Next Gen Claims API

## Rules Based Validation Framework

For the Delivery of

RCA-4315

Contact

Spring Zhou

Sr. RCA Engineer

Cotiviti, Inc.

4752667353

Spring.Zhou@cotiviti.com

2/13/20

# Overview

In current NG (Next Generation) Claims API, we are using SQL stored procedures to validate if a claim submission is valid or not. Our long-term vision is moving validation logic out of database layer and use a dedicated service layer to conduct validation logic. This document describes a proposed design for the rule-based validation framework. Along with this document, I created a POC and the source code is located in [this folder](file:///\\ccaintranet.com\dfs-dc-01\DataProcessing\Teams\YMProductSupport\Applications\NextGen\Validation%20Framework\POC).

# Design

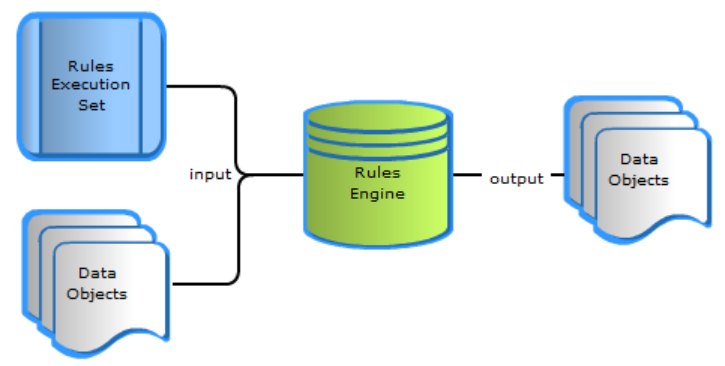
## Rule Engine Concept

A rules engine is all about providing an alternative computational model. Instead of the usual imperative model, which consists of commands in sequence with conditionals and loops, a rules engine is based on a Production Rule System. This is a set of production rules, each of which has a condition and an action - simplistically you can think of it as a bunch of if-then statements.

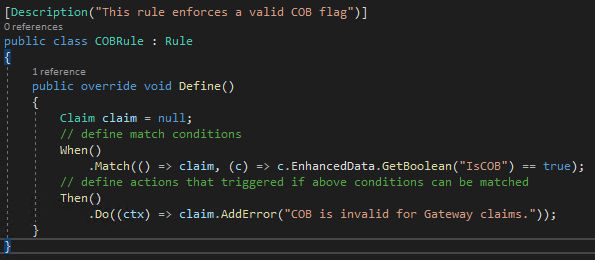
There are three common objects in a rule engine.

1. Rule Definition / Execution Set
2. Rule Engine (Runtime)
3. Data Objects (Fact Objects)

In below diagram, we describe how those objects interactive with each other.



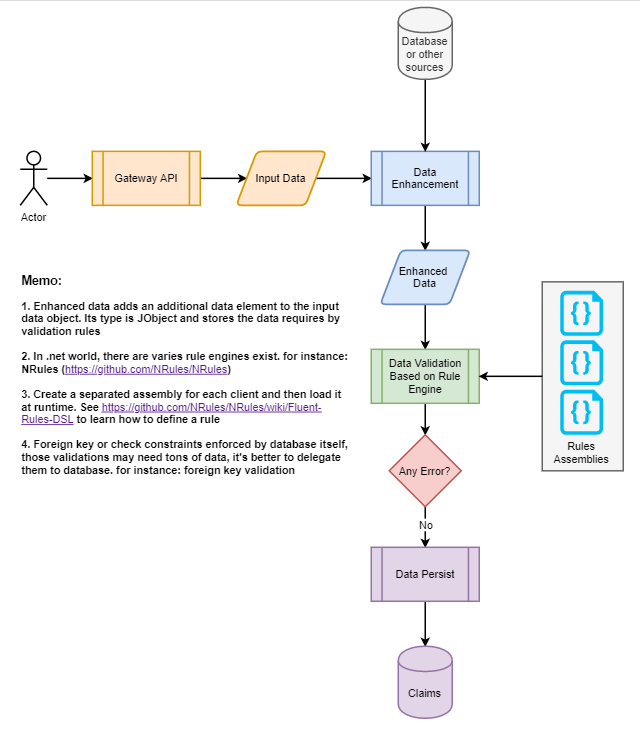
In our POC, we use [NRules](https://github.com/NRules/NRules/wiki/Getting-Started) as our rule engine. NRules use the fundamental algorithm [Rete](https://en.wikipedia.org/wiki/Rete_algorithm) which used by a bunch of rule engines today to implement pattern matching algorithm. NRules use its own DSL to define rules (conditions and actions). A simple rule can be defined as below:



## How validation works in NG Claims API

In below diagram, we describe how validation works in our NG Claims API context.

1. End users submit claims via our gateway API, and we store payloads into our internal queue system
2. Enrichment service consumes payloads and calls backends (databases, caching system, APIs and file system, etc.) to get data that used in below validation service. For instance: the audit function is used during the validation and it is not in the original payload, so we need to get this value during the enrichment process. The returned data attached to original payload and its data type is a JSON object or token. This data type can provide flexibilities to our data model because different clients can have totally different enrichment data.
3. Validation service provides below functions:
4. Initialize rule engine sessions via scanning rule definition assemblies for clients. Each client can have one or more dedicated rule definition .net assemblies, those assemblies include validation rules that specific to this client, platform or audit. Rule definition assembly follows a naming convention to allow the validation framework to map assemblies to according company, platform or audit.
5. Pass the enhanced data object (fact object) into a validation session that specific to the company, platform or audit and trigger the validation rules that defined in rule assemblies. If an error condition matched, then an error message populated into the original data object.
6. If validation failed, then store the validation errors and discontinue the pipeline
7. If validation succeed, then continue the pipeline to pass valid claims to downstream services



## Features of Validation Framework

The purpose of validation framework is to define a standard, scalable and extensible way to create, run and manage validation logic for different clients inside Claims API.

### Standard

In our validation framework, we standardize below items:

1. How to create a rule definition

Use a .net class to define a rule and this class inherits from **NRules.Fluent.Dsl.Rule** base class and implements the virtual method Define. Inside the Define method, use NRules DSL to express the conditions and actions. See the sample rule definition class in previous section. A class should only define a validation rule to make the class easy to maintain and understand.

1. How to organize rule definition assembly

Each combination of client, platform and audit can have a dedicated .net class library project, inside the project, we create classes to define validation rules that specific to the combination. In our POC, we define two .net class library projects. One is for WellPoint, other is for Gateway. See below screenshot:



1. How to name a rule definition assembly

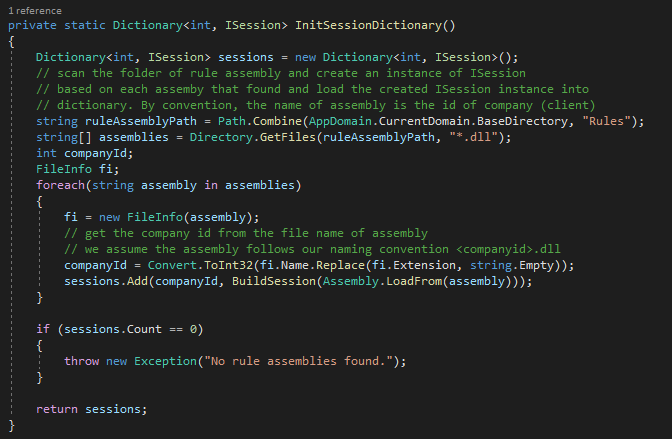
The output assembly follows the below naming convention:

<**company id**>-<**platform code:ignore:placeholder**>-<**audit id:ignore**>.dll

The company id is required, but platform code and audit id are optional. This convention provides flexibilities to organize the rule definition assemblies for a client. If a client shares validation logic across platforms and audits, then we only need to create an assembly dll <company id>.dll, but if this client has different validation rules among platforms or audits, then we can create dedicated assemblies for the combination of company id, platform code and audit id.

1. How to load a rule definition assembly

At runtime, the validation service scans rule definition assemblies in a predefined folder and initializes the NRules sessions based on the loaded assemblies. The code snippet looks like below:



1. Data model object

Create a dedicated .net class library to hold shared data model classes that used in Claims API context. A base class can be defined to include the JObject enhanced data property and methods that related to validation error collection.

### Scalability

In today’s implementation, the validation logic is stored and executed in a client specific database. As we know, the SQL server database is not easy to scale out, it means the single database is our only horsepower to execute our validation logic. When we have a light load, this approach can work for a while, but overtime this single database will be our bottleneck and single point of failure.

Our validation framework essentially is a .net based service and we can deploy this service on multiple nodes much easier (If we can use containerized deployment environment like K8S, it’s just one-line configuration change even it can be done via auto scaling feature)

### Extensibility

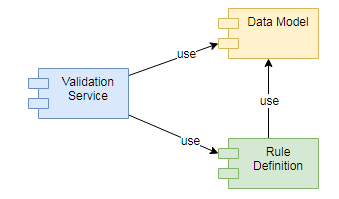
In our validation framework, if we need a new validation rule, what we need to do is creating a new rule class and we do not have a limitation of how many rules we can have.

## Components of Validation Framework

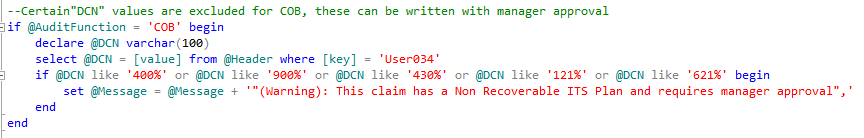
The validation framework includes below three main components:

1. Validation service (works as a host of rule engines at runtime)
2. Data model (Fact object model)
3. Rule definition assembly (.net assemblies)

The interactive diagram of above three main components looks like below:



Except those internal components, validation framework depends on the enrichment service to provide necessary data for validation. The enrichment service may still need to call a stored procedure to get data, but the main responsibility of stored procedure is to provide data that validation service needs, not the validation itself. For instance: in today’s client validation stored procedure, we use the audit function to execute multiple validations, see below code snippet:



In new client validation stored procedure, we need to include the value of audit function in the output JSON, and then we use the retrieved value of audit function to express the validation logic in a rule class.

# Appendix

## References

NRules API

<http://nrules.net/api/html/R_Project_NRules.htm>

Rete Algorithm

<https://en.wikipedia.org/wiki/Rete_algorithm>

Bogus Fake Library

<https://github.com/bchavez/Bogus>