**Lab Exercise 7- Kie Stateful and Stateless Sessions in Drools**



**Step 1: Define the Same Java Classes (Order.java)**

We'll reuse the Order.java class as defined earlier:

package com.example.model;

public class Order {

private String id;



private double totalAmount;

private boolean valid;



public Order(String id, double totalAmount) {

this.id = id;

this.totalAmount = totalAmount;

this.valid = false;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public double getTotalAmount() {

return totalAmount;

}

public void setTotalAmount(double totalAmount) {

this.totalAmount = totalAmount;

}

public boolean isValid() {

return valid;

}

public void setValid(boolean valid) {

this.valid = valid;

}

@Override

public String toString() {

return "Order{id='" + id + "', totalAmount=" + totalAmount + ", valid=" + valid + '}';

}

}

**Step 2: Define the Rules (orderValidationRules.drl)**

package com.example.rules;

import com.example.model.Order;

rule "Validate Order Amount"

when



$order : Order(totalAmount >= 100)

then



$order.setValid(true);

System.out.println("Order " + $order.getId() + " is valid.");

end

**Step 3: Test Using Stateless and Stateful Sessions**

We will now create two test classes to clearly demonstrate the difference in how state is retained between **stateless** and **stateful** sessions.

**Test 1: Stateless Session (StatelessSessionTest.java)**

package com.example.model;

import org.kie.api.KieServices;

import org.kie.api.runtime.KieContainer;

import org.kie.api.runtime.StatelessKieSession;

public class StatelessSessionTest {

public static void main(String[] args) {

// Load the KieServices and get the KieContainer

KieServices ks = KieServices.Factory.get();

KieContainer kContainer = ks.getKieClasspathContainer();

// Create a stateless Kie session

StatelessKieSession kSession = kContainer.newStatelessKieSession("statelessOrderKSession");

// Create an order with a total amount of 80

Order order = new Order("Order-001", 80);



// Execute the rules first time

System.out.println("First execution (stateless):");

kSession.execute(order);



System.out.println("Order status after first execution: " + (order.isValid() ? "Valid" : "Invalid"));

// Modify the order total and try again

order.setValid(false);

order.setTotalAmount(150);

// Execute the rules a second time

System.out.println("\nSecond execution (stateless):");

kSession.execute(order);

System.out.println("Order status after second execution: " + (order.isValid() ? "Valid" : "Invalid"));

}

}

**Test 2: Stateful Session (StatefulSessionTest.java)**

package com.example.model;

import org.kie.api.KieServices;

import org.kie.api.runtime.KieContainer;

import org.kie.api.runtime.KieSession;

public class StatefulSessionTest {

public static void main(String[] args) {

// Load the KieServices and get the KieContainer

KieServices ks = KieServices.Factory.get();

KieContainer kContainer = ks.getKieClasspathContainer();

// Create a stateful Kie session

KieSession kSession = kContainer.newKieSession("statefulOrderKSession");

// Create an order with a total amount of 80

Order order = new Order("Order-001", 80);

// Insert the order into the session

kSession.insert(order);

// Fire rules for the first time

System.out.println("First execution (stateful):");

kSession.fireAllRules();

System.out.println("Order status after first execution: " + (order.isValid() ? "Valid" : "Invalid"));

// Modify the order total and try again

order.setValid(false);

order.setTotalAmount(150);

// Fire the rules again without inserting the order again

System.out.println("\nSecond execution (stateful):");

kSession.fireAllRules();

System.out.println("Order status after second execution: " + (order.isValid() ? "Valid" : "Invalid"));

// Dispose of the session

kSession.dispose();

}

}

**Explanation:**

1. **Stateless Session**:
   * After the first execution, the order is marked as **invalid** because its total amount is less than 100.
   * Before the second execution, we modify the order's total amount to 150, but the session starts fresh each time, and it does not remember any previous executions or changes unless explicitly reinserted into the session.
2. **Stateful Session**:
   * After the first execution, the order remains in memory (the **working memory**) of the session. Any updates to the order object are retained between rule executions.
   * Before the second execution, we modify the order, and since the session retains the order in memory, the rule will re-evaluate the modified order when fireAllRules() is called again, without needing to reinsert the order into the session.

**Step 4: Modify kmodule.xml**

Make sure that your kmodule.xml includes configurations for both stateful and stateless sessions:

<?xml version="1.0" encoding="UTF-8"?>

<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">

<kbase name="orderValidationKBase" packages="com.example.rules">

<ksession name="statelessOrderKSession" type="stateless"/>

<ksession name="statefulOrderKSession" type="stateful"/>

</kbase>

</kmodule>

**Expected Output:**

**Stateless Session Output:**

First execution (stateful):

Order status after first execution: Invalid

Second execution (stateful):

Order status after second execution: Invalid

**Stateful Session Output:**

First execution (stateful):

Order Order-001 is not valid.

Order status after first execution: Invalid

Second execution (stateful):

Order Order-001 is valid.

Order status after second execution: Valid

**Summary:**

* In the **stateless session**, the order is re-evaluated fresh with no memory of previous executions. Hence, modifying the order and re-executing the rules requires manually resetting the object's state.
* In the **stateful session**, the order remains in working memory across multiple executions, so changes to the order are reflected automatically in subsequent rule executions, demonstrating that the session retains the state.

This difference demonstrates **state retention** in the stateful session compared to the stateless session, which forgets everything after each execution.