# SECO Automation framework

**What is Apache Camel?**

**Apache Camel** is an open-source integration framework that simplifies communication between different systems, applications, and protocols. It provides a **rule-based routing and mediation engine** that allows developers to connect diverse technologies seamlessly.

**Key Features of Apache Camel:**

1. **Enterprise Integration Patterns (EIPs):**
   * Implements standard **messaging patterns** for system integration.
   * Example: Routing, transformation, error handling, and message aggregation.
2. **Multiple Protocols & Technologies:**
   * Supports **HTTP, FTP, JMS, Kafka, SQL, REST, SOAP, SSH**, and more(around 350 components).
   * Helps connect databases, message queues, web services, and cloud services and much more.
3. **Declarative Routing (DSLs):**
   * Allows developers to **define integration logic in a simple and readable way** using Java, XML, Kotlin, or YAML.
4. **Error Handling & Resilience:**
   * Built-in retry mechanisms, logging, and fault tolerance.
   * Ensures seamless communication between systems.
5. **Scalability & Performance:**
   * Works well in cloud and on prem environments with **microservices, Kubernetes, and Docker**.
   * Can process millions of messages efficiently.

**Why Use Apache Camel in Our System?**

* **Simplifies Integration:** Reduces the complexity of connecting different systems.
* **Reduces Development Time:** Pre-built connectors and patterns speed up implementation.
* **Reliable & Scalable:** Handles large data flows efficiently.
* **Open-Source & Active Community:** No licensing cost and continuously improved.

**Scenario: Reading a File, Transforming Data, and Sending to a Message Queue**

**1. Normal Java Code (Without Apache Camel)**

In traditional Java, handling integration requires a lot of boilerplate code for file reading, transformation, and message queuing.

import java.io.\*;

import javax.jms.\*;

import org.apache.activemq.ActiveMQConnectionFactory;

public class ManualIntegration {

public static void main(String[] args) {

try {

// Read file

File file = new File("input/orders.txt");

BufferedReader br = new BufferedReader(new FileReader(file));

StringBuilder content = new StringBuilder();

String line;

while ((line = br.readLine()) != null) {

content.append(line).append("\n");

}

br.close();

// Transform content (convert to uppercase as an example)

String transformedContent = content.toString().toUpperCase();

// Send to JMS Queue

ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://localhost:61616");

Connection connection = factory.createConnection();

Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);

Queue queue = session.createQueue("orders");

MessageProducer producer = session.createProducer(queue);

TextMessage message = session.createTextMessage(transformedContent);

producer.send(message);

// Cleanup

producer.close();

session.close();

connection.close();

System.out.println("Message sent successfully!");

} catch (Exception e) {

e.printStackTrace();

}

}

}

**Issues with Normal Java Code:**

* **Too much boilerplate** (file handling, string manipulation, JMS connection setup).
* **Error-prone** (requires manual exception handling and resource management).
* **Difficult to scale** (if more transformations or endpoints are needed, the complexity increases).

**2. Apache Camel Code (Simplified Integration)**

Apache Camel significantly reduces code complexity by using a declarative DSL.

import org.apache.camel.CamelContext;

import org.apache.camel.impl.DefaultCamelContext;

import org.apache.camel.builder.RouteBuilder;

public class CamelIntegration {

public static void main(String[] args) throws Exception {

CamelContext context = new DefaultCamelContext();

context.addRoutes(new RouteBuilder() {

public void configure() {

from("file:input?noop=true") // Read from file

.convertBodyTo(String.class) // Convert to string

.transform().simple("${body.toUpperCase()}") // Transform data

.to("jms:queue:orders"); // Send to JMS Queue

}

});

context.start();

Thread.sleep(5000);

context.stop();

}

}

**Why Apache Camel is More Productive?**

| **Feature** | **Normal Java Code** | **Apache Camel Code** |
| --- | --- | --- |
| **Lines of Code** | ~50+ lines | ~10 lines |
| **Complexity** | High (manual coding for each step) | Low (declarative routing) |
| **Error Handling** | Manual exception handling | Built-in error handling |
| **Scalability** | Hard to scale with new endpoints | Easily add new components |
| **Maintenance** | Difficult (tight coupling) | Easy (loose coupling) |
| **Integration** | Manual API calls | Pre-built connectors (JMS, FTP, Kafka, etc.) |

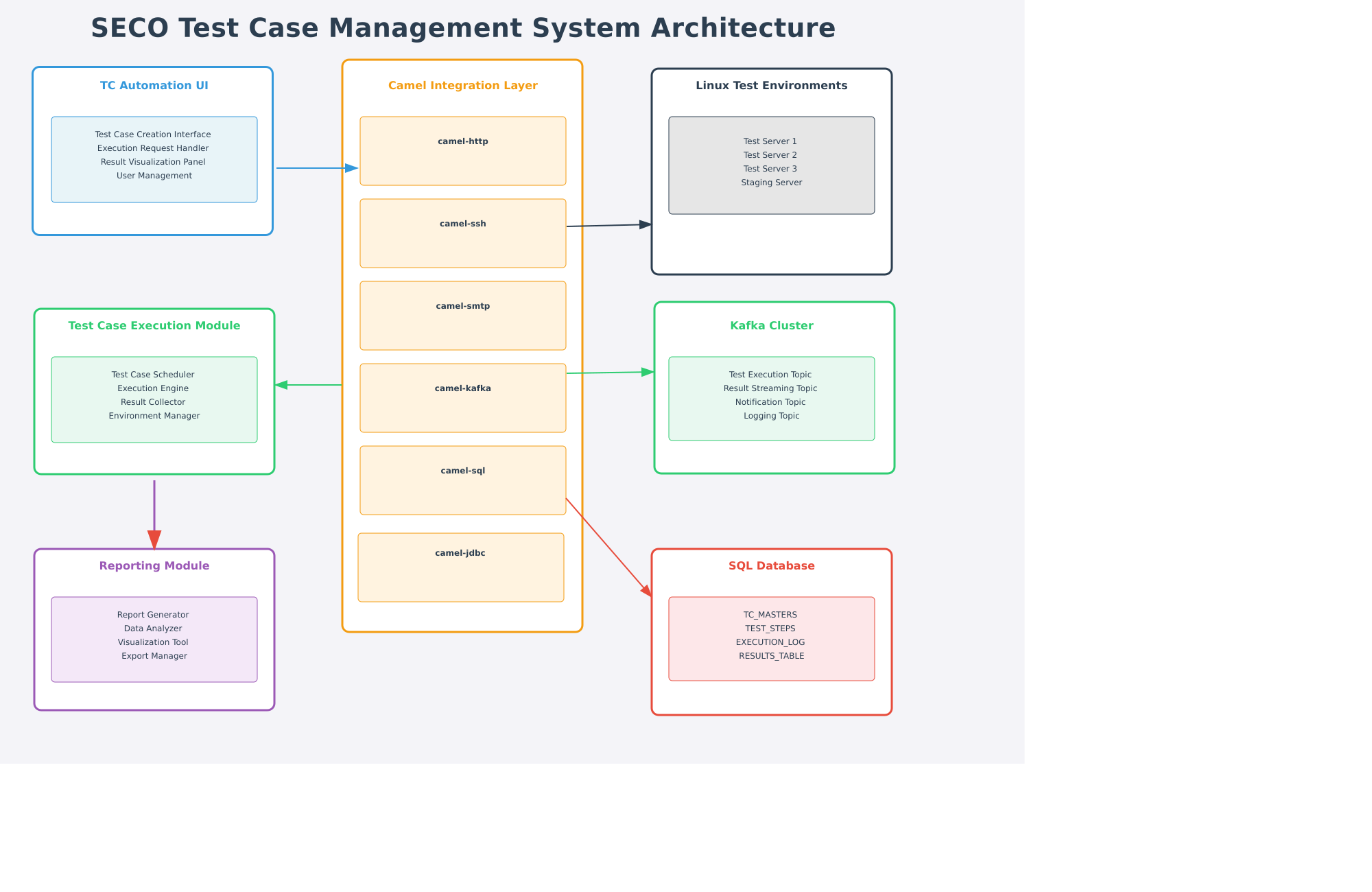
**Key Benefits of Apache Camel:**

✅ **Less Code, More Functionality** – Reduces boilerplate code significantly.  
✅ **Easier Maintenance** – Declarative routing makes it readable and easy to modify.  
✅ **Built-in Connectors** – No need to manually implement integration with message queues, databases, etc.  
✅ **Scalability** – Adding new endpoints (like Kafka, HTTP, or a database) is just one line of configuration.

**SECO Test Case Management System Architecture**

**Overview**

The SECO Test Case Management System is designed to facilitate test automation, execution, reporting, and integration with various infrastructure components. This document outlines the system architecture, including its major modules, interactions, and key technologies.



**System Components**

The architecture is structured into three main sections:

* **Test Case Management** (Left Column)
* **Camel Integration Layer** (Center Column)
* **Infrastructure** (Right Column)

**1. Test Case Management**

This section manages test case automation, execution, and reporting.

**TC Automation UI (Blue)**

This is the user interface for managing test cases.

* **Features:**
  + Test Case Creation Interface
  + Execution Request Handler
  + Result Visualization Panel
  + User Management

**Test Case Execution Module (Green)**

Handles the execution of test cases and environment management.

* **Components:**
  + Test Case Scheduler
  + Execution Engine
  + Result Collector
  + Environment Manager

**Reporting Module (Purple)**

Generates test execution reports and analyzes data.

* **Features:**
  + Report Generator
  + Data Analyzer
  + Visualization Tool
  + Export Manager

**2. Camel Integration Layer (Middleware)**

The **Camel Integration Layer** acts as middleware, facilitating communication between different system components.

**Subcomponents:**

* **camel-http** - Handles HTTP-based communication.
* **camel-ssh** - Enables SSH communication with Linux servers.
* **camel-smtp** - Manages email notifications.
* **camel-kafka** - Integrates with the Kafka messaging system.
* **camel-sql** - Manages database interactions.
* **camel-jdbc** - Handles JDBC-based database transactions.

**3. Infrastructure**

This section includes the core infrastructure that supports test execution and data management.

**Linux Servers (Dark Gray)**

Used as test environments for executing test cases.

* **Servers:**
  + Test Server 1
  + Test Server 2
  + Test Server 3
  + Staging Server

**Kafka Cluster (Green)**

Handles messaging and event streaming between different modules.

* **Topics:**
  + Test Execution Topic
  + Result Streaming Topic
  + Notification Topic
  + Logging Topic

**SQL Database (Red)**

Stores test case information, execution results, and logs.

* **Key Tables:**
  + TC\_MASTERS - Stores test case details.
  + TEST\_STEPS - Stores step-by-step test execution details.
  + EXECUTION\_LOG - Logs all test executions.
  + RESULTS\_TABLE - Stores test results.

**Interactions & Communication**

The system components interact through a series of communication pathways:

* **UI to Execution Module** (Blue Arrow)
* **Execution Module to Camel Integration Layer** (Green Arrow)
* **Camel to Reporting Module** (Red Arrow)
* **Camel SQL to Database** (Red Arrow)
* **Camel SSH to Linux Servers** (Gray Arrow)
* **Camel Kafka to Kafka Cluster** (Green Arrow)

## Opportunity / Problem Statement

The current test case management system relies on manual processes, including test case creation, execution, result collection, and reporting. While this approach has worked in the past, the increasing complexity of applications and the need for faster testing cycles have exposed limitations in scalability, efficiency, and accuracy. The system currently executes test cases sequentially across multiple environments, leading to delays, inefficiencies, and higher operational overhead. Manual data handling further increases the risk of errors and inconsistencies, impacting decision-making and software quality.

### Current State (Manual):

- Test Case Creation: Manual via spreadsheets/documents, prone to errors.

- Test Execution: Sequential execution of test cases across different environments.

- Result Collection: Manual data gathering, time-consuming.

- Reporting: Manually compiled reports, delays in insights.

- Environment Management: Manual setup, inconsistent configurations.

- Data Handling: Prone to loss and corruption.

- Process Management: Manually managed test execution on servers.

- Scalability: Limited due to manual processes and static infrastructure.

### Target State (Automated):

- Test Case Creation: Web-based UI enables structured, role-based test management.

- Test Execution: Supports parallel execution using camel-load-balancer for optimized performance.

- Result Collection: Uses camel-sql & camel-jdbc for structured data storage.

- Reporting: Automated report generation via camel-smtp for email distribution.

- Environment Management: camel-ssh automates remote setup and monitoring.

- Data Handling: Automated pipelines ensure data integrity and security.

- Process Management: Uses camel-ssh for remote execution and process automation.

- Scalability: camel-load-balancer optimizes execution distribution across multiple servers.

## Apache Camel Integration & Benefits

- camel-sql / camel-jdbc – Automates database operations, ensuring structured logging.

- camel-ssh – Enables secure remote execution and environment provisioning.

- camel-smtp – Automates test execution report notifications.

- camel-kafka – Enables real-time communication between execution, reporting, and logging systems.

- camel-http – Connects with external REST APIs for integrations.

- camel-load-balancer – Enables parallel test execution by distributing workload across multiple nodes.

- camel-scheduler – Schedules test executions without manual intervention.

### #### \*\*Accelerated Test Execution\*\*

### \*\*[Faster]\*\*

\*\*XX%\*\* improvement in execution speed with parallel execution.

Automated scheduling eliminates delays and ensures continuous testing cycles.

Testing duration reduced from months to a few days.

#### \*\*Significant Cost Reduction\*\*

\*\*[Cheaper]\*\*

\*\*YY%\*\* reduction in operational costs by eliminating manual testing efforts.

Reduction from 8 testers to only one required resource to complete the SIT testing

Automation reduces the need for additional infrastructure, minimizing hardware and maintenance expenses.

#### \*\*Enhanced Product/Solution Offering\*\*

\*\*[Better]\*\*

\*\*ZZ%\*\* increase in system efficiency and accuracy.

Automation ensures reliable test execution, improving software quality and user experience.

Appendixes:

The lack of parallel execution, automated scheduling, and dynamic resource allocation limits the system’s ability to handle growing workloads. Additionally, the reliance on static environments creates bottlenecks in managing test infrastructure efficiently. To address these challenges, a shift to an automated, scalable, and resilient test management framework is necessary.  
  
By integrating Apache Camel components, the proposed solution automates test execution, optimizes resource utilization, and enhances system scalability. Components such as camel-sql, camel-ssh, camel-smtp, camel-kafka, camel-http, camel-load-balancer, and camel-scheduler enable seamless workflow automation, real-time communication, and parallel execution of test cases. This transformation ensures faster execution, improved accuracy, real-time reporting, and better resource management, making the test management system more efficient and future-ready.