**Test Automation Walkthrough - Demo Script**

**1. Introduction (2-3 mins)**

**Welcome & Objective:**  
"Good [morning/afternoon], everyone! Today, I’ll be giving a walkthrough of the **Test Automation Framework** that I’ve implemented using Java, Selenium, and Cucumber. The goal of this framework is to be **environment-agnostic, scalable, and maintainable** while improving test execution efficiency."

**Tech Stack Used:**

* **Automation:** Java + Selenium + Cucumber
* **Data Handling:** JSON
* **Reports:** Extent Reports
* **Logging & Monitoring:** Log4j, ADO Bug Logging, Teams Notification
* **CI/CD Integration:** Jenkins (if applicable)

**2. Core Implementations (15-20 mins)**

**✅ Environment Agnostic Command**

* Run tests across different environments (dev, qa, prod) using:
* mvn test -Denv=qa
* **Fallback Mechanism** ensures execution even if env is not set.

**✅ Configuration Management (config.properties)**

* Stores **Application URLs, Credentials, Timeouts**
* Ensures **no hardcoded credentials** in feature files

**✅ Externalizing Test Data using JSON**

* **Why JSON?**
  + Easy to update without modifying feature files
  + Supports nested data structures
  + Enhances test case reusability
* **Example JSON File Structure (qa/connectorData.json)**:
* {
* "connectors": {
* "ADO\_Azure": {
* "ToolAPIURL": "https://dev.azure.com/PhoenixSolutions/",
* "ToolProject": "Phoenix Solutions",
* "CKProject": "RTPN"
* }
* }
* }
* **Usage in Step Definition:**
* JsonDataReader jsonReader = new JsonDataReader("connectorData.json");
* String apiUrl = jsonReader.getValue("ADO\_Azure", "ToolAPIURL");

**✅ Showing JSON-based Data in Feature Files**

* **Before (Hardcoded Data):**
* When I enter the text as "https://dev.azure.com/PhoenixSolutions/" in Tool API URL
* **After (Dynamic JSON Data):**
* When I enter the text as "<ToolAPIURL>" in Tool API URL

**✅ Extent Report Integration**

* Captures screenshots on failures
* Shows step-wise execution details
* Includes **dynamic values from JSON**

**✅ Project Creation Approaches**

* **Approach 1:** Hardcoded in Feature File
* **Approach 2:** Externalized Data from JSON
* **Approach 3:** **ContextManager** (Data-driven, reusable, adaptable)

**✅ Logger File Implementation (Log4j)**

* Captures execution logs & error traces

**✅ Error Handling Mechanism**

* Centralized exception handling for **missing elements, API failures, and JSON parsing errors**

**✅ Automated Bug Logging in ADO Board**

* Logs bug automatically **on test failure** with **error screenshots & logs**

**✅ Teams Notification on Test Status**

* Sends real-time **pass/fail notifications**

**✅ Email Test Report on Completion**

* Automatically emails Extent Reports to stakeholders

**✅ Auto Logging of Test Cases in Test Plans**

* Uses API to sync executed test cases with Test Plans in ADO

**3. Demo of Test Execution & Reports (10-15 mins)**

* **Run a Test in QA Environment**
* mvn test -Denv=qa
* **Show Dynamic JSON-based Execution**
* **View Logs & Extent Report**
* **Check ADO Bug Logging for Failed Scenarios**
* **Verify Teams Notification & Email Reports**

**4. Scope for Improvement & Q&A (5-10 mins)**

🔹 **Planned Enhancements:**

* **🔄 Rerun Failed Tests** before logging bugs
* **⚡ Parallel Execution** to reduce test time
* **⏳ Optimize Element Waiting Time**
* **📊 Customization of Reports**
* **🚀 Improve False Positive/Negative Handling**
* **🌐 Unified Exception Handling Framework**

**5. Conclusion**

**Final Takeaway:**  
💡 "With this framework, we achieve a robust, scalable, and maintainable test automation solution!"

**❓ Q&A Session**  
*Open the floor for questions!*

**🔹 Suggested Improvements for Test Automation**

**1️⃣ Introduce a Dedicated Database for Test Automation**

✅ **Why?**

* Centralized storage for **test configurations, execution results, logs, test data, and bug tracking**.
* Avoid relying on **JSON or external files**, making test data **easier to update and track**.
* Maintain historical execution data for **trend analysis and debugging**.

✅ **Implementation Plan:**

* Use a relational DB like **PostgreSQL, MySQL, or MongoDB**.
* Store details for **connectors, test cases, results, execution history, logs, and failure reasons**.
* Introduce a **Test Execution Dashboard** that pulls data from the DB.

✅ **Example Table Structure:**

| **Table Name** | **Purpose** | **Key Columns** |
| --- | --- | --- |
| test\_suites | Stores details of test suites | id, name, description, created\_at |
| test\_cases | Stores test cases mapped to test suites | id, suite\_id, name, type, status, created\_at |
| test\_runs | Stores each test execution run | id, test\_case\_id, environment, start\_time, end\_time, result |
| connectors | Stores connector-specific test data | id, name, tool\_url, project\_name, api\_token |
| test\_logs | Stores logs and error details | id, test\_run\_id, log\_details, timestamp |

✅ **Execution Flow:**

1. **Test Script Reads Data from DB → Executes Test → Logs Results Back to DB**
2. **Dashboard Displays Live Execution Stats**

**Implement a Test Execution Dashboard**

**🎯 Why?**

* Centralized **UI for test reports, logs, failures, and bug tracking**.
* **Real-time tracking** of execution status, failures, and trends.

**🛠 Tech Stack:**

* Backend: **Spring Boot (REST APIs)**
* Frontend: **ReactJS / Angular**
* DB: **PostgreSQL / MySQL / MongoDB**

**3️⃣ Automate Bug Logging & Retest Failed Cases**

**🔄 Implement Auto Bug Logging to ADO (Azure DevOps)**

* On **multiple failures**, automatically create a **new bug** in ADO.
* Attach **logs, screenshots, and stack traces**.

**🔄 Implement Auto-Rerun for Failed Cases**

* **First Attempt**: Run tests normally.
* **Second Attempt (on failure)**: Rerun failed cases before logging bugs.
* **Third Attempt (if failed again)**: Log bug & notify team via Teams/Slack.

✅ **How?**

* Implement a **Retry Mechanism** in Cucumber Test Runner
* Store **failure reasons in DB** to track common issues

**4️⃣ Parallel Execution & Test Data Isolation**

**🔹 Run Tests in Parallel**

* Use **Selenium Grid, TestNG Parallel Execution, or Dockerized Selenium**
* Reduce overall execution time by running tests **concurrently**

**🔹 Isolate Test Data Per Execution**

* Instead of overwriting **shared test data**, create **unique test records per run**
* Use **Dynamic Data Generation** for new test cases

**5️⃣ Customizable Extent Report Enhancements**

✅ **Advanced Features for Extent Reports:**

* **Attach JSON data dynamically** to reports
* **Display historical execution trends**
* **Link execution logs & DB entries to reports**

**🔹 Summary of Enhancements:**

1️⃣ **Use a dedicated database** for automation data storage  
2️⃣ **Build a Test Execution Dashboard** for real-time tracking  
3️⃣ **Implement retry logic & auto bug logging** in ADO  
4️⃣ **Enable parallel execution** to reduce test time  
5️⃣ **Enhance reporting & analytics** for better insights

**Why This Approach is the Best for Test Automation?**

✅ **Centralized Test Data Management**

* Eliminates dependency on scattered JSON files or config files.
* Ensures **data consistency** across different environments.

✅ **Improved Test Execution Tracking & Debugging**

* Stores **historical execution data** for trend analysis.
* **Live dashboards** provide real-time visibility into test execution.

✅ **Efficient Bug Handling & Reporting**

* Automatically logs **bugs in Azure DevOps (ADO)** on repeated failures.
* Captures **detailed logs, screenshots, and stack traces** for easy debugging.

✅ **Parallel Execution for Faster Testing**

* Enables **concurrent test execution** to reduce test run time.
* Supports **scalable automation infrastructure** using Selenium Grid/Docker.

✅ **Dynamic & Scalable Automation Framework**

* Can be **extended to support multiple connectors** or applications.
* **Data-driven testing** approach allows easy test case modifications.

✅ **Seamless Integration with CI/CD Pipelines**

* Test results can be **pushed to CI/CD pipelines** for automated deployments.
* Triggers **real-time notifications via Teams, Slack, or Email**.

**⚠️ Challenges in Implementation & How to Overcome Them**

❌ **Complex Database Management**

**Challenge**: Designing a scalable DB structure for tracking test executions.

* **Solution**: Define a **well-structured schema** & use ORM (e.g., Hibernate).

❌ **Additional Overhead in Maintaining DB & APIs**

**Challenge**: Setting up a dedicated **Spring Boot backend** to manage test data dynamically.

* **Solution**: Use **Spring Boot REST APIs** to fetch/update test data dynamically.

❌ **Setup Effort for Parallel Execution**

 **Challenge**: Scaling test execution requires proper infrastructure setup (e.g., Selenium Grid, Docker).

* **Solution**: Use **Dockerized Selenium Grid** or cloud-based execution (e.g., BrowserStack).

❌ **Handling Dynamic Test Data & Isolation**

**Challenge**: Ensuring **test data uniqueness** and managing multiple test scenarios.

* **Solution**: Implement **test data versioning** & dynamically generate unique records.

❌ **Ensuring Robust Reporting Mechanism**

 **Challenge**: Extending Extent Reports to show **database-driven** test execution details.

* **Solution**: Enhance **Extent Reports** with DB integration for detailed analysis.

❌ **Effort & Time Required for Implementation**

* **Challenge**: High effort and time investment in setting up DB, APIs, and automation.
* **Solution**: **Phased implementation approach**—start with JSON-based automation, then gradually move to DB integration.

❌ **Technology Stack Dependency (Spring Boot & ReactJS)**

* **Challenge**: Requires expertise in **Spring Boot** for backend APIs and **ReactJS** for dashboards.
* **Solution**: Allocate **dedicated resources** for backend & frontend development OR use a simpler reporting tool like Kibana/Power BI for visualization.

🚀 **Final Thoughts:**  
This approach **transforms traditional automation** into a **scalable, efficient, and intelligent framework** that:  
✔ Provides **real-time insights**  
✔ Reduces **manual effort** in debugging  
✔ Improves **execution efficiency**  
✔ Ensures **seamless integration with DevOps workflows**

**📊 Estimated Efforts & Time Breakdown**

| **Task** | **Estimated Time** | **Dependencies** |
| --- | --- | --- |
| Database Design & Setup | 1-2 weeks | DBA, Spring Boot Developer |
| API Development for Test Data Management | 2-3 weeks | Spring Boot Developer |
| Integration with Test Automation | 2 weeks | Selenium, Cucumber, Java |
| Enhancing Reporting with DB Data | 1-2 weeks | Java, Extent Reports |
| Parallel Execution Setup | 2 weeks | Selenium Grid, Docker |
| CI/CD & Notifications Integration | 1 week | DevOps, Azure/Teams API |
| Dashboard Development (ReactJS) | 2-3 weeks | ReactJS Developer |
| End-to-End Testing & Optimization | 2-3 weeks | QA, Automation Team |

