

**Instituto Politécnico Nacional
Escuela Superior de Cómputo**

Trabajo Terminal No. 2019-B052

Herramienta para transformar un modelo
entidad-relación a un modelo no relacional

Presentan:
Aparicio Quiroz Omar
Martínez Acosta Eduardo

Directoras:
M. en C. Ocotitla Rojas Nancy
D. en C. Chavarria Baez Lorena

Índice general

1. Introducción	1
1.1. Descripción del problema	2
1.2. Propuesta de solución	3
1.3. Justificación	3
1.4. Objetivos	4
1.5. Alcance y limitaciones	4
2. Estado del arte	8
2.1. Kashliev Data Modeler	8
2.2. NoSQL Workbench for Amazon DynamoDB	9
2.3. HacKolade	9
2.4. Mortadelo	10
2.5. NoSE: Schema Design for NoSQL Applications	12
2.6. Conclusiones	14
3. Marco teórico	15
3.1. Modelos de datos para bases de datos	15
3.1.1. Modelo entidad-relación	16
3.1.1.1. Entidades	16
3.1.1.2. Atributos	16
3.1.1.3. Relaciones	17
3.1.1.4. Resumen de la notación para los diagramas ER	19
3.1.2. Modelo relacional	20
3.1.2.1. Relaciones	21
3.1.2.2. Claves	22
3.1.2.3. Restricciones de integridad	22
3.1.2.4. Propiedades de las relaciones	23

3.1.2.5. Structured Query Language	23
3.1.3. Modelos NoSQL	23
3.1.3.1. Clave-Valor	24
3.1.3.2. Orientado a documentos	25
3.1.3.3. Orientado a columnas	26
3.1.3.4. Orientado a grafos	27
3.2. Ingeniería Dirigida por Modelos	28
3.2.1. Lenguaje específico de dominio	28
3.2.2. Transformación entre modelos	29
3.3. Tecnologías a usar	30
3.3.1. Hypertext Transfer Protocol	30
3.3.2. HTML 5	31
3.3.3. Style Sheet Language: CSS vs SASS	32
3.3.4. JavaScript vs TypeScript	34
3.3.5. JavaScript Web Frameworks: Vue/Nuxt vs React vs Angular .	36
3.3.6. CSS Frameworks: Vuetify vs Bootstrap	38
3.3.7. MySQL vs MongoDB	39
3.3.8. Bibliotecas JavaScript para diagramado: GoJS vs Fabric.js vs D3.js	40
3.3.9. Python 3	41
3.3.10. Back end: Django vs Flask	42
3.3.11. Herramientas para ingeniería dirigida por modelos	43
3.4. Conclusiones	45
4. Análisis del diseño	46
4.1. Metodología	46
4.1.1. Scrum	47
4.2. Factibilidad	49
4.2.1. Factibilidad técnica	49
4.2.2. Factibilidad económica	51
4.2.3. Costos de desarrollo	54
4.3. Análisis del sistema	56
4.3.1. Historias de usuario	56
4.3.2. Lista de producto	62
4.3.3. Algoritmos para el desarrollo del sistema	65

4.3.3.1.	Validación estructural diagrama entidad-relación	65
4.3.3.2.	Modelo entidad-relación a relacional	70
4.3.3.3.	Obtención de esquema SQL desde modelo relacional	73
4.3.3.4.	Modelo entidad-relación básico a Generic Data Metamodel	74
4.3.3.5.	Generic Data Metamodel a modelo lógico NoSQL	75
4.3.3.6.	Modelo lógico NoSQL a modelo físico en MongoDB	79
4.4.	Conclusiones	81
5.	Diseño del sistema	83
5.1.	Diagrama de clases	83
5.2.	Diagrama de la base de datos	85
5.3.	Arquitectura del sistema	86
5.3.1.	Arquitectura cliente-servidor	86
5.3.2.	Patrón de diseño MVC	86
5.4.	Conclusiones	88
6.	Desarrollo del sistema	89
6.1.	Front end	89
6.1.1.	Pantallas del front end	91
6.1.2.	Diagramado del modelo entidad-relación	96
6.1.3.	Sentencias SQL	98
6.1.4.	Modelo conceptual NoSQL	99
6.1.5.	Modelo lógico y físico NoSQL	99
6.2.	Back end	102
6.2.1.	El api flask	103
6.2.2.	Los servicios web	105
6.2.3.	Servicios para login	107
6.2.4.	Servicios para el usuario	108
6.2.5.	Servicios para el diagrama	109
6.2.6.	Servicio para el modelo relacional	110
6.3.	Algoritmos	112
6.3.1.	Validación estructural del diagrama entidad-relación	112
6.3.2.	Transformación del diagrama entidad-relación al modelo relacional y generar las sentencias SQL	116

6.3.3.	Transformación modelo entidad-relación a entidades del modelo conceptual GDM	122
6.3.4.	Parser del archivo de texto simple GDM a su .model	124
6.3.5.	Transformación de una instancia GDM al modelo lógico orientado a documentos	125
6.3.6.	Transformación del modelo lógico orientado a documentos a MongoDB	128
6.4.	Conclusiones	130
7.	Caso de estudio	131
7.1.	Diagramado modelo entidad-relación básico	131
7.2.	Validación modelo entidad-relación básico	131
7.3.	Generación sentencias SQL	132
7.4.	Modelo conceptual NoSQL	134
7.5.	Modelo lógico NoSQL	136
7.6.	Modelo físico NoSQL	137
7.7.	Conclusiones	140
8.	Trabajo a futuro	141
A.	Apéndice	142
A.1.	Unified Modeling Language	142
A.1.1.	Diagramas de clases	142

Índice de figuras

2.1. Mortadelo	10
2.2. Generic Data Metamodel	11
2.3. Notación textual del GDM	12
3.1. Tabla en el modelo relacional	21
3.2. <i>Bucket</i> en el almacenamiento clave-valor	24
3.3. Colección en el almacenamiento de documentos	25
3.4. Familia de columnas	27
3.5. modelo conceptual orientado a grafos	27
4.1. Consulta básica	74
4.2. Modelo lógico orientado a documentos propuesto por Alfonso de la Vega	75
4.3. Generic Data Metamodel	76
4.4. Access Tree - Modelo lógico orientado a documentos	77
4.5. Access query Q4	77
5.1. Diagrama de clases	84
5.2. Diagrama de la base de datos.	85
5.3. Arquitectura cliente-servidor.	86
5.4. Esquema MVC	87
6.1. Ciclo de vida de Vue/Nuxt	90
6.2. Pantalla de bienvenida	91
6.3. Pantalla de login	92
6.4. Pantalla de alta de usuario	92
6.5. Pantalla del diagramador entidad-relación	93
6.6. Pantalla del diagramador entidad-relación	93
6.7. Pantalla de erroresal validar un diagrama.	94

6.8. Pantalla de erroresal validar un diagrama.	94
6.9. Pantallas de las sentencias SQL equivalentes al diagrama ER.	95
6.10. Pantalla para agregar una consulta de acceso.	96
6.11. Pantalla con multiples consultas de acceso.	96
6.12. Estructura del proyecto backend.	104
6.14. Generación del token de autenticación.	108
6.15. Pantalla de los servicios relacionados a un usuario.	109
6.17. Pantalla del servicio para validar un diagrama ER con las reglas en la descripción.	110
6.18. Respuesta para un diagrama ER no valido.	111
6.19. Respuesta con las sentencias SQL equivalentes al diagram ER.	111
6.20. Fragmento de código del template para construir la sentencia <i>CREATE TABLE</i>	121
6.21. Fragmento de código del template para construir la sentencia <i>PRIMARY KEY</i>	121
6.22. Fragmento de código del template para construir la sentencia <i>FOREING KEY</i>	122
7.1. Diagrama entidad-relación de Venues	131
7.3. Validación del modelo Venues.	132
7.4. Sentencias SQL de Venues.	133
7.5. Ejecución de las sentencias SQL en MySQL Workbench.	134
7.6. Entidades del modelo conceptual GDM	134
7.7. Entidades y consultas del modelo conceptual GDM	135
7.8. Modelo lógico orientado a documentos de Venues	137
7.9. Prueba de sentencias del modelo físico orientado a documentos de Venues en MongoDB	140
A.1. Asociación	143
A.2. Agregación	144
A.3. Asociación rombo sin rellenar y composición rombo negro	144

Índice de tablas

2.1. Tabla comparativa de las herramientas estudiadas y la propuesta de solución	14
3.1. Notación del modelo entidad-relación	20
4.1. Computadoras con las que se cuenta	51
4.2. Cálculo de las métricas por puntos de función	52
4.3. Factores de ajuste	53
4.4. Costos del personal	54
4.5. Costos por licencias de software	55
4.6. Costos por servicios	55
4.7. Lista de producto	64
4.8. Tipos de relación recursiva válidos según las restricciones de cardinalidad	68
4.9. Resumen de reglas de validez para relaciones recursivas con ejemplos	69
4.10. Resumen de reglas de validez para relaciones binarias con ejemplos	70

Resumen

En el presente trabajo se propone una aplicación web que brinde la funcionalidad de una herramienta CASE de acuerdo con reglas de validación establecidas para el modelo entidad-relación, permita el mapeo del modelo entidad-relación básico con consultas al modelo relacional o a un modelo conceptual NoSQL y permita generar sus esquemas de bases de datos, respectivamente; se da una introducción a los sistemas NoSQL, la problemática que pretende resolver este trabajo con la propuesta de solución; se justifica la aplicación en un alcance para un estudiante de medio superior o superior; se exponen los objetivos y limitaciones del trabajo a desarrollar; se compara la propuesta de solución con herramientas similares; se muestran los diferentes conceptos con los que el lector debe estar relacionado para comprender la solución que se propone; se hace un análisis de la metodología de diseño a usar; se estudia si es o no factible la propuesta de solución con los recursos disponibles por el equipo de desarrollo; se exponen los principales algoritmos a implementar; se muestran los diagramas usados para diseñar el sistema; se presenta la arquitectura a seguir en la propuesta de solución; finalmente, en el apéndice está un breve resumen sobre la notación UML en diagramas de clases por si el lector no está familiarizado con esta notación gráfica y una breve descripción del prototipo funcional de la propuesta de solución.

keywords— IPN, ESCOM, modelo conceptual, NoSQL, entidad-relación, modelo relacional, validación estructural, GDM, Generic Data Metamodel, orientado a documentos, bases de datos, Scrum, modelo lógico, modelo físico, SQL, MongoDB, Nuxt, Python.

Capítulo 1

Introducción

En los últimos años, los sistemas *Not only SQL* o NoSQL han surgido como alternativa a los sistemas de bases de datos relacionales y se enfocan, principalmente, en almacenar datos que probablemente se consultarán juntos.

Los sistemas NoSQL admiten diferentes modelos de datos como los de clave-valor, documentos, columnas y grafos; con ello se enfocan en guardar datos de una manera apropiada de acuerdo con los requisitos actuales para la gestión de datos en la web o en la nube; enfatizando la escalabilidad, la tolerancia a fallos y la disponibilidad a costa de la consistencia.

De acuerdo a Google Trends^[1], en 2009 aumentó el interés en los sistemas NoSQL y, desde entonces, también empezaron a resaltar algunas problemáticas propias de estos sistemas, porque si bien resuelven algunos dilemas de las bases de datos relacionales, por enfocarse en la ya mencionada escalabilidad, la tolerancia a fallos o la flexibilidad para realizar cambios en el esquema de la base de datos, la heterogeneidad de los sistemas NoSQL ha llevado a una amplia diversificación de las interfaces de almacenamiento de datos, provocando la pérdida de un paradigma de modelado común o de un lenguaje de consultas estándar como SQL.

Respecto al modelado de datos, en el diseño de las bases de datos tradicionales, el modelo entidad-relación^[2] es el modelo conceptual más usado y tiene procedimientos bien establecidos como resultado de décadas de investigación; sin embargo, para los sistemas NoSQL los enfoques tradicionales de diseño de bases de datos no proporcionan un soporte adecuado para satisfacer el modelado de sus diferentes modelos de datos y para abordar esta problemática se han creado varias metodologías de diseño para sistemas NoSQL en los últimos años, porque los diseñadores de bases de datos deben tener en cuenta no solo qué datos se almacenarán en la base de datos, sino también cómo se accederán a ellos para modelar estos sistemas ^{[3]-[5]}.

Para tener en cuenta cómo se accederán a los datos se debe conocer cómo se realizarán las consultas de los mismos y de acuerdo al trabajo de Mosquera^[6], que es una investigación de 1376 *papers* sobre el modelado de sistemas NoSQL, se muestra que de las metodologías propuestas la mayoría usa el modelo entidad-relación para modelado conceptual, mientras que otros proponen su propio modelo conceptual y en cada propuesta se presenta una manera distinta de representar las consultas de los datos; en resumen, no hay una tendencia en el modelado de datos NoSQL y en

la literatura sobre el tema solo existen unas cinco herramientas implementadas de las diversas propuestas para el modelado de las bases de datos no relacionales.

Lo que resta del capítulo está organizado de la siguiente manera: primero se muestra la problemática a resolver, después la propuesta de solución, la justificación, los objetivos del proyecto y por último se menciona el alcance con las limitaciones del mismo.

1.1. Descripción del problema

Como se ha visto en la introducción, son pocas las herramientas propuestas en la literatura para el modelado de sistemas NoSQL en sus tres niveles de abstracción (nivel conceptual, lógico y físico).

Solo en el modelado conceptual, el modelo entidad-relación puede considerarse como una tendencia; sin embargo, el modelo entidad-relación por sí mismo no es suficiente para representar cómo se consultarán los datos ni tampoco con qué frecuencia se accederán a ellos; por lo tanto, para modelar sistemas NoSQL es necesario conocer enfoques de desarrollo como el *query-driven design*, el *domain-driven design*, el *data-driven design* o el *workload-driven design*.

En general, para los modelos de datos NoSQL no hay un modelo estándar ni tampoco existe un acuerdo sobre la mejor definición de reglas de transformación entre sus tres niveles de abstracción; por ejemplo, en la literatura sobre el tema hay unos 36 estudios^[6] que proponen diferentes enfoques para las transformaciones.

Por estas dificultades, el modelado de bases no relacionales aún no se ve reflejado en los programas de estudio de nivel medio superior o superior; o solo se da una introducción del tema a los estudiantes de licenciatura en ingeniería en sistemas o carreras afines.

Asimismo, el estudiante de nivel medio superior o superior que lleva la materia de Base de Datos, como sucede en ESCOM, antes de pensar en cómo diseñar una base de datos NoSQL, enfrenta problemas para diseñar una base de datos relacional, porque desde el modelado conceptual tiene que confiar en sus conocimientos recientemente adquiridos para verificar la validez de los diagramas entidad-relación que desarrolla o consultarlos con el maestro de turno.

En consecuencia, para el estudiante de ESCOM es un problema tener que enfrentarse a diseñar un sistema NoSQL, porque no tiene la certeza de desarrollar diagramas entidad-relación válidos, no se le enseña ninguna metodología de desarrollo para sistemas NoSQL, tampoco conoce los distintos modelos de datos NoSQL que hay, mucho menos sus ventajas o desventajas para poder elegir el modelo más apropiado para su aplicación y, como resultado, no visualiza el diseño de una base de datos no relacional a partir de los conocimientos adquiridos en la asignatura de Bases de Datos.

Además, la escasez de herramientas CASE (*Computer Aided Software Engineering*) para el diagramado de esquemas no relacionales o herramientas que apoyen la migración de un modelo entidad-relación o relacional a uno NoSQL aumenta la complejidad para que el estudiante haga uso de una base de datos no

relacional.

1.2. Propuesta de solución

De acuerdo con la problemática antes mencionada, se propone desarrollar una aplicación web que permita:

- La edición de un diagrama entidad-relación básico (no extendido).
- Realizar la validación estructural del diagrama entidad-relación.
- Transformar el diagrama entidad-relación al modelo relacional.
- Obtener la definición del esquema relacional de la base de datos.
- Obtener entidades del modelo conceptual de un modelo de datos NoSQL a partir de un diagrama entidad-relación básico.
- Obtener el modelo lógico de un modelo de datos NoSQL desde su modelo conceptual por medio de consultas a las entidades del modelo conceptual NoSQL.
- Obtener la definición del esquema NoSQL de la base de datos para probarlo en un NoSQL DBMS (*NoSQL Database Management System*).

1.3. Justificación

Actualmente, las pocas herramientas para modelado conceptual NoSQL no ofrecen validaciones para sus diagramas ni dan la posibilidad de obtener el esquema SQL, ya que no están enfocados en sistemas relacionales, sino en un modelo lógico y físico específico para cada tipo de sistema NoSQL (clave-valor, orientado a documentos, columnas o grafos).

Asimismo, como se mencionó en la introducción, por los pocos años que han pasado desde que el interés y el uso de los sistemas NoSQL aumentó, estos sistemas todavía no están reflejados en los programas de estudio de nivel superior, como es el caso de ESCOM.

En consecuencia, para abordar esta problemática en la que el estudiante diseña un sistema NoSQL sin haber aprendido ninguna metodología de desarrollo para dichos sistemas y sin que logre visualizar en primera instancia el diseño de una base de datos no relacional a partir de lo que ve en la asignatura de Bases de Datos, se propone una aplicación web que brinde la funcionalidad de una herramienta CASE que apoye al estudiante a diseñar una base de datos NoSQL a partir de los conocimientos adquiridos del estudiante.

Se usará como modelo conceptual el modelo entidad-relación básico por ser un modelo bien conocido para el estudiante de licenciatura en sistemas, que está probado en el área de las bases de datos y de acuerdo con Mosquera^[6], es también el más usado en las investigaciones sobre el modelado conceptual de sistemas NoSQL.

1.4. Objetivos

Objetivo general

Desarrollar una aplicación web que permita la edición de un diagrama de bases de datos bajo el modelo entidad-relación básico y realice la validación del mismo; el diagrama se transformará al modelo relacional para obtener el esquema de la base de datos en sentencias SQL, o bien, obtendrá el modelo lógico de un modelo de datos no relacional y podrá obtener las sentencias para generar el modelo no relacional en un NoSQL DBMS (*NoSQL Database Management System*).

Objetivos específicos

- La edición del diagrama entidad-relación básico se implementará con alguna biblioteca para diagramado, en donde al usuario se le permitirá arrastrar los entidades, atributos y relaciones del diagrama entidad-relación básico desde una “paleta” a la zona para diagramar, conocida también como *canvas*.
- La validación del diagrama entidad-relación básico se realizará con eventos de escucha en el *canvas* o con un botón para validar y será de acuerdo a reglas de validación que se expondrán más adelante.
- La transformación del diagrama entidad-relación básico al modelo relacional se realizará con alguno de los algoritmos conocidos en la literatura del tema para generar la definición del esquema de sentencias SQL desde el modelo relacional.
- Se realizará la transformación del modelo entidad-relación básico a entidades del modelo conceptual NoSQL.
- Por medio de preguntas definidas por el usuario para las entidades del modelo conceptual NoSQL se generará el modelo lógico NoSQL.
- Se mostrará el modelo lógico NoSQL usando alguna biblioteca para diagramado donde no se le permitirá al usuario editar ningún elemento del mismo.
- Se generará un *script* con instrucciones para generar el esquema de una base de datos NoSQL desde el modelo lógico elegido y se probará en un único NoSQL DBMS.

1.5. Alcance y limitaciones

Generales

1. Se usará la notación oficial en los modelos de datos conceptuales entidad-relación y relacional, es decir, la notación de Chen y la notación de Codd, respectivamente.

-
2. Respecto al registro de los usuarios, se podrá acceder a la aplicación con un correo válido o no válido, teniendo en cuenta de que si no crea su cuenta con un correo válido, no le llegará notificación de correo que le indica que se ha registrado exitosamente junto con su contraseña.
 3. La propuesta de solución se podrá ejecutar en el navegador web Google Chrome versión 81 en adelante.
 4. Solo se podrá guardar un único diagrama entidad-relación en la aplicación.
 5. No se guardarán las preguntas definidas por el usuario del modelo conceptual NoSQL.

Diagramado entidad-relación

1. El diagramador entidad-relación básico solo podrá editar elementos del modelo entidad-relación, no se podrá diagramar elementos del modelo entidad-relación extendido ni de ningún otro modelo.
2. El grado de partición en el modelo entidad-relación será máximo de dos entidades.
3. No se podrá diagramar atributos parciales.
4. No se podrá diagramar atributos compuestos.

Validación diagrama entidad-relación básico

1. De acuerdo con Dullea[7], un diagrama entidad-relación básico es válido solo si es estructural y semánticamente válido; sin embargo, hasta donde se sabe, en los últimos 17 años no hay estudios sobre la validez semántica de un diagrama entidad-relación básico, porque como expresa Dullea en su trabajo, la validez semántica depende del minimundo que se quiere representar en el diagrama y es imposible definir una métrica generalizada; por ello la validez de un diagrama entidad-relación básico será solo estructuralmente.
2. No se hará la validación de atributos compuestos.

Transformación modelo entidad-relación básico a modelo relacional

1. Para la transformación del modelo entidad-relación básico al modelo relacional no se realizará ninguna normalización, porque está fuera del alcance para el equipo programar la lógica necesaria y se necesita un diagrama no normalizado para la transformación al modelo conceptual NoSQL.
2. No se generará la transformación de atributos compuestos.
3. No se mostrará el diagrama generado del modelo relacional, porque solo se crea para generar las sentencias SQL para ser probadas en MySQL.

Obtención esquema de la base de datos desde el modelo relacional

1. Para la generación de *scripts* del esquema de datos relacional se usará el lenguaje de consultas SQL.
2. Al realizar ingeniería inversa del esquema generado en MySQL Workbench, pueden no mostrarse las relaciones entre las tablas graficamente (esto es un tema completemateamente del propio gestor MySQL).
3. Asimismo, los *scripts* generados se probarán con MySQL porque es uno de los DBMS que se usa en la asignatura de Base de Datos y están familiarizados los estudiantes de ESCOM.

Transformación modelo entidad-relación básico a modelo conceptual NoSQL

1. Dado que hay varias propuestas en la literatura sobre modelos conceptuales NoSQL, se optará por usar la propuesta de Alfonso de la Vega [8], ya que el modelo que propone, conocido como *Generic Data Metamodel*, es un metamodelo conceptual que describe un modelo de datos NoSQL independientemente de si es de clave-valor, orientado a documentos, a columnas o grafos.
2. No se implementará la opción de marcar una entidad en el *Generic Data Metamodel* como *muy usada*.
3. Se hará uso de las definiciones de lenguaje de Alfonso de la Vega [8], porque se usará su definición de modelo conceptual y modelo lógico NoSQL.

Obtención modelo lógico modelo NoSQL desde el modelo conceptual NoSQL

1. Se usará la propuesta de Alfonso de la Vega [8] para el modelo lógico orientado a documentos.
2. Si el modelo conceptual NoSQL no contiene consultas, no se podrá generar el modelo lógico NoSQL, porque de acuerdo a Mosquera [6], no es posible obtener un modelo lógico NoSQL sin consultas.

Obtención de setencias de MongoDB

1. Los *scripts* NoSQL generados por la aplicación solo serán probados y ejecutados en un único NoSQL DBMS.
2. Se eligirá el modelo de datos orientado a documentos y se usará MongoDB para probar los *scripts* NoSQL generados por la aplicación, porque de acuerdo a Mosquera [6], las de bases de datos orientadas a documentos son las más

estudiadas y MongoDB es el NoSQL DBMS más probado para modelado físico en la literatura del tema.

Capítulo 2

Estado del arte

De acuerdo con varios autores [4], [5], [9], un modelo conceptual que especifique solo qué entidades conforman el sistema no es suficiente, porque para los sistemas NoSQL es clave conocer cómo las entidades desde el modelo conceptual serán consultadas.

Debido a lo anterior, estos autores complementan los modelos conceptuales tradicionales como el modelo entidad-relación con propuestas sobre cómo conocer los patrones de acceso; sin embargo, ¿qué herramientas usan estas propuestas?

Para responder esta pregunta se presenta en este capítulo una investigación de herramientas similares, empezando por dos inspiradas en el modelo conceptual propuesto por Chebotko en 2015.

Además, se muestran otras tres herramientas, todas basadas en diferentes propuestas sobre modelado conceptual, dando una introducción a Mortadelo, la herramienta que hace uso del modelo conceptual elegido en el proyecto para representar una base de datos NoSQL.

Lo que resta de este capítulo está organizado de la siguiente manera: se presenta la descripción de cada oferta y se finaliza en las conclusiones con una tabla comparativa.

2.1. Kashliev Data Modeler

De acuerdo con el sitio web de KDM (Kashliev Data Modeler)[10], esta es una herramienta de modelado desarrollada el profesor asistente del Departamento de informática de la Universidad del Oeste Michigan, Andrii Kashliev, que automatiza el diseño de esquemas para una base de datos NoSQL orientada a columnas en Apache Cassandra.

Por medio de una aplicación web, KDM ayuda al usuario comenzando con un modelo conceptual ER más consultas asociadas para generar un modelo lógico y físico de datos o *scripts* CQL (Cassandra Query Language).

KDM automatiza:

1. El mapeo conceptual a lógico.

-
2. El mapeo lógico a físico.
 3. La generación de *scripts* CQL.

El modelo conceptual usado en KDM es el propuesto por Chebotko[4] y automatiza el proceso de transformación entre los tres niveles de modelos (conceptual, lógico y físico).

Además, cuenta con una versión de prueba de tiempo indefinido con características limitadas que permite la generación de un modelo lógico y guardar los proyectos.

2.2. NoSQL Workbench for Amazon DynamoDB

De acuerdo con sitio web de Amazon[11], NoSQL Workbench for Amazon DynamoDB es una herramienta desarrollada por Amazon que proporciona funciones de desarrollo de consultas, modelado y visualización de datos para diseñar, crear, consultar y administrar bases de datos NoSQL orientadas a columnas.

NoSQL Workbench for Amazon DynamoDB usa el modelo conceptual propuesto por Chebotko[4].

El visualizador de modelo de datos proporciona un lienzo donde se asignan consultas y visualizan las facetas (parte de la base de datos) de la aplicación sin tener que escribir código.

Cada faceta corresponde a un patrón de acceso diferente en DynamoDB, donde cada patrón se agrega manualmente; cuenta con un generador de operaciones para ver, explorar y consultar conjuntos de datos.

Por último, admite la proyección, la declaración de expresiones condicionales y permite generar código de muestra en varios idiomas.

2.3. Hackolade

De acuerdo con sitio web de la herramienta[12], Hackolade es un *software* con capacidad de representar objetos JSON anidados; la aplicación combina la representación gráfica de colecciones (término usado en las bases de datos orientados a documentos) y vistas en un diagrama.

La aplicación está basada en la denormalización, el polimorfismo y las matrices anidadas JSON; la representación gráfica de la definición del esquema JSON de cada colección está en una vista de árbol jerárquica.

Lamentablemente, en el sitio web de Hackolade no hay información sobre el modelo conceptual en el que está basado su aplicación.

Es una herramienta de modelado de datos para MongoDB, Neo4j, Cassandra, Couchbase, Cosmos DB, DynamoDB, Elasticsearch, HBase, Hive, Google BigQuery, Firebase/Firestore, MarkLogic, entre otros.

2.4. Mortadelo

De acuerdo con la Vega[8], Mortadelo está basado en el *model driven*; es decir que para diseñar bases de datos NoSQL necesita de modelos conceptuales definidos.

Lo anterior implica usar dos modelos diferentes pero interrelacionados que añaden complejidad al modelado de la base datos; por esta razón de la Vega propone el GDM (*Generic Data Metamodel*), que es un modelo conceptual NoSQL donde la estructura (entidades, atributos, relaciones entre entidades) y patrones de acceso (cómo se consultarán los datos) están integradas en un mismo modelo conceptual.

Cabe destacar que el GDM es un modelo conceptual independiente del paradigma NoSQL, por lo que puede representar una base de datos NoSQL de clave-valor, orientado a documentos, orientado a columnas u orientado a grafos.

La figura 2.1 muestra los tres pasos principales de Mortadelo, empezando desde la izquierda con su propuesta de modelo conceptual NoSQL, el *Generic Data Metamodel*, siguiendo con una representación de los modelos lógicos orientado a columnas o documentos en los que realiza “modelo a modelo” (M2M) para generar el modelo lógico, para finalizar una transformación “modelo a texto” (M2T) con el modelo físico de cada modelo lógico, respectivamente.

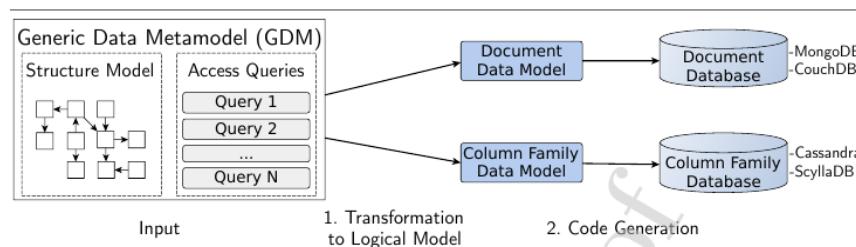


Figura 2.1: Mortadelo

Mortadelo: modelo conceptual (*Generic Data Metamodel*)

La figura 2.2 muestra el *Generic Data Metamodel*, que está compuesto por clases interrelacionadas entre sí con notación UML¹ y consta de dos secciones principales: los elementos de la estructura (*structure model elements*) y el cómo se realizarán las consultas (*access queries elements*).

¹La simbología usada en los diagramas de clase UML está en el apéndice.

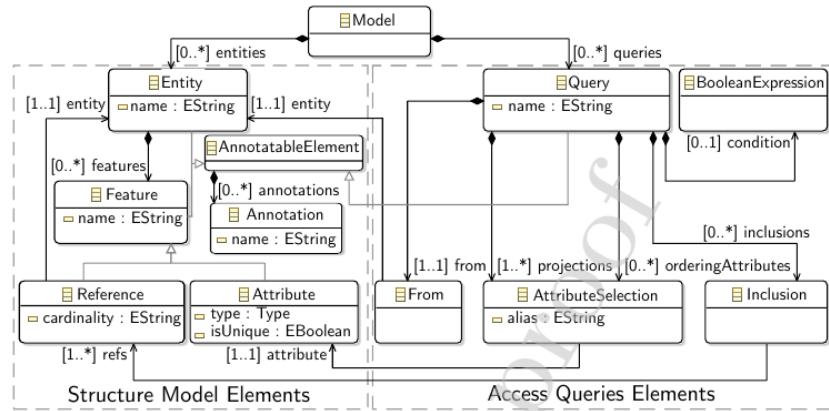


Figura 2.2: Generic Data Metamodel

A continuación se da una explicación de cada elemento de la figura 2.2 donde la clase *model* es para indicar que un modelo GDM tiene n entidades y n consultas donde $n = 0, 1, \dots, n$.

Structure model elements

- Clase *entity*: contiene *features* y solo es referenciada directamente desde las clases *from* y *reference*.
- Clase *feature*: es una clase abstracta, una *reference*, un *attribute*, o hereda de la clase *annotatable element* para que la instancia de la clase sea comentada con indicadores de texto que proveen información extra para generar el modelo lógico.
- Clase *reference*: empieza con la palabra clave “ref”, un nombre de tipo de entidad, una cardinalidad y un nombre de referencia; por ejemplo: “ref Category[*] categories” define una referencia llamada *categories*, del tipo de entidad *category* con una cardinalidad de cero a varios.
- Clase *attribute*: contiene un tipo y un identificador de unicidad.
- Clase *annotatable element*: es una clase abstracta para permitir que una clase contenga anotaciones.
- Clase *annotation*: es un indicador de texto que provee información extra para generar el modelo lógico.

Access queries elements

- Clase *query*: tiene solo un elemento de la clase *from*, n elementos de la clase *attribute selection*, n elementos de la clase *inclusion* y tiene o no un único elemento de la clase *boolean expression*.
- Clase *from*: asocia una clase *query* con la clase *entity*; es la clase que permite referenciar un tipo de entidad.

- Clase *attribute selection*: accede a los atributos de la *entity* referenciada por la clase *from* o la clase *inclusion*.
- Clase *boolean expression*: expresa una expresión booleana para declarar alguna restricción.
- Clase *inclusion*: permite acceder en una *query* a los atributos de otros tipos de entidad.

La figura 2.3 es una instancia del modelo conceptual GDM en su notación textual en la que se nota que, por ejemplo, en la tercera consulta se accede a los elementos de la entidad *category* a través del elemento *ref* de la entidad *product*.

```
// Entities
entity Product {
    id productId
    text name
    text description
    number price
    ref Category[*]
        categories
    ref Provider[1]
        provider
}
entity Category {
    id categoryId
    text name
    text description
}
entity Provider {
    id providerId
    text name
}

// Queries
query Q1_productsById:
    select prod.productId,
        prod.name, prod.price,
        prod.description
    from Product as prod
    where prod.productId = "?"

query Q2_productsAndCategoryByName:
    select prod.name, prod.price,
        prod.description,
        cat.name
    from Product as prod
    including prod.categories as cat
    where prod.name = ?"

query Q3_productsByCategory:
    select prod.name, prod.price,
        prod.description,
        cat.name
    from Product as prod
    including prod.categories as cat
    where cat.name = ?"
    order by prod.price
```

Figura 2.3: Notación textual del GDM

Para más detalles sobre su modelo lógico orientado a documentos y algoritmos asociados, visite la sección 4.3.3.5.

2.5. NoSE: Schema Design for NoSQL Applications

De acuerdo con Michael J. Mior[5], NoSE usa un modelo conceptual junto con el *workload* para describir cómo se accederán a los datos y así genera un modelo físico de una base de datos NoSQL orientado a columnas.

NoSE debe tener un modelo conceptual que describa la información que se almacenará, por ello NoSE espera este modelo conceptual en forma de un grafo de entidad.

Los grafos de entidad son un tipo restringido del modelo de entidad-relación; cada cuadro representa un tipo de entidad, tienen atributos en los que uno más

sirven como clave para identificarla, cada borde es una relación entre entidades y la cardinalidad asociada de la relación (uno a muchos, uno a uno o muchos a muchos).

Respecto al *workload*, se describe como un conjunto de consultas parametrizadas y declaraciones de “actualización”; cada consulta y actualización está asociada con un peso que indica su frecuencia relativa en la carga de trabajo.

2.6. Conclusiones

Como se ha podido ver en este capítulo, son pocas las herramientas para modelado conceptual NoSQL; de ellas, dos usan la propuesta de Chebotko para modelar conceptualmente bases de datos orientadas a columnas y las demás herramientas modelan bases de datos NoSQL con base en diferentes propuestas.

Asimismo, es de notar que ninguna herramienta permite la validación estructural del modelo conceptual que usan, ni tampoco generan el esquema del modelo relacional, porque no están enfocadas a modelar bases de datos relacionales y por la misma razón no producen los *scripts* de sentencias SQL del esquema relacional.

Por ello se pretende desarrollar una aplicación web que brinde la funcionalidad de una herramienta CASE que realice los puntos antes mencionados.

Para finalizar este capítulo, la tabla 2.1 muestra una comparación de las diferentes características de cada herramienta expuesta.

Herramienta	Creador	Objetivo	Licencia	Plataforma	Fecha de publicación	Modelo			Patrones de Acceso	Metodología	Validación Estructural	Modelado Relacional	Esquema SQL
						Conceptual	Lógico	Físico					
KDM	Andrii Kashlev	comercial	privativa	Web	2015	entidad-relación	columnas	Cassandra	querys	query-driven	N/A	N/A	N/A
HacKolade	IntegrIT SA / NV	comercial	privativa	Windows, Mac, Linux	2016	entidad-relación	multiparadigma	multiparadigma	sin especificar	sin especificar	N/A	N/A	N/A
NoSQL Workbench for Amazon DynamoDB	Amazon	comercial	privativa	Windows, Mac	2019	entidad-relación	columnas	MongoDB	querys	query-driven	N/A	N/A	N/A
Mortadelo	Alfonso de la Vega	investigación	libre	Web	2018	GDM	columnas documentos	Cassandra MongoDB	querys	query-driven	N/A	N/A	N/A
NoSE	Michael J. Mior	investigación	libre	Web	2016	grafos de entidades	documentos	MongoDB	workload	workload-driven	N/A	N/A	N/A
Propuesta de solución	TT 2019-B052	investigación	libre	Web	2020	entidad-relación GDM	orientado a documentos	MongoDB	querys	query-driven	Sí	Sí	Sí

Tabla 2.1: Tabla comparativa de las herramientas estudiadas y la propuesta de solución.

Capítulo 3

Marco teórico

Del estado del arte se ha concluido que el modelo entidad-relación es el modelo conceptual más usado para describir modelos de datos NoSQL; también es de notar que ninguna herramienta de las estudiadas ofrece validación estructural de su modelo conceptual, obtención del esquema relacional o de las sentencias SQL.

En este apartado se muestran los diferentes conceptos con los que el lector debe estar relacionado para comprender la solución que se propone y el resto del capítulo está organizado de la siguiente manera: primero se muestran los modelos de datos para bases de datos, empezando con el modelo entidad-relación, donde está cómo interactúan entre sí sus elementos (entidades, atributos y relaciones) para representar una base de datos; después se expone el modelo relacional y una descripción de SQL (su lenguaje de consultas); también se muestra de forma concisa y breve los cuatro modelos de datos NoSQL (clave-valor, orientado a columnas, orientado a documentos y orientado a grafos); se explica cada tecnología usada en la propuesta de solución y el porqué ha sido elegida; finalmente, se termina con las conclusiones del capítulo.

3.1. Modelos de datos para bases de datos

De acuerdo con Elmasri [13], un modelo de datos es una colección de conceptos que describen la estructura de una base de datos.

Los modelos de datos conceptuales o de alto nivel ofrecen conceptos visuales simples que representan un modelo de datos, mientras que los modelos de datos físicos o de bajo nivel son detalles de cómo se implementa el almacenamiento de los datos en el sistema de la base de datos.

Una entidad representa un objeto o concepto del mundo real; un atributo representa alguna propiedad que describe una entidad y una relación es una asociación entre entidades.

A continuación se presentan los modelos de datos que se utilizarán en el desarrollo de este trabajo.

3.1.1. Modelo entidad-relación

De acuerdo con Elmasri [13], el modelo entidad-relación (o modelo ER) fue creado por Peter Chen en 1976[14] para el diseño conceptual de bases de datos y está conformado de entidades, atributos y relaciones.

3.1.1.1. Entidades

El objeto básico representado por el modelo ER es una entidad, que es una cosa del mundo real con una existencia independiente; una entidad es un objeto con existencia física o conceptual.

Clasificación de tipos de entidad Un tipo de entidad define un conjunto de entidades con los mismos atributos; un tipo de entidad sin atributo clave se denomina tipo de entidad débil; en contraposición, un tipo de entidad regular con atributo clave se denomina tipo de entidad fuerte.

Las entidades que pertenecen a un tipo de entidad débil se identifican como relacionadas con un tipo de entidad propietaria en combinación con uno de sus valores de atributo.

El tipo de relación que relaciona un tipo de entidad débil con su propietaria se llama relación identificativa del tipo de entidad débil.

Un tipo de entidad débil siempre tiene una restricción de participación total (dependencia de existencia) respecto a su relación identificativa, porque una entidad débil no se identifica sin una entidad propietaria; no obstante, no toda dependencia de existencia produce un tipo de entidad débil.

Un tipo de entidad débil posee una clave parcial, que es el conjunto de atributos que identifican inequívocamente las entidades débiles que están relacionadas con la misma entidad propietaria.

En los diagramas ER tanto el tipo de la entidad débil como la relación identificativa se distinguen rodeando sus rectángulos y rombos mediante líneas dobles.

El atributo de clave parcial aparece subrayado con una línea discontinua o punteada; los tipos de entidades débiles se representan a veces como atributos complejos (compuestos o multivalor).

En general, se define cualquier cantidad de niveles de tipos de entidad débil; un tipo de entidad propietaria puede ser un tipo de entidad débil.

Además, es posible que un tipo de entidad débil tenga más de un tipo de entidad identificativa y un tipo de relación identificativa de grado superior a dos.

3.1.1.2. Atributos

Cada tipo de entidad o tipo de relación posee atributos, que son propiedades particulares que la describen y en el modelo ER hay varios tipos: simple, compuesto,

monovalor, multivalor, almacenado, derivado y nulo.

Atributos compuestos vs atributos simples Los atributos compuestos se dividen en subpartes más pequeñas que representan atributos más básicos con significados independientes.

Los atributos que no son divisibles se denominan atributos simples o atómicos, mientras que los atributos compuestos forman una jerarquía.

El valor de un atributo compuesto es la concatenación de los valores de sus atributos simples.

Atributos monovalor vs multivalor La mayoría de los atributos poseen un solo valor para una entidad en particular; dichos atributos reciben el nombre de monovalor o de un solo valor.

En algunos casos, un atributo es un conjunto de valores para la misma entidad y se denominan multivalor.

Un atributo multivalor tiene un límite superior y uno inferior para restringir el número de valores permitidos para cada entidad individual.

Atributos almacenados vs derivados El atributo derivado se calcula u obtiene a partir de un atributo almacenado.

Atributos complejos Los atributos complejos son atributos compuestos y multivalor que se anidan arbitrariamente.

Atributos clave de un tipo de entidad Un atributo clave identifica inequívocamente cada tipo de entidad, cuenta con un nombre subrayado dentro del óvalo y algunos tipos de entidad poseen más de un atributo clave y si carece de clave, se le denomina tipo de entidad débil.

Atributos de los tipos de relación Los atributos de los tipos de relación 1:1 o 1:N se trasladan a uno de los tipos de entidad participantes.

En el caso de un tipo de relación 1:N, un atributo de relación solo se migra al tipo de entidad que se encuentra en el lado N de la relación.

Para los tipos de relación M:N, algunos atributos se determinan mediante la combinación de entidades participantes en una instancia de relación, no mediante una sola relación; dichos atributos deben especificarse como atributos de relación.

3.1.1.3. Relaciones

Un tipo de relación R entre n tipos de entidades E_1, E_2, \dots, E_n define un conjunto de relaciones entre las entidades de esos tipos de entidades.

Como en el caso de los tipos de entidades y los conjuntos de entidades, normalmente se hace referencia a un tipo de relación y su correspondiente conjunto de relaciones con el mismo nombre R .

Matemáticamente, el conjunto de relaciones R es un conjunto de instancias de relación r_i , donde cada r_i asocia n entidades individuales (e_1, e_2, \dots, e_n) y cada entidad e_j de r_i es un miembro del tipo de entidad E_j , $1 \leq j \leq n$. Por tanto, un tipo de relación es una relación matemática en E_1, E_2, \dots, E_n .

En los diagramas ER, los tipos de relación se representan gráficamente mediante rombos, conectados a su vez mediante líneas a los rectángulos que representan los tipos de entidad participantes y el nombre de la relación se muestra dentro del rombo.

Grado de un tipo de relación El grado de un tipo de relación es el número de tipos de entidades participantes; un tipo de relación de grado dos se denomina binario, de grado tres ternario y de grado n n-ario.

Nombres de rol y relaciones recursivas Cada tipo de entidad que participa en un tipo de relación juega un papel o rol particular en la relación.

El nombre de rol hace referencia al papel que una entidad participante del tipo de entidad juega en cada instancia de relación y ayuda a explicar el significado de la relación.

Los nombres de rol no son técnicamente necesarios en los tipos de relación donde todos los tipos de entidad participantes son distintos, puesto que cada nombre de tipo de entidad participante se utiliza como participación.

Cuando un tipo de entidad se relaciona consigo misma, se tiene una relación recursiva y es necesario indicar los roles que juegan los miembros en la relación.

Restricciones en los tipos de relación Los tipos de relación normalmente tienen ciertas restricciones que limitan las posibles combinaciones entre las entidades que participan en el conjunto de relaciones correspondiente y están determinadas por la situación del minimundo representado; se distinguen dos tipos principales de restricciones de relación: razón de cardinalidad y participación.

Razones de cardinalidad para las relaciones binarias La razón de cardinalidad de una relación binaria especifica el número máximo de instancias de relación en las que una entidad participa.

Las posibles razones de cardinalidad para los tipos de relación binaria son 1:1, 1:N, N:1 y M:N.

1. Uno a uno: una relación R de X a Y es uno a uno si cada entidad en X se asocia con cuando mucho una entidad en Y e, inversamente, cada entidad en Y se asocia con cuando mucho una entidad en X .

-
- 2. Uno a muchos: una relación R de X a Y es uno a muchos si cada entidad en X se asocia con muchas entidades en Y , pero cada entidad en Y se asocia con cuando mucho una entidad en X .
 - 3. Muchos a uno: una relación R de X a Y es muchos a uno si cada entidad en X se asocia con cuando mucho una entidad en Y , pero cada entidad en Y se asocia con muchas entidades en X .
 - 4. Muchos a muchos: una relación R de X a Y es muchos a muchos si cada entidad en X se asocia con muchas entidades en Y y cada entidad en Y se asocia con muchas entidades en X .

Restricciones de participación y dependencias de existencia Hay dos tipos de restricciones de participación, total y parcial; la restricción de participación determina si la existencia de una entidad depende de su relación con otra entidad y especifica el número mínimo de instancias de relación en las que participa cada entidad.

- 1. Participación total: si todo miembro de un conjunto de entidades debe participar en una relación, es una participación total del conjunto de entidades en la relación. Esto se denota al dibujar una línea doble desde el rectángulo de entidades hasta el rombo de relación.
- 2. Participación parcial: una línea sencilla indica que algunos miembros del conjunto de entidades no deben participar en la relación.

La razón de cardinalidad y las restricciones de participación, en conjunto, son restricciones estructurales de un tipo de relación.

3.1.1.4. Resumen de la notación para los diagramas ER

Finalmente, la tabla 3.1 es un resumen de la simbología usada en los diagramas entidad-relación en la que se muestra en la primera columna la representación gráfica y a su lado su significado en el modelo.

Notación del modelo entidad-relación	
Símbolo	Significado
	Entidad
	Entidad débil
	Relación
	Relación de identificación
	Atributo
	Atributo clave
	Atributo multivalor
	Atributo compuesto
	Atributo derivado
	Participación total
	Razón de cardinalidad
	Restricción estructural

Tabla 3.1: Notación del modelo entidad-relación

3.1.2. Modelo relacional

De acuerdo con Elmasri[13], el modelo relacional introducido por Ted Codd en 1970[2] utiliza el concepto de una relación matemática como bloque de construcción básico y tiene su base teórica en la teoría de conjuntos y la lógica del predicado.

La lógica de predicado, utilizada ampliamente en matemáticas, proporciona un marco en el que una afirmación (declaración de hecho) se verifica como verdadera o falsa.

La teoría de conjuntos es una ciencia matemática que trata con conjuntos o grupos de cosas y se utiliza como base para la manipulación de datos en el modelo relacional.

La figura 3.1 muestra de manera visual el modelo relacional; una tabla en el modelo relacional está compuesto de atributos, tiene un nombre de relación y tiene n tuplas.

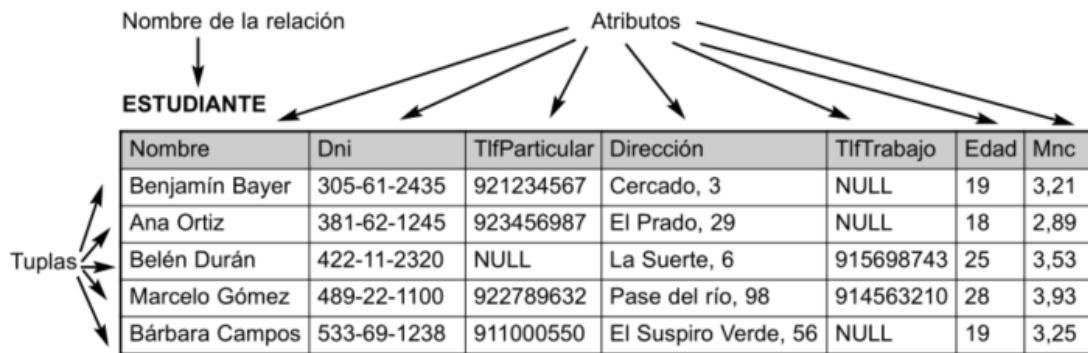


Figura 3.1: Tabla en el modelo relacional

Como se observa en la imagen 3.1, el modelo relacional representa la base de datos como una colección de relaciones, siendo cada relación una tabla de valores, donde cada fila representa una colección de valores relacionados.

Asimismo, cada fila de la tabla representa un hecho que, por lo general, corresponde con una relación o entidad real; el nombre de la tabla y de las columnas se utiliza para ayudar a interpretar el significado de cada uno de los valores de las filas.

En terminología formal, una fila recibe el nombre de tupla, una cabecera de columna es un atributo y el nombre de la tabla una relación; el tipo de dato que describe los valores en cada columna está representado por un dominio de posibles valores.

Basado en estos conceptos, el modelo relacional tiene tres componentes bien definidos:

1. Una estructura de datos lógica representada por relaciones.
2. Un conjunto de reglas de integridad para garantizar que los datos sean consistentes.
3. Un conjunto de operaciones que define cómo se manipulan los datos.

3.1.2.1. Relaciones

En este modelo, las tablas se usan para contener información acerca de los objetos a representar en la base de datos.

Una relación se representa como una tabla bidimensional en la que las filas de la tabla corresponden a registros individuales y las columnas corresponden a atributos.

Formalmente, un esquema de relación R , denotado por $R(A_1, A_2, \dots, A_n)$ está constituido por un nombre de relación R y una lista de atributos A_1, A_2, \dots, A_n .

Un esquema de relación se utiliza para describir una relación; se dice que R es el nombre de la misma y el grado de una relación es su número de atributos n .

En el modelo relacional cada fila se llama tupla y la tabla que representa una relación tiene las siguientes características:

-
- Cada celda de la tabla contiene solo un valor.
 - Cada columna tiene el nombre del atributo que representa.
 - Todos los valores en una columna provienen del mismo dominio, pues todos son valores del atributo correspondiente.
 - Cada tupla o fila es distinta; no hay tuplas duplicadas.
 - El orden de las tuplas o filas es irrelevante.

Relaciones y tablas de bases de datos Una relación r del esquema $R(A_1, A_2, \dots, A_n)$, también especificado como $r(R)$ es un conjunto de n-tuplas $r = t_1, t_2, \dots, t_m$.

Cada tupla t es una lista ordenada de n valores $t = < v_1, v_2, \dots, v_n >$, donde v_i , $1 \leq i \leq n$ es un elemento de $\text{dom}(A_i)$ o un valor especial nulo.

3.1.2.2. Claves

En el modelo relacional, las claves son importantes porque aseguran que cada fila en una tabla sea únicamente identificable; son usadas para establecer relaciones entre tablas y asegurar la integridad de los datos.

Una clave es un atributo o grupo de atributos que identifican los valores de otros atributos y puede ser compuesta, clave o superclave.

Clave compuesta Una clave compuesta es una clave que se compone de más de un atributo y si forma parte de una clave se denomina atributo clave.

Superclave Un atributo o atributos que identifican de manera única cualquier fila de una tabla.

3.1.2.3. Restricciones de integridad

Integridad de dominio La integridad de dominio es la validez de las restricciones que debe cumplir una determinada columna de la tabla.

Integridad de entidad Todas las claves principales son únicas y ninguna clave primaria debe ser nula.

Integridad referencial Una clave externa puede ser nula siempre que no sea parte de la clave principal de su tabla o tiene el valor que coincide con el valor de la clave primaria en una tabla con la que está relacionada.

3.1.2.4. Propiedades de las relaciones

Grado El número de columnas en una tabla se llama grado de la relación, es parte de la intención de la relación y nunca cambia; una relación con una sola columna es de grado uno y se llama relación unaria; con dos columnas se llama binaria; con tres columnas se llama ternaria y con más columnas n-aria.

Cardinalidad La cardinalidad de una relación es el número de entidades a las que otra entidad mapea dicha relación.

3.1.2.5. Structured Query Language

Structured Query Language (o SQL) está basado en el álgebra relacional, en el cálculo relacional y es un lenguaje de manipulación de datos, un lenguaje de definición de datos, un lenguaje de control de transacciones y un lenguaje de control de datos.

Lenguaje de manipulación de datos (DML) Un *Data Manipulation Language* (o DML) incluye comandos para insertar, actualizar, eliminar y recuperar datos dentro de las tablas de la base de datos.

Lenguaje de definición de datos (DDL) Un *Data Definition Language* (o DDL) incluye comandos para crear objetos de base de datos como tablas, índices y vistas, así como comandos para definir accesos a objetos de la base de datos.

Lenguaje de control de transacciones (TCL) Los comandos de un *Transaction Control Language* (o TCL) se ejecutan dentro del contexto de una transacción, que es una unidad lógica de trabajo compuesta por una o más instrucciones SQL y proporciona comandos para controlar el procesamiento de estas transacciones atómicas.

Lenguaje de control de datos (DCL) Los comandos de un *Data Control Language* (o DCL) se utilizan para controlar el acceso a los objetos de datos, como otorgar a un usuario permiso para ver solo una tabla y otorgar a otro usuario permiso para cambiar los datos de la misma tabla.

3.1.3. Modelos NoSQL

De acuerdo con Catherine [15], el término NoSQL significa *Not only SQL* y hay cuatro tipos principales de bases de datos NoSQL: clave-valor, orientado a columnas, orientado a documentos y orientado a grafos.

Se usan para agrupar sistemas de bases de datos diferentes a los relacionales y surgieron por los nuevos requerimientos de disponibilidad total, tolerancia a fallos, almacenamiento de penta *bytes* de información distribuida en miles de servidores, la

necesidad de nodos con escalabilidad horizontal, entre otros; a continuación se una breve explicación de cada modelo de datos NoSQL.

3.1.3.1. Clave-Valor

De acuerdo con Coronel[16], una base de datos de clave-valor es un paradigma de modelo de datos diseñado para almacenar, recuperar y administrar arreglos asociativos.

Comúnmente se usa un diccionario o tabla *hash* que contiene una colección de registros anidados, secuencias de bits que se almacenan y se recuperan utilizando una clave que identifica de manera única el registro y se utiliza para encontrar rápidamente los datos dentro de la base de datos.

No obstante, es responsabilidad de las aplicaciones que hagan uso de este tipo de base de datos interpretar el significado de los datos; no hay claves foráneas y las relaciones no son rastreables entre claves, lo que permite que el DBMS sea rápido y escalable.

La figura 3.2 es una representación visual de un *bucket* usado en las bases de datos NoSQL de clave-valor.

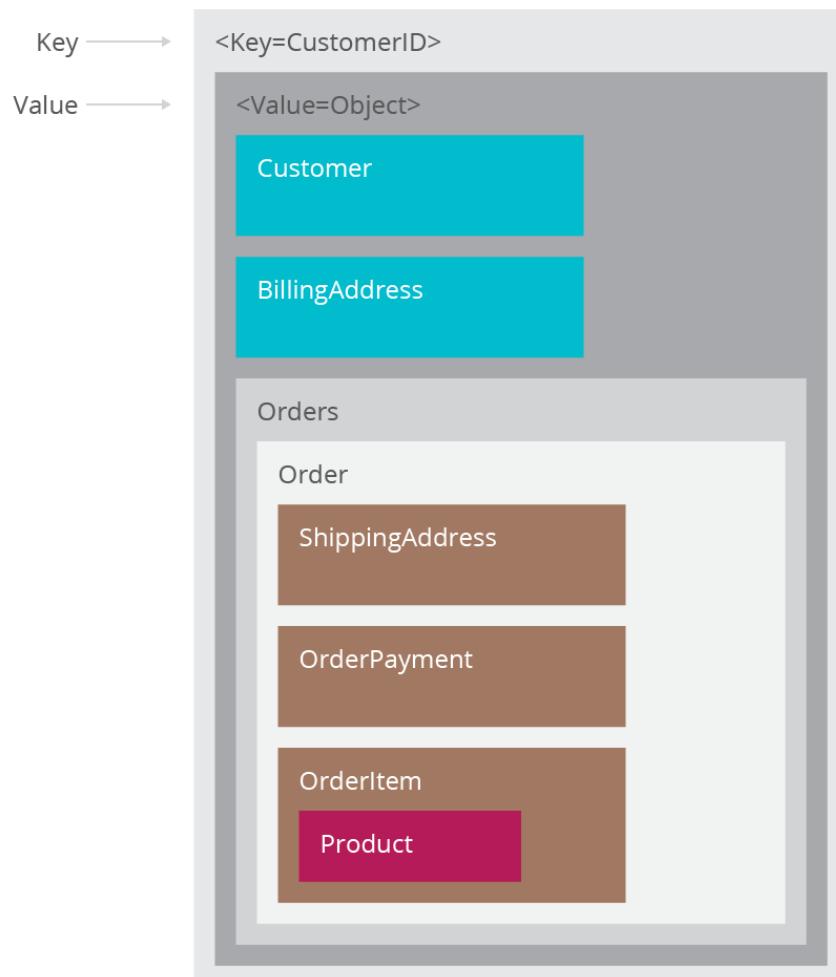


Figura 3.2: *Bucket* en el almacenamiento clave-valor

Los pares de clave-valor generalmente se organizan en *buckets*; todas las claves dentro un *bucket* deben ser únicas, pero está permitido que se repitan en otros *buckets* y todas las operaciones se basan en el *bucket* + la clave.

En este tipo de bases de datos se usan las operaciones de *get*, *store* y *delete*; la operación *get* o *fetch* es usada para obtener el valor de un par; el operador de *store* almacena datos en una clave.

Si la combinación de *bucket* + clave no existe, se añade como un nuevo par de clave-valor; en cambio, si existe la combinación de *bucket* + clave, el valor es reemplazado por el nuevo; el operador de *delete* es para eliminar un par de clave-valor.

De acuerdo con Sadalge[17], algunas de las bases de datos de clave-valor populares son Riak, Redis, Memcached DB, Berkeley DB, HamsterDB, Amazon DynamoDB y Project Voldemort (una implementación de código abierto de Amazon DynamoDB).

3.1.3.2. Orientado a documentos

De acuerdo con Coronel[16], una base de datos NoSQL orientada a documentos almacena datos en documentos etiquetados en pares clave-valor; sin embargo, a diferencia de una base de datos clave-valor donde el componente de valor contiene cualquier tipo de datos, una base de datos de documentos siempre almacena un documento en el componente de valor y puede estar en cualquier formato codificado como XML, JSON o BSON.

La figura 3.3 es la representación visual de una colección, donde tiene una clave como identificador único y documentos anidados.



Figura 3.3: Colección en el almacenamiento de documentos

Otra diferencia importante es que si bien las bases de datos clave-valor no

intentan comprender el contenido del componente de valor, las bases de datos de documentos sí lo hacen; por ejemplo, hay documentos con etiquetas para identificar qué texto en el documento representa el título, el autor y el cuerpo del documento.

Como se ve en la figura 3.3, dentro del cuerpo del documento existen etiquetas adicionales para indicar capítulos y secciones; a pesar del uso de etiquetas en los documentos, las bases de datos de documentos se consideran sin esquema, es decir, no imponen una estructura predefinida en los datos almacenados.

Para una base de datos de documentos, no tener esquemas significa que aunque todos los documentos tienen etiquetas, no todos tienen las mismas etiquetas, por lo es posible que cada documento tenga su propia estructura.

Las etiquetas en una base de datos de documentos son extremadamente importantes porque son la base de la mayoría de las capacidades adicionales que tienen las bases de datos de documentos sobre las bases de datos clave-valor.

Las etiquetas dentro del documento hacen posible consultas complejas para el DBMS; asimismo, al igual que las bases de datos clave-valor agrupan pares clave-valor en grupos lógicos llamados *buckets*, las bases de datos de documentos agrupan documentos en grupos lógicos llamados colecciones.

Si bien es posible recuperar un documento especificando la colección y la clave, también es posible realizar consultas en función del contenido de las etiquetas.

Las bases de datos de documentos tienden a funcionar bajo el supuesto de que un documento es independiente, o sea que no está en diferentes tablas como en una base de datos relacional.

Una base de datos de documentos asume que todos los datos relacionados de una consulta están en un solo documento; por ejemplo, cada consulta en una colección contendría datos sobre el cliente, el pedido en sí y los productos comprados.

Por último, las bases de datos de documentos no almacenan relaciones como se hace en el modelo relacional y generalmente no tienen soporte para operaciones como la unión.

3.1.3.3. Orientado a columnas

De acuerdo con Coronel[16], el modelo de base de datos NoSQL orientado a columnas se originó con el BigTable de Google.

La figura 3.4 representa una familia de columnas; la imagen de arriba es la partición de las diferentes familias de columnas y en la imagen de abajo se nota cada partición individual.

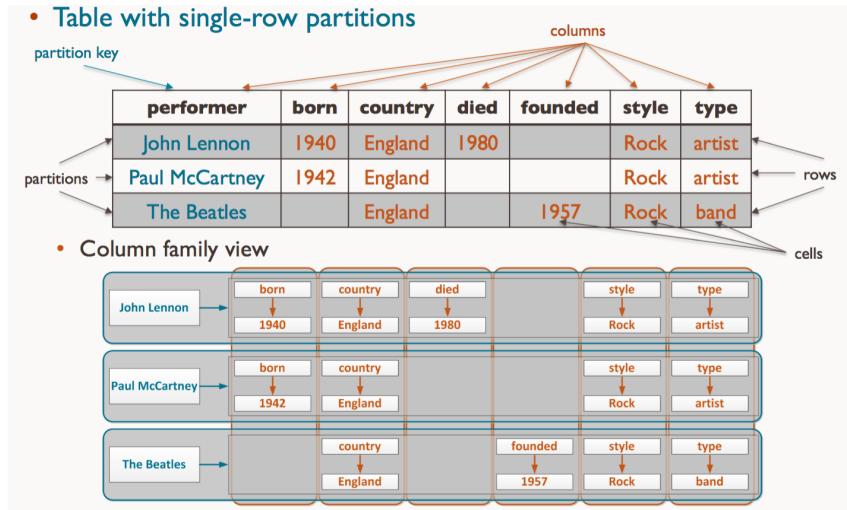


Figura 3.4: Familia de columnas

Las bases de datos de la familia de columnas son parecidas a las relaciones del modelo relacional y organizan datos en pares nombre-valor donde el nombre actúa también como la clave; como se nota en la figura 3.4, un par de clave-valor representa una columna y siempre contiene una fecha que sirve para resolver conflictos de escritura o datos expirados.

3.1.3.4. Orientado a grafos

De acuerdo con Coronel[16], una base de datos NoSQL orientada a grafos está basada en la teoría de grafos para almacenar datos con muchas relaciones.

La figura 3.5 representa un grafo de una biblioteca, donde cada rectángulo es un nodo y están asociados entre sí por relaciones.

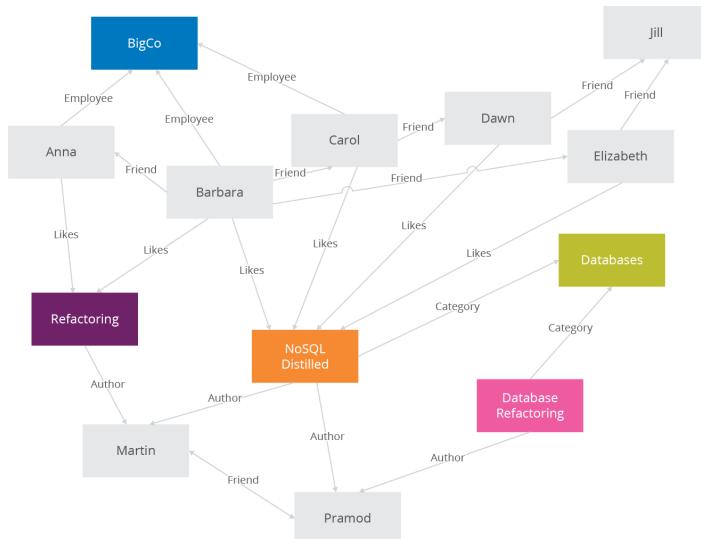


Figura 3.5: modelo conceptual orientado a grafos

Como se muestra en la figura 3.5, los componentes principales de las bases de

datos de grafos son nodos, aristas y propiedades; el nodo es una instancia específica para guardar datos.

Las propiedades son como atributos; son los datos que se necesitan almacenar sobre el nodo; todos los nodos tienen propiedades como nombre y apellido, pero no todos los nodos deben tener las mismas propiedades.

Un borde es una relación entre nodos, está representada por una flecha en la figura 3.5 y es posible que estén en una dirección o ser bidireccionales.

Para hacer una consulta se atraviesa el grafo y los recorridos se enfocan en las relaciones entre nodos, como la ruta más corta y el grado de conexión.

3.2. Ingeniería Dirigida por Modelos

De acuerdo con Scherp [18], la ingeniería dirigida por modelos (en inglés *model-driven software development*) es el desarrollo de *software* mediante modelos y transformaciones entre modelos con el objetivo de automatizar el mapeo entre modelos a código fuente.

Asimismo, de acuerdo con Reussner [19], los conceptos principales en la ingeniería dirigida por modelos son:

1. **Modelo:** es una vista simplificada y abstracta de un sistema real.
2. **Metamodelo:** define elementos y reglas para generar modelos; consiste en una sintaxis abstracta, sintaxis concreta y una semántica.
3. **Transformación entre modelos:** es un mapeo computable que toma como entrada una instancia de un tipo de modelo y como salida genera una instancia de otro tipo de modelo.
4. **Lenguaje específico de dominio:** es un lenguaje definido por un metamodelo que contiene conceptos para un dominio específico.

3.2.1. Lenguaje específico de dominio

De acuerdo con Fowler [20], un lenguaje específico de dominio (en inglés *domain specific language*) es un lenguaje de programación dedicado a un dominio en particular (un dominio puede ser en el contexto de un banco, o de una aplicación web, o de consultas de bases de datos), porque representa un problema específico y provee una técnica para solucionar una situación particular.

Hay dos tipos de lenguajes en un lenguaje DSL:

1. DSL: el lenguaje en el que es escrito un DSL.
2. Lenguaje del host: el lenguaje en el que es ejecutado y procesado el DSL.

Asimismo, un DSL es externo si es un lenguaje diferente al lenguaje del host, pero es interno si el lenguaje DSL es solo un subconjunto de instrucciones del lenguaje del host.

Metodología de diseño

Debido a que un DSL puede ser externo se necesitan de herramientas existentes o desarrollar las herramientas que permitan al lenguaje host interpretar el DSL externo.

De acuerdo con Deursen[21], un DSL se puede diseñar en los siguientes pasos:

1. Identificar el problema de dominio
2. Reunir información relevante sobre el problema de dominio escogido.
3. Crear una gramática que exprese semánticamente el problema de dominio.
4. Desarrollar un compilador que traduzca programas del lenguaje DSL al lenguaje host.

Implementación

Desarrollar un intérprete o un compilador es el enfoque clásico para implementar un nuevo lenguaje, aunque se pueden utilizar herramientas de compilación estándar o herramientas dedicadas a la implementación de DSL como Draco, ASF + SDF o Xtext.

La principal ventaja de desarrollar un compilador o intérprete es que la implementación está completamente adaptada al DSL y no es necesario hacer concesiones en cuanto a notación o tipos de datos. Por otra parte, un problema importante es el costo de desarrollar un compilador o intérprete desde cero, y la falta de reutilización de otras implementaciones[21].

3.2.2. Transformación entre modelos

De acuerdo con Kapferer[22], hay diferentes tipos de transformaciones entre modelos, siendo las más importantes endógena/exógena, *in-place/out-place* y horizontal/vertical.

Endógena vs exógena

Para comparar las transformaciones del modelo, se debe hacer una primera distinción importante entre las transformaciones de un mismo lenguaje y las transformaciones entre diferentes lenguajes; una transformación se llama endógena si un modelo se transforma en otro modelo en el mismo lenguaje o metamodelo. Por otro lado, una transformación se denomina exógena si el modelo de origen y de destino no están representados en el mismo lenguaje.

Usualmente, Las transformaciones endógenas son optimizaciones o refactorizaciones donde se mejoran ciertos atributos de calidad de un modelo manteniendo el lenguaje de representación y la semántica; un ejemplo de transformación exógena podría ser la migración de un programa de un lenguaje a otro.

In-place vs out-place

Esta distinción se refiere únicamente a las transformaciones endógenas. Una transformación endógena se denomina *in-place* si el modelo de origen y el de destino son el mismo, lo que significa que la transformación opera directamente en el modelo de entrada. Si una transformación endógena utiliza un modelo como fuente, pero crea o cambia otro modelo, implicando que hay más de un modelo en juego se denomina *out-place*. Las transformaciones exógenas siempre son *out-place*.

Horizontal vs vertical

Esta distinción entre transformaciones de modelos se refiere al nivel de abstracción. Si el modelo de origen y destino de una transformación se encuentran en el mismo nivel de abstracción, se denomina horizontal, mientras que las transformaciones entre diferentes niveles de abstracción se denominan verticales.

Transformación modelo a modelo y modelo a texto

De acuerdo con Scherp[18], una transformación de modelo se define con un lenguaje de transformación que generalmente proporciona una definición de reglas de transformación.

Una regla de transformación define la asignación de elementos de metamodelo de origen particulares a elementos de metamodelo de destino (transformación M2M) o texto (transformación M2T).

Los lenguajes de transformación se pueden distinguir entre lenguajes de transformación imperativos/operacionales y lenguajes de transformación declarativos/relacionales.

3.3. Tecnologías a usar

Para seleccionar las tecnologías a usar en las propuesta de solución se ha optado por investigar tecnologías similares y esta sección está organizada de la siguiente manera: primero se muestra cada tecnología usar; en caso de que sea la única opción se describirá qué es y en caso de que haya varias opciones, se explicará cada opción y se tendrá un apartado al final de cada comparación sobre la tecnología que se ha elegido.

3.3.1. Hypertext Transfer Protocol

De acuerdo con la W3C[23], Hypertext Transfer Protocol (HTTP, o protocolo de transferencia de hipertexto) es el protocolo de comunicación que permite transferir información en la World Wide Web.

HTTP fue desarrollado por el World Wide Web Consortium y la Internet Engineering Task Force, colaboración que culminó en 1999 con la publicación de

varios RFC, siendo el más importante el RFC 2616 que especifica la versión 1.1 del protocolo.

HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores; sin embargo, el desarrollo de aplicaciones web necesita frecuentemente mantener un estado, por lo que se usan *cookies*, que son archivos generados en un servidor que son almacenados en el sistema cliente.

Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor; al cliente se le suele llamar “agente de usuario” (o *user agent*), que realiza una petición enviando un mensaje con cierto formato al servidor, mientras que al servidor se le suele llamar servidor web y envía un mensaje de respuesta.

HTTP tiene métodos de petición flexibles que permiten añadir nuevos métodos o funcionalidades; el número de métodos ha ido en aumento según se avanza en las versiones del protocolo donde los más importantes son:

1. Método *get*: solicita una representación del recurso especificado; solo deben recuperar datos.
2. Método *head*: pide una respuesta idéntica a la que correspondería a una petición *get*, pero en la respuesta no se devuelve el cuerpo; esto es útil para poder recuperar los metadatos de los encabezados de respuesta, sin tener que transportar todo el contenido.
3. Método *post*: envía datos para que sean procesados por un recurso identificado que se incluirán en el cuerpo de la petición.
4. Método *put*: sube o carga un recurso especificado (archivo o fichero) y es más eficiente que el método *post*, porque permite escribir un archivo en una conexión socket establecida con el servidor.

3.3.2. HTML 5

De acuerdo con la documentación de Mozilla[24], HyperText Markup Language (HTML o lenguaje de marcado de hipertextos) es la pieza más básica en la construcción de la web, usada para definir el sentido y estructura del contenido en una página web.

Es un estándar a cargo del World Wide Web Consortium (W3C o Consorcio WWW), organización dedicada a la estandarización de casi todas las tecnologías ligadas a la web.

HTML hace uso de enlaces que conectan las páginas web entre sí, ya sea dentro de un mismo sitio web o entre diferentes sitios web.

Un elemento HTML se separa de otro texto en un documento por medio de “etiquetas”, las cuales consisten en elementos rodeados por “<,>”.

HTML 5 (HyperText Markup Language, versión 5) es la última revisión importante del lenguaje HTML en el que establece elementos y atributos que reflejan el uso de sitios web modernos.

Características:

1. Incorpora etiquetas: *canvas* 2D & 3D, audio y vídeo con códecs para mostrar los contenidos multimedia; actualmente hay una lucha entre imponer códecs libres (WebM + VP8) o privados (H.264/MPEG-4 AVC).
2. Etiquetas para manejar grandes conjuntos de datos: permiten generar tablas dinámicas para filtrar, ordenar y ocultar contenido en cliente.
3. Mejoras en los formularios: nuevos tipos de datos como *email*, *number*, *url*, *datetime* y facilidades para validar contenido sin JavaScript.
4. Visores: MathML (fórmulas matemáticas) y SVG (gráficos vectoriales).
5. Drag & Drop: nueva funcionalidad para arrastrar objetos como imágenes.

Respecto a la compatibilidad con los navegadores, la mayoría de elementos de HTML5 son compatibles con Firefox 19, Chrome 25, Safari 6 y Opera 12 en adelante.

3.3.3. Style Sheet Language: CSS vs SASS

De acuerdo con Lie[25], un *style sheet language* es un lenguaje que representa los estilos o elementos visuales de documentos estructurados.

CSS

De acuerdo con la documentación de la W3C[26], Cascading Style Sheets (CSS, o hojas de estilo en cascada) es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado.

La separación entre el contenido del documento y la presentación busca mejorar la accesibilidad, proveer más flexibilidad y control, permitir que varios documentos HTML compartan un mismo estilo usando una sola hoja de estilos separada en un archivo .css, reducir la complejidad y la repetición de código.

La especificación CSS describe un esquema prioritario para determinar qué reglas de estilo se aplican si más de una regla coincide para un elemento en particular; estas reglas son aplicadas con un sistema llamado de cascada, de modo que las prioridades son calculadas y asignadas a las reglas, así que los resultados son predecibles.

La especificación CSS es mantenida por el World Wide Web Consortium; el tipo MIME *text/css* está registrado para su uso por CSS descrito en el RFC 23185.

CSS se ha creado en varios niveles y perfiles, donde cada nivel de CSS se construye sobre el anterior, generalmente añadiendo funciones al nivel previo.

La última versión del estándar, CSS3.1, está dividida en varios documentos separados, llamados “módulos”; cada módulo añade nuevas funcionalidades a las definidas en CSS2, de manera que se preservan las anteriores para mantener la compatibilidad.

Los trabajos en CSS3.1 comenzaron a la vez que se publicó la recomendación oficial de CSS2 y su primeros borradores fueron liberados en junio de 1999.

Debido a la modularización de CSS3.1, diferentes módulos están en diferentes estados de su desarrollo, hay alrededor de cincuenta módulos publicados, tres de ellos se convirtieron en recomendaciones oficiales de la W3C en 2011: “selectores”, “espacios de nombres” y “color”.

Respecto al soporte de los navegadores web, cada navegador web usa un motor de renderizado para renderizar páginas web y el soporte de CSS no es exactamente igual en ninguno de los motores de renderizado; ya que los navegadores no aplican el CSS correctamente, muchas técnicas de programación han sido desarrolladas para ser aplicadas por un navegador específico (comúnmente conocida esta práctica como *CSS hacks* o *CSS filters*).

Sass

De acuerdo con su documentación oficial[27], Sass es un lenguaje de preprocesado que genera hojas de estilo en cascada (CSS) y consta de dos sintaxis.

Características

Variables Las variables comienzan con un signo de dólar y la asignación de valor se realiza con dos puntos y permite 4 tipos de datos:

1. Números (incluyendo las unidades).
2. Strings (con comillas o sin ellas).
3. Colores (código o nombre).
4. Booleanos.

Las variables son resultados o argumentos de varias funciones disponibles.

Anidamiento CSS soporta anidamiento lógico, pero los bloques de código no son anidados; Sass permite que el código anidado sea insertado dentro de cualquier otro bloque.

Mixins Como CSS no soporta *mixins*, cualquier código duplicado debe ser repetido en cada lugar; un *mixin* en Sass es una sección de código que contiene código Sass.

Cada vez que se llama un *mixin* en el proceso de conversión, el contenido del mismo es insertado en el lugar de la llamada; los *mixin* permiten una solución limpia a las repeticiones de código, así como una forma fácil de alterar el mismo.

Elección

Tomando en cuenta la experiencia del equipo con CSS, además de que el proyecto no se enfocará en hacer muchas hojas de estilo para cada componente, página o vista de la aplicación web, se usará CSS en lugar de Sass.

3.3.4. JavaScript vs TypeScript

De acuerdo con Stack Overflow^[28], los web *frameworks* más populares están escritos en JavaScript/TypeScript, por ello se realizó una investigación del lenguaje más apto para el proyecto.

JavaScript

De acuerdo con la documentación de Mozilla^[29], JavaScript es una marca registrada con licencia de Sun Microsystems (ahora Oracle) que se usa para describir la implementación del lenguaje de programación JavaScript.

Debido a problemas de registro de marcas en la Asociación Europea de Fabricantes de Computadoras, la versión estandarizada del lenguaje tiene el nombre de ECMAScript, sin embargo, en la práctica se conoce como lenguaje JavaScript.

Abreviado como JS, es un lenguaje ligero e interpretado, orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico; es usado en node.js, Apache CouchDB y Adobe Acrobat.

El núcleo del lenguaje JavaScript está estandarizado por el Comité ECMA TC39 como un lenguaje llamado ECMAScript y la última versión de la especificación es ECMAScript 6.0 que define:

1. Sintaxis: reglas de análisis, palabras clave, flujos de control, inicialización literal de objetos.
2. Mecanismos de control de errores: *throw*, *try/catch*, habilidad para crear tipos de errores definidos por el usuario.
3. Tipos: *boolean*, *number*, *string*, *function*, *object*.
4. Objetos globales: en un navegador, los objetos globales son los objetos de la ventana, pero ECMAScript solo define una API no específica para navegadores, como *parseInt*, *parseFloat*, *decodeURI* o *encodeURI*.
5. Mecanismo de herencia basada en prototipos.
6. Objetos y funciones incorporadas.
7. Modo estricto.

La sintaxis básica es similar a Java y C++ con la intención de reducir el número de nuevos conceptos necesarios para aprender el lenguaje; las construcciones del lenguaje, como sentencias *if*, bucles *for*, *while*, bloques *switch* y *try catch* funcionan de la misma manera que en estos lenguajes (o casi).

JavaScript funciona como lenguaje procedimental y como lenguaje orientado a objetos; los objetos se crean añadiendo métodos y propiedades a lo que de otra forma serían objetos vacíos en tiempo de ejecución, en contraposición a las definiciones sintácticas de clases comunes en los lenguajes compilados como C++ y Java.

Las capacidades dinámicas de JavaScript incluyen construcción de objetos en tiempo de ejecución, listas variables de parámetros, variables que contienen funciones, creación de scripts dinámicos (mediante eval), introspección de objetos (mediante *for ... in*), y recuperación de código fuente.

Desde 2012 todos los navegadores modernos soportan completamente ECMAScript 5.1 y el 17 de Julio de 2015 ECMA International publicó la sexta versión de ECMAScript, oficialmente llamada ECMAScript 2015 y fue inicialmente nombrada como ECMAScript 6 o ES6. Desde entonces, los estándares ECMAScript están en ciclos de lanzamiento anuales.

TypeScript

De acuerdo con TypeScript Publishing, TypeScript es por definición JavaScript para el desarrollo de aplicaciones, siendo también un superconjunto del mismo.

TypeScript es un lenguaje compilado orientado a objetos, fue diseñado por Anders Hejlsberg (diseñador de C#) en Microsoft; es tanto un lenguaje como un conjunto de herramientas y es un superconjunto de JavaScript porque genera código JavaScript.

Características

- Compilación: cuenta con un transpilador para la verificación de errores si hay errores de compilación, cosa que no es posible con JavaScript.
- Típico estático fuerte: provee un sistema opcional de tipado estático y de inferencia de tipos a través del TypeScript Language Service, lo que permite inferir el tipo de una variable declarada sin tipo en función de su valor.
- Definiciones de tipo: permite la extensión del lenguaje con bibliotecas externas JavaScript.
- Programación orientada a objetos: admite conceptos como clases, interfaces, herencia, etc.

Elección

De acuerdo con el sitio Stack Overflow[30], JavaScript es el lenguaje más popular de 2019 y aunque TypeScript es de los lenguajes que tienen un mayor

nivel de aceptación, se usará JavaScript no solo por ser el lenguaje más popular y, en consecuencia, con más compatibilidad y material de ayuda, sino también porque el equipo está acostumbrado a este lenguaje y tiene experiencia con web *frameworks* escritos en JavaScript.

3.3.5. JavaScript Web Frameworks: Vue/Nuxt vs React vs Angular

De acuerdo con Wired[31], un web *framework* es un conjunto de *software* que permite el desarrollo de una aplicación web y en el lenguaje JavaScript hay varias opciones, incluidas las más populares Vue, React y Angular.

React

De acuerdo con su documentación oficial[32], React es una biblioteca JavaScript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de *single page applications*.

Características

1. Virtual DOM: React usa un virtual DOM propio en lugar del navegador.
2. Props: son definidos como atributos de configuración para cada componente.
3. Estado de cada componente: lleva un registro de las propiedades y atributos del componente.
4. Ciclos de vida: son la serie de estados por los cuales pasan los componentes *statefull* a lo largo de su existencia.

Angular

De acuerdo con su documentación oficial[33], Angular es un web *framework* desarrollado en TypeScript de código abierto mantenido por Google que se utiliza para crear y mantener *single page applications*.

1. Generación de código.
2. Componentes.
3. Ciclos de vida de componentes.

Vue

De acuerdo con su documentación oficial[34], Vue es un *framework* progresivo para desarrollar interfaces de usuario; a diferencia de otros *frameworks*, Vue está diseñado desde para ser utilizado incrementalmente.

La biblioteca central está enfocada solo en la capa de visualización y es fácil de utilizar e integrar con otras bibliotecas o proyectos existentes; por otro lado, Vue también es capaz de impulsar sofisticadas *single-page applications* cuando se utiliza en combinación con bibliotecas de apoyo.

Comparación con React React y Vue comparten muchas similitudes; ambos utilizan un DOM virtual, proporcionan componentes de vista reactivos, enrutamiento y la gestión global del estado manejado por bibliotecas asociadas.

Tanto React como Vue ofrecen un rendimiento comparable en los casos de uso más comunes, con Vue normalmente un poco por delante debido a su implementación más ligera del DOM virtual.

En Vue las dependencias de un componente se rastrean automáticamente durante su renderizado, por lo que el sistema sabe con precisión qué componentes deben volver a renderizarse cuando cambia el estado; se considera que cada componente tiene un *shouldComponentUpdate* automáticamente implementado.

Comparación con Angular En términos de rendimiento, ambos *frameworks* son excepcionalmente rápidos y no hay suficientes datos de casos de uso en el mundo real para hacer un veredicto.

Vue es mucho menos intrusivo en las decisiones del desarrollador que Angular, ofreciendo soporte oficial para una variedad de sistemas de desarrollo, sin restricciones sobre cómo estructurar su aplicación; muchos desarrolladores disfrutan de esta libertad, mientras que algunos prefieren tener solo una forma correcta de desarrollar cualquier aplicación.

Para empezar con Vue, todo lo que se necesita es familiarizarse con HTML y ES5 JavaScript, mientras que la curva de aprendizaje de Angular es mucho más pronunciada.

La complejidad de Angular se debe en su enfoque para diseñar aplicaciones grandes y complejas, pero eso hace que el *framework* sea mucho más difícil de entender.

Nuxt.js

De acuerdo con su documentación oficial[35], el objetivo de Nuxt.js es hacer que el desarrollo web en Vue sea eficaz con herramientas de desarrollo como Webpack, Babel y PostCSS;

Características

1. Manejo de archivos Vue (*.vue).
2. División automática de código.
3. Representación del lado del servidor.

-
- 4. Potente sistema de enrutamiento con datos asincrónicos.
 - 5. Servicio de archivos estáticos.
 - 6. Soporte sintaxis ES2015+ (Javascript ES6).
 - 7. Gestión de elementos <head> <title>, <meta> y similares.
 - 8. Preprocesador: Sass, Less, Stylus, etc..

Elección

Se usará Vue/Nuxt por ser el *framework* con el que el equipo está más acostumbrado, además de ser el máx flexible de las opciones expuestas.

3.3.6. CSS Frameworks: Vuetify vs Bootstrap

De acuerdo con Wikipedia^[36], un CSS *framework* es una biblioteca de estilos genéricos usada para implementar diseños web y aportan una serie de utilidades que son aprovechadas frecuentemente en los distintos diseños web.

Vuetify

De acuerdo con la documentación de Google^[37], Material Design es un lenguaje visual que sintetiza los principios clásicos del buen diseño respecto a las ideas de Google y en estos principios está basado Vuetify.

El objetivo de Material Design es crear un lenguaje visual que sintetice los principios clásicos del buen diseño, unificar el desarrollo de un único sistema subyacente para la experiencia del usuario en plataformas y dispositivos, así como personalizar el lenguaje visual de Material Design.

Asimismo, de acuerdo con la documentación de Vuetify^[38], este CSS *framework* está integrado para ser usado en los componentes de Vue/Nuxt como botones, barras de navegación, *layouts* y demás.

Bootstrap

De acuerdo con su documentacion oficial^[39], Bootstrap es un CSS *framework* orientado al diseño responsivo de una aplicación web.

Tiene *templates* para botones, barras de navegación, estilos de tipografía entre otros; es de fácil integración con React, Angular o Vue y tiene una comunidad extensa por los años y popularidad que tiene.

Elección

Se ha elegido usar en una primera instancia Vuetify porque es un CSS *framework* que está integrado en las tecnologías asociadas de Vue, como Vue Router, Vue Meta;

asimismo, sus componentes son simples de entender y de implementar.

3.3.7. MySQL vs MongoDB

MongoDB

De acuerdo con su documentación oficial[40], MongoDB (del inglés humongous, “enorme”) es un sistema de base de datos NoSQL, orientado a documentos y de código abierto.

En lugar de guardar los datos en tablas, tal y como se hace en las bases de datos relacionales, MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

Características

1. Consultas *ad hoc*: MongoDB soporta la búsqueda por campos, consultas de rangos y expresiones regulares.
2. Indexación: es posible que cualquier campo en un documento de MongoDB sea indexado, al igual que es posible hacer índices secundarios.
3. Replicación: MongoDB soporta el tipo de replicación primario-secundario.
4. Balanceo de carga: MongoDB escala de forma horizontal usando el concepto de *sharding*.
5. Almacenamiento de archivos: MongoDB es utilizado como un sistema de archivos, aprovechando su capacidad para el balanceo de carga y la replicación de datos en múltiples servidores.
6. Agregación: MongoDB proporciona un framework de agregación que permite realizar operaciones similares a la operación *group by* de SQL.

MySQL

De acuerdo con su documentación oficial[41], MySQL es un gestor de base de datos relacionales de código abierto con un modelo cliente-servidor.

Archiva datos en tablas separadas en lugar de guardar todos los datos en un gran archivo, permitiendo tener mayor velocidad y flexibilidad; estas tablas están relacionadas de formas definidas, por lo que se hace posible combinar distintos datos en varias tablas y conectarlos.

Características

1. Permite escojer múltiples motores de almacenamiento para cada tabla.

-
2. Agrupación de transacciones, pudiendo reunirlas de forma múltiple desde varias conexiones con el fin de incrementar el número de transacciones por segundo.
 3. Conectividad segura.
 4. Ejecución de transacciones y uso de claves foráneas.
 5. Presenta un amplio subconjunto del lenguaje SQL.

Elección

De acuerdo con Mosquera[6], MongoDB es la base de datos NoSQL orientada a documentos más popular y usada en los *papers* de investigación; por ello el equipo ha decidido usar MongoDB como base de datos.

3.3.8. Bibliotecas JavaScript para diagramado: GoJS vs Fabric.js vs D3.js

GoJS

De acuerdo con la documentación de GoJS[42], GoJS es una biblioteca de JavaScript y TypeScript para crear diagramas interactivos; permite crear todo tipo de diagramas, desde diagramas de flujo y organigramas hasta diagramas industriales altamente específicos, diagramas SCADA y BPMN, diagramas médicos como genogramas y diagramas de modelos de brotes.

Ofrece muchas funciones para la interactividad del usuario, como arrastrar y soltar, copiar y pegar, edición de texto, información sobre herramientas, menús contextuales, diseños automáticos, plantillas, enlace de datos y modelos, gestión de estado, paletas, descripciones generales, controladores de eventos, comandos, herramientas extensibles para operaciones personalizadas y animaciones personalizables.

Está escrita en TypeScript y puede usarse como una biblioteca de JavaScript o incorporarse a su proyecto desde fuentes de TypeScript; normalmente se ejecuta completamente en el navegador, renderizando a un elemento *canvas* HTML o SVG sin ningún requisito del lado del servidor.

Fabric.js

De acuerdo con la documentación de Fabric[43], Fabric.js es una biblioteca de JavaScript que proporciona un modelo para trabajar sobre un *canvas* HTML5 para poder agregar objetos como rectas, circunferencias, rectángulos, etc.

Características

1. Drag & Drop integrado en cada objeto de Fabric.js

-
2. Permite la especialización de clases para crear objetos personalizados.

D3.js

De acuerdo con la documentación de D3[44], D3.js es una biblioteca de JavaScript para producir infogramas dinámicos e interactivos en navegadores web.

Características

1. Selecciones: es posible la selección de elementos del documento HTML y asignarle propiedades.
2. Transiciones: permiten interpolar en el tiempo valores de atributos, lo que produce cambios visuales en los infogramas.
3. Asociación de datos: se asocia a cada elemento un objeto SVG con propiedades (forma, colores, valores) y comportamientos (transiciones, eventos).

Elección

Se llevó a la práctica en el prototipo funcional las tres opciones antes expuestas junto con algunas otras y se decidió usar GoJS para el proyecto por ser la biblioteca JavaScript más completa para diagramado y que genera los diagramas con un JSON simple para parsear esos datos y realizar la conversión.

3.3.9. Python 3

De acuerdo con Mark Lutz[45], Python es un lenguaje de programación interpretado, interactivo y orientado a objetos; incorpora módulos, excepciones, tipo dinámico, tipos de datos dinámicos y clases.

La biblioteca estándar del lenguaje es extensible en C o C++, cubre áreas como el procesamiento de cadenas (expresiones regulares, Unicode, cálculo de diferencias entre archivos), protocolos de Internet (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, programación CGI), ingeniería de software (pruebas unitarias, registro, creación de perfiles, análisis del código Python) e interfaces del sistema operativo (llamadas al sistema, sistemas de archivos, *sockets TCP/IP*).

Características

1. Es extensible.
2. Tiene escritura dinámica.
3. Gestión automática de la memoria.
4. Tipos de objetos incorporados.

-
- 5. Herramientas incorporadas.
 - 6. Utilidades de biblioteca.
 - 7. Utilidades de terceros.
 - 8. Es orientado a objetos y funcional.

Elección

Se ha elegido Python para desarrollar los algoritmos del proyecto dado que es multiplataforma y es de fácil integración con el *framework* web elegido para el *back end*.

3.3.10. Back end: Django vs Flask

Para elegir el lenguaje a usar para el *back end* se realizó un estudio de lenguajes apropiados para usar con un JavaScript web *framework*.

Flask

De acuerdo con su documentación oficial[46], Flask es un *micro web framework* escrito en Python; se clasifica como micro porque no requiere herramientas o bibliotecas particulares; está basado en la especificación WSGI de Werkzeug, el motor de templates Jinja2 y tiene una licencia BSD.

No tiene capa de abstracción de base de datos, validación de formularios ni ningún otro componente donde las bibliotecas de terceros preexistentes brinden funciones comunes; sin embargo, Flask admite extensiones que agregan características de la aplicación como si se implementaran en el propio Flask.

Existen extensiones para mapear relaciones de objetos, validación de formularios, manejo de carga, varias tecnologías de autenticación de licencia libre.

Características

- 1. Es un *framework* que se destaca en instalar extensiones o complementos de acuerdo al tipo de proyecto que se va a desarrollar, es decir, es perfecto para el prototipado rápido de proyectos.
- 2. Incluye un servidor web, así podemos evitamos instalar uno como Apache o Nginx; además, ofrece soporte para pruebas unitarias y para *cookies*, apoyándose en el motor de plantillas Jinja2.
- 3. Su velocidad es mejor a comparación de Django; generalmente el desempeño que tiene Flask es superior debido a su diseño minimalista que tiene en su estructura.
- 4. Flask permite combinarse con herramientas para potenciar su funcionamiento, por ejemplo: Jinja2, SQLAlchemy, Mako y Peewee.

Django

De acuerdo con su documentación oficial[46], Django es un *framework* de desarrollo web de código abierto escrito en Python, que sigue el patrón de diseño conocido como MVC (Modelo–Vista–Controlador).

Características

1. Aplicaciones “enchufables” que pueden instalarse en cualquier página gestionada con Django.
2. Una API de base de datos robusta.
3. Un sistema incorporado de “vistas genéricas” que ahorra tener que escribir la lógica de ciertas tareas comunes.
4. Un sistema extensible de plantillas basado en etiquetas, con herencia de plantillas.
5. Un despachador de URL basado en expresiones regulares.
6. Un sistema *middleware* para desarrollar características adicionales.
7. Documentación incorporada accesible a través de la aplicación administrativa.

Elección

Se ha elegido Flask por ser un *framework* web conocido por el equipo, porque es ideal para el prototipado rápido de proyectos y al equipo le ha dado resultados en proyectos anteriores.

3.3.11. Herramientas para ingeniería dirigida por modelos

De acuerdo con Kahani [47], en su estudio sobre unas 60 herramientas en la literatura del tema para desarrollar transformaciones entre modelos, el 75 % de las herramientas están hechas en Java[48] y están implementadas como un *plugin* del IDE Eclipse[49]. Igualmente, en las herramientas de otros lenguajes, por ejemplo, Python, aún no existe una integración entre transformaciones modelo a modelo y modelo a texto con la definición de gramáticas.

Sin embargo, como PyEcore [50] permite crear la definición de clases equivalentes en Python desde un archivo .ecore, se decidió usar PyEcore para implementar el modelo conceptual y lógico del modelo NoSQL.

Metamodelo para describir un modelo orientado a documentos: Ecore

Ecore es el metamodelo de Eclipse Modeling Framework; como metamodelo permite definir otros modelos. El uso de Ecore como un metamodelo permite que aprovechar el ecosistema y las herramientas del EMF.

Transformación modelo a modelo: Python

PyEcore genera una definición de clases estáticas de acuerdo a una definición de lenguaje Ecore.

Usando PyEcore se genera la definición de clases del modelo conceptual y lógico del modelo NoSQL para poder implementar los algoritmos en el lenguaje de programación Python.

Tramsformación modelo a texto: Python

No se considera necesario definir una nueva gramática y generar la transformación modelo a texto per se. Para el alcance de la propuesta de solución basta escribir un *parser* simple de la denificación obtenida de la transformación modelo a modelo.

Metamodelo GDM: Ecore

Se hará uso de la definición de lenguaje ecore del [Generic Datametamodel](#) y del [Document Data Metamodel](#) definidos por Alfonso de la Vega.

3.4. Conclusiones

Los modelos de datos que han sido presentados en este capítulo son el fundamento para diseñar bases de datos; tradicionalmente, el modelado de bases de datos relacionales empieza con el modelo entidad-relación (para el modelo conceptual), se transforma al modelo relacional para normalizar cada relación y por medio de SQL se obtiene el esquema de la base de datos.

Sin embargo, esta manera de diseñar bases de datos no es la más apropiada para el modelado NoSQL, porque como comenta Chebotko[4], es clave conocer desde el modelo conceptual cómo se realizarán las consultas; además, por la naturaleza de los modelos de datos NoSQL, operaciones como la normalización no son lo más apropiado, porque la normalización es contraria a un concepto fundamental de los modelos de datos NoSQL: la agregación de datos anidados.

Asimismo, de las tecnologías expuestas para desarrollar la propuesta de solución se ha elegido Flask para el *back end* por ser un *framework* web conocido por el equipo, es ideal para el prototipado rápido de proyectos y el equipo tiene experiencia con este *framework*.

Se ha elegido JavaScript porque es el lenguaje más popular de 2019, tiene compatibilidad, material de ayuda y también porque el equipo tiene experiencia con web *frameworks* escritos en JavaScript.

Como ya se había mencionado, de acuerdo con Mosquera[6], MongoDB es la base de datos NoSQL orientada a documentos más popular y usada en los *papers* de investigación; por ello el equipo ha decidido usar MongoDB como base de datos.

La elección de GoJS es por ser la biblioteca JavaScript más completa para diagramado de todas las que se probaron en el prototipo funcional.

La decisión de usar CSS es porque la propuesta de solución no está enfocada en diseñar estilos para los diferentes componentes de Vue; además, se ha decidido delegar el aspecto visual a un CSS *framework*, Vuetify, que está integrado en las tecnologías asociadas de Vue.

La decisión de usar PyEcore [50] es porque es la única opción viable de integrar las transformaciones modelo a modelo y las definiciones de las gramáticas creadas por De la Vega [8].

Finalmente, se han elegido Vue/Nuxt por ser los *frameworks* con los que el equipo tiene más experiencia y para implementar los algoritmos de la propuesta de solución se ha optado por usar Python, dado que es multiplataforma y es de fácil integración con el *framework* web elegido para el *back end*.

Capítulo 4

Análisis del diseño

En este capítulo se muestra la metodología a usar y el porqué; se ve si es factible o no la propuesta de solución con estudios como la factibilidad técnica, económica y costos de desarrollo; asimismo, en el análisis del sistema se muestran las historias de usuario y la lista del producto a desarrollar; se muestran los principales algoritmos a implementar, como la validación estructural, el mapeo del modelo entidad-relación básico al modelo relacional, la obtención del esquema relacional en sentencias SQL, el mapeo del modelo entidad-relación al GDM, el mapeo del GDM a un modelo lógico orientado a documentos, la obtención del esquema en sentencias de MongoDB y las conclusiones del capítulo.

4.1. Metodología

De acuerdo con la Universidad Católica los Ángeles^[51], en el campo del desarrollo de *software* hay dos grupos de metodologías: las tradicionales y las ágiles.

Las tradicionales se centran en cumplir con un plan rígido de trabajo establecido en la etapa inicial del proyecto, mientras que las ágiles permiten realizar cambios en los requerimientos conforme avance el mismo.

Dado que cualquier cambio en el proceso de una metodología tradicional genera la necesidad de una reconstrucción del plan de trabajo (invirtiendo tiempo que se podría usar para desarrollar), surgieron las metodologías ágiles, que permiten realizar cambios en los requerimientos conforme avance el proyecto.

Se han propuesto muchos modelos ágiles de proceso y están en uso en toda la industria; entre ellos se encuentran los siguientes:

- DAS (Desarrollo Adaptativo de Software): tiene como fundamento la teoría de sistemas adaptativos complejos; por ello, interpreta los proyectos de *software* como sistemas adaptativos complejos compuestos por agentes (los interesados), entornos (organizacional o tecnológico) y salidas (el producto desarrollado)^[52].
- Scrum: es un marco de trabajo diseñado para lograr la colaboración eficaz de

equipos en proyectos, que emplea un conjunto de reglas, artefactos y define roles que generan la estructura necesaria para su correcto funcionamiento[52].

- MDSD (Método de Desarrollo de Sistemas Dinámicos): es un marco de trabajo creado para entregar la solución correcta en el momento correcto; utiliza un ciclo de vida iterativo, fragmenta el proyecto en periodos cortos de tiempo y define entregables para cada uno de estos periodos; tiene roles claramente definidos y especifica su trabajo en periodos de tiempo[52].
- Crystal: la filosofía de Crystal define el desarrollo como un juego cooperativo de invención y comunicación cuya meta principal es entregar *software* útil, que funcione y su objetivo secundario es preparar el próximo juego[52].

4.1.1. Scrum

De acuerdo con Ken Schwaber[53], Scrum es un marco de trabajo para la entrega de productos incrementales y de máximo valor productivo.

Un artefacto es un elemento que garantiza la transparencia, es el registro de la información fundamental del proceso Scrum y a continuación se describen sus cuatro artefactos principales:

- Lista de producto (*product backlog*): es el listado de todas las tareas que necesita el proyecto para alcanzar su realización; al iniciar el desarrollo del proyecto, esta lista no se encuentra completa y conforme avanzan los *sprints* se le añaden tareas para solventar las necesidades que van surgiendo gracias a la retroalimentación del cliente.
- Lista de pendientes del *sprint* (*sprint backlog*): es la lista de tareas seleccionadas del *product backlog* que se planifica realizar durante el periodo del *sprint* y define a los responsables de cada tarea.
- *Sprint*: es el corazón de Scrum, tiene un periodo de tiempo determinado de un mes o incluso menos donde el equipo completa conjuntos de tareas incluidas en el *backlog* para crear un incremento del producto utilizable.
- Incremento: es la suma de todos los elementos de la lista de productos completados durante un *sprint* unido con los incrementos de los *sprints* anteriores; al finalizar el *sprint*, el nuevo incremento debe estar en condiciones de ser utilizado.

El equipo Scrum (*scrum team*) consiste en los siguientes roles:

- El dueño de producto (*product owner*): es la persona responsable de maximizar el valor del producto y el trabajo del equipo de desarrollo; es el único responsable de gestionar la lista de producto y cualquier cambio a esa lista debe ser revisada y aprobada por él.

-
- El equipo de desarrollo (*development team*): son los profesionales que realizan el trabajo para la entrega de un incremento en el producto en cada *sprint*; es un grupo autoorganizado y multifuncional donde cada miembro del equipo tiene habilidades especializadas, pero que la responsabilidad de las tareas completadas, incrementos del producto o retrasos recaen en el equipo como un todo.
 - El Scrum Master: es la persona responsable de asegurar que Scrum es entendido y adoptado por todos los involucrados en el proyecto, asegurándose de ayudar a las personas externas al equipo a entender qué interacciones son útiles con el equipo de desarrollo.

Scrum tiene cuatro eventos principales en un *sprint*, que sirven para la inspección y adaptación del producto que se describen a continuación:

- Planificación del *sprint* (*sprint planning*): es una reunión con el equipo de desarrollo que tiene una duración máxima de 8 horas para el *sprint* de un mes; el Scrum Master es el encargado de que los asistentes entiendan el propósito de dicha reunión.
- Scrum diario (*daily scrum*): es una reunión de un máximo de 15 minutos en la cual el equipo expone sus actividades y planifica las tareas de las próximas 24 horas.
- Revisión del *sprint* (*sprint review*): al finalizar cada *sprint* se lleva a cabo una reunión para la revisión del incremento del producto y en caso de ser necesario realizar ajustes a la lista de producto.
- Retrospectiva del *sprint* (*sprint retrospective*): es cuando el equipo de desarrollo tiene la oportunidad de pensar en mejoras para el próximo *sprint*.

¿Por qué Scrum?

El proyecto requiere de entregas regulares de su avance para poder realizar modificaciones con ayuda de la retroalimentación constante del cliente, en este caso las directoras del proyecto.

Esto otorga beneficios como poder responder con flexibilidad, adaptación a los requisitos de cliente, estrechar la relación con el mismo y mantener al equipo motivado con pequeñas entregas funcionales del producto.

Tomando en cuenta la experiencia del equipo, esta forma de trabajo permite mostrar avances funcionales en el producto en un periodo de tiempo corto para realizar una evaluación y en caso de ser requerido se sugieran cambios.

¿Cómo se aplica Scrum en el proyecto?

Al ser un marco para el desarrollo de proyectos ágil enfocado en el trabajo colaborativo para entregas parciales y regulares, se puede adaptar a las necesidades del proyecto y en caso de ser necesario, modificar los tiempos de cada evento; la forma en cómo se aplicará Scrum es de la siguiente manera:

-
- Se realizarán planificaciones de las actividades semanalmente.
 - El periodo de entrega será de 4 semanas.
 - Las reuniones con el cliente para retroalimentación serán al finalizar cada *sprint* con un tiempo aproximado de 2 horas.

Como la metodología permite definir un periodo de hasta un mes para cada *sprint*, se ha optado por un periodo de 30 días, contemplándose un total de ocho *sprints*, donde al término de cada uno se tendrá un avance del sistema.

4.2. Factibilidad

De acuerdo con Sommerville[54], un estudio de factibilidad es un estudio breve que responde tres preguntas clave:

1. ¿El sistema contribuye a los objetivos generales de la organización?
2. ¿Se puede implementar el sistema dentro del cronograma y el presupuesto utilizando las tecnologías actuales?
3. ¿Se puede integrar el sistema con otros sistemas que se utilizan?

Para responder estas preguntas se expone la factibilidad técnica, factibilidad económica y costos de desarrollo; en las conclusiones del capítulo están las respuestas a las preguntas anteriores.

4.2.1. Factibilidad técnica

De acuerdo con Pressman[55], este estudio determina si el equipo de desarrollo cuenta con los recursos técnicos necesarios para la realización del sistema propuesto; esto se realiza considerando la disponibilidad de los recursos tanto de *hardware*, *software* y recurso humano.

Sistema operativo

De acuerdo con Stallings[56], un sistema operativo es el *software* principal o conjunto de programas de un sistema informático que gestiona los recursos de *hardware* y provee servicios a los programas de aplicación de *software*, ejecutándose en modo privilegiado respecto de los restantes.

Este es un elemento importante, ya que debe cumplir con las características de estabilidad, velocidad, seguridad y escalabilidad para soportar la instalación del sistema.

A continuación se presentan diferentes sistemas operativos que cumplen con las características mencionadas y que son suficientes para albergar el sistema:

-
- Windows: es un grupo de varias familias de sistemas operativos gráficos patentados, son desarrollados y comercializados por Microsoft; cada familia atiende a un determinado sector de la industria informática[57].
 - GNU/Linux: es una familia de sistemas operativos tipo Unix de código abierto basados en el núcleo de Linux creado por Linus Torvalds[58].

El sistema operativo elegido para el desarrollo de la propuesta de solución es GNU/Linux porque es de libre acceso, es gratuito y los integrantes del equipo tienen experiencia con este sistema.

Lenguaje de desarrollo

De acuerdo con Aaby[59], un lenguaje de programación es un lenguaje formal (es decir, un lenguaje con reglas gramaticales definidas) que le proporciona a una persona, en este caso el programador, la capacidad de escribir una serie de instrucciones o secuencias de órdenes en forma de algoritmos con el fin de controlar el comportamiento físico o lógico de una computadora, de manera que es posible obtener diversas clases de datos o ejecutar determinadas tareas.

Se valora que el lenguaje de programación para el desarrollo de la propuesta de solución debe tener soporte para conexión a base de datos, sea posible usarlo para el desarrollo, sea vigente (que esté en continua mejora), sea fácil de administrar y se integre con algún *framework* web.

A continuación se presenta una lista de lenguajes de desarrollo que cumplen dichas características:

- Java: es un lenguaje de programación de propósito general que está basado en clases, orientado a objetos y diseñado para tener la menor cantidad posible de dependencias de implementación; su objetivo es permitir seguir el *write once, run anywhere*[48].
- Python: es un lenguaje de programación interpretado, de alto nivel y de propósito general; la filosofía de diseño de Python enfatiza la legibilidad del código con su uso notable de espacios en blanco significativos; Sus construcciones de lenguaje y su enfoque orientado a objetos tienen como objetivo ayudar a los programadores a escribir código claro y lógico para proyectos de pequeña y gran escala[60].
- C#: es un lenguaje de programación multiparadigma de propósito general como imperativo, declarativo, funcional, genérico, orientado a objetos y orientado a componentes[61].
- JavaScript: es un lenguaje de programación interpretado, de alto nivel y de propósito general; se escribe dinámicamente; es compatible con múltiples paradigmas de programación, incluida la programación orientada a objetos[29].

Los lenguajes de programación Python y JavaScript se han elegido para desarrollar la propuesta de solución; para más detalles revise la sección 3.4

Sistema gestor de base de datos

De acuerdo con Connolly[62], es un sistema de *software* que permite a los usuarios definir, crear, mantener y controlar el acceso a la base de datos.

Este es un factor muy importante, ya que determinará cómo se almacenará la información del sistema, por lo tanto debe ser escalable, seguro, contar con soporte para grandes cantidades de información y soporte para conexión con distintos lenguajes de programación.

A continuación se presenta una lista de sistemas gestores de bases de datos que cumplen dichas características:

- MySQL: es un sistema de gestión de bases de datos relacionales de código abierto[63].
- MongoDB: es un gestor de bases de datos orientado a documentos y utiliza documentos similares a JSON con esquemas opcionales[64].

El gestor de base de datos que se ha elegido para desarrollar la propuesta de solución es MongoDB; para más detalles revise la sección 3.3.7.

La tabla 4.1 muestra las características de las computadoras con los que el equipo de desarrollo dispone; la primera columna hace referencia al número de computadora del que se dispone; la siguiente hace referencia a qué elementos contiene, la tercera columna sus especificaciones y la cuarta columna el costo de cada computadora.

Equipos	Elementos	Especificaciones	Costo
Laptop 1	Memoria RAM	8 GB	
	Almacenamiento	500 GB HDD	
	Procesador	Intel Core i5 6ta gen.	22 500.00 mxn
Laptop 2	Memoria RAM	8 GB	
	Almacenamiento	256 GB SSD	
	Procesador	Intel Core i5 8va gen.	32 000.00 mxn

Tabla 4.1: Computadoras con las que se cuenta

Con los datos que están en la tabla 4.1, se concluye que la tecnología para el desarrollo del sistema existe y se cuenta con los recursos de *hardware* suficientes para iniciar con su implementación.

4.2.2. Factibilidad económica

De acuerdo con Pressman[55], el punto de función es una “unidad de medida” para expresar la cantidad de funcionalidad comercial que un sistema de información (como producto) proporciona a un usuario; los puntos de función se utilizan para calcular una medición de tamaño funcional de *software*.

La tabla 4.2 muestra la ponderación para el cálculo de los puntos de función sin ajustar; para este proyecto se consideran las funciones identificadas multiplicadas por la ponderación marcada en color verde para obtener el total.

Parámetro	Cuenta	Factores de ponderación			Total
		baja	media	alta	
Entradas	6	3	4	6	36
Salidas	5	4	5	7	25
Tablas	1	3	4	6	4
Interfaces	4	7	10	15	40
Consultas	4	5	7	10	28
Conteo total				Σ	133

Tabla 4.2: Cálculo de las métricas por puntos de función

Como señala Pressman en su libro, se utiliza una métrica por puntos de función para realizar una estimación del costo total del proyecto, incluyendo los salarios de los desarrolladores que harán la implementación del sistema, así como los gastos por pagos de servicios que sean necesarios como se muestra en la tabla 4.2.

Métricas orientadas a la función

La tabla 4.3 muestra una serie de preguntas y respuestas; cada respuesta tiene una ponderación, F_i , que va de $i = 1..14$ y representan los factores de ajuste de valor para los puntos de función; asimismo, los valores pueden ir de 0 (no importante o aplicable) a 5 (absolutamente esencial).

Pregunta	Ponderación
¿Requiere el sistema métodos de seguridad y recuperación fiables?	3
¿Se requiere comunicación especializada?	5
¿Existen funciones de procesamiento distribuido?	2
¿Es crítico el rendimiento?	4
¿Se ejecutará el sistema en un entorno operativo existente y fuertemente utilizado?	4
¿Requiere el sistema una entrada de datos interactiva?	5
¿Requiere la entrada de datos interactiva que las transacciones de entrada se lleven a cabo sobre múltiples operaciones?	5
¿Se actualizan los archivos maestros de forma interactiva?	3
¿Son complejas las entradas, salidas, archivos o consultas?	4
¿Es complejo el procesamiento interno?	4
¿Se ha diseñado el código para ser reutilizable?	4
¿Están incluidas en el diseño la instalación y conversión?	3
¿Se ha diseñado el sistema para soportar múltiples instalaciones en diferentes organizaciones?	4
¿Se ha diseñado el sistema para facilitar los cambios y para ser fácilmente utilizable?	4
$\sum F_i =$	54

Tabla 4.3: Factores de ajuste

Puntos de función

La fórmula para obtener los puntos de función, PF, con los factores de ajuste de la tabla 4.3 es la siguiente:

$$PF = \text{conteo total} * (0.65 + (0.01 * \sum F_i)) \quad (4.1)$$

De (4.1) se deben sustituir los valores del conteo total y los factores de ajuste:

$$PF = 133 * (0.65 + (0.01 * 54)) \quad (4.2)$$

$$PF = 158.27 \approx 159 \text{ puntos ajustados} \quad (4.3)$$

De (4.3) aproximadamente se obtienen 159 puntos de función ajustados; una vez obtenidos utilizando la llamada *ball-park* (o estimación indicativa), que es la técnica de macro-estimación que se utiliza en situaciones de falta de información sobre el proyecto y de acuerdo con Carper[65], la siguiente ecuación determina el esfuerzo de desarrollo de un proyecto:

$$\text{Esfuerzo} = \left(\frac{\text{PF}}{150}\right) * \text{PF}^{0.4} \quad (4.4)$$

donde PF son puntos de función ajustados y al sustituir los valores en (4.4):

$$\text{Esfuerzo} = \left(\frac{159}{150}\right) * 159^{0.4} \quad (4.5)$$

$$\text{Esfuerzo} = 8.05 \text{ meses} \quad (4.6)$$

Los 8.05 meses de (4.6); considerando un total de 40 horas a la semana de trabajo y 4.34 semanas por mes, el total de horas para el desarrollo y conclusión del proyecto se obtiene con la siguiente fórmula:

$$\text{Tiempo de desarrollo} = \text{horas/semana} * \text{semanas/mes} * \text{esfuerzo} \quad (4.7)$$

Al sustituir los valores en (4.7) tenemos:

$$\text{Tiempo de desarrollo} = 40 * 4.34 * 8.05 \quad (4.8)$$

$$\text{Tiempo de desarrollo} = 1397.48 \approx 1398 \text{ horas} \quad (4.9)$$

Por ejemplo, una sola persona trabajando en el desarrollo del proyecto debería invertir 1398 horas con una jornada de 8 horas diarias de lunes a viernes hasta su finalización, por lo que si un equipo de desarrollo es de 2 personas con un horario de lunes a viernes de 4 horas diarias, el proyecto concluiría en 8.05 meses.

4.2.3. Costos de desarrollo

De acuerdo con *Software Guru* [66], en una publicación que recopila los datos de salarios en el área de desarrollo de *software* para febrero de 2020, un desarrollador con 0 a 2 años de experiencia, como es el caso de un estudiante de nivel superior, tiene en un salario de 15 000 mxn mensuales en una jornada completa.

La tabla 4.4 muestra un desglose del costo total de salarios del personal del proyecto, empezando por el costo semanal aproximado por integrante, su costo mensual aproximado y el monto total.

Concepto	Costo semanal aproximado	Costo mensual aproximado	Monto total
Salario	1850 mxn	7500 mxn	67 500 mxn

Tabla 4.4: Costos del personal

Teniendo en cuenta que este proyecto contempla jornadas de medio tiempo (4 horas) de lunes a viernes por un periodo de 9 meses, que es el tiempo aproximado de duración del proyecto, el costo total por salarios para el equipo de desarrollo está desglosado en la tabla 4.4.

Considerado a 2 personas en el equipo de desarrollo y un periodo de 9 meses (se agregó un mes más para el caso de estudio) y utilizando los salarios de la tabla 4.4, tenemos que los gastos totales se obtienen con la siguiente fórmula:

$$\text{salarios} = \text{No. de integrantes} * \text{salario/mes} * \text{tiempo de desarrollo mxn} \quad (4.10)$$

Sustituyendo los valores en (4.10)

$$\text{salarios} = 2 * 7500 * 9 \text{ mxn} \quad (4.11)$$

Esto da como resultado un total final de 135 000 mxn por los salarios de los 2 integrantes del equipo de desarrollo; se tomaron en cuenta 9 meses para todos los gastos, un mes extra a lo obtenido en estimación para utilizarse en el caso de estudio del sistema una vez concluido.

La tabla 4.5 muestra los costos del *software* usado en el proyecto y la tabla 4.6 los costos de servicios usados para el mismo.

Software	Licencia	Costo
Visual Studio Code	MIT	0
Gunicorn(flask Server)	MIT	0
MongoDB Atlas	Apache v2	0

Tabla 4.5: Costos por licencias de software

Concepto	Costo Mensual	Monto total
Luz	250	2250
Internet	349	3141
Heroku hosting	0 (free plan)	0
Netlify hosting	0 (free plan)	0

Tabla 4.6: Costos por servicios

Los gastos por pagos de licencia de *software* quedan excluidos, ya que las tecnologías seleccionadas son libres o gratuitas, lo cual no supone un costo para su uso; de igual manera, esto se encuentra simplificado en la tabla 4.5; otros gastos necesarios son los pagos por servicios requeridos están listados en la tabla 4.6.

Habiendo realizado la suma de todas las cantidades antes mencionadas, el total final se obtiene de la siguiente manera:

$$\text{servicio totales} = \text{servicios por mes} * \text{tiempo de desarrollo mxn} \quad (4.12)$$

$$\text{servicio totales} = 5391 * 9 \text{ mxn} \quad (4.13)$$

$$\text{servicio totales} = 48519.00 \text{ mxn} \quad (4.14)$$

$$\text{Gastos totales} = \text{salarios} + \text{servicios totales} + \text{costos de equipos} \quad (4.15)$$

$$\text{Gastos totales} = 135000 + 48519 + 54500 \text{ mxn} \quad (4.16)$$

$$\text{Gastos totales} = 238019 \text{ mxn} \quad (4.17)$$

$$\text{salarios} = 135000.00 \text{ mxn} \quad (4.18)$$

$$\text{servicios} = (2250 + 3141) * 9 = 48519.00 \quad (4.19)$$

$$\text{Gastos totales} = 135000.00 + 48519.00 = 183519.00 \text{ mxn} \quad (4.20)$$

4.3. Análisis del sistema

De acuerdo con Pressman[55], las condiciones del mercado cambian con rapidez, clientes y usuarios finales necesitan cambios constantes por nuevas amenazas competitivas; por ello los profesionales deben enfocar la ingeniería de *software* en forma que les permita mantenerse ágiles para definir procesos maniobrables, adaptativos y esbeltos que satisfagan las necesidades de los negocios modernos.

Una filosofía ágil para la ingeniería de *software* pone el énfasis en cuatro aspectos clave: la importancia de los equipos con organización propia que tienen el control sobre el trabajo que realizan, la comunicación y colaboración entre los miembros del equipo, profesionales y sus clientes, el reconocimiento de que el cambio representa una oportunidad y la insistencia en la entrega rápida de *software* que satisfaga al consumidor.

4.3.1. Historias de usuario

De acuerdo con Scrum México[67], las historias de usuario conforman la técnica por la que el usuario especifica de manera general los requerimientos que el sistema debe cumplir.

Normalmente estas redacciones se llevan a cabo en tarjetas de papel donde se describen brevemente las funciones que el producto final debe poseer, ya sean requisitos funcionales o no.

El tratamiento de las historias de usuario es flexible y dinámico, cada una de ellas es lo suficientemente detallada y delimitada para que el equipo de desarrollo implemente durante la duración del *sprint*.

Es habitual que se siga una plantilla para estas tarjetas, como la que se expone a continuación:

- Como <Usuario>
- Quiero <algún objetivo>
- Para <motivo>

Una de sus grandes ventajas, dado el caso de que un usuario no sea lo suficientemente detallista con la historia, es que esta se puede partir en historias más pequeñas antes de que el equipo empiece a trabajar en ella.

Este es un ejemplo de historia de usuario para el desarrollo:

- Como usuario,
- quiero ingresar al sistema con mi correo y contraseña
- para tener acceso a sus funciones.

Otra forma de darle detalle a las historias de usuario es mediante el añadido de un criterio de aceptación; un criterio de aceptación es una prueba que será cierta cuando el equipo de desarrollo complete la descripción de la tarjeta.

A continuación se listan las principales historias de usuario que se consideraron para el desarrollo de la propuesta de solución; tenga en cuenta que algunas de ellas tienen criterios de aceptación, pero en otras no se consideró necesarias porque son explícitamente claras.

- N.^o 1
 - Como usuario,
 - quiero ingresar al sistema con mi correo y contraseña,
 - para tener acceso a sus funciones.
- Criterios de aceptación:
 - El usuario recibirá un correo electrónico de confirmación de su alta en el sistema con el correo y contraseña que ingresó para tenerlos de respaldo.

-
- N.^o 2

-
- Como usuario,
 - quiero recuperar mi contraseña en caso de olvidarla,
 - para no perder el trabajo realizado en el sistema.
 - Criterios de aceptación:
 - El usuario podrá ingresar una nueva contraseña siempre y cuando recuerde el correo electrónico con el que se dio de alta en el sistema.
 - Al ingresar una nueva contraseña, recibirá un correo de confirmación del cambio de contraseña y sus datos permanecerán intactos.
-

- N.^o 3
- Como usuario del sistema, quiero darme de alta con una contraseña fácil de recordar,
- pero que esté segura en la base de datos,
- para no tener comprometidos los diagramas que genere en el sistema.
- Criterios de aceptación:
 - Asegurarse de que el usuario ingrese una contraseña de al menos 8 caracteres.
 - Se le solicitará al usuario que ingrese 2 veces la misma contraseña para asegurarse de que le es fácil recordarla y que efectivamente es la misma.
 - Antes de guardar la contraseña, esta deberá pasar por un método que la haga ilegible para el usuario (algún algoritmo de digestión o cifrado).

-
- N.^o 4
 - Como usuario quiero crear un diagrama ER arrastrando y soltando elementos de una “paleta”,
 - para hacerlo de manera más fácil e intuitiva.
 - Criterios de aceptación:
 - El usuario podrá empezar un nuevo diagrama al seleccionar la opción de diagramador ER.
 - Tendrá a su disposición una paleta con los elementos permitidos en un diagrama ER básico.
 - Podrá arrastrar y soltar los elementos de la paleta a un área delimitada para empezar con el diseño de su diagrama.

-
-
- N.^o 5
 - Como usuario quiero guardar mi último trabajo realizado en el diagramador ER,
 - para consultarlos en otro momento.
 - Criterios de aceptación:
 - Dispondrá de un botón para guardar en la base de datos el diagrama que esté creando o editando.
 - Antes de almacenar el diagrama en el *canvas* (o zona de diagramado), se le mostrará un mensaje de confirmación para guardar su diagrama actual.
-

- N.^o 6
 - Como usuario me gustaría ver el último trabajo que realice
 - cuando seleccione la opción “Entidad-Relación”,
 - para modificar el diseño.
-

- N.^o 7
 - Como usuario quiero tener la opción de cargar un diagrama a partir de un archivo,
 - para hacer modificación de dicho diagrama y guardarlo de ser necesario.
 - Criterios de aceptación:
 - El usuario tendrá un botón “cargar” en el menú del diagramador ER para importar un archivo con extensión .json.
 - Al importar el archivo, este pasará por un proceso de validación para asegurarse de que es un archivo “.json” válido.
 - Durante el proceso de validación, se verificará que el contenido del archivo sea un diagrama compatible con la estructura de los generados por el diagramador ER.
 - Al contener información compatible, se mostrará en la zona de diagramado el contenido del archivo.
-

- N.^o 8

-
- Como usuario quiero descargar el diagrama que esté visible en la página web,
 - para distribuirlo como yo desee.
 - Criterios de aceptación:
 - El usuario dispondrá de un botón “Descargar diagrama” en el diagramador ER para obtener un archivo con el contenido del diagrama visible en la zona de diagramado.
 - El archivo generado será de extensión “.json” con la información necesaria para que el diagramador lo cargue en otro momento.
-

- N.^o 9
- Como usuario quiero tener una forma de validar mi diagrama ER,
- porque es importante saber si el diagrama que estoy creando es un diagrama válido del modelo ER.
- Criterios de aceptación:
 - El usuario tendrá disponible un botón que al darle clic, iniciará un proceso de validación del diagrama actual en la zona de diagramado.
 - Al término del proceso de validación, se le mostrará un mensaje al usuario indicando si el diagrama cumple o no las reglas del modelo ER.

-
- N.^o 10
 - Como usuario, después de tener un diagrama ER válido
 - quiero visualizar y poder descargar las sentencias SQL equivalentes,
 - para crear el esquema de base de datos relacional en un DBMS.
 - Criterios de aceptación:
 - Las sentencias SQL solo podrán ser descargadas en el menú “Sentencias SQL” en un archivo con extensión “.sql” dando clic a un botón con la leyenda “Descargar”.
 - Solo se obtendrán las sentencias SQL de un diagrama ER creado y/o validado por el sistema.

-
- N.^o 11
 - Como usuario quiero transformar mi diagrama ER en su equivalente modelo conceptual NoSQL,

-
- para observar el cambio entre modelos.
 - Criterios de aceptación:
 - El usuario dispondrá de la opción de transformar al modelo NoSQL solamente después de haber validado que el diagrama ER cumple con las reglas.
 - Posterior a la validación, se le mostrará al usuario un mensaje de confirmación y un botón para disparar el proceso de transformación al modelo conceptual NoSQL.
 - Una vez validado el diagrama ER e iniciado el proceso para la transformación al modelo conceptual NoSQL, se le indicará al usuario que el proceso tardará un tiempo.
 - Al término del proceso de transformación, se le redirigirá al menú “Modelo lógico y físico NoSQL” donde observará el modelo conceptual NoSQL equivalente a su diagrama ER.
-

- N.^o 12
 - Como usuario quiero obtener el *script* desde el modelo conceptual NoSQL,
 - para generar la base de datos en un gestor de base de datos NoSQL orientado a documentos.
 - Criterios de aceptación:
 - El usuario dispondrá de la opción de obtener el *script* solamente después de haber validado que el diagrama ER cumple con las reglas.
 - Posterior a la validación, se le mostrará al usuario un mensaje de confirmación y un botón para disparar el proceso de generación de *scripts* para el gestor de base de datos orientado a documentos.
 - Una vez empezado el proceso de generación de *scripts*, se le indicará al usuario que el proceso tardará un tiempo.
 - Al término del proceso de transformación, el usuario obtendrá un archivo con extensión “.json” con las sentencias necesarias para crear las colecciones NoSQL equivalentes al diagrama ER.

-
- N.^o 13
 - Como usuario me gustaría tener un reporte técnico y
 - quiero que la redacción sea legible y referenciada,
 - para compartirlo en el futuro con equipos de desarrollo y ver la posibilidad de agregar nuevas funciones al sistema.

Teniendo en cuenta que se está trabajando con una metodología ágil, estas historias de usuario pueden aumentar o en su defecto dividirse en historias más pequeñas dependiendo de los criterios del equipo de desarrollo durante el proceso de la implementación de cada historia.

4.3.2. Lista de producto

De acuerdo con Trigas Gallego[68], la lista de producto es una lista ordenada de todo lo que sería necesario en el producto y es la fuente de requisitos para cualquier cambio a realizarse en el mismo; enumera las características, funcionalidades, requisitos, mejoras y correcciones que constituyen cambios a realizarse en el producto para entregas futuras.

La tabla 4.7 muestra la lista de producto para el proyecto; muestra el número de historia, la tarea a realizar, así como su encargado.

N.º de Historia de Usuario	Requerimiento/Tarea	Responsable
14	Investigación de bases de datos relacionales.	Eduardo/Omar
14	Redacción y selección de las tecnologías a utilizar para el desarrollo de la plataforma.	Eduardo
14	Investigación de bases de datos relacionales.	Eduardo/Omar
14	Redacción de bases de datos relacionales en el documento técnico.	Eduardo
14	Investigación de bases de datos no relacionales.	Eduardo/Omar
14	Redacción de bases de datos no relacionales en el documento técnico.	Eduardo
14	Investigación y selección del modelo de base de datos no relacional a utilizar junto a las tecnologías a utilizar.	Eduardo/Omar
14	Ánalisis y diseño de la arquitectura web.	Eduardo/Omar
1	Desarrollo de la estructura básica del <i>backend</i> .	Omar
1	Desarrollo de la estructura básica del <i>frontend</i> .	Eduardo
1	Agregar servicio <i>backend</i> para registrar un usuario.	Omar
1	Agregar formulario para captura de datos de registro de un usuario en el <i>frontend</i> .	Eduardo
2	Agregar servicio <i>backend</i> para recuperar contraseña del usuario.	Omar
2	Agregar servicio <i>backend</i> para envío de correo al usuario registrado y de recuperación de contraseña.	Omar

Continuación de tabla de lista de producto

2	Agregar vista con formulario para recuperación de contraseña del usuario en el <i>frontend</i> .	Eduardo
2	Integración de los servicios de registro y recuperación de contraseña en el <i>frontend</i> .	Eduardo
3	Agregar servicio <i>backend</i> para hacer ilegible la contraseña del usuario en la base de datos.	Omar
4	Planteamiento de escenarios de los esquemas entidad-relación.	Eduardo
4	Agregar a la interfaz gráfica de la aplicación web el menú “Entidad-Relación”.	Eduardo
4	Agregar íconos de los elementos básicos de un diagrama ER en el diagramador.	Eduardo
5	Agregar servicio <i>backend</i> para guardar un diagrama ER en formato JSON en la base de datos e integrarlo al <i>frontend</i> .	Omar
5	Agregar servicio <i>backend</i> para recuperar el diagrama guardado del usuario de la base de datos y regresarlo en formato JSON.	Omar
6	Recuperar el último diagrama del usuario del <i>backend</i> y mostrarlo en el <i>frontend</i> .	Eduardo
6	Manejar el estado de la interfaz web para no perder el diagrama ER que está editando el usuario.	Eduardo
6	Definición de las reglas del modelo entidad-relación.	Eduardo
4	Implementar la edición de diagramas ER en el <i>frontend</i> .	Eduardo
7	Habilitar la carga de un archivo en la aplicación web.	Omar
7	Agregar la función para validar el contenido del archivo .json y pintarlo en la zona de diagramado.	Eduardo
8	Agregar la descarga del diagrama visible en la zona de diagramado a un archivo .json.	Eduardo
9	Agregar botón de validar al <i>frontend</i> y mostrar el <i>loader</i> mientras se procesa el diagrama ER.	Eduardo/Omar
9	Agregar servicio <i>backend</i> para la validación del diagrama entidad-relación.	Eduardo/Omar
9	implementación de algoritmo para validación del diagrama ER en el <i>backend</i> .	Eduardo/Omar
9	Pruebas de captura de distintos diagramas entidad-relación.	Eduardo/Omar
9	Pruebas para validar el algoritmo de validación.	Eduardo/Omar
10	Agregar servicio al <i>backend</i> para transformación del esquema entidad-relación al modelo relacional.	Omar
10	Implementación del algoritmo de transformación ER -> relacional	Eduardo/Omar

Continuación de tabla de lista de producto

10	Agregar menú relacional a la interfaz gráfica.	Eduardo/Omar
10	Prueba de transformación de distintos diagramas ER al modelo relacional.	Eduardo/Omar
10	Visualización de la transformación del modelo ER al modelo relacional.	Eduardo
14	Revisión de la redacción del reporte técnico para presentación de TT1	Eduardo
11	Agregar servicio <i>backend</i> para la descarga del archivo .sql con las sentencias equivalentes.	Eduardo/Omar
11	Pruebas de coherencia de las sentencias equivalentes en el DBMS.	Eduardo/Omar
12	Definición de las reglas de transformación al modelo NoSQL.	Eduardo/Omar
12	Pruebas de distintos escenarios del modelo relacional al modelo NoSQL.	Eduardo/Omar
12	Agregar servicio al <i>backend</i> para transformación del esquema relacional al modelo NoSQL.	Omar
12	Agregar servicio al <i>backend</i> para guardar el modelo NoSQL en la base de datos.	Omar
12	Agregar menú no relacional a la interfaz gráfica.	Omar
12	Implementación del algoritmo de transformación de modelo relacional al modelo conceptual NoSQL.	Eduardo
12	Comprobación de la coherencia de la transformación entre modelos relacional a no relacional.	Eduardo/Omar
13	Agregar servicio <i>backend</i> para transformación del modelo ER al modelo no relacional.	Eduardo/Omar
13	Ajustar la interfaz del menú ER para mostrar mensaje de transformación al modelo NoSQL.	Omar
13	Manejar el estado del diagrama ER y redireccionar al menú no relacional al terminar la transformación.	Eduardo
13	Pruebas de caso de estudio para verificar la correcta transformación y coherencia de los datos.	Eduardo/Omar
14	Revisión de la redacción del reporte técnico para presentación de TT2	Eduardo

Tabla 4.7: Lista de producto

Se considera la tabla 4.7 como la lista de producto con las tareas necesarias para cumplir con todas las historias de usuario mencionadas en la sección anterior, considerando que es posible que cambien conforme avancen los *sprints* y así añadir nuevas tareas.

4.3.3. Algoritmos para el desarrollo del sistema

De acuerdo con Cormen [69], un algoritmo es cualquier procedimiento computacional definido que toma algún valor, o conjunto de valores, como entrada y produce algún valor, o conjunto de valores, como salida; por lo tanto, un algoritmo es una secuencia de pasos computacionales que transforman la entrada en la salida.

En esta sección se describen los algoritmos a implementar; se empieza con el algoritmo para la validación estructural diagrama entidad-relación; se sigue con algoritmo para el mapeo modelo entidad-relación a relacional; después está la obtención de esquema SQL desde modelo relacional, el mapeo modelo entidad-relación a *Generic Data Metamodel*; el mapeo del *Generic Data Metamodel* a modelo lógico NoSQL y se finaliza con el algoritmo para el modelo lógico NoSQL a modelo físico en MongoDB.

4.3.3.1. Validación estructural diagrama entidad-relación

Las reglas de validación básicas para el diagrama ER:

Generales

1. No puede haber elementos sin conectar.
2. Tampoco puede haber enlaces sin conectar.

Entidad

1. Una entidad es válida si tiene atributos, porque no tiene propósito una entidad sin atributos.
2. La clave primaria puede ser simple o compuesta.
3. La clave primaria no es una clave foránea.
4. La clave primaria debe ser un atributo clave asociado a la entidad (restricción de la propuesta de solución).
5. Dos entidades solo pueden estar conectadas entre sí mediante una relación.
6. Todas las entidades deben tener un atributo clave.
7. En una entidad débil, un atributo solo puede estar asociado a un solo atributo o a una sola entidad.
8. Una entidad débil no debe existir si no tiene una relación con otra entidad.

Atributo

1. Un atributo solo puede estar asociado a un solo atributo o a una sola entidad.
2. Un atributo puede ser compuesto.
3. Un atributo puede ser multivalor.
4. Un atributo puede ser derivado.
5. Un atributo debe tener un nombre.
6. Un atributo no puede usar una relación para asociarse a otro elemento.
7. Un atributo compuesto solo puede estar asociado a una entidad.
8. Un atributo derivado solo puede estar asociado a una entidad.

Relación

1. Una relación solo puede ser entre entidades.
2. El grado de participación máximo es dos (restricción de la propuesta de solución).
3. Una relación puede ser unaria (recursiva).
4. No están permitidas relaciones ternarias o de grado n (restricción de la propuesta de solución).

Para los diferentes tipos de relaciones, de acuerdo con Dullea[7], el modelo ER está compuesto por entidades, las relaciones entre entidades y restricciones en esas relaciones.

La conectividad entre entidades y relaciones se denomina ruta; las rutas son los bloques de construcción de la validez estructural; definen visualmente la asociación semántica y estructural que cada entidad tiene simultáneamente con todas las demás entidades o consigo misma en una ruta.

En el modelado conceptual es posible clasificar la validez en dos tipos: validez semántica y validez estructural; un diagrama ER es válido cuando es válido semántica y estructuralmente; asimismo, los términos validez estructural y validez semántica se definen de la siguiente manera:

1. Validez estructural: un diagrama ER es estructuralmente válido solo cuando la consideración simultánea de todas las restricciones estructurales impuestas en el modelo no implica una inconsistencia lógica en ninguno de los posibles estados;
2. Validez semántica: un diagrama ER es semánticamente válido solo cuando cada relación representa exactamente el concepto del dominio del problema.

Un diagrama ER semánticamente válido muestra la representación correcta del dominio de aplicación que se está modelando; sin embargo, dado que la validez semántica depende de la aplicación, no es posible definir criterios de validez generalizados, por lo que no se considerará la validez semántica, sino únicamente la validez estructural.

En términos generales, un diagrama ER es estructuralmente inválido si contiene construcciones que son contradictorias entre sí o al menos una restricción de cardinalidad es inconsistente.

Un diagrama ER representa la semántica de la aplicación en términos de restricciones de cardinalidad máxima y mínima.

Cada conjunto de restricciones de cardinalidad en una sola relación debe ser coherente con todas las restricciones restantes en el modelo y en todos los estados posibles.

Una relación recursiva es estructuralmente inválida cuando las restricciones de participación y cardinalidad no respaldan la existencia de instancias de datos como lo requiere el usuario y hace que todo el diagrama sea inválido.

En general, un diagrama estructuralmente inválido refleja reglas de negocio semánticamente inconsistentes; para que un modelo sea válido, todas las rutas del modelo también deben ser válidas.

La tabla 4.8 muestra los distintos tipos de relaciones recursivas, mostrando el tipo de relación, la dirección de esa relación, sus restricciones de participación, de cardinalidad y un ejemplo.

En la tabla 4.8, tabla 4.9 y tabla 4.10 se usa la notación uno (1) y muchos (M) para la máxima cardinalidad; una sola línea para indicar la participación opcional y una línea doble para mostrar la participación obligatoria; las palabras “obligatorio” y “opcional” se utilizan en la tabla para indicar la cardinalidad mínima obligatoria (o total) y opcional (o parcial), respectivamente; además, la notación $|E|$ representa el número de instancias en la entidad E .

La tabla 4.8 resume cada relación recursiva por sus propiedades direccionales, la combinación de restricciones de cardinalidad mínima/máxima y ejemplos.

La tabla 4.9 muestra primero reglas de validación para relaciones recursivas y un ejemplo válido; después se da un corolario de la validez de las relaciones recursivas y un ejemplo no válido.

La tabla 4.10 tiene la misma estructura que la tabla 4.9 en la que muestra reglas de validez para las relaciones binarias y después un corolario con un ejemplo no válido.

Tipo de relación	Dirección de la relación	Restricción de participación	Restricciones de cardinalidad	Ejemplo Relación	Roles
Simétrica (reflexiva)	Bidireccional	Opcional-opcional Obligatoria-obligatoria	1-1 M-N	Cónyuge de	Persona
Asimétrica (no reflexiva)	Jerárquica	Unidireccional	Opcional-opcional	1-M 1-1	Supervisa Es supervisado por
			Opcional-opcional Opcional-obligatoria Obligatoria-obligatoria	M-N	Supervisa Es supervisado por
			Opcional-opcional Obligatoria-obligatoria	1-1	Apoya Es apoyado por
Jerárquica Circular	Unidireccional	Opcional-opcional Opcional-obligatoria	1-M	Apoya Es apoyado por	Gerente Empleado
		Opcional-opcional Opcional-obligatoria	M-N	Supervisa	Gerente- empleado
Reflejada	Unidireccional	Opcional-opcional	1-1	Gestiona Se gestiona	CEO

Tabla 4.8: Tipos de relación recursiva válidos según las restricciones de cardinalidad

De la tabla 4.8, una relación recursiva es simétrica o reflexiva cuando todas las instancias que participan en la relación toman un solo papel y el significado semántico de la relación es exactamente el mismo para todas las instancias que participan en la relación independientemente de la dirección en la que se ve; estos tipos de relación se denominan bidireccionales.

De la tabla 4.8, una relación recursiva es asimétrica o no reflexiva cuando hay una asociación entre dos grupos de roles diferentes dentro de la misma entidad y el significado semántico de la relación es diferente dependiendo de la dirección en la que se ven las asociaciones entre los grupos de roles; estos tipos de relación se denominan unidireccionales.

De la tabla 4.8, una relación recursiva es jerárquica cuando un grupo de instancias dentro de la misma entidad se clasifican en calificaciones, órdenes o clases, una encima de otra; implica un comienzo (o arriba) y un final (o abajo) para el esquema de clasificación de instancias.

De la tabla 4.8, una relación recursiva es circular cuando una relación recursiva asimétrica tiene al menos una instancia que no cumple con la jerarquía de clasificación; la relación es unidireccional, ya que se puede ver desde dos direcciones con un significado semántico diferente.

De la tabla 4.8, una relación reflejada existe cuando la semántica de una relación permite que una instancia de una entidad se asocie a sí misma a través de la relación.

Reglas de validación para relaciones recursivas	Ejemplo válido
Solo las relaciones recursivas 1:1 con restricciones de cardinalidad mínimas obligatorias-obligatorias u opcionales son estructuralmente válidas; válido para relaciones simétricas y completamente circular.	
Para las relaciones recursivas 1:M o M:1, la cardinalidad mínima opcional-opcional es estructuralmente válida; válido solo para relaciones asimétricas.	
Para 1:M las relaciones recursivas del tipo jerárquico-circular, la cardinalidad mínima opcional-obligatoria son estructuralmente válidas; válido solo para relaciones jerárquico-circulares.	
Todas las relaciones recursivas con cardinalidad máxima de muchos a muchos son estructuralmente válidas independientemente de las restricciones mínimas de cardinalidad; válido para relaciones simétricas, jerárquicas y jerárquicas-circulares.	
Todas las relaciones recursivas con cardinalidad mínima opcional-opcional son estructuralmente válidas; válido para relaciones simétricas y asimétricas.	
Colorarios de validez para relaciones recursivas	
Todas las relaciones recursivas 1: 1 con restricciones de cardinalidad mínima obligatoria-opcional u opcional-obligatoria son estructuralmente inválidas.	
Todas las relaciones recursivas 1:M o M:1 con restricciones de cardinalidad mínimas obligatorias-obligatorias son estructuralmente inválidas.	
Todas las relaciones recursivas 1:M o M:1 con restricción de participación obligatoria en “uno” y una restricción de participación opcional en las “muchas” restricciones son estructuralmente inválidas.	

Tabla 4.9: Resumen de reglas de validez para relaciones recursivas con ejemplos

Reglas de validez para relaciones binarias.	Ejemplo válido
Una ruta acíclica que contiene todas las relaciones binarias siempre es estructuralmente válida.	
Una ruta cíclica que contiene todas las relaciones binarias y una o más relaciones opcional-opcional siempre es estructuralmente válida.	
Una ruta cíclica que contiene todas las relaciones binarias y una o más relaciones de muchos a uno con participación opcional del lado "uno" siempre es estructuralmente válida.	
Una ruta cíclica que contiene todas las relaciones binarias y una o más relaciones de muchos a muchos es siempre estructuralmente válida.	
Las rutas cíclicas que contienen al menos un conjunto de relaciones opuestas siempre son válidas.	
Una ruta cíclica que contiene todas las relaciones binarias uno a uno que son todas obligatorias-obligatorias o al menos una restricción de cardinalidad mínima opcional-opcional siempre es estructuralmente válida.	

Corolarios de validez para relaciones binarias	Ejemplo no válido
Las rutas cíclicas que no contienen relaciones opuestas ni relaciones autoajustables son estructuralmente inválidas y se denominan relaciones circulares.	
La presencia de una relación "uno a uno obligatorio-obligatorio" no tiene ningún efecto sobre la validez estructural (o invalidez) de una ruta cíclica que contiene otros tipos de relación. (Este corolario se aplica a todas las reglas anteriores).	

Tabla 4.10: Resumen de reglas de validez para relaciones binarias con ejemplos

En la tabla 4.9 se pone un resumen de reglas válidas para relaciones recursivas y en la tabla 4.10 reglas válidas para relaciones binarias.

4.3.3.2. Modelo entidad-relación a relacional

De acuerdo con la propuesta de Elmasri[13], un algoritmo en siete pasos basta para convertir las estructuras de un modelo ER básico con tipos de entidades fuertes y débiles, relaciones binarias con distintas restricciones estructurales, relaciones de grado n y atributos (simples, compuestos y multivalor) en relaciones; a continuación se explica cada paso del algoritmo:

Paso 1: Mapeado de los tipos de entidad regulares Por cada entidad (fuerte) regular E del esquema ER cree una relación R que incluya todos los atributos simples de E .

Incluya únicamente los atributos simples que conforman un atributo compuesto; seleccione uno de los atributos clave de E como clave principal para R .

Si la clave elegida de E es compuesta, el conjunto de los atributos simples que la forman constituirán la clave principal de R .

Si durante el diseño conceptual se identificaron varias claves para E , la información que describe los atributos que forman cada clave adicional conserva su orden para especificar las claves (únicas) secundarias de la relación R .

Paso 2: Mapeado de los tipos de entidad débiles Por cada tipo de entidad débil W del esquema ER con el tipo de entidad propietario E , cree una relación R e incluya todos los atributos simples (o componentes simples de los atributos compuestos) de W como atributos de R .

Además, incluya como atributos de la *foreign key* de R , los atributos de la o las relaciones que correspondan al o los tipos de entidad propietarios; esto se encarga de identificar el tipo de relación de W .

La clave principal de R es la combinación de las claves principales del o de los propietarios y la clave parcial del tipo de entidad débil W si la hubiera.

Si hay un tipo de entidad débil E_2 cuyo propietario también es un tipo de entidad débil E_1 , E_1 debe asignarse antes que E_2 para determinar primero su clave principal.

Paso 3: Mapeado de los tipos de relación 1:1 binaria Por cada tipo de relación 1:1 binaria R del esquema ER, identifique las relaciones S y T que corresponden a los tipos de entidad que participan en R ; hay tres metodologías posibles: (1) la metodología de la *foreign key*, (2) la metodología de la relación mezclada y (3) la metodología de referencia cruzada o relación de relación; la primera metodología es la más útil y la que debe seguirse salvo que se den ciertas condiciones especiales, como las que se explican a continuación:

- 1 **Metodología de la *foreign key*:** seleccione una de las relaciones (por ejemplo, S) e incluya como *foreign key* en S la clave principal de T ; lo mejor es elegir un tipo de entidad con participación total en R en el papel de S . Incluya todos los tributos simples (o los componentes simples de los atributos compuestos) del tipo de relación 1:1 R como atributos de S .
- 2 **Metodología de la relación mezclada:** una asignación alternativa de un tipo de relación 1:1 es posible al mezclar los dos tipos de entidad y la relación en una sola relación.
- 3 **Metodología de referencia cruzada o relación de relación:** consiste en configurar una tercera relación R con el propósito de crear una referencia cruzada de las claves principales de las relaciones S y T que representan los tipos de entidad; como se verá, esta metodología es necesaria para las relaciones M:N binarias; la relación R se denomina relación de relación (y, en algunas ocasiones, tabla de búsqueda), porque cada tupla de R representa una instancia de relación que relaciona una tupla de S con otra de T .

Paso 4: Mapeado de tipos de relaciones 1:N binarias Por cada relación 1:N binaria regular R , identifique la relación S que representa el tipo de entidad participante en el lado N del tipo de relación.

Incluya como *foreign key* en S la clave principal de la relación T que representa el otro tipo de entidad participante en R , porque cada instancia de entidad en el lado N está relacionada, a lo sumo, con una instancia de entidad del lado 1 del tipo de relación.

Incluya cualesquiera atributos simples (o componentes simples de los atributos compuestos) del tipo de relación 1:N como atributos de S .

De nuevo, una metodología alternativa es la opción de relación de relación (referencia cruzada), como en el caso de las relaciones 1:1 binarias.

Se crea una relación R separada cuyos atributos son las claves de S y T , y cuya clave principal es la misma que la clave de S ; esta opción puede utilizarse si pocas tuplas de S participan en la relación para evitar excesivos valores nulos en la *foreign key*.

Paso 5: Mapeado de tipos de relaciones M:N binarias Por cada tipo de relación M:N binaria R cree una nueva relación S para representar a R .

Incluya como atributos de la *foreign key* en S las claves principales de las relaciones que representan los tipos de entidad participantes; su combinación formará la clave principal de S .

Incluya también cualesquiera atributos simples del tipo de relación M:N (o los componentes simples de los atributos compuestos) como atributos de S .

No es posible representar un tipo de relación M:N con un atributo de *foreign key* en una de las relaciones participantes (como se hizo para los tipos de relación 1:1 o 1:N) debido a la razón de cardinalidad M:N; se deben crear una relación de relación S separada.

Siempre es posible asignar relaciones 1:1 o 1:N de un modo similar a las relaciones M:N utilizando la metodología de la referencia cruzada (relación de relación).

Esta alternativa es particularmente útil cuando existen pocas instancias de relación, a fin de evitar valores nulos en las *foreign keys*.

En este caso, la clave principal de la relación de relación sólo será una de las *foreign keys* que hace referencia a las relaciones de entidad participantes.

Para una relación 1:N, la clave principal de la relación de relación será la *foreign key* que hace referencia a la relación de la entidad en el lado N .

En una relación 1:1, cada *foreign key* es posible de utilizar como la clave principal de la relación de relación, siempre y cuando no haya entradas nulas en la relación.

Paso 6: Mapeado de atributos multivalor Por cada atributo multivalor A , cree una nueva relación R .

Esta relación incluirá un atributo correspondiente a A , más el atributo clave principal K (como *foreign key* en R) de la relación que representa el tipo de entidad o tipo de relación que tiene A como un atributo.

La clave principal de R es la combinación de A y K ; si el atributo multivalor es compuesto, se incluyen sus componentes simples.

Paso 7: Mapeado de los tipos de relación de grado n Por cada tipo de relación de grado $n > 2$, cree una nueva relación S para representar R .

Incluya como atributos de la *foreign key* en S las claves principales de las relaciones que representan los tipos de entidad participantes.

Incluya también cualesquiera atributos simples del tipo de relación de grado n (o los componentes simples de los atributos compuestos) como atributos de S .

Normalmente, la clave principal de S es una combinación de todas las *foreign keys* que hacen referencia a las relaciones que representan los tipos de entidad participantes.

No obstante, si las restricciones de cardinalidad en cualquiera de los tipos de entidad E que participan en R es 1, entonces la clave principal de S no debe incluir el atributo de la *foreign key* que hace referencia a la relación E' correspondiente a E .

4.3.3.3. Obtención de esquema SQL desde modelo relacional

De acuerdo con la sintaxis de la documentación de MySQL[41], para obtener el esquema SQL del modelo relacional se realizarán los siguientes pasos:

1. Por cada relación (tabla) del modelo relacional se agrega una sentencia CREATE TABLE, seguida del nombre de la relación y de sus atributos encerrados entre paréntesis.
2. Por cada atributo (columna) se crea un campo con su nombre correspondiente y se indica el tipo de dato, para hacerlo más específico, se le puede añadir alguna de las siguientes características:
 - Los atributos que estén obligados a tener un valor se les agrega la palabra NOT NULL.
 - Si la columna tomará un valor por defecto, se agrega la instrucción AUTO_INCREMENT siempre y cuando el tipo de dato sea INTEGER.
 - Si la columna es la que identificará la tabla de forma única en todo el esquema, se debe agregar como llave primaria de la tabla.
3. La llave primaria se agrega como un atributo más con la palabra PRIMARY KEY, seguido del nombre de atributo (encerrada entre paréntesis) que identifica de forma única a la tabla.
4. Todos los atributos de la tabla se encuentran encerrados entre paréntesis separados por comas.
5. Al final de la lista de atributos de la tabla (después del paréntesis) se agrega el motor de almacenamiento: MySQL,ENGINE=InnoDB.

4.3.3.4. Modelo entidad-relación básico a Generic Data Metamodel

Se ha decidido que el algoritmo debe ser basado en *querys* como el de Chebotko; por ello para obtener consultas válidas, debe existir una ruta en el modelo entidad-relación básico que permita llegar desde la entidad n a la entidad m recorriendo las rutas del diagrama.

La figura 4.1 muestra el boceto de la consulta simple a implementar, está inspirada en el trabajo de Chebotko; cada consulta tiene un nombre, la columna *find* representa el atributo a buscar respecto a los diferentes atributos de la columna *given*. En la parte de abajo está un campo para poner una descripción breve de la consulta y botones para limpiar los respectivos campos de la consulta. En la figura 4.1 *User* y *Artifact* son entidades y *id* e *email* son atributos de sus respectivas entidades.

The screenshot shows a user interface titled "Simple Access Patterns". At the top, there are four tabs: Q1, Q2, Q3, and a plus sign. Below the tabs is a "Name :" label followed by a text input field containing "Query name". To the right of the input field are two tables: "Given" and "Find". The "Given" table has one row with "attribute" and "Artifact.id". The "Find" table has two rows: the first with "attribute" and "User.id", and the second with "order by" and "User.email". Below the tables is a "Description (optional):" label followed by a text input field containing "simple description ...". At the bottom are four buttons: "Clear given", "Clear find", "Clear query", and "Delete query".

Figura 4.1: Consulta básica

El algoritmo por entrada tiene una instancia del modelo conceptual entidad-relación básico + consultas básicas como las de la figura 4.1 y de salida una instancia del modelo GDM.

La definición del modelo orientado a documentos será la usada por Alfonso de la Vega en su publicación sobre el GDM.

Se crean primero los elementos de consultas del GDM a partir de las consultas básicas de la instancia del modelo entidad-relación básico como las de la figura 4.1:

1. Por cada consulta se crea un elemento de la clase *query* asignando el nombre de la consulta a realizar.

2. Se crea un elemento *select* con los parámetros de la columna *given* del diagrama entidad-relación básico.
3. Se crea el elemento *from* desde la primera entidad en la que se realiza la consulta.
4. Se recorre la ruta del diagrama entidad-relación básico y se van añadiendo cada entidad del recorrido como un elemento *including* con el nombre de la relación como referencia.
5. Finalmente, se crea un elemento de la clase *boolean expression* para contener la expresión booleana de la consulta.

Para los elementos del *structure model elements* del GDM:

1. Por cada entidad se crea una clase entidad correspondiente en el GDM.
2. Por cada atributo se crea un clase atributo correspondiente y se indica su tipo.
3. Por cada relación se crea una referencia correspondiente.

4.3.3.5. Generic Data Metamodel a modelo lógico NoSQL

De acuerdo con de la Vega[8], las bases de datos orientadas a documentos tienen como objetivo almacenar jerarquías de objetos que son probables que se consultarán juntas.

Estas jerarquías de objetos se conocen como documentos y se agrupan en colecciones; asimismo, una colección almacena documentos de la misma entidad y por lo general un documento está compuesto de pares clave-valor y contiene otros documentos incrustados.

Modelo lógico orientado a documentos En la figura 4.2 se muestra el modelo lógico orientado a documentos propuesto por Alfonso de la Vega.

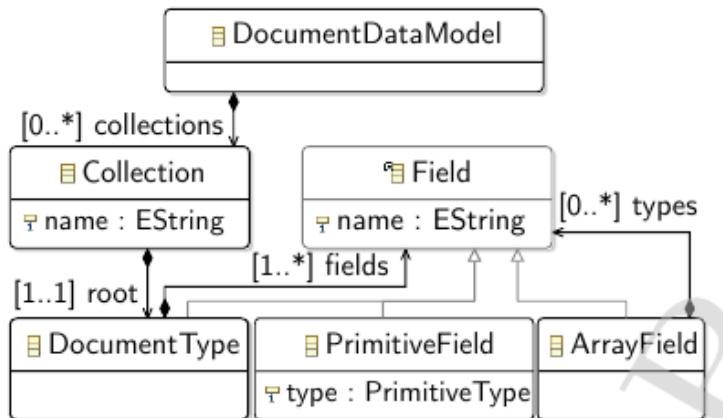


Figura 4.2: Modelo lógico orientado a documentos propuesto por Alfonso de la Vega

A continuación se explica cada clase que conforma el modelo de la figura 4.2:

- Clase *document data model*: está compuesta de n *collections*.
- Clase *collection*: tiene un nombre que la identifica y en una colección se almacenan documentos que, en general, comparten la misma estructura.
- Clase *document type*: define la estructura de los documentos a guardar con un conjunto de instancias de la clase *field*; es la estructura principal que se guardará en cada colección.
- Clase *field*: tiene instancias de las clases *primitive field* (atributos simples), *array field* (más instancias de la clase *field*) o *document type* (otras estructuras de documentos), permitiendo guardar elementos anidados.
- Clase *primitive field*: contiene un tipo primitivo de dato.
- Clase *array field*: permite anidar más instancias de la clase *field*.

La figura 4.3 muestra el *Generic Data Metamodel*, que está compuesto por clases interrelacionadas entre sí con notación UML y consta de dos secciones principales: los elementos de la estructura (*structure model elements*) y el cómo se realizarán las consultas (*access queries elements*).

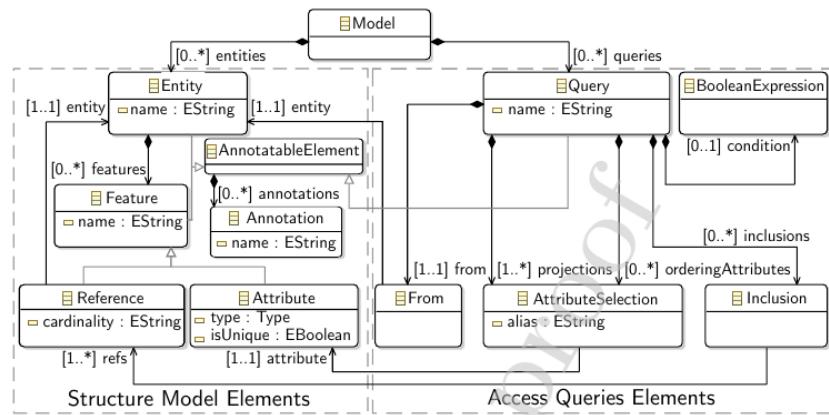


Figura 4.3: Generic Data Metamodel

Sea *Queries* el conjunto de todas las queries q_n en una instancia GDM como la de la figura 4.3:

$$\text{Queries} = \{q_1, q_2, \dots, q_n\} \text{ donde } q_n \in \text{Queries} \text{ y } n = 1, 2, \dots \quad (4.21)$$

Sea $\text{Entities}_{\text{consulted}}$ el conjunto de entidades que son consultadas en una instancia GDM como la de la figura 4.3 (es decir, las entidades que están asociadas a la clase *query* desde un elemento *from*):

$$\text{Entities}_{\text{consulted}} = \{e_{c1}, e_{c2}, \dots, e_{cn}\} \text{ donde } e_{cn} \in \text{Entities}_{\text{consulted}} \text{ y } n = 1, 2, \dots \quad (4.22)$$

Sea Collections el conjunto de colecciones c_n donde cada c_n corresponde a una entidad consultada:

$$\text{Collections} = \{c_1, c_2, \dots, c_n\} \text{ donde } c_n \in \text{Collections} \text{ y } n = 1, 2, \dots \quad (4.23)$$

Por cada c_n en (4.23) se generan los contenidos de cada documento raíz.

La figura 4.4 es un ejemplo del árbol de acceso de la consulta q_4 de la figura 4.5; está formado por el único elemento *from* de la consulta y con los elementos *including* se forma el árbol, donde las aristas son los identificadores de cada *including*.

La figura 4.5 es una consulta en el GDM en modo texto.

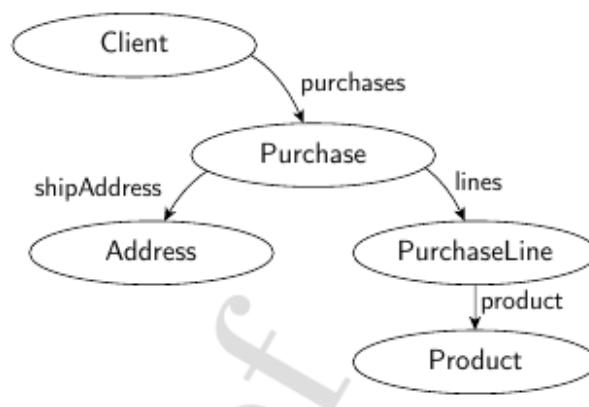


Figura 4.4: Access Tree - Modelo lógico orientado a documentos

```

query Q4_clientPurchasesNearChristmas:
  select client.clientId, client.name,
         client.nationality,
         purchases.purchaseId, purchases.year,
         purchases.month, purchases.day,
         lines.quantity, lines.unitPrice,
         product.name, address.postalCode
  from Client as client
  including client.purchases as purchases,
            client.purchases.lines as lines,
            client.purchases.lines.product as product,
            client.purchases.shipAddress as address
  where client.clientId = "?1" and purchases.year = "?2"
        and purchases.month >= 10
  order by purchases.month, product.price
  
```

Figura 4.5: Access query Q4

Por cada q_n en (4.21) se obtiene primero su único elemento *from* y después todos los elementos *including* para generar un árbol de acceso como el de la figura 4.4.

Se añade al documento raíz todos los atributos simples de su entidad correspondiente; nótese que se está analizando tanto la instancia de la clase *entity* y el árbol de acceso creado.

Si el elemento “ref” de la *entity* está en el árbol de acceso, se necesita saber su cardinalidad y la entidad objetivo de esa referencia.

Si la entidad objetivo está en el árbol de acceso, los datos de esa referencia se incluyen como un subdocumento y se llama recursivamente la función para generar el contenido de este subdocumento; por el contrario, en caso de que la referencia no esté en el árbol de acceso, se incluye la referencia como un identificador si la cardinalidad es 1 o como un arreglo de referencias cuando la cardinalidad es n .

Por último, el autor menciona dos optimizaciones si se quiere reducir el nivel de denormalización que las puede consultar en su investigación[8].

En resumen, el algoritmo quedaría de la forma:

Algorithm 1: Transformación del modelo conceptual GDM al modelo lógico orientado a documentos

Input : una instancia del modelo GDM, gdm
Output: un modelo lógico orientado a documentos, ddm

```
1 mainEntities  $\leftarrow gdm.queries.collect((q)|q.from);$ 
2 foreach me  $\in mainEntities$  do
3   collection  $\leftarrow newCollection();$ 
4   collection.name  $\leftarrow me.name;$ 
5   accessTree  $\leftarrow allQueryPaths(me, gdm.queries);$ 
6   collection  $\leftarrow populateDocumentType(collection.root, accessTree);$ 
7   ddm.collections.add(collection);
8 end
```

Donde la función `populateDocumentType()` es otro algoritmo de la forma:

Algorithm 2: Generar el contenido de un *DocumentType* dado un árbol de acceso

Input : Un “document type”, *dt*
Output: un nodo del arbol de acceso

```
1 nodeAttributes ← node.entity.features.select(f|f.isTypeOf(Attribute))
2 nodeReferences ←
    node.entity.features.select(f|f.isTypeOf(Reference))
3 foreach attr ∈ nodeAttributes do
4     pf ← newPrimitiveField()
5     pf.name ← attr.name
6     pf.type ← attr.type
7     dt.fields.add(pf)
8 end
9 foreach ref ∈ nodeReferences do
10    targetNode ← node.arcs.find(a|a.name = ref.name).target
11    if exists(targetNode) then
12        baseType ← newDocumentType()
13        populateDocumentType(baseType, targetNode)
14    else
15        baseType ← newPrimitiveField()
16        baseType.type ← findIdType(ref.entity)
17    end
18    baseType.name ← ref.name
19    if ref.cardinality == 1 then
20        dt.field.add(baseType)
21    else
22        arrayField ← newArrayField()
23        arrayField.type ← baseType
24        dt.fields.add(arrayField)
25    end
26 end
```

4.3.3.6. Modelo lógico NoSQL a modelo físico en MongoDB

De acuerdo con la documentación de MongoDB[40], se puede escribir un *script* para crear un esquema de base de datos en un archivo con extensión .js y con el contenido de los datos en formato JSON válido; para generar el esquema completo se deben seguir los siguientes pasos:

1. Crear un archivo con extensión .js.
2. Abrir el archivo con los permisos de lectura y escritura para poder añadir contenido.
3. Agregar al archivo una línea con la instrucción `conn = new Mongo();`.

-
4. Agregar una nueva línea al archivo .js con la instrucción `db = conn.getDB('DB_NAME');` esto creará una base de datos con el nombre que se le indique en caso de que no exista.
 5. Agregar al archivo la instrucción `db.dropDatabase()` para eliminar todos los registros del esquema de la base de datos en caso de que existan y así evitar datos erróneos.
 6. Agregar una línea con la instrucción `db.createCollection('COLLECTION_N');` por cada entidad del GDM, donde `COLLECTION_N` es sustituido por el nombre de la entidad.
 7. Agregar una línea con el nombre de un elemento de la primera colección seguida de un signo igual (=) y un carácter de llave ‘{’.
 8. Agregar una línea por cada atributo simple de la entidad del GDM con la instrucción `“ATTR_NAME”:“DEFAULT_VALUE”`, sustituyendo `ATTR_NAME` por el nombre de cada atributo y `DEFAULT_VALUE` por un valor por defecto que corresponda al tipo de dato del atributo.
 9. Agregar una línea con el carácter ‘}', para indicar el cierre del objeto JSON.

Los pasos del 1 al 5 son suficientes para generar un esquema de base de datos vacío en MongoDB, solamente debe ejecutarse el archivo con el comando `mongo file.js` y se puede comprobar que todo es correcto entrando a la consola de MongoDB y revisar los esquemas disponibles con el comando `show dbs;`.

Los incisos posteriores generan un objeto de prueba para cada colección de datos, esto se hace principalmente para visualizar un elemento en cada colección; de lo contrario solamente se apreciarían colecciones vacías en MongoDB y estas pueden almacenar todo tipo de información siempre y cuando se trate de un objeto JSON válido.

Finalmente, una vez verificada que la estructura es correcta, se deben truncar todas las colecciones y de esta manera tener un esquema de la base de datos NoSQL sin registros y empezar a almacenar los documentos; esto se logra con la instrucción `db.COLLECTION_NAME.remove({})` sustituyendo `COLLECTION_NAME` por cada uno de los nombres de las colecciones.

4.4. Conclusiones

La factibilidad operativa permite predecir si es posible poner en marcha el sistema propuesto, aprovechando todos los beneficios que se ofrecen a todos los usuarios involucrados en ello.

La herramienta va dirigida a estudiantes de nivel medio o nivel superior que tengan un primer acercamiento a los modelos de bases de datos entidad-relación o relacional desde un enfoque conceptual, como es el caso en ESCOM, en la asignatura de Base de Datos; el sistema propuesto cuenta con una interfaz intuitiva para que el usuario final, el estudiante, visualice, cree y edite un diagrama ER con las opciones que la herramienta le brinde de manera comprensible.

Por lo explicado anteriormente, el sistema propuesto tiene una alta probabilidad de aceptación por parte de los usuarios finales al encontrarse en un entorno en el que se trabaja con *software* continuamente, además del beneficio que aporta al plan de estudios actual al ofrecer una forma práctica de ver aplicado los conceptos adquiridos en la asignatura de Bases de Datos, el cual solo contempla un alcance hasta la normalización de bases de datos relacionales y tener una introducción a los modelos no relacionales NoSQL.

Un estudiante que ha cursado dicha asignatura se dará cuenta que el tiempo disponible durante el curso es limitado por la cantidad de módulos que pretende cubrir y en muchas ocasiones los docentes deben prescindir de ciertos temas para completar el temario.

Con la implementación de la propuesta de solución, los estudiantes que cursen la asignatura de Bases de Datos tendrán la oportunidad de conocer una opción más en cuanto a tecnologías de almacenamiento de datos para implementar en sus propios sistemas; de igual manera, es posible que los impulse a solicitar la apertura de una asignatura optativa sobre los modelos de datos NoSQL si hay interés por estos temas.

Se concluye que el sistema propuesto tendrá un uso en la institución y un potencial beneficio para los estudiantes y los involucrados.

A continuación están las respuestas (con preguntas incluidas) de la sección 4.2:

1. ¿El sistema contribuye a los objetivos generales de la organización?

Sí, ya que la misión en ESCOM es formar profesionales líderes en saberes de ingeniería, tecnología y ciencias de la computación con una visión globalizada; así como contribuir con investigación y desarrollo tecnológico para el crecimiento del país; por lo que la propuesta de solución contribuye directamente a la visión de ESCOM.

2. ¿Se puede implementar el sistema dentro del cronograma y el presupuesto utilizando las tecnologías actuales?

Es posible, tal como se muestra en la sección 4.2.1, se cuenta con las tecnologías para el desarrollo del producto y el esfuerzo es ajustado para el equipo de desarrollo, pero queda en los límites del tiempo establecido en el cronograma; además de ser desarrollado con una metodología ágil que ofrece productos funcionales por cada iteración.

3. ¿Se puede integrar el sistema con otros sistemas que se utilizan?

Sí, el sistema es integrable a otros sistemas por estar disponible en la web; por ejemplo, puede integrarse directamente en la asignatura de Bases de Datos como una opción más para el modelado de diagramas ER con la ventaja de tener el acercamiento a los modelos no relacionales de manera práctica.

Como se ha mencionado en el documento, se ha considerado Scrum como metodología para desarrollar la propuesta de solución, porque el proyecto requiere de entregas regulares de su avance para realizar modificaciones con ayuda de la retroalimentación constante del cliente, en este caso las directoras del proyecto.

Esto otorga beneficios como poder responder con flexibilidad, adaptación a los requisitos de cliente, estrechar la relación con el mismo y mantener al equipo motivado con pequeñas entregas funcionales del producto; además, tomando en cuenta la experiencia del equipo, esta forma de trabajo permite mostrar avances funcionales en el producto en un periodo de tiempo corto para realizar una evaluación y en caso de ser requerido se sugieran cambios.

De la sección de algoritmos, para la validez estructural de un diagrama entidad-relación se hace uso del trabajo de Dullea [7] para el análisis de las relaciones unarias y binarias; además, se proponen restricciones para las entidades, atributos y relaciones del modelo entidad-relación básico.

Del mapeo modelo entidad-relación básico al modelo relacional se ha optado por usar la propuesta de Elmasri [13]; para la obtención del esquema SQL se ha decidido parsear el modelo lógico relacional obtenido por el algoritmo anterior y se indica en la sección 4.3.3.3 las reglas básicas para implementar el algoritmo en el DBMS MySQL.

Para el algoritmo de mapeo entre el modelo entidad-relación y el GDM se ha optado por proponer consultas similares al trabajo de Chebotko[4] para generar las consultas del GDM; asimismo, las relaciones en el modelo entidad-relación son referencias en el GDM y una consulta válida en el modelo entidad-relación es toda consulta que permita llegar a un atributo de una entidad Y desde una entidad X recorriendo las rutas del diagrama entidad-relación.

Para generar el modelo lógico orientado a documentos a partir del GDM se hace uso del algoritmo propuesto por Alfonso de la Vega, en el que se generan árboles de acceso de las consultas en el GDM para generar los documentos anidados en el modelo lógico orientado a documentos.

Finalmente, para obtener el esquema de sentencias en MongoDB se da una serie de pasos en la sección 4.3.3.6 para implementar el algoritmo.

Capítulo 5

Diseño del sistema

De acuerdo con Pressman [55], el objetivo del diseño del *software* es aplicar un conjunto de principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad; la meta del diseño es crear un modelo de *software* que implantará correctamente todos los requerimientos del usuario y causará placer a quienes lo utilicen; los diseñadores del *software* deben elegir entre muchas alternativas de diseño y llegar a la solución que mejor se adapte a las necesidades de los participantes en el proyecto.

En este capítulo se muestra el diseño del sistema; se ve el funcionamiento de los módulos del sistema, así como el diagrama de clases y el esquema de la base de datos.

5.1. Diagrama de clases

De acuerdo con Visual Paradigm [70], el diagrama de clases es un modelo conceptual que permite visualizar la estructura de las clases de un sistema, sus atributos, métodos y las relaciones entre objetos.

El diagrama de clases de la propuesta de solución se muestra en la figura 5.1; en la cual se aprecia la clase dominante *usuario* y las demás clases dependen de esta.

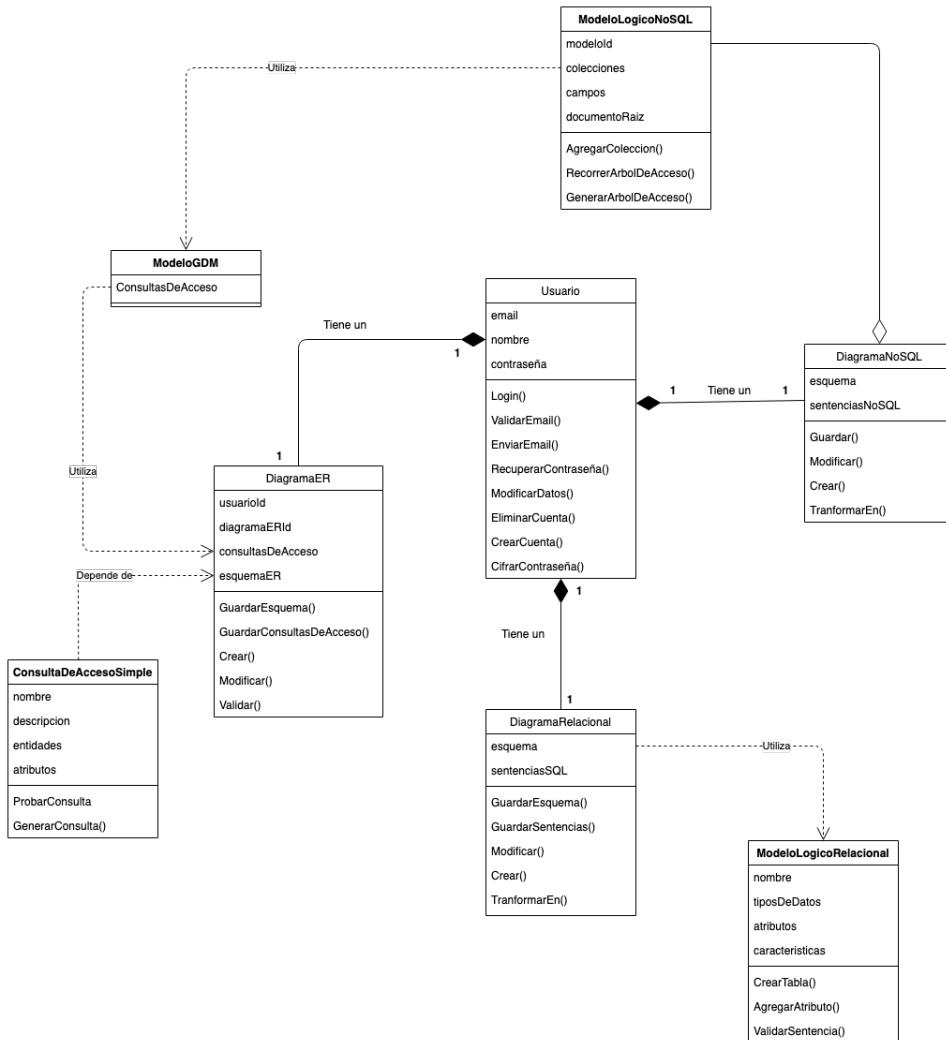


Figura 5.1: Diagrama de clases

Es posible interpretar el significado del diagrama de clases de la figura 5.1 leyendo los puntos de la siguiente manera:

- El usuario es la clase principal.
- Existe una agregación entre Usuario y DiagramaER.
- El ModeloGDM es parte del ModeloLogicoNoSQL; el ModeloGDM puede existir por sí mismo.
- El GDM depende del DiagramaER; sin embargo, el DiagramaER no depende del GDM.
- La ConsultaDeAccessoSimple depende del Diagrama ER.
- El ModeloLogicoRelacional depende del DiagramaRelacional.

5.2. Diagrama de la base de datos

Como MongoDB es una base de datos NoSQL orientada a documentos, los datos se almacenan en documentos JSON; teniendo en cuenta el diagrama de clases de la figura 5.1, se aprecia que las relaciones son por composición de 1 a 1 y la clase dominante es Usuario, es decir, las clases DiagramaER, DiagramaRelacional y DiagramaNoSQL no pueden existir si no existe una clase Usuario asociada a ellas.

Por esto la mejor manera de almacenar los datos es en documentos anidados, donde el documento principal es el usuario y este contendrá los documentos de las otras clases; en la figura 5.2 se representa la relación por composición de las clases de la figura 5.1.

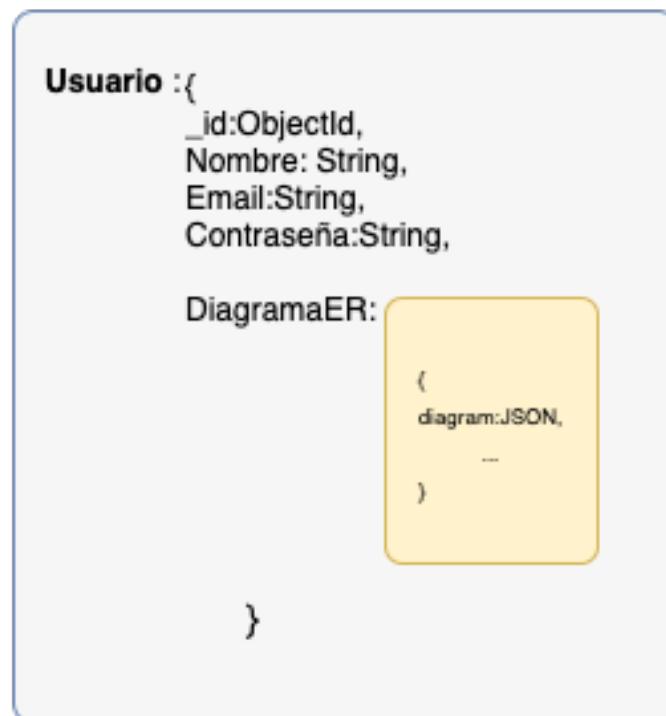


Figura 5.2: Diagrama de la base de datos.

En la figura 5.2 se pone en práctica el comportamiento del diagrama de clases de la figura 5.1, porque al eliminar un usuario internamente se eliminan los diagramas asociados a este; es por este motivo que el usuario debe estar 100 % seguro al momento de eliminar su cuenta, ya que será imposible recuperar los diagramas que haya realizado un vez completada la acción.

5.3. Arquitectura del sistema

De acuerdo con Pressman [55] en su sección diseño de la arquitectura, el diseño arquitectónico de un *software* es la representación de los componentes o módulos del sistema y cómo estos interactúan entre sí; en esta sección se muestra el diseño arquitectónico que usa el sistema como parte de su implementación.

5.3.1. Arquitectura cliente-servidor

Pressman menciona en su trabajo [55] que la arquitectura cliente-servidor es un modelo de diseño del *software* compuesta por 2 partes, el cliente que solicita información por medio de una petición y el servidor quien se encarga de proveer los datos que le son requeridos; las aplicaciones web utilizan frecuentemente este modelo teniendo como clientes a un navegador web y el servidor es quien provee la información que se solicite, pudiendo atender a más de un cliente al mismo tiempo.

La figura 5.3 describe la arquitectura cliente-servidor del sistema.

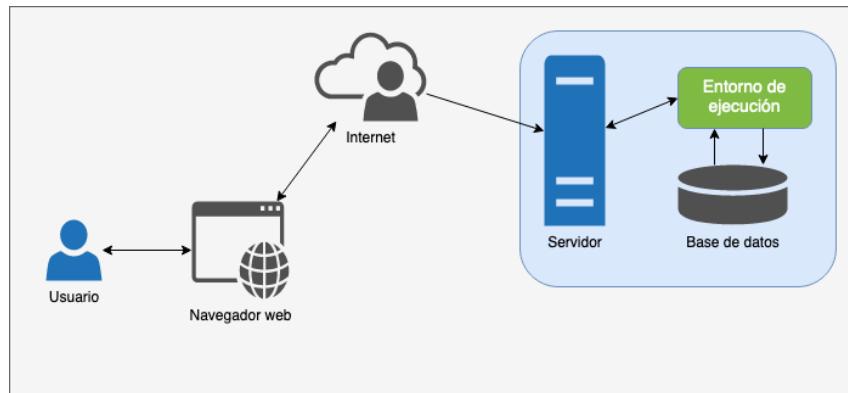


Figura 5.3: Arquitectura cliente-servidor.

En la figura 5.3 se observa que para poder hacer uso del sistema, el usuario debe contar con un dispositivo capaz de ejecutar un navegador web como una computadora o un *smartphone*; el servidor será una computadora que se encuentra en un instancia de la nube que contiene los requisitos necesarios para el funcionamiento del sistema como son: el sistema operativo GNU/Linux, el servidor web, las aplicaciones en Flask y Nuxt, conexión a Internet y una configuración de red que permita la comunicación con la base de datos de MongoDB.

5.3.2. Patrón de diseño MVC

Según Dragos-Paul Pop menciona en su trabajo [71], el patrón MVC fue concebido por primera vez en la década de los 70 en la empresa Xerox y su principal propósito era cerrar la brecha entre el modelo mental del usuario y el modelo digital de las aplicaciones en una computadora.

Más tarde, el paradigma MVC fue descrito por Krasner y Pope[72] donde destacan que se pueden obtener enormes beneficios si se piensa en la modularidad

de las aplicaciones; el patrón se divide en 3 partes principalmente: el modelo, la vista y el controlador.

- El modelo: es el dominio principal de la aplicación, es quien se encarga del manejo de los datos y toda la lógica del negocio, es decir, no tiene que tener conocimiento sobre la interfaz gráfica, ya sea una aplicación web, de escritorio o móvil; idealmente el modelo debe ser independiente de la plataforma en que se ejecuta.
- La vista: contiene todos los elementos visibles al usuario y a diferencia del modelo, esta no debe tener conocimiento de la lógica del negocio de la aplicación; sus responsabilidades deben limitarse a definir la estructura y apariencia de los datos presentados en la interfaz gráfica.
- El controlador: es el intermediario entre la vista y el modelo, es quien gestiona el flujo de la aplicación por medio de la comunicación entre los otros 2 componentes.

La figura 5.4 describe la implementación del patrón MVC en la propuesta de solución junto a las tecnologías utilizadas en cada componente del paradigma; como se aprecia la vista queda a cargo del *framework* Nuxt quien es el encargado del renderizado de la interfaz gráfica para el usuario mostrando elementos como el formulario de *login* y el diagramador entidad-relación.

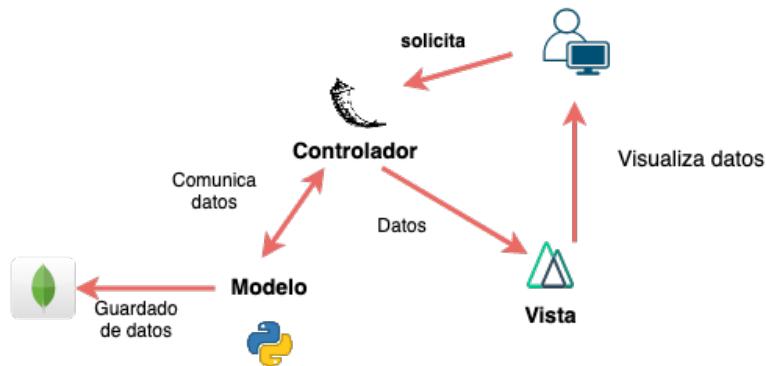


Figura 5.4: Esquema MVC

De igual manera, el *framework* flask será el encargado del rol de controlador; llevará a cabo la comunicación de los datos que el usuario proporcione en la vista para transformarlos en algo que el modelo pueda entender como el guardado del objeto JSON del diagrama entidad-relación para ser almacenado en la base de datos; finalmente, el modelo queda a cargo del lenguaje Python para el manejo de los datos y es el encargado de mantener la comunicación con la base de datos, pudiéndose apoyar en bibliotecas de terceros como el propio *framework* flask para transformar los datos de la lógica de negocio y convertirlos en algo que la vista pueda mostrar al usuario.

5.4. Conclusiones

La arquitectura cliente-servidor es apta para un desarrollo ágil de la propuesta de solución; asimismo, la arquitectura elegida permite al equipo usar lenguajes de programación que estén consolidados en el desarrollo web o en el diagramado, como es el caso de los lenguajes elegidos en la sección [3.4](#).

El patrón MVC está bien integrado con el *framework* Nuxt y Flask, permitiendo que desde etapas tempranas del desarrollo de la propuesta de solución haya un *live demo* que puede revisar en la sección [6](#).

Finalmente, el diagrama de clases y el diagrama de la base de datos también están integrados, porque desde el diseño conceptual es fácil usar el concepto de la anidación de datos y es solo cuestión de implementarlo en MongoDB.

Capítulo 6

Desarrollo del sistema

En este capítulo se muestra cómo se implementa cada uno de los objetivos 1.4 de la propuesta de solución, explicando cada función principal del *front end*, del *back end*, y explicando los algoritmos de la sección 4.3.3.

Asimismo, también se muestran fragmentos del código fuente del *front end*, *back end* y algoritmos.

Para cumplir con los objetivos 1.4, la propuesta de solución usa para la autenticación de usuarios *web tokens*; se dispone del código fuente y un *live demo* de la aplicación.

El *front end* (Nuxt/JS/GoJS) lo puede consultar en [repositorio front](#), el *back end* (Python/Flask) en [repositorio back](#), el documento de TT (Latex) en [repositorio documento](#) y el *live demo* está disponible en [dirección](#).

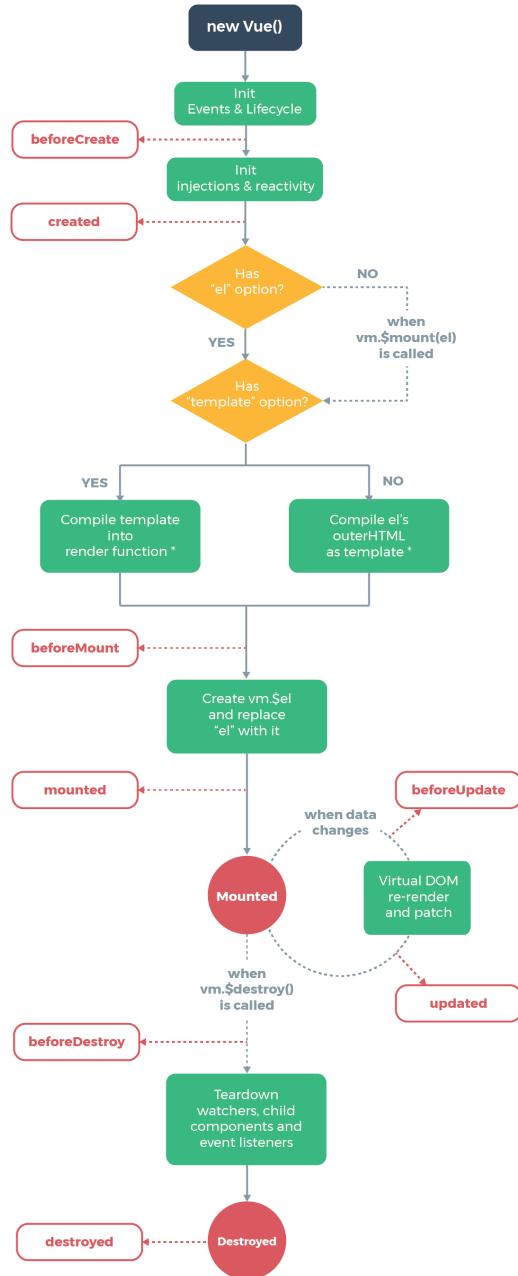
El *live demo* del *front end* está usando el *deploy* automático en Netlify de la rama master y el código de desarrollo está en la rama dev o en las ramas *feature*.

Asimismo, en la página de GitHub del proyecto de *front end* es posible encontrar de manera breve un resumen del propósito de la aplicación web, estado actual del proyecto, tecnologías usadas y cosas que faltan por implementar.

El *live demo* del *back end* está usando el *deploy* automático en Heroku de la rama master y el código de desarrollo está en las ramas *feature*.

6.1. Front end

Como se especificó en la sección 3.3.5, el *web framework* a utilizar es Nuxt; Nuxt usa un “ciclo de vida” para cada componente Vue y se muestra en la figura 6.1.



* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

Figura 6.1: Ciclo de vida de Vue/Nuxt

Por la reactividad de Nuxt/Vue y la necesidad de implementar GoJS por sobre Nuxt/Vue, todas las instancias de GoJS se crean en el *hook mounted* y se restaura el estado de cada componente usando Vuex; se eliminan todos los escuchas necesarios y se limpian los componentes de ser necesario en el *hook beforeDestroy*.

6.1.1. Pantallas del front end

De acuerdo con el sitio de Netlify [73], Netlify ofrece funcionalidad HTTPS para todos sus sitios, tiene una mitigación activa en contra de ataques DDoS, todo su tráfico está cifrado en redes TLS y los *tokens* están cifrados.

De acuerdo con el sitio de JWT [74], un JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una forma compacta y autónoma para transmitir información de forma segura entre las partes como un objeto JSON.

Esta información se puede verificar y confiar porque está firmada digitalmente; los JWT se firman usando un clave denominada “secreto” (con el algoritmo HMAC) o un par de claves pública/privada usando RSA o ECDSA.

Aunque los JWT se pueden cifrar para proporcionar “secreto” entre las partes, los *jwt web tokens* se enfocan en *tokens* firmados; los *tokens* firmados pueden verificar la integridad de los reclamos que contiene, mientras que los *tokens* cifrados ocultan esos reclamos de otras partes; cuando los *tokens* se firman utilizando pares de claves públicas / privadas, la firma también certifica que solo la parte que posee la clave privada es la que la firmó.

La figura 6.2 muestra la pantalla de bienvenida de la aplicación, donde el mensaje de iniciar sesión o registrarse es visible si el usuario no ha iniciado sesión.

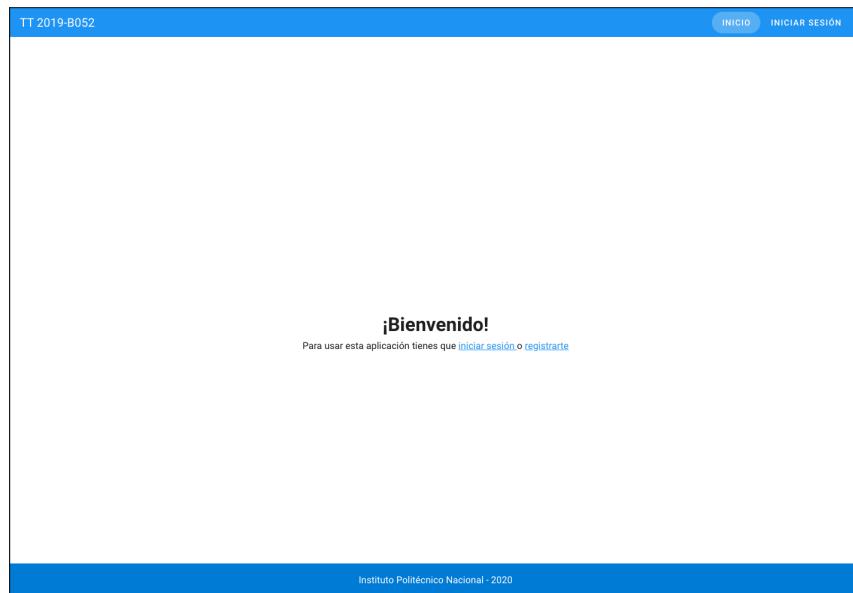


Figura 6.2: Pantalla de bienvenida

La figura 6.3 muestra la pantalla de inicio de sesión de la aplicación, donde se pide el correo y la contraseña para iniciar sesión.

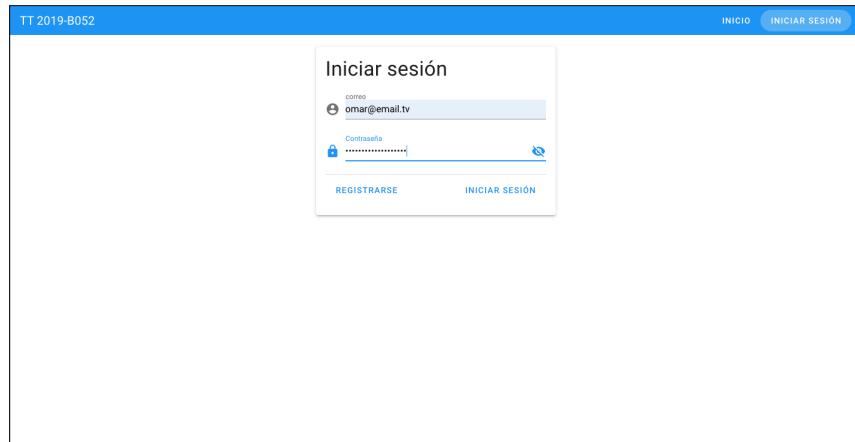


Figura 6.3: Pantalla de login

La figura 6.4 muestra el formulario de registro de la aplicación; para la validación de los campos se ha usado Vuelidate y muestra mensajes de error en caso de que algún campo esté rellenado incorrectamente.

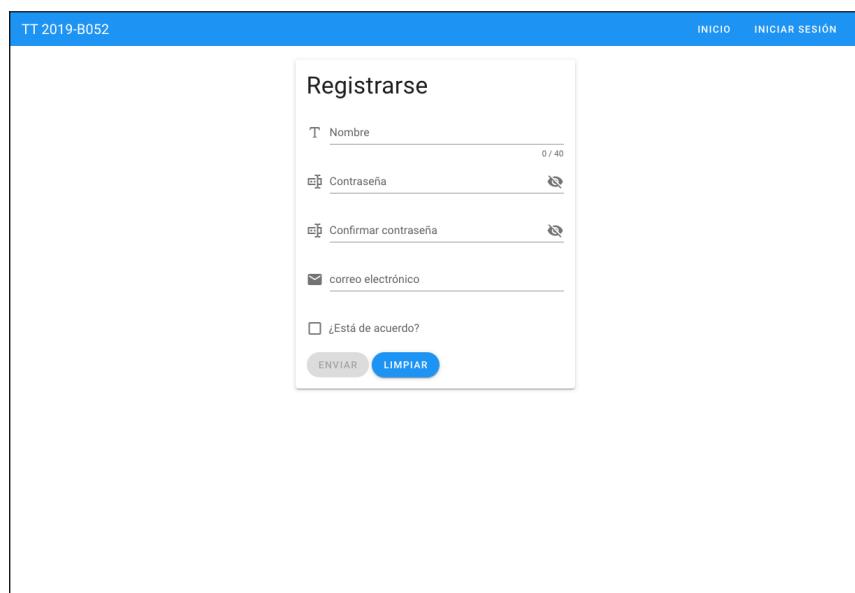


Figura 6.4: Pantalla de alta de usuario

La figura 6.5 muestra la vista del diagramador entidad-relación básico, del lado izquierdo están los botones para guardar y cargar un diagrama en formato .json válido para la aplicación; en la segunda columna está la zona de diagramado para crear un diagrama entidad-relación y del lado derecho está la paleta de elementos donde están los elementos de un diagrama entidad-relación básico; en la parte de abajo está el JSON equivalente del diagrama que está en la zona de diagramado y a su derecha está un menú para modificar algunas propiedades de los elementos del diagramador.

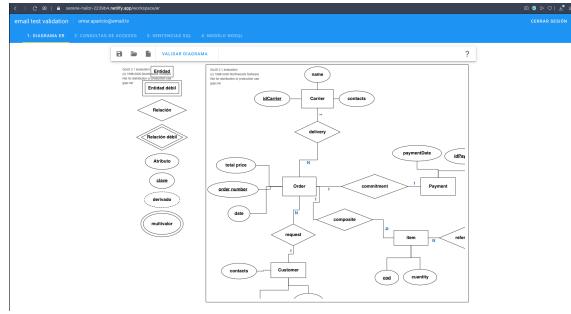


Figura 6.5: Pantalla del diagramador entidad-relación

Las figuras 6.2, 6.3, 6.4 y 6.5 son capturas de pantalla del prototipo de la propuesta de solución, como es de notar el prototipo permite editar, guardar y cargar un diagrama entidad-relación básico.

Pantallas del diagramador entidad-relación

La figura 6.6 muestra lo que visualiza el usuario una vez que concluyó su registro o inició sesión; se puede apreciar las herramientas para crear/editar un diagrama entidad-relación, además de la zona de trabajo conocida como *canvas*.

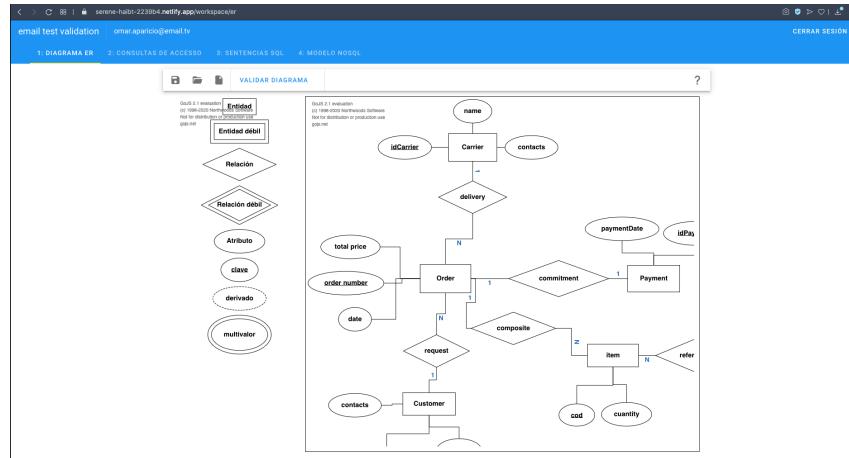


Figura 6.6: Pantalla del diagramador entidad-relación

La figura 6.7 muestra los errores del diagrama entidad-relación después que el usuario hace click en el botón “validar diagrama”, en caso que el diagrama no cumpla con las reglas mencionadas en la sección 4.3.3.1.

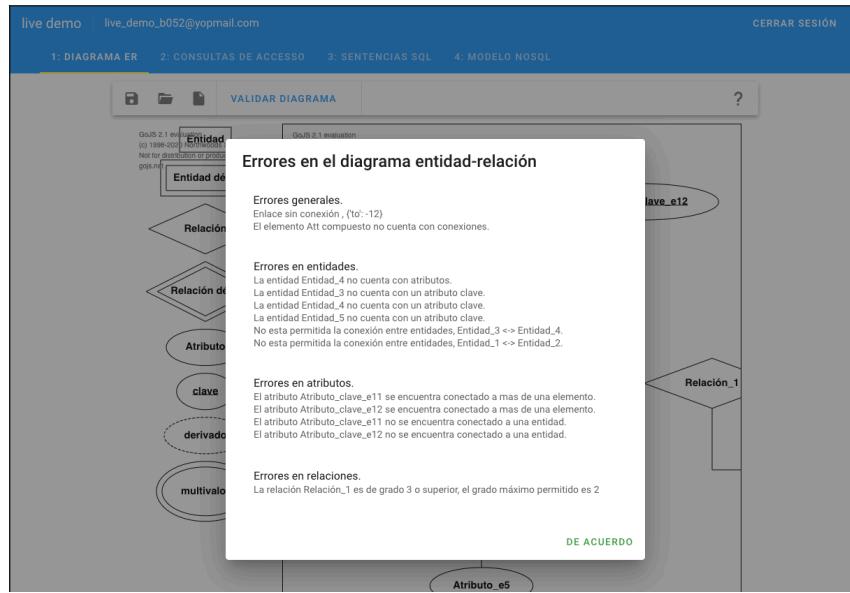


Figura 6.7: Pantalla de errores al validar un diagrama.

La figura 6.8 muestra el modal que el usuario visualiza después de hacer click en el botón “validar diagrama” y este cumple con todas las reglas de validación estructural.

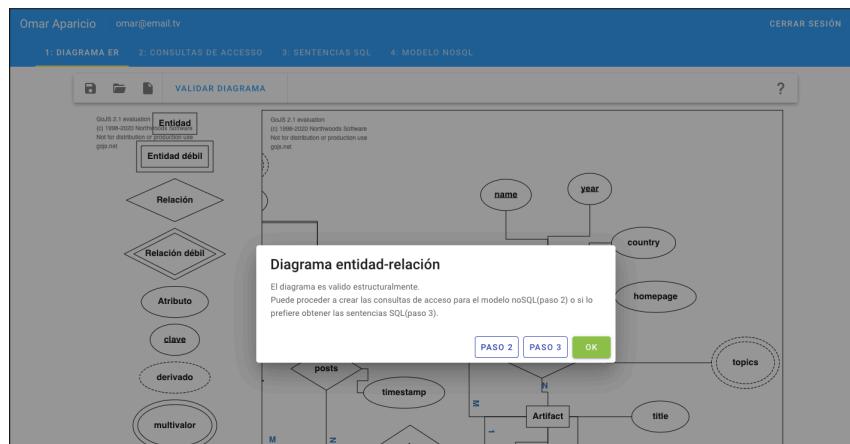
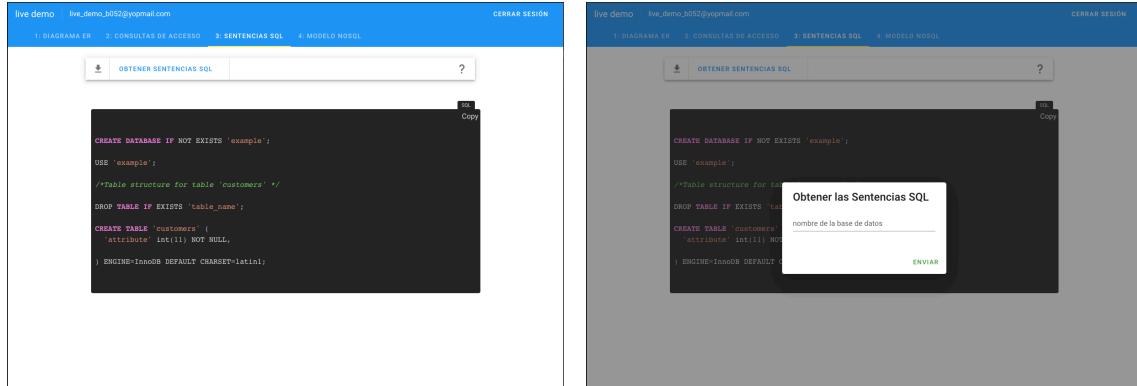


Figura 6.8: Pantalla de errores al validar un diagrama.

Pantallas de las sentencias SQL equivalentes

La figura 6.9 muestra las pantallas del módulo de obtención de las sentencias equivalentes del diagrama que el usuario generó en la figura 6.6; este paso solo es posible después de haber pasado por el proceso de validación para el diagrama entidad-relación. Del lado derecho en la figura 6.9a se aprecia el código en el lenguaje SQL necesario para crear la base de datos relacional en el sistema gestor de base de datos MySQL, y del lado izquierdo en la figura 6.9b se muestra el modal que el usuario visualiza al hacer click en el botón “Obtener sentencias SQL” en el cual deberá colocar el nombre que tendrá la base de datos a generar.

Si el usuario necesita exportar el script de sql a un archivo, cuenta con un botón en la parte superior para realizar esta acción.



- (a) Sentencias SQL equivalentes al diagrama ER. (b) Modal para nombrar la base de datos sql.

Figura 6.9: Pantallas de las sentencias SQL equivalentes al diagrama ER.

Pantallas de las consultas de acceso

La figura 6.10 es lo que el usuario visualiza al ingresar a este módulo, este paso solo es posible después de haber pasado por el proceso de validación para el diagrama entidad-relación. Es aquí donde puede agregar las consultas de acceso que desea que sea utilicen en el proceso de transformación al modelo noSQL teniendo del lado izquierdo el diagrama ER en modo de solo lectura y al hacer click derecho en los atributos clave de una entidad visualizará un menú con las opciones para generar dicha consulta.

Las consultas de acceso pueden ser tantas como se necesiten, como se aprecia en la figura 6.11, es importante mencionar que puede agregar tantos elementos al apartado “Respecto al atributo” como quiera pero debe existir al menos un elemento en el apartado “Encontrar” para que la consulta tenga sentido.

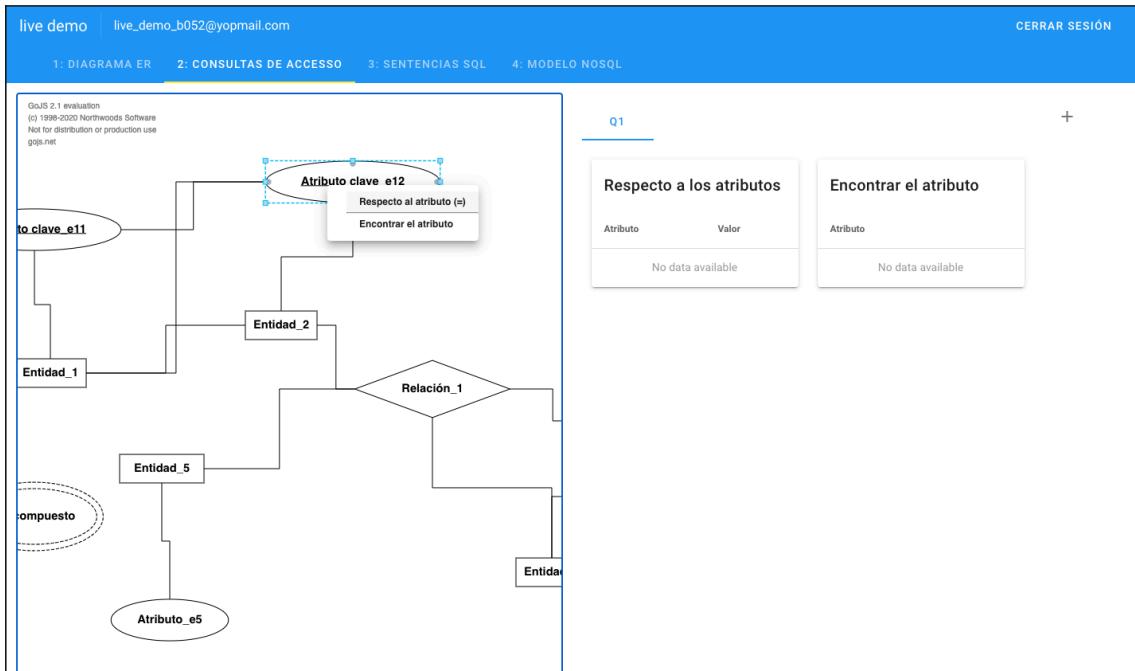


Figura 6.10: Pantalla para agregar una consulta de acceso.

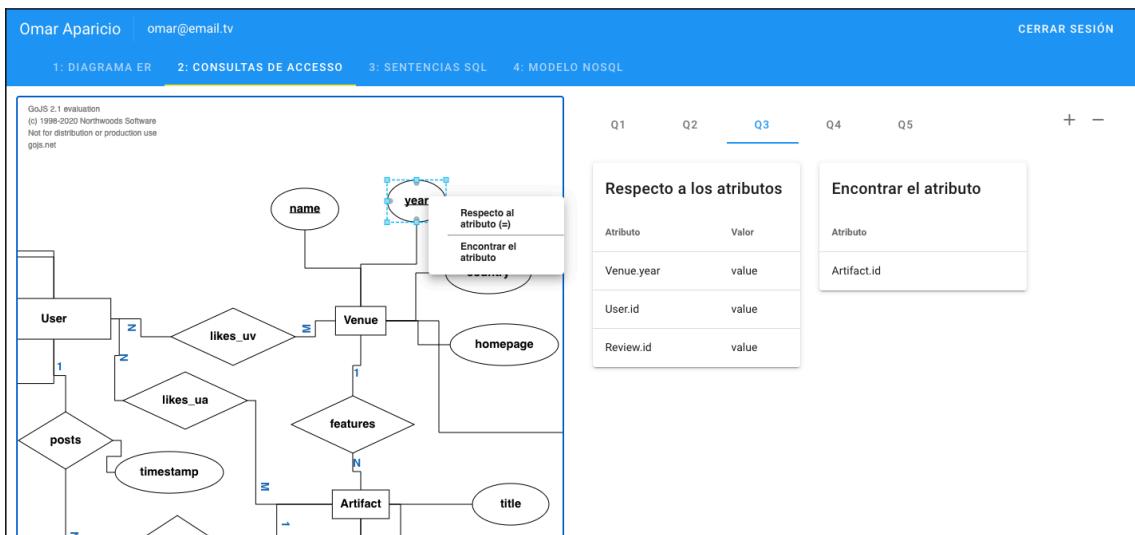


Figura 6.11: Pantalla con multiples consultas de acceso.

6.1.2. Diagramado del modelo entidad-relación

De acuerdo con Shahzad [75], GoJS es una biblioteca de JavaScript para implementar diagramas interactivos y visualizaciones en plataformas y navegadores web modernos.

Código fuente 6.1: Diagrama GoJS

```

1 // Definimos el diagrama
2 this.myDiagram = $(
3     go.Diagram,

```

```

4      'myDiagramDiv',
5      {
6          grid: $(go.Panel, 'Grid'),
7          'draggingTool.dragsLink': false,
8          'draggingTool.isGridSnapEnabled': true,
9          'linkingTool.isUnconnectedLinkValid': false,
10         'linkingTool.portGravity': 10,
11         'relinkingTool.isUnconnectedLinkValid': false,
12         'relinkingTool.portGravity': 10,
13         'relinkingTool.fromHandleArchetype': $(go.Shape, 'Diamond', {
14             segmentIndex: 0,
15             cursor: 'pointer',
16             desiredSize: new go.Size(8, 8),
17             fill: 'tomato',
18             stroke: 'darkred'
19         })
20     }
21 )
22
23 /*****Enlaces*****/
24 this.myDiagram.linkTemplate = $(
25     go.Link, // definición de un enlace
26     {
27         selectionAdorned: true,
28         selectionAdornmentTemplate: linkSelectionAdornmentTemplate,
29         layerName: 'Foreground',
30         reshapable: true,
31         routing: go.Link.AvoidsNodes,
32         curve: go.Link.JumpOver
33     },
34     $(
35         go.TextBlock, // la cardinalidad del enlace
36         {
37             text: '',
38             textAlign: 'center',
39             font: 'bold 14px sans-serif',
40             stroke: '#1967B3',
41             editable: true,
42             segmentOffset: new go.Point(0, -10),
43             segmentOrientation: go.Link.OrientUpright
44         },
45         new go.Binding('text', 'cardinality').makeTwoWay()
46     )
47 )
48 /*****Paleta*****/
49 model: new go.GraphLinksModel([
50     {
51         type: 'entity',
52         text: 'Entidad',
53         figure: 'Rectangle',
54         dataType: 'varchar',

```

```

55         fill: 'white'
56     },
57     {
58         type: 'weakEntity',
59         text: 'Entidad débil',
60         figure: 'FramedRectangle',
61         dataType: 'varchar',
62         fill: 'white'
63     },
64     {
65         type: 'relation',
66         text: 'Relación',
67         figure: 'Diamond',
68         dataType: 'varchar',
69         fill: 'white'
70     }
71 )

```

Como se especificó en la sección 3.3.8, el diagrama se implementó en GoJS y parte del código fuente en 6.1 muestra una instancia de cómo se configura el *canvas* del diagramador, los enlaces entre nodos y la definición de la paleta de elementos; como se nota todos son objetos *json* desde su creación y para la manipulación de los objetos se hace uso de las bibliotecas integradas de GoJS para *json*.

6.1.3. Sentencias SQL

De acuerdo con los objetivos específicos de la sección 1.4, se implementó la obtención de sentencias SQL desde el modelo entidad-relación. Para revisar el algoritmo asociado, por favor, revise la sección 6.3.2.

Código fuente 6.2: Diagrama GoJS

```

1  getSentencesSQL() {
2      const diagram = this.currentDiagram
3      this.$store
4          .dispatch('axiosER/convertToSQL', {
5              diagram,
6              dbName: this.db_name
7          })
8          .then((response) => {
9              this.sentences = response.data
10         })
11         .catch((error) => {
12             if (error.response.status === 500) {
13                 this.$notify.warning(
14                     '¡Algo ocurrió! No fue posible obtener las sentencias SQ del
15                     → diagrama, intente más tarde.'
16                 )
17             }
18         })
19     }

```

```

18     this.convertToSQLDialog = false
19   },
20   downloadScript() {
21     const scriptData = encodeURIComponent(this.sentences)
22     this.urlFile = `data:text/plain;charset=utf-8,${scriptData}` // 
23     → application/sql
24     const dbname = this.db_name ? this.db_name : 'tt2019-B052'
25     this.scriptName = dbname + '.sql'
26     this.$notify.success('Archivo descargado. ')
27   }

```

El fragmento de código 6.2 obtiene el evento *listener* del botón obtener sentencias SQL y realiza una llamada por medio de Axios al *back end*. Posteriormente, recibe el esquema de sentencias SQL y lo muestra al usuario para que pueda descargarlo y probarlo en MySQL.

6.1.4. Modelo conceptual NoSQL

El módulo del modelo conceptual NoSQL está implementado en Nuxt, haciendo uso de Vuex para guardar las entidades del GDM.

El algoritmo y el código fuente de la transformación del modelo entidad-relación básico a las entidades del GDM lo puede consultar en .

6.1.5. Modelo lógico y físico NoSQL

El módulo del modelo lógico y físico de un modelo de datos orientado a documentos permite ver gráficamente la estructura del modelo lógico por medio de la biblioteca de GoJS.

Respecto a los algoritmos usados, el algoritmo para la generación de sentencias de MongoDB desde la definición de un modelo lógico orientado a documentos revise la sección y el algoritmo usado para la generación del diagrama se muestra a continuación en el código fuente 6.3 que como entrada recibe un XML serializado por PyEcore de una instancia del modelo lógico orientado a documentos y de salida genera los nodos y links necesarios para diagramar con GoJS.

Código fuente 6.3: Generación de los nodos y links del modelo orientado a documentos para el diagramador NoSQL

```

1      with open(input_file) as xml_file:
2
3          data_dict = xmltodict.parse(xml_file.read())
4          xml_file.close()
5
6          # generate the object using json.dumps()
7          # corresponding to json data
8
9          #json_data = json.dumps(data_dict)
10

```

```

11     collections =
12     ↪  data_dict['documentDataModel:DocumentDataModel']['collections']
13     nodedataArray = []
14     linksdataArray = []
15
16     for collection in collections:
17         # Obtenemos el documento raíz
18         root = collection['root']
19         collection_data = {}
20         collection_data['name'] = collection['@name']
21         collection_data['subtype'] = 'Collection'
22         collection_data['key'] = generateKey()
23         stack = LifoQueue(maxsize = 1000)
24         collection_data['items'] = []
25
26         # Cada elemento del documento raíz puede ser de tipo
27         # → primitivo, documento o un arreglo
28         for field in root['fields']:
29
30             # Si es un tipo primitivo
31             if ( field['@xsi:type'] ==
32                 'documentDataModel:PrimitiveField'):
33
34                 ↪  collection_data['items'].append(getPrimitiveField(field))
35                 print(field)
36
37             # Si es un documento
38             if ( field['@xsi:type'] == 'documentDataModel:Document'):
39                 # Añadimos la referencia del documento
40                 collection_data['items'].append({
41                     'name': field['@name'],
42                     'type': field['@xsi:type'],
43                     'subtype': 'Document',
44                     'parentKey': collection_data['key'],
45                     '#key': generateKey(),
46                     'figure': 'Rectangle',
47                     'color': '#FFD700'
48                 })
49                 field['parentKey'] = collection_data['key']
50                 stack.put(field)
51
52                 print(field)
53
54             # Si es un arreglo
55             if ( field['@xsi:type'] ==
56                 'documentDataModel:ArrayField'):
57
58                 # Añadimos la referencia del arreglo
59                 collection_data['items'].append({
60                     'name': field['@name'],

```

```

57             'type': field['@xsi:type'],
58             'subtype': 'Array',
59             'parentKey': collection_data['key'],
60             #'key': generateKey(),
61             'figure': 'Hexagon',
62             'color': '#6ea5f8'
63         }
64     )
65     field['parentKey'] = collection_data['key']
66     stack.put(field)
67     print(field)

68
69     while not stack.empty():
70         child = stack.get()

71         # Si es un documento
72         if ( child['@xsi:type'] == 'documentDataModel:Document'):
73             # Añadimos la referencia del documento
74             items = getItemsDocument(child)
75             new_key_parent = generateKey()
76             nodedataArray.append({
77                 'name': child['@name'],
78                 'type': child['@xsi:type'],
79                 'items': items,
80                 'subtype': 'Document',
81                 'parentKey': child['parentKey'],
82                 'key': new_key_parent,
83                 'figure': 'Rectangle',
84                 'color': '#FFD700'
85             }
86         )
87         for i in child['fields']:
88             i['parentKey'] = new_key_parent
89             stack.put(i)
90         print(field)

91
92         # Si es un arreglo
93         if ( child['@xsi:type'] ==
94             'documentDataModel:ArrayField'):
95             items = getItemsArray(child)
96             # Añadimos la referencia del arreglo

97             new_key_parent = generateKey()
98             nodedataArray.append({
99                 'name': child['@name'],
100                'type': child['@xsi:type'],
101                'items': items,
102                'subtype': 'Array',
103                'parentKey': child['parentKey'],
104                'key': new_key_parent,
105                'figure': 'Hexagon',
106            }

```

```

107                     'color': '#6ea5f8'
108                 }
109             )
110             child['type'][ 'parentKey' ] = new_key_parent
111             stack.put( child[ 'type' ] )
112
113             # Obtenemos contenidos de cada hijo recursivamente
114
115
116
117             nodedataArray.append( collection_data )
118
119
120             for node in nodedataArray:
121                 if node[ 'subtype' ] != 'Collection':
122                     linksdataArray.append( {
123                         'to': node[ 'parentKey' ] , 'from': node[ 'key' ]
124                     } )
125
126             # Write the json data to output
127             # json file
128             data = {}
129             json_nodedataArray = json.dumps( nodedataArray )
130             json_linksdataArray = json.dumps( linksdataArray )
131             data[ "nodedataArray" ] = nodedataArray
132             data[ "linkdataArray" ] = linksdataArray
133             ofile = "venuesLalo.json"
134             with open( ofile, "w" ) as json_file:
135                 json_file.write( json.dumps( data ) )
136                 json_file.close()

```

6.2. Back end

De acuerdo con el sitio de Heroku [76], la plataforma ofrece distintos mecanismos para la seguridad de los proyectos que aloja, uno de estos mecanismos es el uso por defecto del protocolo HTTPS, que asegura el cifrado de los datos en la Internet.

De igual manera, ofrece un constante escaneo de las aplicaciones en búsqueda de vulnerabilidades para mitigar los ataques DDoS, además de utilizar la infraestructura de la empresa Amazon, las cuales se encuentran acreditadas por diversos estándares de seguridad; esto da la confianza para que la aplicación Flask se ejecute de manera segura y tener salvaguardados los datos de los usuarios.

De acuerdo con el sitio Pallets Projects [77], flask es un *framework* web escrito en python. Está diseñado para que el desarrollo de una aplicación web sea rápida y sencilla sin imponer ninguna dependencia o diseño sobre el proyecto, porque depende del desarrollador implementar las bibliotecas o herramientas que desee utilizar y, al contar con una comunidad amplia, hay una gran cantidad de extensiones para ampliar sus funcionalidades.

6.2.1. El api flask

Como se mencionó en la sección 3.3.10, flask es un *microframework*, pero es posible extender sus capacidades por medio de bibliotecas. Para el desarrollo de la api de servicios web se utiliza la biblioteca *flask restplus*, que de acuerdo con su documentación oficial [78], es una extensión de flask con soporte para el desarrollo de API REST de manera rápida, fomenta buenas prácticas con una configuración mínima y proporciona una colección de decoradores y herramientas para describir un API de manera sencilla y exponer su documentación correctamente usando Swagger.

Para el desarrollo de la aplicación, se creó la estructura mostrada en la figura ?? donde la carpeta *config* contiene un archivo con el nombre *app.conf* con las variables de configuración de la aplicación, como son la llave secreta para validar el JWT, credenciales para conexión a la base de datos, un usuario de pruebas y características de la interfaz swagger. El archivo *database.py* es una clase *singleton* para mantener una sola instancia de conexión a la base de datos para toda la aplicación.

```

→ api-tt-2019-b052 git:(QA) tree

.
├── Pipfile
├── Pipfile.lock
├── Procfile
├── README.md
└── apis
    ├── __init__.py
    ├── diagram.py
    ├── login.py
    ├── relational_model.py
    └── user.py
├── config
    └── app.conf
        └── database.py
├── server.py
└── service
    ├── __pycache__
    │   └── relational.cpython-37.pyc
    └── relational.py
    └── util.py

4 directories, 15 files
→ api-tt-2019-b052 git:(QA)

```

Figura 6.12: Estructura del proyecto backend.

Para lograr la implementación de flask con restplus es necesaria la siguiente configuración: se deben importar de la biblioteca *flask* en el archivo principal del proyecto las clases *Flask*, *jsonify*, *request*, *make_response*, *Response* y, al mismo tiempo, la configuración descrita en el archivo *config/app.conf* para tener disponible en toda la aplicación la conexión a la base de datos como la llave secreta del *token* de autenticación y las credenciales del servidore SMTP para el envío de correo. Dado como resultado el fragmento de código 6.4.

Código fuente 6.4: Clase principal del api flask

```

1  from flask import Flask, jsonify, request, make_response, Response
2  from flask_cors import CORS
3  from flask_pymongo import PyMongo
4  import os
5  from apis import api
6  from config.database import initialize_db
7
8  app = Flask(__name__)
9  app.config.from_pyfile(os.path.join(".", "config/app.conf"),
10    ↳ silent=False)
11
12 mongo = PyMongo(app)
13 CORS(app, resources={r'/*': {'origins': '*'}})
14 initialize_db(app)
15 api.init_app(app)
16
17 if __name__ == '__main__':
18     app.run(debug=True)

```

6.2.2. Los servicios web

Para tener una aplicación flask básica ejecutándose, por favor, revise la sección [6.2.1](#), pero para poder exponer servicios web y que estos puedan ser consultados por otras aplicaciones son necesarios pasos adicionales descritos en esta sección. Para poder registrar un servicio en la aplicación flask primero se debe configurar la documentación de swagger. Se recomienda en un archivo separado mantener la importación de los distintos namespaces de los servicios que desea exponer y es la clase **API** de restplus quien se encargará de registrarlos en el api y mostrarlos al usuario como se muestra en el fragmento de código [6.5](#), también es posible apreciar en este fragmento de código la configuración del *header* de autenticación para poder hacer uso de los servicios así como la infamación del api, como el nombre de los desarrolladores, su email de contacto, la versión del api, si utiliza algún mecanismo de seguridad en la autenticación, etc.

Código fuente 6.5: Registro de los namespaces del api.

```

1  from flask_restplus import Api
2  from .user import api as ns_user
3  from .login import api as ns_login
4  from .diagram import api as ns_diagram
5  from .relational_model import api as ns_relational
6
7  #Authorization
8  authorizations = {
9      'Bearer Auth': {
10          'type': 'apiKey',
11          'in': 'header',
12          'name': 'Authorization'
13      }
14  }

```

```

15
16 api = Api(
17     title='API TT 2019-B052',
18     version='0.10.0',
19     description='api de servicios para el trabajo termnal 2019-B052
20     ↵ <style>.models {display: none !important}</style>',
21     contact="Omar Aparicio Quiroz, Eduardo Acosta Martinez",
22     contact_email="omaraparicio07@gmail.com, fx2013630461@gmail.com",
23     security='Bearer Auth',
24     authorizations=authorizations,
25     RESTPLUS_MASK_SWAGGER=False
26     # All API metadatas
27 )
28
29 api.add_namespace(ns_user)
30 api.add_namespace(ns_login)
31 api.add_namespace(ns_diagram)
32 api.add_namespace(ns_relational)

```

Es recomendable tener separados cada uno de los namespaces en archivos separados, esto hace más legible la integración de nuevos servicios al api. Como se aprecia la figura 6.12, todos estos archivos se encuentran en la carpeta *apis/*, tanto la configuración como cada uno de los servicios expuestos asociados por contexto, es decir, todos los servicios relacionados con los datos del usuario se encuentran en *apis/user.py*.

Para crear un recurso de un nuevo servicio web debemos agregar el decorador `@api.route()` antes de una clase o método para mapear la ruta por la que el usuario podrá acceder a ese servicio, la clase debe tener como parámetro la clase *Resource* de restplus para identificarlo como un elemento del api. Una de las buenas prácticas que implementa la herramienta restplus es la limitante que por cada clase solamente es posible tener definido un método por cada tipo de petición HTTP, es decir, por cada clase solo se permite una método `def get(self):` que atenderá la petición http que tenga en el *header request method* la opción *GET*.

Asimismo, se implementó un decorador personalizado para validar que las peticiones que tengan que hacer uso de los datos del usuario o la base de datos cuenten con el *token* de autenticación en los *headers* de la petición. Este decorador se muestra en el fragmento de código 6.6, una vez creado el decorador basta con colocarlo antes de la definición de la clase o método que requiera de su uso.

Código fuente 6.6: Decorador personalizado para validar el token en los headers de la petición HTTP.

```

1 def token_required(f):
2     @wraps(f)
3     def decorated(*args, **kwargs):
4         try:
5             token = request.headers['Authorization']
6         except:
7             response = {

```

```

8         'message': 'Token no encontrado en la petición'
9     }
10    return response,401
11
12    if "Bearer " in token:
13        token = token[token.index(' ')+1:]
14
15    try:
16        jwt.decode(token, current_app.config['SECRET_KEY'])
17    except:
18        response = {
19            'message': 'Token inválido'
20        }
21        return response,401
22
23    return f(*args, **kwargs)
24
25 return decorated

```

6.2.3. Servicios para login

Este *namespace* tiene los servicios relacionados con la autenticación del usuario. Como es de esperarse, este *namespace* no requiere del uso del decorador 6.6, ya que es aquí donde un usuario previamente registrado generará un *token* de autenticación haciendo uso de su correo electrónico y contraseña. De la misma forma, si este quiere cambiar su contraseña puede hacerlo. La figura 6.13a muestra el uso del método *PUT* para el cambio de contraseña, proporcionando un correo electrónico registrado en la base de datos. Una vez que el cambio de contraseña sea efectuado, el usuario recibirá un correo de confirmación como se muestra en la figura 6.13b.

The figure consists of two parts. Part (a) shows a screenshot of a POST /login endpoint in a tool like Postman or Swagger. The payload is defined as:

```

{
  "email": "string",
  "password": "string"
}
  
```

Part (b) shows an email from "Scheme Converter Tool <omaraparicio07@gmail.com>" with the subject "Hola! Bienvenido de vuelta". The email body contains:

Hola! Bienvenido de vuelta

¿Has cambiado tu contraseña??

usuario : user_001
nueva contraseña : Pas5w0rd*

- (a) Método PUT del servicio para cambio de contraseña de un usuario. (b) Correo de confirmación para cambio de contraseña.

El servicio de login regresa al usuario un token de autenticación como se muestra en la figura 6.14. El usuario debe almacenar el token para poder proporcionarlo al api en cada petición que requiera de permisos para realizar las operaciones.

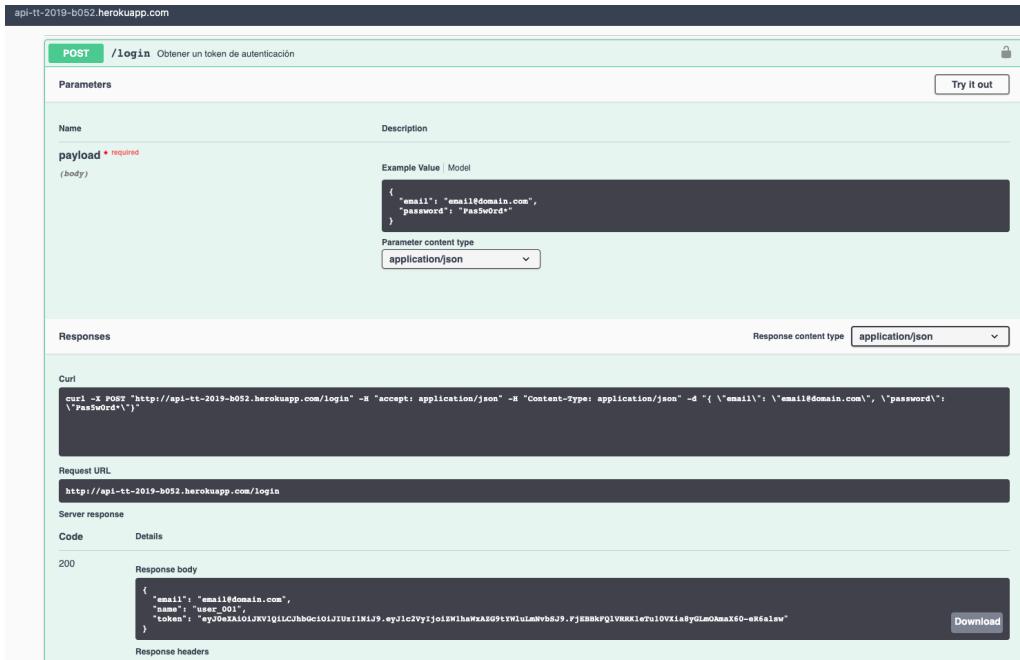


Figura 6.14: Generación del token de autenticación.

6.2.4. Servicios para el usuario

Los métodos o funciones implementadas en este *namespace* constan de un CRUD (*Create, Read, Update, Delete*) para el manejo de usuarios. Es aquí donde se permite el registro de un usuario nuevo, así como la consulta de todos los usuarios registrados en la base de datos, así como la consulta de un usuario en específico. Para limitar los datos que el servicio acepta en una petición se puede hacer uso del método `model` que proporciona la clase *Namespace* de `restplus`, como se aprecia en el fragmento de código 6.7 el modelo de un usuario conta de los campos `_id`, `username`, `name`, `email`, `password`, `diagram` de los cuales `_id` y `diagram` no son obligatorios para el registro de un nuevo usuario. Toda la lógica de esta operaciones se encuentra definida en el archivo `apis/user.py` dando como resultado la figura 6.15 en la cual se muestran los servicios disponibles para los usuarios.

Código fuente 6.7: Modelo para registrar un usuario.

```

1 user = api.model('User', {
2     '_id': fields.String(required=False, readonly=True),
3     'username': fields.String(required=False, readonly=True,
4         description="a username", example="username"),
5     'name': fields.String(required=True ),
6     'email': fields.String(required=True, example="email@domain.com"),
7     'password': fields.String(required=True, example="P4ssw0rd*"),
8     'diagram': fields.String(example="{}", readonly=True),
9 })

```

The screenshot shows the `/user` endpoint in a Swagger UI. The `POST` method is used to create a new user, requiring a JSON payload with fields `name`, `email`, and `password`. The `GET` method retrieves a user by their email address.

Figura 6.15: Pantalla de los servicios relacionados a un usuario.

6.2.5. Servicios para el diagrama

Este namespace tiene los servicios relacionados al diagrama entidad-relación del usuario. Por razones de seguridad este namespaces si requiere del uso del decorador [6.6](#), para asegurar que solamente usuarios registrados en la base de datos puedan consultar su diagrama o en su defecto guardar un nuevo diagrama en su cuenta, es en estos casos donde entra en uso la configuración de restplus para la solicitud del token de autenticación de usuario para poder hacer uso de estos servicios como se muestra en la figura [6.16a](#).

(a) Solicitud de token de autenticación. This part of the figure shows a modal dialog titled "Available authorizations" with a "Bearer Auth (apiKey)" section. It asks for a "Name Authorization" and "Value" (which is a placeholder for a token key). Below the modal is a list of API endpoints under the "login" namespace, including `POST /LogIn` for obtaining a token.

(b) Token no encontrado en la petición. This part shows a screenshot of a browser developer tools' network tab. A request to `http://127.0.0.1:3000/Espacio` resulted in a 401 Unauthorized status code. The response body indicates "Error: UNAUTHORIZED" and "message": "Token no encontrado en la petición".

En caso de no proporcionar un token valido la operación no podrá ser completada y el api mostrara un mensaje como el de la figura [6.16b](#), de igual forma que al registrar un usuario estos servicio requieren que en la petición HTTP se encuentren ciertos parámetros. En este caso solamente se requiere que se encuentre un objeto json para guardar un diagrama, el cual será el mismo que se devolverá al usuario al consultar el servicio `/diagram` por medio de una petición `GET`.

6.2.6. Servicio para el modelo relacional

Este *namespace* tiene los servicios que tienen una relación con el modelo relacional, como son la obtención de las sentencias SQL o la validación del diagrama entidad-relación. Para estos servicios también es necesario un *token* de autenticación para validar un diagrama ER, así como la obtención de las sentencias SQL equivalentes. En ambos casos, los servicios se deben hacer por una petición del método *POST* para que el protocolo *HTTPS* mantenga cifrada la información que se transmite por la red.

La figura 6.17 muestra el servicio de validación de diagrama ER, con las reglas que este debe cumplir para ser considerado válido en su estructura. En caso de que el diagrama no cumpla con alguna de estas reglas, el servicio le muestra al usuario una lista de los errores presentes en el diagrama, tal como se aprecia en la figura 6.18.

POST /relational/validate Método para validar la estructura del diagrama ER con los siguientes criterios:

General :

- No pueden existir elementos sin conexión ✓
- No pueden existir enlaces sin conexión ✓

Entidades :

- Debe tener al menos un atributo ✓
- Debe tener al menos un atributo clave ✓
- La clave primaria puede ser simple o compuesta(tener mas de un atributo clave) ✓
- La clave primaria no es una clave foránea ✓
- La clave primaria debe ser un atributo asociado a la entidad ✓
- Dos entidades solo pueden conectarse entre si mediante una relación ✓

Atributos :

- Solo se puede asociar a un único atributo o a una entidad ✓
- Puede ser del tipo compuesto, derivado, multivalor, clave ✓
- No puede conectarse a una relación ✓
- Los atributos compuestos y derivados solo pueden estar asociados a una entidad ✓

Relaciones :

- Solo pueden existir entre entidades ✓
- El grado máximo de participación es dos
- Una relación puede ser unaria ✓
- No se permiten relaciones ternarias o de grado n ✓

Parameters

Name	Description
payload • required (body)	<p>Edit Value : Model</p> <pre>diagram ▾ { db_name* string diagram* string example: db_name.sql }</pre>

Execute Clear Cancel

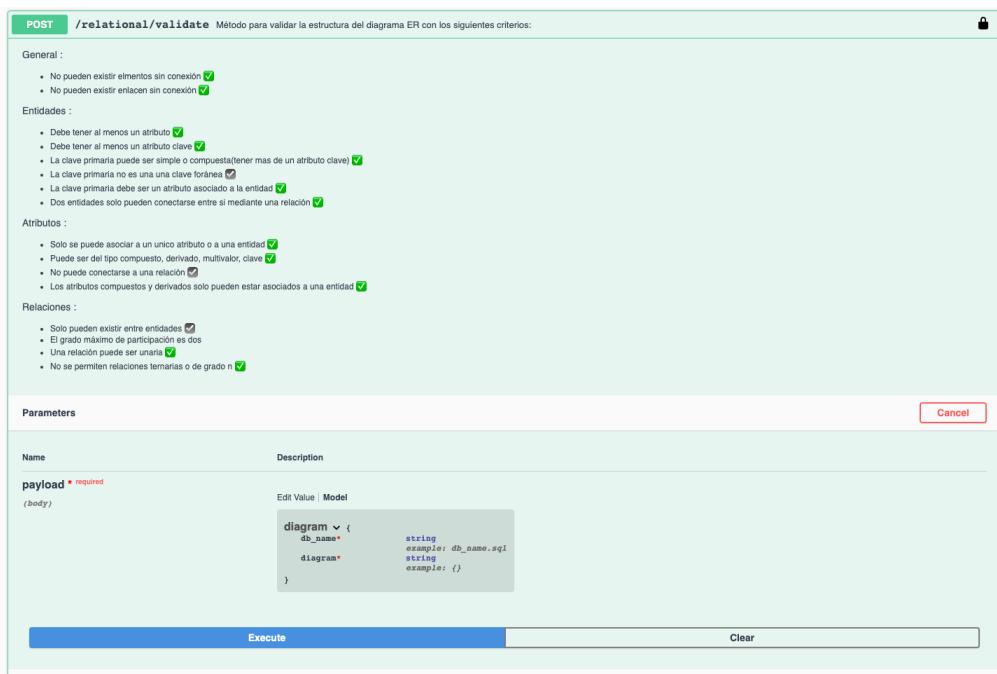


Figura 6.17: Pantalla del servicio para validar un diagrama ER con las reglas en la descripción.



Figura 6.18: Respuesta para un diagrama ER no valido.

Para obtener las sentencias SQL al diagrama ER se creó el servicio `/relational` definido en el archivo `apis/relacional.py`. Este servicio solo debe ser usado para un diagrama ER valido. Este servicio no ejecuta ninguna validación sobre el diagrama recibido porque ese no es su propósito, para eso se tiene el servicio de validación previamente mencionado.

Para conocer las sobre el proceso de obtención de las sentencias SQL equivalentes consulte la sección [6.3.2](#). Una vez realizado el proceso el servicio muestra al usuario un script de SQL como se muestra en la figura [6.19](#) que puede descargar o copiar para ejecutar en el sistema gestor de MySQL.

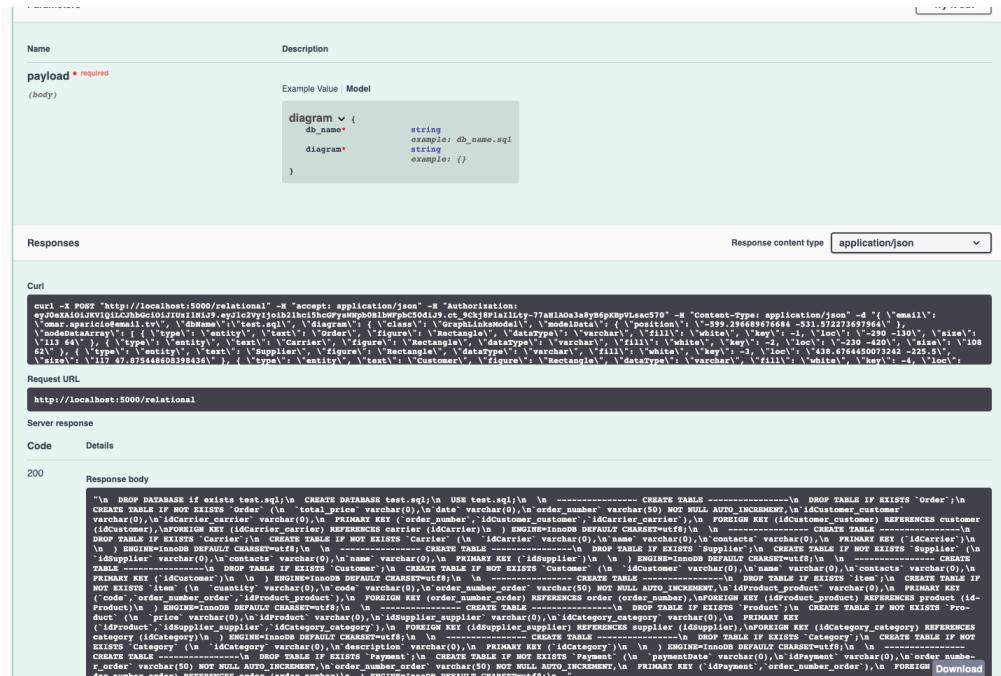


Figura 6.19: Respuesta con las sentencias SQL equivalentes al diagrama ER.

6.3. Algoritmos

Como se mencionó en la sección 4.3.3, un algoritmo es cualquier procedimiento computacional definido que toma algún valor, o conjunto de valores, como entrada y produce algún valor, o conjunto de valores, como salida; por lo tanto, un algoritmo es una secuencia de pasos computacionales que transforman la entrada en la salida.

En esta sección se describen cómo se implementan los algoritmos de la sección 4.3.3, empezando con el algoritmo para la validación estructural diagrama entidad-relación; se sigue con algoritmo para el mapeo modelo entidad-relación a relacional; después está la obtención de esquema SQL desde modelo relacional, el mapeo modelo entidad-relación a *Generic Data Metamodel*; el mapeo del *Generic Data Metamodel* a modelo lógico NoSQL y se finaliza con el algoritmo para el modelo lógico NoSQL a modelo físico en MongoDB.

6.3.1. Validación estructural del diagrama entidad-relación

Para cumplir con los objetivos del proyecto es importante que la validación estructural de la sección 4.3.3.1 sea el primer paso a realizar, ya que no tiene sentido obtener el modelo NoSQL de un diagrama no válido; de igual manera, las sentencias SQL equivalentes pueden ser generadas, pero sin una validación estructural no se asegura la coherencia de la estructura de la base de datos relacional resultante.

Validaciones generales Para este punto es necesario comprobar que todos los elementos del diagrama se encuentran conectados entre sí y no existen conexiones libres; para eso debemos recorrer todas las conexiones en la propiedad *linksDataArrray* del objeto *json diagram* y comprobar que existen los elementos *'from'* y *'to'* en cada uno de los nodos de esta lista. En caso de encontrarse ambas propiedades en alguno de los nodos, indica que es una conexión libre y se agrega a una lista de conexiones libres que será mostrada al usuario para corregir esos detalles.

Recorrer la propiedad *nodeDataArrray* del objeto *json diagram* para obtener los identificadores del cada elemento del diagrama y buscarlo en la lista *linksDataArrray* permite asegurar que todos los elementos se encuentran conectados entre sí. En caso de no encontrar el identificador del elemento en la lista de conexiones, se agrega a una lista de elementos desconectados que será mostrada al usuario para corregir esos

detalles.

Algorithm 3: Lista de conexiones libres en el diagrama.

Entrada: una instancia del diagram ER en formato *json*, *diagram*
Salida : una lista de conexiones libres, *unconnectedLinks*

```
1 linksList ← diagram['linksDataArrray']
2 foreach link ∈ linksList : do
3   | if !(link['from']! = " & link['to']! = ") : then
4   |   | unconnectedLinks ← link
5   |   end
6 end
```

Algorithm 4: Lista de elementos desconectados.

Entrada: una instancia del diagram ER en formato *json*, *diagram*
Salida : una lista de elementos sin conexión, *unconnectedElements*

```
1 linksList ← diagram['linksDataArrray']
2 itemsList ← diagram['nodeDataArrray']
3 foreach item ∈ itemsList : do
4   | foreach link ∈ linksList : do
5   |   | if !(link['from'] == itemKey OR link['to'] == itemKey) : then
6   |   |   | unconnectedElements ← item
7   |   |   end
8   |   end
9 end
```

Validaciones en las entidades Para este paso es necesario comprobar que toda entidad en la propiedad *nodeDataArrray* del objeto *json* *diagram* cuente con al menos un elemento atributo del tipo clave o del tipo simple que no se encuentre conectado a otra entidad de forma directa. Para esto se realiza un proceso similar a la búsqueda de elementos desconectados, con la excepción que se requieren los identificadores de los elementos de tipo atributo, en caso de no encontrar elementos del tipo atributo o un atributo clave, la entidad se agrega a una lista de entidades sin atributos, de la misma forma, si se encuentran conexiones directas entre atributos, se genera una lista con las entidades conectadas, porque ambas listas serán mostradas al usuario

para corregir esos detalles.

Algorithm 5: Lista de entidades sin atributos.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de entidades sin atributos o sin clave primaria,
entitiesWithoutAttrs

```
1 entitiesWithAttrs ← diagram['nodeDataArrray']
2 foreach entity ∈ entitiesWithAttrs : do
3     if notentity['attributes'] : then
4         | entitiesWithoutAttrs ← entitys, entityT
5     end
6     if notentity['primaryKey'] : then
7         | entitiesWithoutAttrs ← entitys, entityT
8     end
9 end
```

Algorithm 6: Lista de entidades conectadas directamente.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de entidades conectadas directamente,
connectionBetweenEntities

```
1 linksList ← diagram['linksDataArrray']
2 entitiesKeyList ← diagram['nodeDataArrray']
3 foreach link ∈ linksList : do
4     if link['from'] ∈ entitiesKeyList & link['to'] ∈ entitiesKeyList : then
5         | connectionBetweenEntities ← entitys, entityT
6     end
7 end
```

Validaciones en los atributos Los atributos solo pueden tener una conexión a una entidad o a una relación, no se pueden asociar a más de un elemento o entre sí mismos de forma directa. Para comprobar estos puntos es necesario inspeccionar la propiedad *nodeDataArrray* del objeto *json* *diagram* en conjunto con la propiedad *linksDataArrray* para determinar las conexiones entre elementos. En caso de encontrar una conexión directa entre atributos, agregarlo a una lista de conexión entre atributos. De la misma forma, si el identificador del atributo es encontrado en más de una ocasión en los nodos de *linksDataArrray*, entonces indica que el atributo está conectado a más de un elemento, por lo que se agrega a una lista de atributo con multiconexiones para mostrar ambas listas al usuario y que pueda

corregir esos detalles.

Algorithm 7: Lista de atributos con conexiones multiples.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de atributos con conexiones multiples,
attrMulticonnections

```
1 linksList ← diagram['linksDataArrray']
2 attrsKeyList ← diagram['nodeDataArrray']
3 foreach link ∈ linksList : do
4   | if link['from'] ∈ attrsKeyList OR link['to'] ∈ attrsKeyList : then
5   |   | count ++
6   | end
7 end
8 if count > 1 then
9   | attrMulticonnections ← attr
10 end
```

Validaciones en las relaciones Las relaciones del diagrama están limitadas por una restricción de la propuesta de solución, esta no permite relaciones de grado superior a 2 y por obvias razones no tiene caso que exista una relación con una sola conexión. Esto no es lo mismo que las relaciones unarias o recursivas, para que una relación pueda ser considerada recursiva deben existir 2 conexiones en la propiedad *linksDataArrray* del objeto *json* *diagram*, en ambas deben existir el identificador de la relación y la entidad, por lo tanto la validación puede limitarse a comprobar que los elementos del tipo relación en el diagrama cuenten con maximo 2 nodos en *linksDataArrray*. De ser encontrado una relación con menos o más conexiones, se agrega a una lista de relaciones inválidas que se muestra al usuario para que pueda corregir esos detalles.

Algorithm 8: Lista de atributos con conexiones multiples.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de relaciones invalidas, *invalidrelationship*

```
1 linksList ← diagram['linksDataArrray']
2 relationsKeyList ← diagram['nodeDataArrray']
3 foreach link ∈ linksList : do
4   | if link['from'] ∈ relationsKeyList OR link['to'] ∈ relationsKeyList :
5   |   | then
6   |   |   | count ++
7   | end
8 if count! = 1 then
9   | invalidrelationship ← relationship
10 end
```

6.3.2. Transformación del diagrama entidad-relación al modelo relacional y generar las sentencias SQL

Para lograr la transformación del modelo entidad-relación del diagramador de la propuesta de solución al modelo relacional, se implementó el algoritmo de 7 pasos descrito en la sección 4.3.3.2, a excepción del paso siete que por restricciones autoimpuestas en la propuesta de solución del proyecto no aplica. El séptimo paso hace referencia a la transformación de relaciones de grado 3 o superior, pero el diagramador no permite relaciones de grado 3 o superior, por lo cual este paso es omitido.

Cabe destacar que por simplicidad se implementó en un solo algoritmo la transformación del modelo entidad-relación al modelo relacional y la generación de las sentencias SQL a partir del modelo relacional.

Paso 1: Mapeado de los tipos de entidad regulares

Algorithm 9: Asociar entidades con sus atributos.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de entidades con sus atributos, *entityWithAttrs*

```
1 entities ← diagram['nodeDataArrray']
2 attrsList ← diagram['nodeDataArrray']
3 linksList ← diagram['linksDataArrray']
4 foreach entity ∈ entities : do
5     foreach link ∈ linksList : do
6         foreach attr ∈ attrsList : do
7             if link = attrKey & link = entityKey : then
8                 entityWithAttrs ← entity : attr
9             end
10            end
11        end
12    end
```

Para cada entidad fuerte en la propiedad *nodeDataArrray* del objeto *json diagram*, se asocian a cada uno de los atributos simples con los que tiene una conexión directa en la propiedad *linksDataArrray* del objeto *json diagram*.

Paso 2: Mapeado de los tipos de entidad débiles

Algorithm 10: Asociar entidades débiles con sus atributos.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de entidades débiles con sus atributos,

weakEntityWithAttrs

```
1 weakEntities ← diagram['nodeDataArrray']
2 attrsList ← diagram['nodeDataArrray']
3 linksList ← diagram['linksDataArrray']
4 foreach entity ∈ entities : do
5   | foreach link ∈ linksList : do
6     |   | foreach attr ∈ attrsList : do
7       |     |   | if link = attrKey & link = entityKey : then
8         |       |     |   weakEntityWithAttrs ← entity.attr
9       |     |   | end
10      |   | end
11    |   | end
12  | end
```

En caso de que el objeto *json* *diagram* cuente con entidades débiles, se debe seguir el mismo proceso para asociar sus atributos. Si este cuenta con un atributo clave parcial, este debe formar parte del atributo clave de la relación con la entidad fuerte asociada.

Paso 3: Mapeado de los tipos de relación 1:1 binaria Primero creamos una lista que contenga las relaciones 1:1 con los identificadores de las entidades que la conforman así como la cardinalidad que debe ser igual a 1. Para el manejo de este tipo de relaciones existen tres metodologías de las cuales se utiliza la metodología de la clave foránea, posteriormente se procede a agregar el atributo clave de la entidad

S a los atributos de la entidad T como clave foránea.

Algorithm 11: Asociar entidades que participan en una relación 1:1 binaria.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de relaciones 1:1 con los identificadores de las entidades participantes, *relations1_1*

```
1 linksList ← diagram['linksDataArrray']
2 foreach link ∈ linksList : do
3   | if link['cardinality'] = 1 : then
4     |   | relations1_1 ← (link, cardinality)
5   | end
6 end
```

Algorithm 12: Agregar el atributo clave de la entidad S a los atributos de la entidad T como clave foránea en una relación 1:1 binaria.

Entrada: una lista de relaciones 1:1 binaria, *relations1_1*

Salida : una lista de entidades con atributos y claves foráneas, *entiesWithAttrs*

```
1 foreach relation ∈ relations1_1 : do
2   | entityS ← entiesWithAttrs[relationKey[0]]
3   | entityT ← entiesWithAttrs[relationKey[1]]
4   | pks ← entityS
5   | entityT.attrs ← pks
6   | entityT.fks ← pks
7 end
```

Paso 4: Mapeado de tipos de relaciones 1:N binarias Primero creamos una lista que contenga las relaciones 1:N con los identificadores de las entidades que la conforman así como la cardinalidad (1 o N). Para el manejo de este tipo de relaciones se debe identificar a la entidad con el tipo de participación N a la que se le debe agregar como clave foránea el atributo clave del otro lado de la participación en la relación, esto se debe a que cada instancia de entidad del lado de participación N esta relacionado, a lo sumo, con una única instancia de la entidad con participación

1 de la relación.

Algorithm 13: Asociar entidades que participan en una relación 1:N binaria.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de relaciones 1:N con los identificadores de las entidades participantes, *relations1_N*

```
1 linksList ← diagram['linksDataArrray']
2 foreach link ∈ linksList : do
3   | if (link['cardinality'] ∈ [1, N] : then
4   |   | relations1_N ← (link, cardinality)
5   | end
6 end
```

Algorithm 14: Agregar el atributo clave de la entidad T a los atributos de la entidad S como clave foránea en una relación 1:N binaria.

Entrada: una lista de relaciones 1:N binaria, *relations1_N*

Salida : una lista de entidades con atributos y claves foráneas, *entiesWithAttrs*

```
1 foreach relation ∈ relations1_1 : do
2   | if (link['cardinality'] = N : then
3   |   | entitys ← entiesWithAttrs[relationKey]
4   | end
5   | if (link['cardinality'] = 1 : then
6   |   | entityT ← entiesWithAttrs[relationKey]
7   | end
8 end
9 entitys.attrs ← pkS
10 entitys.fks ← pkS
```

Paso 5: Mapeado de tipos de relaciones N:M binarias Creamos una lista que contenga las relaciones N:M con los identificadores de las entidades que la conforman así como la cardinalidad(N o N). Para el manejo de este tipo de relaciones se debe crear una nueva entidad S y agregar todos los atributos simples o compuestos con los que se tengan una conexión. Adicionalmente, se deben agregar los atributos claves de las entidades participantes de la relación como claves foráneas de la nueva relación S, de la misma forma la combinación de estas claves foráneas formaran el atributo

clave de la nueva entidad S.

Algorithm 15: Asociar entidades que participan en una relación N:M binaria.

Entrada: una instancia del diagram ER en formato *json*, *diagram*

Salida : una lista de relaciones 1:N con los identificadores de las entidades participantes, *relationsM_N*

```
1 linksList ← diagram['linksDataArrray']
2 foreach link ∈ linksList : do
3   | if (link['cardinality'] ∈ [M, N] : then
4   |   | relationsM_N ← (link, cardinality)
5   | end
6 end
```

Algorithm 16: Agregar los atributo simples, atributos claves y claves foráneas en una relación 1:N binaria.

Entrada: una lista de relaciones N:N binaria, *relationsN_M*

Salida : una entidad con atributos y claves foráneas, *entiesWithAttrs*

```
1 attrsList ← diagram['nodeDataArrray']
2   linksList ← diagram['linksDataArrray'] foreach
3     relation ∈ relationsN_M : do
4       | foreach link ∈ linksList : do
5         |   | foreach attr ∈ attrsList : do
6           |   |   | if link = attrKey & link = relationKey : then
7             |   |   |   | entityS ← entity : attr
8             |   |   |   | entityS.pk = entiesWithAttrs[link]
9             |   |   |   | entityS.fk = entiesWithAttrs[link]
10            |   |   | end
11          |   | end
12        |   | end
13      | end
14    | end
15  | end
```

Paso 6: Mapeado de atributos multivalor Para realizar esto primero creamos una lista que contenga las entidades del tipo multivalor, por cada atributo multivalor E se debe crear una nueva entidad R el cual tendrá como atributos un atributo de la entidad E y el atributo clave de la entidad S de la relación relación como clave

foránea.

Algorithm 17: Asociar entidades multivalor con sus atributos.

Entrada: una instancia del diagrama ER en formato *json*, *diagram*
Salida : una lista de entidades del tipo multivalor con sus atributos,
 emvWithAttrs

```
1 entitiesMultivalue ← diagram['nodeDataArrray']
2 attrsList ← diagram['nodeDataArrray']
3 linksList ← diagram['linksDataArrray']
4 foreach entity ∈ entities : do
5     foreach link ∈ linksList : do
6         foreach attr ∈ attrsList : do
7             if entity ='multiValue' & link = entityKey : then
8                 emvWithAttrs ← entity : attr
9                 emvWithAttrs.pk = entity.pk
10            end
11        end
12    end
13 end
```

Con los pasos anteriores tenemos de manera separada los elementos necesarios para obtener las sentencias SQL equivalentes al diagrama ER de entrada, como todos los componentes se encuentran en listas resulta fácil iterarlos y poder construir los bloques del *script* (con la ayuda de un template previamente generado) en el lenguaje de consultas SQL para que puedan ser ejecutadas en el SGDB MySQL.

Se hizo uso de los siguientes *templates* para crear las sentencias correspondientes y crear las tablas con sus atributos como se aprecia en la figura 6.20 y elementos de distinción como son la clave primaria de la figura 6.21 y las claves foráneas de la figura 6.22 dependiendo del tipo de relación entre los elementos del diagrama ER.

```
def build_table_sentence(self, table_dict):
    table_template = """
DROP TABLE IF EXISTS `{table_name}`;
CREATE TABLE IF NOT EXISTS `{table_name}` (
{attrs_sentences},
{primary_key}
{foreign_keys}
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
"""
    {-
    }
    return table_template.format(table_name=table_name, attrs_sentences=attr_by_table, primary_key=primary_key, foreign_keys=foreign_keys)
```

Figura 6.20: Fragmento de código del template para construir la sentencia *CREATE TABLE*.

```
def buildPrimaryKey(self, attr_list):
    primary_key_sentence = "PRIMARY KEY ({})"
    primary_key = [f" {attr[0]}{" for attr in attr_list if attr[2] == 'keyAttribute'}"]
    return primary_key_sentence.format(", ".join(primary_key))
```

Figura 6.21: Fragmento de código del template para construir la sentencia *PRIMARY KEY*.

```

def buildForeignKeys(self, attr_list):
    foreing_key_sentence = "FOREIGN KEY ({attr_name}) REFERENCES {ref_table_name} ({attr_ref_table})"
    fk_list = []
    for attr in attr_list:
        attr_ref_table, ref_table = attr.rsplit("_",1)
        fk_list.append(f"FOREIGN KEY ({attr}) REFERENCES {ref_table} ({attr_ref_table})")
    return ",\n".join(fk_list)

```

Figura 6.22: Fragmento de código del template para construir la sentencia *FOREING KEY*.

6.3.3. Transformación modelo entidad-relación a entidades del modelo conceptual GDM

La generación de las entidades del modelo GDM desde el modelo entidad-relación básico es de la siguiente manera:

A continuación se explica brevemente la implementación.

Algorithm 18: Generar los *features* y las *references* del GDM desde el modelo entidad-relación.

Entrada: diagrama entidad-relación básico, *er*
Salida : entidades del Generic Datametamodel, *entities_{gdm}*

```

1 entities ← list(filter(lambda item : item['type'] ==' entity',nodeData))
2 entities2pair ← list(map(lambda entity :
3     (entity,getNodeConnectedNodes(entity['key'], nodeData, linkData)), entities))
4 entitiesgdm ← array
5 foreach pair ∈ entities2pair : do
6     references ← references(pair)
7     features ← features(pair)
8     entitypair.key
9     entity.references ← references
10    entity.features ← features
11    entitiesgdm.append(entity)
12 end

```

Código fuente 6.8: Generación de entidades del GDM desde el modelo entidad-relación básico

```

1      def main():
2
3          input_file = open('er.json')
4          data = json.load(input_file)
5          input_file.close()
6
7          nodeData = data['nodedataArray']
8          linkData = data['linkdataArray']
9
10         # Obtenemos entidades del modelo ER
11         entities = list(filter(lambda item: item['type'] ==
12             "entity",nodeData))

```

```

12
13     # Obtenemos lista de pares, donde la clave es una entidad y su valor
14     # son los nodos a los que está conectada la entidad
15     entities2pair = list(map(lambda entity: (entity,
16                               getConnectedNodes(entity['key'], nodeData, linkData)),entities))
16     count = 0
17     for pair in entities2pair:
18         entity = pair[0]
19         features = pair[1]
20         attributes = []
21         references = []
22         for feature in features:
23             if (feature['type'] == "relation"):
24                 reference = {}
25                 relation = getRelationInfo( feature["key"], linkData,
26                                         nodeData)
27
28                 if (relation["from"]["key"] != entity["key"]):
29                     reference["entity"] = relation["from"]["text"]
30                     reference["cardinality"] = relation["fromC"]
31                     if relation["fromC"] != '1':
32                         reference["name"] = feature["text"] +
33                         relation["from"]["text"]
34                     else:
35                         reference["name"] =
36                             toFirstLower(relation["from"]["text"])
37                 else:
38                     reference["entity"] = relation["to"]["text"]
39                     reference["cardinality"] = relation["toC"]
40                     # Si la relación es a N, va el nombre de la relación
41                     # concatenado con el nombre de la entidad
42                     if relation["toC"] != '1':
43                         reference["name"] = feature["text"] +
44                         relation["to"]["text"]
45                     else:
46                         reference["name"] =
47                             toFirstLower(relation["to"]["text"])
48
49             references.append(reference)
50             if ( isAttribute(feature['type']) ):
51                 attribute = {}
52                 attribute["name"] = feature['text']
53                 attribute["type"] = feature["gdmType"]
54                 attribute["array"] = True if feature["type"] ==
55                     "multivalueAttribute" else False
56                 attribute["unique"] = True if feature["type"] ==
57                     "keyAttribute" else False
58                 if attribute["unique"] == True:
59                     attribute["type"] = "id"
60                 attributes.append(attribute)

```

```

53
54     # Escribimos el archivo de texto
55     if count == 0:
56         count += 1
57     with open("laloVenues.gdm", 'w') as output_file:
58         output_file.write("entity " + entity["text"] + " {\n")
59         for attribute in reversed(attributes):
60             output_file.write(parseAttribute(attribute))
61         for reference in reversed(references):
62             output_file.write(parseReference(reference))
63         output_file.write("}\n\n")
64         output_file.close()
65     else:
66         with open("laloVenues.gdm", 'a') as output_file:
67             output_file.write("entity " + entity["text"] + " {\n")
68             for attribute in reversed(attributes):
69                 output_file.write(parseAttribute(attribute))
70             for reference in reversed(references):
71                 output_file.write(parseReference(reference))
72             output_file.write("}\n\n")
73             output_file.close()
74
    return

```

El modelo GDM del código 6.11 está compuesta por entidades y consultas, cada entidad contiene *features*, y cada *feature* puede ser atributo o una referencia a otra entidad. Todos los atributos tienen un tipo y una cardinalidad.

Este algoritmo solo obtiene las entidades del modelo GDM. Se necesitan además consultas definidas por el usuario para generar el modelo lógico orientado a documentos.

6.3.4. Parser del archivo de texto simple GDM a su .model

Una vez obtenido el archivo de texto simple GDM que está conformado por las entidades (atributos y referencias) y las consultas definidas por el usuario es necesario generar un modelo que entienda PyEcore. A continuación se muestra el algoritmo y parte de su implementación.

Algorithm 19: Generar el .model desde un modelo GDM en archivo de texto simple

Entrada: modelo GDM en texto simple, *gdmTextoSimple*

Salida : modelo GDM en XML, *gdmXML*

- 1 *entities* \leftarrow *Entities(gdmTextoSimple)*
 - 2 *queries* \leftarrow *Queries(gdmTextoSimple)*
 - 3 *gdm* \leftarrow *gdm.Model(entities, queries)*
 - 4 *gdm.saveXML()*
-

Código fuente 6.9: Parser GDM de texto simple a XML del GDM

```

1      def main():
2
3          # Creamos el modelo GDM
4          model = gdm.Model()
5
6          input_file = open('laloVenues.gdm', 'r')
7          lines = input_file.readlines()
8
9          # Primero creamos las entidades y las consultas, porque las
10         ↳ necesitamos para crear las referencias
11         for line in lines:
12             if "entity" in line:
13                 entity = gdm.Entity(name=line.split()[1])
14                 model.entities.append(entity)
15             if "query" in line:
16                 query = gdm.Query(name=line.split()[1].strip(":"))
17                 model.queries.append(query)
18
19         # Parseamos el documento para agregar llenar las entidades
20         for i in range(len(lines)):
21             line = lines[i]
22
23             # Ignoramos los comentarios
24             if "/*" in line or "* " in line or "*/" in line:
25                 continue
26
27             # Si es una entidad
28             if "entity" in line:
29                 populateEntity(model, lines, i)
30
31         # Parseamos el documento para agregar llenar las consultas
32         for i in range(len(lines)):
33             line = lines[i]
34             # Si es una consulta
35             if "query" in line:
36                 populateQuery(model, lines, i)
37
38         saveModel(model)

```

6.3.5. Transformación de una instancia GDM al modelo lógico orientado a documentos

Como se ha explicado en la sección 4.3.3.5, el algoritmo a implementar es una transformación entre modelos. A continuación se adjunta una breve descripción del

algoritmo y su implementación.

Algorithm 20: Transformación del modelo conceptual GDM al modelo lógico orientado a documentos

Input : una instancia del modelo GDM, *gdm*
Output: un modelo lógico orientado a documentos, *ddm*

```
1 mainEntities ← gdm.queries.collect((q)|q.from);  
2 foreach me ∈ mainEntities do  
3     collection ← newCollection();  
4     collection.name ← me.name;  
5     accessTree ← allQueryPaths(me, gdm.queries);  
6     collection ← populateDocumentType(collection.root, accessTree);  
7     ddm.collections.add(collection);  
8 end
```

Donde la función *populateDocumentType()* es otro algoritmo de la forma:

Algorithm 21: Generar el contenido de un *DocumentType* dado un árbol de acceso

Input : Un “document type”, *dt*
Output: un nodo del arbol de acceso

```
1 nodeAttributes ← node.entity.features.select(f|f.isTypeOf(Attribute))  
2 nodeReferences ←  
    node.entity.features.select(f|f.isTypeOf(Reference))  
3 foreach attr ∈ nodeAttributes do  
4     pf ← newPrimitiveField()  
5     pf.name ← attr.name  
6     pf.type ← attr.type  
7     dt.fields.add(pf)  
8 end  
9 foreach ref ∈ nodeReferences do  
10    targetNode ← node.arcs.find(a|a.name = ref.name).target  
11    if exists(targetNode) then  
12        baseType ← newDocumentType()  
13        populateDocumentType(baseType, targetNode)  
14    else  
15        baseType ← newPrimitiveField()  
16        baseType.type ← findIdType(ref.entity)  
17    end  
18    baseType.name ← ref.name  
19    if ref.cardinality == 1 then  
20        dt.field.add(baseType)  
21    else  
22        arrayField ← newArrayField()  
23        arrayField.type ← baseType  
24        dt.fields.add(arrayField)  
25    end  
26 end
```

Código fuente 6.10: Transformación modelo a modelo: GDM a modelo lógico orientado a documentos

```
1      def main():
2
3          gdmModel = loadModel("Static_model_test.xmi")
4
5          ddmModel = ddm.DocumentDataModel()
6
7          # Obtenemos las entidades de los elementos From
8          # mainEntities = gdm.queries.map[q / q.from.entity].toSet
9          mainEntities = set(map(lambda q: q.from_.entity, gdmModel.queries))
10
11         # entityToQueries = mainEntities.map[me / me ->
12         #                                     ↵ gdm.queries.filter[q / q.from.entity.equals(me)]]
13         entityToQueries = list(map(lambda me: (me, list(filter(lambda q:
14         #                                     ↵ q.from_.entity.name == me.name, gdmModel.queries))),
15         #                                     ↵ mainEntities)))
16
17         # val entity2accessTree = newImmutableMap(entityToQueries.map[e2q /
18         #                                     ↵ e2q.key -> createAccessTree(e2q.value)])
19         entity2accessTree = list(map(lambda e2q: (e2q[0],
20         #                                     ↵ createAccessTree(e2q[1])), entityToQueries))
21
22
23         # Completamos cada árbol de acceso
24         for entity in mainEntities:
25             tree = getTree(entity2accessTree, entity)
26             othersTrees = getAllTrees(entity2accessTree)
27             completeAccessTree(entity, tree, othersTrees)
28
29         # Generamos las colecciones
30         for entity in mainEntities:
31             tree = getTree(entity2accessTree, entity)
32             collection = ddm.Collection()
33             collection.name = entity.name
34             docType = ddm.Document()
35             docType.name = "root"
36             collection.root = docType
37             populateDocument(docType, entity, tree, mainEntities)
38             ddmModel.collections.append(collection)
39             saveModelDDM(ddmModel)
40
41         saveModelDDM(ddmModel)
```

La implementación del código fuente 6.10 toma de entrada una instancia del modelo GDM en XML (un .model generado por el *parser implementado*) y de salida genera otro .model correspondiente al modelo lógico orientado a documentos.

Cada entidad principal tiene un arbol de acceso asociado, para generar los documentos anidados se hace uso de ese arbol de acceso.

6.3.6. Transformación del modelo lógico orientado a documentos a MongoDB

La generación de las sentencias de MongoDB es desde el modelo lógico orientado a documentos y se hace uso de un validador para definir la estructura de las colecciones.

Algorithm 22: Generar las sentencias de MongoDB desde un modelo lógico orientado a documentos

Entrada: modelo DDM, *ddm*

Salida : sentencias MongoDB, *mongo*

1 *mongo* \leftarrow *parse*(*ddm*)

2 *mongo.save()*

Código fuente 6.11: Generación de sentencias MongoDB desde el DDM

```
1      import langs.ddmLang as ddm
2      from pyecore.resources import ResourceSet, URI
3      from pyecore.resources.xmi import XMIResource
4
5      def loadModelDDM(input_file):
6          rset = ResourceSet()
7          # register the metamodel (available in the generated files)
8          rset.metamodel_registry[ddm.nsURI] = ddm
9          rset.resource_factory['ddm'] = lambda uri: XMIResource(uri)
10         resource = rset.get_resource(URI(input_file))
11         model = resource.contents[0]
12         return model
13
14     def getMongoDBType(type_):
15         if type_.name == "INT":
16             tipo = "int"
17         elif type_.name == "FLOAT":
18             tipo = "double"
19         elif type_.name == "TEXT":
20             tipo = "string"
21         elif type_.name == "DATE":
22             tipo = "date"
23         elif type_.name == "TIMESTAMP":
24             tipo = "timestamp"
25         elif type_.name == "BOOLEAN":
26             tipo = "boolean"
27         elif type_.name == "ID":
28             tipo = "objectId"
29         else:
30             tipo = "string"
31
32         return tipo
33
34     def generateField(output_file,field):
35         if isinstance(field, ddm.PrimitiveField):
```

```

36         generatePrimitiveField(output_file,field)
37     elif isinstance(field, ddm.Document):
38         generateDocumentField(output_file,field)
39     elif isinstance(field, ddm.ArrayField):
40         generateArrayField(output_file,field)
41     return
42
43 def generateArrayField(output_file, field):
44     output_file.write(field.name + ": {\n")
45     output_file.write("    bsonType: [\"array\"],\n")
46     output_file.write("    items: {\n")
47     output_file.write("        properties: {\n")
48     generateField(output_file,field.type)
49     output_file.write("            }\n")
50     output_file.write("        }\n")
51     output_file.write("    }\n")
52     return
53
54 def generatePrimitiveField(output_file, field):
55     output_file.write(field.name + ": { bsonType: \""
56     ↪  getMongoDBType(field.type) + "\"}\n")
57     return
58
59 def generateDocumentField(output_file,document):
60
61     for field in document.fields:
62         if isinstance(field, ddm.PrimitiveField):
63             generatePrimitiveField(output_file,field)
64         elif isinstance(field, ddm.Document):
65             generateDocumentField(output_file,field)
66         elif isinstance(field, ddm.ArrayField):
67             generateArrayField(output_file,field)
68         if field != document.fields[-1]:
69             output_file.write(", ")
70
71     return
72
73 def writeFile(output_file,collection):
74     output_file.write("db.createCollection(\"" + collection.name +
75     ↪  "\", {\n")
76     output_file.write("    validator: {\n")
77     output_file.write("        $jsonSchema: {")
78     output_file.write("            bsonType: \"object\", \n")
79     output_file.write("            properties: {\n")
80
81     generateDocumentField(output_file,collection.root)
82
83     output_file.write("        }\n")
84     output_file.write("    }\n} }\n")
85     output_file.close()
86     return

```

```
85
86     def main():
87         ddmModel = loadModelDDM("prueba_ddm.xmi")
88         ofile = "laloMongo.json"
89         count = 0
90         for collection in ddmModel.collections:
91             if count == 0:
92                 count += 1
93                 with open(ofile, 'w') as output_file:
94                     writeFile(output_file,collection)
95             else:
96                 with open(ofile, 'a') as output_file:
97                     writeFile(output_file,collection)
98
99
100        return
```

6.4. Conclusiones

Durante el proceso de desarrollo de este trabajo terminal se han presentado algunos retos que no se tenían contemplados en un inicio, desde la creación o separación del proyecto en 2 aplicaciones para un mejor manejo del trabajo en equipo hasta las tecnologías a utilizar.

El resolver cada uno esos retos ha dejado un aprendizaje para el equipo de desarrollo, ya que enseña que aun teniendo un plan de trabajo y fechas establecidas, siempre existen inconvenientes con los que uno no cuenta, por lo que el equipo se debe adaptar a estos cambios para tratar de obtener los resultados y, en caso de ser necesario, modificar el plan inicial.

Uno de los mas grandes retos fue la transformación del diagrama entidad-relación al modelo conceptual NoSQL, ya que se en el plan inicial no se contempló la falta de tecnologías disponibles para transformaciones entre gramáticas en distintos niveles de abstracción. Al no tener experiencia ni el contexto de todo lo que implica la ingeniería orientada a modelos, y por las limitantes en tiempo para este trabajo terminal, el equipo pensó en usar parte del proyecto de un tercero para lograr el alcance esperado. Sin embargo, gracias a PyEcore[50] se logró implementar los algoritmos de transformación en el lenguaje de programación Python.

Capítulo 7

Caso de estudio

Para el caso de estudio se ha decidido usar el diagrama entidad-relación más usado en la literatura NoSQL, que es el modelo Venues que puede encontrar en el *paper* de Chebotko [4].

7.1. Diagramado modelo entidad-relación básico

El diagramado de Venues se ha hecho con la propuesta de solución desarrollada y en la figura 7.1

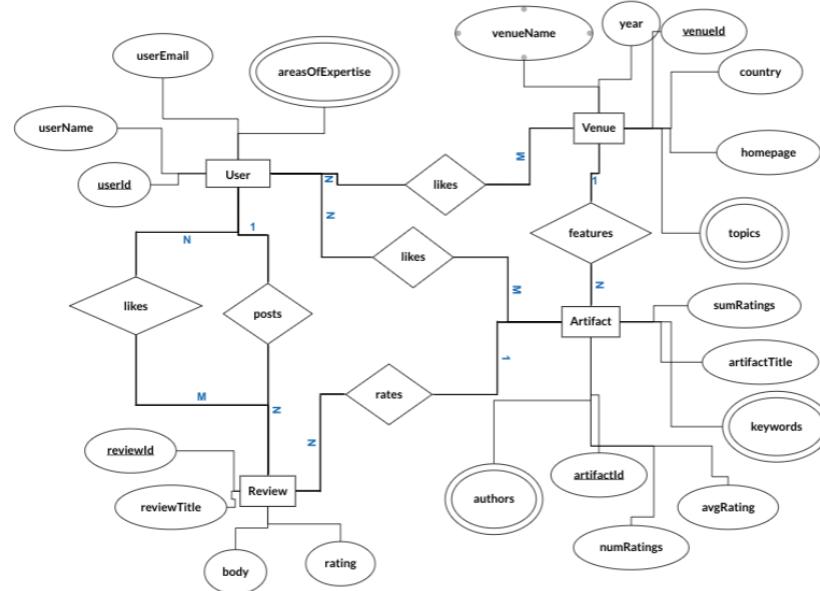
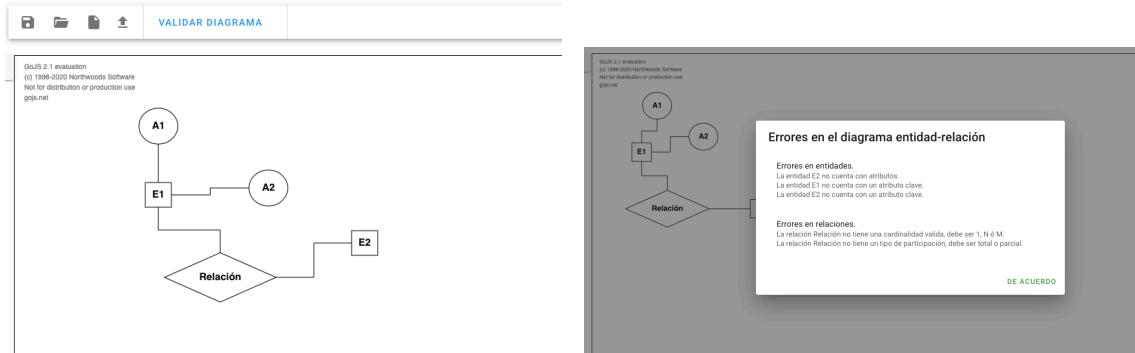


Figura 7.1: Diagrama entidad-relación de Venues

7.2. Validación modelo entidad-relación básico

Una vez que el usuario cree tener un diagrama válido o quiere conocer si su diagrama actual presenta algún error, puede hacer click en el botón “validar”

diagrama” con lo que se le mostrará una ventana con los errores que la aplicación detecta. La figura 7.2a muestra un ejemplo de diagrama sencillo mientras que en la figura 7.2b se aprecian los errores estructurales del mismo.



Siguiendo con el ejemplo del modelo Venues la figura 7.3 muestra la validación del diagrama y como se espera este es estructuralmente válido.

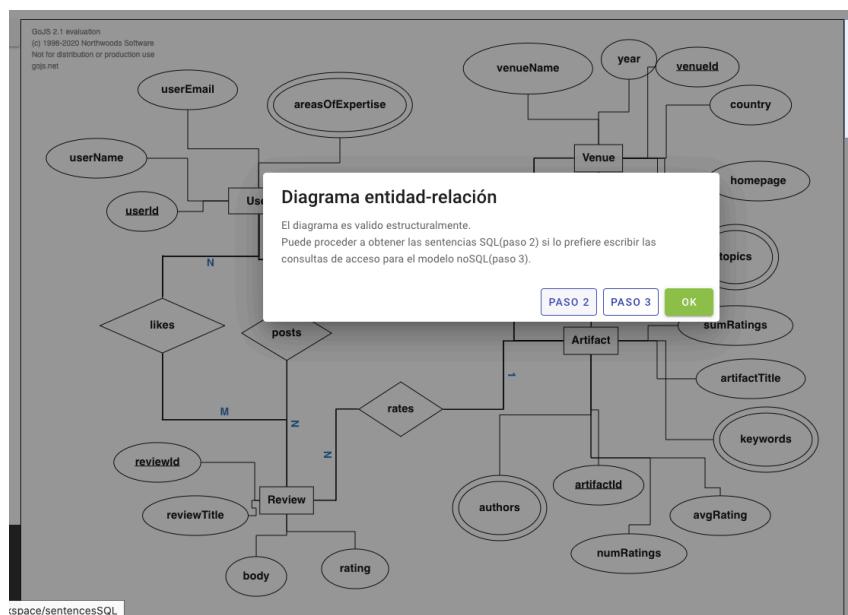


Figura 7.3: Validación del modelo Venues.

7.3. Generación sentencias SQL

Una vez validado el diagrama entidad-relación básico se pueden generar las sentencias SQL que se muestra en la figura 7.4

1: DIAGRAMA ER 2: SENTENCIAS SQL 3: MODELO CONCEPTUAL NOSQL 4: MODELO LÓGICO Y FÍSICO NOSQL

[DESCARGAR](#) | [OBTENER SENTENCIAS SQL](#)

```

DROP DATABASE IF EXISTS `venues`;
CREATE DATABASE `venues`;
USE `venues`;

DROP TABLE IF EXISTS `User`;
CREATE TABLE IF NOT EXISTS `User` (
  `userEmail` varchar(1),
  `userName` varchar(1),
  `userId` varchar(1),
  PRIMARY KEY (`userId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS `Artifact`;
CREATE TABLE IF NOT EXISTS `Artifact` (
  `artifactTitle` varchar(1),
  `avgRating` varchar(1),
  `sumRatings` varchar(1),
  `numRatings` varchar(1),
  `artifactId` varchar(1),
  `venueId_venue` varchar(1),
  PRIMARY KEY (`artifactId`),
  FOREIGN KEY (`venueId_venue`) REFERENCES venue(`venueId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS `Venue`;
CREATE TABLE IF NOT EXISTS `Venue` (
  `country` varchar(1),
  `homepage` varchar(1),
  `venueId` varchar(1),
  `year` varchar(1),
  `venueName` varchar(1),
  PRIMARY KEY (`venueId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS `Review`;
CREATE TABLE IF NOT EXISTS `Review` (
  `reviewId` varchar(1),
  `reviewTitle` varchar(1),
  `body` varchar(1),
  PRIMARY KEY (`reviewId`)
)

```

Figura 7.4: Sentencias SQL de Venues.

Prueba de sentencias SQL en MySQL

Como se muestra en la figura 7.5 se ha probado la ejecución de las sentencias sql en el gestor gráfico MySQL Workbench en su versión 8.0.22 y al no apreciar en la parte inferior de la figura ningún error se concluye que la ejecución ha sido exitosa.

```

1  DROP DATABASE IF EXISTS `t2019-b052`;
2  CREATE DATABASE `t2019-b052`;
3  USE `t2019-b052`;
4
5  DROP TABLE IF EXISTS `User`;
6  CREATE TABLE IF NOT EXISTS `User` (
7    `userId` varchar(1),
8    `userName` varchar(1),
9    `userEmail` varchar(1),
10   `userPass` varchar(1),
11   PRIMARY KEY (`userId`)
12 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
13
14  DROP TABLE IF EXISTS `Venue`;
15  CREATE TABLE IF NOT EXISTS `Venue` (
16    `venueId` varchar(1),
17    `venueName` varchar(1),
18    `venueAddress` varchar(1),
19    `venueLat` varchar(1),
20    `venueLong` varchar(1),
21    `venueCapacity` varchar(1),
22    PRIMARY KEY (`venueId`)
23 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
24
25  DROP TABLE IF EXISTS `Artifact`;
26

```

Action Output C

Time	Action	Response	Duration / Fetch Time
1 09:04:05	DROP DATABASE If exists t2019-b052	8 row(s) affected	0.024 sec
2 09:04:05	CREATE DATABASE t2019-b052	1 row(s) affected	0.00000 sec
3 09:04:05	USE t2019-b052	0 row(s) affected	0.00020 sec
4 ▲ 09:04:05	DROP TABLE IF EXISTS `User` (`userId` varchar(1), `userName` varchar(1), `userEmail` varchar(1), `userPass` varchar(1), PRIMARY KEY (`userId`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;	0 row(s) affected; 1 warning(s); 1051 Unknown table 't2019-b052.user'	0.0008 sec
5 ▲ 09:04:05	CREATE TABLE IF NOT EXISTS `User` (`userId` varchar(1), `userName` varchar(1), `userEmail` varchar(1), `userPass` varchar(1), PRIMARY KEY (`userId`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;	0 row(s) affected; 1 warning(s); 1051 Unknown table 't2019-b052.user'	0.0068 sec
6 ▲ 09:04:05	DROP TABLE IF EXISTS `Venue` (`venueId` varchar(1), `venueName` varchar(1), `venueAddress` varchar(1), `venueLat` varchar(1), `venueLong` varchar(1), `venueCapacity` varchar(1), PRIMARY KEY (`venueId`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;	0 row(s) affected; 1 warning(s); 1051 Unknown table 't2019-b052.venue'	0.0009 sec
7 ▲ 09:04:05	CREATE TABLE IF NOT EXISTS `Venue` (`venueId` varchar(1), `venueName` varchar(1), `venueAddress` varchar(1), `venueLat` varchar(1), `venueLong` varchar(1), `venueCapacity` varchar(1), PRIMARY KEY (`venueId`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;	0 row(s) affected; 1 warning(s); 1051 Unknown table 't2019-b052.venue'	0.0050 sec
8 ▲ 09:04:05	DROP TABLE IF EXISTS `Artifact` (`artifactTitle` varchar(1), `keywords` varchar(1), `ratesReview` ref `Review`[], `likesArtifact` ref `User`[], `likesVenue` ref `Venue`[], `likesReview` ref `Review`[])	0 row(s) affected; 1 warning(s); 1051 Unknown table 't2019-b052.artifact'	0.0007 sec
9 ▲ 09:04:05	CREATE TABLE IF NOT EXISTS `Artifact` (`artifactTitle` varchar(1), `keywords` varchar(1), `ratesReview` ref `Review`[], `likesArtifact` ref `User`[], `likesVenue` ref `Venue`[], `likesReview` ref `Review`[])	0 row(s) affected; 1 warning(s); 1051 Unknown table 't2019-b052.artifact'	0.0007 sec
10 ▲ 09:04:05	DROP TABLE IF EXISTS `Keywords_artPart`	0 row(s) affected; 1 warning(s); 1051 Unknown table 't2019-b052.keywords_artPart'	0.0007 sec

Figura 7.5: Ejecución de las sentencias SQL en MySQL Workbench.

7.4. Modelo conceptual NoSQL

Con un diagrama entidad-relación básico válido es posible generar las entidades del modelo conceptual NoSQL como se muestra en la figura 7.6.

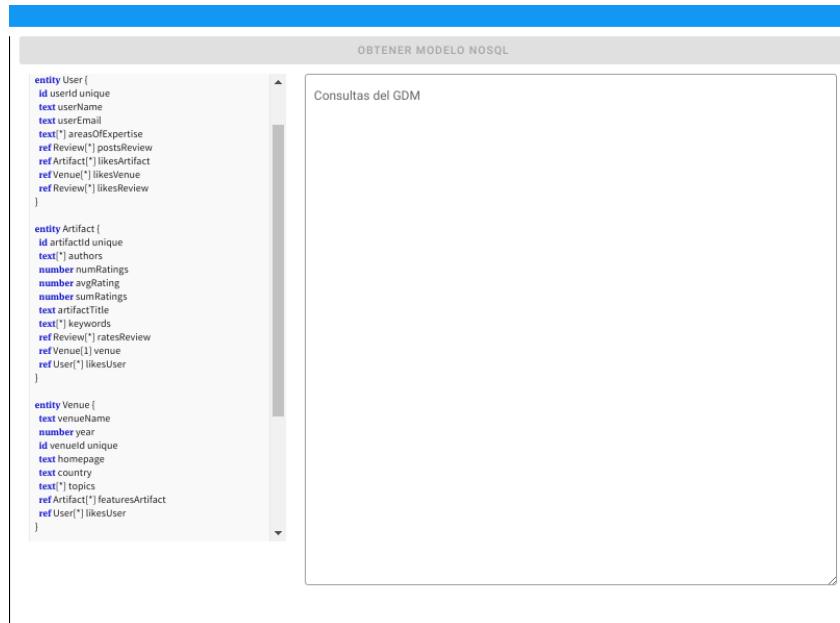


Figura 7.6: Entidades del modelo conceptual GDM

Una vez generadas las entidades del modelo conceptual GDM es necesario que el usuario defina las consultas en la sección de consultas del GDM como se muestra en la figura 7.7.

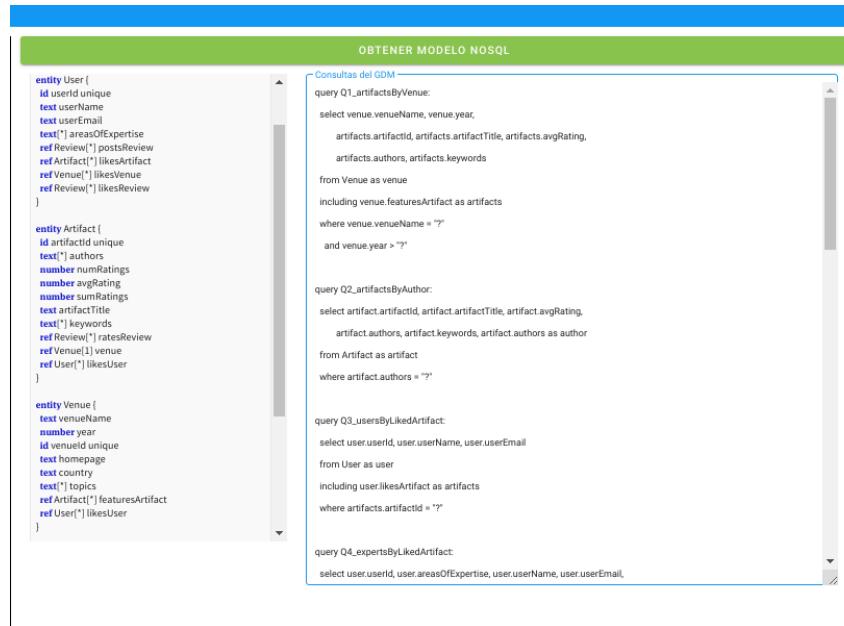


Figura 7.7: Entidades y consultas del modelo conceptual GDM

Las nueve consultas que se han definido son las siguientes:

```

query Q1_artifactsByVenue:
select venue.venueName, venue.year,
       artifacts.artifactId, artifacts.artifactTitle, artifacts.avgRating,
       artifacts.authors, artifacts.keywords
from Venue as venue
including venue.featuresArtifact as artifacts
where venue.venueName = "?"
and venue.year > "?"

query Q2_artifactsByAuthor:
select artifact.artifactId, artifact.artifactTitle, artifact.avgRating,
       artifact.authors, artifact.keywords, artifact.authors as author
from Artifact as artifact
where artifact.authors = "?"

query Q3_usersByLikedArtifact:
select user.userId, user.userName, user.userEmail
from User as user
including user.likesArtifact as artifacts
where artifacts.artifactId = "?"

query Q4_expertsByLikedArtifact:
select user.userId, user.areasOfExpertise, user.userName, user.userEmail,
       user.areasOfExpertise as areaOfExpertise
from User as user
including user.likesArtifact as artifacts
where artifacts.artifactId = "?"

```

```

        and user.areasOfExpertise = "?"

query Q5_ratingByArtifact:
    select artifact.artifactId, artifact.avgRating
    from Artifact as artifact
    where artifact.artifactId = "?"

query Q6_venuesLikedByUser:
    select user.userId, venues.venueName, venues.year,
           venues.country, venues.homepage
    from User as user
    including user.likesVenue as venues
    where user.userId = "?"

query Q7_artifactsLikedByUser:
    select user.userId,
           likesArtifacts.artifactId,
           likesArtifacts.artifactTitle, likesArtifacts.authors,
           venue.venueName
    from User as user
    including user.likesArtifact as likesArtifacts,
               user.likesArtifact.venue as venue
    where user.userId = "?"
        and venue.year > "?"

query Q8_reviewsByUser:
    select user.userId,
           reviews.reviewId, reviews.reviewTitle, reviews.body,
           artifact.artifactId, artifact.artifactTitle
    from User as user
    including user.postsReview as reviews,
               user.postsReview.artifact as artifact
    where user.userId = "?"
        and reviews.rating > "?"

query Q9_artifacts:
    select artifact.artifactId, artifact.artifactTitle, artifact.avgRating,
           artifact.authors, artifact.keywords,
           venue.venueName
    from Artifact as artifact
    including artifact.venue as venue
    where artifact.artifactId = "?"

```

7.5. Modelo lógico NoSQL

A partir de un modelo conceptual GDM, que está conformado por entidades y consultas, se puede generar el modelo lógido orientado a documentos. En la figura 7.8

se muestra el diagrama correspondiente al modelo lógico orientado a documentos de Venues.

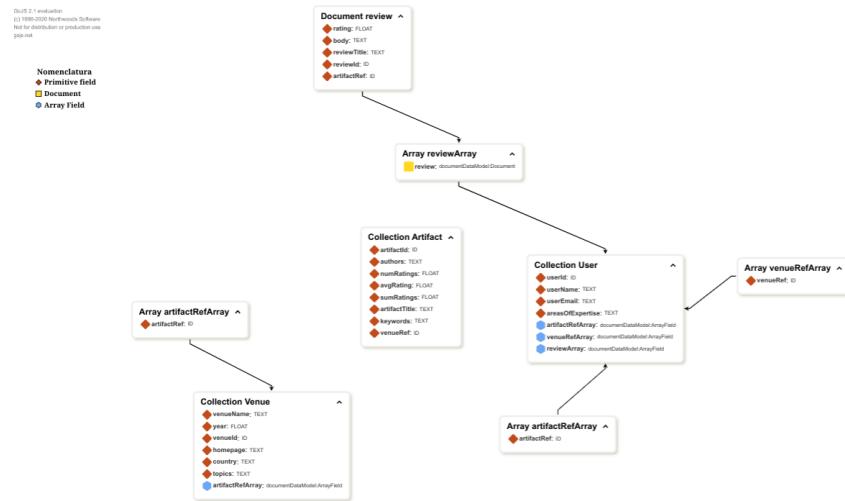


Figura 7.8: Modelo lógico orientado a documentos de Venues

7.6. Modelo físico NoSQL

Tambien se pueden obtener las sentencias para MongoDB. Para el modelo Venues se muestra a continuación las sentencias correspondientes a su modelo lógico.

```

1   db.createCollection("Venue",
2 {
3     validator: {
4       $jsonSchema: {
5         bsonType: "object",
6         properties: {
7           venueName: { bsonType: "string" },
8           year: { bsonType: "double" },
9           venueId: { bsonType: "objectId" },
10          homepage: { bsonType: "string" },
11          country: { bsonType: "string" },
12          topics: { bsonType: "string" },
13          artifactRefArray: {
14            bsonType: [
15              "array"
16            ],
17            items: {
18              properties: {
19                artifactRef: { bsonType: "objectId"
20                }
21              }
22            }
23          }
24        }
25      }
26    }
27  }

```

```

23     }
24   }
25 }
26 })
27 db.createCollection("Artifact",
28 {
29   validator: {
30     $jsonSchema: { bsonType: "object",
31       properties: {
32         artifactId: { bsonType: "objectId"
33           }, authors: { bsonType: "string"
34           }, numRatings: { bsonType: "double"
35           }, avgRating: { bsonType: "double"
36           }, sumRatings: { bsonType: "double"
37           }, artifactTitle: { bsonType: "string"
38           }, keywords: { bsonType: "string"
39           }, venueRef: { bsonType: "objectId"
40           }
41         }
42       }
43     }
44 })
45 db.createCollection("User",
46 {
47   validator: {
48     $jsonSchema: { bsonType: "object",
49       properties: {
50         userId: { bsonType: "objectId"
51           }, userName: { bsonType: "string"
52           }, userEmail: { bsonType: "string"
53           }, areasOfExpertise: { bsonType: "string"
54           }, artifactRefArray: {
55             bsonType: [
56               "array"
57             ],
58             items: {
59               properties: {
60                 artifactRef: { bsonType: "objectId"
61                   }
62                 }
63               }
64             }, venueRefArray: {
65               bsonType: [
66                 "array"
67               ],
68               items: {
69                 properties: {
70                   venueRef: { bsonType: "objectId"
71                     }
72                 }
73               }
74             }
75           }
76         }
77       }
78     }
79   }
80 }
81 )
82 })
83 db.createCollection("Venue",
84 {
85   validator: {
86     $jsonSchema: { bsonType: "object",
87       properties: {
88         venueId: { bsonType: "objectId"
89           }, name: { bsonType: "string"
90           }, address: { bsonType: "string"
91           }, city: { bsonType: "string"
92           }, state: { bsonType: "string"
93           }, zipCode: { bsonType: "string"
94           }, latitude: { bsonType: "double"
95           }, longitude: { bsonType: "double"
96           }, description: { bsonType: "string"
97           }
98         }
99       }
100     }
101   }
102 }
103 )
104 })
105 db.createCollection("Rating",
106 {
107   validator: {
108     $jsonSchema: { bsonType: "object",
109       properties: {
110         ratingId: { bsonType: "objectId"
111           }, artifactId: { bsonType: "objectId"
112             }, userId: { bsonType: "objectId"
113               }, ratingValue: { bsonType: "double"
114               }
115             }
116           }
117         }
118       }
119     }
120   }
121 }
122 )
123 })
124 db.createCollection("Comment",
125 {
126   validator: {
127     $jsonSchema: { bsonType: "object",
128       properties: {
129         commentId: { bsonType: "objectId"
130           }, artifactId: { bsonType: "objectId"
131             }, userId: { bsonType: "objectId"
132               }, commentText: { bsonType: "string"
133               }
134             }
135           }
136         }
137       }
138     }
139   }
140 }
141 )
142 })
143 db.createCollection("Feedback",
144 {
145   validator: {
146     $jsonSchema: { bsonType: "object",
147       properties: {
148         feedbackId: { bsonType: "objectId"
149           }, artifactId: { bsonType: "objectId"
150             }, userId: { bsonType: "objectId"
151               }, feedbackText: { bsonType: "string"
152               }
153             }
154           }
155         }
156       }
157     }
158   }
159 }
160 )
161 })
162 db.createCollection("Follow",
163 {
164   validator: {
165     $jsonSchema: { bsonType: "object",
166       properties: {
167         followId: { bsonType: "objectId"
168           }, followerId: { bsonType: "objectId"
169             }, followedId: { bsonType: "objectId"
170               }
171             }
172           }
173         }
174       }
175     }
176   }
177 }
178 )
179 })
180 db.createCollection("Like",
181 {
182   validator: {
183     $jsonSchema: { bsonType: "object",
184       properties: {
185         likeId: { bsonType: "objectId"
186           }, artifactId: { bsonType: "objectId"
187             }, userId: { bsonType: "objectId"
188               }
189             }
190           }
191         }
192       }
193     }
194   }
195 }
196 )
197 })
198 db.createCollection("Share",
199 {
200   validator: {
201     $jsonSchema: { bsonType: "object",
202       properties: {
203         shareId: { bsonType: "objectId"
204           }, artifactId: { bsonType: "objectId"
205             }, userId: { bsonType: "objectId"
206               }
207             }
208           }
209         }
210       }
211     }
212   }
213 }
214 )
215 })
216 db.createCollection("Review",
217 {
218   validator: {
219     $jsonSchema: { bsonType: "object",
220       properties: {
221         reviewId: { bsonType: "objectId"
222           }, artifactId: { bsonType: "objectId"
223             }, userId: { bsonType: "objectId"
224               }, reviewText: { bsonType: "string"
225               }
226             }
227           }
228         }
229       }
230     }
231   }
232 }
233 )
234 })
235 db.createCollection("Bookmark",
236 {
237   validator: {
238     $jsonSchema: { bsonType: "object",
239       properties: {
240         bookmarkId: { bsonType: "objectId"
241           }, artifactId: { bsonType: "objectId"
242             }, userId: { bsonType: "objectId"
243               }
244             }
245           }
246         }
247       }
248     }
249   }
250 }
251 )
252 })
253 db.createCollection("CommentLike",
254 {
255   validator: {
256     $jsonSchema: { bsonType: "object",
257       properties: {
258         likeId: { bsonType: "objectId"
259           }, commentId: { bsonType: "objectId"
260             }, userId: { bsonType: "objectId"
261               }
262             }
263           }
264         }
265       }
266     }
267   }
268 }
269 )
270 })
271 db.createCollection("CommentFollow",
272 {
273   validator: {
274     $jsonSchema: { bsonType: "object",
275       properties: {
276         followId: { bsonType: "objectId"
277           }, commentId: { bsonType: "objectId"
278             }, userId: { bsonType: "objectId"
279               }
280             }
281           }
282         }
283       }
284     }
285   }
286 }
287 )
288 })
289 db.createCollection("CommentShare",
290 {
291   validator: {
292     $jsonSchema: { bsonType: "object",
293       properties: {
294         shareId: { bsonType: "objectId"
295           }, commentId: { bsonType: "objectId"
296             }, userId: { bsonType: "objectId"
297               }
298             }
299           }
300         }
301       }
302     }
303   }
304 }
305 )
306 })
307 db.createCollection("CommentReview",
308 {
309   validator: {
310     $jsonSchema: { bsonType: "object",
311       properties: {
312         reviewId: { bsonType: "objectId"
313           }, commentId: { bsonType: "objectId"
314             }, userId: { bsonType: "objectId"
315               }
316             }
317           }
318         }
319       }
320     }
321   }
322 }
323 )
324 })
325 db.createCollection("CommentComment",
326 {
327   validator: {
328     $jsonSchema: { bsonType: "object",
329       properties: {
330         commentId: { bsonType: "objectId"
331           }, commentText: { bsonType: "string"
332             }, userId: { bsonType: "objectId"
333               }
334             }
335           }
336         }
337       }
338     }
339   }
340 }
341 )
342 })
343 db.createCollection("CommentFollow",
344 {
345   validator: {
346     $jsonSchema: { bsonType: "object",
347       properties: {
348         followId: { bsonType: "objectId"
349           }, commentId: { bsonType: "objectId"
350             }, userId: { bsonType: "objectId"
351               }
352             }
353           }
354         }
355       }
356     }
357   }
358 }
359 )
360 })
361 db.createCollection("CommentShare",
362 {
363   validator: {
364     $jsonSchema: { bsonType: "object",
365       properties: {
366         shareId: { bsonType: "objectId"
367           }, commentId: { bsonType: "objectId"
368             }, userId: { bsonType: "objectId"
369               }
370             }
371           }
372         }
373       }
374     }
375   }
376 }
377 )
378 })
379 db.createCollection("CommentReview",
380 {
381   validator: {
382     $jsonSchema: { bsonType: "object",
383       properties: {
384         reviewId: { bsonType: "objectId"
385           }, commentId: { bsonType: "objectId"
386             }, userId: { bsonType: "objectId"
387               }
388             }
389           }
390         }
391       }
392     }
393   }
394 }
395 )
396 })
397 db.createCollection("CommentComment",
398 {
399   validator: {
400     $jsonSchema: { bsonType: "object",
401       properties: {
402         commentId: { bsonType: "objectId"
403           }, commentText: { bsonType: "string"
404             }, userId: { bsonType: "objectId"
405               }
406             }
407           }
408         }
409       }
410     }
411   }
412 }
413 )
414 })
415 db.createCollection("CommentFollow",
416 {
417   validator: {
418     $jsonSchema: { bsonType: "object",
419       properties: {
420         followId: { bsonType: "objectId"
421           }, commentId: { bsonType: "objectId"
422             }, userId: { bsonType: "objectId"
423               }
424             }
425           }
426         }
427       }
428     }
429   }
430 }
431 )
432 })
433 db.createCollection("CommentShare",
434 {
435   validator: {
436     $jsonSchema: { bsonType: "object",
437       properties: {
438         shareId: { bsonType: "objectId"
439           }, commentId: { bsonType: "objectId"
440             }, userId: { bsonType: "objectId"
441               }
442             }
443           }
444         }
445       }
446     }
447   }
448 }
449 )
450 })
451 db.createCollection("CommentReview",
452 {
453   validator: {
454     $jsonSchema: { bsonType: "object",
455       properties: {
456         reviewId: { bsonType: "objectId"
457           }, commentId: { bsonType: "objectId"
458             }, userId: { bsonType: "objectId"
459               }
460             }
461           }
462         }
463       }
464     }
465   }
466 }
467 )
468 })
469 db.createCollection("CommentComment",
470 {
471   validator: {
472     $jsonSchema: { bsonType: "object",
473       properties: {
474         commentId: { bsonType: "objectId"
475           }, commentText: { bsonType: "string"
476             }, userId: { bsonType: "objectId"
477               }
478             }
479           }
480         }
481       }
482     }
483   }
484 }
485 )
486 })
487 db.createCollection("CommentFollow",
488 {
489   validator: {
490     $jsonSchema: { bsonType: "object",
491       properties: {
492         followId: { bsonType: "objectId"
493           }, commentId: { bsonType: "objectId"
494             }, userId: { bsonType: "objectId"
495               }
496             }
497           }
498         }
499       }
500     }
501   }
502 }
503 )
504 })
505 db.createCollection("CommentShare",
506 {
507   validator: {
508     $jsonSchema: { bsonType: "object",
509       properties: {
510         shareId: { bsonType: "objectId"
511           }, commentId: { bsonType: "objectId"
512             }, userId: { bsonType: "objectId"
513               }
514             }
515           }
516         }
517       }
518     }
519   }
520 }
521 )
522 })
523 db.createCollection("CommentReview",
524 {
525   validator: {
526     $jsonSchema: { bsonType: "object",
527       properties: {
528         reviewId: { bsonType: "objectId"
529           }, commentId: { bsonType: "objectId"
530             }, userId: { bsonType: "objectId"
531               }
532             }
533           }
534         }
535       }
536     }
537   }
538 }
539 )
540 })
541 db.createCollection("CommentComment",
542 {
543   validator: {
544     $jsonSchema: { bsonType: "object",
545       properties: {
546         commentId: { bsonType: "objectId"
547           }, commentText: { bsonType: "string"
548             }, userId: { bsonType: "objectId"
549               }
550             }
551           }
552         }
553       }
554     }
555   }
556 }
557 )
558 })
559 db.createCollection("CommentFollow",
560 {
561   validator: {
562     $jsonSchema: { bsonType: "object",
563       properties: {
564         followId: { bsonType: "objectId"
565           }, commentId: { bsonType: "objectId"
566             }, userId: { bsonType: "objectId"
567               }
568             }
569           }
570         }
571       }
572     }
573   }
574 }
575 )
576 })
577 db.createCollection("CommentShare",
578 {
579   validator: {
580     $jsonSchema: { bsonType: "object",
581       properties: {
582         shareId: { bsonType: "objectId"
583           }, commentId: { bsonType: "objectId"
584             }, userId: { bsonType: "objectId"
585               }
586             }
587           }
588         }
589       }
590     }
591   }
592 }
593 )
594 })
595 db.createCollection("CommentReview",
596 {
597   validator: {
598     $jsonSchema: { bsonType: "object",
599       properties: {
600         reviewId: { bsonType: "objectId"
601           }, commentId: { bsonType: "objectId"
602             }, userId: { bsonType: "objectId"
603               }
604             }
605           }
606         }
607       }
608     }
609   }
610 }
611 )
612 })
613 db.createCollection("CommentComment",
614 {
615   validator: {
616     $jsonSchema: { bsonType: "object",
617       properties: {
618         commentId: { bsonType: "objectId"
619           }, commentText: { bsonType: "string"
620             }, userId: { bsonType: "objectId"
621               }
622             }
623           }
624         }
625       }
626     }
627   }
628 }
629 )
630 })
631 db.createCollection("CommentFollow",
632 {
633   validator: {
634     $jsonSchema: { bsonType: "object",
635       properties: {
636         followId: { bsonType: "objectId"
637           }, commentId: { bsonType: "objectId"
638             }, userId: { bsonType: "objectId"
639               }
640             }
641           }
642         }
643       }
644     }
645   }
646 }
647 )
648 })
649 db.createCollection("CommentShare",
650 {
651   validator: {
652     $jsonSchema: { bsonType: "object",
653       properties: {
654         shareId: { bsonType: "objectId"
655           }, commentId: { bsonType: "objectId"
656             }, userId: { bsonType: "objectId"
657               }
658             }
659           }
660         }
661       }
662     }
663   }
664 }
665 )
666 })
667 db.createCollection("CommentReview",
668 {
669   validator: {
670     $jsonSchema: { bsonType: "object",
671       properties: {
672         reviewId: { bsonType: "objectId"
673           }, commentId: { bsonType: "objectId"
674             }, userId: { bsonType: "objectId"
675               }
676             }
677           }
678         }
679       }
680     }
681   }
682 }
683 )
684 })
685 db.createCollection("CommentComment",
686 {
687   validator: {
688     $jsonSchema: { bsonType: "object",
689       properties: {
690         commentId: { bsonType: "objectId"
691           }, commentText: { bsonType: "string"
692             }, userId: { bsonType: "objectId"
693               }
694             }
695           }
696         }
697       }
698     }
699   }
700 }
701 )
702 })
703 db.createCollection("CommentFollow",
704 {
705   validator: {
706     $jsonSchema: { bsonType: "object",
707       properties: {
708         followId: { bsonType: "objectId"
709           }, commentId: { bsonType: "objectId"
710             }, userId: { bsonType: "objectId"
711               }
712             }
713           }
714         }
715       }
716     }
717   }
718 }
719 )
720 })
721 db.createCollection("CommentShare",
722 {
723   validator: {
724     $jsonSchema: { bsonType: "object",
725       properties: {
726         shareId: { bsonType: "objectId"
727           }, commentId: { bsonType: "objectId"
728             }, userId: { bsonType: "objectId"
729               }
730             }
731           }
732         }
733       }
734     }
735   }
736 }
737 )
738 })
739 db.createCollection("CommentReview",
740 {
741   validator: {
742     $jsonSchema: { bsonType: "object",
743       properties: {
744         reviewId: { bsonType: "objectId"
745           }, commentId: { bsonType: "objectId"
746             }, userId: { bsonType: "objectId"
747               }
748             }
749           }
750         }
751       }
752     }
753   }
754 }
755 )
756 })
757 db.createCollection("CommentComment",
758 {
759   validator: {
760     $jsonSchema: { bsonType: "object",
761       properties: {
762         commentId: { bsonType: "objectId"
763           }, commentText: { bsonType: "string"
764             }, userId: { bsonType: "objectId"
765               }
766             }
767           }
768         }
769       }
770     }
771   }
772 }
773 )
774 })
775 db.createCollection("CommentFollow",
776 {
777   validator: {
778     $jsonSchema: { bsonType: "object",
779       properties: {
780         followId: { bsonType: "objectId"
781           }, commentId: { bsonType: "objectId"
782             }, userId: { bsonType: "objectId"
783               }
784             }
785           }
786         }
787       }
788     }
789   }
790 }
791 )
792 })
793 db.createCollection("CommentShare",
794 {
795   validator: {
796     $jsonSchema: { bsonType: "object",
797       properties: {
798         shareId: { bsonType: "objectId"
799           }, commentId: { bsonType: "objectId"
800             }, userId: { bsonType: "objectId"
801               }
802             }
803           }
804         }
805       }
806     }
807   }
808 }
809 )
810 })
811 db.createCollection("CommentReview",
812 {
813   validator: {
814     $jsonSchema: { bsonType: "object",
815       properties: {
816         reviewId: { bsonType: "objectId"
817           }, commentId: { bsonType: "objectId"
818             }, userId: { bsonType: "objectId"
819               }
820             }
821           }
822         }
823       }
824     }
825   }
826 }
827 )
828 })
829 db.createCollection("CommentComment",
830 {
831   validator: {
832     $jsonSchema: { bsonType: "object",
833       properties: {
834         commentId: { bsonType: "objectId"
835           }, commentText: { bsonType: "string"
836             }, userId: { bsonType: "objectId"
837               }
838             }
839           }
840         }
841       }
842     }
843   }
844 }
845 )
846 })
847 db.createCollection("CommentFollow",
848 {
849   validator: {
850     $jsonSchema: { bsonType: "object",
851       properties: {
852         followId: { bsonType: "objectId"
853           }, commentId: { bsonType: "objectId"
854             }, userId: { bsonType: "objectId"
855               }
856             }
857           }
858         }
859       }
860     }
861   }
862 }
863 )
864 })
865 db.createCollection("CommentShare",
866 {
867   validator: {
868     $jsonSchema: { bsonType: "object",
869       properties: {
870         shareId: { bsonType: "objectId"
871           }, commentId: { bsonType: "objectId"
872             }, userId: { bsonType: "objectId"
873               }
874             }
875           }
876         }
877       }
878     }
879   }
880 }
881 )
882 })
883 db.createCollection("CommentReview",
884 {
885   validator: {
886     $jsonSchema: { bsonType: "object",
887       properties: {
888         reviewId: { bsonType: "objectId"
889           }, commentId: { bsonType: "objectId"
890             }, userId: { bsonType: "objectId"
891               }
892             }
893           }
894         }
895       }
896     }
897   }
898 }
899 )
900 })
901 db.createCollection("CommentComment",
902 {
903   validator: {
904     $jsonSchema: { bsonType: "object",
905       properties: {
906         commentId: { bsonType: "objectId"
907           }, commentText: { bsonType: "string"
908             }, userId: { bsonType: "objectId"
909               }
910             }
911           }
912         }
913       }
914     }
915   }
916 }
917 )
918 })
919 db.createCollection("CommentFollow",
920 {
921   validator: {
922     $jsonSchema: { bsonType: "object",
923       properties: {
924         followId: { bsonType: "objectId"
925           }, commentId: { bsonType: "objectId"
926             }, userId: { bsonType: "objectId"
927               }
928             }
929           }
930         }
931       }
932     }
933   }
934 }
935 )
936 })
937 db.createCollection("CommentShare",
938 {
939   validator: {
940     $jsonSchema: { bsonType: "object",
941       properties: {
942         shareId: { bsonType: "objectId"
943           }, commentId: { bsonType: "objectId"
944             }, userId: { bsonType: "objectId"
945               }
946             }
947           }
948         }
949       }
950     }
951   }
952 }
953 )
954 })
955 db.createCollection("CommentReview",
956 {
957   validator: {
958     $jsonSchema: { bsonType: "object",
959       properties: {
960         reviewId: { bsonType: "objectId"
961           }, commentId: { bsonType: "objectId"
962             }, userId: { bsonType: "objectId"
963               }
964             }
965           }
966         }
967       }
968     }
969   }
970 }
971 )
972 })
973 db.createCollection("CommentComment",
974 {
975   validator: {
976     $jsonSchema: { bsonType: "object",
977       properties: {
978         commentId: { bsonType: "objectId"
979           }, commentText: { bsonType: "string"
980             }, userId: { bsonType: "objectId"
981               }
982             }
983           }
984         }
985       }
986     }
987   }
988 }
989 )
990 })
991 db.createCollection("CommentFollow",
992 {
993   validator: {
994     $jsonSchema: { bsonType: "object",
995       properties: {
996         followId: { bsonType: "objectId"
997           }, commentId: { bsonType: "objectId"
998             }, userId: { bsonType: "objectId"
999               }
1000             }
1001           }
1002         }
1003       }
1004     }
1005   }
1006 }
1007 )
1008 })
1009 db.createCollection("CommentShare",
1010 {
1011   validator: {
1012     $jsonSchema: { bsonType: "object",
1013       properties: {
1014         shareId: { bsonType: "objectId"
1015           }, commentId: { bsonType: "objectId"
1016             }, userId: { bsonType: "objectId"
1017               }
1018             }
1019           }
1020         }
1021       }
1022     }
1023   }
1024 }
1025 )
1026 })
1027 db.createCollection("CommentReview",
1028 {
1029   validator: {
1030     $jsonSchema: { bsonType: "object",
1031       properties: {
1032         reviewId: { bsonType: "objectId"
1033           }, commentId: { bsonType: "objectId"
1034             }, userId: { bsonType: "objectId"
1035               }
1036             }
1037           }
1038         }
1039       }
1040     }
1041   }
1042 }
1043 )
1044 })
1045 db.createCollection("CommentComment",
1046 {
1047   validator: {
1048     $jsonSchema: { bsonType: "object",
1049       properties: {
1050         commentId: { bsonType: "objectId"
1051           }, commentText: { bsonType: "string"
1052             }, userId: { bsonType: "objectId"
1053               }
1054             }
1055           }
1056         }
1057       }
1058     }
1059   }
1060 }
1061 )
1062 })
1063 db.createCollection("CommentFollow",
1064 {
1065   validator: {
1066     $jsonSchema: { bsonType: "object",
1067       properties: {
1068         followId: { bsonType: "objectId"
1069           }, commentId: { bsonType: "objectId"
1070             }, userId: { bsonType: "objectId"
1071               }
1072             }
1073           }
1074         }
1075       }
1076     }
1077   }
1078 }
1079 )
1080 })
1081 db.createCollection("CommentShare",
1082 {
1083   validator: {
1084     $jsonSchema: { bsonType: "object",
1085       properties: {
1086         shareId: { bsonType: "objectId"
1087           }, commentId: { bsonType: "objectId"
1088             }, userId: { bsonType: "objectId"
1089               }
1090             }
1091           }
1092         }
1093       }
1094     }
1095   }
1096 }
1097 )
1098 })
1099 db.createCollection("CommentReview",
1100 {
1101   validator: {
1102     $jsonSchema: { bsonType: "object",
1103       properties: {
1104         reviewId: { bsonType: "objectId"
1105           }, commentId: { bsonType: "objectId"
1106             }, userId: { bsonType: "objectId"
1107               }
1108             }
1109           }
1110         }
1111       }
1112     }
1113   }
1114 }
1115 )
1116 })
1117 db.createCollection("CommentComment",
1118 {
1119   validator: {
1120     $jsonSchema: { bsonType: "object",
1121       properties: {
1122         commentId: { bsonType: "objectId"
1123           }, commentText: { bsonType: "string"
1124             }, userId: { bsonType: "objectId"
1125               }
1126             }
1127           }
1128         }
1129       }
1130     }
1131   }
1132 }
1133 )
1134 })
1135 db.createCollection("CommentFollow",
1136 {
1137   validator: {
1138     $jsonSchema: { bsonType: "object",
1139       properties: {
1140         followId: { bsonType: "objectId"
1141           }, commentId: { bsonType: "objectId"
1142             }, userId: { bsonType: "objectId"
1143               }
1144             }
1145           }
1146         }
1147       }
1148     }
1149   }
1150 }
1151 )
1152 })
1153 db.createCollection("CommentShare",
1154 {
1155   validator: {
1156     $jsonSchema: { bsonType: "object",
1157       properties: {
1158         shareId: { bsonType: "objectId"
1159           }, commentId: { bsonType: "objectId"
1160             }, userId: { bsonType: "objectId"
1161               }
1162             }
1163           }
1164         }
1165       }
1166     }
1167   }
1168 }
1169 )
1170 })
1171 db.createCollection("CommentReview",
1172 {
1173   validator: {
1174     $jsonSchema: { bsonType: "object",
1175       properties: {
1176         reviewId: { bsonType: "objectId"
1177           }, commentId: { bsonType: "objectId"
1178             }, userId: { bsonType: "objectId"
1179               }
1180             }
1181           }
1182         }
1183       }
1184     }
1185   }
1186 }
1187 )
1188 })
1189 db.createCollection("CommentComment",
1190 {
1191   validator: {
1192     $jsonSchema: { bsonType: "object",
1193       properties: {
1194         commentId: { bsonType: "objectId"
1195           }, commentText: { bsonType: "string"
1196             }, userId: { bsonType: "objectId"
1197               }
1198             }
1199           }
1200         }
1201       }
1202     }
1203   }
1204 }
1205 )
1206 })
1207 db.createCollection("CommentFollow",
1208 {
1209   validator: {
1210     $jsonSchema: { bsonType: "object",
1211       properties: {
1212         followId: { bsonType: "objectId"
1213           }, commentId: { bsonType: "objectId"
1214             }, userId: { bsonType: "objectId"
1215               }
1216             }
1217           }
1218         }
1219       }
1220     }
1221   }
1222 }
1223 )
1224 })
1225 db.createCollection("CommentShare",
1226 {
1227   validator: {
1228     $jsonSchema: { bsonType: "object",
1229       properties: {
1230         shareId: { bsonType: "objectId"
1231           }, commentId: { bsonType: "objectId"
1232             }, userId: { bsonType: "objectId"
1233               }
1234             }
1235           }
1236         }
1237       }
1238     }
1239   }
1240 }
1241 )
1242 })
1243 db.createCollection("CommentReview",
1244 {
1245   validator: {
1246     $jsonSchema: { bsonType: "object",
1247       properties: {
1248         reviewId: { bsonType: "objectId"
1249           }, commentId: { bsonType: "objectId"
1250             }, userId: { bsonType: "objectId"
1251               }
1252             }
1253           }
1254         }
1255       }
1256     }
1257   }
1258 }
1259 )
1260 })
1261 db.createCollection("CommentComment",
1262 {
1263   validator: {
1264     $jsonSchema: { bsonType: "object",
1265       properties: {
1266         commentId: { bsonType: "objectId"
1267           }, commentText: { bsonType: "string"
1268             }, userId: { bsonType: "objectId"
1269               }
1270             }
1271           }
1272         }
1273       }
1274     }
1275   }
1276 }
1277 )
1278 })
1279 db.createCollection("CommentFollow",
1280 {
1281   validator: {
1282     $jsonSchema: { bsonType: "object",
1283       properties: {
1284         followId: { bsonType: "objectId"
1285           }, commentId: { bsonType: "objectId"
1286             }, userId: { bsonType: "objectId"
1287               }
1288             }
1289           }
1290         }
1291       }
1292     }
1293   }
1294 }
1295 )
1296 })
1297 db.createCollection("CommentShare",
1298 {
1299   validator: {
1300     $jsonSchema: { bsonType: "object",
1301       properties: {
1302         shareId: { bsonType: "objectId"
1303           }, commentId: { bsonType: "objectId"
1304             }, userId: { bsonType: "objectId"
1305               }
1306             }
1307           }
1308         }
1309       }
1310     }
1311   }
1312 }
1313 )
1314 })
1315 db.createCollection("CommentReview",
1316 {
1317   validator: {
1318     $jsonSchema: { bsonType: "object",
1319       properties: {
1320         reviewId: { bsonType: "objectId"
1321           }, commentId: { bsonType: "objectId"
1322             }, userId: { bsonType: "objectId"
1323               }
1324             }
1325           }
1326         }
1327       }
1328     }
1329   }
1330 }
1331 )
1332 })
1333 db.createCollection("CommentComment",
1334 {
1335   validator: {
1336     $jsonSchema: { bsonType: "object",
1337       properties: {
1338         commentId: { bsonType: "objectId"
1339           }, commentText: { bsonType: "string"
1340             }, userId: { bsonType: "objectId"
1341               }
1342             }
1343           }
1344         }
1345       }
1346     }
1347   }
1348 }
1349 )
1350 })
1351 db.createCollection("CommentFollow",
1352 {
1353   validator: {
1354     $jsonSchema: { bsonType: "object",
1355       properties: {
1356         followId: { bsonType: "objectId"
1357           }, commentId: { bsonType: "objectId"
1358             }, userId: { bsonType: "objectId"
1359               }
1360             }
1361           }
1362         }
1363       }
1364     }
1365   }
1366 }
1367 )
1368 })
1369 db.createCollection("CommentShare",
1370 {
1371   validator: {
1372     $jsonSchema: { bsonType: "object",
1373       properties: {
1374         shareId: { bsonType: "objectId"
1375           }, commentId: { bsonType: "objectId"
1376             }, userId: { bsonType: "objectId"
1377               }
1378             }
1379           }
1380         }
1381       }
1382     }
1383   }
1384 }
1385 )
1386 })
1387 db.createCollection("CommentReview",
1388 {
1389   validator: {
1390     $jsonSchema: { bsonType: "object",
1391       properties: {
1392         reviewId: { bsonType: "objectId"
1393           }, commentId: { bsonType: "objectId"
1394             }, userId: { bsonType: "objectId"
1395               }
1396             }
1397           }
1398         }
1399       }
1400     }
1401   }
1402 }
1403 )
1404 })
1405 db.createCollection("CommentComment",
1406 {
1407   validator: {
1408     $jsonSchema: { bsonType: "object",
1409       properties: {
1410         commentId: { bsonType: "objectId"
1411           }, commentText: { bsonType: "string"
1412             }, userId: { bsonType: "objectId"
1413               }
1414             }
1415           }
1416         }
1417       }
1418     }
1419   }
1420 }
1421 )
1422 })
1423 db.createCollection("CommentFollow",
1424 {
1425   validator: {
1426     $jsonSchema: { bsonType: "object",
1427       properties: {
1428         followId: { bsonType: "objectId"
1429           }, commentId: { bsonType: "objectId"
1430             }, userId: { bsonType: "objectId"
1431               }
1432             }
1433           }
1434         }
1435       }
1436     }
1437   }
1438 }
1439 )
1440 })
1441 db.createCollection("CommentShare",
1442 {
1443   validator: {
1444     $jsonSchema: { bsonType: "object",
1445       properties: {
1446         shareId: { bsonType: "objectId"
1447           }, commentId: { bsonType: "objectId"
1448             }, userId: { bsonType: "objectId"
1449               }
1450             }
1451           }
1452         }
1453       }
1454     }
1455   }
1456 }
1457 )
1458 })
1459 db.createCollection("CommentReview",
1460 {
1461   validator: {
1462     $jsonSchema: { bsonType: "object",
1463       properties: {

```

```
72         }
73     }
74 },reviewArray: {
75   bsonType: [
76     "array"
77   ],
78   items: {
79     properties: {
80       rating: { bsonType: "double"
81         },body: { bsonType: "string"
82       },reviewTitle: { bsonType: "string"
83       },reviewId: { bsonType: "objectId"
84       },artifactRef: { bsonType: "objectId"
85     }
86   }
87 }
88 }
89 }
90 }
91 })
92 })
```

Prueba de sentencias mongo en MongoDB

Como se muestra en la figura 7.9 se ha probado crear las colecciones en el manejador MongoDB y se solicita a MongoDB la información sobre la colección creada. Posteriormente, MongoDB muestra correctamente las reglas de validación para la colección Venue.

```

File Edit View Bookmarks Settings Help
~ :mongo — Konsole
> db.createCollection("Venue", {
...   validator: {
...     $jsonSchema: {
...       bsonType: "object",
...       properties: {
...         venueName: { bsonType: "string" }
...       },
...       year: { bsonType: "double" }
...     },
...     venueId: { bsonType: "objectId" }
...   },
...   homepage: { bsonType: "string" }
... },
... country: { bsonType: "string" }
... topics: { bsonType: "string" }
... artifactRefArray: {
...   bsonType: "[array]",
...   items: {
...     properties: {
...       artifactRef: { bsonType: "objectId" }
...     }
...   }
... }
... }
... })
{ "ok": 1 }
> db.getCollectionInfos()
[ {
    "name" : "Venue",
    "type" : "collection",
    "options" : {
      "validator" : {
        "$jsonSchema" : {
          "bsonType" : "object",
          "properties" : {
            "venueName" : {
              "bsonType" : "string"
            },
            "year" : {
              "bsonType" : "double"
            },
            "venueId" : {
              "bsonType" : "objectId"
            },
            "homepage" : {
              "bsonType" : "string"
            },
            "country" : {
              "bsonType" : "string"
            },
            "topics" : {
              "bsonType" : "string"
            },
            "artifactRefArray" : {
              "bsonType" : [
                "array"
              ],
              "items" : {
                "type" : "object"
              }
            }
          }
        }
      }
    }
  ]
}

```

Figura 7.9: Prueba de sentencias del modelo físico orientado a documentos de Venues en MongoDB

7.7. Conclusiones

Como se muestra en este caso de estudio, la aplicación diagrama correctamente el modelo Venues, genera sus sentencias SQL que están probadas en MySQL, también genera correctamente el modelo conceptual, lógico y físico de un modelo NoSQL. Se diagrama el modelo lógico y se prueban las sentencias del modelo físico en MongoDB.

Capítulo 8

Trabajo a futuro

Se consideran las siguientes acciones como trabajo a futuro:

- La mutación de los elementos del diagrama ER con la ayuda de un menú contextual.
- La inclusión de atributos parciales para ejemplos de una complejidad mayor en los diagramas ER.
- El manejo de atributos compuestos para ampliar las posibilidades para diagramas que acepta la aplicación.
- Robustecer la base de datos de la aplicación
 - Almacenar mas de un diagrama ER por usuario.
 - Guardar las consultas de acceso para cada diagrama ER.
- Una forma mas amigable para el usuario al momento de crear sus consultas de acceso para los modelos NoSQL.
- Considerar otros modelos NoSQL como el orientado a columnas

Apéndice A

Apéndice

A.1. Unified Modeling Language

De acuerdo con Fowler[79], UML (*Unified Modeling Language*) es una familia de notaciones gráficas, respaldada por un metamodelo único, que ayuda a describir y diseñar sistemas de *software*, particularmente sistemas de *software* descritos utilizando el estilo orientado a objetos.

A.1.1. Diagramas de clases

De acuerdo con Fowler[79], un diagrama de clases describe los tipos de objetos en el sistema y los diversos tipos de relaciones estáticas que existen entre ellos.

Los diagramas de clase también muestran las propiedades y operaciones de una clase y las restricciones que se aplican a la forma en que se conectan los objetos.

UML usa el término característica como un término general que cubre las propiedades y operaciones de una clase; proporciona mecanismos para representar los miembros de la clase, como atributos y métodos, así como información adicional sobre ellos.

Visibilidad Para especificar la visibilidad de un miembro de la clase (es decir, cualquier atributo o método), se coloca uno de los siguientes signos delante de ese miembro:

1. + Público.
2. - Privado.
3. # Protegido.
4. / Derivado (se puede combinar con otro).
5. ~ Paquete.

Ámbitos UML especifica dos tipos de ámbitos para los miembros: instancias y clasificadores y estos últimos se representan con nombres subrayados.

1. Los miembros clasificadores se denotan comúnmente como “estáticos” en muchos lenguajes de programación; su ámbito es la propia clase.
 - a) Los valores de los atributos son los mismos en todas las instancias.
 - b) La invocación de métodos no afecta al estado de las instancias.
2. Los miembros instancias tienen como ámbito una instancia específica.
 - a) Los valores de los atributos pueden variar entre instancias.
 - b) La invocación de métodos afecta al estado de las instancias (es decir, cambiar el valor de sus atributos).

Para indicar que un miembro posee un ámbito de clasificador, hay que subrayar su nombre; de lo contrario, se asume por defecto que tendrá ámbito de instancia.

La figura A.1 es una asociación entre dos clases, la clase *Person* y la clase *Magazine*.

La figura A.2 es una agregación entre dos clases, la clase *Professor* y la clase *Class* donde la cardinalidad es una a muchos.

La figura A.3 muestra una asociación y composición entre clases, la clase Almacén está asociada a la clase Cuentas y con la clase Cliente hay una relación de composición.

Relaciones Una relación es un término general que abarca los tipos específicos de conexiones lógicas que se encuentran en los diagramas de clases y objetos. UML presenta las siguientes relaciones:

Enlace Un enlace es la relación más básica entre objetos.

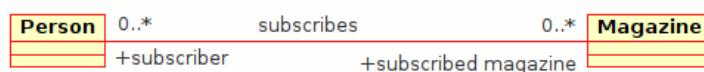


Figura A.1: Asociación

Asociación Una asociación como la de la figura A.1 representa una familia de enlaces; una asociación binaria (entre dos clases) normalmente se representa con una línea continua; una misma asociación relaciona cualquier número de clases y una asociación que relate tres clases se llama asociación ternaria.

A una asociación se le asigna un nombre y en sus extremos se hacen indicaciones, como el rol que desempeña la asociación, los nombres de las clases relacionadas, su multiplicidad, su visibilidad y otras propiedades.

Hay cuatro tipos diferentes de asociación: bidireccional, unidireccional, agregación (en la que se incluye la composición) y reflexiva; las asociaciones unidireccional y bidireccional son las más comunes.

Agregación

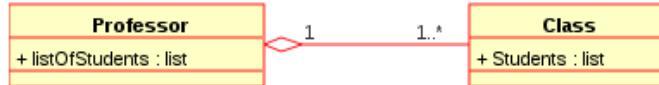


Figura A.2: Agregación

La agregación o agrupación como la de la figura A.2 es una variante de la relación de asociación “tiene un”: la agregación es más específica que la asociación; se trata de una asociación que representa una relación de tipo parte-todo o parte-de.

Al ser un tipo de asociación, una agregación es posible que tenga un nombre y las mismas indicaciones en los extremos de la línea; sin embargo, una agregación no debe incluir más de dos clases, debe ser una asociación binaria.

Una agregación es posible que se dé cuando una clase es una colección o un contenedor de otras clases, pero a su vez, el tiempo de vida de las clases contenidas no tienen una dependencia fuerte del tiempo de vida de la clase contenedora (del todo).

En UML, como está en la figura A.2 se representa gráficamente con un rombo hueco junto a la clase contenedora con una línea que lo conecta a la clase contenida; todo este conjunto es, semánticamente, un objeto extendido que es tratado como una única unidad en muchas operaciones, aunque físicamente está hecho de varios objetos más pequeños.

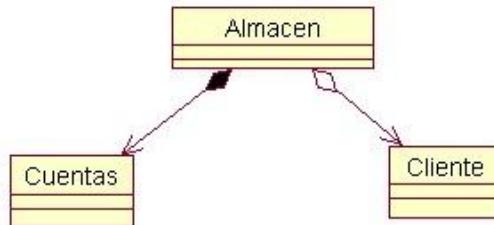


Figura A.3: Asociación rombo sin rellenar y composición rombo negro

Composición

La representación en UML de una relación de composición es mostrada en la figura A.3 como un diamante relleno del lado de la clase contenedora, es decir, al final de la línea que conecta la clase contenido con la clase contenedora.

Diferencias entre Composición y Agregación

Relación de Composición

1. Cuando se intenta representar un todo y sus partes.
2. Cuando se elimina el contenedor, el contenido también es eliminado.

Relación de Agrupación

1. Cuando se representan las relaciones en un *software* o base de datos.
2. Cuando el contenedor es eliminado, el contenido usualmente no es destruido.

Bibliografía

- [1] Google, *Google Trends - NoSQL*, 2020. dirección: <https://trends.google.es/trends/explore?date=all&q=NoSQL>.
- [2] E. F. Codd, «A relational model of data for large shared data banks,» *Communications of the ACM*, vol. 13, DOI: [10.1145/362384.362685](https://doi.org/10.1145/362384.362685).
- [3] C. Li, «Transforming relational database into HBase: A case study,» en *2010 IEEE International Conference on Software Engineering and Service Sciences*, jul. de 2010, págs. 683-687. DOI: [10.1109/ICSESS.2010.5552465](https://doi.org/10.1109/ICSESS.2010.5552465).
- [4] A. Chebotko, A. Kashlev y S. Lu, «A Big Data Modeling Methodology for Apache Cassandra,» en *2015 IEEE International Congress on Big Data*, jun. de 2015, págs. 238-245. DOI: [10.1109/BigDataCongress.2015.41](https://doi.org/10.1109/BigDataCongress.2015.41).
- [5] M. J. Mior, K. Salem, A. Aboulnaga y R. Liu, «NoSE: Schema Design for NoSQL Applications,» *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, n.º 10, págs. 2275-2289, oct. de 2017, ISSN: 2326-3865. DOI: [10.1109/TKDE.2017.2722412](https://doi.org/10.1109/TKDE.2017.2722412).
- [6] D. Martinez-Mosquera, R. Navarrete y S. Lujan-Mora, «Modeling and Management Big Data in Databases—A Systematic Literature Review,» *Sustainability*, 2020. DOI: <https://doi.org/10.3390/su12020634>.
- [7] J. Dullea, I.-Y. Song e I. Lamprou, «An analysis of structural validity in entity-relationship modeling,» *Data & Knowledge Engineering*, vol. 47, n.º 2, págs. 167-205, 2003, Publisher: Elsevier.
- [8] A. de la Vega, D. García-Saiz, C. Blanco, M. Zorrilla y P. Sánchez, «Mortadelo: Automatic generation of NoSQL stores from platform-independent data models,» *Future Generation Computer Systems*, vol. 105, págs. 455-474, 2020, Publisher: Elsevier.
- [9] C. de Lima y R. dos Santos Mello, «A workload-driven logical design approach for NoSQL document databases,» en *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, 2015, págs. 1-10.
- [10] datafluent, *Kashliev Data Modeler*, 2020. dirección: <https://www.datafluent.org/>.
- [11] Amazon, *NoSQL Workbench for Amazon DynamoDB*, 2020. dirección: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/workbench.html>.
- [12] Hackolade, *Hackolade*, 2020. dirección: <https://hackolade.com/>.

-
- [13] Ramez Elmasri, *Fundamentos de Sistemas de Bases de Datos*. Pearson, ISBN: 84-7829-085-0.
 - [14] P. Chen, *The entity-relationship model: Toward a unified View of data*. dirección: <https://www.citeSeerX.ist.psu.edu/viewdoc/summary?doi=10.1.1.523.6679>.
 - [15] Cristina Marta Bender, Claudia Deco, Juan Sebastián González Sanabria, María Hallo y Julio César Ponce Gallegos, *Tópicos Avanzados de Bases de Datos*, 1.^a ed. Iniciativa Latinoamericana de Libros de Texto Abiertos.
 - [16] C. Coronel y S. Morris, *Database Systems: Design, Implementation, and Management*, 13.^a ed. Cengage, ISBN: 978-1-337-62790-0.
 - [17] P. J. Sadalage y M. Fowler, *Nosql Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 1.^a ed. Pearson Education, ISBN: 978-0-321-82662-6.
 - [18] G. Scherp, *A framework for model-driven scientific workflow engineering*. BoD–Books on Demand, 2013.
 - [19] R. Reussner y W. Hasselbring, *Handbuch der Software-Architektur*. dpunkt.verlag, 2006.
 - [20] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.
 - [21] A. Van Deursen, P. Klint y J. Visser, «Domain-specific languages: An annotated bibliography,» *ACM Sigplan Notices*, vol. 35, n.^o 6, págs. 26-36, 2000.
 - [22] S. Kapferer, «Model Transformations for DSL Processing,» PhD Thesis, HSR, 2019.
 - [23] *HTTP - Hypertext Transfer Protocol*. dirección: <https://www.w3.org/Protocols/>.
 - [24] *HTML*. dirección: <https://developer.mozilla.org/es/docs/Web/HTML>.
 - [25] H. W. Lie y B. Bos, *Cascading style sheets: Designing for the web, Portable Documents*. Addison-Wesley Professional, 2005.
 - [26] *What is CSS?* Dirección: <https://www.w3.org/standards/webdesign/htmlcss#whatcss>.
 - [27] *Documentation Sass*. dirección: <https://sass-lang.com/documentation>.
 - [28] *Most Loved, Dreaded, and Wanted Web Frameworks - stack overflow*. dirección: <https://insights.stackoverflow.com/survey/2019#technology--most-loved-dreaded-and-wanted-web-frameworks>.
 - [29] *JavaScript*. dirección: <https://developer.mozilla.org/es/docs/Web/JavaScript>.
 - [30] *stack overflow - Developer Survey Results 2019 - Most Popular Technologies*. dirección: <https://insights.stackoverflow.com/survey/2019#most-popular-technologies>.
 - [31] Wired, *Wired - Get Started with Web Frameworks*, 2020. dirección: https://www.wired.com/2010/02/get_started_with_web_frameworks/.

-
- [32] React, *React - A JavaScript library for building user interfaces*, 2020. dirección: <https://reactjs.org/>.
 - [33] Angular, *Angular - One framework, mobile & desktop*, 2020. dirección: <https://angular.io/>.
 - [34] ¿Qué es Vue.js? Dirección: <https://es.vuejs.org/v2/guide/>.
 - [35] What is NuxtJS? Dirección: <https://nuxtjs.org/guide>.
 - [36] Wikipedia contributors, *CSS framework — Wikipedia, The Free Encyclopedia*. 2020. dirección: https://en.wikipedia.org/w/index.php?title=CSS_framework&oldid=952876853.
 - [37] *Introduction Material Design*. dirección: <https://material.io/design/introduction#goals>.
 - [38] *Vuetify Component API Overview*. dirección: <https://vuetifyjs.com/en/components/api-explorer/>.
 - [39] *Documentation Bootstrap*. dirección: <https://getbootstrap.com/docs/4.5/getting-started/introduction/>.
 - [40] MongoDB, *MongoDB - MongoDB Documentation*, 2020. dirección: <https://docs.mongodb.com/manual/>.
 - [41] MySQL, *MySQL - MySQL Documentation*, 2020. dirección: <https://dev.mysql.com/doc/>.
 - [42] *GoJs*. dirección: <https://gojs.net/latest/index.html>.
 - [43] *Fabric Documentation*, 2020. dirección: <http://fabricjs.com/fabric-intro-part-1>.
 - [44] D3, *D3.js - D3.js Documentation*, 2020. dirección: <https://github.com/d3/d3/wiki>.
 - [45] M. Lutz, *Learning Python*, 5.^a ed. O'Reilly, jun. de 2013, ISBN: 978-1-4493-5573-9.
 - [46] P. Projects, *Flask - Flask web development, one drop at a time*. 2020. dirección: <https://flask.palletsprojects.com/en/1.1.x/>.
 - [47] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel y D. Varró, «Survey and classification of model transformation tools,» *Software & Systems Modeling*, vol. 18, n.^o 4, págs. 2361-2397, 2019, Publisher: Springer.
 - [48] B. Joy, G. Steele, J. Gosling y G. Bracha, *The Java language specification*. Addison-Wesley Reading, 2000.
 - [49] I. Eclipse, «Eclipse IDE,» *Website www.eclipse.org* Last visited: July, 2009.
 - [50] PyEcore, *PyEcore*. 2020. dirección: <https://pyecore.readthedocs.io/en/latest/>.
 - [51] U. C. l. Ángeles, *Metodología de desarrollo de software*. 2020. dirección: <https://www.uladech.edu.pe/images/stories/universidad/documentos/2018/metodologia-desarrollo-software-v001.pdf>.
 - [52] A. N. Cadavid, J. D. F. Martínez y J. M. Vélez, «Revisión de metodologías ágiles para el desarrollo de software,» *Prospectiva*, vol. 11, n.^o 2, págs. 30-39, 2013, Publisher: Universidad Autónoma del Caribe.

-
- [53] T. S. Guide, *The Definitive Guide to Scrum: The Rules of the Game*. 2020. dirección: <https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf#zoom=100>.
 - [54] I. Sommerville, «Software engineering 9th Edition,» *ISBN-10*, vol. 137035152, 2011.
 - [55] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
 - [56] W. Stallings, *Operating systems: internals and design principles*. Boston: Prentice Hall, 2012.
 - [57] K. Wilson, «About Windows,» en *Everyday Computing with Windows 8.1*, Springer, 2015, págs. 1-4.
 - [58] R. Love, *Linux kernel development*. Pearson Education, 2010.
 - [59] A. Aaby, «Introduction to programming languages,» *Computer Science Department, Walla Walla College*. <http://www.worldcolleges.info/sites/default/files/aaby.pdf>, 1996.
 - [60] G. Van Rossum y col., «Python Programming Language.,» en *USENIX annual technical conference*, vol. 41, 2007, pág. 36.
 - [61] A. Hejlsberg, S. Wiltamuth y P. Golde, *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.
 - [62] T. M. Connolly y C. E. Begg, *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005.
 - [63] P. DuBois y M. Foreword By-Widenius, *MySQL*. New riders publishing, 1999.
 - [64] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.
 - [65] A. Abran, *Applied Software Measurement: Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress*. Shaker, 2006.
 - [66] P. Galván, *Estudio de salarios SG 2020*. 2020. dirección: <https://sg.com.mx/estudios/salarios/2020>.
 - [67] S. México, *SCRUM México - Escribiendo Historias de Usuario*. 2020. dirección: <https://www.scrum.mx/informe/historias-de-usuario>.
 - [68] M. T. Gallego, *Metodología SCRUM*. 2020. dirección: <http://openaccess.uoc.edu/webapps/o2/bitstream/10609/17885/1/mtrigasTFC0612memoria.pdf>.
 - [69] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, *Introduction to algorithms*. MIT press, 2009.
 - [70] V. Paradigm, *Visual Paradigm - What is Class Diagram?* 2020. dirección: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/>.
 - [71] D.-P. Pop y A. Altar, «Designing an MVC model for rapid web application development,» *Procedia Engineering*, vol. 69, págs. 1172-1179, 2014, Publisher: Elsevier.

-
- [72] K. G. Pope y S. Krasner, «A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80,» *Journal of Object-Oriented Programming*, vol. 1, 1988.
 - [73] Netlify, *Netlify - Netlify Security*. dirección: <https://www.netlify.com/security/>.
 - [74] JWT, *Web Tokens*, 2020. dirección: <https://jwt.io/introduction/>.
 - [75] F. Shahzad, T. R. Sheltami, E. M. Shakshuki y O. Shaikh, «A review of latest web tools and libraries for state-of-the-art visualization,» *Procedia Computer Science*, vol. 98, págs. 100-106, 2016, Publisher: Elsevier.
 - [76] Heroku, *Heroku - Heroku Security*, 2020. dirección: <https://www.heroku.com/policy/security>.
 - [77] Pallets Projects - Flask. 2020. dirección: <https://palletsprojects.com/p/flask/>.
 - [78] Flask-RESTPlus. 2020. dirección: <https://flask-restplus.readthedocs.io/en/stable/>.
 - [79] M. Fowler y U. Distilled, *A brief guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.