WIPRO NGA Program – NMS Batch

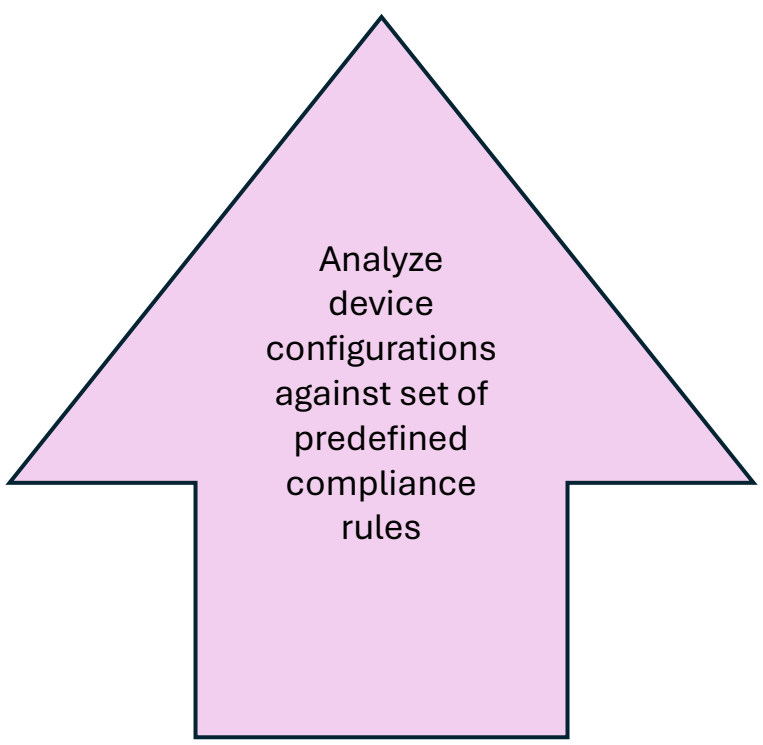
Capstone Project Presentation – 24 July 2024

Compliance and Reporting Microservice

Presented by – NMS_24NAG0969_TeamD_Presentation

INTRODUCTION

- PURPOSE: Ensure device configurations meet compliance standards
- KEY COMPONENTS:
  - Compliance Rule Engine
  - Reporting Service



What is compliance and reporting microservice?

- A compliance and reporting microservice is a small, self-contained service that specifically handles tasks related to adhering to regulations and generating reports.

- It essentially breaks down compliance and reporting functionalities into bite-sized, manageable pieces within a larger software application.

Breakdown of key aspects

- **Compliance:** This microservice ensures the application meets industry standards and regulations. This might involve tracking specific data points, following data security protocols, or generating reports required by regulators.

- **Reporting:** This microservice focuses on generating various reports that demonstrate compliance or provide insights into the application's activity. These reports can be for internal use or submitted to external regulatory bodies.

Benefits of using a compliance and reporting microservice

- **Increased Efficiency:** By isolating compliance and reporting functions, developers can focus on core functionalities without getting bogged down by regulatory concerns.

- **Improved Scalability:** Microservices are inherently scalable, allowing you to easily scale up the compliance and reporting service to handle increased reporting needs.

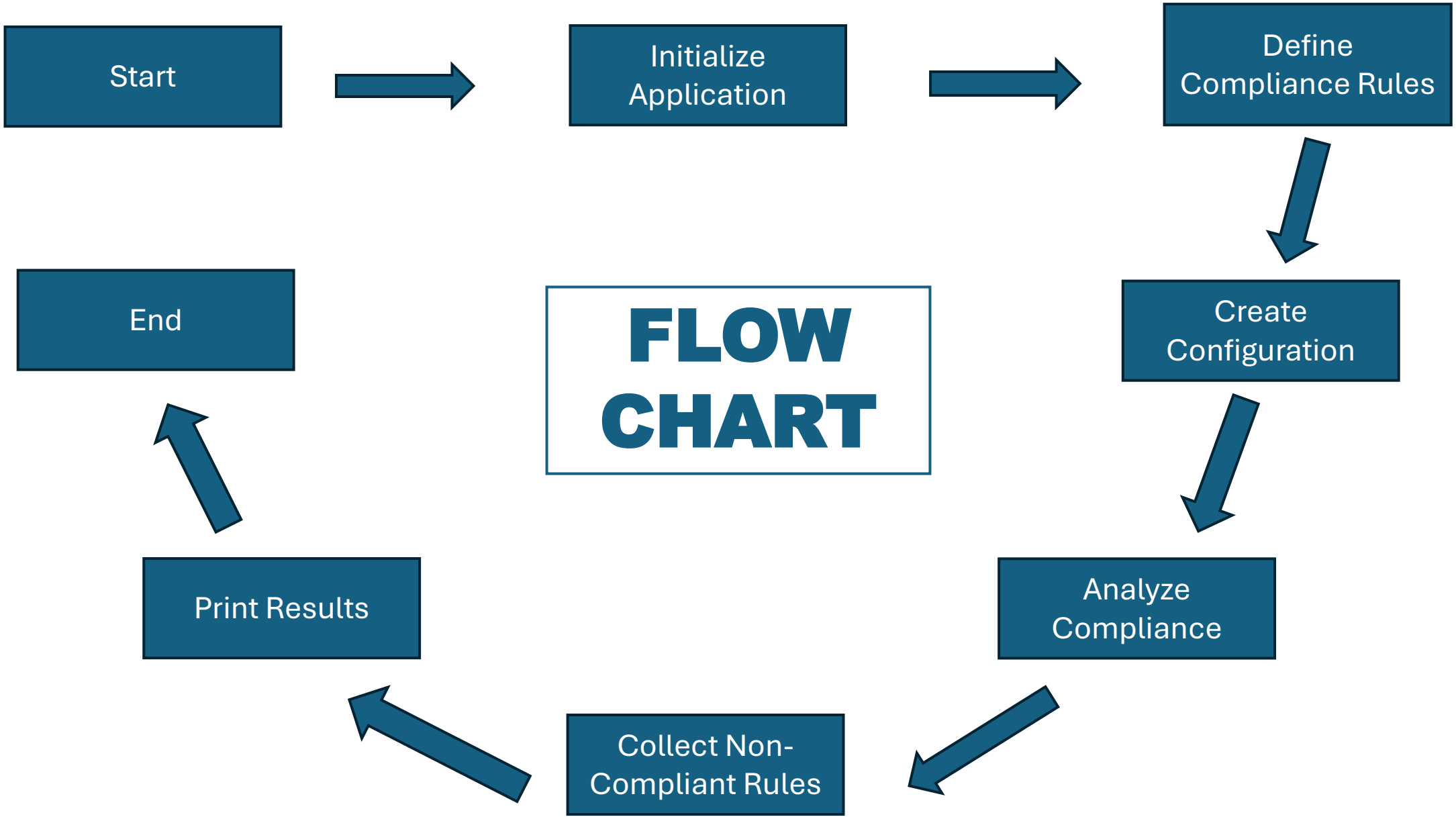
- **Enhanced Maintainability:** Since it's a dedicated service, maintaining compliance and reporting logic becomes simpler.

Use Case: Check Device Compliance

- **Pre conditions:**
  - Compliance rules are defined.
  - Device configuration is available.

- **Main Flow:**
  - The administrator inputs compliance rules into the system.
  - The system stores these rules in the Compliance Database.
  - Device configuration details are provided to the system.
  - The system creates a configuration object from these details.
  - The system evaluates the configuration against the stored compliance rules.
  - The system generates a report based on the evaluation results.

- **Postconditions**:
  - Compliance report is generated, highlighting compliant and non-compliant rules.



Flow Chart Explanation

- Start: The process begins.

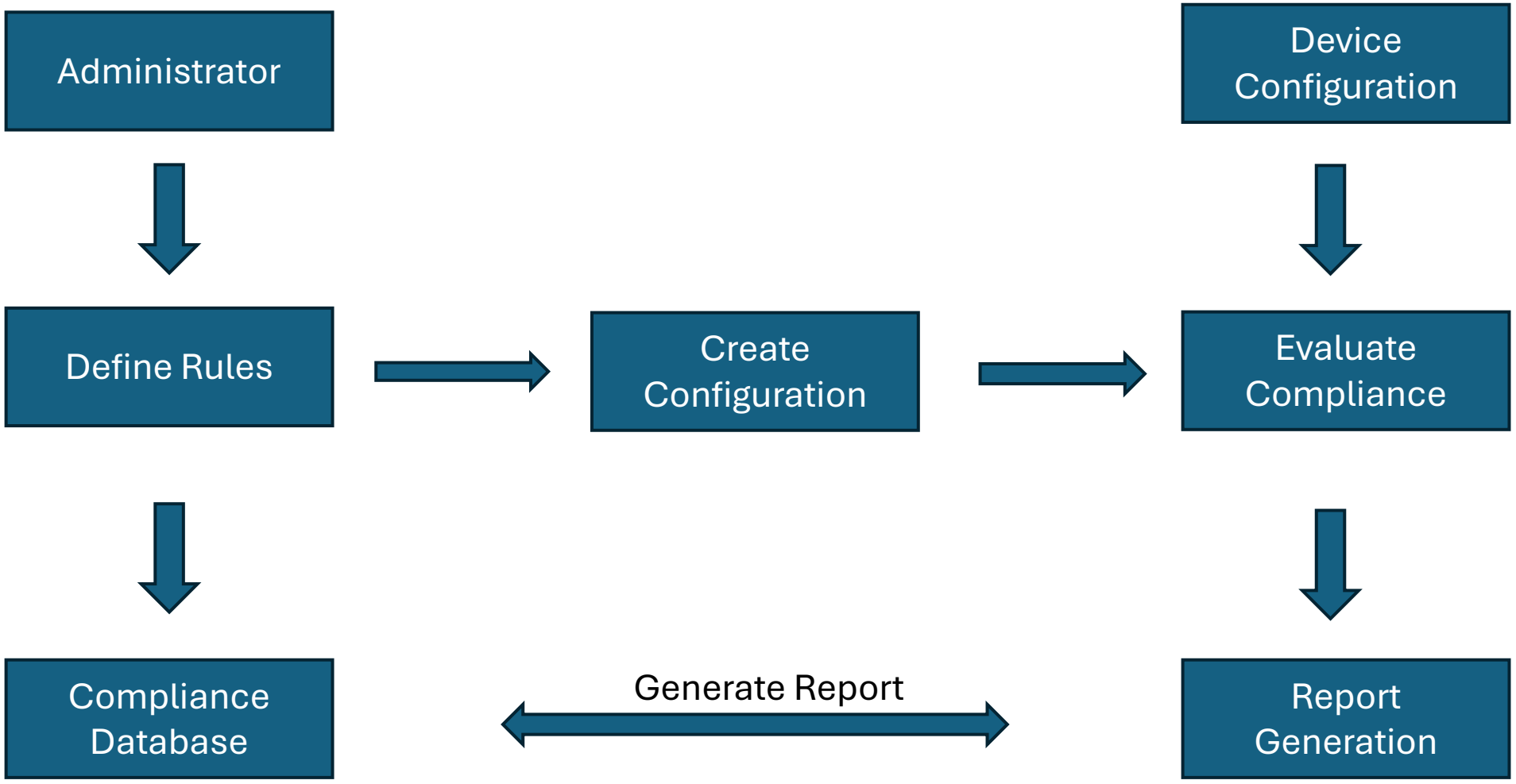
- Initialize the Application: This step likely involves setting up the application environment, loading necessary resources, and getting it ready for analysis.

- Define Compliance Rules: Here, the system establishes the criteria the application needs to follow. This could involve loading predefined rules or allowing for user-defined checks.

- Analyze Compliance: The application itself or its configuration is evaluated against the defined compliance rules. This step performs the actual comparison.

- Results: The flowchart ends by displaying the results.
  This might involve:
  a. Pass: If the application complies with all rules, a success message is shown.
  b. Fail: If any non-compliant rules are found, the specific violations are displayed. (The flowchart doesn't explicitly show this branch, but it's implied.)

DATA FLOW CHART



Data Flow Chart Explanation

- Define Rules:
  Input: Administrator inputs compliance rules.
  Output: Rules are stored in the Compliance Database.

- Create Configuration:
  Input: Device configuration details.
  Output: Configuration object created.

- Evaluate Compliance:
  Input: Configuration object and compliance rules.
  Output: List of non-compliant rules (if any).

- Generate Report:
  Input: Compliance evaluation results.
  Output: Compliance report (e.g., PDF).

INITIALIZATION

- Code Snippet

```java
package com.team4.compliance;

import java.util.List;

public class ComplianceRuleEngineApplication {

    public static void main(String[] args) {
        // Define a compliance rule
        ComplianceRule rule = new ComplianceRule("rule1", "Device ID must be device1", config -> config.getDeviceId().equals("device1"));

        // Create a compliance service with the defined rule
        ComplianceService service = new ComplianceService(List.of(rule));

        // Create a configuration to check compliance
        Configuration config = new Configuration("device1", "{...}");

        // Analyze the configuration against the compliance rules
        List<String> result = service.analyzeCompliance(config);

        // Print the results
        if (result.isEmpty()) {
            System.out.println("All rules are compliant.");
        } else {
            System.out.println("Non-compliant rules: " + result);
        }
    }
}
```

COMPLIANCE RULE

- Purpose: Define and evaluate compliance rules
- Core Class: ‘ComplianceRule’
- Attributes: ruleId, description, rule (Predicate)
- Methods: Constructor, evaluate, getters and setters

➢ Code Snippet:

```java
package com.team4.compliance;

import java.util.function.Predicate;

public class ComplianceRule {
    private String ruleId;
    private String description;
    private Predicate<Configuration> rule;

    public ComplianceRule(String ruleId, String description, Predicate<Configuration> rule) {
        this.ruleId = ruleId;
        this.description = description;
        this.rule = rule;
    }

    public boolean evaluate(Configuration config) {
        return rule.test(config);
    }

```

Explanation of Compliance Rule Java File

1. **Compliance Rule Representation**: This code defines a Compliance Rule class in Java to represent a single compliance rule.

2. **Rule Components**: Each Compliance Rule has three key components:
   - ruleId: A unique identifier for the rule.
   - description: A human-readable explanation of the rule's purpose.
   - rule: A Predicate<Configuration> object defining the actual compliance check logic.

3. **Predicate for Rule Logic**: The Predicate<Configuration> interface allows for defining a custom function that takes a Configuration object and returns true if the configuration complies with the rule, false otherwise.

4. **Evaluating Compliance**: The evaluate(Configuration config) method takes a Configuration object and uses the internal rule predicate to assess if the configuration complies with the defined rule.

5. **Getters and Setters**: Standard getters and setters are provided for accessing and modifying the ruleId, description, and rule properties.

EXAMPLE COMPLIANCE RULE

- Rule : Device Id must be ‘device1’

- Code Snippet:

```
6
7⊖    public static void main(String[] args) {
8            ComplianceRule rule = new ComplianceRule("rule1", "Device ID must be device1",
9                    config -> config.getDeviceId().equals("device1"));
0
```

CONFIGURATION ANALYSIS

- Purpose: Analyze configurations against compliance rules
- Core Class: 'ComplianceService'
- Attributes: rules (List of ComplianceRule)
- Methods: Constructor, analyzeCompliance

➢ Code Snippet:

```
public class ComplianceService {
    private List<ComplianceRule> rules;

    public ComplianceService(List<ComplianceRule> rules) {
        this.rules = rules;
    }

    public List<String> analyzeCompliance(Configuration config) {
        List<String> nonCompliantRules = new ArrayList<>();
        for (ComplianceRule rule : rules) {
            if (!rule.evaluate(config)) {
                nonCompliantRules.add(rule.getDescription());
            }
        }
        return nonCompliantRules;
    }
```

Explanation Compliance Service Java File

1. **Rule Storage**: The class has a private member rules which is a List<ComplianceRule>. This list stores the collection of ComplianceRule objects used for compliance checks.
2. **Constructor:** The constructor takes a List<ComplianceRule> argument during object creation. This allows the service to be initialized with the specific set of rules it needs to enforce.

3. **Compliance Analysis Method:** The analyzeCompliance(Configuration config) method is responsible for evaluating the provided configuration against the defined rules.
4. **Iterating Through Rules:** The method iterates through each ComplianceRule in the internal rules list.

5. **Evaluating and Collecting Non-Compliance:** For each rule, it calls the evaluate(config) method of the ComplianceRule. If the evaluation returns false (meaning the configuration is non-compliant with that rule), the description of the non-compliant rule (obtained using rule.getDescription()) is added to a List<String> named nonCompliantRules.

EXAMPLE CONFIGURATION ANALYSIS

- Configuration Class:

```
public class Configuration {
    private String deviceId;
    private String settings;

    public Configuration(String deviceId, String settings) {
        this.deviceId = deviceId;
        this.settings = settings;
    }
```

- Configuration Analysis example:

```
config -> config.getDeviceId().equals( device1 ));

ComplianceService service = new ComplianceService(List.of(rule));
Configuration config = new Configuration("device1", "{...}");
List<String> result = service.analyzeCompliance(config);
```

ANALYSIS OUTCOME

- ❑ After analyzing configuration against set of rules
- ❑ Prints result: All rules are compliant or list of non-compliant rules

➢ Code Snippet:

```
    if (result.isEmpty()) {
        System.out.println("All rules are compliant.");
    } else {
        System.out.println("Non-compliant rules: " + result);
    }
}
```

REPORT SERVICE

- Class: ReportService
- Purpose: Generate Compliance Report
- Methods: generateReport, getCompiledReport

➢ Code Snippet:

```
public class ReportService {

    public byte[] generateReport(List<Configuration> configs) throws JRException {
        JRBeanCollectionDataSource dataSource = new JRBeanCollectionDataSource(configs);
        JasperPrint print = JasperFillManager.fillReport(getCompiledReport(), new HashMap<>(), dataSource);
        return JasperExportManager.exportReportToPdf(print);
    }

    private net.sf.jasperreports.engine.JasperReport getCompiledReport() throws JRException {

        return JasperCompileManager.compileReport(
            getClass().getClassLoader().getResourceAsStream("compliance_report.jrxml")
        );
```

Explanation of Report Service Java File

1. **Report Generation Method:** The generateReport(List<Configuration> configs) method is responsible for creating a compliance report based on a list of configurations. It throws a JRException to signal potential errors during report generation using the JasperReports library.

2. **Data Source Preparation:** The method creates a JRBeanCollectionDataSource object using the provided configs list. This data source allows the report to access data from the Configuration objects.

3. **Report Filling:** It calls JasperFillManager.fillReport with three arguments:
   i. The compiled report template retrieved using getCompiledReport (explained later).
   ii. An empty HashMap<> (potentially used for additional report parameters if needed).
   iii. The JRBeanCollectionDataSource containing the configuration data.
4. **Compiling Report Template (Private):** The private getCompiledReport method retrieves the compiled report template. It uses the class loader to access a resource named "compliance_report.jrxml" which is likely a Jasper Reports template file defining the report layout and data binding logic.

5. **Exporting Report to PDF:** Finally, the method uses JasperExportManager.exportReportToPdf to convert the filled report (JasperPrint object) into a PDF byte array. This byte array can then be saved to a file or streamed as needed.

TEST CASES

- **TEST CASE 1: Testing Compliance Analysis**
- Purpose: Verify that the compliance rule engine correctly identifies compliant configurations.
- Class: ComplianceRuleEngineApplicationTests
- Methods: testAnalyzeCompliance

➢ Code Snippet:

```
public class ComplianceRuleEngineApplicationTests {

    @Test
    public void testAnalyzeCompliance() {
        ComplianceRule rule = new ComplianceRule("rule1", "Test Rule",
                config -> config.getDeviceId().equals("device1"));
        ComplianceService service = new ComplianceService(List.of(rule));

        Configuration config = new Configuration("device1", "{...}");
        List<String> result = service.analyzeCompliance(config);

        assertTrue(result.isEmpty());
    }
}
```

TEST CASES

- **TEST CASE 2 : Testing Report Generation**
- Purpose: Verify that the report generation functionality works correctly.
- Class: ReportServiceTest
- Methods: testGenerateReport

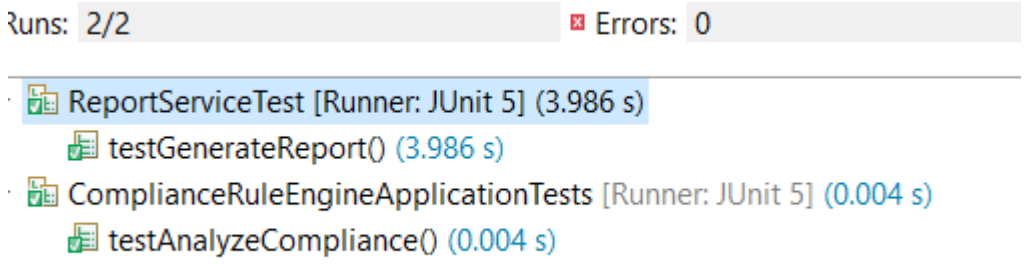
➢ Code Snippet:

```java
public class ReportServiceTest {

    @Test
    public void testGenerateReport() throws JRException {
        ReportService service = new ReportService();
        List<Configuration> configs = List.of(new Configuration("device1", "{...}"));

        byte[] report = service.generateReport(configs);
        assertNotNull(report);
    }
```

➢ Test cases passed successfully:

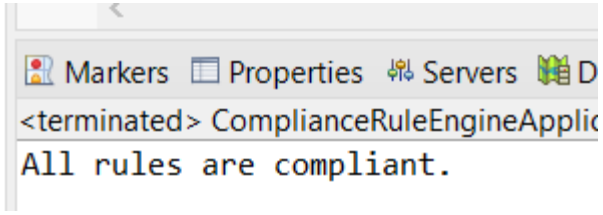


OUTPUT EXPLANATION

- The system's output depends on the compliance evaluation results:
  - **All Rules Compliant**:

```
Configuration config = new Configuration("device1", "{...}");
```

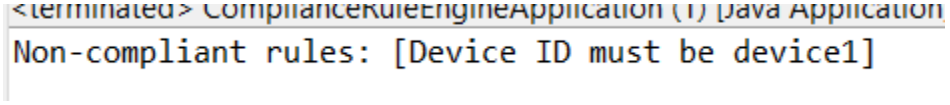
    - Output: "All rules are compliant."
    - Indicates that the device configuration meets all defined compliance standards.



  - **Non-Compliant Rules**:

```
Configuration config = new Configuration("device2", "{...}");
```

    - Output: "Non-compliant rules: [Description of non-compliant rule]"
    - Lists any rules that the device configuration does not meet, allowing for corrective actions



TECHNOLOGIES USED

- JAVA: Core Programming Language
- ECLIPSE: Integrated Environment Development
- SPRING: Application Development Framework
- MAVEN: Project Management and build automation tool
- JASPER-REPORTS: Report generation library
- MYSQL: Database for storing configuration and compliance tool

CONCLUSION

❑ Summary of the project's achievements:

- Successful development of a compliance rule engine.
- Automated analysis and reporting of device configurations.
- Ensured configurations meet predefined standards.

❑ FUTURE WORK

- Adding more compliance rules to cover additional scenarios.
- Improving reporting features with more detailed analytics and visualizations.
- Integrating with other systems for better automation and monitoring.
- Enhancing user interface for easier rule management and report generation.

THANK YOU

PRESENTERS:

Rijin Raj

Charuli Pramod Shirsath

Ravi Kumaran P

Rathod Sushma

Dhrubangshu Prabal Goswami

Punith N R

CHAITHANYA Ramesh R

DEEPIKA NAIK

Ragula Sagar

Priyanka Thakur