Builder Portfolio Management System — Code Logic Documentation

1. Project Overview

The **Builder Portfolio Management System (BPMS)** is a modular Java application designed to help **construction firms, builders, and clients** efficiently track and manage ongoing, upcoming, and completed projects.
It provides a structured way to manage **users**, **projects**, **documents**, **budgets**, and **timelines**, enabling clear visibility across the entire project lifecycle.

The project follows a **clean, enterprise-grade layered architecture (MVC + Service/DAO pattern)**, ensuring maintainability, scalability, and clear separation of concerns between presentation, business logic, and data persistence layers.

Architecture: Controller → Service → DAO → Model → Util

---

2. System Architecture and Design

Layer	Package	Responsibility	Key Classes
Controller	`com.builder.portfolio.controller`	Handles user input, menu navigation, and triggers appropriate business logic.	`AdminController`, `BuilderController`, `ClientController`
Service	`com.builder.portfolio.service`	Encapsulates core business logic and workflow coordination between controller and DAO.	`UserServiceImpl`, `ProjectServiceImpl`, `DocumentServiceImpl`
DAO (Data Access)	`com.builder.portfolio.dao`	Manages SQL operations and database persistence logic.	`UserDAOImpl`, `ProjectDAOImpl`, `DocumentDAOImpl`
Model	`com.builder.portfolio.model`	Represents real-world entities and data transfer	`User`, `Project`, `Document`, `BudgetReport`

		objects (DTOs).	
Utility	`com.builder.portfolio.util`	Provides shared helpers, constants, and connection utilities.	`DBConnectionUtil`, `BudgetUtil`, `StatusConstants`, `GanttChartUtil`

Layered StructureCore Technologies

- **Language:** Java 17
- **Database:** PostgreSQL
- **Design Pattern:** MVC + DAO + Service Layered Architecture
- **Build Tool:** IntelliJ IDEA / Maven
- **Logging:** `java.util.logging`
- **Persistence:** JDBC with custom connection utility

---

3. Login & Registration Logic

Workflow Summary

This module manages user authentication and role-based routing within the system.
Each user has a role (`ADMIN`, `BUILDER`, or `CLIENT`) that determines their access privileges and dashboard.

Registration Flow

1. The user selects **“Register”** from the main menu.
2. Inputs details: `name`, `email`, `password`, and `role`.
3. `UserController` sends this data to `UserService.registerUser(User)`.

4. `UserService` validates email uniqueness using `UserDAO.findByEmail(email)`.
5. If valid, `UserDAO.addUser(User)` inserts a record into the `users` table.
6. The user receives a registration success message.

Login Flow

1. User chooses **“Login”**.
2. Inputs `email` and `password`.
3. `UserService.login(email, password)` checks credentials using `UserDAO.findByEmailAndPassword()`.
4. If verified:
   - **Admin** → routed to `AdminController.showMenu()`
   - **Builder** → routed to `BuilderController.showMenu()`
   - **Client** → routed to `ClientController.showMenu()`
5. If invalid, an error message is displayed.

> **Security Note:** Passwords are currently stored in plaintext for demonstration. In a production environment, they should be hashed using BCrypt or Argon2.

---

4. Project Management Logic

The **Project Management module** is the heart of the application, handling creation, updates, deletion, and viewing of projects.

a. Add Project

1. Builder chooses “Add Project” from the menu.
2. Inputs project details including name, description, budgets, and dates.

3. If `status` is null or empty, the system defaults it to `UPCOMING`.

`ProjectService.addProject()` validates data and calls `ProjectDAO.addProject()` to execute:

```
INSERT INTO projects (...);
```

4. On success, confirmation is printed:
   "Project added successfully."

b. Update Project Status

1. Builder selects "Update Project".
2. System fetches the record via `ProjectService.getProject(id)`.
3. Builder updates fields (status, budget, or timeline).
4. `ProjectService.updateProject()` triggers DAO update.
5. If status changes to `IN_PROGRESS` or `COMPLETED`, `GanttChartUtil.printSimpleGantt(project)` is invoked to display a textual progress chart.

c. View Portfolio

- **Admin:** Lists all projects (`listAllProjects()`)
- **Builder:** Lists projects they created (`listProjectsByBuilder(builderId)`)
- **Client:** Lists assigned projects (`listProjectsByClient(clientId)`)

d. Delete Project

- Admin or Builder can delete a project they own.

The DAO validates ownership before deletion:

```
DELETE FROM projects WHERE id=? AND builder_id=?;
```

---

5. Document Management (Mock File Upload)

Overview

The **Document Management module** simulates a real-world file upload system by storing metadata of project-related documents (e.g., blueprints, permits, invoices).
While no actual file transfer occurs, the workflow accurately reflects how file metadata is handled in production systems.

Workflow Summary

1. Builder selects **“Add Document Metadata”** from their menu.
2. Inputs:
   - Document name
   - Document type (e.g., Blueprint, Contract)
   - Associated Project ID
   - Uploaded By (builder’s user ID)
   - Upload date (auto-filled using `LocalDate.now()`)
3. Controller constructs a `Document` object and calls `DocumentService.addDocument(Document)`.
4. `DocumentService` validates and forwards the call to `DocumentDAO.addDocument(Document)`.

DAO executes:

```
 INSERT INTO documents (project_id, document_name, document_type,
uploaded_by, upload_date)
VALUES (?, ?, ?, ?, ?);
```

5. System displays:
   "Document metadata saved successfully."

Viewing Documents

Builders or admins can view project documents using:

```
documentService.listDocumentsByProject(projectId);
```

Output (console view):

```
Document ID | Name         | Type         | Uploaded By | Upload Date
-------------------------------------------------------------------
1           | Plan_A.pdf   | Blueprint    | Builder_2   | 2025-10-09
2           | Contract.doc | Legal        | Builder_2   | 2025-10-09
```

Why "Mock Upload"?

This version only records **metadata**—not the physical file—making it lightweight and IDE-friendly.
However, it can easily be extended to:

- Store real files in a directory or AWS S3.
- Add `file_path` or `file_url` columns.
- Enforce role-based access to documents.

6. Budget & Timeline Tracking Logic

Budget Report

1. Builder selects **“View Budget Report”**.
2. `BuilderController` calls `ProjectService.getProject(projectId)`.
3. `ProjectService.buildBudgetReport(Project)` uses:
    - `BudgetUtil.calculateVariance()` to compute difference between `budgetUsed` and `budgetPlanned`.
    - `BudgetUtil.determineBudgetHealth()` to label project as:
        - **UNDER** (spent less)
        - **ON_TRACK** (within range)
        - **OVER** (overspent)

Report displayed:

```
Planned Budget: ₹10,00,000
Used Budget: ₹9,20,000
Variance: ₹-80,000
Health: UNDER
```

Timeline Tracking

If a project’s status is `IN_PROGRESS` or `COMPLETED`, `GanttChartUtil.printSimpleGantt(project)` prints:

```
Design      |#######........|
Permits     |....####.......|
Build       |.......########|
Testing     |..........#####|
```

This provides a quick textual visualization of project progress.

---

7. Database Schema

The PostgreSQL database schema defines users, projects, and documents, with relational integrity between entities.

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100) UNIQUE,
  password VARCHAR(100),
  role VARCHAR(20)
);

CREATE TABLE projects (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100),
  description TEXT,
  status VARCHAR(20),
  builder_id INT REFERENCES users(id),
  client_id INT REFERENCES users(id),
  budget_planned DOUBLE PRECISION,
  budget_used DOUBLE PRECISION,
  start_date DATE,
  end_date DATE
);

CREATE TABLE documents (
  id SERIAL PRIMARY KEY,
  project_id INT REFERENCES projects(id),
  document_name VARCHAR(100),
  document_type VARCHAR(50),
  uploaded_by INT REFERENCES users(id),
  upload_date DATE
);
```

Key Relationships:

- A **Builder** can manage multiple **Projects**.
- A **Client** may be linked to multiple **Projects**.
- Each **Project** can have multiple **Documents**.

8. Utilities and Helpers

Utility	Purpose
DBConnectionUtil	Manages PostgreSQL database connection using JDBC.
BudgetUtil	Calculates budget variance and determines health (Under/Over/On Track).
GanttChartUtil	Prints a visual timeline (mock Gantt chart) on the console.
StatusConstants	Central repository for project status constants (UPCOMING, IN_PROGRESS, COMPLETED).

9. Setup and Execution Guide

Prerequisites

- Java 17+
- PostgreSQL installed locally
- IntelliJ IDEA (recommended)
- SQL execution access

Steps to Run

Create Database

```
CREATE DATABASE builder_portfolio_db;
\c builder_portfolio_db;
```

1. Execute the schema provided above.

Configure Database Connection

Update the `DBConnectionUtil` file:

```
db.url=jdbc:postgresql://localhost:5432/builder_portfolio_db
db.username=postgres
db.password=your_password
```

2. **Build & Run**
    - Open project in IntelliJ.

Run main class:

```
com.builder.portfolio.Main
```

- Follow console prompts to register and login.

3. **Testing**
    - Register at least one Admin, Builder, and Client.
    - Login as each role to test menu options and workflows.
    - Add projects, documents, and view reports.

---

10. Repository Link

📎 GitHub Repository:
https://github.com/Faiq0602/BuilderPortfolioManagementSystem

---

11. Future Enhancements

- Integrate **password hashing** using BCrypt.
- Convert into **RESTful API** (Spring Boot version).
- Implement **real file uploads** with file storage path tracking.
- Add **user notification system** for project updates.
- Include **role-based access control** at DAO level.
- Extend GanttChartUtil into a GUI or web visualization.

In Summary

The **Builder Portfolio Management System** is a modular, production-style Java console application that demonstrates clear architectural layering, business logic separation, and realistic workflows.
It models how real construction management software operates — from authentication and project tracking to document metadata and budget control — making it a strong, professional-grade submission.