RCB Cyclops: Rule Engine micro service

As part of the RCB Cyclops framework, the rule engine micro service is a generic business rule system that is internally utilising Java Drools, an inference engine, for its rule execution. It also exposes template instantiation, as well as truth maintenance over RESTful APIs, and allows developers to write their own business rules and provide graphical user interfaces for sales representatives to work with.

The most prominent features are as follows:

- Rule templates and their instantiation
- Rule instances and their execution
- Stateless and stateful sessions
- ORM mapping using Hibernate
- Custom and generic fact types
- Facts parsing and recognition
- Counters and helper objects
- Sync and Async messages
- Output controlled by rules
- Facts persisted via JTA
- Logging and auditing
- SDK for developers
- One time execution
- Stream processing
- Alarm notifications
- Batch processing
- Easy debugging
- State recovery

This documentation will explain how to work with rule engine micro service, how to write rules and rule templates, which RESTful APIs are available for instantiation and execution, how to extend existing and add new fact types, how to automatically and transactionally persist long living stateful facts, as well as how to define output format directly from the rule definition.

Table of Contents

1 Inference Engine

The rule engine micro service utilises Drools inference engine and its forward chaining data driven rule execution, thus system is completely reactional. Individual facts are asserted into working memory and based on pattern matching appropriate rules are being fired.

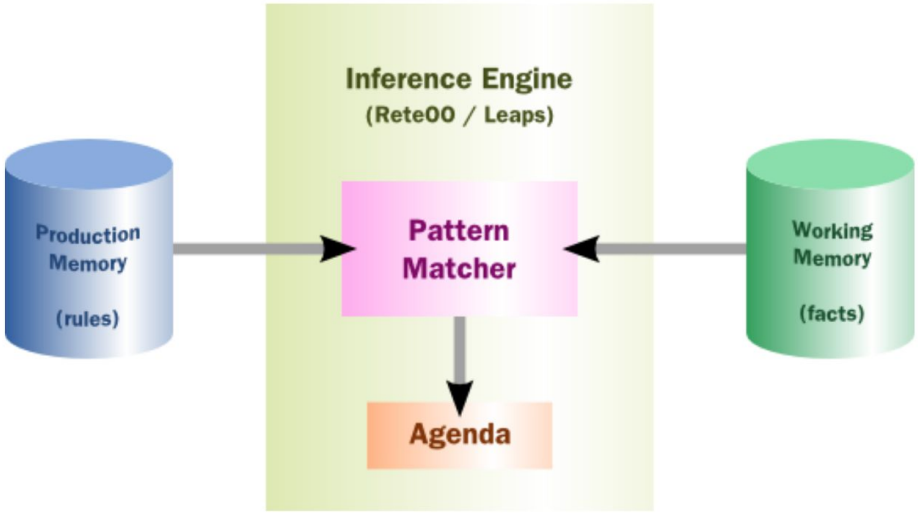

Figure 1: Drools inference engine

As can be seen in the picture above, an inference engine requires a production memory to store created rules, as well as working memory to manage facts. Java Drools was designed to work with rule repositories (files in directories with versioning), performing truth maintenance in-memory, which for a generic approach would not be sufficient enough.

Therefore the rule engine micro service utilises dynamic instantiation of rules, based either on provided templates, or rules themselves directly. They are automatically stored into backend DB, independent of technology, and can be configured by administrators during deployment.

The same applies to the truth maintenance, where rules can easily control which facts should be transactionally persisted, thus being considered as stateful objects. By executing rules and understanding fact semantics, the rule engine micro service recognizes type of the sessions and automatically performs certain optimisations for better performance and response times.

2 RESTful APIs

In order for this micro service to become truly generic and support dynamic rule definition and execution, the following features have been designed and implemented:

- *Template instantiation* - that allows for rule template definition and instantiation based on provided parameters, thus supporting dynamic environments that require automatic rule creation using predefined templates that were provided by business analysts. As well as useful for sales representatives in order to create a client specific rules using standardised interfaces and predefined templates.
- *Rule registration* - it is also possible to add and load rules independently of template instantiation, which can be useful for certain rules that are not intended to be accessed by sales representatives, or are solving specific template edge cases.
- *Fact upload* - execution of specified rules always depends on available facts and truth maintenance, which can be achieved by uploading facts into working memory and manipulating with their state over previous rule definition.
- *Response* - the generic nature of the rule engine micro service enables and allows for specification of the answer returned, including its format and content, which is always being controlled by provided rules.

2.1 Template operations

- **POST** */ruleengine/template* - add a template
- **POST** */ruleengine/template/1* - instantiate selected template
- **POST** */ruleengine/template/1?execute=true* - instantiate and execute template

- **GET** */ruleengine/templates* - list available templates
- **GET** */ruleengine/template/1* - display selected template

- **DELETE** */ruleengine/template/1* - delete selected template

2.2 Rule operations

- **POST** */ruleengine/rule* - add a rule
- **POST** */ruleengine/rule?execute=true* - add and execute rule
- **POST** */ruleengine/rule?onetime=true* - execute rule without persisting it

- **GET** */ruleengine/rules* - list available rules
- **GET** */ruleengine/rules?template=1* - list template instances
- **GET** */ruleengine/rule/1* - display selected rule instance

- **DELETE** */ruleengine/rule/1* - delete selected rule

2.3 Fact operations

- **POST** */ruleengine/facts* - load fact or facts into memory
- **POST** */ruleengine/facts?execute=false* - load but don't execute rules

If you want the rule engine micro service to use batch processing, just upload array of objects and it will be recognised automatically. In order to load facts into memory but without any rule execution, set optional *"execute"* parameter to *false*.

It is also possible to debug the state and presence of facts that are currently being persisted into database, working memory, or both. In order to execute debugging rule, without storing it permanently set optional *"onetime"* parameter to *true*.

2.4 Example workflow

All the API endpoints are described in detail and fully demonstrated using the API BluePrint notation in the *examples* folder, including input parameters and responses, as well as couple of rule and template definitions.

The workflow and functionality of the rule engine micro service can be easily showcased by using the *threshold example* (included in the mentioned folder), where steps are as follows:

1. Add a threshold template
2. Instantiate threshold template with rate 15
3. Instantiate threshold template with rate 7
4. Add rule - create counter automatically
5. Add rule - update counter on new UDR
6. Add rule - alarm when counter is above 2000 (both REST and exchange publish)
7. Add rule - reset counter above 2500 (both REST and exchange broadcast)
8. List stored templates
9. List all rule instances
10. Upload fact - UDR with usage of 400
11. Upload fact - UDR with usage of 600
12. Batch upload facts - couple of UDRs
13. Upload UDR and trigger alarm (including exchange publish)
14. Upload UDR and trigger counter reset (including exchange broadcast)
15. Batch upload facts - couple of CDRs
16. One time rule execution - list and remove all CDRs

3 Rules and Templates

This micro service supports both template instantiation, as well as direct rule definition using provided RESTful API calls referenced above. The syntax (Java or MVEL notation) of those templates and rules is well documented, with many examples and use cases available online.

The language and its execution is Turing complete, with a focus on knowledge representation to express first order logic in a concise, non-ambiguous and declarative manner. Every rule is a two-part structure using first order logic for reasoning over knowledge representation:

```
when
        <conditions>
then
        <actions>
```

Everything about rule creation can be found at docs.jboss.org, particularly describing:

- Rule syntax
- Rule language reference
- Rule templates

3.1 Template format

A template always starts with *template* keyword, following with an empty line and all available parameters, enclosed with another empty line. Then there is a section of Java library/package imports, as well as global variables (that are specified at a compilation time). After the header section, we follow with *template* keyword and its unique name. The same applies with a rule definition, where parameters in format *@{parameter}* are to be used.

The rule syntax is as referenced before, with *when-then* notation, ending using *end* keyword and the whole template is encapsulated once again with *end template*.

Even though utilised inference engine can support multiple rules in one template, or multiple templates per one request, available RESTful APIs work best when rules and templates are provided separately, always one by one. Therefore it is recommended to upload one template with one rule, or only one rule directly.

An example of threshold template (also used in the workflow mentioned before) can be found in the picture no. 2 on the next page.

```
template header

greaterThan
lessThan
rate

import ch.icclab.cyclops.facts.model.*;
global ch.icclab.cyclops.publish.Messenger messenger;

template "Threshold template"

rule "Threshold between @{greaterThan} and @{lessThan} with rate
@{rate}"
when
    $udr: UDR(usage >= @{greaterThan}, usage < @{lessThan})
then
    CDR cdr = new CDR($udr);
    cdr.setRate(@{rate});
    cdr.setCharge(cdr.getUsage() * cdr.getRate());

    // this list will be returned
    messenger.restful(cdr);
end

end template
```

Figure 2: Threshold template definition

3.2 Template instantiation

Next logical step is to instantiate a template, which can be easily achieved by providing values over JSON format, as can be seen in the picture below.

```
{
   "greaterThan":500,
   "lessThan":1000,
   "rate":7
}
```

Figure 3: Template instantiation

Based on defined template and specified parameters, the mentioned instantiation will result in threshold rule creation, that will output a CDR record for every new UDR, with the calculated usage consumption. This way it is very easy to implement GUI over template instantiation and enable sale representative to create specific rules based on customer's needs.

All templates, as well as individual rules are automatically and transactionally persisted into backend DB via Hibernate (driver and dialect can be configured in *config/configuration.txt*).

4 Fact types

Rule engine micro service provides certain features for rule and template developers, in order to assist with stateful sessions, extended fact types, as well as custom response formats. This section describes how the truth maintenance is being controlled, how fact types are registered and uploaded via mentioned API endpoints.

4.1 Automatic JSON mapping

All facts being inserted into working memory have to be mapped from JSON to POJO first, so they can be later pattern matched by the inference engine automatically. Mapping is based on mandatory *_class* field (case sensitive), as can be seen in the picture below.

```
1 {
2   "_class": "CloudStackVMUsageData",
3   "time": 15451212,
4   "account": "martin.kti",
5   "usage": 56
6 }
```

Figure 4: Simplest UDR fact type example

Uploaded facts that have *_class* field but their representation is not available in form of a local POJO class (explained in the next section), or *_class* field is omitted entirely, will not be added to the working memory, as it could lead to inconsistent state. In case that received JSON is an array, only valid facts will be added to the working memory, where others will be excluded.

4.2 Custom fact types

To register a new fact type, with the full support for automatic mapping, just add a POJO class definition into package *ch.icclab.cyclops.facts.model* and let the class inherit from *ParentFact.* The rule engine micro service will automatically (on server restart) pick up newly added object from specified directory (*src/main/java/ch/icclab/cyclops/facts/model*) and make it global for all of your rules to see. Inner packages are supported, thus the structure doesn't have to be flat.

If you want to declare new types that are referenced only in one of your rules, but never being uploaded via mentioned API endpoints, you can work with Drools' declare keyword. However, these objects can be only local (for rules where they are declared) and thus not accessible globally, as well as without any direct persistence support. On the other hand local declaration doesn't require server restart.

4.3 Hibernate persistence

In order to fully utilise stateful sessions, it is important to transactionally persist the state of the object, that is being created, updated or deleted. *ParentFact* extends *SessionFact* that offers an attribute that controls the truth maintenance and consequently object's life cycle.

Every POJO class that either extends *ParentFact* (used for automatic mapping) or inherits from *SessionFact* directly, will have a *setStateful()* setter and *isStateful()* getter. Both methods can be used directly in your rules, therefore allowing you to control what is being automatically and transactionally persisted into backend database.

Second step is to let Hibernate know about POJO class ORM mapping, simple as annotating the class with *@Entity* and the unique identifier with *@Id*, as can be seen in the picture below. All other class variables and their types are automatically detected and mapped via Hibernate ORM functionality.

```
@Entity
public class Counter extends SessionFact{
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long id;

    private String account;
    private String meter;

    private String offering;
    private String instance;

    private double value;

    public Counter() {
        // we want to preserve counters
        this.setStateful(true);
    }
```

Figure 5: Hibernate POJO annotation

If you know that a certain class is always stateful and you want it to be automatically stored in database, just create a constructor in your POJO class, that sets *stateful* variable to true. The rule engine micro service automatically detects fact manipulation states, so you don't have to do it manually. What it means is that for stateful objects that are supposed to be synchronised with backend database, you just need to use mentioned setter on object creation, and then all is handled for you in the background. If you choose so and decide to delete stateful fact from working memory, follow Drool's retract command and rule engine micro service will pick up on delete event and synchronise backend database accordingly.

To ensure a correct state recovery of a working memory, please refer to section 4.5.

4.4 Stateful counters

Once a class is registered in Hibernate configuration file, is annotated for ORM mapping, and extends either *ParentFact*, or *SessionFact*, the class is ready for automatic and transactional synchronisation with backend database. An useful example for such implementation would be a stateful counter, that any developer can use directly in rules.

An object representation for a stateful counter in this micro service is tailored for UDR records as can be seen in the picture no. 5 on the previous page. It is stateful by default, therefore any object change will be automatically synchronised with the backend database.

Pictures no. 6 and 7 represent rules for creating and updating counters, where the first rule is responsible for initialising and inserting a new counter, based on unique combination of user's meters, into working memory. The latter is a logical continuation of the former rule, resulting in counter update, also importing from *ch.icclab.cyclops.facts.model* and *helper* packages. As a rule of thumb, all automatically mapped facts should be stored in *model* package, and classes carrying supporting role, like counters, in *helper* package.

```
1  import ch.icclab.cyclops.facts.model.*;
2  import ch.icclab.cyclops.facts.helper.*;
3
4  rule "Add counter object for every new meter and user combination"
5  when
6      $udr: UDR($a: account, $m: meter, $o: offering, $i: instance)
7      not Counter(account == $a, meter == $m, offering == $o, instance == $i)
   then
8      Counter counter = new Counter($udr);
9      counter.setValue($udr.getUsage());
10     insert(counter);
11 end
12
```

Figure 6: Insert new counter

```
1  import ch.icclab.cyclops.facts.model.*;
2  import ch.icclab.cyclops.facts.helper.*;
3
4  rule "Update counter when new UDR arrives"
5      lock-on-active true
6  when
7      $udr: UDR($a: account, $m: meter, $o: offering, $i: instance)
8      $counter: Counter(account == $a, meter == $m, offering == $o, instance == $i)
   then
9      $counter.setValue($counter.getValue() + $udr.getUsage());
10     update($counter);
11 end
12
```

Figure 7: Update counter

4.5 State recovery

Because the rule engine micro service persists templates, rules and stateful facts in database, on every server restart or failure, the latest known state has to be re-loaded again, in order to ensure that the state of production and working memory is the same as it was before.

Production memory stored in the backend database and its re-loading strategy is pretty easy and straightforward, thus fully automatic, where developers do not need to configure anything. However, in order to know what fact types are considered to be long lived, one has to specify and register classes with *@Entity* annotation.

As mentioned before, one example of such use is *Counter.java* class, that is both stateful and long-lived, therefore automatically re-loaded on server re-start, along with rest of the rules and templates.

5 Custom response

The rule engine micro service supports two streams of communication, one with the originator of the RESTful requests, and second as asynchronous messaging via AMQP.

5.1 Synchronous connection

Every single time a new fact is uploaded via RESTful API calls mentioned above, developers may or may not decide to return a JSON response. It is therefore crucial to be able to specify what exactly should be returned in a synchronous call as a RESTful response, thus allowing rules to control output format and its content dynamically.

If we look back at the picture no. 2, of a threshold template example, on 7th page, we will spot a global *messenger* variable in the import section of the rule, via the following statement:

global ch.icclab.cyclops.publish.Messenger messenger;

In order to include an object in RESTful response, add it using the following method:

messenger.restful(object)

The rule engine micro service will automatically collect all the objects added to the container, in the same context and as expected, output their JSON representation as a result of the POST call. This way you can have a rule execution and fact pattern matching and still return a desired output, allowing developers to write rules and templates that are easy to understand, according to the single responsibility principle.

5.2 Asynchronous messaging

On top of RESTful responses that are usually implemented as synchronous calls, rule engine micro service also supports an asynchronous model by utilising Advanced Message Queuing Protocol implemented using RabbitMQ message broker. To cover all possible use cases your rules can use two methods for dispatching messages, both using the same global variable

global ch.icclab.cyclops.publish.Messenger messenger;

5.2.1 Publishing

Based on your specific needs you can start publishing to an exchange using *routing key*, and configure your queues, as well as consumers, however you like. The same way as with REST, all objects are automatically outputted as JSON. The method to be used is as follows:

messenger.publish(object, routing_key)

As a direct result of using routing keys and direct exchanges, it is easy to set up a topology of consumers in many different ways. If you decide to bind two queues to an exchange with the same routing key, as can be seen in the picture no. 8 on the next page, both queues will start receiving the same messages.

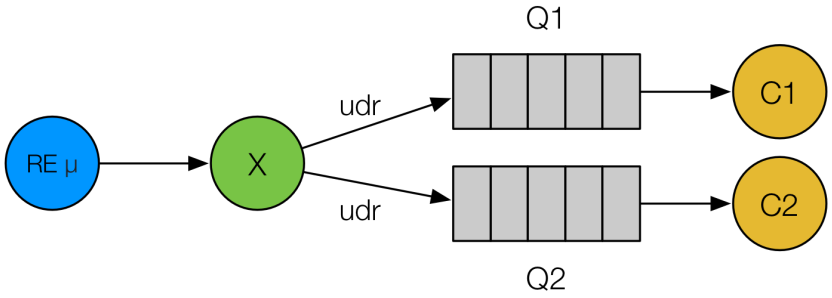

Figure 8: Binding two queues with the same routing key

This is very useful for dispatching messages to many different systems, where on the other hand a round robin dispatching can be easily achieved by subscribing couple of consumers to the same queue, as per picture no. 9.

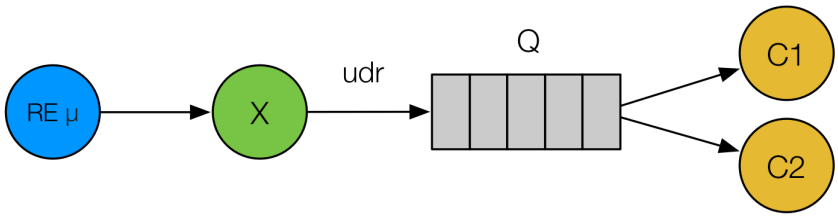

Figure 9: Subscribing two consumers to the same queue

Usually you will end up dispatching rule engine messages based on routing keys to different systems, therefore utilising a topology similar to the one in the picture no. 10.

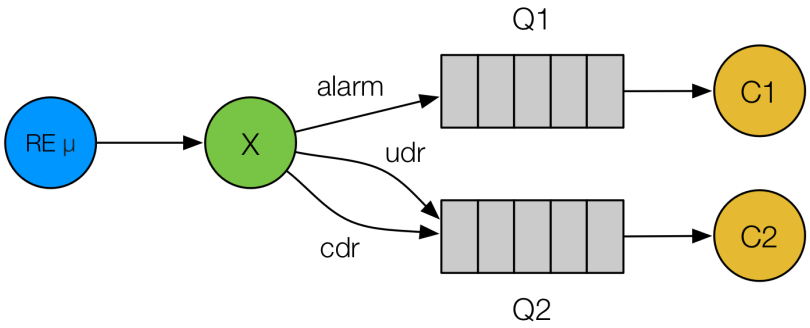

Figure 9: Two queues with three different routing keys

5.2.2 Broadcasting

In order to publish a message to all queues that are binded to a particular exchange, the rule engine micro service offers a method that doesn't require a routing key and reads as follows:

messenger.broadcast(object)

The following picture shows an example of a topology with three queues (for different systems or applications) receiving messages over broadcast, and four message consumers. Moreover, the queue labeled as Q2 will be dispatching messages in round robin as it has two consumers connected to it.

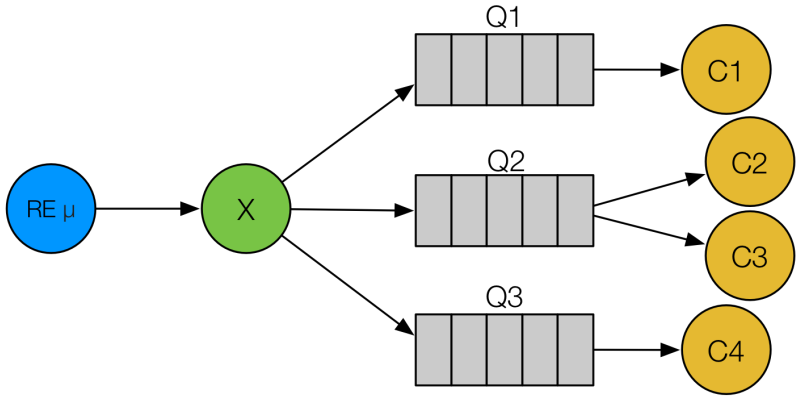

Figure 10: Broadcasting messages to three queues and four consumers

5.2.3 Configuration

The rule engine micro service requires RabbitMQ credentials properly set up and available on server start in *config/configuration.txt*. Here you can also specify what should be the names for *publish* and *broadcast* exchanges.

To understand more about RabbitMQ and its binding of queues to exchanges, read more and consult the Routing page, as well as its administrative GUI - management plugin.

5.3 Communication example

The following rule is a practical example of both communication streams - sync connection, as well as as async messaging. If you need your rules to deliver messages via AMQP simply use *publish()* and/or *broadcast()* methods. For other workflows, that rely on synchronous delivery and therefore utilise RESTful API calls, simply include your objects via *restful()* method. They will be transferred on the next fact upload.

```
1  import ch.icclab.cyclops.facts.model.UDR;
2
3  global ch.icclab.cyclops.publish.Messenger messenger;
4
5  rule "Publish every UDR on RabbitMQ"
6  when
7      $udr: UDR()
8  then
9      // publish it
10     messenger.publish($udr, "routing_udr");
11
12     // broadcast it
13     messenger.broadcast($udr);
14
15     // add it to restful container
16     messenger.restful($udr);
17 end
```

Figure 11: Rule example with RESTful response and AMQP publish/broadcast

6 Stream processing

Another way how to upload facts to the rule engine is to use AMQP via RabbitMQ, which runs alongside the RESTful API calls, therefore can be utilised at the same time. It requires proper configuration in the *config/configuration.txt* file.

It is capable of doing exactly the same as RESTful API calls for uploading facts, allowing for single/batch processing (including automatic mapping), where rules themselves don't need to change. Even though stream processing workflows may impose new set of requirements, it is still possible to combine both approaches, and get combined response on the next RESTful fact upload.

The following picture displays a rule that will detect newly added CDRs, received either thru specified queue, or uploaded over RESTful interface, and immediately the rule will publish the content to configured exchange, as well as add notification to the next RESTful response.

```
1  import ch.icclab.cyclops.facts.model.CDR;
2
3  global ch.icclab.cyclops.publish.Messenger messenger;
4
5  rule "Resend CDR to queue, retract it and add notification"
6  when
7      $cdr: CDR()
8  then
9      // resend it to queue
10     messenger.publish($cdr, "routing_cdr");
11
12     // add notification to restful container
13     messenger.restful("CDR received, resent and retracted");
14
15     retract($cdr);
16 end
17
```

Figure 12: Example of a rule that re-sends received CDRs

7 Logging and Auditing

The rule engine micro service automatically logs rule execution, truth maintenance, template and rule manipulation, as well as pattern matching and fact's life cycle. During the installation process couple of log files are created, all stored in */var/log/cyclops/ruleengine* folder.

- *timeline.log* - template and rule manipulation
- *rules.log* - pattern matching and rule execution
- *facts.log* - fact truth maintenance
- *dispatch.log* - publishing and broadcasting
- *stream.log* - queue processing facts
- *errors.log* - micro service errors
- *trace.log* - micro service debug
- *hibernate.log* - database connection

In order to understand what is happening and what operations are being currently executed in the rule engine micro service, administrators should consult *trace.log*, monitor micro service's health status via *errors.log* and debug database connection using *hibernate.log*.

Auditability can be provided by a proper tamper proof setup, where by combination of certain log files mentioned above, auditors can easily backtrack the exact steps the rule engine micro service has taken, and based on inputs provided reconstruct the flow and billable events.