Builder Portfolio Management System — Code Logic Documentation

1. Project Overview

The **Builder Portfolio Management System (BPMS)** is a modular Java 17 application that enables builders, clients, and admins to manage projects end-to-end: registration/login, project creation and updates, document metadata, budget health, and timeline visualization.

The system follows a clean, enterprise-style **layered architecture (Controller → Service → DAO → Model → Util)** for maintainability and testability.
New: The service layer now supports **safe multi-user concurrency** with per-project locking, optimistic version checks, and parallel report generation.

---

2. System Architecture and Design

Layer	Package	Responsibility	Key Classes
Controller	`com.builder.portfolio.controller`	Menus, input handling, route to services	`AdminController`, `BuilderController`, `ClientController`
Service	`com.builder.portfolio.service`	Core business logic, validation, orchestration	`UserServiceImpl`, `ProjectServiceImpl`, `DocumentServiceImpl`, `ReportServiceImpl`
DAO	`com.builder.portfolio.dao`	Persistence via JDBC	`UserDAOImpl`, `ProjectDAOImpl`, `DocumentDAOImpl`
Model	`com.builder.portfolio.model`	Domain entities & DTOs	`User`, `Project`, `Document`, `BudgetReport`

Util	`com.builder.portfolio.util`	Shared helpers, constants, infra	`DBConnectionUtil`, `BudgetUtil`, `StatusConstants`, `GanttChartUtil`
Concurrency	`com.builder.portfolio.concurrent`	Locks, executors, caches	**`LockRegistry`**, **`BackgroundTaskManager`**, **`ProjectCache`**

Core Tech: Java 17 • Maven • PostgreSQL • JDBC • (Logging via SLF4J/JUL)
Concurrency Overlay: `ReadWriteLock`s per project, optimistic versioning, thread-safe caches, fixed & scheduled executors, and parallel reporting.

---

3. Login & Registration Logic

Registration

1. User selects **Register**, provides name/email/password/role.
2. `UserService.registerUser` validates uniqueness via `UserDAO.findByEmail`.
3. On success, `UserDAO.addUser` persists the record; success message is shown.

Login

1. User selects **Login**, enters email/password.
2. `UserService.login` fetches user via `UserDAO.findByEmailAndPassword`.
3. Role-based routing:
   Admin → `AdminController.showMenu()`
   Builder → `BuilderController.showMenu()`
   Client → `ClientController.showMenu()`

> **Security note:** For production, replace plaintext with strong hashing (e.g., BCrypt/Argon2).

---

4. Project Management Logic

a) Add Project

- Builder inputs details; default status to `UPCOMING` if absent.
- `ProjectService.addProject` validates and delegates to `ProjectDAO.addProject`.
- Confirmation printed on success.

b) Update Project (Status/Budget/Timeline)

- Fetch via `ProjectService.getProject(id)`.
- Edits flow through **concurrency-aware** service methods: `updateProjectStatus(...)`, `updateProjectBudget(...)`.
- If status becomes `IN_PROGRESS` or `COMPLETED`, `GanttChartUtil.printSimpleGantt(project)` renders a textual timeline.

c) View Portfolio

- Admin: all projects
- Builder: own projects
- Client: assigned projects

d) Delete Project

Ownership validated in DAO; guarded delete:

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

-

---

5. Document Management (Mock File Upload)

Intent: Store **metadata only** (no binary files) to simulate upload workflows.

Create

1. Builder selects **Add Document Metadata**.
2. `DocumentService.addDocument` validates and calls `DocumentDAO.addDocument`.

Insert:

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

VALUES (?, ?, ?, ?, ?);
```

3.

List

- `documentService.listDocumentsByProject(projectId)` prints tabular output with IDs, names, types, uploader, and date.

Why "Mock Upload"?

- Lightweight for console/Maven flow.
- Ready to evolve to real storage (local path/S3) with `file_path`/`file_url` fields and RBAC.

---

6. Budget & Timeline Tracking Logic

Budget

- `ProjectService.buildBudgetReport(project)`:
    - `BudgetUtil.calculateVariance(used, planned)`
    - `BudgetUtil.determineBudgetHealth(...)` → `UNDER` / `ON_TRACK` / `OVER`

Timeline

If status is `IN_PROGRESS` or `COMPLETED`:

```
 Design      |#######........|

Permits     |....####.......|
Build       |.......########|

Testing     |..........#####|
```

-

---

7. Concurrency & Multithreading Enhancements

7.1 Objectives

- Prevent **lost updates** and **races** under concurrent edits.
- Make heavy operations **responsive** via parallelism.
- Centralize thread pools and ensure **graceful shutdown**.

7.2 Design Summary

Area	Approach	Highlights
Per-Project Coordination	`LockRegistry` → `ReadWriteLock` per project	Read for reads; write for mutations; strict lock ordering
Stale-Write Prevention	**Optimistic versioning** on `projects`	`WHERE id=? AND version=?` then `SET version=version+1`; retries on conflict
Document Uploads	Route via `ProjectService.uploadDocument`	Reuses project write lock and logs timing
Caching	`ProjectCache` (thread-safe snapshots)	`ConcurrentHashMap`, immutable DTOs

Background	`BackgroundTaskManager`	Fixed thread pool + `ScheduledExecutorService`, graceful shutdown
Parallel Reports	`ReportServiceImpl.generatePortfolioReportParallel`	Fan-out per project (`CompletableFuture`/executor) with timeouts
Observability	Structured logs around lock waits, retries, durations	Aids profiling with JConsole/JFR

7.3 Locking & Versioning (Service Layer)

LockRegistry

```
public final class LockRegistry {

  private final ConcurrentHashMap<Long, ReadWriteLock> locks = new
ConcurrentHashMap<>();

  public ReadWriteLock forProject(long projectId) {

    return locks.computeIfAbsent(projectId, id -> new
ReentrantReadWriteLock());

  }

}
```

Optimistic Update (DAO)

```
-- Status update with version check

UPDATE projects

SET status = ?, version = version + 1

WHERE id = ? AND version = ?;
```

Service Retry Skeleton

```
boolean updated = false;

for (int attempt = 1; attempt <= 3 && !updated; attempt++) {

  int rows = projectDao.updateStatusWithVersion(id, newStatus,
expectedVersion);

  updated = rows == 1;

  if (!updated) { // version changed by someone else

    project = projectDao.findById(id);          // refresh

    expectedVersion = project.getVersion();     // bump expected

  }

}

if (!updated) throw new ConcurrentModificationException("Project modified
concurrently");
```

Write-Lock Guard for Mutations

```
var lock = lockRegistry.forProject(projectId).writeLock();

lock.lock();

try {

  // validate + DAO update with optimistic versioning

} finally {

  lock.unlock();

}
```

7.4 Document Upload Concurrency

- `ProjectService.uploadDocument` executes under the **project write lock**; batches can be serialized safely.
- Duration logged to help spot hotspots during concurrent metadata writes.

7.5 Thread-Safe Caching

- `ProjectCache` stores **immutable** snapshots keyed by project ID.
- Refresh on successful updates; controllers read through cache for fast list views.

7.6 Background Task Management

```
public final class BackgroundTaskManager implements AutoCloseable {

  private final ExecutorService workers =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

  private final ScheduledExecutorService scheduler =
Executors.newSingleThreadScheduledExecutor();



  public <T> CompletableFuture<T> submit(Callable<T> task) { return
CompletableFuture.supplyAsync(() -> {

      try { return task.call(); } catch (Exception e) { throw new
CompletionException(e); }

  }, workers); }



  public ScheduledFuture<?> schedule(Runnable r, long period, TimeUnit
unit) {

    return scheduler.scheduleAtFixedRate(r, period, period, unit);

  }



  @Override public void close() {

    workers.shutdown(); scheduler.shutdown();
    // awaitTermination(...) + fallback shutdownNow()
  }
}
```

7.7 Parallel Portfolio Reports

- `ReportServiceImpl.generatePortfolioReportParallel` partitions work by project and aggregates with `CompletableFuture.allOf(...)`.
- Timeouts protect the UI; slow projects are logged and skipped (configurable).

7.8 Testing the Concurrency Layer

- **`ProjectConcurrencyTest`**: parallel status/budget mutations; asserts single-writer visibility and version increments.
- **`DocumentUploadConcurrencyTest`**: concurrent metadata writes using the same project; verifies serialization and ordering.
- **`ParallelReportPerfTest`**: compares sequential vs. parallel run time and validates deterministic results.

7.9 Console Concurrency Demo

- Builder menu item triggers **asynchronous mock jobs** (`Thread.sleep` to simulate work), printing thread names and progress so users see background execution in real time.

> **Deadlock Discipline:** Always acquire project locks in **ascending projectId** order when multiple projects are touched in one operation.

---

8. Database Schema

```
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,

  -- NEW: optimistic concurrency control

  version INT NOT NULL DEFAULT 0

);


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

);
```

Migration note (existing DB):

```
ALTER TABLE projects ADD COLUMN IF NOT EXISTS version INT NOT NULL DEFAULT 0;
```

---

9. Utilities and Helpers

Utility	Purpose
`DBConnectionUtil`	PostgreSQL connectivity via JDBC
`BudgetUtil`	Variance & budget-health calculation
`GanttChartUtil`	Console-based timeline rendering
`StatusConstants`	`UPCOMING`, `IN_PROGRESS`, `COMPLETED`
`LockRegistry`	Per-project `ReadWriteLock` factory
`BackgroundTaskManager`	Fixed & scheduled executors with graceful shutdown

`ProjectCache`	Thread-safe, immutable DTO snapshots

---

10. Setup and Execution Guide

Prerequisites

- Java 17+
- PostgreSQL (local)
- IntelliJ IDEA / Maven

Database

```
CREATE DATABASE builder_portfolio_db;

\c builder_portfolio_db;

-- Execute the schema (incl. projects.version)
```

Configuration (`DBConnectionUtil` / properties)

```
db.url=jdbc:postgresql://localhost:5432/builder_portfolio_db

db.username=postgres

db.password=your_password
```

Build & Run

```
mvn clean install

# Run from IntelliJ (main class): com.builder.portfolio.Main
```

Concurrency Demo (console)

- From Builder menu, choose **Concurrency Demo** to start async jobs.
- Observe interleaved progress and thread names in console.

Optional Monitoring (JDK-only)

```
jconsole

# Or capture quick stats:

jps -l

jstat -gcutil <PID> 1000 5

jcmd  <PID> GC.heap_info
```

---

11. Repository Link

- **GitHub:** https://github.com/Faiq0602/BuilderPortfolioManagementSystem

---

12. Future Enhancements

- Secure password storage (BCrypt/Argon2).
- Spring Boot REST API; JPA with `@Version` for OCC.
- Real file uploads (local/S3) with RBAC and audit logging.
- Distributed locks for clustered deployments (Redis/DB-based).
- Metrics & tracing (Micrometer/Prometheus/OpenTelemetry).
- Caching library (Caffeine) with size/TTL policies.

---

13. Summary

BPMS now combines **clear layered design** with **robust concurrency controls**:

- **LockRegistry** prevents write races per project.
- **Optimistic versioning** stops stale writes and enables safe retries.
- **ProjectCache** and **BackgroundTaskManager** deliver responsive reads and predictable background work.
- **Parallel report generation** shortens heavy workloads.
- A focused **test suite** validates correctness and performance under load.

This upgrade makes BPMS safer for **multi-user** scenarios while preserving a simple operational model for local development and evaluation.