P4

Due Date: Nov. 10, 2019

Overview

This project will continue where P3 left off. We will introduce some new elements into our system, though we will not add them to our OWL ontology. We will instead represent them as Java objects, and process them through rule-based programming using Drools.
You can choose to use the provided OWL and Java code for P3, as long as it is acknowledged in your submitted code or a separate README file.

Getting started

- Review the documentation link on http://drools.org/
    - Read Chapter 3 for useful background. Concentrate on Chapter 4 to learn about Drools rule syntax with examples.
- Integrate Drools with Play framework.
    - Refer to https://github.com/jamesward/play-drools
        - Note that this example is a few years old, and it is meant to be used as a reference.
    - Add the following lines into *build.sbt*

```
resolvers += "public-jboss" at "http://repository.jboss.org/nexus/content/groups/public-jboss/"
libraryDependencies ++= Seq(
  "org.drools" % "drools-core" % "7.28.0.Final",
  "org.drools" % "drools-compiler" % "7.28.0.Final"
)
```

    - Create directory *conf/META-INF,* put *kmodule.xml* into it. Refer to the play-drools repository for example kmodule.xml and Chapter 2.2 in documentation for explanation.
    - Create the package directory under *conf/,* the directory name is the package name specified in *kmodule.xml* (“drools” in the example in play-drools).
    - Put your *.drl* files into the package directory.
    - When initializing your KieContainer, make sure to use the following piece of code (from play-drools) :

```
import play.Environment;
...
KieContainer kc = kieServices.getKieClasspathContainer(environment.classLoader());
```

    - Play framework now uses CompletableFuture, so the following lines will have to be replaced in play-drools/app/plugins/Drools.java to make that part of the code work:

```
~~import play.libs.F;~~
import java.util.concurrent.CompletableFuture;
```

```
...
    lifecycle.addStopHook(() -> {
      kieSession.destroy();
        return F.Promise.pure(null);
        return CompletableFuture.completedFuture("Done");
    });
```

Transaction risk assessment

A payment processor (e.g., PayPal) faces several monetary and legal risks. A transaction may be fraudulent, or otherwise illegal, opening them up to legal action. A merchant might not ship a product after accepting money, leading the consumer to demand their money back. And so on. For this reason, risk assessment and management is a big part of payments business.

Risk assessment can often be very complex, with many statistical variables as well as hard laws that must be complied with. For this reason, it is often implemented as a combination of a machine learning and rule-based system. In this project, we will implement a toy example in which we will weed out transactions that are too risky before adding them to our ontology. We will use rule-based programming in Drools to do that.

You will need some kind of representation of the following two entities in your system:

Banks

A bank:

- Can process multiple transactions
- Is either local or international
- Can be blacklisted, in which case it can no longer process transactions
- You will probably need to maintain other attributes too

Transaction request

This is a representation of a transaction before we have decided if it's too risky to add to our ontology.
A transaction request:

- Is associated with a single bank
- Has a specific amount
- Has a sender and a receiver, identified by their IDs as added to the OWL ontology
- Has a single category (one of): medical, dining, gambling, wages, weapons, other.
- Has a timestamp (generated by your system)

Note that these additional attributes are just for rules processing, and don't need to be added to your ontology. You won't need to modify your OWL file for this project.

Assessment rules

We will use our knowledge base about the banks and the incoming transaction request to decide whether we should insert the transaction into our Ontology, or reject it. This should be done according to the following rules:

1. A bank, once blacklisted, can no longer process any transactions
2. A transaction request belonging to the category of “Medical”, *must* go through, regardless of what the following rules evaluate to (unless the bank is blacklisted)
3. A transaction request with category “Weapons” requires that (necessary, but not sufficient conditions):
    a. Both the sender and the receiver be trusted (note that only merchants can be trusted, but you just need to check for membership in the class Trusted in your ontology)
    b. The bank must be local
4. A transaction request with amount >$100,000 must have at least one of the participants Trusted.
5. A transaction request with amount >10 times the average amount for the given bank should be rejected (except if it’s the bank’s first transaction).
6. If <25% of a bank’s past transaction involved a trusted participant, then the bank can no longer process transactions that don’t involve a trusted participant (until the fraction goes up to 25% again).
7. If a bank suffers 3 transaction rejections in a row, blacklist the bank.

If the transaction survives all the above rules, add it to the OWL ontology.

Logging

You should maintain **two log files**, one for accepted transactions, and one for rejected ones. The acceptance log should have the following info per entry:

- Transaction ID
- Bank ID
- Sender ID
- Receiver ID
- Amount
- Category
- Timestamp

The rejection log should have the above, and in addition list the **Rule number** that it was rejected because of. The rule numbers are as in the above list of rules. One way to do this would be to use the numbers as names for your rules, and call drools.getRule().getName().

These log files can just be txt’s, and should be accessible through the API as detailed in the following section. The logs (as well as the server state) should be reset if the server is restarted.

API

Some of these APIs are from P3. Any modifications and additions are in blue.

Request	Response	Description
POST /addmerchant/:uniqueID	{ “status”: “success” }	Add an individual to Merchant class. Assume ID will be unique across *everything*, not just merchants. **Use the IDs as IRI suffixes. Example, http://your.IRI.here#merchant1**
POST /addconsumer/:uniqueID	{ “status”: “success” }	Add an individual to Consumer class. Use ID as above
POST /addbank/:nationality/:bankID	{ “status”: “success” }	Nationality will be one of “local” or “international”, in lowercase. ID will be unique.
~~POST /addtransaction/:senderID/:receiverID/:transactionID~~	~~{ “status”: “success” }~~	~~Add an individual to Transaction class. Use ID as above.~~ ~~Remember to set associated properties!~~ **This has been replaced by the /transactionrequest API.**
POST /transactionrequest/:senderID/:receiverID/:bankID/:category/:amount/:transactionRequestID	{ “status”: “success” } OR { “status”: “failure”, “reason”: “<ruleNumber>” }	Try to add a transaction. If successful, use the transactionRequestID as the transactionID. If unsuccessful, use the ID in the log anyway. A failure should specify the rule number that caused the failure. Category will be one of the ones listed before, in all lowercase. Amount will be a whole number.
GET /iscommercial/:transactionID	{ “status”: “success”,	Return whether a transaction is commercial.

	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}
		The transaction ID is what was supplied while sending the transaction request. If you used it as an IRI suffix, you shouldn’t have any problems fetching it.
		
		The additional error checking is to make sure you didn’t add a transaction that should have been rejected.

GET /ispersonal/:transactionID	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}
		Return whether a transaction is personal.

GET /ispurchase/:transactionID	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}
		Return whether a transaction is a purchase transaction.

GET /isrefund/:transactionID	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}
		Return whether a transaction is a refund transaction.

GET /istrusted/:merchantID	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a merchant”
	}
		Returns whether a merchant is trusted. If the ID doesn’t belong to a merchant, returns an error message.

GET /isblacklisted/:bankID	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a bank”
	}
		Returns whether a bank is blacklisted.

GET /bankrejections/:bankID	{
	“status”: “success”,
	“rejections”: “<number of rejections>”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a bank”
	}
		Returns the number of rejections suffered by a bank.

POST /reset	{
	“status”: “success”
	}
		This should reload the ontology (or delete all added individuals), so that I can start the testing afresh.
		
		Also delete all data in your knowledge base about banks, and any other data you were maintaining. Delete the log files too. This should reset the system to a “clean” state.

GET /rejectionlog	<rejection log contents>	Refer to the logging section.

GET /acceptancelog	<acceptance log contents>	Refer to the logging section.

Deliverables & Grading Rubric

You must submit your project as a single zip file. The grading rubric is subject to change but is given here as a guideline.

Part 0: Valid Submission (5%)

A valid submission means your submitted project executes out of box. In other words, it can be executed using 'sbt run' on a clean computer with only JDK and sbt installed. Make sure your project:

- Is submitted completely
- Has no compiling error
- Includes essential libraries, either in your code repository or configured as remote libraries in build.sbt script
- Includes no absolute path

A safe way is to test your program on another computer before you submit.

Remove target and intermediate files. For a sbt project, this can be achieved by deleting nested target directories, such as ./target, ./project/target, and ./project/project/target. Refer to: https://stackoverflow.com/questions/4483230/an-easy-way-to-get-rid-of-everything-generated-by-sbt

Part 1: Drools Rules (.drl file; 15%)

Make your .drl file easy to locate by using the package name "drools" in kmodule.xml and putting the .drl file in conf/drools/ directory.

- Contains all the rules: 10
- Rules are written so that they will be applied in the correct order: 5

Part 2: Server code (80%)

- Implements modifications to REST APIs over P3: 20
- Transactions are processed according to specified rules: 50
- Provides acceptance and rejection log over REST APIs: 10