EDPP 2.0 – SECO Regression Testing Framework using Apache Camel  
Presentation Script (Spoken Style)  
  
Slide 1: Overview Flow – Manual to Automated Testing  
  
Let me walk you through how we transformed our manual regression testing process using Apache Camel.  
  
First, let’s understand the problem statement.  
  
We are not dealing with a typical web or desktop application where we can just use tools like Selenium or Tosca to automate end-to-end flows. This is a deeply integrated regression testing scenario that spans across multiple systems — including SQL databases, Kafka messaging layers, and Linux-based servers.  
  
Each test case involves connecting to multiple platforms, executing commands, validating intermediate results, and pushing output to downstream systems.  
  
Since no off-the-shelf test automation tool supports this kind of heterogeneous, multi-system orchestration out of the box, building a custom framework was not just preferred — it was inevitable.  
  
Now on the left, you can see the current state. This is where testers are manually connecting to SQL servers, Kafka topics, and Linux machines. They run validations, execute commands, and collect results one by one. This process is not only slow but also prone to human error.  
  
Now, if we move to the center of the slide, we introduce Apache Camel. Camel is our automation engine. It connects to all these systems automatically — using Camel-JDBC to talk to databases, Camel-Kafka to interact with messaging topics, and Camel-SSH to run remote scripts on Linux. Once configured, Camel takes care of the entire test flow end to end.  
  
Finally, on the right, the results are pushed directly to the SECO TCA Dashboard. This means no more manual logging or waiting for test results. Everything is centralized, fast, and consistent.  
  
To summarize this slide: we’ve moved from a completely manual testing approach to a smart, automated, and scalable one — powered entirely by Apache Camel.  
  
Slide 2: Current State vs Target State (Refined Comparison)  
  
Let’s now look at this slide with a side-by-side, apple-to-apple comparison between the current state and our target state with Apache Camel.  
  
Integration Effort:  
In the current state, testers have to manually log into each system — be it SQL Server, Linux machines, or Kafka — one by one, in a fixed sequence. Every connection is manual and script-driven.  
In the target state, Apache Camel handles all these integrations automatically using built-in connectors like camel-jdbc, camel-ssh, and camel-kafka. The moment a test case is triggered, Camel initiates all the required connections without human effort.  
  
Execution Speed & Sequential Bottleneck:  
Currently, execution is strictly sequential. If one tester is executing a test case on a server, others must wait for that execution to finish to avoid conflicts — especially when they share access to the same Kafka topics or database tables. This causes delays and underutilizes resources.  
In contrast, our target state uses parallel execution. Apache Camel routes are independent and stateless. Each test case is scheduled with isolated threads or flows, meaning multiple test cases can be executed simultaneously across different servers and environments. Camel also supports auto-scaling, so resources can be dynamically allocated based on the test load.  
  
Monitoring & Logging:  
Today, we have very minimal logging — mostly console outputs or manual log capture by testers.  
With automation, real-time monitoring and logging is fully integrated. Apache Camel logs every event — connections, errors, executions, and results — into centralized logs, giving instant visibility into what's running and what's failing.  
  
Test Case Management:  
In the manual flow, creating test cases is difficult. It involves writing scripts, copying from previous runs, and ensuring compatibility with target systems.  
In the target setup, we provide a UI-based interface to create both prepopulated and custom test cases easily — without needing to touch backend systems. It’s clean, fast, and repeatable.  
  
Error Handling & Recovery:  
Manual error handling is reactive — if something fails, the tester has to check logs, identify what broke, and rerun everything from scratch.  
In the automated flow, we’ve built retry logic and exception handling directly into the Apache Camel routes. If a test step fails, Camel retries automatically or logs detailed error messages for partial recovery — without interrupting other test cases.  
  
Test Results:  
In the current model, test results are manually noted and pasted into Excel or shared trackers.  
With the new framework, results are stored in a structured format in the database and automatically visualized on the dashboard — ensuring consistency and eliminating manual work.  
  
Historical Data Analysis:  
Today, historical test results are scattered — sometimes in Excel sheets, sometimes in screenshots, and often missing entirely.  
The target state maintains a central repository of all test results, making trend analysis, audit checks, and debugging significantly easier.  
  
QA Resource Utilization:  
We currently rely on 7 QA engineers just to run, track, and report these tests manually.  
With automation in place, we need only 5 people, focused on maintaining the framework and performing strategic testing — not just repetitive execution.  
  
So, as you can see, this is a true one-to-one transformation — every major limitation in the current model is addressed with a scalable, resilient solution powered by Apache Camel.  
More importantly, we’ve removed the dependency bottleneck by enabling parallel, independent execution, which is a game-changer in regression cycles.  
  
Slide 3: Architecture Diagram  
  
This slide gives a technical overview of how the system works behind the scenes.  
  
It all starts with the tester providing the test steps and the server they want to target. This input goes through a user-friendly interface we’ve built — which lets testers create cases, trigger executions, and view results.  
  
The heart of our framework is the Apache Camel integration layer. This is where all the magic happens. Camel uses different components — like camel-http to trigger services, camel-ssh to connect to Linux boxes, camel-kafka for messaging, and camel-jdbc to handle database operations.  
  
On the right, you can see all the systems this framework connects to — our test servers, Kafka clusters, execution engines, and SQL databases. Every interaction is automated.  
  
Finally, everything flows into a live dashboard that shows real-time updates. And for deeper insights, we have a reporting module that helps us analyze, export, and visualize test data.  
  
In essence, Apache Camel is the conductor of this testing orchestra — making sure everything runs smoothly, on time, and without human intervention.  
  
Thank you.