user_field(username, "bio", bio) and returns its boolean result; exceptions result in False.

Session manager

```
class SessionManager:
  """Advanced session state management"""

  @staticmethod
  def logout_user() -> None:
    """Logout user and clear session"""
    st.session_state.logged_in = False
    st.session_state.user = None
```

- Simple static utilities for session control.
- logout_user sets logged_in flag to False and clears user data from Streamlit session state.

```
  @staticmethod
  def validate_authentication() -> Optional[UserCtx]:
    """Validate user authentication and return user context"""
    user_ctx = UserCtx.from_session()
    if not user_ctx:
      st.warning("Please log in to access your profile.")
      st.stop()
    return user_ctx
```

- validate_authentication verifies whether a user is logged in by calling UserCtx.from_session():

- If not authenticated, shows a Streamlit warning and calls st.stop() to halt further UI execution.
- Otherwise returns the UserCtx instance for downstream components.

Advanced UI components — base class and style injection

```
class UIComponent(ABC):
  """Abstract base class for UI components"""

  def __init__(self, cfg: UIConfig, db: DatabaseManager):
    self.cfg = cfg
    self.db = db

  @abstractmethod
  def render(self, user_ctx: UserCtx) -> None:
    """Render the UI component"""
    pass
```

- UIComponent is an abstract base class that all UI parts inherit.
- Ensures each component receives cfg and db references and implements a render(user_ctx) method.

```
class StyleManager(UIComponent):
  """Handles CSS styling for the application"""

  def render(self, user_ctx: UserCtx) -> None:
    """Apply custom CSS styling exactly as original"""
    st.markdown(""" ... large CSS ... """, unsafe_allow_html=True)
```

- StyleManager injects a large CSS block via st.markdown(..., unsafe_allow_html=True) that:

    - Defines color variables for a brown theme,
    - Styles buttons, profile cards, inputs, headers, success/error message wrappers, etc.
- This centrally controls the look-and-feel; render(None) is called early to apply styling to the whole page.

AccountInfoDisplay component

```
class AccountInfoDisplay(UIComponent):
  """Account information display component"""

  def render(self, user_ctx: UserCtx) -> None:
    """Render account information section"""
    with st.container():
      st.markdown("<div class='glass'>", unsafe_allow_html=True)
      st.markdown("<div class='header'><h2> Account Information</h2></div>", unsafe_allow_html=True)
      st.write(f"**Username:** {user_ctx.username}")
      st.write(f"**Email:** {user_ctx.email}")
      st.write(f"**Role:** {user_ctx.role.display_name()}")
      st.markdown("</div>", unsafe_allow_html=True)
```

- AccountInfoDisplay.render:
    - Creates a Streamlit container() (logical grouping for layout).
    - Emits HTML and header markup to apply CSS.
    - Uses st.write() to print username, email, and role (with display_name()).
    - Closes the HTML div. This component just shows current account info.

BioUpdateForm component

```
class BioUpdateForm(UIComponent):
    """Bio update form component"""

    def render(self, user_ctx: UserCtx) -> None:
        """Render bio update form"""
        with st.container():
            st.markdown("<div class='glass'>", unsafe_allow_html=True)
            st.markdown("<div class='header'><h2>📝 About You</h2></div>",
unsafe_allow_html=True)
```

- render opens a container and the glass-styled wrapper and header.

```
            new_bio = st.text_area(
                f"Bio (max {self.cfg.max_bio_length} characters):",
                value=user_ctx.bio,
                max_chars=self.cfg.max_bio_length
            )
```

- Shows a text_area populated with the user's current bio:
  - Label includes the configured max length.
  - value pre-fills with user_ctx.bio.
  - max_chars limits input length to self.cfg.max_bio_length.

```
            if st.button("Save Bio"):
                self._handle_bio_update(user_ctx, new_bio)

            st.markdown("</div>", unsafe_allow_html=True)
```

- When the Save Bio button is clicked, the component calls _handle_bio_update to persist the bio and update session.
- Closes the wrapper div.

```
def _handle_bio_update(self, user_ctx: UserCtx, new_bio: str) -> None:
    """Handle bio update submission"""
    try:
        if self.db.update_user_bio(user_ctx.username, new_bio):
            user_ctx.update_session_bio(new_bio)
            st.success("✅ Bio updated successfully!")
        else:
            st.error("❌ Could not update your bio.")
    except Exception as e:
        logger.error(f"Error updating bio: {e}")
        st.error("❌ Could not update your bio.")
```

- _handle_bio_update:
    - Calls self.db.update_user_bio(username, new_bio) to persist change in DB.
    - On success, calls user_ctx.update_session_bio(new_bio) to update the session state immediately, then shows a success message.
    - If DB call fails or raises an exception, logs the error and shows an error message to the user.

PasswordChangeForm component

```
class PasswordChangeForm(UIComponent):
    """Password change form component"""

    def render(self, user_ctx: UserCtx) -> None:
        """Render password change form"""
        with st.container():
            st.markdown("<div class='glass'>", unsafe_allow_html=True)
```

```
    with st.expander("🔐 Change Password"):
        st.markdown("Update your password below:")

        # Use unique form key to prevent conflicts
        form_key = f"change_password_form_{user_ctx.username}"

        with st.form(form_key):
            current_pw = st.text_input("Current Password", type="password")
            new_pw = st.text_input("New Password", type="password")
            confirm_pw = st.text_input("Confirm New Password", type="password")
            submitted = st.form_submit_button("Update Password")

            if submitted:
                self._handle_password_change(user_ctx, current_pw, new_pw, confirm_pw)

    st.markdown("</div>", unsafe_allow_html=True)
```

- render builds a collapsible expander for the password change form:
    - Uses a unique form key per user to avoid widget key collisions.
    - Has three password inputs (current, new, confirm) and a submit button.
    - On submit, calls _handle_password_change.

```
def _handle_password_change(self, user_ctx: UserCtx, current_pw: str,
                            new_pw: str, confirm_pw: str) -> None:
    """Handle password change submission"""
    # Create password change request
    request = PasswordChangeRequest(current_pw, new_pw, confirm_pw)
```

```
# Validate request format
is_valid, error_msg = request.validate()
if not is_valid:
    st.warning(f"⚠️ {error_msg}")
    return
```

- Builds a PasswordChangeRequest object and runs its validate():
    - If validation fails (empty fields or mismatch) shows a warning and stops.

```
try:
    # Verify current password
    if not self.db.verify_user_password(user_ctx.username, current_pw):
        st.error("❌ Current password is incorrect.")
        return
```

- Verifies the current password by calling db.verify_user_password(username, current_pw):
    - If verification fails, informs the user and stops.

```
    # Validate new password strength
    valid, message = self.db.validate_password_strength(new_pw)
    if not valid:
        st.warning(f"⚠️ {message}")
        return
```

- Calls db.validate_password_strength(new_pw) to ensure the new password meets policy. If invalid, informs the user and stops.

```
    # Update password
    if self.db.change_user_password(user_ctx.email, new_pw):
        st.success("✅ Password updated successfully!")
```

else:

st.error("❌ Could not update password. Please try again later.")

- Attempts to persist the new password with db.change_user_password(user_ctx.email, new_pw):
    - On success shows success message.
    - On failure shows an error message.

except Exception as e:

logger.error(f"Error changing password: {e}")

st.error("❌ Could not update password. Please try again later.")

- Catches and logs any unexpected exception and shows a generic error to the user.

LogoutSection component

class LogoutSection(UIComponent):

"""Logout section component"""

def render(self, user_ctx: UserCtx) -> None:

"""Render logout section"""

st.markdown("---")

if st.button("🚪 Logout"):

SessionManager.logout_user()

st.switch_page("pages/Login.py")

- Renders a horizontal divider (---) then a Logout button.
- On click:
    - Calls SessionManager.logout_user() to clear session state.
    - Uses st.switch_page("pages/Login.py") to navigate to the login page (Streamlit multi-page navigation).

Main application — composition & runtime

```
class ProfileApplication:
  """Main profile application with component-based architecture"""


  def __init__(self):
    self.cfg = UIConfig()
    self.db = DatabaseManager()
    self._initialize_components()
```

- Main app class that wires everything together:
    - Creates UI config,
    - Creates DatabaseManager() (attempts to import DB helpers),
    - Calls _initialize_components to create UI parts.

```
  def _initialize_components(self) -> None:
    """Initialize all UI components"""
    self.style_manager = StyleManager(self.cfg, self.db)
    self.account_info = AccountInfoDisplay(self.cfg, self.db)
    self.bio_form = BioUpdateForm(self.cfg, self.db)
    self.password_form = PasswordChangeForm(self.cfg, self.db)
    self.logout_section = LogoutSection(self.cfg, self.db)
```

- Instantiates each UI component and passes config + DB manager into them so they all share same resources.

```
  def run(self) -> None:
    """Main application entry point"""
    # Apply styling
    self.style_manager.render(None)
```

```
    # Validate authentication
    user_ctx = SessionManager.validate_authentication()

    # Render page content
    self._render_content(user_ctx)
```

- run() is the app entry:
    - Calls style_manager.render(None) to inject CSS.
    - Validates authentication via SessionManager.validate_authentication(); this either returns UserCtx or stops execution (if not logged in).
    - Renders the page content with _render_content(user_ctx).

```
def _render_content(self, user_ctx: UserCtx) -> None:
    """Render main page content"""
    # Page title
    st.title(f"{self.cfg.page_icon} {self.cfg.page_title}")
    st.markdown("---")

    # Render components
    self.account_info.render(user_ctx)
    self.bio_form.render(user_ctx)
    self.password_form.render(user_ctx)
    self.logout_section.render(user_ctx)
```

- _render_content:
    - Shows page title using config icon and title.
    - Renders a divider.
    - Calls the render method on all components in order:

- Account info (display),
- Bio update form,
- Password change form,
- Logout section.

Application entry point

```
def main() -> None:
  """Application main function"""
  try:
    app = ProfileApplication()
    app.run()
  except Exception as e:
    logger.error(f"Application error: {e}")
    st.error("Application failed to load. Please try again.")
```

- Top-level main() that instantiates ProfileApplication() and runs it.
- Wraps in try/except: if any error occurs while setting up/running the app, logs the error and shows a user-visible error message.

```
if __name__ == "__main__":
  main()
```

- Standard Python guard: if this file is executed directly, call main(). In a Streamlit multipage app this may be executed by Streamlit's page runner.

Quick flow summary (runtime path)

1. Module loads class & function definitions; nothing runs until main() is invoked.
2. main() creates a ProfileApplication instance:
    - DatabaseManager tries to import utils helpers. If missing, DB ops marked unavailable and DB calls will return failure defaults.

3. app.run() injects CSS, validates user session (stops and prompts login if not authenticated), then renders the account info, bio update form, password change form, and logout area.
4. Each interactive action (save bio, change password, logout) calls into DatabaseManager which delegates to utils functions within safe try/except contexts and returns friendly success/failure messages.

Notes / gotchas & suggestions

- DatabaseManager._initialize_operations() swallows ImportError and sets operations_available = False. UI flows should gracefully indicate features disabled — currently the code will show errors when calls fail; consider surfacing a banner if DB unavailable.
- SessionManager.validate_authentication() calls st.stop() which halts execution — that is correct for blocking access to the page, but be mindful when composing pages with sidebars.
- StyleManager uses unsafe_allow_html=True to inject CSS. Ensure your CSS selectors match Streamlit's markup across Streamlit versions — these class names and structures can change.
- st.switch_page("pages/Login.py") assumes your multi-page layout uses that path & that Streamlit pages are configured accordingly.
- Ensure utils functions conform to the ProfileOpsProtocol signatures (return types as assumed here), especially is_valid_password which must return (bool, message).