Motor de Reglas desacoplado orientado a formato JavaScript Object Notation.

Marcos Maciel
CAETI, Buenos Aires, Argentina
Mmaciel03@hotmail.com

Resumen: Las empresas afrontan el desafío de competir en escenarios que varían de acuerdo con nuevas tendencias, clientes que buscan calidad a menor precio, o a un repentino cambio de interés desde un producto y/o servicio a otros. El acceso a dispositivos móviles con internet como medio de comunicación generan tanta curiosidad que los usuarios continuamente navegan buscando alternativas. Para satisfacer con las demandas de estos clientes tecnológicos las compañías deben cambiar sus reglas de negocios periódicamente y adaptarse inmediatamente al medio que los rodea. Con el propósito de cumplir a la demanda de cambios dinámicos y exageradamente rápido, se presenta en este trabajo un modelo orientado a crear reglas de negocios en lenguaje natural del dominio para evaluar las condiciones de verdad y modificar la información resultante. Esta propuesta está orientada a sistemas con arquitecturas orientadas a servicios que conectan aplicaciones front-end con back-end a través del formato Json (JavaScript Object Notation) como medio de intercambio de información. Las estadísticas realizadas demuestran una performance aceptable en la validación de atributos Json y una ganancia considerable entre el desarrollo de la regla en código contra la parametrización de la regla.

Palabras clave: Motor de reglas, BRMS, reglas de negocios, Json, microservicios, tecnología, lenguaje del dominio del negocio.

I. Estado del arte.

Los motores de reglas tienen una larga trayectoria, la cual comienza con la inteligencia artificial, donde fueron aplicados como método para representar conocimiento [01]. Un motor que evalúa reglas es un sistema experto especifico porque la regla representa un entendimiento del dominio del negocio. Estos sistemas inteligentes son agiles cuando gestionan la lógica de negocio de forma independiente de la aplicación [01]. Un motor de reglas apoya los cambios volátiles porque separa la regla de los procesos [02]
En el mercado existe motores de reglas comerciales y open source tales como
FICO Blaze Advisor, las reglas en Blaze Advisor son escritas en su propio lenguaje llamado SRL (Lenguaje de reglas estructurado). Previamente a crear una regla un modelo de objeto debe ser creado. Blaze es orientado al desarrollador y como producto comercial tiene un costo asociado.
Jena es un framework open source [04] fuertemente orientado dentro del marco de la Web Semántica, incluye un motor de reglas genérico, aunque incluye un ambiente
de programación para interactuar con RDF, RDFS, OWL y SPARQL [05].
Drools [06] es un sistema de administración de reglas de negocios, incluye el motor de inferencia basado en reglas de tipo inferencia hacia adelante para Java. Su lenguaje declarativo es lo suficientemente flexible para soportar cualquier dominio y está orientado a objetos, por otro lado, también es necesario contar con conocimientos de Eclipse Java®
Clips [07] es una herramienta para construir sistemas expertos de dominio público. Utiliza un lenguaje orientado a objetos llamado. Está orientado a desarrolladores, es necesario contar con conocimientos técnicos.
Oracle Policy Automation [08] son un grupo de productos para modelar e implementar reglas. Las reglas pueden ser creadas sin un modelo, en un lenguaje natural y en un editor tipo Word, esto puede causar algún problema derivado de la flexibilidad del lenguaje. La implementación de las reglas lleva una serie de pasos que requieren cierto conocimiento de herramientas externas al ecosistema del motor de regla.

II. Introducción.

En un mundo globalizado por internet las personas tienen muchas alternativas al alcance de la mano. Todos pueden acceder a teléfonos móviles en busca de opciones y de todas estas, elegir la que mas se ajusta a sus necesidades. La competencia entre organizaciones es mas variada y por ello la flexibilidad para cambiar ondemand debe ser proactiva y dinámica. Para las organizaciones que cuentan con sistemas orientados a servicios se propone un gestor de reglas desacoplado del sistema Core de negocios, orientado a Json [14], sin compilación, ni despliegue. El propósito es brindar una solución controlada por el negocio e independiente del equipo técnico. Otros aspectos para destacar de este motor de reglas son que al ser orientado a servicios no existe dependencia del lenguaje de programación con el sistema Core, No es orientado a objetos y por lo tanto no es necesario trabajar con clases o entidades del dominio, es también factible de implementar sobre un sistema ya productivo.

La ventaja de los servicios api rest es que usan un formato estándar para el intercambio de información, llamado Json cuyos atributos contienen valores no tipados. Otros servicios pueden recibir una estructura Json no tipado y procesarlo sin importar el lenguaje con el que haya sido construido. Esta propuesta implementa el uso de Json para parametrizar los distintos tipos de atributos que modelan las entidades de negocios y en conjunto forman el dominio del conocimiento de la organización. Por ejemplo, una tabla representa un nivel y un atributo de la tabla representa una entidad. Esta forma permite agrupar las entidades por niveles, aspecto importante a la hora de armar una regla. Cuando un sistema genera una representación en Json explícitamente lo modela orientado a objetos, es decir el conjunto de atributos se encuentra ajustado al modelo de dominio que los usuarios del negocio entienden, esta característica es aprovechada por el motor propuesto para evitar la dependencia de las clases, objetos y lenguaje de programación seleccionado para desarrollar el sistema Core. El modelo de datos que sustenta al motor de reglas puede ser generado desde tablas de base de datos o directamente desde los distintos tipos de response Json que el sistema genere. Si bien el estándar Json no es tipado, la propuesta incluye relacionar una entidad del negocio con un tipo de datos para mejorar la validación entre <término><hechos><término>. Desde el punto de vista técnico, la ejecución de la regla se inyecta entre el modulo front-end y back-end. En los sistemas orientados a servicios el front-end realiza una seria de pasos que llevan a los usuarios finales a componer un conjunto de datos, estos son enviados con una estructura formulada por el equipo de sistemas. Esta estructura Json es recibida por el módulo back-end quien previo a ejecutar las rutinas Core, reenvía la estructura al servicio de rule engine. Este ejecuta la/s regla/s previamente configuradas, si corresponde modifica el/los valor/es de los atributos y devuelve la estructura al back-end. Este último paso califica al motor de reglas como desacoplado, porque no es necesario incluir código intruso en el sistema Core. Existen distintas variantes para la implementación, dando lugar por ejemplo a la construcción de un servicio Facade [20] que capture las llamadas al Core, derive al motor de reglas, obtenga una respuesta y envié al sistema Core.

Este paper está organizado de la siguiente forma: descripción del proceso de implementación de cambios tradicional en la sección III, en la sección IV se desarrolla la propuesta de implementación de cambios con motor de reglas, Testing y estadísticas en la sección V, y por último se hallan conclusiones y trabajos futuros en sección VI.

III. Proceso de implementación de cambios tradicional.

Las aplicaciones Core del negocio tomadas como base para esta investigación tienen un ciclo de vida para incorporar cambios que se ajusta a una metodología de grandes compañías.

A continuación, se lista las tareas y horas involucradas en el ciclo de vida de un cambio:

- Definición del requisito por parte de usuario final del negocio.
- Análisis de la necesidad por parte del analista del negocio.
- Reunión de análisis entre el analista del negocio y del analista de sistemas.
- Reunión de transferencia del conocimiento entre el analista de sistemas y el líder de sistemas.
- Reunión de transferencia del conocimiento entre el líder de sistemas y desarrolladores front-end y back-end.
- Reunión de transferencia del conocimiento entre el líder de sistemas y tester de sistemas.

- Crear de una rama del repositorio.
- Armar ambiente local para desarrollo.
- Desarrollo de los cambios.
- Pruebas del desarrollador de sistemas en ambiente local.
- Publicación de los cambios en ambiente QA (pruebas).
- Testing del tester de sistemas
- Validación del analista de sistemas.
- Pruebas del analista de negocios y aceptación.
- Crear de una rama del repositorio.
- Planificación y armado de implementación.
- Revisión y aprobación de pasaje a producción.
- Implementación en servidores productivos.
- Merge branch-trunk en repositorio.
- Comunicación de los cambios implementados al usuario final del negocio.
- Almacenamiento y cierre del ticket.

El tiempo medio que toma implementar un cambio se ajusta al siguiente cuadro.

	Usuario Negocio	Analista del Negocio	Analista de Sistemas	Lider de Sistemas	Desarrolladores	Tester de Sistemas
Definición del requisito	10					
Análisis de requisito		5				
Reunión funcional		4	4			
Reunión tecnica			3	3		
Reunion Diseño				4	4	
Reunión testing				2	2	
Gestion repositorio codigo QA				0.5		
Armado ambiente desarrollo					2	
Desarrollo de los cambios					4	
Pruebas de sistemas					1	
Publicación de los cambios en QA					0.5	
Validacion tester					2	
Validacion funcional					1	
Gestion repositorio codigo a produccion					0.5	
Planificación y armado de implementación					0.5	
Revisión y aprobación de pasaje a producción					0.5	
Implementación produccion					1	
Merge branch-trunk en repositorio					1	
Comunicación					0.5	
Almacenamiento y cierre del ticket					0.5	
Sub-Total	10	9	7	9.5	21	0
Total	56.5					

Tabla. 1. Tabla de tareas involucradas en un cambio menor.

La Tabla. 1 describe los pasos involucrados en el proceso de cambio, desde la definición de un usuario del negocio pasando por sistemas hasta la implementación en producción.

Esta medición es para un cambio del tipo básico:

Cuando Cliente es Cliente Afinidad
Cuando Importe >= $ 1000 entonces Importe = Importe * 0.9

Es decir, se quiere *“premiar al cliente que compra asiduamente y por valor igual o superior a un valor x con el 10% de descuento”*.

En código de sistemas se puede representar de la siguiente forma

```
Function bool IsVIP (object item)
{
      var client = repository.getClient(item);
      return client.vip;
}
Function decimal GetDiscount (object item)
{
      return repository.getDiscount(item);
}
Function decimal GetConstant ()
{
      return repository.getConstant ();
}

Function main(objeto item)
{
      If(IsVip(item))
      {
if(item.importe >= GetConstant)
      {
      Var discount = GetDiscount (item);
            Item.importe = Item.import * discount;
      }
}
      return item;
}
```

La carencia de flexibilidad para incorporar cambios de forma rápida obliga a involucrar tiempo de distintos roles, quienes aportan valor al proceso, pero también agregan horas de trabajo.

En el código de ejemplo se puede cuantificar 4 métodos, al menos 2 parámetros para % de descuento y un valor para comparar cuando se supera un límite, además de una rutina principal. Es probablemente necesaria la creación de tablas o atributos sobre tablas existente para almacenar estos valores que el desarrollador necesita codificar.

En resumen, el ciclo tradicional de implementación de cambios necesita de varios actores y pasos para cumplir los requerimientos del negocio. El tiempo dedicado puede variar en función de una a otra metodología, pero el resultado global es un valor horario que esta propuesta propone acotar. Para lograrlo es necesario separar el diseño y gestión de las reglas de los procesos de sistemas [15]. Esta división de responsabilidades evita depender exclusivamente del profesional informático y la carga se reparte con los analistas del negocio.

IV. Proceso de implementación de cambios con Motor de Reglas.

El modelo propuesto en este documento utiliza como tecnologías de construcción Node.js® [13], ReactJs© [09], Redux© [10], npm© [11], PostgreSQL© [12] y una arquitectura orientada a microservicios. El prototipo se encuentra codificado en Node.js® para el backend y ReactJs©, Redux© para el front-end. Existen otros frameworks de desarrollo que permiten construir el módulo, pero se decide continuar una línea de desarrollo adoptando las nuevas tendencias del mercado.

El motor se encuentra completamente apoyado sobre Json como estructura de intercambio de datos, entre sus características se pueden listar: no usa tag de fin, es más legible y liviano (ver tabla 2).

{"employees":[{ "firstName":"John", "lastName":"Doe" }, { "firstName":"Anna", "lastName":"Smith" }, { "firstName":"Peter", "lastName":"Jones" }]}	<employees> <employee> <firstName>John</firstName> <lastName>Doe</lastName> </employee> <employee> <firstName>Anna</firstName> <lastName>Smith</lastName> </employee> <employee> <firstName>Peter</firstName> <lastName>Jones</lastName> </employee> </employees>
Tamaño: 157 bytes (157 bytes)	Tamaño: 291 bytes (291 bytes)

Tabla. 2. Tabla de comparación entre formatos Json y xml.

IV.A. Definición del modelo de datos del motor de reglas.

El modelo de datos propuesto esta implementado en 7 tablas de base de datos. En la Tabla. 3 se explican 2 tablas de sistemas: function y type las cuales contiene:

Tabla type	numeric date string float boolean calculo Json	
Tabla function	numeric	= > <= >= <> asignar
	date	= > <= >= <> asignar
	string	contiene - igual - comienza con - termina con - asignar
	boolean	= asignar
	Json	asignar
	calculo	+= -= *= asignar

Tabla. 3. Tabla de tipos de datos y funciones.

Las tablas de sistemas tienen como finalidad definir de acuerdo al tipo de datos que función se puede ejecutar sobre una entidad de negocio. Por ejemplo, de una entidad Código de Cliente se le asigna el tipo numeric y la funciones a comparar están determinada por comparaciones numéricas.

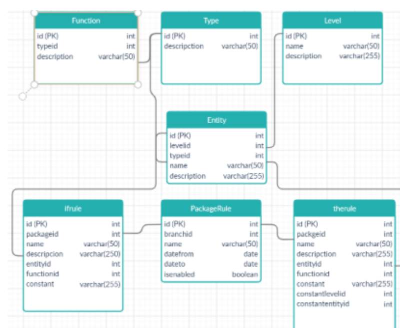

Fig. 1. Diagrama de entidad relación usado por el motor de reglas.

El resto de las tablas son PackageRule, Level, Entity, IfRule, y ThenRule. La tabla PackageRule tiene por objetivo agrupar reglas ifrule y thenrule. Level agrupa entidades y tabla Entity agrupa columnas de las entidades de negocios. Una regla esta conformada por al menos un registro en la tabla IfRule y un registro en la

tabla ThenRule. Este modelo de datos denominado de negocios esta orientado a usuarios que conocen el dominio.

IV.B. Definición de datos.

Los datos necesarios para parametrizar reglas se encuentran relacionado con las tablas de base de datos del negocio, es así como las tablas (ej. Clientes, productos, etc.) son registradas como niveles en la tabla Level y los atributos (ej. Nombre cliente, nombre producto) de cada tabla son registrados como entidades en la tabla Entity. Un registro de una tabla contiene un objeto del negocio representado en los sistemas Core como clases. Este motor se abstrae de las clases y el lenguaje de programación para usar las tablas y columnas como método para representar objetos. Estos pueden ser tablas y/o vistas de base de datos.

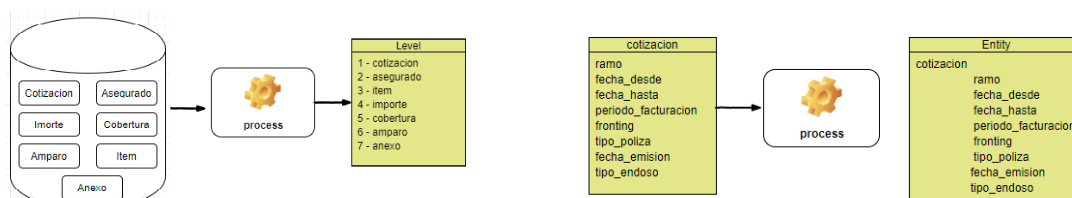

Fig. 2. Proceso de transformación de modelo de negocio en tabla del motor de reglas.

	id [PK] integer	level character varying (255)	level_description character varying (255)
1	1	cotizacion	header
2	2	item	Item o Riesgo
3	3	asegurado	Tomador
4	4	cobertura	Cobertura
5	5	amparo	Amparo
6	6	importe	Importes
7	7	anexo	Anexos

Fig. 3. Datos de tabla Level.

	id [PK] integer	levelid integer	typeid integer	name character varying (255)	description character varying (255)
1	2	1	1	ramo	Ramo
2	3	1	1	periodo_refacturacion	Periodo de Refacturacion
3	4	1	2	fec_desde	Fecha Vigente Desde
4	5	1	2	fec_hasta	Fecha Vigente Hasta
5	6	1	1	fronting	Número de Poliza
6	7	1	1	tipo_poliza	Tipo de Poliza

Fig. 4. Datos de tabla Entity.

En la Fig. 2 se describe el proceso general para entender como el motor de reglas relaciona los datos del dominio. En principio las tablas candidatas de la base de datos (las tablas de parámetros en general son excluidas) son enumeradas y su nombre es usado para llenan la tabla Level (ver Fig. 3). En este caso se toma de ejemplo un modelo de cotización de autos de la industria de seguros, los datos necesarios son un encabezado o cotización, datos del asegurado, vehículo, coberturas y amparos contratados, clausulado etc. En segundo lugar, por cada tabla candidata se listan las columnas y su tipo de datos que en conjunto llenan la tabla Entity (ver Fig. 4). Este par Nivel y Entidad da sentido a la programación de la regla que esta orientada al lenguaje natural y permite ordenar las entidades según el nivel que corresponda al dominio. Para un usuario del negocio es simple de entender una sentencia del tipo Ramo de la Cotización o Cotizacion.Ramo ya que es parte diario de su lenguaje de comunicación.
Cada registro de la tabla Entity tiene asociado a un tipo de datos (Fig. 4). Cuando el usuario del negocio está creando una regla y elige una entidad, su tipo de datos filtra las funciones que se pueden aplicar. Por ejemplo, una entidad numérica solo puede ser comparada por es mayo, es menor, es igual, es distinta pero no puede ser comparada por una función alfabética, como se muestra en Tabla. 4.

Cotizacion.Ramo es Autos Cotizacion.Ramo = 4	Cotizacion.Ramo es Autos Cotizacion.Ramo comienza con 4
correcto	incorrecto

Tabla. 4. Tabla de comparación de funciones.

IV.C. Servicio de Ejecución de Regla.

La regla se ejecuta en un servicio api rest [16], debido a la elección de los equipos de sistemas por sobre SOAP, RPC etc, [17][18] que tiene las siguientes tareas:

a) Recibe una estructura en formato Json desde un cliente.
b) Recupera los paquetes de reglas almacenados en la base de datos
c) Evalúa regla a regla
d) Si corresponde modifica la misma estructura recibida en punto a.

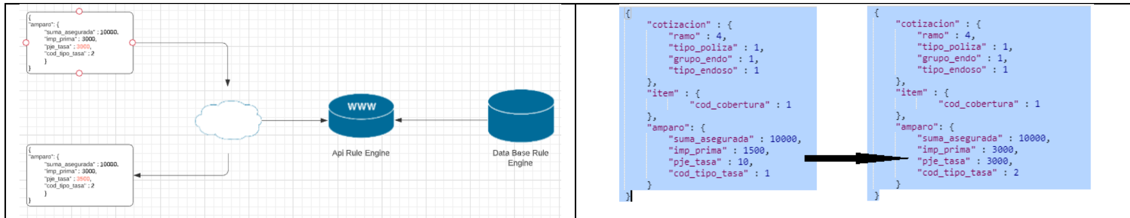

Fig. 5. Servicio Api rest Motor de regla

La estructura recibida es almacenada en memoria y se ejecuta la validación la sintaxis de Json [19], si esto es correcto continua con la recuperación de los paquetes de reglas. Cada paquete contiene un conjunto de reglas ifrule y thenrule o condición y acción. El paso siguiente es validar regla a regla, este proceso se ejecuta encadenado hacia adelante [20], el grupo de condiciones -como un todo- debe ser verdadero para desencadena una o varias acciones, estos pasos están graficados en Fig. 5. La descripción de proceso de validación de regla en seudocódigo se encuentra resumida en Fig. 6.

```
Procedimiento main(json)
    si formato de json es invalido se termina la ejecucion
    j <-- json
    k <-- buscar paquetes de regla en la base de datos
    mientras(k)
        m <-- asigna las reglas ifrule(condicion)
        mientras(m)
            se obtiene el valor del atributo de m
            se busca el atributo con j[m.level] y j[m.entity] (json de entrada)
            si existe el atributo en j entonces
                se obtiene el tipo de datos de m
                se obtiene la funcion de comparacion de m
                si valida si regla es verdadero (j[level].j[entity].value funcion j[m.level].j[m.entity].value)
                 continuar
        n <-- asigna las reglas thenrule(accion)
        if(ifrule es verdadero)
            mientras(n)
                asignar al atributo json j[m.level].j[m.entity] = funcion n.constante
```

Fig. 6. Servicio Api rest Motor de regla

El método de comparación usado es primero buscar la regla parametrizada en base de datos, entonces se busca en el cuerpo de la estructura Json recibida por coincidencia, es decir por pares atributo padre y atributo hijo, si se encuentra este par se toma el valor contenido en el atributo. El tipo de datos y su función asociada es recuperada del par atributo hijo almacenado en base de datos (en la regla) entonces se aplica la función al valor recibido contra el valor almacenado en la regla. Si el valor recibido en Json no se corresponde con el tipo de datos asociado en el atributo hijo de la regla, entonces la validación se toma como no verdadera, dejando al paquete de regla como falso y no se aplican cambios.

IV.D. Front-end para gestionar las reglas.

La interfaz para componer o crear regla se encuentra implementada en forma simple y clara, el concepto principal es llamado paquete de reglas, este contiene un conjunto de n reglas agrupadas en dos grupos independientes en cuanto a su gestión, pero dependientes en relación con los aspectos funcionales del dominio de la regla.

El módulo tiene por objetivo crear una o varias reglas del tipo condición que se evaluara en un futuro proceso, y que puede ser verdadera o falsa. Cuando se desarrolla un paquete de reglas, este tiene una o varias condiciones del tipo **Si** *esto es verdadero* **Entonces** *cambiar el valor* de una variable del dominio. Este proceso encadena una lista de condiciones, si todas ellas se cumplen entonces se ejecuta su proceso dependiente cuya finalidad es modificar valores existentes incluso en la misma condición previamente evaluada. Una regla de negocio se define de la siguiente forma:

ISBN 978-987-4417-90-9

“Si *<condición>* **entonces** *<acción>*” o **“if** *<condition>* **then** *<action>***”**

Regla A			**Regla B**	
Condición	**Acción**		**Condición**	**Acción**
1=1	A asignar B		1=2	
2=2			2=3	
3=3				
4=4				

Tabla. 5. Proceso de validación y modificación de datos.

Como se describe en Tabla 5, todas las condiciones deben ser verdaderas para desencadenar una acción, ej. Regla A. Si una de las condiciones es falsa no se desencadena acción, ej. Regla B

A continuación, se presenta la interfaz de usuario propuesta para gestionar las reglas.

+	Name	Description	Enabled	Date From	Date To
	Nuevo Negocio	Nuevo Negocio Plan Normal	✓	01/01/2020	31/12/2020

« ‹ Page 1 of 1 › » 10 Display 1-10 of 10

Fig. 7. Interfaz de usuario para administrar paquetes de reglas.

La Fig. 7 contiene la lista de paquetes de reglas gestionadas por la interfaz con nombre y descripción, estos dos campos combinados deben describir en forma concreta la validación de negocio completa. Un caso concreto puede ser Paquete: Nueva póliza de Autos, donde las reglas contenidas validan todos los aspectos para vender una póliza del ramo automotores.

+	Name	Description	Enabled
	Grupo de Cobertura	Si Grupo de Cobertura es RC	✓
	Prima Minima	Si Prima Minima es menor a $ 1500	✓
	Ramo Autos	Si esl ramo es Autos	✓
	Plan Normal	Si el plan es Normal	✓
	Grupo de Endoso	Si Grupo de Endoso es Nuevo Negocio	✓
	Tipo de Endoso	Si Tipo de Endoso es Nuevo Negocio	✓

« ‹ Page 1 of 10 › » 10 Display 1-10 of 6

Name: Grupo de Cobertura — Description: Si Grupo de Cobertura es RC — Enabled
Level: Cobertura — Entity: Código de Cobertura — Function: = — Constant: 1
Validation
Acept Close

```
{ cobertura :
    {
        cod_cobert : 1
    }
}
```

Fig. 8. Interfaz de usuario para administrar paquetes de reglas if o condición.

+	Name	Description	Enabled
	Tipo de Tasa	Entonces asignar Tipo de Tasa FIJO	✓
	Tasa de la Cobertura	Entonces asignar Prima Mínima	✓
	Prima Minima	Entonces asignar $ 1500 a Prima Minima	✓

« ‹ Page 1 of 10 › » 10 Display 1-10 of 3

Name: Tipo de Tasa — Description: Entonces asignar Tipo de Tasa FIJO — Enabled
Level: Amparo — Entity: Tipo de Tasa — Function: asignar — Constant: 2
Or
Select a Level — Select an entity
Acept Close

```
{ amparo :
    {
        tipo_tasa : 2
    }
}
```

Fig. 9. Interfaz de usuario para administrar paquetes de reglas then o acción.

Las Fig. 8 y 9 muestran la interfaz del usuario del negocio en el lenguaje que este comprende, lado izquierdo de las imágenes y superior derecho. En este caso la regla valida que una venta para un automotor cuando el usuario selecciona la cobertura obligatoria por ley: Responsabilidad civil, tenga un valor mínimo. En caso de validación verdadera (un valor x es menor a $1500) se ejecuta la acción de asignar el $1500, estas validaciones son estándar en la industria y el valor continuamente es modificado de acuerdo con la inflación país. El almacenamiento de esta regla en base de datos respeta el lenguaje del área informática y que

internamente el código entiende como estructura Json, este formato está expuesto en las imágenes inferiores derecha de las mismas figuras.

IV.D. Implementación con Json

El modelado de datos que soporta el motor de reglas descripto en primer lugar es expuesto en la interfaz de usuario que permite administrar los paquetes de reglas. El servicio como modulo independiente ejecuta en orden los paquetes habilitados cuando es invocado por un cliente externo.
Este paso es llamado desacoplado: el servicio de ejecución de reglas puede existir sin clientes que lo consuman, sin clases previamente programadas, sin importar el lenguaje de programación y/o modelo de base de datos de los sistemas Core clientes que consuman el servicio.

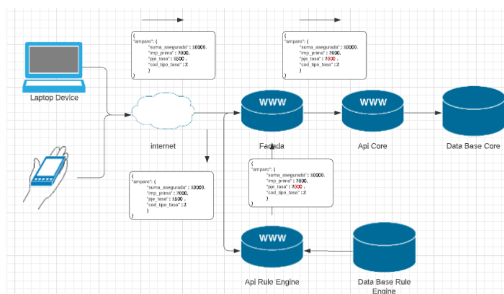

Fig. 10. Proceso de transformación de modelo de negocio en tabla del motor de reglas.

Un cliente en su dispositivo ya sea desktop, laptop o móvil hace un request o pedido a una página html o aplicación móvil. Este request puede ser un formulario que deba completar para solicitar una cotización online. Una vez completa el formulario el cliente hace un postback o envió de datos a su servicio de dominio con la estructura definida por el desarrollador informático. En este punto de la arquitectura, se propone implementar una capa de servicios por delante de Api Core que actúe como un Facade [21] que intercepte el mensaje, lo derive al api de motor de reglas. El servicio de motor de reglas ejecutara la validación sobre el/los paquete/s de regla/s parametrizado/s por usuarios del negocio. El paso posterior a la validación puede resultar en la modificación o no de la estructura json y como ultima tarea se retorna la respuesta, como se ejemplifica en la Fig. 10.

V. Testing

Se realizan pruebas de performance para determinar si el peso en el tamaño de un Json estándar puede ser un limitante para la ejecución de reglas. Las estructuras dependen de cada sistema cliente y también los valores contenidos por los atributos, para el caso de estudio el módulo del sistema tomado de ejemplo tiene un peso total máximo de 11256kb. El peso mínimo del mismo modulo es de 254kb, este valor es tomado como limite inferior para las pruebas. El test se ejecutó sobre un ambiente controlado en un pc desktop de desarrollo con las siguientes especificaciones:
Procesador Intel® Core™ i5-8350U CPU @ 1.70GHz 1.90 GHZ 8.00 GB procesador 64bits.

Peso kb	iteraciones 50	iteraciones 100	iteraciones 150	iteraciones 200	iteraciones 300	Iteraciones 400
245kb	11ms	10ms	10ms	10ms	9ms	10ms
398kb	11ms	10ms	9ms	10ms	9ms	9ms
736kb	11ms	10ms	10ms	10ms	10ms	9ms
984kb	10ms	13ms	10ms	10ms	10ms	9ms
1180kb	11ms	11ms	12ms	10ms	10ms	10ms
1716kb	11ms	11ms	10ms	11ms	11ms	10ms
11265kb	14ms	13ms	14ms	11ms	11ms	11ms

Tabla. 6. Test de performance sobre ejecución de reglas.

En la tabla 6 se detalla el resultado de las pruebas, se realizaron 6 ciclos de pruebas, en cada ciclo se ejecutaron distintas iteraciones secuenciales y con un aumento en la cantidad de ejecuciones que van desde las

50 iteraciones hasta 400 iteraciones. Las aplicaciones de negocios son complejas y la estructura Json enviada por un front-end a un servicio back-end puede variar dependiendo de esta complejidad, por este motivo se toma el peso de la estructura Json (atributos y valores) como un factor importante para entender cómo puede degradar la performance. El rango de los distintos pesos de las estructuras Json que pueden llegar al motor de reglas va desde 245kb hasta 11256kb para este modelo de aplicación. Luego de las pruebas se puede concluir que el peso no es un factor de degradación para el motor de reglas propuesto, el tiempo de ejecución medio de un paquete de reglas se encuentra entre los 9ms y los 14ms.

VI. Conclusión y trabajos futuros

En este primer trabajo sobre motor de reglas desacoplado orientado a servicios con formato de intercambio de información Json, se presenta un modelo independiente del lenguaje de programación de clientes que lo implementa, no es necesario embeber el motor en un programa o modulo existente, no orientado a objetos, de fácil parametrización e instalación. El principal aporte de este modelo es un motor de reglas simple, liviano e independiente que tiene como objetivos maximizar los cambios dinámicos del negocio, acercar a usuarios del negocio una idea que los ayude a autogestionarse y disminuir el esfuerzo-tiempo de programación y mantenimiento de los sistemas Core. La evolución de este trabajo está orientado a usuarios del negocio, que puedan gestionar reglas de forma sencilla y práctica. Desde el punto de vista de la complejidad, mejorar el desarrollo de reglas agregando rangos numéricos para contemplar por ejemplo en una condición valor entre x e y, misma característica para las fechas. Para los casos de valores constantes, la alternativa de invocar a un servicio que retorne este valor. Agregar el versionado de reglas para registrar la evolución en el tiempo de los cambios en el negocio.

Referencias

[01] Bajec, Marko & Krisper, Marjan. (2005). Issues and Challenges in Business Rule-Based Information Systems Development.. 887-898.
[02] Business Rule Group. http://www.businessrulesgroup.org/theBRG.htm [Acceso 01/02/2020].
[03] FICO® Blaze Advisor® https://www.fico.com/es/products/fico-blaze-advisor-decision-rules-management-system [Acceso 06/04/2020].
[04] Jena Apache para Java. http://jena.sourceforge.net/ [Acceso 05/02/2020].
[05] SPARQL Query Language for RDF https://www.w3.org/TR/rdf-sparql-query/ [Acceso 05/02/2020].
[06] Drools. https://www.drools.org/ [Acceso 24/02/2020].
[07] Clips C Language Integrated Production System http://www.clipsrules.net/ [Acceso 20/02/2020].
[08] Oracle Policy Automation. https://www.oracle.com/applications/customer-experience/service/intelligent-advisor/policy-automation.html [Acceso 03/01/2020].
[09] React https://reactjs.org/ [Acceso 10/10/2019].
[10] Redux https://redux.js.org/ [Acceso 15/10/2019].
[11] NPM https://www.npmjs.com/ [Acceso 11/10/2019].
[12] Postgresql https://www.postgresql.org/ [Acceso 04/11/2019].
[13] Nodejs https://nodejs.org/es/ [Acceso 10/10/2019].
[14] Json Javascript Object Notation https://www.w3schools.com/Js/js_Json_intro.asp [Acceso 28/06/2019].
[15] Ross, R. G. The Business Rule Approach. Computer 36 (2003), 85–87.
[16] Servicio Api Rest https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design [Acceso 08/07/2019].
[17] A. Neumann, N. Laranjeiro and J. Bernardino, "An Analysis of Public REST Web Service APIs," in IEEE Transactions on Services Computing, doi: 10.1109/TSC.2018.2847344.
[18] Rodríguez Carlos et al., "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices", International Conference on Web Engineering, 2016.
[19] Json Syntax https://www.w3schools.com/js/js_json_syntax.asp [Acceso 28/06/2019].
[20]Giarratano, J., & Riley, G. (1998). Expert systems: Principles and programming (3rd ed.). Brooks/Cole Publishing Co. Pacific Grove, CA, USA.
[21] Gamma, E., Helm, R., Johnson, R.,, Vlissides, J. M. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. ISBN: 0201633612