

Expert Rules Engine

Toolkit



Rest API Rules Engine

@Spring Boot

Parham
Data Products

Copyright © 2020, 2024 *Wayne Parham*

All rights reserved. No part of this software or documentation may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author. For usage permission requests, write to the author at the address below.

To clarify, this software is publically accessible, and you are free to download, inspect, study and run it, *but only for personal use*. You are not allowed to copy or distribute this software or manual without written permission, and you are not allowed to use it in whole or in part in commercial systems, whether they be for-profit or non-profit. If you want to use this software in your company or organization, you must first request and be granted written permission.

Parham Data Products

P. O. Box 5811

Bella Vista, AR 72714

ParhamData.com

wayne@parhamdata.com

918-663-2131

Usage and Ordering Information:

This software is available for purchases by corporations, associations, and others. For details, contact the author at the address above.

Printed in the United States of America

Expert Rules Engine

Programming and Users Guide

The *Expert Rules Engine* is a tool that can be used as a rules engine microservice. It is built with Spring Boot and uses a Rest API for interface with other systems. The *Expert Rules Engine* can be used as-is in many situations but it is also fully extensible so that domain-specific logic can be easily incorporated.

Table of Contents

Rules Engine Introduction	1
What is a Rules Engine?	1
What a Rules Engine is not	1
Overview of the <i>Expert Rules Engine</i>	2
Dependencies and Prerequisites	3
Administration RestAPI definition	4
Get All Rulesets	4
Get All Rules in Domain	5
Get Rule by Domain and ID	5
Get Rule(s) by ID	6
Create or Update Rule	6
Bulk Load or Update Rules	7
Delete All Rules	7
Delete Rule	8
Getting ready to run your first Ruleset	8
Data Definitions	8

Table of Contents (continued)

RulesDetails Implementations	9
Input Definition	9
Output Definition	9
VariableResolver	10
RulesDetails Directory Structure	11
Domain RestAPI Definition	12
Load the Rules	13
Loan Domain Ruleset	14
Running the Ruleset	15
RulesDetails Implementation for Data containing Multiple Input Records	17
Input Definition for Data containing Multiple Input Records	17
RulesDetails Helper Functions	18
Output Definition	19
Promo Domain Ruleset	19
Running the Ruleset	21
RulesDetails Implementation for Multiple Output Records	22
RestAPI Mapping to Support Multiple Output Records	23
MultiPromo Domain Ruleset	23
Running the Ruleset	24
RulesDetails Implementation for Compound Rulesets	27
RestAPI Mapping to Support Cascaded Compound Records	28
CompoundPromo Domain Ruleset	29
PromoAggregate Domain Ruleset	30
Running the Ruleset(s)	32
Logfile	35

Rules Engine Introduction

What is a Rules Engine?

A rules engine is a system that encapsulates executable human-readable rules in a storage medium that is easily changed. The rules can be created, edited and stored without language compilation often required by computer programming languages. Each rule uses a simplified language that expresses the goals and tasks required of the domain it is used in.

Rule engines allow you to describe what to do without having to know how to do it. Since the rules are described in a human-readable format, business analysts, subject matter experts and other non-programmers are able to understand them. The goal is to shift the focus to the problem at hand rather than on the programming language used to implement it.

Another advantage of using a rules engine is its separation of business rules with application programming. Most companies require a formal process of development, testing and deployment of application code which is both time-consuming and expensive. Having the business rules outside the application code essentially treats the rules as data, which can be quickly and easily modified. So updates to business rules are easier to do.

What a Rules Engine is not

A rules engine does not maintain any sort of state information, so it cannot be used for total process workflow.

Most applications have some sort of state engine and business rules processing functionality. Historically, state information and business rules were encoded in the programming language that the applications programs were written in. However, it is useful for an application to employ a state machine that calls one or more rules engines to provide complete workflow management.

As an example, consider a Manufacturing Control System. It contains inventory control, bill of materials processing and scheduling functionality. A Manufacturing Control System is used to plan purchasing, tooling operations and delivering activities. Each software subsystem may employ one or more rules engines, but they are all driven by some sort of state machine or workflow engine. The Manufacturing Control System needs to keep track of inventory, know when to order parts and when to schedule manufacturing, packaging and freight operations.

Trying to shoehorn workflow or state information into a rules engine would be a bad choice. The inputs and outputs to a rules engine may contain state information, because knowing state may be required for a business decision implemented by a ruleset. But the rules engine itself is stateless.

Another case where rules engines aren't needed is when the "rules" are better implemented by simple lookup tables. An example would be a tax table or any other sort of linear list. It is not necessary to provide rules processing if what is needed is simply a lookup or one-to-one translation. Rules Engines are better choices for applying decision processing that's more complicated than what a simple lookup table can provide.

Overview of the *Expert Rules Engine*

The *Expert Rules Engine* is executed using *Java*. The executable command is `java -jar <expert.jarfile>`, which is usually named something like `expert-3.2.0-RELEASE.jar`.

The Rules Engine takes instructions from a *Ruleset* stored in a database. *Rulesets* are collections of *Rules* grouped together with a unique *Domain* identifier. Each rule has one or more *Conditions* and *Actions*. When an input message is received by the Rules Engine, it processes the message through each of the Rules, and creates an output message based on the results of the Ruleset.

Conditions can be something as simple as "`var1 == 7`" or "`var1 == var2`." Or they can be a little more complex like "`monthlySalary >= 2000 && creditScore >= 680 && debtToIncome < 20 && requestedLoanAmount < 1000000 && age >= 18`." Rules are "if statements" that test whether input conditions or a set of values meet a defined business rule.

Actions are "setter" operations that set values in the output message. They are generally of the form "`setApprovalStatus(true); setMaximumPercentage(90); setInterestRate(4.25); setProcessingFee(4000); setNotes(Approved at 4.25%)`." These setter operations set all the fields in the output message.

Each rule – or a collection of rules which is called a ruleset – is associated with a *domain*. The default domain is "EXPERT" and in many cases, this is the only one needed. If multiple rulesets are required, each will be assigned its own unique domain.

Each rule is also assigned a *priority*. This is used to select which rule is chosen if multiple rules match a condition and the rules engine is configured to provide a single result. The default value is 100. In the case of multiple matches, the one with the lowest numeric value will be selected. If multiple matching rules have the same priority value, the results of the first one encountered will be returned.

Every rule also has fields for *ID* and *description*. The description field is there purely for convenience, and allows for a comment to be added to each rule, if desired. The ID field is combined with domain to form a compound key. This allows each domain to have its own set of unique ID values. As an example, you can have a rule with domain “promo” and ID=1 and another rule with domain=“aggregate” and ID=1. Other than that, the ID is not used for rules processing. It is just a way to identify rule records within the database for maintenance and administrative purposes.

Dependencies

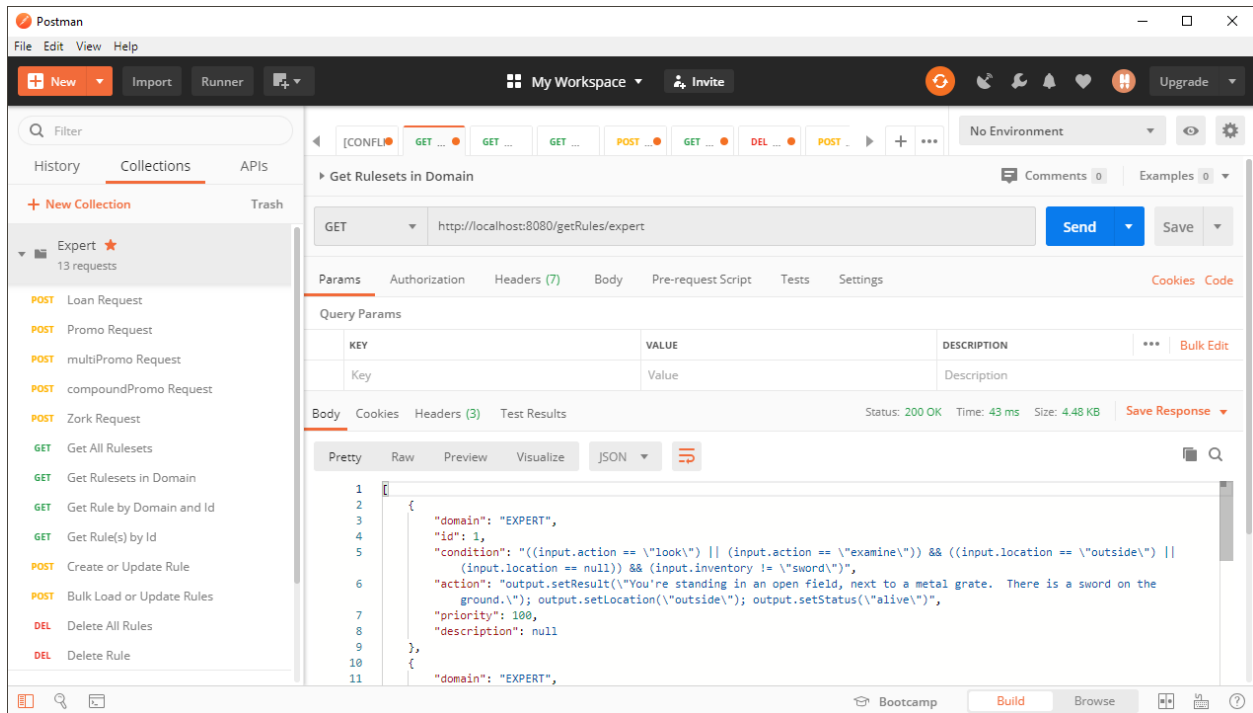
- Java 17
- Maven 3.3 or later
- MySQL 8.0 or later
- IntelliJ is optional, but recommended
- Postman or any REST client application

Prerequisites

1. Download and install Java. Follow the instructions to setup environment variables and verify your installation.
2. Download and install Maven. As with Java, follow instructions and verify that it works.
3. Download and install MySQL. Create an account and use it to login to the command line or Workspace client. Run the command, “**Create database expert**” to make the db.
4. In the `src/main/resources` directory, find and update the `application.properties` file to include your database login credentials.
5. At the top directory, type “**mvn clean install**” to build the jarfile. If successful, it will create the file `expert-3.2.0-RELEASE.jar` in the target directory.
6. From the target directory, run the application with the command “**java -jar expert-3.2.0-RELEASE.jar.**” Using your REST client, perform any of the commands below.

Administration RestAPI definition

It is useful to setup Postman (or another RestAPI client) for *Expert* Rules Engine administration.



GET View All RuleSets

{hostname}:8080/getRules

Returns JSON:

```
[
  {
    "domain": "LOAN",
    "id": 1,
    "condition": "input.monthlySalary >= 5000 && input.creditScore >= 760 && input.debtToIncome < 20 && input.requestedLoanAmount < 1000000 && input.age >= 18",
    "action": "output.setApprovalStatus(true); output.setMaximumPercentage(90); output.setInterestRate($(bank.prime_plus_half)); output.setProcessingFee(4000); output.setNotes(\\\"Approved!\\\")",
    "priority": 1,
    "description": "Eligibility for a 3.75% loan"
  },
  {
    ...
  }
]
```

Administration RestAPI definition (continued)

GET View All Rules in Domain {hostname}:8080/getRules/{domain} Returns JSON:

```
[
  {
    "domain": "LOAN",
    "id": 1,
    "condition": "input.monthlySalary >= 5000 && input.creditScore >= 760 && input.debtToIncome < 20 && input.requestedLoanAmount < 1000000 && input.age >= 18",
    "action": "output.setApprovalStatus(true); output.setMaximumPercentage(90); output.setInterestRate($(bank.prim_plus_half)); output.setProcessingFee(4000); output.setNotes(\"Approved!\")",
    "priority": 1,
    "description": "Eligibility for a 3.75% loan"
  },
  {
    ...
  }
]
```

GET View Rule by Domain and ID {hostname}:8080/getRule/{domain}/{id} Returns JSON:

```
{
  "domain": "LOAN",
  "id": 1,
  "condition": "input.monthlySalary >= 5000 && input.creditScore >= 760 && input.debtToIncome < 20 && input.requestedLoanAmount < 1000000 && input.age >= 18",
  "action": "output.setApprovalStatus(true); output.setMaximumPercentage(90); output.setInterestRate($(bank.prim_plus_half)); output.setProcessingFee(4000); output.setNotes(\"Approved!\")",
  "priority": 1,
  "description": "Eligibility for a 3.75% loan"
}
```

Administration RestAPI definition (continued)

GET View Rule(s) by ID

{hostname}:8080/getRule/{id}

Returns JSON:

```
[
  {
    "domain": "LOAN",
    "id": 1,
    "condition": "input.monthlySalary >= 5000 && input.creditScore >= 760 && input.debtToIncome < 20 && input.requestedLoanAmount < 1000000 && input.age >= 18",
    "action": "output.setApprovalStatus(true); output.setMaximumPercentage(90); output.setInterestRate($(bank.prim_plus_half)); output.setProcessingFee(4000); output.setNotes(\"Approved!\")",
    "priority": 1,
    "description": "Eligibility for a 3.75% loan"
  },
  {
    ...
  }
]
```

POST Create or Update Rule

{hostname}:8080/setRule

JSON body:

```
{
  "domain": "",
  "id": 13,
  "condition": "",
  "action": "output.setResult(\"Nothing to see here.\")",
  "priority": 99999,
  "description": "default"
}
```

Returns JSON:

```
{
  "domain": "",
  "id": 13,
  "condition": "",
  "action": "output.setResult(\"Nothing to see here.\")",
  "priority": 99999,
  "description": "default"
}
```

Note: If ID is omitted, the system will auto-generate a unique value. If domain is omitted, it defaults to the “EXPERT” domain. A blank condition is one that always evaluates true. So this rule will be chosen if there are no other matching rules with priority value less than 99999.

Administration RestAPI definition (continued)

POST Bulk Load or Update Rules

{hostname}:8080/loadRules

JSON body:

```
[
  {
    "domain": "LOAN",
    "id": 1,
    "condition": "input.monthlySalary >= 5000 && input.creditScore >= 760 && input
.debtToIncome < 20 && input.requestedLoanAmount < 1000000 && input.age >= 18",
    "action": "output.setApprovalStatus(true); output.setMaximumPercentage(90); ou
tput.setInterestRate($(bank.prim_plus_half)); output.setProcessingFee(4000); output.
setNotes(\"Approved!\")",
    "priority": 1,
    "description": "Eligibility for a 3.75% loan"
  },
  {
    ...
  }
]
```

Returns JSON:

```
[
  {
    "domain": "LOAN",
    "id": 1,
    "condition": "input.monthlySalary >= 5000 && input.creditScore >= 760 && input
.debtToIncome < 20 && input.requestedLoanAmount < 1000000 && input.age >= 18",
    "action": "output.setApprovalStatus(true); output.setMaximumPercentage(90); ou
tput.setInterestRate($(bank.prim_plus_half)); output.setProcessingFee(4000); output.
setNotes(\"Approved!\")",
    "priority": 1,
    "description": "Eligibility for a 3.75% loan"
  },
  {
    ...
  }
]
```

DEL Delete All Rules

{hostname}:8080/deleteAllRules

Returns JSON:

```
[]
```

Administration RestAPI definition (continued)

DEL Delete Rule

{hostname}:8080/deleteRule/{domain}/{id} Returns JSON:

```
{
  "domain": "",
  "id": 7,
  "condition": "input.action == \"get sword\" && (input.location == \"outside\" || i
nput.location == null)",
  "action": "output.setResult(\"You have the sword.\"); output.setLocation(\"outside
\"); output.setInventory(\"sword\"); output.setStatus(\"alive\")",
  "priority": null,
  "description": null
}
```

Getting ready to run your first Ruleset

The *Expert Rules Engine* comes with five rules engine implementations and sample rulesets. The “Zork” rules engine is a bare-bones implementation with a handful of rules. The “Loan” rules engine is another basic implementation that includes some user-defined variables. It also includes Junit and MockMvc test cases, which run automatically when you do the Maven build. And there are three “Promotions” rules engines that show how to create different kinds of implementations for various input and output requirements.

So let’s start first with the “Loan” Rules Engine.

For a bank to decide whether to offer a loan to its customers, it needs to know customer financial information. Based on this information, it can decide how much money it is willing to loan, and how much earnings it will expect for this risk. So the rules engine must be prepared to work with these kinds of data inputs and outputs.

Data Definitions

The first thing we need to do when designing a rules engine is to define what input and output data is required. It must include all relevant details required to make the decision that the rules will be called upon to make. So to make a loan, a banker would want to know the customer’s income, credit rating and debt-to-income ratio.

A banker may want to know other things as well, like assets, liabilities and length of time on the job. All those kinds of things need to be considered before starting to make rulesets. But for this example rules engine, we’re just using income, credit rating and debt-to-income ratio.

RulesDetail Implementations

After we've decided on the data we'll need, we'll define it as you see below. The input data structure is called "UserDetails" and it includes the variables shown.

Input Definition

```
public class UserDetails {  
    Integer age;  
    String lastName;  
    String firstName;  
    Integer creditScore;  
    Double debtToIncome;  
    Double monthlySalary;  
    Double requestedLoanAmount;  
}
```

Output Definition

The output data is described in a structure called "LoanDetails," which is shown below.

```
public class LoanDetails {  
    Boolean approvalStatus;  
    Float interestRate;  
    Float maximumPercentage;  
    Double processingFee;  
    String notes;  
}
```

Both of these structures are put into Java classes contained in a *RulesDetails* file structure. This is the place where you must put the specific implementation your Rules Engine. We can also include specialized variables, constants and functions in our RulesDetails implementations.

The input definition class is where we generally place "helper functions," as you will see in later examples. But in this case, we just have simple input and output structures.

As a general rule, you will want to put most variables and constants in your rulesets. That is the point of having a rules engine, making the rules drive the results. But there are some cases where defined constants may make sense. You can place those in a VariableResolver.

VariableResolver

```
public class BankResolver implements VariableResolver {
    private static final String RESOLVER = "bank";
    private static final String PRIME_INTEREST = "prime_interest";
    private static final String PRIME_PLUS_HALF = "prime_plus_half";
    private static final String PRIME_PLUS_ONE = "prime_plus_one";
    private static final String PRIME_PLUS_TWO = "prime_plus_two";
    private static final String PRIME_PLUS_THREE = "prime_plus_three";
    private static final Double primeRate = 3.25;

    @Override
    public String getResolverKeyword() {
        return RESOLVER;
    }

    @Override
    public Object resolveValue(String keyword) {
        if (keyword.equalsIgnoreCase(PRIME_INTEREST)) {
            return primeRate;
        }
        else if (keyword.equalsIgnoreCase(PRIME_PLUS_HALF)) {
            return primeRate + 0.5;
        }
        else if (keyword.equalsIgnoreCase(PRIME_PLUS_ONE)) {
            return primeRate + 1;
        }
        else if (keyword.equalsIgnoreCase(PRIME_PLUS_TWO)) {
            return primeRate + 2;
        }
        else if (keyword.equalsIgnoreCase(PRIME_PLUS_THREE)) {
            return primeRate + 3;
        }
        else {
            return null;
        }
    }
}
```

This is our VariableResolver for the Loan Rules Engine. It allows us to define domain-specific constants that can be used in rules. They can be accessed using keywords, as an example `$(bank.prime_interest)`, which returns 3.25. Likewise, `$(bank.prime_plus_half)` returns 3.75.

This code is a good example of how to create domain-specific constants, but it is a bad example of what kinds of constants should be created. Something like *prime rate* is better to put in the rules, because it changes fairly often. A better example might have been to create a keyword for the bank name. As a general rule, only create keyword constants for things that never change.

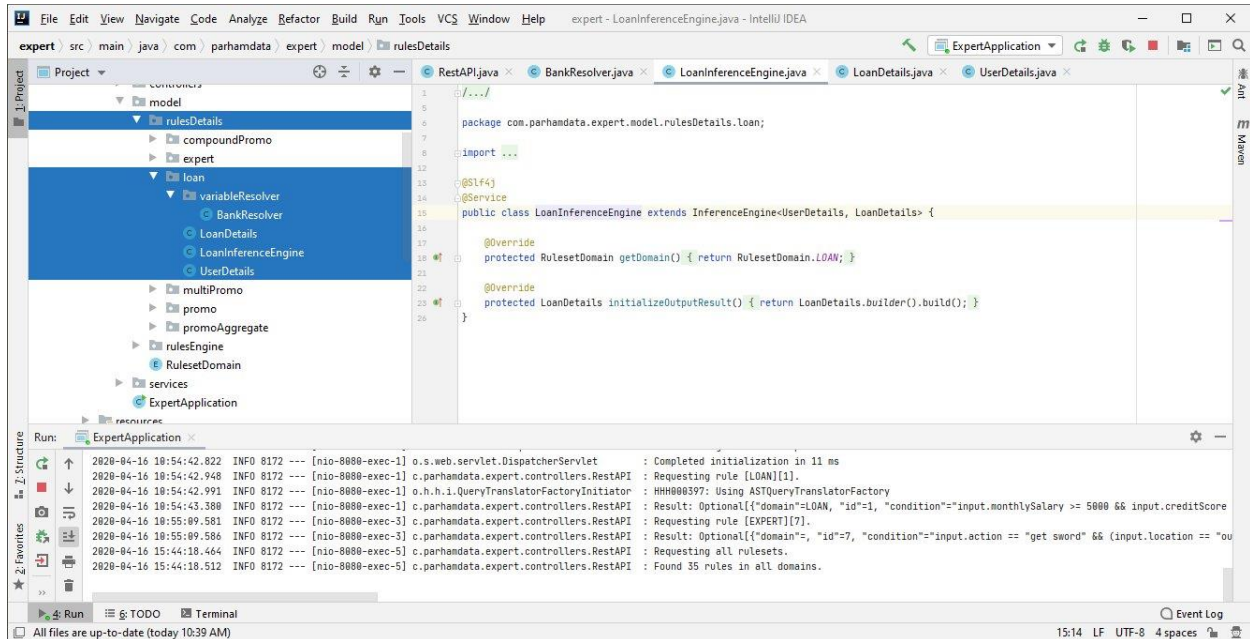
RulesDetail Directory Structure

The directory where you will find RulesDetails is:

expert/src/main/java/com/parhamdata/expert/model/rulesDetails

This is where you will create your Rules Engine implementation.

In the distribution file, you will notice there are directories for *expert*, *loan*, *promo*, *multiPromo*, *compoundPromo* and *promoAggregate*. Each of those is a separate implementation for a specific domain. Actually, the *compoundPromo* and *promoAggregate* implementations are used together to provide a single rules engine that runs two rulesets, one that provides a preliminary set of possibilities and a second that aggregates them into the final result. So we have *expert*, *loan*, *multiPromo* and *compoundPromo* rules engines.



The loan implementation is highlighted above. It shows the variableResolver, LoanDetails and UserDetails files. You will also see the InferenceEngine file, which is in the IntelliJ window to the right. In that file, you will only need to set the *domain*. The domain is the “name” of your ruleset. If there is to be only one domain, you might as well choose the default “EXPERT” domain. But you can name your domain, if you wish. In this case, the domain is “LOAN.”

You will also need to edit the RulesetDomain file to add your domain to the enum list. It is located here:

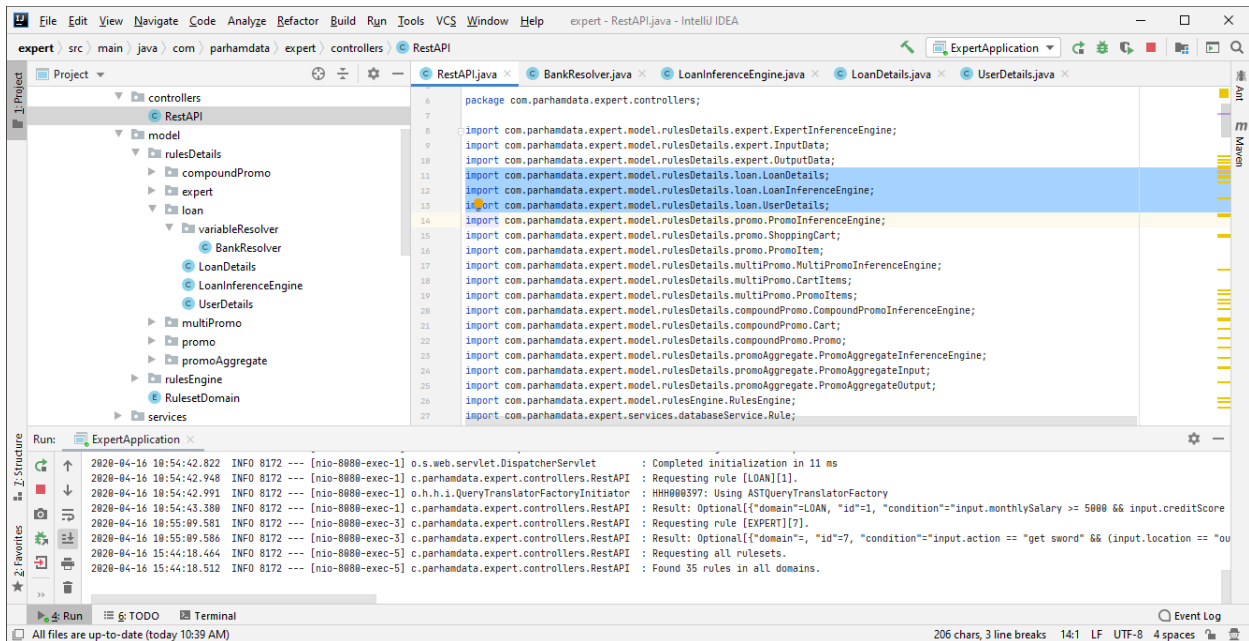
expert/src/main/java/com/parhamdata/expert/model/rulesEngine

Domain RestAPI Definition

The RestAPI definition must be added to the restAPI file, located here:

expert/src/main/java/com/parhamdata/expert/controllers

1. First, add the imports for your RulesDetails implementation:

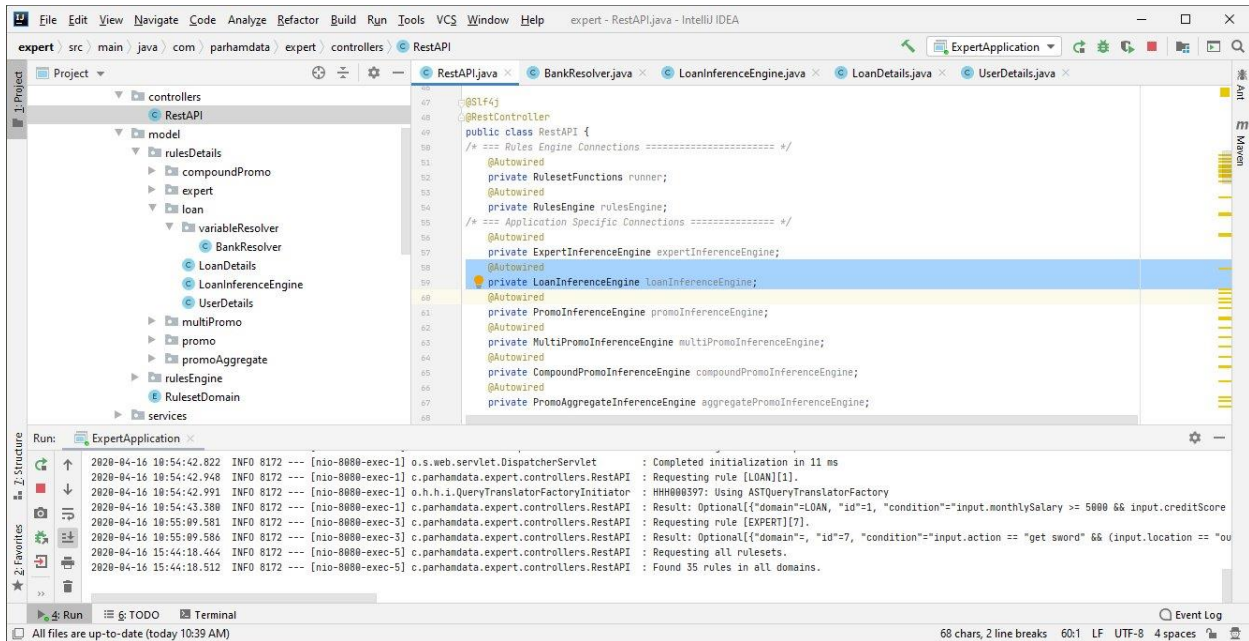


```
6 package com.parhamdata.expert.controllers;
7
8 import com.parhamdata.expert.model.rulesDetails.expert.ExpertInferenceEngine;
9 import com.parhamdata.expert.model.rulesDetails.expert.InputData;
10 import com.parhamdata.expert.model.rulesDetails.expert.OutputData;
11 import com.parhamdata.expert.model.rulesDetails.loan.LoanDetails;
12 import com.parhamdata.expert.model.rulesDetails.loan.LoanInferenceEngine;
13 import com.parhamdata.expert.model.rulesDetails.loan.UserDetails;
14 import com.parhamdata.expert.model.rulesDetails.promo.PromoInferenceEngine;
15 import com.parhamdata.expert.model.rulesDetails.promo.ShoppingCart;
16 import com.parhamdata.expert.model.rulesDetails.promo.PromoItem;
17 import com.parhamdata.expert.model.rulesDetails.multiPromo.MultiPromoInferenceEngine;
18 import com.parhamdata.expert.model.rulesDetails.multiPromo.CartItems;
19 import com.parhamdata.expert.model.rulesDetails.multiPromo.PromoItems;
20 import com.parhamdata.expert.model.rulesDetails.compoundPromo.CompoundPromoInferenceEngine;
21 import com.parhamdata.expert.model.rulesDetails.compoundPromo.Cart;
22 import com.parhamdata.expert.model.rulesDetails.compoundPromo.Promo;
23 import com.parhamdata.expert.model.rulesDetails.promoAggregate.PromoAggregateInferenceEngine;
24 import com.parhamdata.expert.model.rulesDetails.promoAggregate.PromoAggregateInput;
25 import com.parhamdata.expert.model.rulesDetails.promoAggregate.PromoAggregateOutput;
26 import com.parhamdata.expert.model.rulesEngine.RulesEngine;
27 import com.parhamdata.expert.services.databaseService.Rule;
```

Run: ExpertApplication x

```
2020-04-16 10:54:42.822 INFO 8172 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 11 ms
2020-04-16 10:54:42.948 INFO 8172 --- [nio-8080-exec-1] c.parhamdata.expert.controllers.RestAPI : Requesting rule [LOAN][1].
2020-04-16 10:54:42.991 INFO 8172 --- [nio-8080-exec-1] o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
2020-04-16 10:54:43.380 INFO 8172 --- [nio-8080-exec-1] c.parhamdata.expert.controllers.RestAPI : Result: Optional[{"domain"="LOAN", "id"=1, "condition"="input.monthlySalary >= 5000 && input.creditScore
2020-04-16 10:55:09.581 INFO 8172 --- [nio-8080-exec-3] c.parhamdata.expert.controllers.RestAPI : Requesting rule [EXPERT][7].
2020-04-16 10:55:09.586 INFO 8172 --- [nio-8080-exec-3] c.parhamdata.expert.controllers.RestAPI : Result: Optional[{"domain"=", "id"=7, "condition"="input.action == "get sword" && (input.location == "ou
2020-04-16 15:44:18.464 INFO 8172 --- [nio-8080-exec-5] c.parhamdata.expert.controllers.RestAPI : Requesting all rulesets.
2020-04-16 15:44:18.512 INFO 8172 --- [nio-8080-exec-5] c.parhamdata.expert.controllers.RestAPI : Found 35 rules in all domains.
```

2. Second, add the restAPI connection for your implementation:

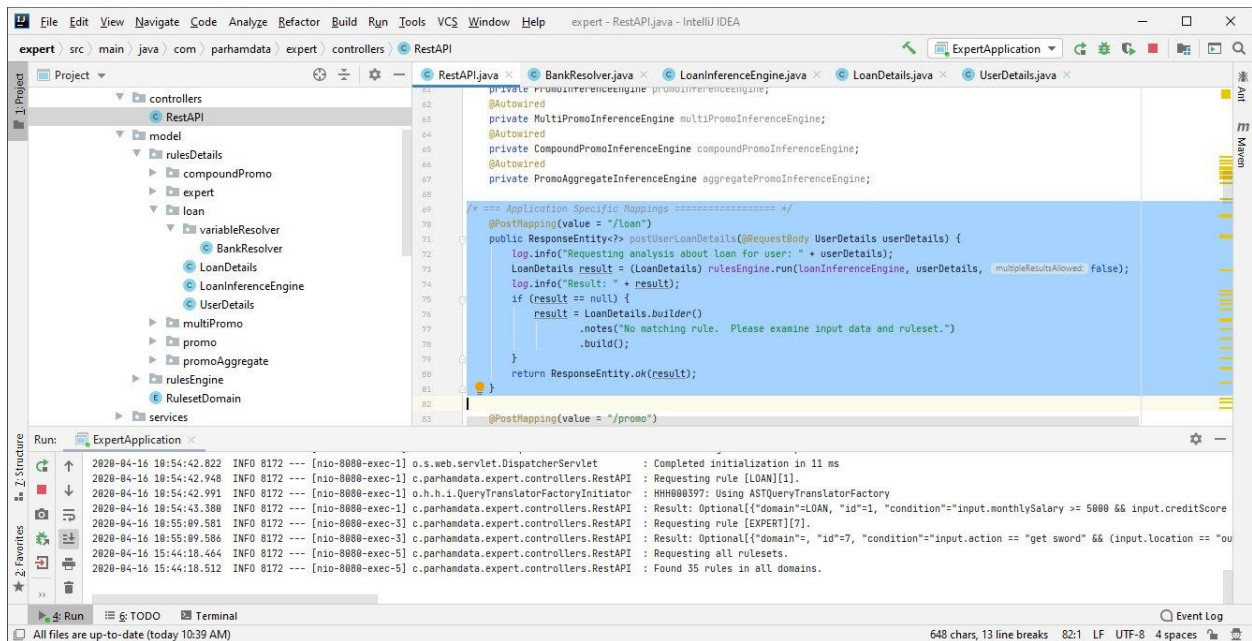


```
57 @Slf4j
58 @RestController
59 public class RestAPI {
60     /* === Rules Engine Connections === */
61     @Autowired
62     private RulesEngine rulesEngine;
63     /* === Application Specific Connections === */
64     @Autowired
65     private ExpertInferenceEngine expertInferenceEngine;
66     @Autowired
67     private LoanInferenceEngine loanInferenceEngine;
68     @Autowired
69     private PromoInferenceEngine promoInferenceEngine;
70     @Autowired
71     private MultiPromoInferenceEngine multiPromoInferenceEngine;
72     @Autowired
73     private CompoundPromoInferenceEngine compoundPromoInferenceEngine;
74     @Autowired
75     private PromoAggregateInferenceEngine aggregatePromoInferenceEngine;
76 }
```

Run: ExpertApplication x

```
2020-04-16 10:54:42.822 INFO 8172 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 11 ms
2020-04-16 10:54:42.948 INFO 8172 --- [nio-8080-exec-1] c.parhamdata.expert.controllers.RestAPI : Requesting rule [LOAN][1].
2020-04-16 10:54:42.991 INFO 8172 --- [nio-8080-exec-1] o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
2020-04-16 10:54:43.380 INFO 8172 --- [nio-8080-exec-1] c.parhamdata.expert.controllers.RestAPI : Result: Optional[{"domain"="LOAN", "id"=1, "condition"="input.monthlySalary >= 5000 && input.creditScore
2020-04-16 10:55:09.581 INFO 8172 --- [nio-8080-exec-3] c.parhamdata.expert.controllers.RestAPI : Requesting rule [EXPERT][7].
2020-04-16 10:55:09.586 INFO 8172 --- [nio-8080-exec-3] c.parhamdata.expert.controllers.RestAPI : Result: Optional[{"domain"=", "id"=7, "condition"="input.action == "get sword" && (input.location == "ou
2020-04-16 15:44:18.464 INFO 8172 --- [nio-8080-exec-5] c.parhamdata.expert.controllers.RestAPI : Requesting all rulesets.
2020-04-16 15:44:18.512 INFO 8172 --- [nio-8080-exec-5] c.parhamdata.expert.controllers.RestAPI : Found 35 rules in all domains.
```

3. And finally, add the restAPI mapping for your implementation:



At this point, the code is complete and the code is ready to compile and run. Compile it either in your IDE or by typing “mvn clean install” from the command line. Since this code was already there, your initial build in the “Dependencies and Prerequisites” section of this guide has already accomplished this step. But these are the tasks you’ll need to perform to create your own implementation going forward.

Load the Rules

At this point, it’s time to run the rules engine, load the ruleset and run it. So start the application, either within your IDE or from the command line, by typing “java -jar expert-2.7.14-RELEASE.jar.” Start Postman and create a POST method request to the following URL that can deliver a JSON body. Copy and paste the Loan rules into the body dialog box and click “Send.”

<http://<servername>:8080/loadRules> (localhost when running on a local machine)

You can find the Loan rules in the top level directory of the *Expert Rules Engine* distribution. The rules message is in a file called *loan_rules.json*. Alternatively, you can load all the rulesets for all the sample applications from the file *all_rules.json*.

Loan Domain Ruleset

The following is the contents of the *loan_rules.json* file. You can easily see what it does. There are only four rules in the ruleset.

```
[
  {
    "domain": "LOAN",
    "id": 1,
    "condition": "input.monthlySalary >= 5000 && input.creditScore >= 760 && input.debtToIncome < 20 && input.requestedLoanAmount < 1000000 && input.age >= 18",
    "action": "output.setApprovalStatus(true); output.setMaximumPercentage(90); output.setInterestRate($(bank.prime_plus_half)); output.setProcessingFee(4000); output.setNotes(\"Approved!\")",
    "priority": 1,
    "description": "Eligibility for a 3.75% loan"
  },
  {
    "domain": "LOAN",
    "id": 2,
    "condition": "input.monthlySalary >= 4000 && input.creditScore >= 700 && input.debtToIncome < 25 && input.requestedLoanAmount < 250000 && input.age >= 18",
    "action": "output.setApprovalStatus(true); output.setMaximumPercentage(90); output.setInterestRate($(bank.prime_plus_one)); output.setProcessingFee(2000); output.setNotes(\"Approved!\")",
    "priority": 2,
    "description": "Eligibility for a 4.25% loan"
  },
  {
    "domain": "LOAN",
    "id": 3,
    "condition": "input.monthlySalary >= 3000 && input.creditScore >= 640 && input.debtToIncome < 30 && input.requestedLoanAmount < 100000 && input.age >= 18",
    "action": "output.setApprovalStatus(true); output.setMaximumPercentage(80); output.setInterestRate($(bank.prime_plus_three)); output.setProcessingFee(1000); output.setNotes(\"Approved!\")",
    "priority": 3,
    "description": "Eligibility for a 6.25% loan"
  },
  {
    "domain": "LOAN",
    "id": 4,
    "condition": "",
    "action": "output.setApprovalStatus(false); output.setNotes(\"So sorry. Your income, credit rating or debt to income ratio prevent a loan for this amount.\")",
    "priority": 99999,
    "description": "Default condition"
  }
]
```

Running the Ruleset

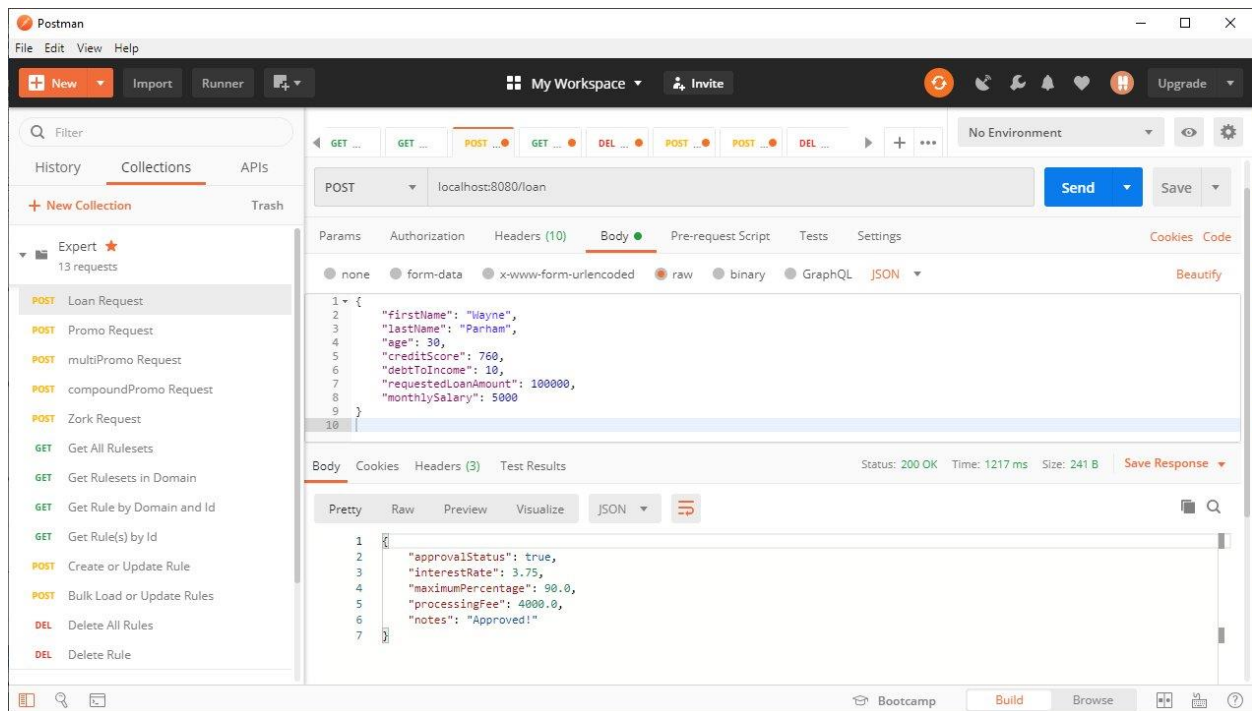
Now we're ready for the moment of truth. We can run the Loan ruleset, and see what it does. Create another POST method request that can deliver a JSON body. But this time, set it to the following URL:

<http://<servername>:8080/loan> (localhost when running on a local machine)

Copy and paste the following data into the body and click "Send."

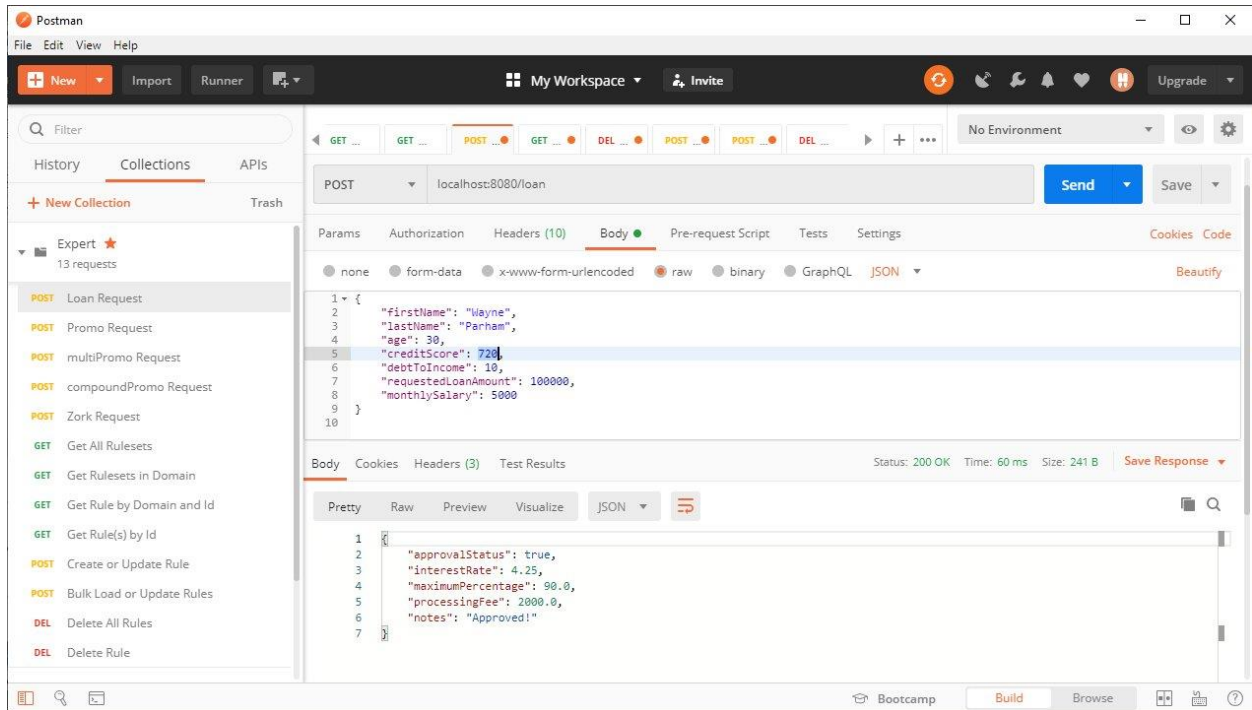
```
{
  "firstName": "Wayne",
  "lastName": "Parham",
  "age": 30,
  "creditScore": 760,
  "debtToIncome": 10,
  "requestedLoanAmount": 100000,
  "monthlySalary": 5000
}
```

This matches the rule that gives the best rate:

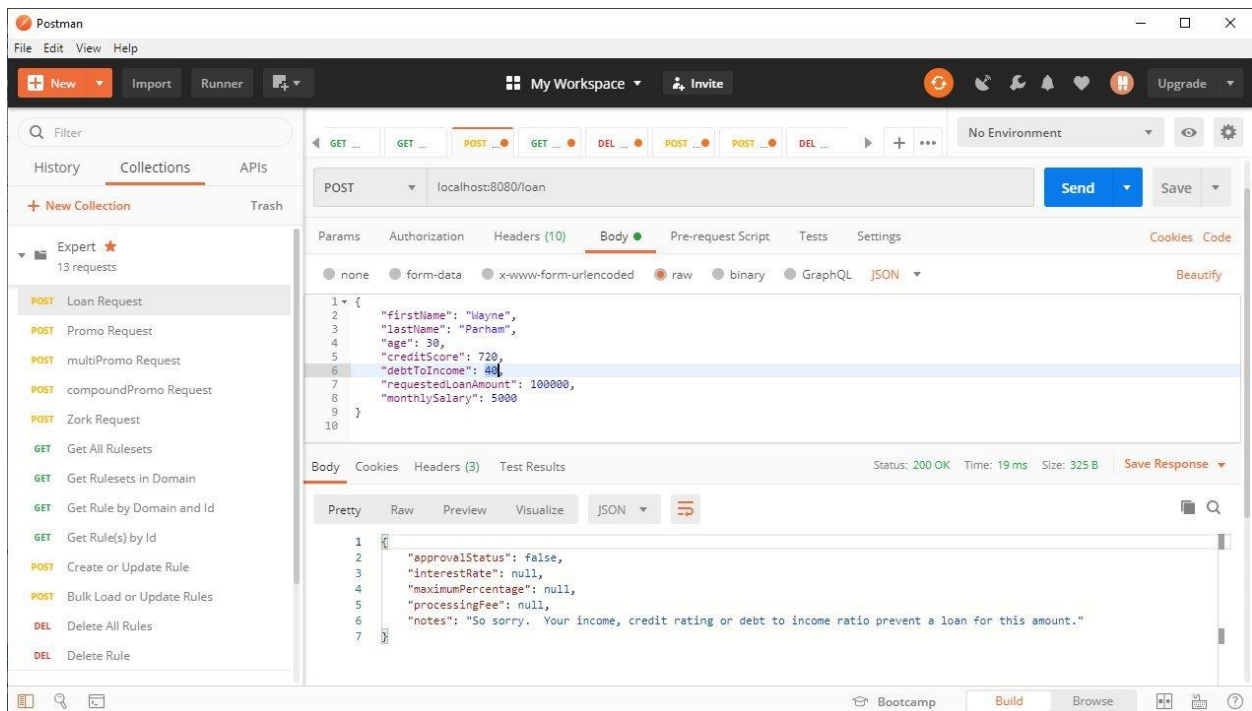


Now try a few other scenarios. Change Debt to Income ratio. Change the income. You can see that the rules work as designed.

Lower the credit score, and the interest rate increases:



Increase the debt to income ratio, and the loan application is rejected:



RulesDetails Implementation for Data containing Multiple Input Records

In the Loan Rules Engine, we use a single input structure and the rules acted upon it to create output having a single output structure. But what about scenarios where the input data is in the form of an array of input structures?

Let's create a Rules Engine where the input data is a shopping cart containing a list of items. The rules will be applied to this list to see if there are any promotional offers available. We will create rules that scan the cart for specific items, a combination of items, a certain quantity of items or an overall total amount purchased.

Input Definition for Data containing Multiple Input Records

So first, let's create the Input Definition:

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ShoppingCart {
    @Data
    public static class Item {
        String upc;
        String description;
        String category;
        Float price;
        Integer qty;
    }

    public Integer id;
    public ArrayList<Item> items;
}
```

Our input definition is implemented by a class called *ShoppingCart*. You will notice that it contains a static nested class called *Item*, which contains the details about each item in the cart. The *ShoppingCart* class defines an *ArrayList* of *Items*, and it also has an *ID*, which is useful for identifying a specific cart.

To make this input definition class useful, we will probably want to add some functions that examine the list of items. We might want to know things like the total value of the cart and the total number of items in the cart. We will probably want to be able to search the list of items for a specific one. So let's add some helper functions.

RulesDetails Helper Functions

```
public Float totalPrice() {
    Float total = (float) 0;
    for (Item item : items) {
        total += lineSubtotal(item);
    }
    return total;
}

public Float lineSubtotal(Item item) {
    return item.getQty() * item.getPrice();
}

public Integer itemCount() {
    int count = 0;
    for (Item item : items) {
        count += item.getQty();
    }
    return count;
}

public Integer categoryCount(String category) {
    int count = 0;
    for (Item item : items) {
        if (item.getCategory().equals(category)) {
            count += item.getQty();
        }
    }
    return count;
}

public boolean containsUpc(String upc) {
    for (Item item : items) {
        if (item.getUpc().equals(upc)) {
            return true;
        }
    }
    return false;
}

public boolean containsDescription(String description) {
    for (Item item : items) {
        if (item.getDescription().equals(description)) {
            return true;
        }
    }
    return false;
}
```

With these functions, our rules can check the total price of the contents of the cart, the number of items in the cart and the number of different categories of items. It can search for a specific item by UPC or by description.

Output Definition

The output data is described exactly the same way as it was in the Loan rules engine, which means it must return a single result. It is described in a structure called “PromoItem,” which is shown below.

```
public class PromoItem {
    Boolean status;
    String notes;
    String promoUpc;
    String promoDescription;
    String promoCategory;
    Float promoPrice;
    Integer promoQty;
}
```

As before in the Loan implementation, both input and output definitions are put into Java classes contained in a *RulesDetails* file structure. You will also define any special constants you might want to use in the VariableResolver. You will need to add the “PROMO” domain in the RulesDomain file. And you will need to add the imports, connection and mapping code to the restAPI. This is the same as described in the “RulesDetails Directory Structure” and the “Domain RestAPI Definition” for the Loan rules engine.

Promo Domain Ruleset

After creating the RulesDetails code and compiling, it’s time to load the ruleset. Run the *Expert Rules Engine* and bulk load the contents of the “*promo_rules.json*” file:

```
[
  {
    "domain": "PROMO",
    "id": 1,
    "condition": "input.totalPrice > 100",
    "action": "output.setStatus(true); output.setNotes(\"This cart has over a hundred bucks of stuff in it!\"); output.setPromoDescription(\"Free gift card\"); output.setPromoUpc(1283925739401); output.setPromoCategory(\"debit card\"); output.setPromoPrice(10); output.setPromoQty(1)",
    "priority": 10,
    "description": "> $100"
  },
]
```



```

{
  "domain": "PROMO",
  "id": 2,
  "condition": "input.containsDescription(\"wrench\")",
  "action": "output.setStatus(true);output.setNotes(\"Cart includes a
wrench.\");output.setPromoDescription(\"Free
battery.\");output.setPromoUpc(2840283167310);output.setPromoCategory(\"electronics\");ou
tput.setPromoPrice(1.50);output.setPromoQty(1)",
  "priority": 30,
  "description": "wrench"
},
{
  "domain": "PROMO",
  "id": 3,
  "condition": "input.categoryCount(\"tool\") > 1",
  "action": "output.setStatus(true);output.setNotes(\"Cart has multiple
tools.\");output.setPromoDescription(\"Free tool
case.\");output.setPromoUpc(6381623943311);output.setPromoCategory(\"tools\");output.set
PromoPrice(3.50);output.setPromoQty(1)",
  "priority": 20,
  "description": "wrench"
},
{
  "domain": "PROMO",
  "id": 4,
  "condition": "",
  "action": "output.setStatus(false); output.setNotes(\"This cart doesn't have anything
special in it.\")",
  "priority": 99999,
  "description": "wrench"
}
]

```

Notice how the domain-specific helper functions are used in the ruleset. The first rule checked the total cart price, to see if there were more than \$100 worth of items in the basket. The second rule checked for a specific item, although not by UPC but by description. It could have been UPC – and probably would have been, if you wanted to check for the presence of a specific item – but using name made the ruleset more readable for this example. And the third rule checked the number of items of a specific category. These are examples of how domain-specific helper functions are used.

Running the Ruleset

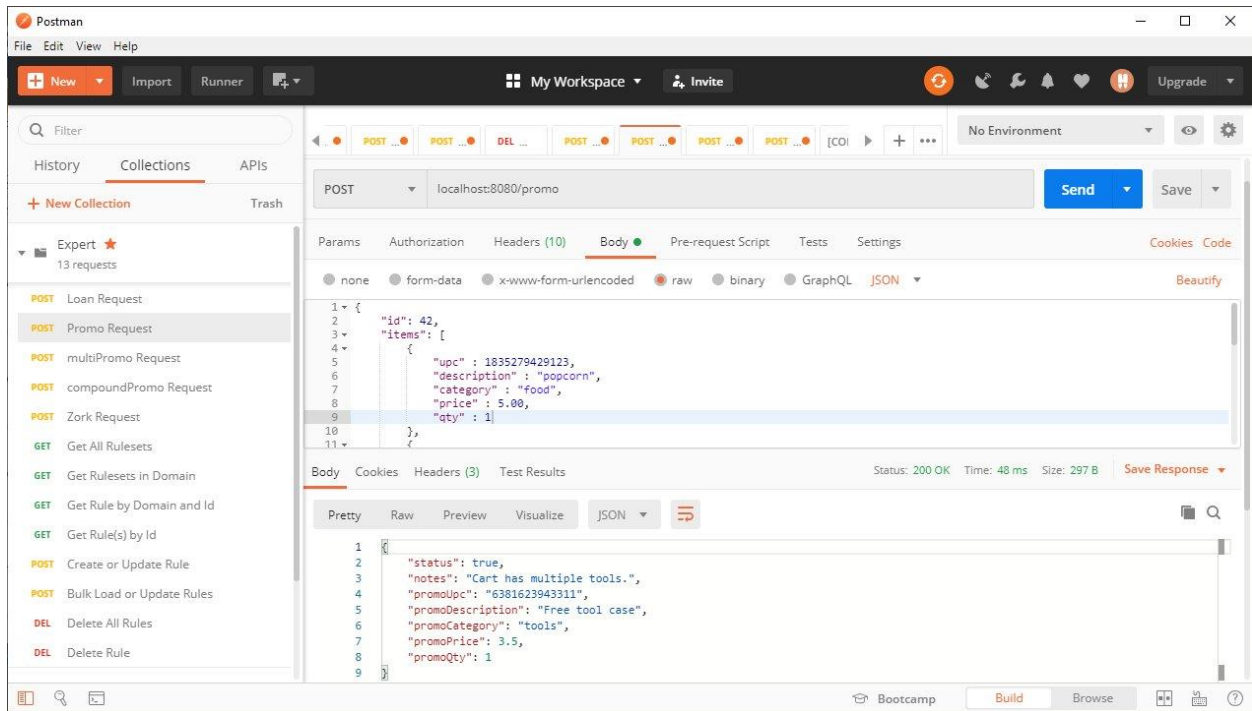
Now we can send a message containing a cart full of items. We can run the Promo ruleset, and see what promotional offer it presents us with.

`http://<servername>:8080/promo` (localhost when running on a local machine)

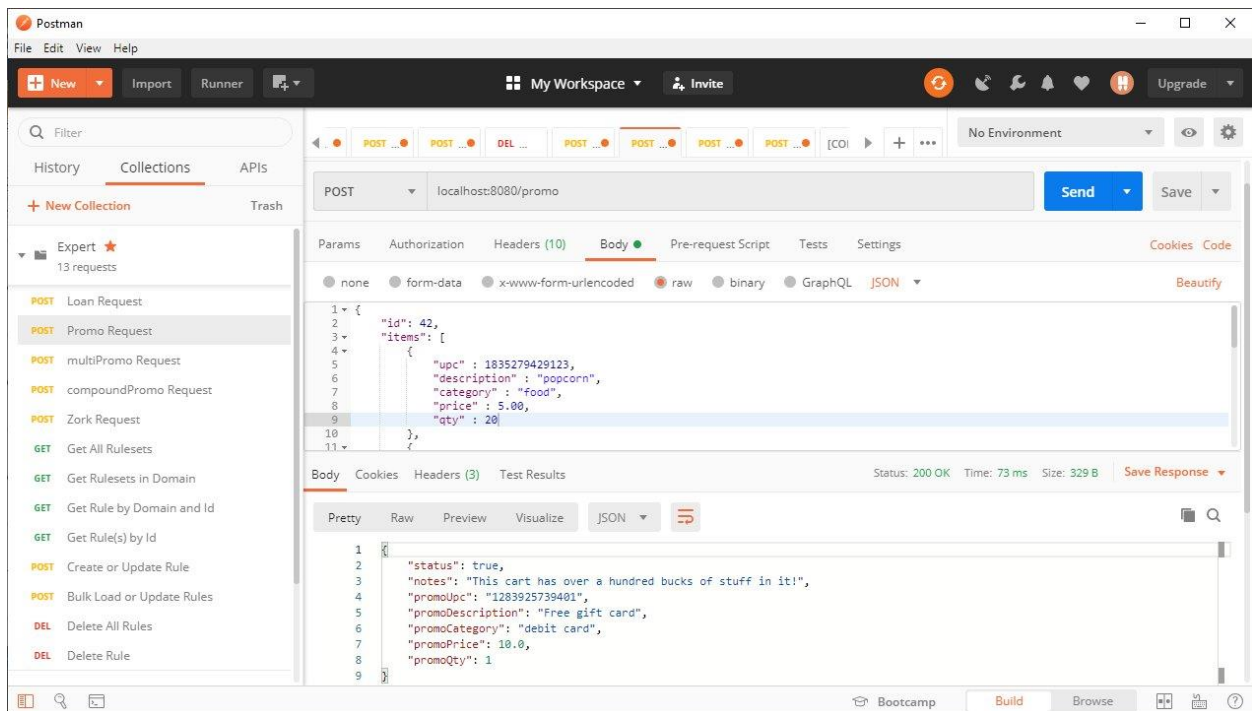
Copy and paste the following data into the body and click "Send."

```
{
  "id": 42,
  "items": [
    {
      "upc" : 1835279429123,
      "description" : "popcorn",
      "category" : "food",
      "price" : 5.00,
      "qty" : 1
    },
    {
      "upc" : 2837298353762,
      "description" : "screwdriver",
      "category" : "tool",
      "price" : 7.00,
      "qty" : 1
    },
    {
      "upc" : 2837232353762,
      "description" : "wrench",
      "category" : "tool",
      "price" : 23.00,
      "qty" : 1
    },
    {
      "upc" : 7729835729893,
      "description" : "light bulb",
      "category" : "electronics",
      "price" : 1.00,
      "qty" : 1
    }
  ]
}
```

This runs the ruleset and chooses our best promotional offer, a \$3.50 tool case:



Now edit the cart and change the quantity of popcorn bags from 1 to 20 for a different offer:



RestAPI Mapping to Support Multiple Output Records

The code to provide support for multiple output records is pretty simple. We change the `ResponseEntity` in the restAPI mapping to a List of output objects rather than to a single object, as in the previous implementations. We also set the last argument in the `rulesEngine.run()` function to “true,” which enables multi-record output from the rules engine. See the code below:

```
@PostMapping(value = "/multiPromo")
public ResponseEntity<?> postMultiPromo(@RequestBody CartItems cart) {
    log.info("Requesting analysis for cart id # " + cart.getId());
    List<PromoItems> response = (List<PromoItems>) rulesEngine.run(multiPromoInferenceEngine, cart, true);
    log.info("Result: " + response);
    if (response == null) {
        String errorMessage = "{\n\t\"status\": false,\"";
        errorMessage += "\n\t\"notes\": \" No matching rule. Please examine input data and ruleset.\n}";
        HttpHeaders responseHeaders = new HttpHeaders();
        responseHeaders.add("Content-Type", "application/json");
        return new ResponseEntity<>(errorMessage, responseHeaders, HttpStatus.OK);
    }
    else {
        return ResponseEntity.ok(response);
    }
}
```

Note: Understand that when multi-record output is enabled, rule priority becomes meaningless. Any rule that matches the input dataset as being true will create an output record. So you wouldn’t want to create a “default” rule with very low priority like you might in a single-output rules engine implementation.

MultiPromo Domain Ruleset

```
[
  {
    "domain": "MULTIPROMO",
    "id": 1,
    "condition": "input.totalPrice > 100",
    "action": "output.setStatus(true); output.setNotes(\"This cart has over a hundred bucks of
stuff in it!\"); output.setPromoDescription(\"Free gift card\");
output.setPromoUpc(1283925739401); output.setPromoCategory(\"debit card\");
output.setPromoPrice(10); output.setPromoQty(1)",
    "priority": 10,
    "description": "> $100"
  },
]
```

```

{
  "domain": "MULTIPROMO",
  "id": 2,
  "condition": "input.containsDescription(\"wrench\")",
  "action": "output.setStatus(true); output.setNotes(\"Cart includes a wrench.\");
output.setPromoDescription(\"Free battery\"); output.setPromoUpc(2840283167310);
output.setPromoCategory(\"electronics\"); output.setPromoPrice(1.50);
output.setPromoQty(1)",
  "priority": 30,
  "description": "wrench"
},
{
  "domain": "MULTIPROMO",
  "id": 3,
  "condition": "input.categoryCount(\"tool\") > 1",
  "action": "output.setStatus(true); output.setNotes(\"Cart has multiple tools.\");
output.setPromoDescription(\"Free tool case\"); output.setPromoUpc(6381623943311);
output.setPromoCategory(\"tools\"); output.setPromoPrice(3.50); output.setPromoQty(1)",
  "priority": 20,
  "description": "wrench"
}
]

```

You might notice that the only difference in this ruleset and the one for the single promo ruleset is the deletion of the default (priority 99999) condition, the one that said *“This cart doesn't have anything special in it.”*

Priority is set in these rules, but it doesn't matter. They all could have been set to null, in which case they would have defaulted to 100. But again, priority doesn't matter in a multi-output rule implementation.

Running the Ruleset

Let's send the exact same cart containing the same items we used in the Promo rules engine. This time, the URL to send our message is what we defined in the multiPromo Post mapping above:

<http://<servername>:8080/multiPromo> (localhost when running on a local machine)

Copy and paste the following data into the body and click "Send."

```
{
  "id": 42,
  "items": [
    {
      "upc" : 1835279429123,
      "description" : "popcorn",
      "category" : "food",
      "price" : 5.00,
      "qty" : 1
    },
    {
      "upc" : 2837298353762,
      "description" : "screwdriver",
      "category" : "tool",
      "price" : 7.00,
      "qty" : 1
    },
    {
      "upc" : 2837232353762,
      "description" : "wrench",
      "category" : "tool",
      "price" : 23.00,
      "qty" : 1
    },
    {
      "upc" : 7729835729893,
      "description" : "light bulb",
      "category" : "electronics",
      "price" : 1.00,
      "qty" : 1
    }
  ]
}
```

This time, the result set looks like this:

```
[
  {
    "status": true,
    "notes": "Cart includes a wrench.",
    "promoUpc": "2840283167310",
    "promoDescription": "Free battery",
    "promoCategory": "electronics",
    "promoPrice": 1.5,
    "promoQty": 1
  },

```

```

{
  "status": true,
  "notes": "Cart has multiple tools.",
  "promoUpc": "6381623943311",
  "promoDescription": "Free tool case",
  "promoCategory": "tools",
  "promoPrice": 3.5,
  "promoQty": 1
}
]

```

We got a free battery because we bought a wrench, and a free tool case because we bought multiple tools.

Now, let's edit the cart and increase the number of popcorn bags from 1 to 20, like we did before. Let's see what offers the ruleset gives us then:

```

[
  {
    "status": true,
    "notes": "This cart has over a hundred bucks of stuff in it!",
    "promoUpc": "1283925739401",
    "promoDescription": "Free gift card",
    "promoCategory": "debit card",
    "promoPrice": 10.0,
    "promoQty": 1
  },
  {
    "status": true,
    "notes": "Cart includes a wrench.",
    "promoUpc": "2840283167310",
    "promoDescription": "Free battery",
    "promoCategory": "electronics",
    "promoPrice": 1.5,
    "promoQty": 1
  },
  {
    "status": true,
    "notes": "Cart has multiple tools.",
    "promoUpc": "6381623943311",
    "promoDescription": "Free tool case",
    "promoCategory": "tools",
    "promoPrice": 3.5,
    "promoQty": 1
  }
]

```

As expected, we still got the free battery and the tool case, but this time the increased total value of the cart now triggers the \$100 rule, giving us a \$10.00 gift card too.

RulesDetails Implementation for Compound Rulesets

So now we've created rules engines that processed a single record and created a single record. We've created rules engines that processed multiple input records and made a single output record, and we've also created multiple output records. But what if we need to create multiple output records and then run those through another ruleset to narrow down the results?

We can do this by cascading rulesets to create a compound ruleset.

To do this, you will essentially create two separate RulesDetails implementations, with the output of one feeding the input of the other. Naturally, you will want the data formats to be similar with the same kinds of fields having the same kind of data.

One or both of the Input Definitions can have helper functions. It may be useful to have helper functions on the intermediate step to get totals and other kinds of information for the second ruleset to act upon, for example.

Let's look at a cascaded rules engine to further our capabilities in promotional offerings selections. To do that, we will create a first stage and call it `compoundPromo`. And we will create a second stage and call it `promoAggregate`.

In this implementation, the `compoundPromo` input definition is very much like the "ShoppingCart" definition in the original Promo rules engine, and also much like the "CartItems" definition in the multiPromo implementation. Each has the same array of Items, and they all have the same helper functions to calculate things like cart totals, item count and so on.

Similarly, the `compoundPromo` output definition is almost exactly the same as the original Promo's output definition, and again, it's also similar to the multiPromo output definition. Of course, the `compoundPromo` output is set to deliver a List, to provide multiple outputs in the restAPI. One minor difference is that the `compoundPromo` output includes a `cartTotal` field, as a pass-through for the second ruleset.

RestAPI Mapping to Support Cascaded Compound Records

A rules engine that has cascaded rulesets need to run the first ruleset with multi-record output, and transfer its results to the input class of the second implementation. That second ruleset can be run with multi-record output or with a single output, whichever is appropriate for your need. The one shown below creates a single record output result.

```
@PostMapping(value = "/compoundPromo")
public ResponseEntity<?> postCompoundPromo(@RequestBody Cart cart) {
    log.info("Requesting analysis for cart id # " + cart.getId());
    List<Promo> prelims = (List<Promo>) rulesEngine.run(compoundPromoInferenceEngine, cart, true);
    log.info("Preliminary result: " + prelims);
    ArrayList<PromoAggregateInput.Promo> availablePromos = new ArrayList<>();
    for(Promo prelim : prelims) {
        PromoAggregateInput.Promo inputPromo = new PromoAggregateInput.Promo();
        inputPromo.setStatus(prelim.getStatus());
        inputPromo.setNotes(prelim.getNotes());
        inputPromo.setCartTotal(prelim.getCartTotal());
        inputPromo.setPromoUpc(prelim.getPromoUpc());
        inputPromo.setPromoDescription(prelim.getPromoDescription());
        inputPromo.setPromoCategory(prelim.getPromoCategory());
        inputPromo.setPromoPrice(prelim.getPromoPrice());
        inputPromo.setPromoQty(prelim.getPromoQty());
        availablePromos.add(inputPromo);
    }
    PromoAggregateInput aggregate = new PromoAggregateInput(availablePromos);
    PromoAggregateOutput response = (PromoAggregateOutput) rulesEngine.run(aggregatePromoInferenceEngine,
aggregate, false);
    log.info("Final result: " + response);
    if (response == null) {
        response = PromoAggregateOutput.builder()
            .notes("No matching rule. Please examine input data and ruleset.")
            .build();
    }
    return ResponseEntity.ok(response);
}
```

This implementation will first create a list of all qualifying promotional offers in the first ruleset, and then will select one from them using the second ruleset. It could have just as easily created a list in the second ruleset, if the goal was to allow for the possibility of more than one promotion from a cart.

CompoundPromo Domain Ruleset

```
[
  {
    "domain": "COMPOUNDPROMO",
    "id": 1,
    "condition": "input.totalPrice > 250",
    "action": "output.setStatus(true); output.setNotes(\"This cart has over $250 worth of stuff in it!\"); output.setPromoDescription(\"Free gift card\"); output.setPromoUpc(1283925739402); output.setPromoCategory(\"debit card\"); output.setPromoPrice(25); output.setPromoQty(1); output.setCartTotal(input.totalPrice)",
    "priority": 100,
    "description": "> $250"
  },
  {
    "domain": "COMPOUNDPROMO",
    "id": 2,
    "condition": "input.totalPrice > 100",
    "action": "output.setStatus(true); output.setNotes(\"This cart has over a hundred bucks of stuff in it!\"); output.setPromoDescription(\"Free gift card\"); output.setPromoUpc(1283925739401); output.setPromoCategory(\"debit card\"); output.setPromoPrice(10); output.setPromoQty(1); output.setCartTotal(input.totalPrice)",
    "priority": 100,
    "description": "> $100"
  },
  {
    "domain": "COMPOUNDPROMO",
    "id": 3,
    "condition": "input.containsDescription(\"wrench\")",
    "action": "output.setStatus(true); output.setNotes(\"Cart includes a wrench.\"); output.setPromoDescription(\"Free battery\"); output.setPromoUpc(2840283167310); output.setPromoCategory(\"electronics\"); output.setPromoPrice(1.50); output.setPromoQty(1); output.setCartTotal(input.totalPrice)",
    "priority": 100,
    "description": "wrench"
  },
  {
    "domain": "COMPOUNDPROMO",
    "id": 4,
    "condition": "input.categoryCount(\"tool\") > 1",
    "action": "output.setStatus(true); output.setNotes(\"Cart has multiple tools.\"); output.setPromoDescription(\"Free tool case\"); output.setPromoUpc(6381623943311); output.setPromoCategory(\"tools\"); output.setPromoPrice(3.50); output.setPromoQty(1); output.setCartTotal(input.totalPrice)",
    "priority": 100,
    "description": "multiple tools"
  },
]
```

```

{
  "domain": "COMPOUNDPROMO",
  "id": 5,
  "condition": "input.categoryCount(\"electronics\") > 3",
  "action": "output.setStatus(true); output.setNotes(\"Cart has three or more electronics devices.\"); output.setPromoDescription(\"Free digital download\"); output.setPromoUpc(7241623943311); output.setPromoCategory(\"electronics\"); output.setPromoPrice(10);output.setPromoQty(1); output.setCartTotal(input.totalPrice)",
  "priority": 100,
  "description": "multiple electronics devices"
},
{
  "domain": "COMPOUNDPROMO",
  "id": 6,
  "condition": "(input.categoryCount(\"food\") > 5) && (input.totalPrice > 50)",
  "action": "output.setStatus(true); output.setNotes(\"Cart has five or more food items.\"); output.setPromoDescription(\"Free kumquat\"); output.setPromoUpc(9462738943311); output.setPromoCategory(\"food\"); output.setPromoPrice(6);output.setPromoQty(1); output.setCartTotal(input.totalPrice)",
  "priority": 100,
  "description": "five food items and cart > $50"
}
]

```

PromoAggregate Domain Ruleset

```

[
  {
    "domain": "PROMOAGGREGATE",
    "id": 1,
    "condition": "input.bestInCategory(\"food\") != null",
    "action": "output.setStatus(true); output.setNotes(input.getNotes(input.bestInCategory(\"food\"))); output.setCartTotal(input.getCartTotal(input.bestInCategory(\"food\"))); output.setPromoUpc(input.getPromoUpc(input.bestInCategory(\"food\"))); output.setPromoDescription(input.getPromoDescription(input.bestInCategory(\"food\"))); output.setPromoCategory(input.getPromoCategory(input.bestInCategory(\"food\"))); output.setPromoPrice(input.getPromoPrice(input.bestInCategory(\"food\"))); output.setPromoQty(input.getPromoQty(input.bestInCategory(\"food\")))",
    "priority": 5,
    "description": "Food item promo, if available."
  },
]

```

```

{
  "domain": "PROMOAGGREGATE",
  "id": 2,
  "condition": "input.bestInCategory(\"electronics\") != null",
  "action": "output.setStatus(true); output.setNotes(input.getNotes(input.bestInCategory(\"electronics\"))); output.setCartTotal(input.getCartTotal(input.bestInCategory(\"electronics\"))); output.setPromoUpc(input.getPromoUpc(input.bestInCategory(\"electronics\"))); output.setPromoDescription(input.getPromoDescription(input.bestInCategory(\"electronics\"))); output.setPromoCategory(input.getPromoCategory(input.bestInCategory(\"electronics\"))); output.setPromoPrice(input.getPromoPrice(input.bestInCategory(\"electronics\"))); output.setPromoQty(input.getPromoQty(input.bestInCategory(\"electronics\")))",
  "priority": 10,
  "description": "Electronics item promo, if available."
},
{
  "domain": "PROMOAGGREGATE",
  "id": 3,
  "condition": "input.best != null",
  "action": "output.setStatus(true); output.setNotes(input.getNotes(input.best)); output.setCartTotal(input.getCartTotal(input.best)); output.setPromoUpc(input.getPromoUpc(input.best)); output.setPromoDescription(input.getPromoDescription(input.best)); output.setPromoCategory(input.getPromoCategory(input.best)); output.setPromoPrice(input.getPromoPrice(input.best)); output.setPromoQty(input.getPromoQty(input.best))",
  "priority": 100,
  "description": "Any promo available."
},
{
  "domain": "PROMOAGGREGATE",
  "id": 4,
  "condition": "(input.cartTotal > 100) && (input.bestInCategory(\"debit card\") != null)",
  "action": "output.setStatus(true); output.setNotes(input.getNotes(input.bestInCategory(\"debit card\"))); output.setCartTotal(input.getCartTotal(input.bestInCategory(\"debit card\"))); output.setPromoUpc(input.getPromoUpc(input.bestInCategory(\"debit card\"))); output.setPromoDescription(input.getPromoDescription(input.bestInCategory(\"debit card\"))); output.setPromoCategory(input.getPromoCategory(input.bestInCategory(\"debit card\"))); output.setPromoPrice(input.getPromoPrice(input.bestInCategory(\"debit card\"))); output.setPromoQty(input.getPromoQty(input.bestInCategory(\"debit card\")))",
  "priority": 1,
  "description": "Total purchase exceeds $100 and debit card promo is available."
},
{
  "domain": "PROMOAGGREGATE",
  "id": 5,
  "condition": "",
  "action": "output.setStatus(false); output.setNotes(\"No promotional offers available.\")",
  "priority": 99999,
  "description": "No promo available."
}
]

```

Running the Ruleset(s)

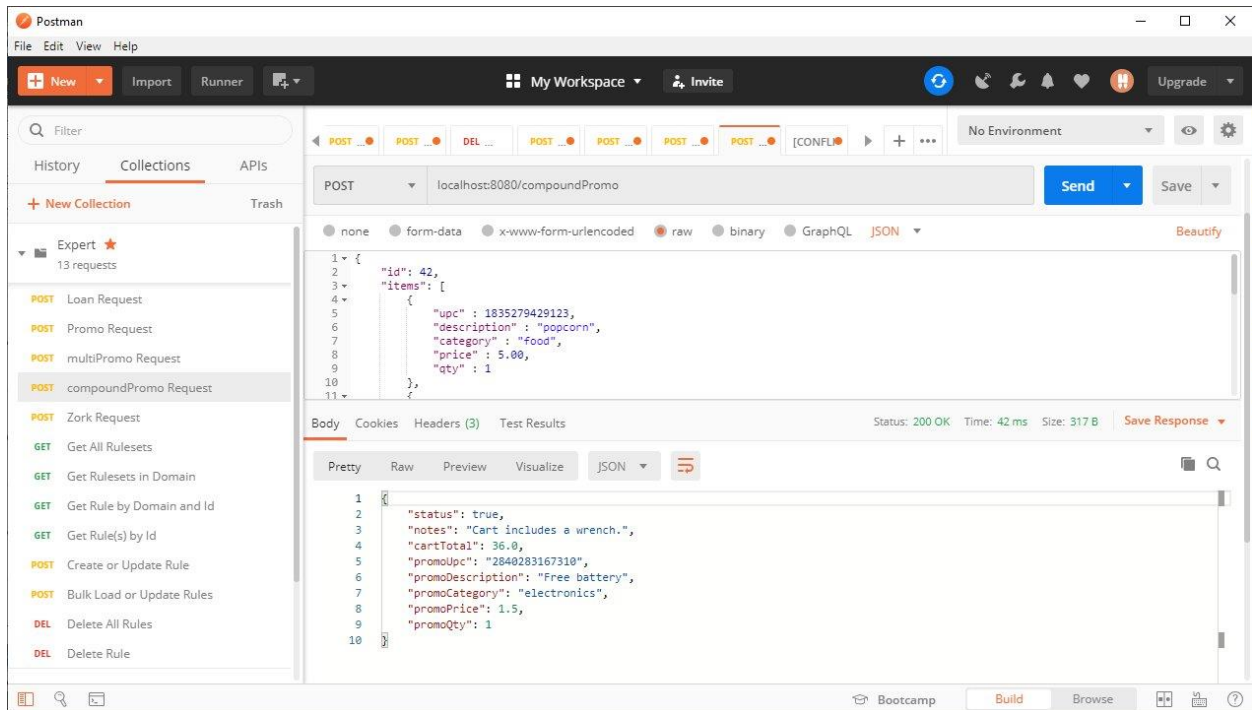
Now we're all setup and we can run our rules engine. The URL is shown below.

<http://<servername>:8080/compoundPromo> (localhost on a local machine)

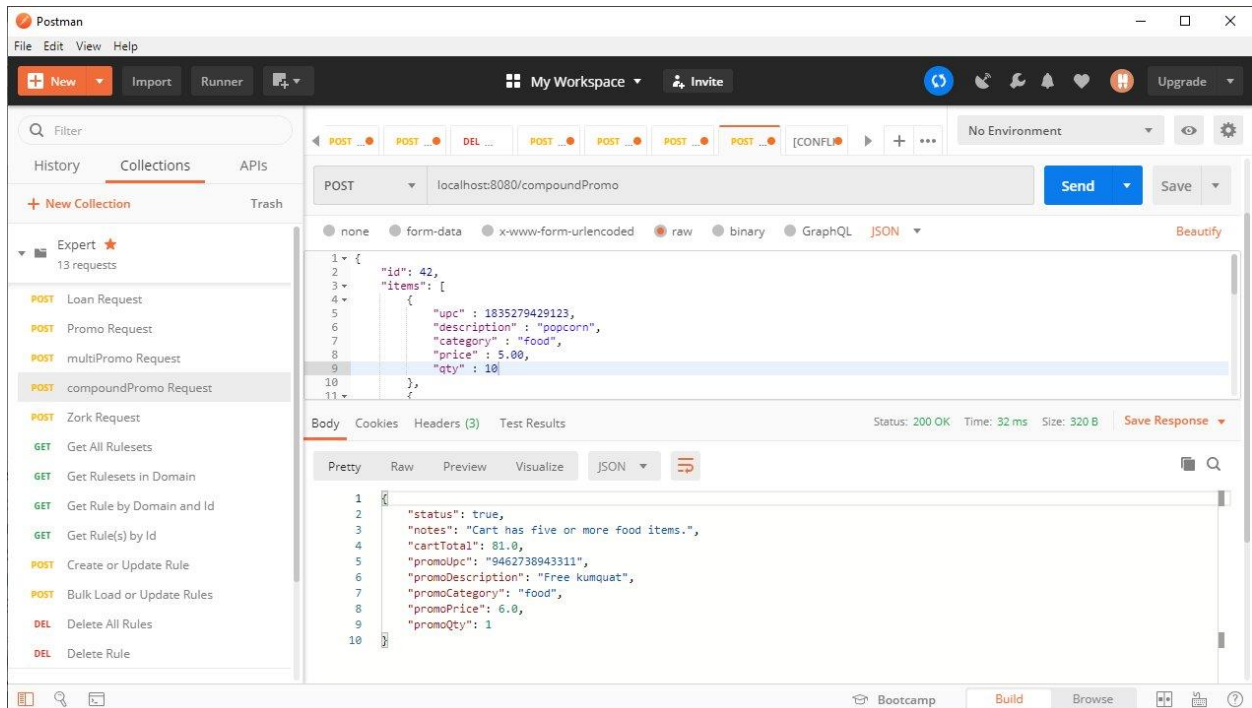
We'll start off with the same cart we've used in previous examples, so copy and paste the following data into the body and click "Send."

```
{
  "id": 42,
  "items": [
    {
      "upc" : 1835279429123,
      "description" : "popcorn",
      "category" : "food",
      "price" : 5.00,
      "qty" : 1
    },
    {
      "upc" : 2837298353762,
      "description" : "screwdriver",
      "category" : "tool",
      "price" : 7.00,
      "qty" : 1
    },
    {
      "upc" : 2837232353762,
      "description" : "wrench",
      "category" : "tool",
      "price" : 23.00,
      "qty" : 1
    },
    {
      "upc" : 7729835729893,
      "description" : "light bulb",
      "category" : "electronics",
      "price" : 1.00,
      "qty" : 1
    }
  ]
}
```

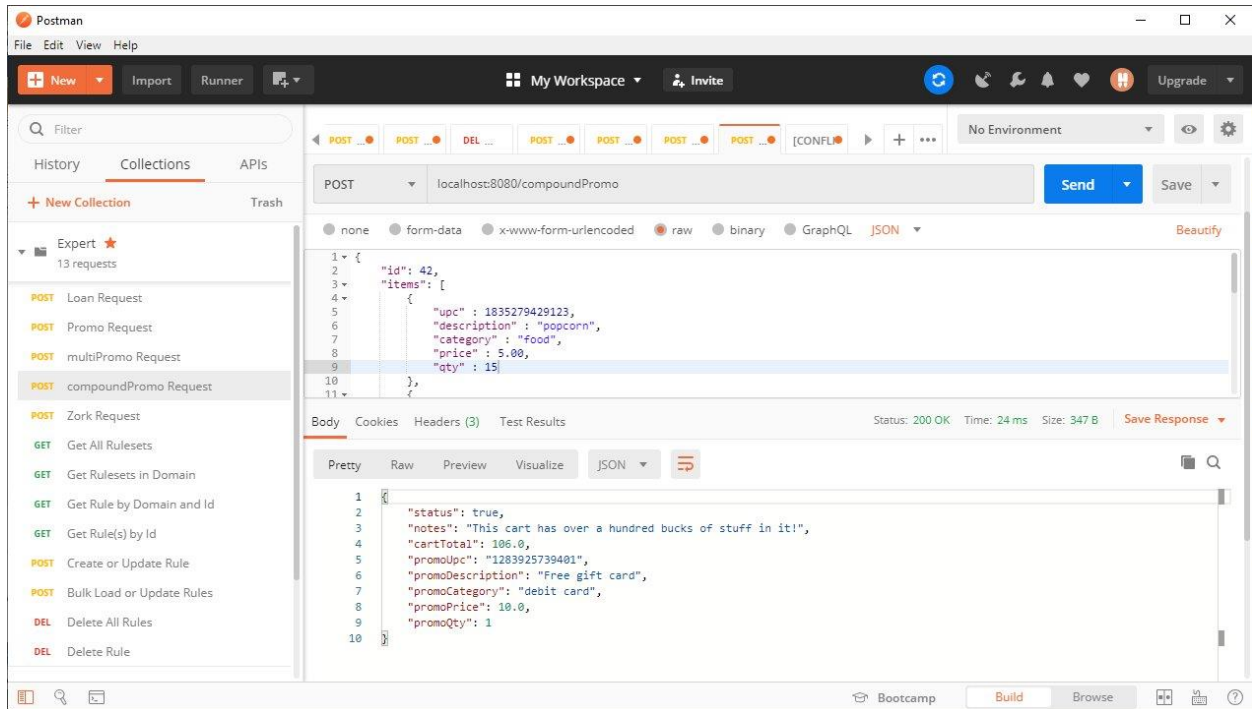
The rulesets choose the \$1.50 battery for this cart:



If we increase our quantity of popcorn bags to ten, we now are given a free kumquat:



And if we increase popcorn count to 15, we now are offered a \$10.00 gift card:



So you can see that we are able to make more choices when use cascaded rulesets. The first ruleset created a list of qualifying promotions. Each of them were promotional offers that we could be presented with. But we don't want to give away all of them, nor do we necessarily want to give the one that costs the most. We wanted to be able to choose from the list selectively, perhaps picking items that we are overstocked in or those that are time-sensitive, like food. The rules in the promoAggregator are what determine this.


```

2020-04-17 11:41:35.286 INFO 11676 --- [          main]
org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache
Tomcat/9.0.16]
2020-04-17 11:41:35.301 INFO 11676 --- [          main]
o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library
which allows optimal performance in production environments was not found on the
java.library.path: [C:\Program
Files\Java\jdk1.8.0_241\bin;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;C:\
Program Files\Java\jdk1.8.0_241\bin;C:\Program Files\Maven\mvn.3.6.3\bin;C:\Program
Files (x86)\Common
Files\Oracle\Java\javapath;C:\ProgramData\Oracle\Java\javapath;C:\Windows\system32;C:\
Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Window
s\System32\OpenSSH\;C:\Program Files\PuTTY\;C:\Program Files\MySQL\MySQL Shell
8.0\bin\;C:\Users\wayne\AppData\Local\Microsoft\WindowsApps;;C:\Program
Files\JetBrains\IntelliJ\bin;.]
2020-04-17 11:41:35.471 INFO 11676 --- [          main]
o.a.c.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded
WebApplicationContext
2020-04-17 11:41:35.471 INFO 11676 --- [          main]
o.s.web.context.ContextLoader : Root WebApplicationContext: initialization
completed in 4227 ms
2020-04-17 11:41:35.811 INFO 11676 --- [          main]
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2020-04-17 11:41:36.646 INFO 11676 --- [          main]
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2020-04-17 11:41:36.745 INFO 11676 --- [          main]
o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
name: default
...]
2020-04-17 11:41:37.081 INFO 11676 --- [          main] org.hibernate.Version
: HHH000412: Hibernate Core {5.3.7.Final}
2020-04-17 11:41:37.081 INFO 11676 --- [          main]
org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
2020-04-17 11:41:37.542 INFO 11676 --- [          main]
o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations
{5.0.4.Final}
2020-04-17 11:41:37.789 INFO 11676 --- [          main]
org.hibernate.dialect.Dialect : HHH000400: Using dialect:
org.hibernate.dialect.MySQL8Dialect
2020-04-17 11:41:38.833 INFO 11676 --- [          main]
j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for
persistence unit 'default'
2020-04-17 11:41:40.247 INFO 11676 --- [          main]
o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService
'applicationTaskExecutor'
2020-04-17 11:41:40.711 INFO 11676 --- [          main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with
context path ''
2020-04-17 11:41:40.711 INFO 11676 --- [          main]
com.parhamdata.expert.ExpertApplication : Started ExpertApplication in 10.281 seconds
(JVM running for 11.14)

```

2020-04-17 11:43:43.505 INFO 11676 --- [nio-8080-exec-2]
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet
'dispatcherServlet'
2020-04-17 11:43:43.505 INFO 11676 --- [nio-8080-exec-2]
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-04-17 11:43:43.519 INFO 11676 --- [nio-8080-exec-2]
o.s.web.servlet.DispatcherServlet : Completed initialization in 14 ms
2020-04-17 11:43:43.802 INFO 11676 --- [nio-8080-exec-2]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis about loan for user:
UserDetails(age=30, lastName=Parham, firstName=Wayne, creditScore=760,
debtToIncome=10.0, monthlySalary=5000.0, requestedLoanAmount=100000.0)
2020-04-17 11:43:43.849 INFO 11676 --- [nio-8080-exec-2]
o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
2020-04-17 11:43:44.566 INFO 11676 --- [nio-8080-exec-2]
c.parhamdata.expert.controllers.RestAPI : Result: LoanDetails(approvalStatus=true,
interestRate=3.75, maximumPercentage=90.0, processingFee=4000.0, notes=Approved!)
2020-04-17 11:49:36.068 INFO 11676 --- [nio-8080-exec-5]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis about loan for user:
UserDetails(age=30, lastName=Parham, firstName=Wayne, creditScore=720,
debtToIncome=10.0, monthlySalary=5000.0, requestedLoanAmount=100000.0)
2020-04-17 11:49:36.081 INFO 11676 --- [nio-8080-exec-5]
c.parhamdata.expert.controllers.RestAPI : Result: LoanDetails(approvalStatus=true,
interestRate=4.25, maximumPercentage=90.0, processingFee=2000.0, notes=Approved!)
2020-04-17 11:50:47.519 INFO 11676 --- [nio-8080-exec-7]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis about loan for user:
UserDetails(age=30, lastName=Parham, firstName=Wayne, creditScore=720,
debtToIncome=40.0, monthlySalary=5000.0, requestedLoanAmount=100000.0)
2020-04-17 11:50:47.525 INFO 11676 --- [nio-8080-exec-7]
c.parhamdata.expert.controllers.RestAPI : Result: LoanDetails(approvalStatus=false,
interestRate=null, maximumPercentage=null, processingFee=null, notes=So sorry. Your
income, credit rating or debt to income ratio prevent a loan for this amount.)
2020-04-17 16:10:36.964 INFO 11676 --- [nio-8080-exec-1]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 77
2020-04-17 16:10:36.971 INFO 11676 --- [nio-8080-exec-1]
c.parhamdata.expert.controllers.RestAPI : Result: PromoItem(status=true, notes=Cart
has multiple tools., promoUpc=6381623943311, promoDescription=Free tool case,
promoCategory=tools, promoPrice=3.5, promoQty=1)
2020-04-17 16:13:39.257 INFO 11676 --- [nio-8080-exec-4]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42
2020-04-17 16:13:39.263 INFO 11676 --- [nio-8080-exec-4]
c.parhamdata.expert.controllers.RestAPI : Result: PromoItem(status=true, notes=Cart
has multiple tools., promoUpc=6381623943311, promoDescription=Free tool case,
promoCategory=tools, promoPrice=3.5, promoQty=1)
2020-04-17 16:16:37.977 INFO 11676 --- [nio-8080-exec-6]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42
2020-04-17 16:16:37.983 INFO 11676 --- [nio-8080-exec-6]
c.parhamdata.expert.controllers.RestAPI : Result: PromoItem(status=true, notes=Cart
has multiple tools., promoUpc=6381623943311, promoDescription=Free tool case,
promoCategory=tools, promoPrice=3.5, promoQty=1)
2020-04-17 16:16:45.935 INFO 11676 --- [nio-8080-exec-9]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42

2020-04-17 16:16:45.940 INFO 11676 --- [nio-8080-exec-9]
c.parhamdata.expert.controllers.RestAPI : Result: PromoItem(status=true, notes=Cart has multiple tools., promoUpc=6381623943311, promoDescription=Free tool case, promoCategory=tools, promoPrice=3.5, promoQty=1)
2020-04-17 16:16:57.096 INFO 11676 --- [io-8080-exec-10]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42
2020-04-17 16:16:57.102 INFO 11676 --- [io-8080-exec-10]
c.parhamdata.expert.controllers.RestAPI : Result: PromoItem(status=true, notes=This cart has over a hundred bucks of stuff in it!, promoUpc=1283925739401, promoDescription=Free gift card, promoCategory=debit card, promoPrice=10.0, promoQty=1)
2020-04-17 17:26:20.426 INFO 11676 --- [nio-8080-exec-1]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42
2020-04-17 17:26:20.437 INFO 11676 --- [nio-8080-exec-1]
c.parhamdata.expert.controllers.RestAPI : Result: [PromoItems(status=true, notes=Cart includes a wrench., promoUpc=2840283167310, promoDescription=Free battery, promoCategory=electronics, promoPrice=1.5, promoQty=1), PromoItems(status=true, notes=Cart has multiple tools., promoUpc=6381623943311, promoDescription=Free tool case, promoCategory=tools, promoPrice=3.5, promoQty=1)]
2020-04-17 17:28:40.711 INFO 11676 --- [nio-8080-exec-4]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42
2020-04-17 17:28:40.726 INFO 11676 --- [nio-8080-exec-4]
c.parhamdata.expert.controllers.RestAPI : Result: [PromoItems(status=true, notes=This cart has over a hundred bucks of stuff in it!, promoUpc=1283925739401, promoDescription=Free gift card, promoCategory=debit card, promoPrice=10.0, promoQty=1), PromoItems(status=true, notes=Cart includes a wrench., promoUpc=2840283167310, promoDescription=Free battery, promoCategory=electronics, promoPrice=1.5, promoQty=1), PromoItems(status=true, notes=Cart has multiple tools., promoUpc=6381623943311, promoDescription=Free tool case, promoCategory=tools, promoPrice=3.5, promoQty=1)]
2020-04-18 12:16:34.187 WARN 11676 --- [l-1 housekeeper]
c.parhamdata.expert.controllers.RestAPI : Requesting rulesets for 'COMPOUNDPROMO'.
2020-04-18 12:45:41.062 INFO 11676 --- [nio-8080-exec-9]
c.parhamdata.expert.controllers.RestAPI : Found 6 rules in the COMPOUNDPROMO domain.
2020-04-18 12:47:39.135 INFO 11676 --- [nio-8080-exec-2]
c.parhamdata.expert.controllers.RestAPI : Requesting rulesets for 'PROMOAGGREGATE'.
2020-04-18 12:47:39.143 INFO 11676 --- [nio-8080-exec-2]
c.parhamdata.expert.controllers.RestAPI : Found 5 rules in the PROMOAGGREGATE domain.
2020-04-18 13:42:52.331 INFO 11676 --- [nio-8080-exec-8]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42
2020-04-18 13:42:52.339 INFO 11676 --- [nio-8080-exec-8]
c.parhamdata.expert.controllers.RestAPI : Preliminary result: [Promo(status=true, notes=Cart includes a wrench., cartTotal=36.0, promoUpc=2840283167310, promoDescription=Free battery, promoCategory=electronics, promoPrice=1.5, promoQty=1), Promo(status=true, notes=Cart has multiple tools., cartTotal=36.0, promoUpc=6381623943311, promoDescription=Free tool case, promoCategory=tools, promoPrice=3.5, promoQty=1)]
2020-04-18 13:42:52.347 INFO 11676 --- [nio-8080-exec-8]
c.parhamdata.expert.controllers.RestAPI : Final result:
PromoAggregateOutput(status=true, notes=Cart includes a wrench., cartTotal=36.0, promoUpc=2840283167310, promoDescription=Free battery, promoCategory=electronics, promoPrice=1.5, promoQty=1)

```
2020-04-18 13:49:16.957 INFO 11676 --- [nio-8080-exec-7]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42
2020-04-18 13:49:16.969 INFO 11676 --- [nio-8080-exec-7]
c.parhamdata.expert.controllers.RestAPI : Preliminary result: [Promo(status=true,
notes=Cart includes a wrench., cartTotal=81.0, promoUpc=2840283167310,
promoDescription=Free battery, promoCategory=electronics, promoPrice=1.5, promoQty=1),
Promo(status=true, notes=Cart has multiple tools., cartTotal=81.0,
promoUpc=6381623943311, promoDescription=Free tool case, promoCategory=tools,
promoPrice=3.5, promoQty=1), Promo(status=true, notes=Cart has five or more food
items., cartTotal=81.0, promoUpc=9462738943311, promoDescription=Free kumquat,
promoCategory=food, promoPrice=6.0, promoQty=1)]
2020-04-18 13:49:16.973 INFO 11676 --- [nio-8080-exec-7]
c.parhamdata.expert.controllers.RestAPI : Final result:
PromoAggregateOutput(status=true, notes=Cart has five or more food items.,
cartTotal=81.0, promoUpc=9462738943311, promoDescription=Free kumquat,
promoCategory=food, promoPrice=6.0, promoQty=1)
2020-04-18 13:52:09.705 INFO 11676 --- [nio-8080-exec-3]
c.parhamdata.expert.controllers.RestAPI : Requesting analysis for cart id # 42
2020-04-18 13:52:09.717 INFO 11676 --- [nio-8080-exec-3]
c.parhamdata.expert.controllers.RestAPI : Preliminary result: [Promo(status=true,
notes=This cart has over a hundred bucks of stuff in it!, cartTotal=106.0,
promoUpc=1283925739401, promoDescription=Free gift card, promoCategory=debit card,
promoPrice=10.0, promoQty=1), Promo(status=true, notes=Cart includes a wrench.,
cartTotal=106.0, promoUpc=2840283167310, promoDescription=Free battery,
promoCategory=electronics, promoPrice=1.5, promoQty=1), Promo(status=true, notes=Cart
has multiple tools., cartTotal=106.0, promoUpc=6381623943311, promoDescription=Free
tool case, promoCategory=tools, promoPrice=3.5, promoQty=1), Promo(status=true,
notes=Cart has five or more food items., cartTotal=106.0, promoUpc=9462738943311,
promoDescription=Free kumquat, promoCategory=food, promoPrice=6.0, promoQty=1)]
2020-04-18 13:52:09.721 INFO 11676 --- [nio-8080-exec-3]
c.parhamdata.expert.controllers.RestAPI : Final result:
PromoAggregateOutput(status=true, notes=This cart has over a hundred bucks of stuff in
it!, cartTotal=106.0, promoUpc=1283925739401, promoDescription=Free gift card,
promoCategory=debit card, promoPrice=10.0, promoQty=1)
```