Generación automática de servicios backend y sus bases de datos utilizando MDE

Sebastian Schwartz[1], Daniel Riesco[1], Corina Abdelahad[1]

[1]Departamento de Informática – Universidad Nacional de San Luis
San Luis, Argentina

seba.schwartz.7@gmail.com, {cabdelah, driesco}@unsl.edu.ar

Abstract. *Since REST (Representational State Transfer) has originated in 2000 in a doctoral thesis written by Roy Fielding, the number of projects that use this style of software architecture for distributed systems has been increasing. Many developers (mobile, web, etc.) use REST backend services interacting with Json documents from their applications, but not all have time or knowledge to implement them. This paper shows that is possible generate automatically the formal representation of REST backend service and the database structure from a set of Json documents.*

Resumen. *Desde que se originó REST (Representational State Transfer) en el año 2000 en una tesis doctoral sobre la web escrita por Roy Fielding hasta la actualidad, ha ido en aumento la cantidad de proyectos que eligen este estilo de arquitectura de software de sistemas distribuidos. Muchos desarrolladores (mobile, web, etc.) hacen uso de servicios backend REST interactuando con documentos Json desde sus aplicaciones, pero no todos tienen tiempo o conocimiento para implementarlos. Este artículo muestra que es posible generar de forma automática la representación formal del servicio backend REST y la estructura de la base de datos partiendo de un conjunto de documentos Json.*

1. Introducción

En el año 2000 se origina REST (Representational State Transfer) en una tesis doctoral sobre la web escrita por Roy Fielding [1]. REST es un estilo de arquitectura de software para sistemas hipermedia distribuidos como la World Wide Web.

Desde ese momento a la actualidad ha ido en aumento la cantidad de proyectos que eligen este estilo de arquitectura de software de sistemas distribuidos. Cuando un servicio REST cumple con ciertas restricciones (mantiene el estilo de arquitectura Cliente-Servidor, es independiente de estados, mantiene una interfaz uniforme, etc) se lo denomina RESTful.

Json Hyper-Schema es un formalismo para describir APIs (Application Programming Interface) RESTful. Soporta la descripción de datos de entradas y salida de una interfaz, enlaces que identifican las URIs, enlaces de relaciones y los métodos que se aplican a esos enlaces [2].

Los datos pueden ser representados en cualquier tipo de formato, como ser XML, Json, etc. Para el alcance de este trabajo se utiliza el concepto de Json como formato de estructura de datos utilizado en API RESTful.

Json (JavaScript Object Notation) es un formato de intercambio de datos ligero, basado en texto e independiente de todo lenguaje de programación. Leer y escribir Json es simple para todas las personas relacionadas a la informática, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript, definido en [3]. Un documento Json define un pequeño conjunto de reglas de formato para la representación portátil de datos estructurados [4].

Json Schema define el tipo "application/schema+Json", el cual es un formato basado en Json y escrito en Json que permite definir la estructura de un documento Json. Provee un contrato para definir el documento Json requerido para una aplicación dada y cómo interactuar con ella. Json Schema define validaciones, documentación, navegación y control de interacción del documento Json [5].

Por otra parte, una transformación de modelo es el proceso que, basado en una serie de reglas, define los mecanismos para el paso de un modelo origen a un modelo destino [6]. La transformación de modelos se considera el proceso central de MDA (Arquitectura Dirigida por Modelos) [7].

El estándar QVT (Query View Transformation), propuesto por la OMG (Object Management Group), se estableció para crear consultas, vistas y transformaciones de modelos [11]. Su especificación depende de otros dos estándares de la OMG como son MOF (Meta Object Facility) 2.0 y OCL (Object Constraint Language) 2.0. Para definir transformaciones QVT necesita que los metamodelos que participan en la transformación estén en formato Ecore para dar soporte al metamodelado. Para más detalle de QVT, el lector puede visitar [11].

Un metamodelo es un modelo de un lenguaje de modelado [20] que describe el conjunto de modelos admisibles. Un metamodelo define el lenguaje con el cual se construyen modelos, como así también define formalmente los elementos de un lenguaje de modelado junto con sus relaciones y restricciones.

Existen diferentes lenguajes para definir metamodelos, por ejemplo MOF [8] es el lenguaje establecido por la OMG para definir metamodelos, y Ecore es el lenguaje utilizado por EMF (Eclipse Modeling Framework) [18]. Todos los modelos en EMF se representan con ese lenguaje. MOF es similar a Ecore en su capacidad de especificar clases con sus características de comportamiento, estructura, herencia y paquetes.

A lo largo del presente artículo se mostrará la solución planteada para cumplir con el objetivo principal de este trabajo. Es decir, permitir a las personas que hacen uso de servicios backend (desarrolladores mobile, frontend, backend, etc.), generar de forma fácil y automática la estructura de la base de datos y la representación formal del servicio backend que gestionará los datos de su proyecto.

El artículo está estructurado en secciones. La sección dos presenta los trabajos relacionados. La Arquitectura que interviene en el flujo de la solución planteada, es presentada en la sección tres. La sección cuatro muestra las características más

importantes de los metamodelos JsonSchema y RDBMS_SQL. La sección cinco expone, de manera general, la transformación JsonSchema2SqlTransformation, que usa los metamodelos presentados en la sección cuatro. La sección seis explica las reglas definidas para la transformación JsonSchema2SqlTransformation. En la sección 7 se muestra un ejemplo de la transformación JsonSchema2SqlTransformation. Finalmente, en la sección 8 se presentan las conclusiones.

2. Trabajos Relacionados

En esta sección se presentan los trabajos relacionados más importantes, estrechamente relacionados con la generación de APIs REST (con diferentes fuentes de originación) o la manipulación de documentos Json para obtener más información, metadatos o su estructura.

En [12], el autor muestra una transformación QVT entre servicios web Soap y API RESTful. En este caso, el autor eligió un enfoque conservador para construir el metamodelo de API RESTful por lo que no se utilizó un formalismo como WSLD (Web Services Description Language) 2.0 o Json Hyper-Schema.

En [13], los autores muestran cómo utilizar la información implícita contenida en los documentos Json para generar modelos que sirvan para ayudar a los programadores a disminuir el tiempo de comprensión de los documentos Json que retorna un servicio.

En [14], se plantean los mapeos para realizar una transformación desde instancias de Json, partiendo de JsonSchema simplificado, a OWL/XML usando la semántica y reglas de OWL (Web Ontology Language).

Por otro lado, en los últimos 5 años surgieron servicios BaaS (Backend as a Service) apoyados por la necesidad de reducir el costo de desarrollo y el tiempo para publicar un proyecto en el mercado (time-to-market). Los 2 principales BaaS son Parse [15] (comprado por Facebook) y Apispark [16] donde la idea principal es permitir al usuario generar Apis REST a partir de la configuración del esquema de base de datos.

En [17], se presenta una solución MDE (Model Driven Engineering) para inferir el esquema de bases de datos NoSQL a partir de documentos Json.

3. Arquitectura General

La Figura 1 muestra los metamodelos y transformaciones que fueron necesarios construir para poder generar de forma automática la estructura de la base de datos y la representación formal de las APIs RESTful, partiendo de un conjunto de documentos Json. El servicio backend será el encargado de gestionar los tipos y estructuras de datos que de manera implícita están representados en los documentos Json.

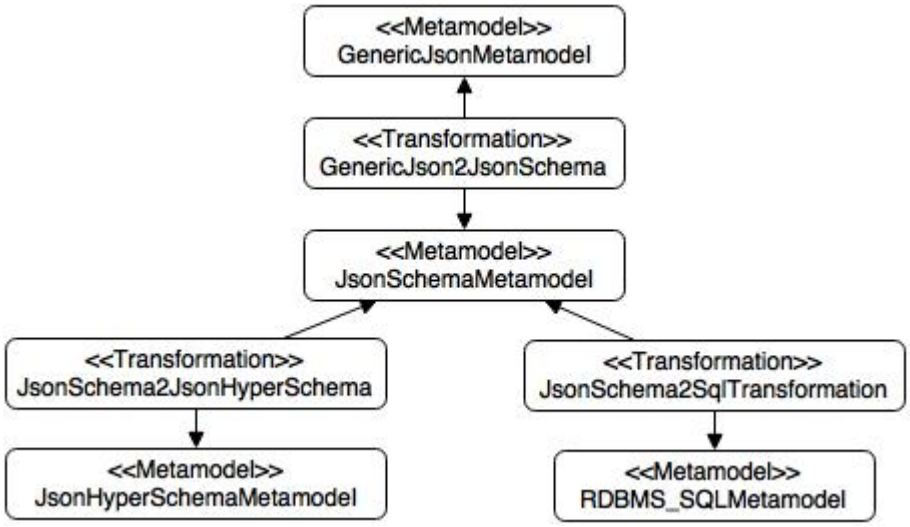

Figura 1. Arquitectura general

A continuación se mencionan las principales características de las transformaciones y metamodelos que intervienen en el proceso:

- Metamodelo GenericJson: Este metamodelo es el punto de partida de la transformación GenericJson2JsonSchema. El objetivo principal es el de agrupar y representar los documentos Json.
- Transformación GenericJson2JsonSchema: Tiene como principal objetivo generar la información más importante obtenida a partir de los documentos Json acumulados para un proyecto, representándola de manera formal con el metamodelo JsonSchema.
- Metamodelo JsonSchema: El objetivo principal de este metamodelo es representar la información más importante obtenida a partir de los documentos Json.
- Transformación JsonSchema2JsonHyperSchema: Tiene como principal objetivo generar la representación formal del servicio backend a partir de un JsonSchema. Este servicio backend es una API RESTful que gestiona los datos de la base de datos que se genera con la transformación JsonSchema2SqlTransformation.
- Metamodelo JsonHyperSchema: El objetivo principal es utilizar un formalismo para describir APIs RESTful, que represente sus datos y métodos necesarios para manipular las entidades del modelo de base de datos (Post, Get, Put y Delete).
- Transformación JsonSchema2SqlTransformation: Tiene como principal objetivo generar la estructura de la base de datos a partir de un JsonSchema.
- Metamodelo RDBMS_SQL: El objetivo principal de este metamodelo es poder representar la estructura de la base de datos.

Los metamodelos GenericJson, JsonSchema y JsonHyperSchema fueron construidos para este trabajo. El metamodelo RDBMS_SQL fue extraído desde el anexo

de [11]. Es importante tener en cuenta que estos metamodelos están expresados en MOF (Meta Object Facility).

4. Metamodelo JsonSchema y RDBMS_SQL

En esta sección se presenta las características más importantes de los metamodelos JsonSchema y RDBMS_SQL. En la siguiente sección se mostrará la transformación JsonSchema2SqlTransformation, una de las utilizadas en este trabajo, la cual hace uso de estos dos metamodelos.

4.1. Metamodelo JsonSchema

"Json Schema es un formato basado en Json y escrito en Json que permite definir y validar la estructura de un documento Json. Provee un contrato para definir el documento Json requerido para una aplicación dada y cómo interactuar con ella. Json Schema define validaciones, documentación, navegación y control de interacción del documento Json" [5].

El objetivo principal del metamodelo JsonSchema, construido para este trabajo, es el de representar la información más importante obtenida a partir de los documentos Json acumulados para un proyecto. Algunos de los datos más importantes que va a contener este metamodelo son los tipos de los atributos y sus propiedades principales (máximos, mínimos, expresiones regulares, formatos, etc). Esta información servirá como entrada al análisis para obtener la estructura de la base de datos y la representación formal del servicio backend (al aplicarse las transformaciones JsonSchema2SqlTransformation y JsonSchema2JsonHyperSchema respectivamente).

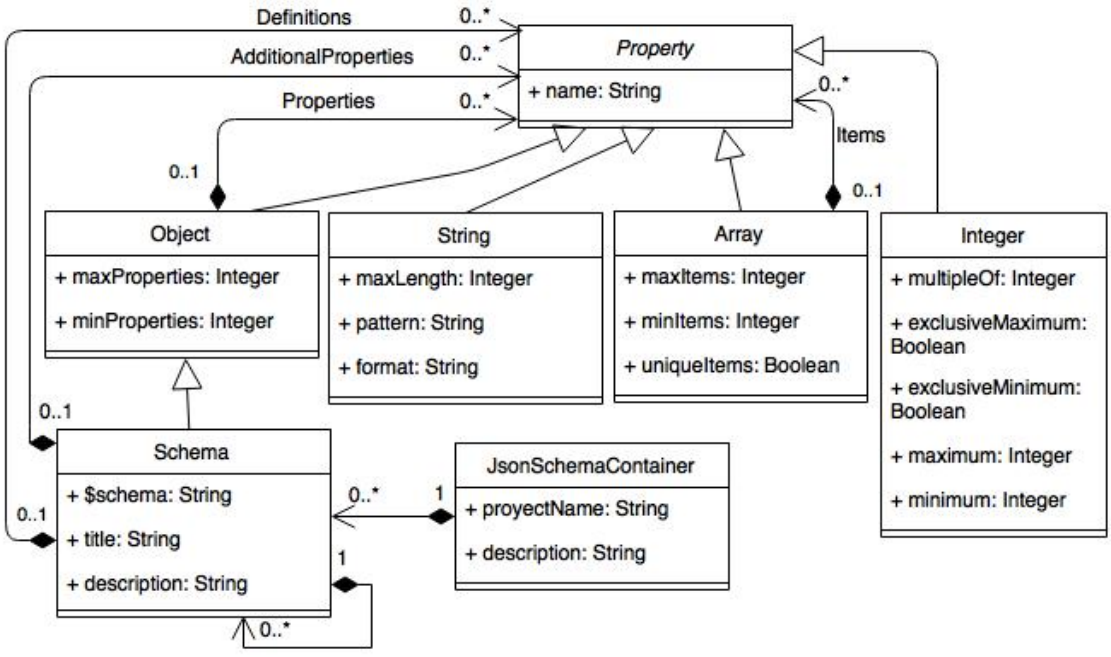

Figura 2. Diseño del Metamodelo JsonSchema

A continuación se presentan los elementos más importantes del metamodelo:

Schema: Es la estructura que define el formato de los datos. Es representado como un objeto que contiene algunas propiedades más, específicas del schema. Cada schema está contenido dentro de un "JsonSchemaContainer".

JsonSchemaContainer: Su principal responsabilidad es la de representar un proyecto, conteniendo todos los schemas y convirtiéndose en el punto de partida desde donde se desprenden los elementos que intervienen en el metamodelo.

Property: Es una clase que representa la abstracción del concepto de propiedad en JsonSchema. Contiene el atributo name para identificar a la propiedad.

Object: Esta clase es fundamental en el metamodelo, ya que un schema se define a partir de un Object. Extiende de Property y cumple el rol de contenedor de propiedades de un schema.

String, Array e Integer: Estas clases representan propiedades del tipo de dato String, Array e Integer respectivamente.

4.2. Metamodelo RDBMS_SQL

RDBMS (Relational Database Management System) hace referencia al sistema de gestión de bases de datos que está basado en el modelo relacional. SQL (Structured Query Language) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en ellas.

El objetivo principal de este metamodelo es poder representar la estructura de la base de datos obtenida a partir de los documentos Json acumulados para un proyecto. Algunos de los datos más importantes que va a contener este metamodelo son las tablas, columnas, claves primarias, claves foráneas, tipos de las columnas y las relaciones entre las tablas. Para más informacion, ver anexo de [11].

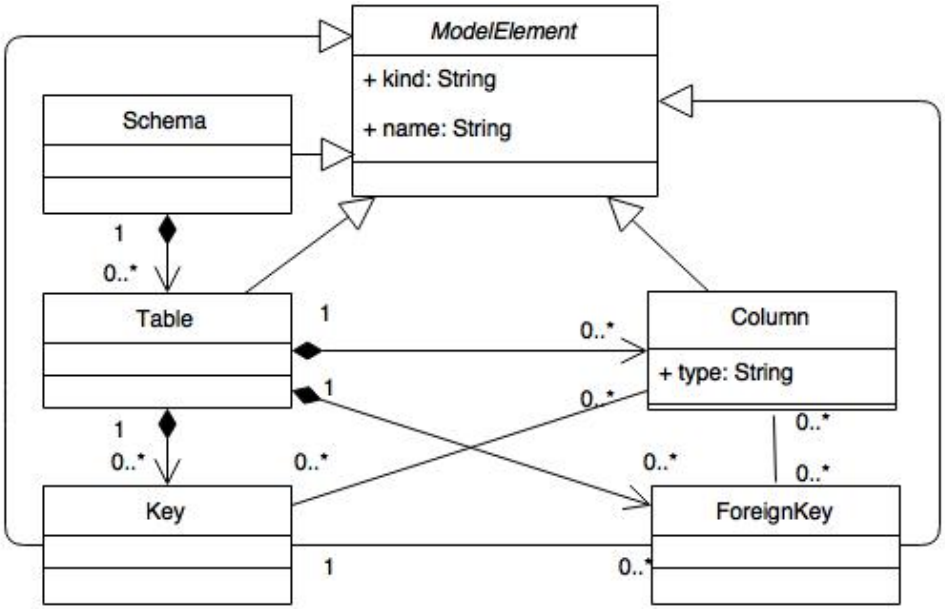

Figura 3. Diseño del Metamodelo RDBMS_SQL [11]

5. Arquitectura de la transformación JsonSchema2SqlTransformation

En esta sección se presenta la transformación JsonSchema2SqlTransformation cuya principal responsabilidad es generar la estructura de la base de datos a partir de un JsonSchema. Esta base de datos será la que luego se utilice desde la API RESTful que se genera con la transformación JsonSchema2JsonHyperSchema.

La transformación JsonSchema2SqlTransformation es aplicada al modelo origen, el cual es una instancia del metamodelo JsonSchema, para obtener el modelo destino, instancia del metamodelo RDBMS_SQL. La Figura 4 muestra la arquitectura de transformación de modelos; en ella se ilustra cómo se definen y aplican las transformaciones que están basadas en la utilización de metamodelos. En el nivel superior se pueden observar los metamodelos utilizados en la definición de la transformación, JsonSchema y RDBMS_SQL respectivamente. En el nivel intermedio se encuentran los modelos específicos a los cuales se les aplicará la transformación para obtener modelos RDBMS_SQL. El nivel inferior representa las instancias de los modelos, los cuales serán ejecutados en el correspondiente motor del flujo de trabajo.

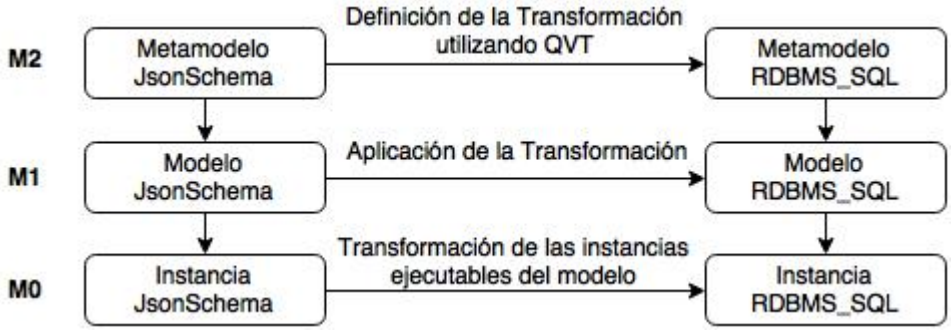

Figura 4. Arquitectura de la transformación JsonSchema2SqlTransformation

La Figura 5 ilustra la correspondencia entre los elementos de JsonSchema y los de RDBMS_SQL. La jerarquía entre los elementos es representada con la estructura de un árbol.

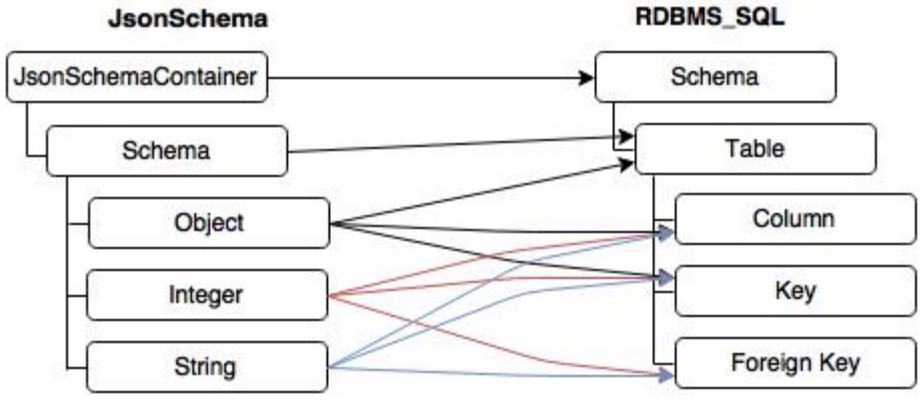

Figura 5. Mapeo de JsonSchema a RDBMS_SQL

En la parte de JsonSchema se observa que el elemento principal es JsonSchemaContainer y con él el elemento Schema el cual contiene sus propiedades (que pueden ser Integer, String y Object).

En la parte de RDBMS_SQL se observa que el elemento principal es Schema y con él el elemento Table el cual contiene columnas (Column), clave (Key) y clave foránea (Foreign key).

Aplicando la transformación, partiendo de JsonSchemaContainer, se puede obtener un schema en RDBMS_SQL que será el elemento responsable de contener las tablas de la base de datos. Por otro lado, cada Schema de JsonSchema genera una tabla en RDBMS_SQL. En la Figura 5 también se puede observar que un Object de JsonSchema genera una tabla como así también puede generar una columna y una clave que la represente. Por último, las propiedades Integer y String son transformadas a columnas y algunas de ellas pueden generar claves primarias y claves foráneas de RDBMS_SQL.

6. Reglas definidas para la transformación JsonSchema2SqlTransformation

Para ilustrar cómo se van transformando los elementos, a continuación se mostrarán las relaciones correspondientes a Schema, Table, Column y la generación de las Key y Foreign Key.

Las transformaciones comienzan definiendo el nombre de la misma junto con los modelos candidatos y sus metamodelos respectivamente. En la Figura 6 se muestra la transformación JsonSchema2SqlTransformation, la cual toma como modelo origen un modelo JsonSchema, el cual es una instancia del metamodelo JsonSchema, y produce como modelo destino un modelo SQL, el cual será una instancia del metamodelo SQL. Es importante aclarar que en el código de la transformación se renombró RDBMS_SQL (tanto en el modelo como en el metamodelo) por SQL para hacerlo más legible durante la implementación.

Tanto el metamodelo de JsonSchema como el de SQL se encuentran en formato ecore. En la Figura 6 también se presenta la relación JsonSchemaContainerToSqlSchema, la cual constituye el punto de entrada para comenzar la transformación en QVT. La finalidad de esta relación es crear el elemento Schema, el cual contiene las tablas de la base de datos, convirtiéndose en el punto de partida desde donde se desprenden los elementos que intervienen en el metamodelo SQL. La relación JsonSchemaContainerToSqlSchema también tiene como responsabilidad invocar a la relación JsonSchemaToTable encargada de completar la transformación para la creación del modelo SQL.

```
transformation jsonSchema2SqlTransformation(jsonSchema:JsonSchemaMetamodel,
                                                                    sql:SqlMetamodel) {
    top relation JsonSchemaContainerToSqlSchema {
            jsContainerName : String;
            jsContainerDescription : String;
            checkonly domain jsonSchema
            jsContainer:JsonSchemaMetamodel::JsSchemaContainer {
                    jsName = jsContainerName,
```

```
                    jsDescription = jsContainerDescription
                };
                enforce domain sql newSqlSchema : SqlMetamodel::SqlSchema{
                    sqlName = jsContainerName
                };
                where {
                    JsonSchemaToTable(jsContainer,newSqlSchema);
                }
        }
.... }
```

Figura 6. JsonSchema2SqlTransformation y JsonSchemaContainerToSqlSchema.

Una vez creado el Schema en la base de datos, la transformación se dirige hacia la cláusula where para validar que la relación JsonSchemaToTable se satisfaga. Esta relación, presentada en la Figura 7, define el mapeo entre los Schema del metamodelo JsonSchema y las tablas (Table) del metamodelo SQL, como así también se asegura de asignar una clave a la tabla generada.

```
relation JsonSchemaToTable {
        jsSchemaName : String;
        checkonly domain jsonSchema jsContainer : JsonSchemaMetamodel::JsSchemaContainer {
                jsSchema = jsSchema2 : JsonSchemaMetamodel::JsSchema {
                    jsName = jsSchemaName
                }
        };
        enforce domain sql newSqlSchema : SqlMetamodel::SqlSchema {
                sqlTable = newSqlTable : SqlMetamodel::SqlTable {
                    sqlName = jsSchemaName,
                    sqlColumn = newSqlKeyColumn : SqlMetamodel::SqlColumn {},
                    sqlKey = newSqlKey : SqlMetamodel::SqlKey {
                        sqlColumn = newSqlKeyColumn:SqlMetamodel::SqlColumn{}
                    }
                }
        };
        where {
            AsignPkByDefault(jsSchema2.oclAsType(JsonSchemaMetamodel::
                                                    JsObject),newSqlKeyColumn,newSqlKey);
            GeneratePkeyFromProperties(jsSchema2.oclAsType(
                                JsonSchemaMetamodel::JsObject), newSqlKeyColumn, newSqlKey);
             PropertyToColumnOrTable(jsSchema2.oclAsType(
                                    JsonSchemaMetamodel::JsObject), newSqlTable,newSqlSchema);
        }
}
```

Figura 7. Relación JsonSchemaToTable.

En esta relación se puede observar cómo un Schema se transforma en una Tabla. Además, contiene una cláusula where que posee la relación AsignPkByDefault y GeneratePkeyFromProperties que son las responsables de asignar una clave por default (si no existe una candidata) o generar una a partir de una propiedad candidata del JsonSchema.

La cláusula where también contiene la relación PropertyToColumnOrTable que es la encargada de gestionar las transformaciones de propiedades String, Integer y Object a columnas o tablas según corresponda, invocando las relaciones PrimitiveAttributeToColumn y ComplexPropertyToTable, como se observa en la Figura 8. En la cláusula también se invoca la relación GenerateForeignKeyExternalTable.

```
relation PropertyToColumnOrTable {
        checkonly domain jsonSchema jsObject : JsonSchemaMetamodel::JsObject {};
        checkonly domain sql sqlTable : SqlMetamodel::SqlTable {};
        checkonly domain sql newSqlSchema : SqlMetamodel::SqlSchema {};
        where {
                PrimitiveAttributeToColumn(jsObject, sqlTable);
                GenerateForeignKeyExternalTable(jsObject,sqlTable);
                ComplexPropertyToTable(jsObject,newSqlSchema, sqlTable);
        }
}
```

Figura 8: Relación PropertyToColumnOrTable .

A continuación se presentan las tres relaciones de la cláusula where: PrimitiveAttributeToColumn, ComplexPropertyToTable y GenerateForeignKey-ExternalTable.

PrimitiveAttributeToColumn : Su principal responsabilidad es la de generar nuevas columnas en una tabla (con su respectivo tipo) cuando la propiedad analizada desde JsonSchema es primitiva (es decir, es String o Integer). Esto se identifica con la query PropertyTypeIsPrimitive. Tener en cuenta que en este artículo se muestran las relaciones principales dejando afuera todas las querys. Las mismas fueron implementadas con las operaciones estándar de OCL especificadas en [9] y [10].

```
relation PrimitiveAttributeToColumn {
        jsPropertyName : String;
        sqltype : String;
        checkonly domain  jsonSchema jsObject : JsonSchemaMetamodel::JsObject {
                jsProperties = objectProperty : JsonSchemaMetamodel::JsProperty {
                        jsName = jsPropertyName
                }
        };
        enforce domain sql sqlTable : SqlMetamodel::SqlTable {
                sqlColumn = newSqlColumn : SqlMetamodel::SqlColumn {
                        sqlName = jsPropertyName,
                        sqlType = sqltype
                }
        };
        when {
                PropertyTypeIsPrimitive(objectProperty);
                not(CanCreateCustomPK(jsPropertyName));
                not(PropertyIsForeignKey(jsObject, jsPropertyName));
        }
        where {
                sqltype = PrimitiveTypeToSqlType(objectProperty);
        }
```

```
}
```

Figura 9. Relación PrimitiveAttributeToColumn.

ComplexPropertyToTable: Su principal responsabilidad es la de generar nuevas tablas de un Schema (con su respectivas claves primarias y foráneas) cuando la propiedad analizada desde JsonSchema es un Object.

```
relation ComplexPropertyToTable {
        jsObjectName : String;
        sqlKeyName : String;
        sqlKeyType : String;
        checkonly domain jsonSchema jsObject : JsonSchemaMetamodel::JsObject {
                jsProperties = objectProperty : JsonSchemaMetamodel::JsProperty {
                        jsName = jsObjectName
                }
        };
        enforce domain sql newSqlSchema : SqlMetamodel::SqlSchema{
                sqlTable = newSqlTable :SqlMetamodel::SqlTable{
                        sqlName = jsObjectName,
                        sqlColumn = newSqlKeyColumn : SqlMetamodel::SqlColumn {},
                        sqlKey = newSqlKey : SqlMetamodel::SqlKey {
                                sqlColumn=newSqlKeyColumn:SqlMetamodel::SqlColumn{}
                        }
                }
        };
        -- Generate Table Relation
        enforce domain sql rootSqlTable : SqlMetamodel::SqlTable {
                sqlForeignKey = newSqlForeignKey : SqlMetamodel::SqlForeignKey {
                        sqlOwner = rootSqlTable,
                        sqlColumn = newSqlColumn : SqlMetamodel::SqlColumn {
                                sqlOwner = rootSqlTable,
                                sqlName = sqlKeyName + '_fk',
                                sqlType = sqlKeyType
                        },
                        sqlRefersTo = newSqlKey,
                        sqlName = sqlKeyName + '_fk'
                }
        };
        when {
                not(PropertyTypeIsPrimitive(objectProperty));
        }
        where {
                AsignPkByDefault(objectProperty,newSqlKeyColumn,newSqlKey);
                GeneratePkeyFromProperties(objectProperty, newSqlKeyColumn, newSqlKey);
                PropertyToColumnOrTable(objectProperty, newSqlTable, newSqlSchema);
                sqlKeyName = newSqlKeyColumn.sqlName;
                sqlKeyType = newSqlKeyColumn.sqlType;
        }
}
```

Figura 10. Relación ComplexPropertyToTable.

GenerateForeignKeyExternalTable: cuando la propiedad de JsonSchema analizada cumple con ciertas condiciones para ser clave foránea, entonces se genera una nueva columna con su tipo y se la asigna como clave foránea.

```
relation GenerateForeignKeyExternalTable {
        jsPropertyName : String;
        sqltype : String;
        checkonly domain jsonSchema jsObject : JsonSchemaMetamodel::JsObject {
                jsProperties = objectProperty : JsonSchemaMetamodel::JsProperty {
                        jsName = jsPropertyName
                }
        };
        -- Generate Table Relation
        enforce domain sql rootSqlTable : SqlMetamodel::SqlTable {
                sqlForeignKey = newSqlForeignKey : SqlMetamodel::SqlForeignKey {
                        sqlOwner = rootSqlTable,
                        sqlName = jsPropertyName + '_fk',
                        sqlColumn = newSqlColumn : SqlMetamodel::SqlColumn {
                                sqlOwner = rootSqlTable,
                                sqlName = jsPropertyName + '_tid_fk' ,
                                sqlType = sqltype
                        }
                }
        };
        when {
                PropertyIsForeignKey(jsObject, jsPropertyName);
        }
        where {
                sqltype = PrimitiveTypeToSqlType(objectProperty);
        }
}
```

Figura 11. Relación GenerateForeignKeyExternalTable.

7. Ejemplo de la transformación JsonSchema2SqlTransformation

Esta sección presenta un ejemplo con el resultado obtenido al aplicar el código de la transformación QVT presentada en la sección anterior. La misma fue definida usando Medini QVT, una herramienta desarrollada por IKV++ integrada con Eclipse [19].

En la Figura 12 se muestra un ejemplo de un JsonSchema que contiene propiedades del tipo Integer, String y Object.

```
{  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "name": "professional_user",
  "properties": {
    "user_id": {"type": "integer", "maximum": 12345678, "minimum": 1234567},
    "creation_date": {"type": "string","format":"date-time","maxLength": 24},
    "status": { "type": "string", "maxLength": 8},
    "group_id": { "type":"integer", "maximum":2, "minimum":1},
    "history_data": {
```

```
        "type": "object",
        "properties": {
          "claims_rate": {"type": "integer", "maximum": 10, "minimum": 2},
          "operation_quantity": {"type": "integer", "maximum": 250, "minimum": 20},
          "tpv_amount": { "type": "integer", "maximum": 1250000, "minimum": 120000}
        }
      }
    }
}
```

Figura 12. JsonSchema professional_user.

La Figura 13, muestra el equivalente al modelo RDBMS_SQL. Como se puede observar la transformación JsonSchema2SqlTransformation, genera la estructura de la base de datos a partir del JsonSchema. En particular, se generó un Schema con dos tablas, donde cada una contiene sus columnas, claves primarias y foráneas transformadas a partir de las properties Integer y Object del JsonSchema.

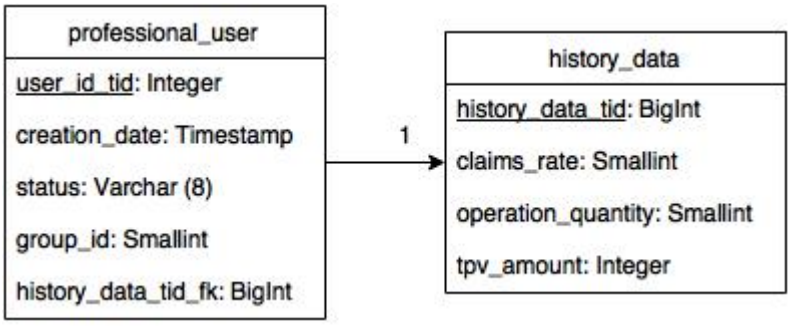

Figura 13. estructura de la base de datos.

Tener en cuenta que en esta sección se presentó un ejemplo simplificado. El ejemplo completo se puede encontrar en [21]. En el mismo se puede observar, en primer lugar, cómo se genera el JsonSchema (más complejo) a partir de un conjunto de documentos Json. En segundo lugar, cómo se genera la representación formal de la API RESTful y por último la estructura de la base de datos.

8. Conclusiones y Trabajos Futuros

En este artículo se presentó un proceso que permite a las personas que hacen uso de servicios backend (desarrolladores mobile, frontend, backend, etc.), generar de forma fácil y automática la estructura de la Base de Datos y la representación formal del servicio backend que va a gestionar los datos de su proyecto.

Como consecuencia de las transformaciones construidas en este trabajo, se puede arribar a las siguientes conclusiones:

- Mientras más representativos sean los documentos Json que se toman como entrada del proceso, se obtiene mayor amplitud de datos y de casos que van a poder ser gestionados por la API RESTful generada.
- El desarrollador frontend puede contar con un proceso donde no necesita tener conocimientos sobre APIs RESTful ni base de datos para crearlos, ni tampoco necesita de un desarrollador que lo haga.

- Se pueden generar APIs RESTful que no requieran de mucha complejidad, deduciendo los tipos de los atributos, relaciones de bases de datos, etc.
- Se disminuye costos y el tiempo en tener un proyecto en producción ya que el backend puede ser generado de forma automática y proporcionar al desarrollador frontend todos los métodos necesarios para manipular la información almacenada en la base de datos (Post, Get, Put y Delete).

Las reglas de transformación, especificadas en este trabajo, han sido validadas experimentalmente aplicándolas a un caso de estudio obtenido de una plataforma de pagos.

A continuación se nombran algunos de los trabajos a futuros propuestos:

- Extender el presente trabajo realizado para que soporte la generación de documentación automática de APIs RESTful.
- Cambiar la transformación de "JsonSchema to RDBMS_SQL" por "JsonSchema to NoSQL_Db" para poder obtener como salida una API RESTful con una base de dato no relacional, la cual está en auge en la actualidad.
- Extender el presente trabajo para generar de forma automática el código necesario para soportar la representación formal de una API RESTful y la creación de la base de datos a partir del modelo de dominio generado.
- Extender el metamodelo JsonHyperSchema y la transformación JsonSchema2JsonHyperSchema presentada en esta tesis para que soporte la gestión de Tokens en cada método que expone la API RESTful, con el objetivo de proveer un mecanismos de autenticación, permitiendo mantener una conexión segura entre el cliente y la API RESTful generada.

Referencias

[1]Fielding, R., (2000). Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, 2012, https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf, último acceso Abril 2016.

[2]Lynn, k. & Dornin, L., (2016). Modeling RESTful APIs with Json Hyper-Schema, draft-lynn-t2trg-model-rest-apis-00 (work in progress), Internet Engineering Task Force, Febrero 2016, http://tools.ietf.org/html/draft-lynn-t2trg-model-rest-apis-00, último acceso Marzo 2016.

[3]Ecma International, The Json Data Interchange Format, Standard ECMA-404, Octubre 2013, http://www.ecma-international.org/publications/standards/Ecma-404.htm, último acceso Enero 2016.

[4]Bray, T. (2015). The JavaScript Object Notation (Json) Data Interchange Format, RFC 7159, Internet Engineering Task Force, Octubre 2015, https://tools.ietf.org/html/rfc7159, último acceso Enero 2016.

[5]Galiegue, F., Zyp, K., & Court, G. (2013). Json Schema: core definitions and terminology, draft-zyp-json-schema-04 (work in progress), Internet Engineering Task Force, Enero 2013, http://tools.ietf.org/html/draft-zyp-json-schema-04, último acceso Febrero 2016.

[6]Frankel, D. (2003). Model Driven Architecture. Applying MDA to Enterprise Computing, Wiley Publishing, Inc, 2003.

[7]OMG, (2014). MDA Guide rev. 2.0, OMG , Junio 2014, http://www.omg.org/cgi-bin/doc?ormsc/14-06-01, último acceso Mayo 2016.

[8]MOF, http://www.omg.org/mof/, último acceso Febrero 2016.

[9]Warmer, J. & Kleppe, A.(2003). The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition, Addison-Wesley, Septiembre 2003.

[10]OMG, (2014). Object Constraint Language, OMG specification, Febrero 2014, http://www.omg.org/spec/OCL/2.4/, último acceso Junio 2016.

[11]OMG, (2015). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, OMG specification, Febrero 2015, http://www.omg.org/spec/QVT/1.2/, último acceso Mayo 2016.

[12]De la Cruz, A. (2013). Una aproximación MDA para la conversión entre servicios web Soap y RESTful, Tesis Maestría en Sistemas Inteligentes, universidad complutense de Madrid , Septiembre 2013, http://eprints.ucm.es/23165/1/TFM_ANAYANSI.pdf, último acceso Mayo 2016.

[13]Cánovas, J. & Cabot, J. (2013). Discovering Implicit Schemas in Json Data. International Conference on Web Engineering, Aalborg, Denmark, Julio 2013, https://hal.inria.fr/hal-00818945/document, último acceso Mayo 2016.

[14]Wischenbart, M., Mitsch, S. & Kapsammer, E. (2013). Automatic Data Transformation-Breaching the Walled Gardens of Social Network Platforms, 9th Asia-Pacific Conference on Conceptual Modelling, Australia, Febrero 2013.

[15]Parse, https://www.parse.com/, último acceso Mayo 2016.

[16]Apispark, https://apispark.restlet.com/, último acceso Mayo 2016.

[17]Morales, S., Molina, J. & Ruiz, D. (2015). Inferencia del esquema en bases de datos NoSQL a través de un enfoque MDE. Universidad de Murcia Campus Espinardo, Murcia, España, 2015, http://eventos.spc.org.pe/cibse2015/pdfs/1_SET15.pdf, último acceso Mayo 2016.

[18]Eclipse, http://www.eclipse.org/modeling/emf/, último acceso Mayo 2016.

[19]IKV++, Medini QVT, http://projects.ikv.de/qvt, último acceso Junio 2016.

[20]Favre J.M. (2004). Towards a Basic Theory to Model Model Driven Engineering, 3Rd Workshop in Software Model Engineering, WiSME, 2004.

[21]Schwartz, S., Riesco, D. & Abdelahad, C. (2016). Generación automática de servicios backend y sus bases de datos utilizando MDE. Reporte Interno, Universidad Nacional de San Luis, 2016.