Issue 1 – JobExecutor does everything with `if/else`

Before

```
public void executeJob(Job job) {
    System.out.printf("[NaiveExecutor] Starting job %s (%s) requested by %s%n",
            job.getName(), job.getType(),
            job.getRequestedBy() == null ? "unknown" : job.getRequestedBy().getName());
    // Acquire connection directly (no pool reuse)
    Connection c = cm.createConnection();
    try {
        if ("EMAIL".equals(job.getType())) {
            executeEmailJob(job, c);
        } else if ("DATA".equals(job.getType())) {
            executeDataJob(job, c);
        } else if ("REPORT".equals(job.getType())) {
            executeReportJob(job, c);
        } else {
            System.out.println("Unknown job type - nothing done.");
        }
    } finally {
        cm.closeConnection(c);
        System.out.printf("[NaiveExecutor] Finished job %s%n", job.getName());
    }
}

private void executeEmailJob(Job job, Connection c) {
    System.out.println("[EmailJob] Preparing to send email using config: " + job.getConfig());
    c.executeQuery("INSERT INTO email_sent (job, status) VALUES ('" + job.getId() + "', 'SENT')");
    // naive: no error handling, no retries
}

private void executeDataJob(Job job, Connection c) {
    System.out.println("[DataJob] Reading & transforming data using config: " + job.getConfig());
    c.executeQuery("SELECT * FROM source_table WHERE job_id = '" + job.getId() + "'");
    c.executeQuery("INSERT INTO processed_results (job_id) VALUES ('" + job.getId() + "')");
}

private void executeReportJob(Job job, Connection c) {
    System.out.println("[ReportJob] Generating report (" + job.getName() + ") using config: " + job.getConf
    c.executeQuery("SELECT * FROM report_source WHERE report = '" + job.getName() + "'");
    c.executeQuery("INSERT INTO generated_reports (job_id, path) VALUES ('" + job.getId() + "', '/reports/"
}
```

Why this is wrong:

Violates (OCP):

Every time a new job type is added, this method must be modified another `else if`. The class is not closed for modification.

Violates (SRP):

`JobExecutor` is responsible for:

1. Deciding which behavior to run (by checking `job.getType()`), and
2. Executing the actual business logic for each job type in three different private methods.

These are two separate responsibilities in one class.

Tight coupling to job types:
The executor must know all existing job types (`"EMAIL"`, `"DATA"`, `"REPORT"`) and how to execute each one.

How this issue appears in the system?

Adding a new job type (e.g., `"NOTIFICATION"`) requires:

1. Changing `JobExecutor.executeJob()` and adding a new `else if`.
2. Adding another `executeNotificationJob(...)` method.
3. Re-testing the whole executor because all job types share the same method and connection logic.

After refactoring – Strategy + JobType enum

We refactored to use a Strategy pattern and a type-safe enum.

```
package edu.najah.cap.advance.assignments.assignment1.job;

public enum JobType {  30 usages  raghadabdelhaq55
    EMAIL,  4 usages
    DATA_PROCESSING,  4 usages
    REPORT  5 usages
}
```

New `JobStrategy` + concrete strategies

```java
public interface JobStrategy {  8 usages  3 implementations  raghadabdelhaq55
    void execute(Job job, Connection connection);  1 usage  3 implementations  raghadabdelhaq55
}
```

```java
public class EmailJobStrategy implements JobStrategy {  1 usage  raghadabdelhaq55

    @Override  1 usage  raghadabdelhaq55
    public void execute(Job job, Connection connection) {
        System.out.println("[EmailStrategy] Preparing to send email using config: " + job.getConfig());
        connection.execute( query: "INSERT INTO email_sent (job, status) VALUES ('" + job.getId() + "', 'SENT')");
        System.out.println("[EmailStrategy] Email sent successfully");
    }
}
```

```java
public class DataProcessingStrategy implements JobStrategy {  1 usage  raghadabdelhaq55

    @Override  1 usage  raghadabdelhaq55
    public void execute(Job job, Connection connection) {
        System.out.println("[DataStrategy] Starting data processing with config: " + job.getConfig());
        connection.execute( query: "UPDATE data_jobs SET status='DONE' WHERE job_id='" + job.getId() + "'");
        System.out.println("[DataStrategy] Data processing completed successfully");
    }
}
```

```java
public class ReportGenerationStrategy implements JobStrategy {  1 usage  raghadabdelhaq55

    @Override  1 usage  raghadabdelhaq55
    public void execute(Job job, Connection connection) {
        System.out.println("[ReportStrategy] Generating report with config: " + job.getConfig());
        connection.execute( query: "INSERT INTO reports (job_id, status) VALUES ('" + job.getId() + "', 'GENERATED')
        System.out.println("[ReportStrategy] Report generated successfully");
    }
}
```

Strategy registry (no switch/if)

```java
public final class StrategyRegistry {  2 usages  raghadabdelhaq55

    private static final Map<JobType, JobStrategy> STRATEGIES =  4 usages
            new EnumMap<>(JobType.class);

    static {
        STRATEGIES.put(JobType.EMAIL, new EmailJobStrategy());
        STRATEGIES.put(JobType.DATA_PROCESSING, new DataProcessingStrategy());
        STRATEGIES.put(JobType.REPORT, new ReportGenerationStrategy());
    }

    private StrategyRegistry() { }  no usages  raghadabdelhaq55

    public static JobStrategy getStrategy(JobType type) {  1 usage  raghadabdelhaq55
        JobStrategy strategy = STRATEGIES.get(type);
        if (strategy == null) {
            throw new IllegalArgumentException("No strategy for type " + type);
        }
        return strategy;
    }
}
```

Refactored Job uses JobType + strategy

Result:
There is no `if/else` on job type anymore.
To add a new job type, we add:

1- a new JobType constant

2- a new `JobStrategy` implementation

3- a single line in `StrategyRegistry`

Issue 2 – `JobExecutor` manages connections directly, no pooling

Before

Same class as above: `JobExecutor`.

```
public class JobExecutor {
    private final ConnectionManager cm;

    public JobExecutor(ConnectionManager cm) {
        this.cm = cm;
    }
```

Why this is wrong:

Tight coupling :
`JobExecutor` must know how to create and close connections.
Business logic is mixed with resource management.

No pooling / reuse:
Every job execution creates a new `Connection`.
There is no resource sharing, which is inefficient under load.

How this issue appears in the system?

1- Under higher load, many short-lived connections are created and closed.

2- It is hard to change the connection strategy (e.g., a pool, different DB, mock connections) without editing business code.

After refactoring – `ConnectionPool` + `JobManager`

We created a dedicated ConnectionPool and a JobManager (proxy/controller) that manages connections.

New `ConnectionPool`

```java
public class ConnectionPool {  6 usages  raghadabdelhaq55
    private static final int MAX_CONNECTIONS = 10;  1 usage
    private static final ConnectionPool INSTANCE = new ConnectionPool();  1 usage

    private final List<Connection> connections = new ArrayList<>();  3 usages
    private int currentIndex = 0;  3 usages

    private ConnectionPool() {  1 usage  raghadabdelhaq55
        for (int i = 1; i <= MAX_CONNECTIONS; i++) {
            connections.add(new Connection( id: "Conn-" + i));
        }
    }

    public static ConnectionPool getInstance() {  1 usage  raghadabdelhaq55
        return INSTANCE;
    }

    public synchronized Connection acquire() {  1 usage  raghadabdelhaq55
        Connection connection = connections.get(currentIndex);
        currentIndex = (currentIndex + 1) % connections.size();
        System.out.println("[ConnectionPool] Acquired connection: " + connection.getId());
        return connection;
    }

    public synchronized void release(Connection connection) {  1 usage  raghadabdelhaq55
        if (connection != null) {
            System.out.println("[ConnectionPool] Released connection: " + connection.getId());
        }
    }
```


New `JobManager` orchestrates execution

```java
public class JobManager {  3 usages  raghadabdelhaq55 *

    private final ConnectionPool connectionPool = ConnectionPool.getInstance();  2 usages
    private final Logger logger = new Logger();  11 usages
    private final PermissionService permissionService = new PermissionService();  1 usage

    public void execute(Job job, User user) {  1 usage  raghadabdelhaq55 *

        // Permission
        if (!permissionService.canExecute(user, job)) {
            String msg = "[Proxy] Permission denied: User '" + user.getName()
                    + "' does not have permission for job type '" + job.getType() + "'";
            logger.log(msg);
            return;
        }

        logger.log("[Proxy] Permission validated for user '" + user.getName()
                + "' on job type '" + job.getType() + "'");

        logger.log("[Proxy] ===== JOB START =====");
        logger.log("[Proxy] Job ID: " + job.getId());
        logger.log("[Proxy] Job Name: " + job.getName());
        logger.log("[Proxy] Job Type: " + job.getType());
        logger.log("[Proxy] Requested By: " + user.getName());
```

```java
Connection connection = connectionPool.acquire();
long start = System.currentTimeMillis();

try {
    logger.log("[RefactoredExecutor] Executing job " + job.getName()
            + " (" + job.getType() + ")");
    job.executeWithStrategy(connection);
} finally {
    long end = System.currentTimeMillis();
    connectionPool.release(connection);
    logger.log("[Proxy] ===== JOB COMPLETE =====");
    logger.log("[Proxy] Job '" + job.getName()
            + "' completed successfully");
    logger.log("[Proxy] Execution time: " + (end - start) + " ms");
}
```

Result:

`JobExecutor` is no longer needed.

Connection lifecycle and logging are centralized in `JobManager`, and connections are reused through `ConnectionPool`.

Issue 3 – `TemplateManager` rebuilds heavy templates every time (duplication)

Before

Class: `templates.TemplateManager`

```
public class TemplateManager {

    public HeavyTemplate buildEmailJobTemplate(String templateName, String config) {
        var templateBody =  simulateHeavyLoad("EmailTemplate:"+templateName);
        HeavyTemplate t = new HeavyTemplate("EMAIL", templateName, config, templateBody);
        System.out.println("Built Email template (heavy): " + templateName);
        return t;
    }

    public HeavyTemplate buildDataProcessingTemplate(String templateName, String config) {
        var templateBody = simulateHeavyLoad("DataTemplate:"+templateName);
        HeavyTemplate t = new HeavyTemplate("DATA", templateName, config, templateBody);
        System.out.println("Built DataProcessing template (heavy): " + templateName);
        return t;
    }

    public HeavyTemplate buildReportJobTemplate(String templateName, String config) {
        var templateBody = simulateHeavyLoad("ReportTemplate:"+templateName);
        HeavyTemplate t = new HeavyTemplate("REPORT", templateName, config, templateBody);
        System.out.println("Built Report template (heavy): " + templateName);
        return t;
    }

    private String simulateHeavyLoad(String msg) {
        System.out.println("Simulating heavy template creation for: " + msg);
        try { Thread.sleep(3000); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }
        return "Large template";
    }
}
```

and `HeavyTemplate.createJobInstance()`:

```
// naive: creates a new Job from scratch (no clone/prototype)
public Job createJobInstance() {
    String id = templateBody + " _ " + type + "-" + System.currentTimeMillis();
    return new Job(id, type, name, config);
}
```

Why this is wrong:

Duplication:
Three methods (`buildEmailJobTemplate`, `buildDataProcessingJobTemplate`, `buildReportJobTemplate`) have almost identical logic.

No reuse (performance issue):
`simulateHeavyLoad` runs for every job creation, even if we build the same template many times.

Tight coupling to string types:
Types `"EMAIL"`, `"DATA"`, `"REPORT"` are hard-coded again.

How this issue appears in the system?

- In `MainApp`, each call like

  `templateManager.buildReportJobTemplate(...).createJobInstance();`
  triggers a 3-second delay and prints
  "Simulating heavy template creation for…".

- Creating several jobs based on the same idea (e.g., monthly report) repeatedly performs the heavy load.

After refactoring – `JobTemplate` + `JobFactory` (prototype-style)

We replaced the naive template builder with simple reusable templates stored in a factory.

New JobTemplate

```java
public class JobTemplate {  5 usages  raghadabdelhaq55

    private final JobType type;  2 usages
    private final String name;  2 usages
    private final String defaultConfig;  2 usages

    public JobTemplate(JobType type, String name, String defaultConfig) {  3 usages  raghadabdelhaq55
        this.type = type;
        this.name = name;
        this.defaultConfig = defaultConfig;
    }

    public Job createJob() {  1 usage  raghadabdelhaq55
        String id = UUID.randomUUID().toString();
        return new Job(id, type, name, defaultConfig);
    }
}
```

New JobFactory

```java
public final class JobFactory {  2 usages  raghadabdelhaq55
    private static final Map<JobType, JobTemplate> TEMPLATES =  4 usages
            new EnumMap<>(JobType.class);

    static {
        // Standard Email / Data / Report :نفس فكرة الصورة
        TEMPLATES.put(JobType.EMAIL,
                new JobTemplate(
                        JobType.EMAIL,
                        name: "Standard Email",
                        defaultConfig: "format=HTML;priority=normal"
                ));

        TEMPLATES.put(JobType.DATA_PROCESSING,
                new JobTemplate(
                        JobType.DATA_PROCESSING,
                        name: "Daily Data Processing",
                        defaultConfig: "source=DB;mode=batch"
                ));

        TEMPLATES.put(JobType.REPORT,
                new JobTemplate(
                        JobType.REPORT,
                        name: "Monthly Report",
                        defaultConfig: "template=FINANCE;format=PDF"
                ));
    }
```

```java
private JobFactory() { }  no usages  raghadabdelhaq55

public static Job createDefaultJob(JobType type) {  1 usage  raghadabdelhaq55
    JobTemplate template = TEMPLATES.get(type);
    if (template == null) {
        throw new IllegalArgumentException("No template for type " + type);
    }
    return template.createJob();
}
```

Result:

1- All default job configurations are in one place.

2- No per-call heavy building; we reuse templates in the factory style.

3- Adding a new template only requires one new `JobTemplate` entry.

Issue 4 – Weak modeling of Job + naive permissions

Before

Job

```
public class Job {
    private String id;
    private String type; // "EMAIL","DATA","REPORT"
    private String name;
    private String config; // naive serialized config
    private User requestedBy;

    public Job(String id, String type, String name, String config) {
        this.id = id;
        this.type = type;
        this.name = name;
        this.config = config;
    }
    public String getId() { return id; }
    public String getType() { return type; }
    public String getName() { return name; }
    public String getConfig() { return config; }
    public void setRequestedBy(User u) { this.requestedBy = u; }
    public User getRequestedBy() { return requestedBy; }
}
```

User

```
public class User {
    private final String name;
    private final List<String> permissions; // naive permission labels

    public User(String name, List<String> permissions) {
        this.name = name;
        this.permissions = permissions == null ? new ArrayList<>() : permissions;
    }

    public String getName() { return name; }
    public boolean hasPermission(String perm) {
        return permissions.contains(perm);
    }
}
```

> **Note:** `JobExecutor` never actually calls `hasPermission`, so permissions are not enforced.

Why this is wrong:

1- `Job.type` is a plain `String` → no compile-time safety.

2- Permissions are also raw strings; there is no clear relation between job types and permissions.

3- Permission logic is not integrated into the execution flow at all.

How this issue appears in the system?

1- A typo like `"EMIAL"` will compile and only fail at runtime.

2- Any user can execute any job because `JobExecutor` never checks `User.hasPermission`.

3- Security rules are unclear and scattered.

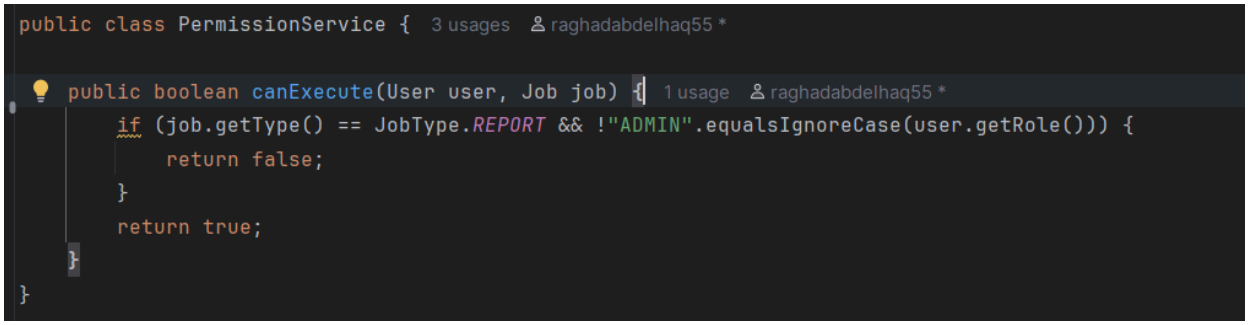
After refactoring – JobType + PermissionService + JobManager

We already introduced JobType and updated Job (see Issue 1).
Now we also add an explicit PermissionService and integrate it in JobManager.

New simple User

```java
public class User {  6 usages  raghadabdelhaq55

    private final String id;  2 usages
    private final String name;  2 usages
    private final String role; // ADMIN / USER  2 usages

    public User(String id, String name, String role) {  1 usage  raghadabdelhaq55
        this.id = id;
        this.name = name;
        this.role = role;
    }

    public String getId() {  no usages  raghadabdelhaq55
        return id;
    }

    public String getName() {  3 usages  raghadabdelhaq55
        return name;
    }

    public String getRole() {  1 usage  raghadabdelhaq55
        return role;
    }
}
```

PermissionService



Integrated in `JobManager.execute` (see earlier):

```
if (!permissionService.canExecute(user, job)) {
    logger.log("[Proxy] Permission denied: User '" +
user.getName() +
            "' does not have permission for job type '" +
job.getType() + "'");
    return;
}
```

Result:
Permissions are now centralized and explicit, and every job execution goes through `JobManager`, which enforces these rules.

Issue 5 – `MainApp` is hard-coded and not interactive

Before

Class: `MainApp`

```
11  public class MainApp {
12      public static void main(String[] args) {
13          System.out.println("=== TMPS Naive Starter App ===");
14
15          // create a naive ConnectionManager (not a pool)
16          ConnectionManager connManager = new ConnectionManager();
17
18          // naive TemplateManager (builds templates from scratch each time)
19          TemplateManager templateManager = new TemplateManager();
20
21          // naive executor (does everything with if/else)
22          JobExecutor executor = new JobExecutor(connManager);
23
24          User alice = new User("alice", Arrays.asList("EMAIL", "REPORT")); // incomplete permissions
25
26          // Demo: create a report job from template and execute
27          System.out.println("\n--- Create Report Job from template (naive build) ---");
28
29          //TODO problem 1: each time want to create job, it takes time to load and create job which includes cre
30          Job reportJob = templateManager.buildReportJobTemplate("MonthlyReport", "format=PDF;brand=TaskMaster").
31          reportJob.setRequestedBy(alice);
32
33          System.out.println("\n--- Execute job (naive executor) ---");
34          executor.executeJob(reportJob);
35
36          Job reportJob2 = templateManager.buildEmailJobTemplate("Monthly email Report", "format=PDF;all=true").c
37          reportJob2.setRequestedBy(alice);
38
39          System.out.println("\n--- Execute job (naive executor) ---");
40          executor.executeJob(reportJob2);
41      }
```

Why this is wrong:

1- Tightly coupled to naive implementations (`ConnectionManager`, `TemplateManager`, `JobExecutor`).

2- Uses a single hard-coded user and two hard-coded jobs.

3- No way for the user to select job type or role at runtime.

How this issue appears in the system?

1- To test different jobs or users, we must edit the code and recompile.

2- We cannot demonstrate permissions (e.g., denied REPORT for a normal user) without changing source code.

After refactoring – Interactive Main using `JobFactory` + `JobManager`

Refactored `MainApp`

public class MainApp {

```
public class MainApp {  raghadabdelhaq55
    public static void main(String[] args) {  raghadabdelhaq55

        while (true) {
            System.out.println();
            System.out.println("Choose job type:");
            System.out.println("1 - EMAIL");
            System.out.println("2 - DATA PROCESSING");
            System.out.println("3 - REPORT");
            System.out.println("0 - EXIT");
            System.out.print("Your choice: ");

            String line = scanner.nextLine().trim();
            int choice;
            try {
                choice = Integer.parseInt(line);
            } catch (NumberFormatException e) {
                System.out.println("Invalid input, try again.");
                continue;
            }

            if (choice == 0) {
                break;
            }

            JobType type;
            switch (choice) {
                case 1:
                    type = JobType.EMAIL;
                    break;
                case 2:
```

```
public class MainApp {  raghadabdelhaq55
    public static void main(String[] args) {  raghadabdelhaq55
            }

            JobType type;
            switch (choice) {
                case 1:
                    type = JobType.EMAIL;
                    break;
                case 2:
                    type = JobType.DATA_PROCESSING;
                    break;
                case 3:
                    type = JobType.REPORT;
                    break;
                default:
                    System.out.println("Invalid choice, try again.");
                    continue;
            }

            Job job = JobFactory.createDefaultJob(type);
            manager.execute(job, user);
        }

        System.out.println("\n=== Application finished ===");
        scanner.close();
    }
}
```

Result:

1- Main app is interactive, not hard-coded.

2- It uses the refactored components: `JobFactory`, `JobManager`, `ConnectionPool`, `JobType`, and strategies.

3- Demonstrates both successful executions and permission denials with clear logs.

Decision Questions

Why each design pattern was chosen ?

Strategy Pattern

The old code used big `if/else` statements to decide how each job should run.
With Strategy, every job type has its own class, so we don't touch the main code when adding a new job.
This makes the system easier to extend and follow.

We used JobTemplate + JobFactory

The original TemplateManager repeated the same code three times.
The new Factory stores ready templates and creates jobs quickly without heavy rebuilding.
This reduces duplication and makes job creation simple and consistent.

ConnectionPool

Before, every job created a new connection.
This is slow and doesn't scale.
With a ConnectionPool, we reuse existing connections, which makes the system faster and more efficient.

Why we added a JobManager

Responsibilities were mixed everywhere (logging, permissions, connection handling, job execution).
JobManager now handles all these steps in one place.
It checks permissions, gets a connection, runs the strategy, and logs everything.

How the patterns work together?

1- The user chooses a job type

2- JobFactory creates the job

3- JobManager controls the whole process

4- The job runs using its Strategy

5- The connection is returned to the Pool

6- Logs show what happened

Everything flows in a clear, simple pipeline.

How the new design improves the system?

1- We reuse connections and can add new job types easily

2- More flexible Each part is separate, so changes don't break the system

3- More maintainable No big `if/else`, no repeated code, and each class has one clear job

The old system was messy and tightly connected, The new one is organized, clean, and easy to expand.