Distributed Processing Code Walkthrough

Rules Project

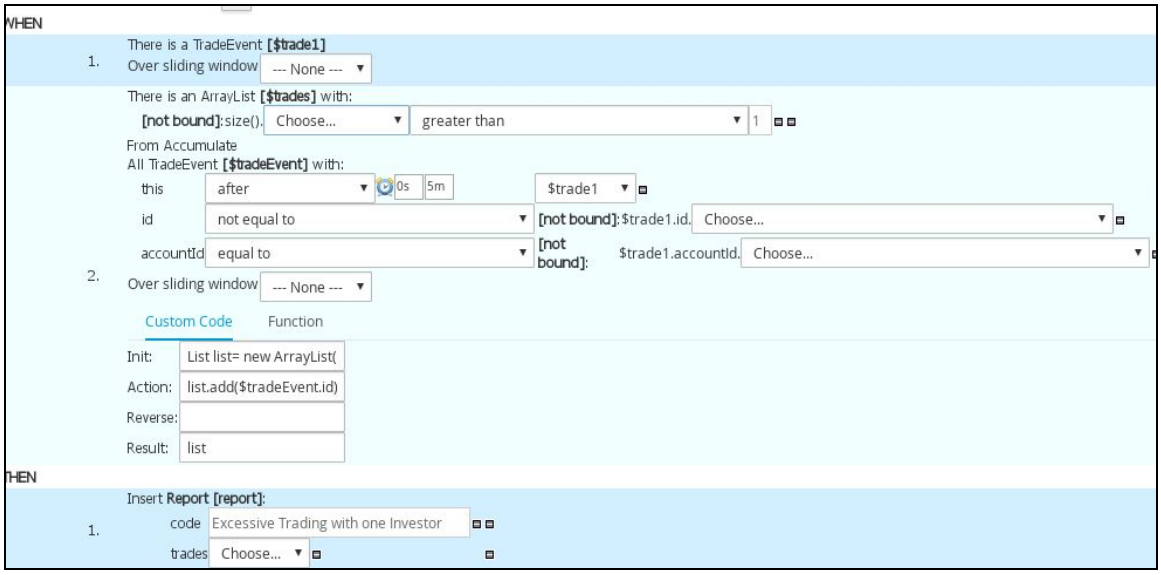
The ‘partition-trades-example’ project in ‘drools-example-rules’ repo demonstrates how you can split up your rules into two different *KieBases* while still keeping them in the same project.

The first step is to create a separate package for each grouping of rules. In this project we only have two rules so we created two packages to house each rule and another package to house our queries which are used to query the facts in our session and help us when it comes to testing

partition-trades-example/src/main/resources/com/redhat/partition-trades-examples:

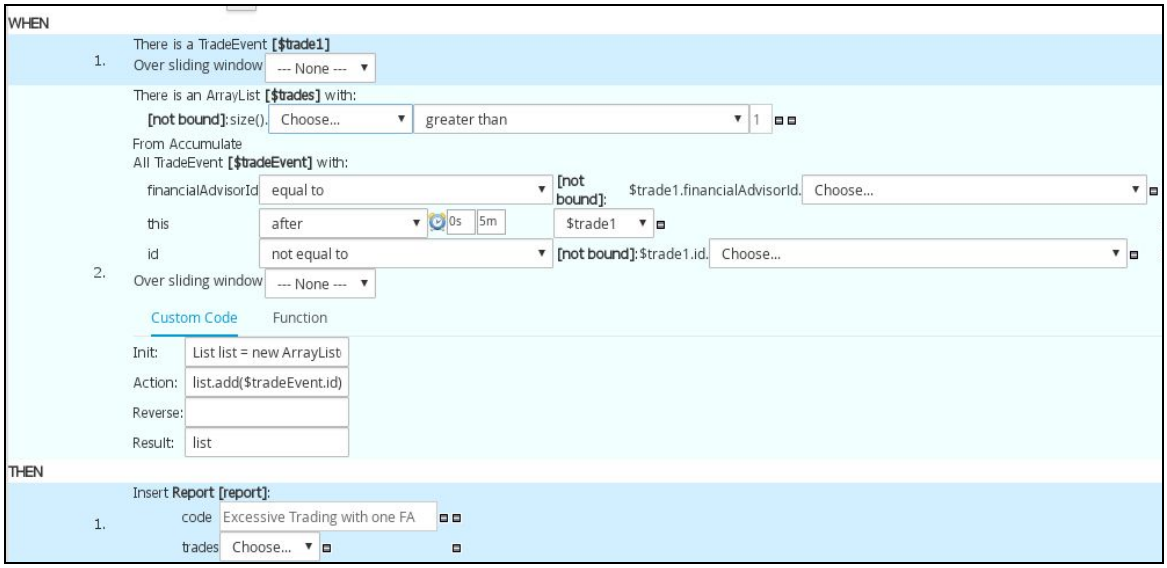
The ‘suspicious_accounts’ package holds a rule that checks if more than one *TradeEvent* with the same ‘accountId’ has occurred within five minutes of each other. If this condition is met then a *Report* will be generated with all the ids of the *TradeEvent*s that matched the condition and a code of “Excessive Trading with one Investor”

partition-trades-example/src/main/resources/com/redhat/partition-trades-examples/suspicious_accounts:



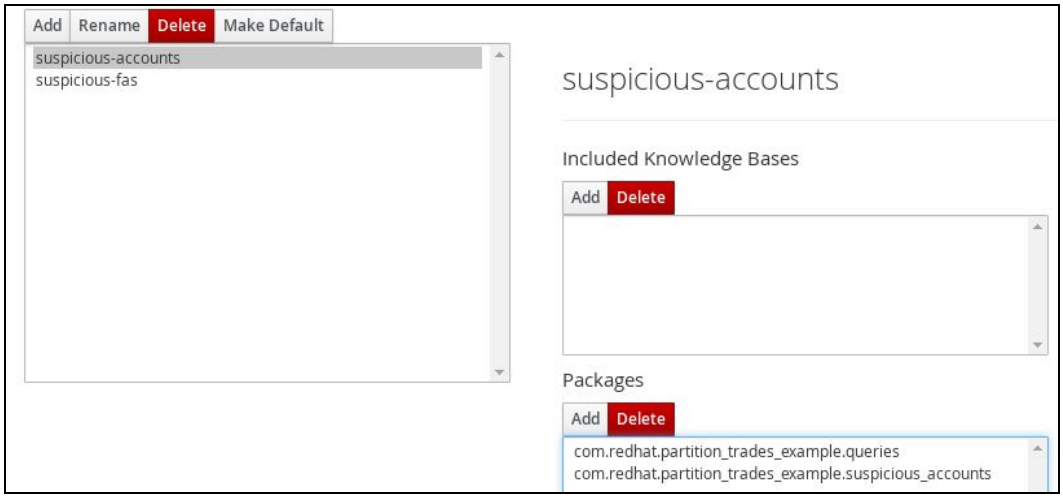
The ‘suspicious_financial_advisors’ package holds a rule that checks if more than one *TradeEvent* with the same ‘financialAdvisorId’ has occurred within five minutes of each other. If this condition is met then a Report will be generated with all the ids of all the *TradeEvents* that matched the condition and a code of “Excessive Trading with one FA”

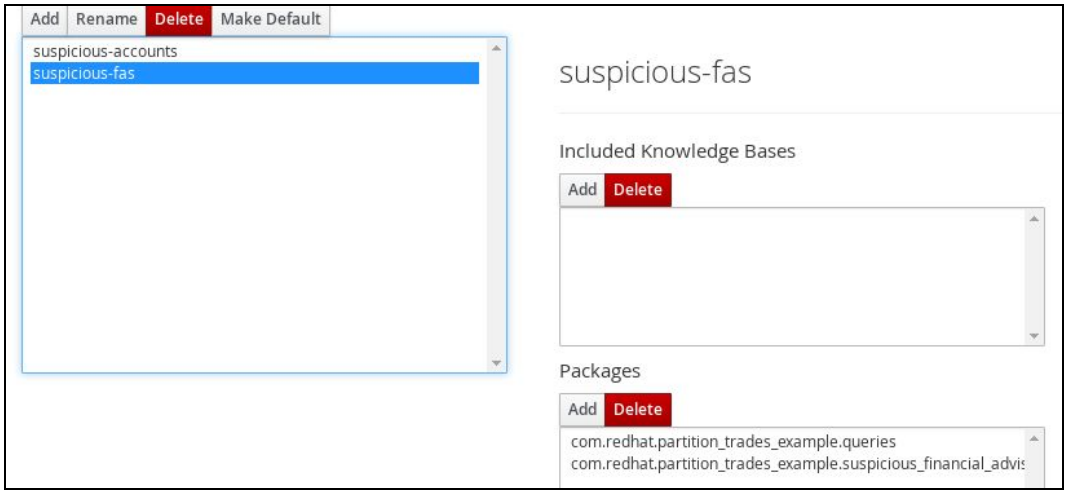
partition-trades-example/src/main/resources/com/redhat/partition-trades-examples/suspicious_financial_advisors:



You can create *KieBases* with these separate packages in the ‘Project Editor’ in business-central shown below. Here you can add *KieBases* with various names and add packages to the *KieBase*. In the project we currently have two *KieBases* called ‘suspicious-accounts’ and ‘suspicious-fas’ each *KieBase* holds the corresponding rule package and the query package to help us with testing later

Partition-trades-example >> Open Project Editor >> Knowledge Base Settings >> Knowledge bases and sessions:





You can also just directly edit this file yourself to configure *KieBases* and *KieSessions*.

src/main/resources/META-INF/kmodule.xml:

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <kbase name="suspicious-accounts" default="false" eventProcessingMode="cloud" equalsBehavior="identity"
  packages="com.redhat.partition_trades_example.queries, com.redhat.partition_trades_example.suspicious_accounts"/>

  <kbase name="suspicious-fas" default="false" eventProcessingMode="cloud" equalsBehavior="identity"
  packages="com.redhat.partition_trades_example.queries, com.redhat.partition_trades_example.suspicious_financial_advisors">

    <ksession name="randomss" type="stateless" default="true" clockType="realtime"/>
  </kbase>
</kmodule>
```

Application Project:

In the ‘usecase-application’ project inside the ‘drools-example-application’ repo you will find the tests that create *TradeEvents*, partition them into different *KieSessions* for the *KieBases* defined in the rules project, and query the *Reports* generated to make sure the rules fired correctly The ‘partitionTradeEvents.feature’ file is a Cucumber Test Feature. It is a readable way to define a test and its data. Each ‘Given’, ‘When’, and ‘Then’ statement is mapped to pieces of Java Code in the ‘CucumberSteps.java’ file shown in the next few images below.

usecase-application/src/test/java/com/redhat/usecase/cucumber/features/partition_example/partitionTradeEvents.feature

```
Feature: Test Feature to demonstrate partitioning rules and data

Scenario: Multiple trades in 5 minutes with same Investor and FA
Given Trade Events:
|Id|Investor Id|Advisor Id| Price |Quantity|Symbol|Type    |  Timestamp          |
|1 | 1         |  3       | 20.00 | 10     | ANS  | Buy    |  27-09-1990 01:29:33 |
|2 | 1         |  5       | 20.00 | 11     | ABS  | Sell   |  27-09-1990 01:28:08 |
|3 | 2         |  2       | 20.00 | 15     | HNG  | Sell   |  27-09-1990 13:25:13 |
|4 | 3         |  1       | 20.00 | 20     | TYI  | Buy    |  27-09-1990 19:27:10 |
|5 | 3         |  1       | 20.00 | 20     | JKJ  | Buy    |  27-09-1990 19:27:22 |
|6 | 3         |  1       | 20.00 | 20     | ANS  | Buy    |  27-09-1990 19:30:09 |
When I run the partition example
Then I expect the following Reports to be created:
|Code                                | Trades          |
|Excessive Trading with one Investor  | 5,6             |
|Excessive Trading with one Investor  | 4,5,6           |
|Excessive Trading with one Investor  | 1,2             |
|Excessive Trading with one FA        | 4,5,6           |
|Excessive Trading with one FA        | 5,6             |
```

The ‘Given’ statement code parses the table of data below the statement in the feature file and creates a list of *TradeEvents*:

usecase-application/src/test/java/com/redhat/usecase/cucumber/CucumberSteps.java

```
@Given("^Trade Events:$")
public void trade_events(DataTable table) throws Throwable {

    for (Map<String, String> row : table.asMaps(String.class, String.class)) {
        TradeEvent trade = new TradeEvent();

        trade.setId(row.get("Id"));
        trade.setAccountId(row.get("Investor Id"));
        trade.setFinancialAdvisorId(row.get("Advisor Id"));
        trade.setPrice(Double.parseDouble(row.get("Price")));
        trade.setQuantity(Integer.parseInt(row.get("Quantity")));
        trade.setSymbol(row.get("Symbol"));
        trade.setType(row.get("Type"));

        if (row.get("Timestamp") != null) {

            trade.setTimestamp(parseDate(row.get("Timestamp")));

        }

        trades.add(trade);

    }
}
```

The ‘When’ statement code calls passes the *TradeEvent* list created in the ‘Given’ statement code to a service which does the work to partition the *TradeEvents* into different sessions for the different *KieBases*

usecase-application/src/test/java/com/redhat/usecase/cucumber/CucumberSteps.java

```
@When("^I run the partition example$")
public void i_run_the_parition_example_rules() throws Throwable {
    reportsGenerated = reportServicePartitioned.fireAllRules(trades);
}
```

The service code in is ‘ReportServicePartitioned.java’. The ‘fireAllRules’ function takes the trades and uses the Apache Spark Java Famework to split up the data two different ways.

First the *TradeEvents* are grouped by ‘financialAdvisorId’. The *KieBase* with the rule to detect suspicious financial advisors is loaded and then the ‘applyRulesInKieBase’ function is called on each group of *TradeEvents* with the same ‘financialAdvisorId’.

Next the *TradeEvents* are groups by ‘accountId’. The *KieBase* with the rule to detect suspicious accounts is loaded and then the ‘applyRulesInKieBase’ function is called on each group of *TradeEvents* with the same ‘accountId’.

The ‘applyRulesInKieBase’ function creates a *StatlessKieSession* for the *KieBase* passed in. It inserts all the *TradeEvents* passed in into to the session and calls ‘fireAllRules’. After the rules fire it queries the session for all the *Reports* generated and adds the *Reports* to the classes report list.

usecase-application/src/main/java/com/redhat/usecase/services/ReportServicePartitioned.java

```
public ArrayList<Report> fireAllRules(ArrayList<TradeEvent> inputTrades) {

    SparkConf conf = new SparkConf().setAppName("Simple Application").setMaster("local");
    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<TradeEvent> trades = sc.parallelize(inputTrades);

    // group by financial advisor and apply the corresponding rules
    JavaPairRDD<String, Iterable<TradeEvent>> tradesByFA = trades.groupBy(t -> t.getFinancialAdvisorId());

    System.out.println("NEW KIEBASE");
    KieBase rulesFA = loadRules("suspicious-fas");
    Broadcast<KieBase> broadcastKieBase1 = sc.broadcast(rulesFA);
    long numberOfGroups = tradesByFA.mapValues(f -> applyRulesInKieBase(f, broadcastKieBase1.value())).count();
    System.out.println("NUMBER OF SESSIONS FINISHED: " + numberOfGroups);

    // group by account and apply the corresponding rules
    JavaPairRDD<String, Iterable<TradeEvent>> tradesByAccounts = trades.groupBy(t -> t.getAccountId());

    System.out.println("NEW KIEBASE");
    KieBase rulesAccount = loadRules("suspicious-accounts");
    Broadcast<KieBase> broadcastKieBase2 = sc.broadcast(rulesAccount);
    numberOfGroups = tradesByAccounts.mapValues(f -> applyRulesInKieBase(f, broadcastKieBase2.value())).count();
    System.out.println("NUMBER OF SESSIONS FINISHED: " + numberOfGroups);

    sc.stop();
    sc.close();
    return reports;
}
```

```
public static Iterable<TradeEvent> applyRulesInKieBase(Iterable<TradeEvent> trades, KieBase rules) {

    TrackingAgendaEventListener listener = new TrackingAgendaEventListener();

    // loops through each group of trades, creates session, and fires rules

    System.out.println("NEW SESSION");
    StatelessKieSession session = rules.newStatelessKieSession();

    // uncomment this to add the event listener
    // session.addEventListener(listener);

    Iterator<TradeEvent> iterator = trades.iterator();

    List<Command> list = new ArrayList<Command>();

    // insert rules into session
    while (iterator.hasNext()) {
        TradeEvent nextTrade = iterator.next();
        System.out.println("TRADE: " + nextTrade.toString());

        list.add(CommandFactory.newInsert(nextTrade));
    }

    list.add(CommandFactory.newFireAllRules());
    list.add(CommandFactory.newQuery("reports generated", "reports generated"));

    ExecutionResults results = session.execute(CommandFactory.newBatchExecution(list));
    QueryResults queryResults = (QueryResults) results.getValue("reports generated");

    for (QueryResultsRow row : queryResults) {
        Report report = (Report) row.get("$report");
        addToReports(report);
    }

    // uncomment this to print out rules that fired
    // System.out.println(listener.getMatchList());
    return trades;

}
```

By the time ‘fireAllRules’ function in *ReportServiceParititioned* is done all the *Reports* generated in every *StatelessKieSession* have been added to the report list field. This is then returned to the ‘CucumberSteps.java’ class. The ‘Then’ statement code then compares the *Reports* returned with the *Reports* expected (parsed from the feature file).

```
@Then("^I expect the following Reports to be created:$")
public void i_expect_the_following_Report_to_be_created(DataTable table) throws Throwable {
    ArrayList<Report> reports = new ArrayList<Report>();
    for (Map<String, String> row : table.asMaps(String.class, String.class)) {
        Report report = new Report();
        report.setCode(row.get("Code"));

        ArrayList<String> list = new ArrayList<String>(Arrays.asList(row.get("Trades").split(",")));
        report.setTrades(list);
        reports.add(report);
    }

    Assert.assertTrue(arraysMatch(reports, reportsGenerated));

}
```

You can run this feature file by locating *src/main/test/com/redhat/usecases/runners/RunParitionExample.java* and running it as a Junit Test in your IDE. You will see in the console some print statements from the *ReportServiceParitioned* class. These statements show you what was in each *StatelessKieSession* that was created.

These are the sessions created for the “suspicious-fas” *KieBase*:

```
NEW SESSION
TRADE: TradeEvent [accountId=1, financialAdvisorId=5,
NEW SESSION
TRADE: TradeEvent [accountId=2, financialAdvisorId=2,
NEW SESSION
TRADE: TradeEvent [accountId=1, financialAdvisorId=3,
NEW SESSION
TRADE: TradeEvent [accountId=3, financialAdvisorId=1,
TRADE: TradeEvent [accountId=3, financialAdvisorId=1,
TRADE: TradeEvent [accountId=3, financialAdvisorId=1,
```

These are the sessions created for the “suspicious-accounts” *KieBase*:

```
NEW SESSION
TRADE: TradeEvent [accountId=2, financialAdvisorId=2,
NEW SESSION
TRADE: TradeEvent [accountId=3, financialAdvisorId=1,
TRADE: TradeEvent [accountId=3, financialAdvisorId=1,
TRADE: TradeEvent [accountId=3, financialAdvisorId=1,
NEW SESSION
TRADE: TradeEvent [accountId=1, financialAdvisorId=3,
TRADE: TradeEvent [accountId=1, financialAdvisorId=5,
```