Facultad de Ciencias

HERRAMIENTA PARA EL DESARROLLO DIRIGIDO POR MODELOS DE BASES DE DATOS DOCUMENTALES SEGURAS.

(Tool for the model-driven development of secure documental databases)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Sergio Gutiérrez Cano

Directores: Carlos Blanco Bueno

y Diego García Saiz

Junio – 2019

Agradecimientos

Me gustaría aprovechar estas líneas para agradecer a todas aquellas personas que me han ayudado y apoyado durante toda esta etapa académica.

En especial, me gustaría agradecer a mis directores Carlos Blanco y Diego García, por darme la oportunidad de realizar este trabajo bajo su supervisión y ayuda.

Resumen

Las características propias de los sistemas Big Data, como el volumen, variedad y velocidad de los datos, han propiciado la aparición de nuevas tecnologías para el manejo de grandes cantidades de datos, como las bases de datos NoSQL. Estas bases de datos pueden ser de distinto tipo (documentales, columnares, grafos, clave-valor) y, además, para cada uno encontramos numerosas herramientas (MongoDB, CouchDB, Cassandra, Neo4J, etc.). Además, este tipo de sistemas manejan información sensible, tanto información de negocio de la empresa como datos personales protegidos por ley.

Por lo tanto, la implementación final del sistema en una herramienta específica debe incluir las medidas de seguridad necesarias para garantizar que dicha información sensible no sea expuesta. Debido a la heterogeneidad de las tecnologías NoSQL, los responsables del diseño y desarrollo deben conocer varias herramientas específicas para realizar su implementación y añadir las reglas de seguridad necesarias.

La estrategia de desarrollo dirigida por modelos puede aplicarse a este tipo de sistemas de modo que el diseñador solo tenga que centrarse en la especificación de un modelo del sistema junto con las reglas de seguridad necesarias y, mediante la aplicación de reglas de transformación, obtenga automáticamente la implementación del sistema en una herramienta concreta. De este modo, el diseñador puede abstraerse de detalles técnicos específicos de cada herramienta.

Este proyecto consiste en la creación de una herramienta que automatice el desarrollo dirigido por modelos de bases de datos NoSQL, en concreto de bases de datos documentales. Para cumplir con el objetivo, se realizaron las siguientes tareas:

- Implementar las reglas de transformación necesarias para, partiendo de un modelo, obtener de forma automática la implementación tanto del esquema de la base de datos como de la parte de seguridad en una herramienta específica para bases de datos documentales, en este caso para MongoDB.
- Validar las transformaciones mediante un caso de estudio.

Palabras clave: Ingeniería dirigida por modelos, NoSQL, MongoDB, seguridad

Abstract

The characteristics of Big Data systems, such as the volume, variety and speed of data, have led to the emergence of new technologies for handling large amounts of data, such as NoSQL databases. These databases can be of different types (documentaries, columns, graphs, key-value) and, in addition, for each one we find numerous tools (MongoDB, CouchDB, Cassandra, Neo4J, etc.). In addition, this type of systems handle sensitive information, both business information of the company and personal data protected by law.

Therefore, the final implementation of the system in a specific tool must include the necessary security measures to ensure that such sensitive information is not exposed. Due to the heterogeneity of NoSQL technologies, those responsible for design and development must know several specific tools to implement them and add the necessary security rules.

The model-driven development strategy can be applied to this type of systems so that the designer only has to focus on specifying one system model along with the necessary security rules and, through the application of transformation rules, automatically obtains the implementation of the system in a specific tool. In this way, the designer can abstract from specific technical details of each tool.

This project consists of the creation of a tool that automates the development directed by NoSQL database models, specifically documentary databases. In order to meet the objective, the following tasks were carried out:

- Implement the necessary transformation rules to automatically obtain, from a model, the implementation of both the database schema and the security part in a specific tool for documentary databases, in this case for MongoDB.
- Validate the transformations through a case study.

Key words: model driven engineering, NoSQL, MongoDB, Security,

ÍNDICE GENERAL

1. INTRODUCCIÓN

Hoy en día, gran parte de los sistemas Big Data actuales requieren almacenar una gran cantidad de datos para funcionar. Hasta hace unos años, estos datos eran almacenados en sistemas relacionales como Oracle o MySql. Sin embargo, con la llegada de la web 2.0 y el nacimiento de sistemas como Twitter, Facebook o Amazon, continuar utilizando bases de datos relacionales resulta inviable. Por este motivo ha surgido la tecnología NoSQL (Not only SQL – No sólo SQL, de sus siglas en inglés).

Las tecnologías NoSQL tienen como objetivo dar respuesta al almacenamiento en aquellas situaciones en las que las bases de datos relacionales presentan problemas, principalmente derivados de la escalabilidad y el rendimiento cuando existen miles de usuarios concurrentes y millones de consultas. Existen diferentes tipos de bases de datos NoSQL. Estas pueden ser del tipo clave-valor, documentales, grafos y familia de columnas. Este proyecto se va a centrar en las bases de datos documentales que consisten en almacenar sus datos como documentos, generalmente en formato JSON o XML, en la figura 1 se muestra un ejemplo de este tipo de bases de datos. Son las más versátiles y se pueden emplear para sistemas idealmente pensados para trabajar en bases de datos relacionales.

Figura 1. Ejemplo de base de datos tipo documental.

Concretamente, en este proyecto se va a trabajar con MongoDB. MongoDB fue lanzado al mercado en 2009 y guarda documentos BSON (una variación de JSON que permite almacenar datos binarios). Además, posee un esquema libre, lo que quiere decir que cada entrada puede tener un esquema totalmente diferente al anterior.

En las bases de datos se almacenan multitud de datos, datos que necesitan ser protegidos, ya sea porque son importantes datos de negocio a los que no todo el mundo puede acceder, porque deben estar protegidos por ley, etc. Es por esto por lo que es necesario aplicar una seguridad para protegerlos.

Tradicionalmente, estas medidas de seguridad se han añadido a la implementación una vez ya desarrollado el sistema, y esta forma no es la más adecuada, ya que la seguridad es un requisito no funcional importante que ha de ser tenido en cuenta en todo el proceso de desarrollo. De esta forma, los requisitos de seguridad definidos se tendrán en cuenta en las decisiones de diseño y desembocarán en una implementación del sistema más robusta [1].

Por otro lado, el desarrollo de bases de datos puede realizarse mediante una estrategia de desarrollo dirigida por modelos. Dicha estrategia se basa en la definición de modelos a distintos niveles de abstracción y de transformaciones que generen de forma automatizada la implementación final del sistema, ahorrando en tiempos de desarrollo. Además, las transformaciones incluyen la lógica necesaria para transformar las partes del modelo de la mejor forma, es decir, podemos garantizar que aspectos como los requisitos de seguridad serán implementados de una forma más correcta que si se añaden medidas de seguridad directamente sobre la implementación final [12].

1.1 Objetivos

Partiendo de las ideas comentadas anteriormente, el objetivo de este proyecto es mejorar el desarrollo de bases de datos documentales NoSQL aplicando un enfoque de desarrollo dirigido por modelos, a la vez que se tiene en especial consideración los requisitos de seguridad del sistema.

Para la consecución de este objetivo se plantea desarrollar una herramienta que permita definir la base de datos documental mediante un diagrama de clases que representa el modelo conceptual del sistema. Dicho modelo estará definido de acuerdo con un metamodelo dado y permite establecer tanto los aspectos estructurales como de seguridad del sistema.

Además, se diseñarán e implementarán un conjunto de transformaciones que, partiendo del modelo conceptual de la base de datos, permitan generar de forma automática su correspondiente implementación. Para ello, se considera MongoDB como base de datos documental de destino.

2. HERRAMIENTAS Y LENGUAJES UTILIZADOS

2.1 Herramientas

Eclipse

Eclipse es una plataforma de desarrollo diseñada para ser extendida de forma indefinida a través de plug-ins. Fue concebida desde sus orígenes para convertirse en una plataforma de integración de herramientas de desarrollo. No tiene en mente un lenguaje específico, sino que es un IDE genérico, aunque goza de mucha popularidad entre la comunidad de desarrolladores del lenguaje Java usando el plug-in JDT, que viene incluido en la distribución estándar del IDE [3]. Proporciona herramientas para la gestión de espacios de trabajo, escribir, desplegar, ejecutar y depurar aplicaciones. Esta herramienta se ha utilizado en el proyecto como entorno de desarrollo en el que generar tanto el modelo como las reglas de transformación a aplicar al modelo.

Robo 3T

Robo 3T (antes Robomongo) es la interfaz gráfica de usuario ligera y gratuita para los entusiastas de MongoDB [4]. Con Robo 3T podremos realizar todas las operaciones que permite realizar MongoDb de una manera más sencilla y cómoda, a través de la propia Shell que tiene incorporada. En el proyecto se ha usado Robo 3T como gestor de bases de datos para MongoDB en el que probar que los ficheros generados por las reglas de transformación funcionaban adecuadamente.

2.2 Lenguajes utilizados

Eclipse modeling framework (EMF)

EMF es un framework de modelado e instalación de generación de código para herramientas de construcción y otras aplicaciones basadas en un modelo de datos estructurado.

A partir de una especificación de modelo descrita en XMI, EMF proporciona herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y edición basada en comandos del modelo, y un editor básico [5]. Con EMF se van a modelar el metamodelo y modelo al que aplicar las reglas de transformación.

Epsilon

Epsilon es una familia de lenguajes y herramientas para la generación de código, transformación de modelo a modelo, validación de modelos, comparación, migración y refactorización que funcionan fuera de la caja con EMF, UML, Simulink, XML y otros tipos de modelos [6]. Entre los múltiples lenguajes que lo componen, el que utilizaremos en el proyecto será EGL. En epsilon se van a escribir las reglas con las que se transformará el modelo.

Java

Java es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995 por Sun Microsystems. Hay muchas aplicaciones y sitios web que no funcionan a menos que tenga Java instalado y cada día se crean más. Java es rápido, seguro y fiable. Puede encontrarse en portátiles, centros de datos, videoconsolas, súper computadores, teléfonos móviles o incluso Internet [7]. Con Java se ha implementado el programa que ejecutará las reglas de transformación.

JSON

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para los humanos, mientras que para la máquina es simple interpretarlo y generarlo. JSON es un formato de texto completamente independiente del lenguaje, pero utiliza convenciones que son ampliamente conocidas por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos [8]. En el proyecto será el lenguaje al que se transformará el modelo para su posterior introducción a Robo3T.

3. METODOLOGÍA

Para la realización de este proyecto se va a seguir una metodología Iterativa e Incremental. Esto quiere decir que se realizarán ciclos cortos de desarrollo repetitivos, de tal manera que en cada iteración se añadirán nuevas funcionalidades a la herramienta con respecto a iteraciones anteriores, así como cambios que produzcan una mejora a las funcionalidades que se encuentran ya implementadas [9,10].

Este proyecto consta de dos grandes etapas bien diferenciadas, una primera de investigación y aprendizaje en la que se familiarizará tanto con la tecnología asociada a las bases de datos documentales (JSON, MongoDB y Robo3T) como con la asociada al desarrollo dirigido por modelos (Epsilon [11] y Eclipse modeling framwork). La segunda etapa, consiste en aplicar los conocimientos adquiridos para desarrollar una herramienta que permita modelar el sistema de acuerdo al metamodelo. Una tercera etapa para crear reglas de transformación que automaticen la implementación de la base de datos documental en MongoDB (véase figura 2). Para la validación de la segunda y tercera etapa se realizara un caso de estudio.

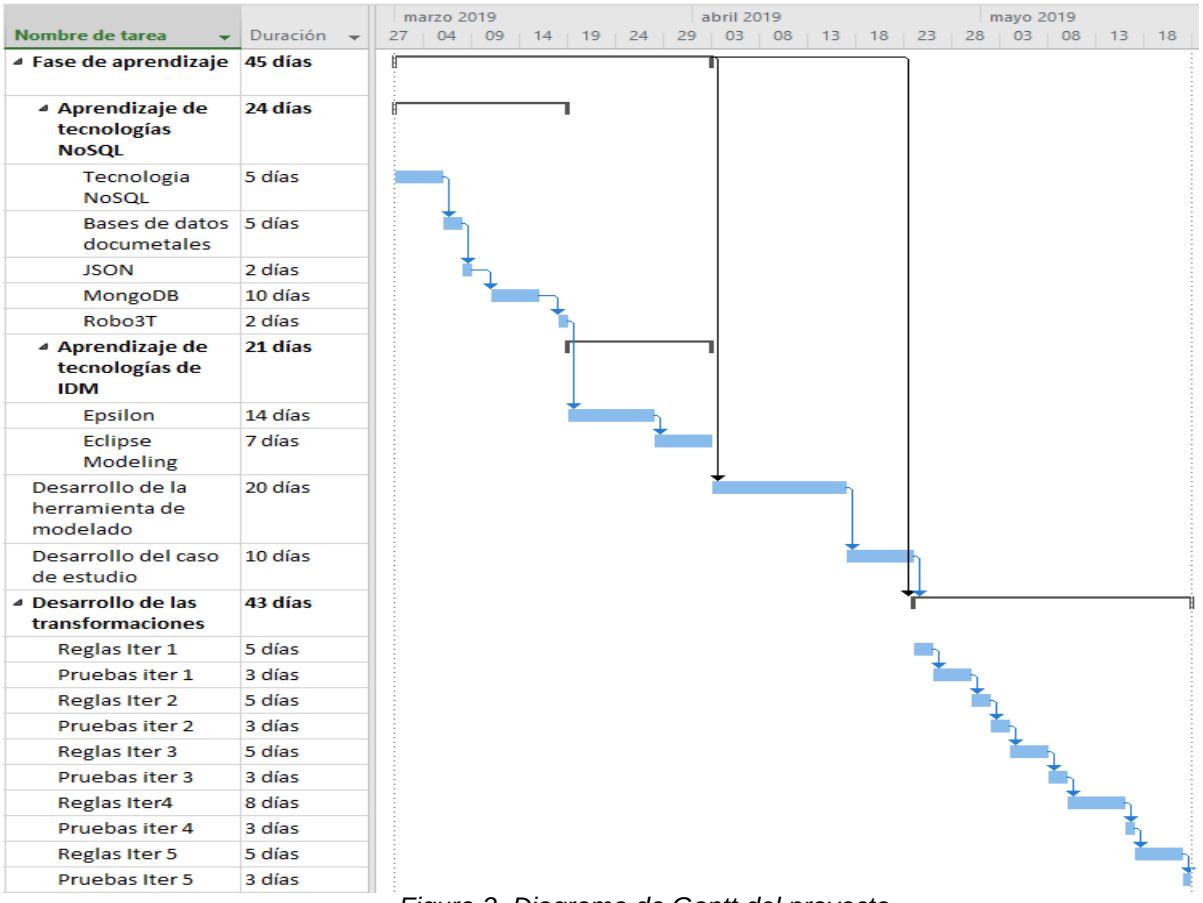


Figura 2. Diagrama de Gantt del proyecto.

La parte de desarrollo de las transformaciones se va a abordar de forma gradual, definiendo varias iteraciones centradas en la transformación de ciertos elementos del modelo.

Cada una de las iteraciones se compone además de varias fases, correspondiéndose éstas con las etapas de desarrollo de las reglas de transformación, aplicación de las reglas al ejemplo e introducción de los ficheros generados a la base de datos para su validación y las pruebas. Las iteraciones que se han definido son las siguientes:

- Colecciones: en esta primera iteración se crea es esquema de la base de datos.
- Usuarios: en esta iteración se van a crear los usuarios. A diferencia del resto de iteraciones esta no contará con una fase de pruebas.
- Rol sin restricciones: en esta iteración se crean los roles.
- Rol con restricción: en la cuarta iteración se añaden las restricciones a los roles.
- Roles heredados: en la última iteración se añade la posibilidad de crear roles que hereden restricciones de sus padres.

4. PRESENTACIÓN DEL PROBLEMA

El problema planteado en este proyecto consiste en la automatización del desarrollo de bases de datos documentales aplicando un enfoque de desarrollo dirigido por modelos, a la vez que se presta especial atención a los requisitos de seguridad del sistema.

En este contexto entran en juego por lo tanto varios elementos a analizar. Por un lado, el elemento de partida, que sería un modelo conceptual del sistema definido de acuerdo a un metamodelo. En este punto habría que analizar este metamodelo conociendo qué elementos estructurales y de seguridad permite definir.

Por otro lado, el elemento de destino, que corresponde con la implementación en un gestor de bases de datos documentales (MongoDB) y por lo tanto habrá que analizar qué mecanismos nos ofrece en cuanto a la especificación de la estructura y seguridad del sistema.

Y, por último, analizar las correspondencias entre el elemento de origen y el de destino para poder definir posteriormente las transformaciones necesarias.

A continuación, se describen en más detalle dichos elementos.

4.1 Metamodelo

Para empezar a realizar el proyecto vamos a partir de un metamodelo inicial, proporcionado por el grupo de Ingeniería Software y Tiempo Real de la UC.

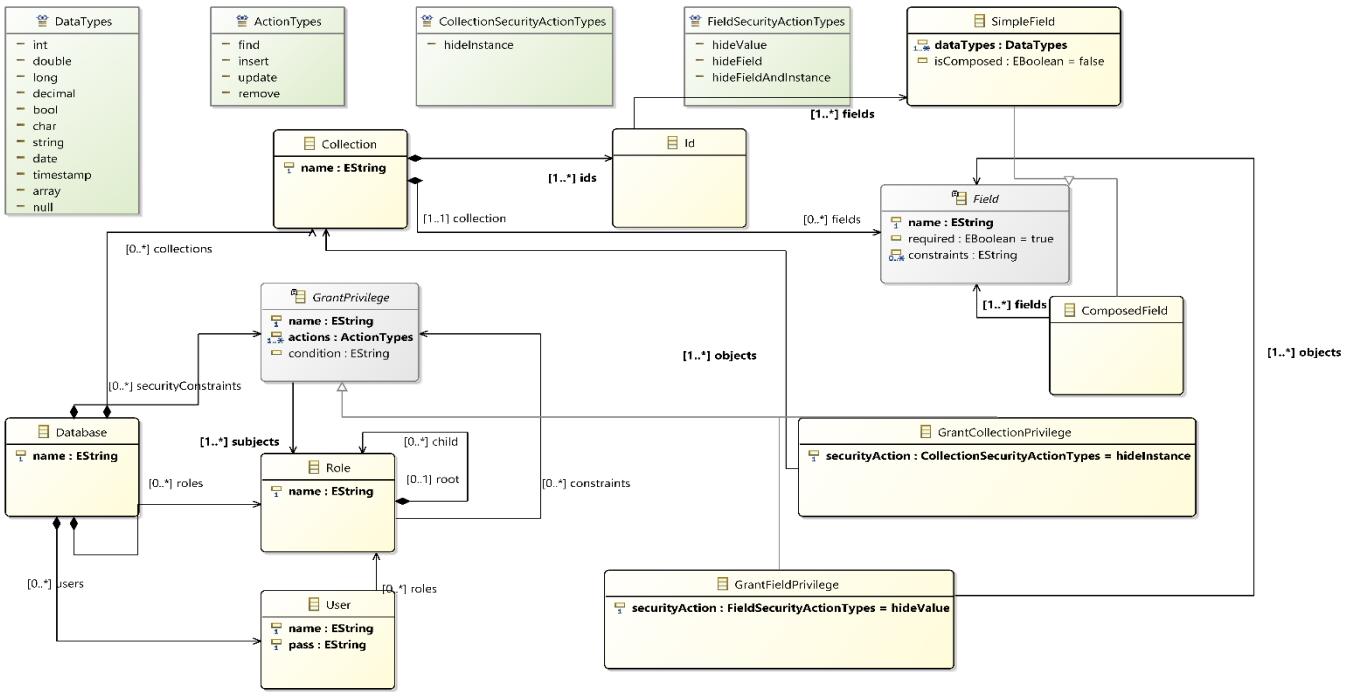


Figura 3. Metamodelo.

El metamodelo permite modelar aspectos estructurales tales como Bases de Datos (en forma de Paquetes), Colecciones y Campos. Los campos a su vez pueden ser campos simples, o campos compuestos por varios campos simples o compuestos. Además, permite especificar identificadores para colecciones, tipos de datos, limitaciones, etc.

La configuración de seguridad del sistema se define mediante el uso de un sistema basado en roles, llamado *Política de Role-Based Access Control* (*RBAC*) en la que los usuarios se clasifican en una jerarquía estructura de los roles de seguridad (*Rol*).
Una vez establecida la configuración de seguridad, se definen las restricciones de seguridad considerando un enfoque de mundo cerrado en el que los usuarios no tienen privilegios sobre ningún objeto salvo que les dé específicamente. En ese caso, tenemos que otorgar los permisos definiendo reglas de seguridad *(Grant privilege*) que indican las acciones *(insert, find, revoke, update*) que ciertos sujetos (*Rol*) pueden llevar a cabo sobre ciertos objetos (*Collection* o *Field*).
Dependiendo de los objetos, se puede seleccionar un conjunto diferente de acciones de seguridad, como por ejemplo valores de ocultación, campos, instancias, etc. Además, las restricciones de seguridad pueden incluir una condición que deba evaluarse en el momento de la ejecución, por ejemplo, para comprobar si el valor de un campo determinado es diferente al de otro [1].

No obstante, ha sido necesario realizar pequeñas modificaciones en el metamodelo (figura 3) proporcionado para poder hacer frente a las necesidades que surgieron durante el proceso de desarrollo. Los cambios realizados fueron los siguientes:

- Nuevo campo *name* a *GrantPrivilege*.
- Nueva relación de *Role* a *GrantPrivilege*.
- Nuevo campo *IsComposed* a *SimpleField*.

4.2 MongoDB

MongoDB, como ya se comentó en la introducción, es una base de datos NoSQL y es el elemento de salida de nuestra herramienta. Basa su contenido en el almacenamiento de documentos en colecciones. Con estas colecciones se define la parte estructural que se comentaba al inicio de este apartado. En cuanto a la parte de seguridad, MongoDB la implementa mediante roles. Estos roles pueden ser asignados a usuarios existentes en la base de datos, a los que se les va a otorgar la capacidad de realizar diferentes acciones que pueden ser de lectura, escritura, modificación o eliminación, sobre

colecciones concretas o campos específicos dentro de ella en función de los roles otorgados. Para poder realizar estos roles, en algunos casos va a ser necesario crear vistas. Los casos en los que es necesario crear estas vistas son todos aquellos en los que se desea otorgar permisos sobre campos concretos de la colección y cuando deseamos otorgar permisos sobre una colección sujeta a condiciones. Estas vistas no son más que consultas a las que se les pone un nombre y se almacenan en el sistema. Éstas pueden ser sobre la colección completa o se pueden hacer ciertas limitaciones dependiendo de qué es lo que queremos consultar y cómo lo queremos mostrar. Un ejemplo de cuando es necesario crear una vista, siguiendo el modelo con el que vamos a trabajar (figura 5), sería el de mostrar los documentos de una colección si el paciente es mayor de edad.
Todo esto es lo que va a producir la herramienta en distintos ficheros para su posterior introducción en Robo3T.

4.3 Correspondencias

No todos los elementos del metamodelo se mapean directamente a MongoDB como un elemento bajo el mismo nombre. En mongoDB existe el elemento *collection* y sus relaciones *id y field* se mapean en propiedades de la colección a la hora de crearla. Otros como *role y user* si contienen su homónimo en la base de datos. El elemento del metamodelo *GrantPrivilege* no se va a mapear a MongoDB, sino que serán sus clases hijo *GrantFieldPrivilege* y *GrantCollectionPrivilege* las que lo hagan, en la mayoría de los casos lo harán en forma de vistas salvo en uno de los casos, aquel que sea un *GrantCollectionPrivilege* y en la clase padre, *GrantPrivilege*, el campo *condition* este vacío, que lo hará como el campo *properties* del elemento *role*.

5. DESARROLLO DE LA HERRAMIENTA DE MODELADO

En este capítulo se desarrolla una herramienta que permite definir la base de datos documental mediante un diagrama de clases que representa el modelo conceptual del sistema. Dicho modelo está definido de acuerdo con el metamodelo descrito en el apartado 4.1.

Para desarrollar la herramienta se ha utilizado EMF. Inicialmente se ha introducido en EMF el metamodelo proporcionado para, una vez registrado, poder crear modelos en base a él. Para poder crear estos modelos se utiliza un menú grafico proporcionado por EMF en el que, inicialmente, se crea una *database* y a partir de la cual se van creando las collection con su *id* y sus *fields*, los *role*, los *users* y los GrantPrivilege (véase 4).

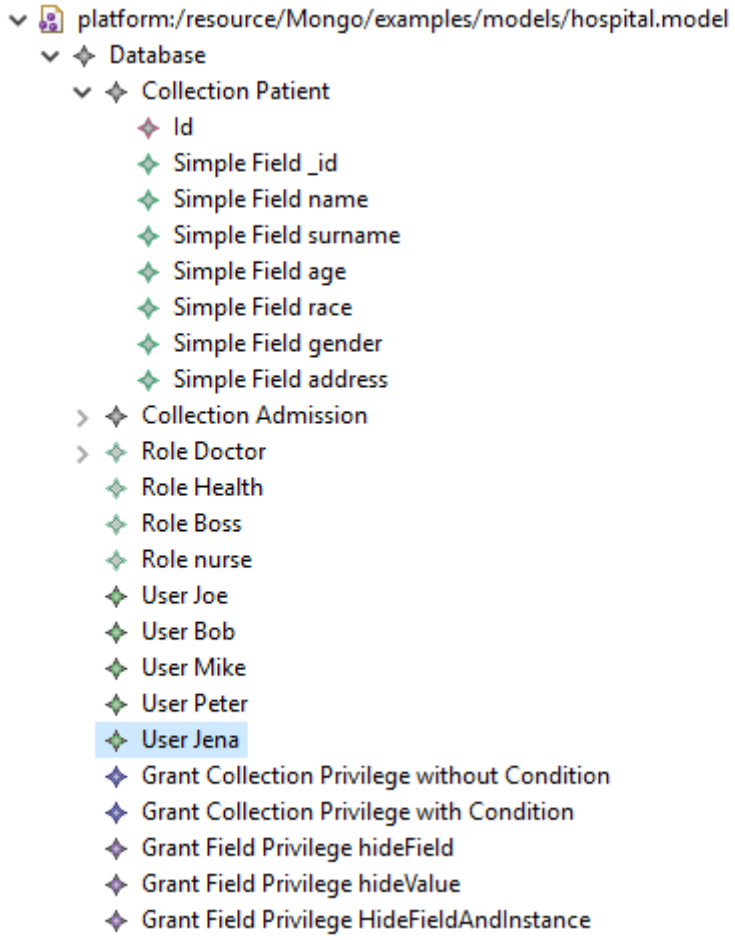


Figura 4. Menú textual de EMF.

A continuación, se ha definido un caso de estudio y se ha modelado con la herramienta. Esto ha servido para validar que la herramienta de modelado es correcta y ademas dicho caso de estudio se va a usar en el desarrollo de las transformaciones para validarlas.

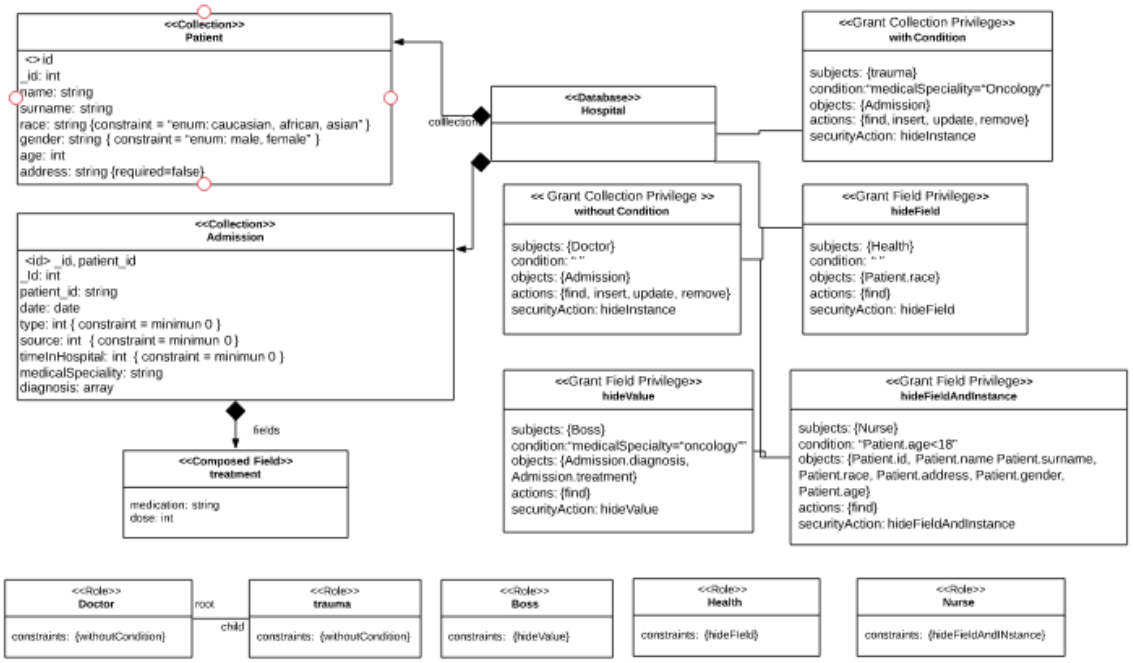


Figura 5. Modelo inicial.

La figura 5 muestra el modelo de la base de datos de un hospital que posee dos colecciones: (1) *Patient* que almacena la información relativa a los pacientes *id, name, surname, race, gender age y address,* con sus correspondientes tipos de datos y restricciones y (2) *Addmission* que almacena admisiones en el hospital, las cuales corresponden a un paciente y tienen unos datos asociados como por ejemplo la especialidad médica de la admisión, el diagnostico o el tratamiento asignado.

En cuanto a la seguridad de la base de datos, se encuentra recogida en el modelo a través de los *Grant Privilege.* Estos *Grant Privilege* los encontramos en el modelo de dos formas, *Grant Collection Privilege* y *Grant Field Privilege.* El primero destinado a otorgar privilegios sobre las colecciones y el segundo a otorgar privilegios sobre campos de las colecciones.

En el modelo se encuentran 5 privilegios, dos sobre colección y tres sobre campo. En lo referido a la seguridad de colección el primero de ellos otorga a aquellos usuarios con el rol *Doctor* permisos de búsqueda, inserción, modificación y eliminación sin ningún tipo de condición sobre la colección *Admission*.

Por el contrario, en la segunda de ellas para que los usuarios con el rol *trauma* puedan realizar estas acciones sobre la misma colección, se debe cumplir que *medicalSpeciality* sea *Oncology.*

En cuanto a la seguridad sobre campos, el primero que encontramos es aquel que permite a los usuarios pertenecientes al rol *Health* consultar la colección *Patient,* la cual mostrará todos sus campos de salvo el campo *race*. El segundo, otorga a los usuarios bajo el rol *Boss* el derecho de realizar consultas sobre los campos *id, patientNumber, date, type, source, timeInHospital, medicalSpeciality* de la coleccion *Admission* bajo cualquier circunstancia y solo se mostrarán *diagnosis y treatment* si se cumple la condición de que *medicalSpeciality* sea *Oncology*. En tercer y último lugar otorga a los usuarios que tengan el rol *nurse* el derecho de consultar aquellos documentos de la colección *Patient* cuyo campo *age* sea mayor de 18.

6. DESARROLLO DE LA SOLUCIÓN

En este capítulo se va a mostrar el desarrollo de las reglas de transformación que permiten transformar el modelo realizado en el capítulo anterior en una serie de ficheros válidos para MongoDB, estas reglas se han construido utilizando el lenguaje Epsilon.

Para realizar estas reglas se ha seguido un proceso incremental en el que se ha comenzado definiendo las transformaciones estructurales relativas a colecciones, luego las de los usuarios, después se han definido las de los roles sin permisos, seguido por la introducción de los permisos a estos roles y por último los roles heredados (véase 6).

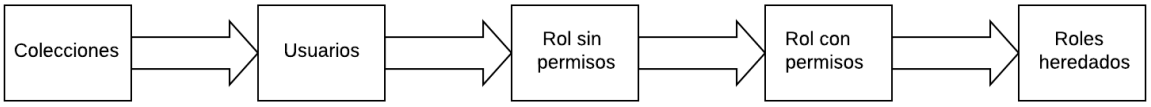


Figura 6. Pasos seguidos en el desarrollo.

Durante todas las secciones de este capítulo, salvo la sección en la que se desarrollan las reglas para generar usuarios, se van a seguir tres pasos:

- Definición de las reglas de transformación del modelo de la base de datos a la implementación en MongoDB, utilizando el lenguaje Epsilon.
- Aplicación de las reglas de transformación al modelo del caso de estudio y obtención del fichero JSON de salida.
- Validación de la implementación generada mediante la carga del fichero JSON de salida en la herramienta Robo3T y la comprobación de su correcto funcionamiento.

6.1 Colecciones simples

Esta primera iteración se divide en dos pasos. En el primer paso se tratan las colecciones que poseen todos sus campos simples.

6.1.1 Definición de las reglas de transformación

En este primer paso se construyen las reglas, que sirven para crear aquellas colecciones que contienen todos sus campos simples, y que además no contienen ninguna restricción, véase figura 7. Esto quiere decir que los campos pueden tomar cualquier valor posible dentro de su tipo de datos.

```
[% for (bd in document!Database){ %]
    [% for (co in bd.collections){%]
        db.createCollection( "[%=co.name%]",{
            validator: {
                $jsonSchema: {
                    bsonType: "object",
                    required: [[% for (fi in co.fields){%][% if(fi.isComposed == false){ %] "[%=fi.name%]", [%} %][%}%] ],
                    additionalProperties: false,
                    properties: {
                        [%for (fi in co.fields){%]
                            [%=fi.name%]: {
                            [% if(fi.isKindOf(SimpleField)==true){ %]
                            [% if(fi.isComposed == false){ %]
                            bsonType:  ["[%=fi.dataTypes.ToString().substring(1,fi.dataTypes.ToString().length()-1)%]"]
                            },
                            [%} %]
                            [%} %]
                        [%} %]
                    }
                }
            }
        })
```

Figura 7. Reglas creación colecciones con campos simples.

6.1.2 Aplicación de las reglas de transformación al modelo

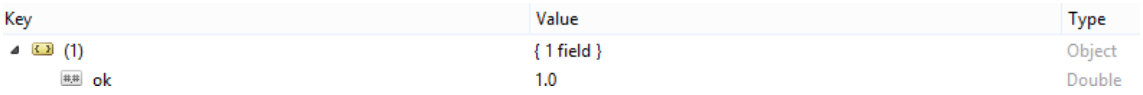
Como podemos observar en la figura 8, tras aplicar las reglas de transformación correspondientes se genera una colección llamada *Patient* cuyos campos son todos simples y ninguno contiene restricciones. Esta colección *Patient* contiene los siguientes datos: *id, name, surname, race, gender, age* y *address*.

```
db.createCollection( "Patient",{
        validator: {
                $jsonSchema: {
                bsonType: "object",
                required: [ "_id", "name", "surname", "race", "gender", "age", "address",],
                        additionalProperties: false,
                        properties: {
                                _id: {
                                        bsonType:  ["string"],
                                },
                                name: {
                                        bsonType:  ["string"],
                                },
                                surname: {
                                        bsonType:  ["string"]
                                },
                                race: {
                                        enum: ["Caucasian","African","Asian",null]
                                },
                                gender: {
                                        enum: ["Male","Female", null],
                                },
                                age: {
                                        bsonType:  ["int"],
                                },
                                address: {
                                        bsonType:  ["string"],
                                },
                        }
                }
        }
});
```

Figura 8. JSON colección *Patient.*

6.1.3 Validación y comprobación de las reglas de transformación

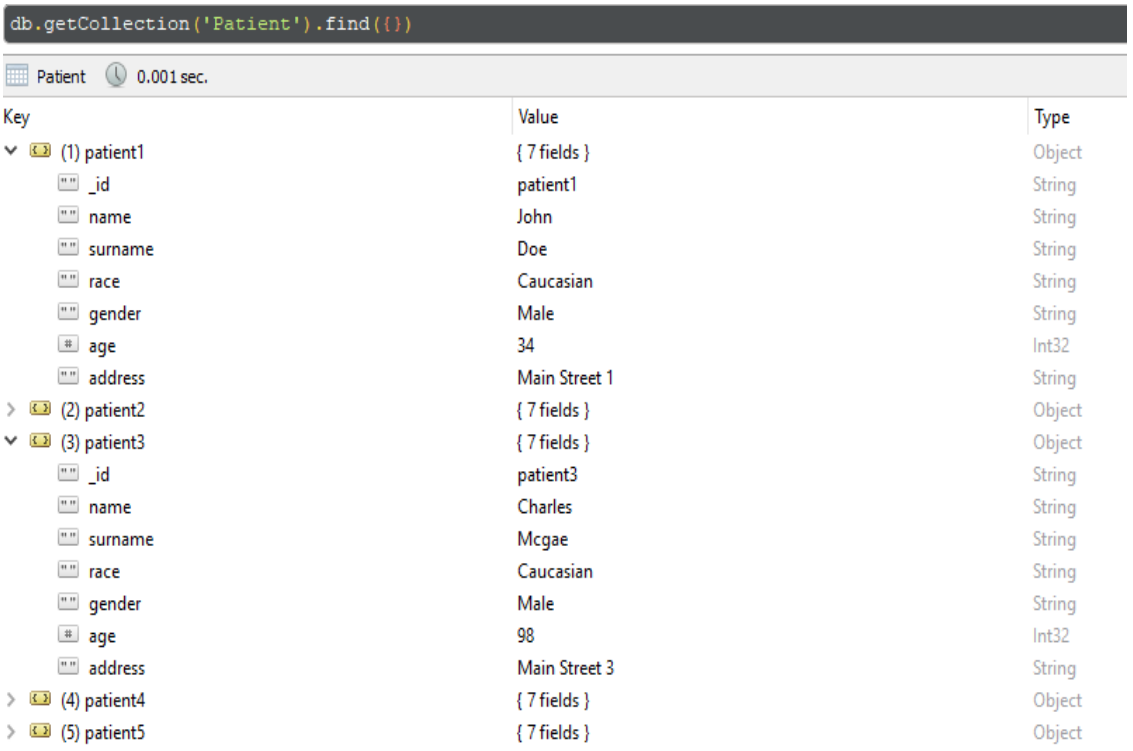
A continuación, se introduce el fichero obtenido en la anterior fase y se obtiene así una retroalimentación por parte de Robo3T, indicando que se ha creado correctamente la colección (figura 9).



Key	Value	Type
(1)	{ 1 field }	Object
ok	1.0	Double

Figura 9. Introducción de la colección *Patient* a la base de datos.

Para realizar la fase de pruebas realizamos un “find” sobre la colección para comprobar que se ha creado adecuadamente (figura 10).



```
db.getCollection('Patient').find({})
```

Patient 0.001 sec.

Key	Value	Type
(1) patient1	{ 7 fields }	Object
_id	patient1	String
name	John	String
surname	Doe	String
race	Caucasian	String
gender	Male	String
age	34	Int32
address	Main Street 1	String
(2) patient2	{ 7 fields }	Object
(3) patient3	{ 7 fields }	Object
_id	patient3	String
name	Charles	String
surname	Mcgae	String
race	Caucasian	String
gender	Male	String
age	98	Int32
address	Main Street 3	String
(4) patient4	{ 7 fields }	Object
(5) patient5	{ 7 fields }	Object

Figura 10. Consulta a la colección *Patient*.

6.2 Colecciones con campos compuestos

En este capítulo se van a añadir dos nuevas funcionalidades: los campos compuestos y las restricciones.

6.2.1 Definición de las reglas de transformación

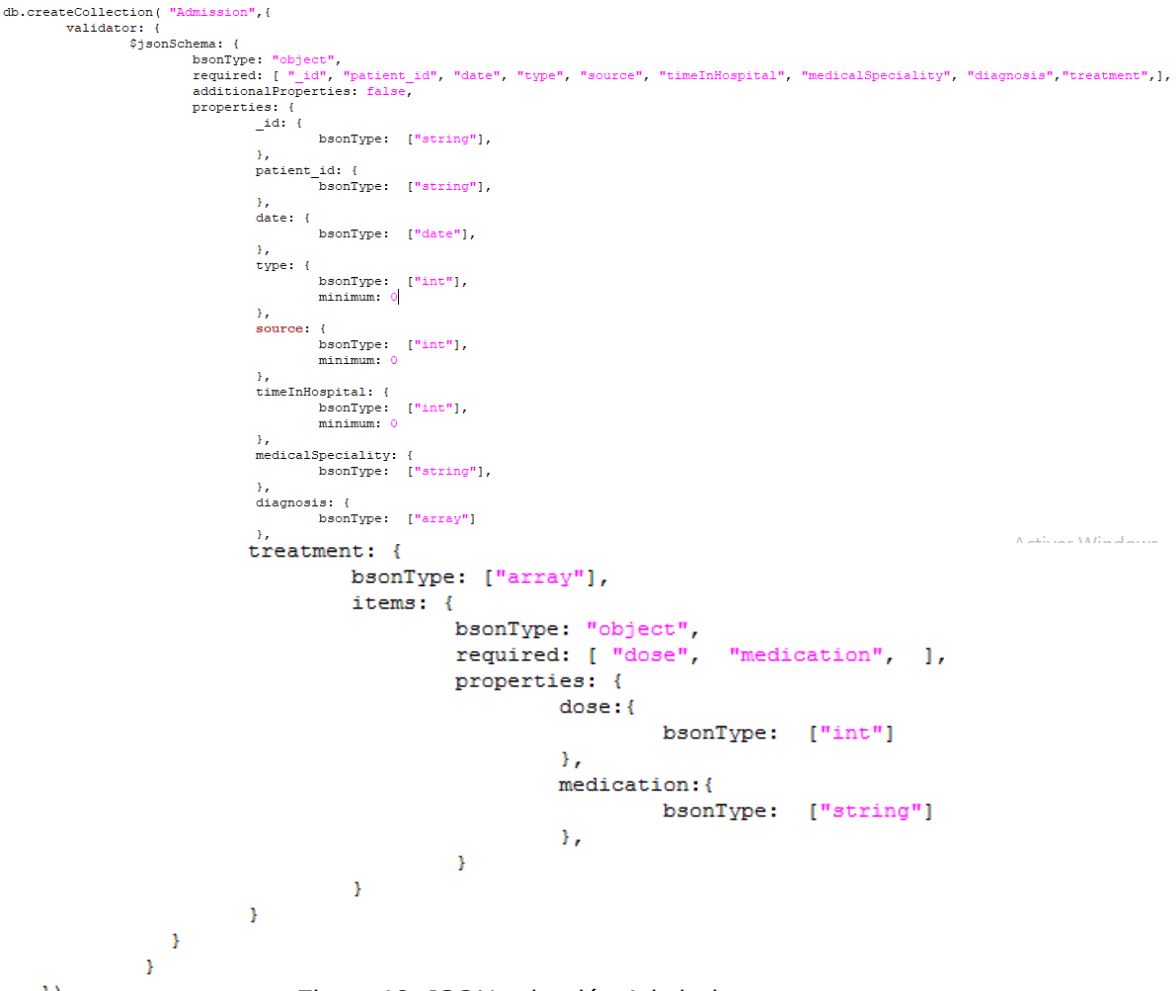
En esta fase se definen las reglas que incorporan las nuevas funcionalidades, los camapos compuestos y las restricciones para los campos, tanto simples como compuestos (ver figura 11).

```
[% for (bd in document!Database){ %]
    [% for (co in bd.collections){%]
        db.createCollection( "[%=co.name%]",{
            validator: {
                $jsonSchema: {
                    bsonType: "object",
                    required: [[% for (fi in co.fields){%][% if(fi.isKindOf(SimpleField)==true ){ %][% if(fi.isComposed == false){ %] "[%=fi.name%]",[%
                    additionalProperties: false,
                    properties: {
                        [%for (fi in co.fields){%]
                        [% if(fi.isKindOf(SimpleField)==true ){ %]
                        [% if(fi.isComposed == false){ %]
                            [%=fi.name%]: {
                                [% if(fi.constraints.toString().startsWith("[enum")==false){ %]
                                    bsonType:  ["[%=fi.dataTypes.ToString().substring(1,fi.dataTypes.ToString().length()-1)%]"][%}%]
                                        [%for (cons in fi.constraints){%],[%=cons%][%}%]
                                    },[%}%]
                            [% } else if(fi.isKindOf(ComposedField)==true) { %]
                                            bsonType: ["array"],
                                            items: {
                                                bsonType: "object",
                                                required: [[% for (fico in fi.fields){%] "[%=fico.name%]", [%} %] ],
                                                properties: {
                                                    [% for (fico in fi.fields){%]
                                                    [%=fico.name%]:{
                                                      bsonType:  ["[%=fico.dataTypes.ToString().substring(1,fico.dataTypes.ToString().length()-1)%]"]
                                                            [%for (const in fico.constraints){%],[%=const%][%}%]
                                                       },[%} %]
                                                }
                                            }[%} %] [%} %]
                    }
                }
            }
        })[%} %][% } %]
```

Figura 11. Reglas para crear todas las colecciones.

6.2.2 Aplicación de las reglas de transformación al modelo

Como se puede observar en la figura 12, tras aplicar las reglas de transformación correspondientes se genera una segunda colección llamada *Admission* para la base de datos, la cual contiene un campo compuesto como es *treatment*. Este está compuesto por los campos simples *dose* y *medication*. Además, existen restricciones para los campos *type, source* y *timeInHospital* (*mínimum 0*).



```
db.createCollection( "Admission",{
        validator: {
                $jsonSchema: {
                        bsonType: "object",
                        required: [ "_id", "patient_id", "date", "type", "source", "timeInHospital", "medicalSpeciality", "diagnosis","treatment",],
                        additionalProperties: false,
                        properties: {
                                _id: {
                                        bsonType:  ["string"],
                                },
                                patient_id: {
                                        bsonType:  ["string"],
                                },
                                date: {
                                        bsonType:  ["date"],
                                },
                                type: {
                                        bsonType:  ["int"],
                                        minimum: 0
                                },
                                source: {
                                        bsonType:  ["int"],
                                        minimum: 0
                                },
                                timeInHospital: {
                                        bsonType:  ["int"],
                                        minimum: 0
                                },
                                medicalSpeciality: {
                                        bsonType:  ["string"],
                                },
                                diagnosis: {
                                        bsonType:  ["array"]
                                },
                                treatment: {
                                        bsonType: ["array"],
                                        items: {
                                                bsonType: "object",
                                                required: [ "dose",  "medication",  ],
                                                properties: {
                                                        dose:{
                                                                bsonType:  ["int"]
                                                        },
                                                        medication:{
                                                                bsonType:  ["string"]
                                                        },
                                                }
                                        }
                                }
                        }
                }
        }
})
```

Figura 12. JSON colección *Admission*.

6.2.3 Validación y comprobación de las reglas de transformación

En la tercera fase se introduce el fichero obtenido en la anterior fase y obtenemos una retroalimentación por parte de Robo3T indicando que se ha creado correctamente la colección (véase figura 13).

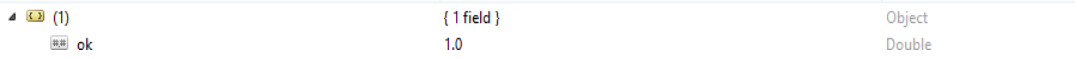


Figura 13. Introducción de la colección *Admission* a la base de datos.

Como en el caso anterior, para realizar la fase de pruebas realizamos un “find” sobre la colección para comprobar que se ha creado adecuadamente (véase figura 14).

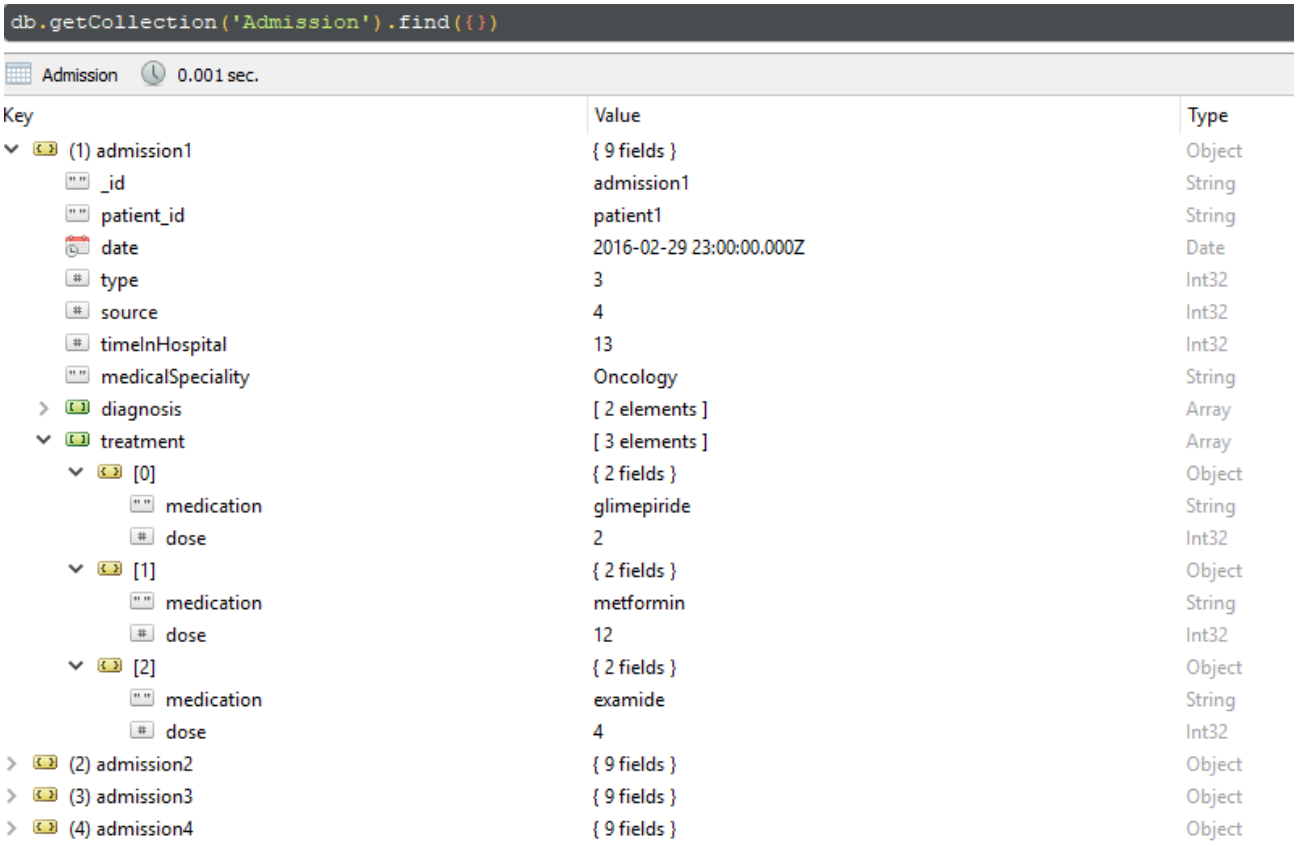


Figura 14. Consulta colección *Admission.*

Finalizada esta primera iteración ya tenemos en la base de datos las dos colecciones con las que vamos a trabajar en las siguientes iteraciones.

6.3 Usuarios

En esta iteración vamos a mostrar solamente la regla que transformará los usuarios en su correspondiente código (figura 15), pues estos son necesarios para comprobar que los roles funcionan.

```
[% for (bd in document!Database){ %]
    [% for (u in bd.users){%]
        db.createUser({
            user:"[%= u.name%]",
            pwd: "[%=u.pass%]",
            roles: [[% for (r in u.roles){%] "[%=r.name%]", [%} %]],
            mechanisms:["SCRAM-SHA-1"]
        })
    [%} %]
[% } %]
```

Figura 15. Regla transformación para creación de usuarios.

Tanto el fichero resultante de salida como las pruebas se mostrarán en las siguientes iteraciones, pues es necesario especificar el rol al que se ajusta el usuario.

6.4 Rol sin permisos

En esta sección creamos un rol sin ningún tipo de permiso sobre la base de datos. Como hemos comentado anteriormente en la sección 6.3, creamos además un usuario para probar este rol.

6.4.1 Definición de las reglas de transformación

En esta fase se definen las reglas de transformación para los roles que no tienen permisos en la base datos (véase figura 16).

```
[% for (bd in document!Database){ %]
[% for (r in bd.roles){%]
[%if(r.child.isDefined()==true){%]
db.createRole({
    role: "[%=r.name%]",
    privileges: []
    roles: []
})
[%} %]
[%} %]
[%} %]
```

Figura 16. Regla rol sin restricciones

6.4.2 Aplicación de las reglas de transformación al modelo

Como muestra la figura 17, aplicando las reglas al modelo definido, se han generado un rol llamado *Health* y un usuario, *Mike*, al que se le asigna este rol.

```
db.createRole({
        role: "Health",
        privileges: [
        ]
        roles: []
})
db.createUser({
        user:"Mike",
        pwd: "1234",
        roles: [ "Health", ],
        mechanisms:["SCRAM-SHA-1"]
})
```

Figura 17. JSON rol y usuario sin restricciones.

6.4.3 Validación y comprobación de las reglas de transformación

A continuación, se introducen el rol *Health* con su usuario, *Mike*, y se puede comprobar que se han creado correctamente (figura 18)

```
{ "role" : "Health", "privileges" : [ ], "roles" : [ ] }
```

```
Successfully added user: {
    "user" : "Mike",
    "roles" : [
        "Health"
    ],
    "mechanisms" : [
        "SCRAM-SHA-1"
    ]
}
```

Figura 18. Inserción del rol y el usuario en la base de datos.

Con este rol *Mike* no puede hacer nada en la base de datos, puesto que no tiene ningún tipo de privilegio.

6.5 Rol con permisos sobre colección sin condición

En esta cuarta iteración introducimos permisos para los roles. Estos permisos pueden ser de dos tipos: sobre la colección completa o sobre campos concretos de la colección. Además, para alguno de estos permisos se necesitará crear vistas a modo de apoyo para cumplir con la condición que piden. De este modo, la elaboración de esta iteración se realiza acorde a la descripción anterior sobre los tipos de permisos.

El permiso más simple que existe, pues no precisa crear una vista porque se van a otorgar permisos sobre la colección completa sin tener una condición, es el de asignar privilegios a un usuario sobre una colección, y es el que hemos implementado en primer lugar sobre el modelo. En este caso creamos un rol que otorga acciones al usuario sobre una de las colecciones que hemos creado anteriormente.

6.5.1 Definición de las reglas de transformación

Definición de las reglas de transformación para la creación de un rol con derechos sobre una colección sin condición (ver figura 19).

```
[% for (bd in document!Database){ %]
[% for (r in bd.roles){%]
[%if(r.child.isDefined()==true){%]
db.createRole({
    role: "[%=r.name%]",
    privileges: [
    [% for (c in r.constraints) {%]
    [% if(c.isKindOf(GrantCollectionPrivilege)==true){ %]
    [% if(c.condition.isEmpty()==true){ %]
    {resource: { db: "[%=bd.name%]",collection: "[% for (co in c.objects){%][%=co.name%]"[%} %]},
    actions:[[% for (a in c.actions){%]"[%=a%]",[%} %]]},
    [%} %]
    [%} %]
    [%} %]
    ],
    roles: []
})
[%} %]
[%} %]
[%} %]
```

Figura 19. Regla para creación de roles con restricciones sobre colecciones sin condición.

6.5.2 Aplicación de las reglas de transformación al modelo

Tras aplicar las reglas de transformación obtenemos el fichero JSON en el que se crea un rol *Doctor* con amplios poderes sobre la colección *Admission* (ver figura 20).

```
db.createRole({
        role: "Doctor",
        privileges: [

        {resource: { db: "Hospital",collection: "Admission"},actions:["find","insert","remove","update",]}
        ],
        roles: []
})
```

Figura 20. JSON rol sobre colección sin condición.

6.5.3 Validación y comprobación de las reglas de transformación

En esta fase en primer lugar se introduce el fichero obtenido en la fase anterior en Robo3T, como podemos observar en la figura 21 el rol Doctor se ha añadido correctamente en la base de datos. Este rol *Doctor* permite que cualquier usuario, que este bajo él, pueda realizar las acciones de buscar, insertar, eliminar y actualizar sobre la colección *Admission.*

Figura 21. Inserción del rol sobre colección sin condición en la base de datos.

En segundo lugar, en la fase de pruebas para probar que el rol funciona correctamente se realiza un find, tras loggearse con un usuario bajo el rol, sobre la colección *Admission* (véase figura 21).

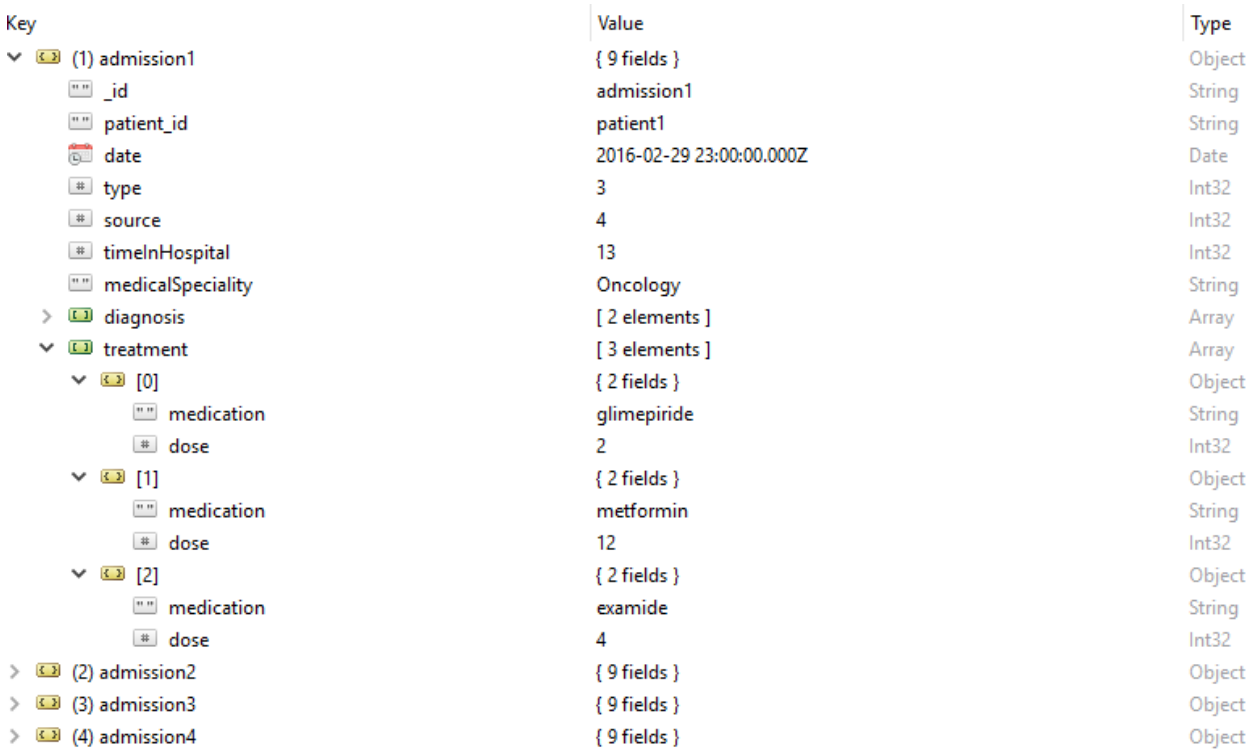


Figura 22. Consulta colección Admisión bajo rol Doctor.

6.6 Rol con permisos sobre colección con condición

6.6.1 Definición de las reglas de transformación

Ahora, trabajamos para asignar las acciones de un usuario sobre una colección bajo una condición, la cual es establecida a la hora de crear la vista (figura 23). Para introducir la condición en el campo *condition* de la clase *Grant Privilege* del metamodelo (figura 3) debemos introducirla con un formato específico, de lo contrario no se podrá crear correctamente esta regla de seguridad:

“$eq: [“$campo”, “valor”]”

donde campo es el campo de la colección al que se ajusta la condición y valor es el valor que debe tomar para que se cumpla la condición y se muestre su contenido.

```
[% for (bd in document!Database){ %]
[% for (sc in bd.securityConstraints){%]
[% if(sc.isKindOf(GrantCollectionPrivilege)==true){ %]
[% if(sc.condition.isEmpty()==false){ %]
db.createView(
    "[%=sc.name%]",
    "[%=sc.objects.name.first()%]",
    [
        {$project: {
[% for (co in bd.collections){%]
[% if(sc.objects.name.first().equals(co.name.toString())){ %]
[% for (fi in co.fields){%]
[% if(fi.isKindOf(SimpleField)==true ){ %]
[% if(fi.isComposed == false){ %]
            [%=fi.name%]: {$cond: {if: {[%=sc.condition%]}, then:"$[%=fi.name%]" , else: null}},
[% } %]
[% }else{ %]
            [%=fi.name%]: {$cond: {if: {[%=sc.condition%]}, then:"$[%=fi.name%]" , else: null}},
[% } %]
[% } %]
[% } %]
[% } %]
}
        }
    ]
);
[% } %]
[% } %]
[% } %]
[% } %]
```

Figura 23. Regla para creación de vista de restricción sobre colección.

6.6.2 Aplicación de las reglas de transformación al modelo

Tras aplicar las reglas de transformación al modelo se han generado el rol y la vista para cumplir el requerimiento. Esta vista muestra el valor de los campos de la colección *Admission* si el valor del campo *medicalSpeciality* es *Oncology*. En caso contrario, todos los campos tendrán un valor de *null* (figura 24).

```
db.createView(
        "v2",
        "Admission",
        [
                {$project: {
                        _id: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$_id" , else: null}},
                        patient_id: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$patient_id" , else: null}},
                        date: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$date" , else: null}},
                        type: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$type" , else: null}},
                        source: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$source" , else: null}},
                        timeInHospital: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$timeInHospital" , else: null}},
                        medicalSpeciality: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$medicalSpeciality" , else: null}},
                        diagnosis: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$diagnosis" , else: null}},
                        treatment: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$treatment" , else: null}},
}
                }
        ]
);
db.createRole({
        role: "Doctor",
        privileges: [
                {resource: {db: "Hospital", collection: "v2"}, actions: ["find", "insert", "update", "remove"]}],
        ],
        roles[]
})
```

Figura 24. Vista y rol para restricción sobre colección.

6.6.3 Validación y comprobación de las reglas de transformación

A continuación, se introduce el fichero de la vista obtenido en la anterior fase y se obtiene así una retroalimentación por parte de Robo3T, indicando que se ha creado correctamente la colección (figura 25).

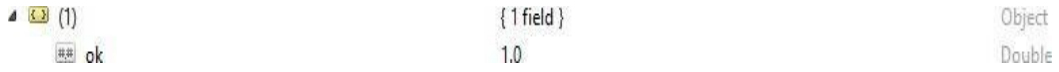


Figura 25. Inserción de la vista en la base de datos.

Para probar que este rol funciona adecuadamente basta con realizar un “find” sobre la colección y comprobar que solo se muestran aquellos valores que se ajustan a la condición del campo *medicalSpeciality* (véase figura 26).

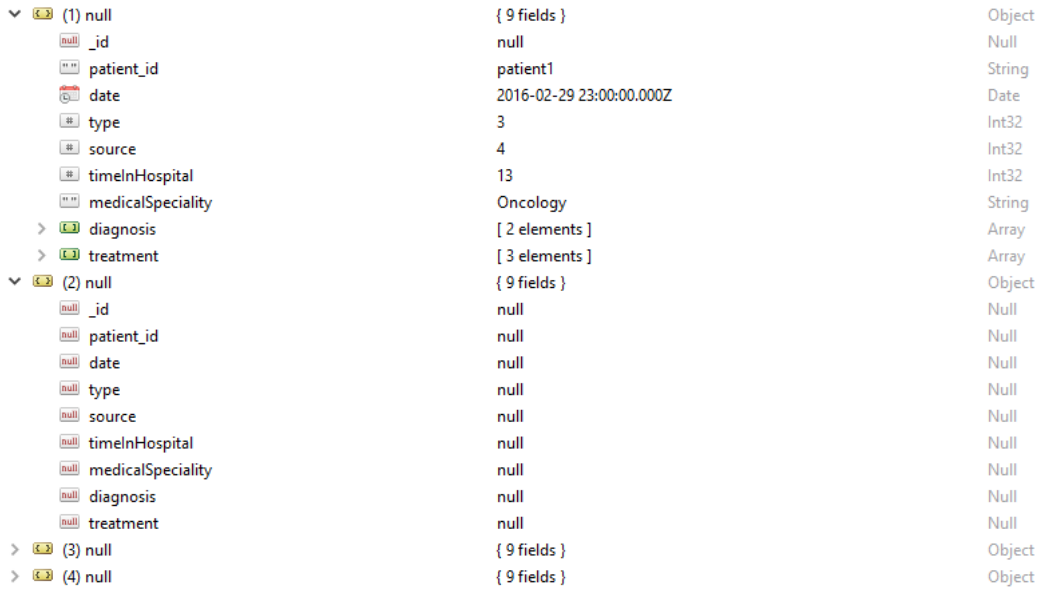


Figura 26. Consulta para probar el rol anterior.

6.7 Rol con permisos con restricción a nivel columna

Una vez se ha conseguido limitar los permisos sobre una colección, solamente queda hacer lo propio sobre los campos que queramos de una colección, ya sea hacia campos simples o campos compuestos.

Para las restricciones sobre los campos existen 3 tipos, y todos ellos requieren la necesidad de realizar una vista para poder cumplir la condición que se requiere.

En primer lugar, trataremos la restricción a nivel de columna. Esto es, cuando no queremos que se muestre, se modifique, elimine o se inserte sobre dicho campo. Esta restricción no requiere de una condición, pues simplemente restringimos las acciones sobre la columna.

6.7.1 Definición de las reglas de transformación

En este primer paso se construyen las reglas, que sirven para crear las vistas que tratan las restricciones a nivel de columna (figura 27).

```
[% for (bd in document!Database){ %]
[% for (sc in bd.securityConstraints){%]
[% if(sc.isKindOf(GrantFieldPrivilege)==true){ %]
[%if(sc.securityAction.ToString().equals("hideField")){%]
db.createView(
    "[%=sc.name%]",
    "[%=sc.objects.first().collection.name%]",
    [
        {$project: {
[% for (co in bd.collections){%]
[% if(sc.objects.first().collection.name.equals(co.name.toString())){ %]
[% var k : Integer =0; while(k<sc.objects.indexOf(sc.objects.last())+1){ %]
[%=sc.objects.at(k).name%]: 0,
[%k=k+1;%]
[%} %]
[% } %]
[% } %]
}
        }
    ]
);
[%} %][%} %][%} %][%} %]
```

Figura 27. Regla creación de vista para restricción a nivel de columna.

6.7.2 Aplicación de las reglas de transformación al modelo

La vista generada tras aplicar las reglas de transformación al modelo para conseguir una restricción a nivel de columna hace que cualquier consulta sobre la colección *Patient,* bajo el rol que la utilice, muestre el valor de todos sus casos excepto el de su *race* (véase figura 28).

```
db.createView(
        "v3",
        "Patient",
        [
                {$project: {
                        race: 0,
                }
                }
        ]
);
```

Figura 28. Vista para restriccion a nivel de columna.

6.7.3 Validación y comprobación de las reglas de transformación

En primer lugar, se introduce el fichero obtenido en la anterior fase y se obtiene así una retroalimentación por parte de Robo3T, indicando que se ha creado correctamente la vista (figura 29).

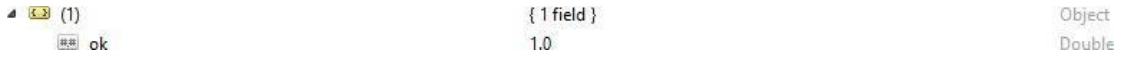
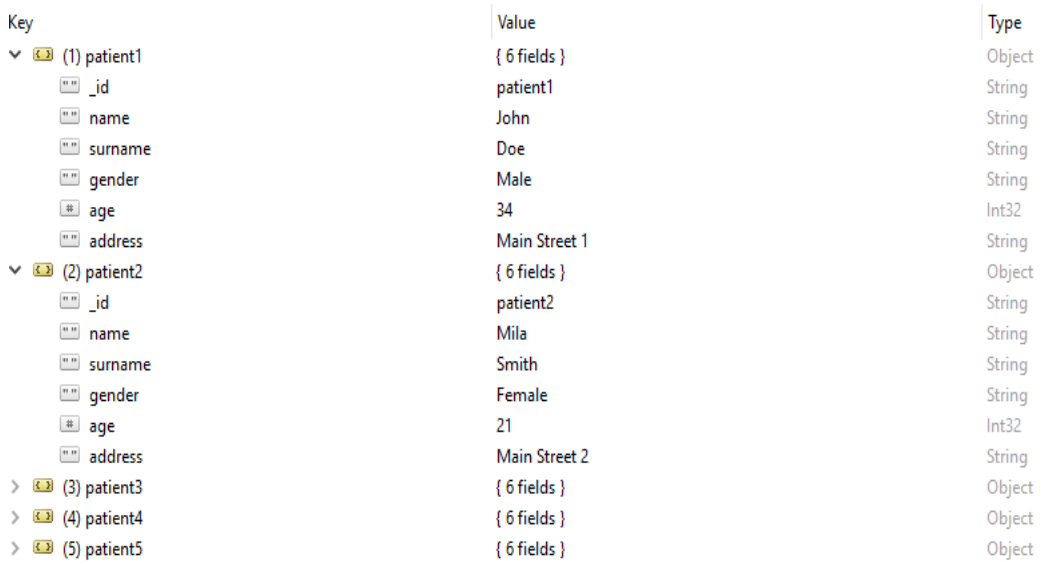


Figura 29. Inserción de la vista en la base de datos.

En segundo lugar, en la fase de pruebas realizamos una consulta sobre la colección para comprobar que efectivamente no se muestra el campo *race* (véase figura 30).



Key	Value	Type
(1) patient1	{ 6 fields }	Object
_id	patient1	String
name	John	String
surname	Doe	String
gender	Male	String
age	34	Int32
address	Main Street 1	String
(2) patient2	{ 6 fields }	Object
_id	patient2	String
name	Mila	String
surname	Smith	String
gender	Female	String
age	21	Int32
address	Main Street 2	String
(3) patient3	{ 6 fields }	Object
(4) patient4	{ 6 fields }	Object
(5) patient5	{ 6 fields }	Object

Figura 30. Consulta para comprobar el funcionamiento del nuevo rol.

6.8 Rol con permisos con mostrado de campos según condición

6.8.1 Definición de las reglas de transformación

Otro tipo de restricción posible es eliminar el valor del campo según una condición. Esta condición ha de tener el formato:

"$eq: [“$campo”, “valor”]”

donde campo es el campo de la colección al que se ajusta la condición y valor es el valor que debe tomar para que se cumpla la condición y se muestre su contenido.

```
[% for (bd in document!Database){ %]
[% for (sc in bd.securityConstraints){%]
[% if(sc.isKindOf(GrantFieldPrivilege)==true){ %]
[%if(sc.securityAction.ToString().equals("hideValue")){%]
db.createView(
    "[%=sc.name%]",
    "[%=sc.objects.first().collection.name%]",
    [
        {$project: {
[% var a : Integer =0; while(a<sc.objects.indexOf(sc.objects.last())+1){ %]
        [%=sc.objects.at(a).name%]: {$cond: {if: {[%=sc.condition%]}, then:"$[%=sc.objects.at(a).name%]" , else: null}},
[% a=a+1; %]
[%} %]
[% for (co in bd.collections){%]
[% if(sc.objects.first().collection.name.equals(co.name.toString())){ %]
[% var a : Integer =0; while(a<sc.objects.indexOf(sc.objects.last())+1){ %]
[% co.fields.removeAt(co.fields.IndexOf(sc.objects.at(a))); %]
[% a=a+1; %]
[%} %]

[%for(fi in co.fields){%]
[% if(fi.isKindOf(SimpleField)==true ){ %]
[% if(fi.isComposed == false){ %]
        [%=fi.name%]:1,
[% }%]
[% }if(fi.isKindOf(ComposedField)==true ){%]
        [%=fi.name%]:1,
[% } %][%} %][%} %][%} %]
}
        }
    ]
);
[%} %][%} %][%} %][%} %]
```

Figura 31. Regla para creación de vista que esconde el valor de un campo según una determinada condición.

6.8.2 Aplicación de las reglas de transformación al modelo

La vista generada tras aplicar las reglas de transformación al modelo muestra todos los campos de la colección *Admission* y solo muestra el valor de los campos *diagnosis* y *treatment* cuando el valor de *medicalSpeciality* es *Oncology* en caso contrario se muestra *null* (figura 32).

```
db.createView(
        "v4",
        "Admission",
        [
                {$project: {
                diagnosis: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$diagnosis" , else: null}},
                treatment: {$cond: {if: { $eq: ["$medicalSpeciality","Oncology"]}, then:"$treatment" , else: null}},


                _id:1,

                patient_id:1,

                date:1,

                type:1,

                source:1,

                timeInHospital:1,

                medicalSpeciality:1,

}
                }
        ]
);
```

Figura 32. JSON vista.

6.8.3 Validación y comprobación de las reglas de transformación

A continuación, se introduce la vista en la base de datos (figura 33).

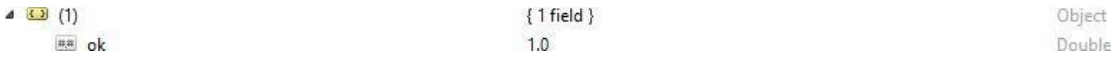


Figura 33. Inserción de la vista en la base de datos

En la fase de pruebas realizamos el mismo proceso que anteriormente, consultamos la colección y comprobamos que los valores de los campos se ocultan cuando se cumple la condición establecida (véase figura 34).

Figura 34. Consulta que prueba el funcionamiento del nuevo rol.

6.9 Rol con permisos con filtrado de documentos

6.9.1 Definición de las reglas de transformación

En esta fase se definen las reglas de transformación para la última restricción, que tiene que ver con filtrar documentos. Con el operador $match se van a mostrar los campos deseados solamente de aquellos documentos de la colección, que cumplen la condición (véase figura 35).

```
[% for (bd in document!Database){ %]
[% for (sc in bd.securityConstraints){%]
[% if(sc.isKindOf(GrantFieldPrivilege)==true){ %]
[%if(sc.securityAction.ToString().equals("hideFieldAndInstance")){%]
db.createView(
    "[%=sc.name%]",
    "[%=sc.objects.first().collection.name%]",
    [
        {$project: {
[% for (co in bd.collections){%]
[% if(sc.objects.first().collection.name.equals(co.name.toString())){ %]
[% var k : Integer =0; while(k<sc.objects.indexOf(sc.objects.last())+1){ %]
[%=sc.objects.at(k).name%]: 1,
[%k=k+1;%]
[%} %]
[% } %]
[% } %]

}
        },
        {$match: {[%=sc.condition%]}}
    ]
);
[%} %][% } %][%} %][%} %]
```

Figura 35. Regla para vista de filtrado de documentos según condición.

6.9.2 Aplicación de las reglas de transformación al modelo

Tras aplicar las reglas de transformación al modelo se ha generado la vista en la que se están mostrando aquellos documentos de la colección *Patient* cuyo valor del campo *age* es mayor que 18 (véase figura 36).

```
db.createView(
        "v5",
        "Patient",
        [
                {$project:
                        {_id: 1, age: 1,address: 1,gender: 1,name: 1,surname: 1,race: 1,}
                },
                {$match: {age :  {$gte: 18}}}
        ]
);
```

Figura 36. JSON vista de filtrado de documentos.

6.9.3 Validación y comprobación de las reglas de transformación

En esta fase en primer lugar se introduce el fichero obtenido en la fase anterior en Robo3T, como podemos observar en la figura 37 la vista se ha añadido correctamente en la base de datos.

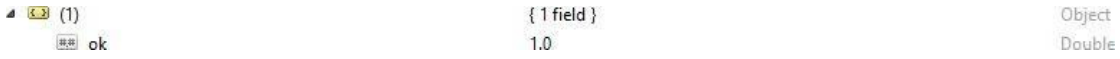
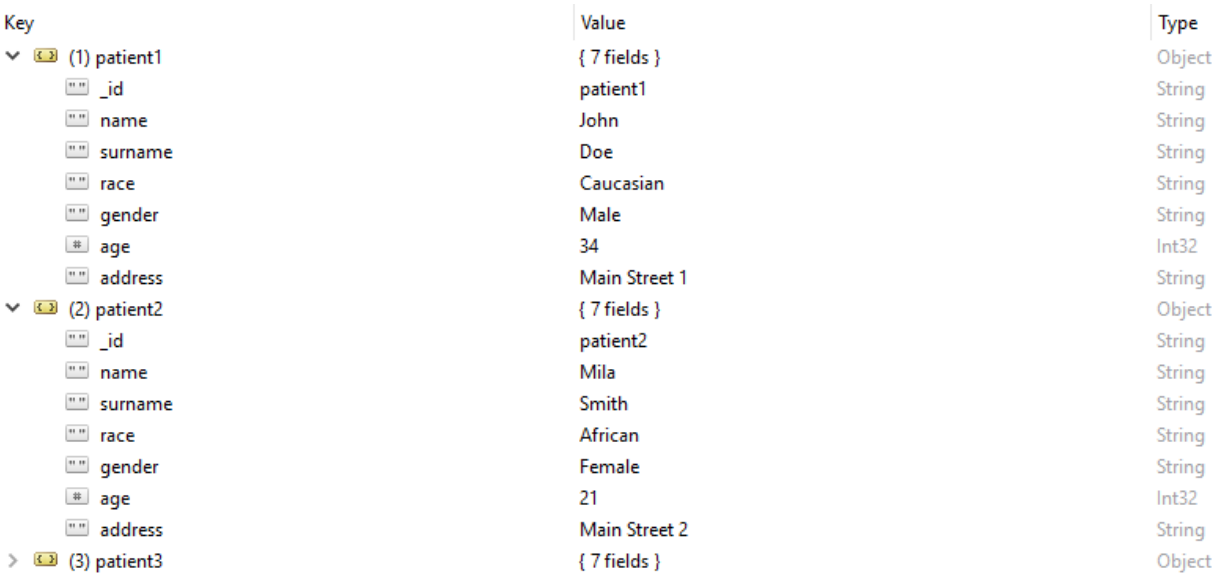


Figura 37. Inserción de la vista de filtrado de documentos en la base de datos.

En la fase de pruebas se realizada la correspondiente consulta para comprobar que efectivamente se cumple la condición, pues de los 5 documentos que tenemos en la base de datos solo se están mostrando 3 (véase figura 38).



Key	Value	Type
(1) patient1	{ 7 fields }	Object
_id	patient1	String
name	John	String
surname	Doe	String
race	Caucasian	String
gender	Male	String
age	34	Int32
address	Main Street 1	String
(2) patient2	{ 7 fields }	Object
_id	patient2	String
name	Mila	String
surname	Smith	String
race	African	String
gender	Female	String
age	21	Int32
address	Main Street 2	String
(3) patient3	{ 7 fields }	Object

Figura 38. Consulta para comprobar funcionamiento del rol.

6.10 Roles heredados

6.10.1 Definición de las reglas de transformación

El metamodelo original permite la existencia de roles hijos. En esta fase de la última iteración se construyen las reglas que sirven para los roles heredados. Esta funcionalidad permite la herencia de todos los privilegios de todos los roles padre sobre el rol hijo.

```
[% for (bd in document!Database){ %]
[% for (r in bd.roles){%]
[%if(r.child.isDefined()==true){%]
[% for (rh in r.child){%]
db.createRole({
    role: "[%=rh.name%]",
    privileges: [
    [% for (c in rh.constraints) {%]
    [% if(c.isKindOf(GrantCollectionPrivilege)==true){ %]
    [% if(c.condition.isEmpty()==true){ %]
    {resource: { db: "[%=bd.name%]", collection: "[% for (co in c.objects) {%][%=co.name%]"[%} %]}, actions:[[% for (a in c.actions) {%]"[%=a%]",[%} %]]},
    [%} %]
    [%else{%]
    {resource: { db: "[%=bd.name%]", collection: "[%=c.name%]"}, actions:[[% for (a in c.actions) {%]"[%=a%]",[%} %]]},
    [%}%]
    [%} %]
    [%} %]
    ],
    roles: [[% for (rp in rh.root) {%] "[%=rp.name%]" [%} %]]
})
[%} %]
[%} %]
[%} %]
[%} %]
```

Figura 39. Regla creación de roles heredados.

6.10.2 Aplicación de las reglas de transformación al modelo

Como se puede observar en el fichero JSON (figura 40) se ha creado un rol llamado *trauma* que, pese a no tener privilegios, hereda todas las restricciones que posee el rol *Doctor*.

```
db.createRole({
        role: "trauma",
        privileges: [
        ],
        roles: [ "Doctor" ]
})
```

Figura 40. JSON rol heredado.

6.10.3 Validación y comprobación de las reglas de transformación

En esta fase en primer lugar se introduce el fichero obtenido en la fase anterior en Robo3T, como podemos observar en la figura 41 el rol *trauma* se ha añadido correctamente en la base de datos.

```
{ "role" : "trauma", "privileges" : [ ], "roles" : [ "Doctor" ] }
```

Figura 41. Introducción del rol heredado en la base de datos.

Para la fase de pruebas es necesario recordar que el rol *Doctor* tiene los privilegios otorgados en el apartado 6.5. Ahora para probar que el rol funciona correctamente se realiza un find, sobre la colección *Admission* (figura 42).

Key	Value	Type
(1) admission1	{ 9 fields }	Object
_id	admission1	String
patient_id	patient1	String
date	2016-02-29 23:00:00.000Z	Date
type	3	Int32
source	4	Int32
timeInHospital	13	Int32
medicalSpeciality	Oncology	String
diagnosis	[2 elements]	Array
treatment	[3 elements]	Array
[0]	{ 2 fields }	Object
medication	glimepiride	String
dose	2	Int32
[1]	{ 2 fields }	Object
medication	metformin	String
dose	12	Int32
[2]	{ 2 fields }	Object
medication	examide	String
dose	4	Int32
(2) admission2	{ 9 fields }	Object
(3) admission3	{ 9 fields }	Object
(4) admission4	{ 9 fields }	Object

Figura 42. Consulta colección *Admission* bajo el rol trauma.

7. CONCLUSIONES

Tras la realización de este proyecto se han alcanzado los objetivos establecidos al inicio de este:

- Se ha desarrollado una herramienta que permite definir la base de datos documental mediante un diagrama de clases que representa el modelo conceptual del sistema. Dicho modelo está definido de acuerdo con un metamodelo dado y permite establecer tanto los aspectos estructurales como de seguridad del sistema.

- Se han diseñado e implementado un conjunto de transformaciones que, partiendo del modelo conceptual de la base de datos documental, permiten generar de forma automática su correspondiente implementación en MongoDB. Dicha implementación incluye tanto los aspectos estructurales como la parte de seguridad definida en el modelo.

- Se ha desarrollado un caso de estudio de un hospital que presenta la suficiente diversidad de situaciones para poder validar tanto la herramienta de modelado como las reglas de transformación.

En cuanto a los próximos pasos para seguir desarrollando esta herramienta, se espera que siga evolucionando para ser capaz:

- Añadir a la herramienta de modelado una opción de definición de modelos más cercana a la utilizada por las herramientas de modelado UML usadas por los diseñadores, es decir, basada en una paleta de elementos que pueden ser arrastrados a un área de dibujo, modificados, conectados entre sí, etc.
- Añadir a la herramienta la capacidad de generar código para otro tipo de base de datos documental de destino, como CouchDB.
- Añadir a la herramienta la capacidad de generar código para otro tipo de base de datos NoSQL con una tecnología distinta a la documental, como de grafos o columnar.

BIBLIOGRAFÍA

[1] Blanco C., García-Saiz D., Peral J., Maté A., Oliver A., Fernández-Medina E. (2018) How the Conceptual Modelling Improves the Security on Document Databases. In: Trujillo J. et al. (eds) Conceptual Modeling. ER 2018. Lecture Notes in *Computer Science*, vol 11157. Springer, Cham

[2] de la Vega A., García-Saiz D., Blanco C., Zorrilla M., Sánchez P. (2018) Mortadelo: A Model-Driven Framework for NoSQL Database Design. In: Abdelwahed E., Bellatreche L., Golfarelli M., Méry D., Ordonez C. (eds) Model and Data Engineering. MEDI 2018. Lecture Notes in *Computer Science*, vol 11163. Springer, Cham

[3] Eclipse Foundation | The Eclipse Foundation. (s.f.). Recuperado 15 junio, 2019, de https://www.eclipse.org/org/foundation/

[4] Robo 3T - formerly Robomongo — native MongoDB management tool (Admin UI). (s.f.). Recuperado 16 junio, 2019, de https://robomongo.org/

[5] Eclipse Modeling Project | The Eclipse Foundation. (s.f.). Recuperado 16 junio, 2019, de https://www.eclipse.org/modeling/emf/

[6] Epsilon. (s.f.). Recuperado 16 junio, 2019, de https://www.eclipse.org/epsilon/

[7] ¿Qué es Java y para qué es necesario? (2014, 8 agosto). Recuperado 16 junio, 2019, de https://www.java.com/es/download/faq/whatis_java.xml

[8] JSON. (s.f.). Recuperado 16 junio, 2019, de https://www.json.org/

[9] Sommerville. *Ingeniería del Software*, 9a Edición. Addison-Wesley, 2012. 4

[10] Figueroa, R. G., Solís, C. J., & Cabrera, A. A. (2008). *Metodologías tradicionales vs. Metodologías ágiles*. Universidad Técnica Particular de Loja, Escuela de Ciencias de la Computación.

[11] Dimitris Kolovos, Louis Rose, Antonio García-Domínguez, Richard Paige. (2018) *The Epsilon Book*.

[12] D. Poole, John. (2001). Model-Driven Architecture: Vision, Standards And Emerging Technologies.

[13] Eric Redmond. (2012) Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.

[14] Lith, A.P., & Mattsson, J. (2010). Investigating storage solutions for large data - A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data.