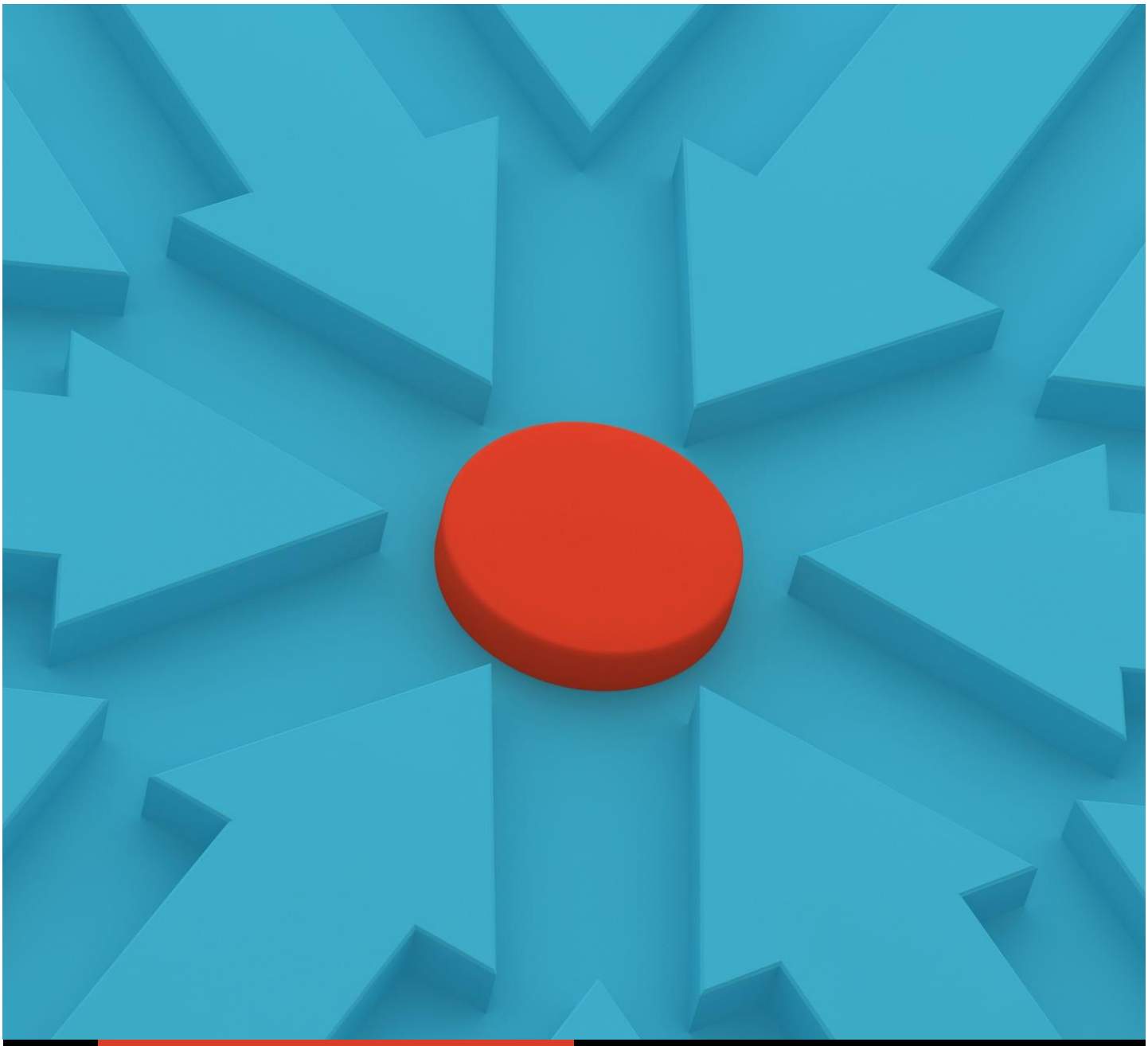


TEMA 04 BASES DE DATOS NOSQL

PRACTICA 04: ACCESDATA SL V2.0

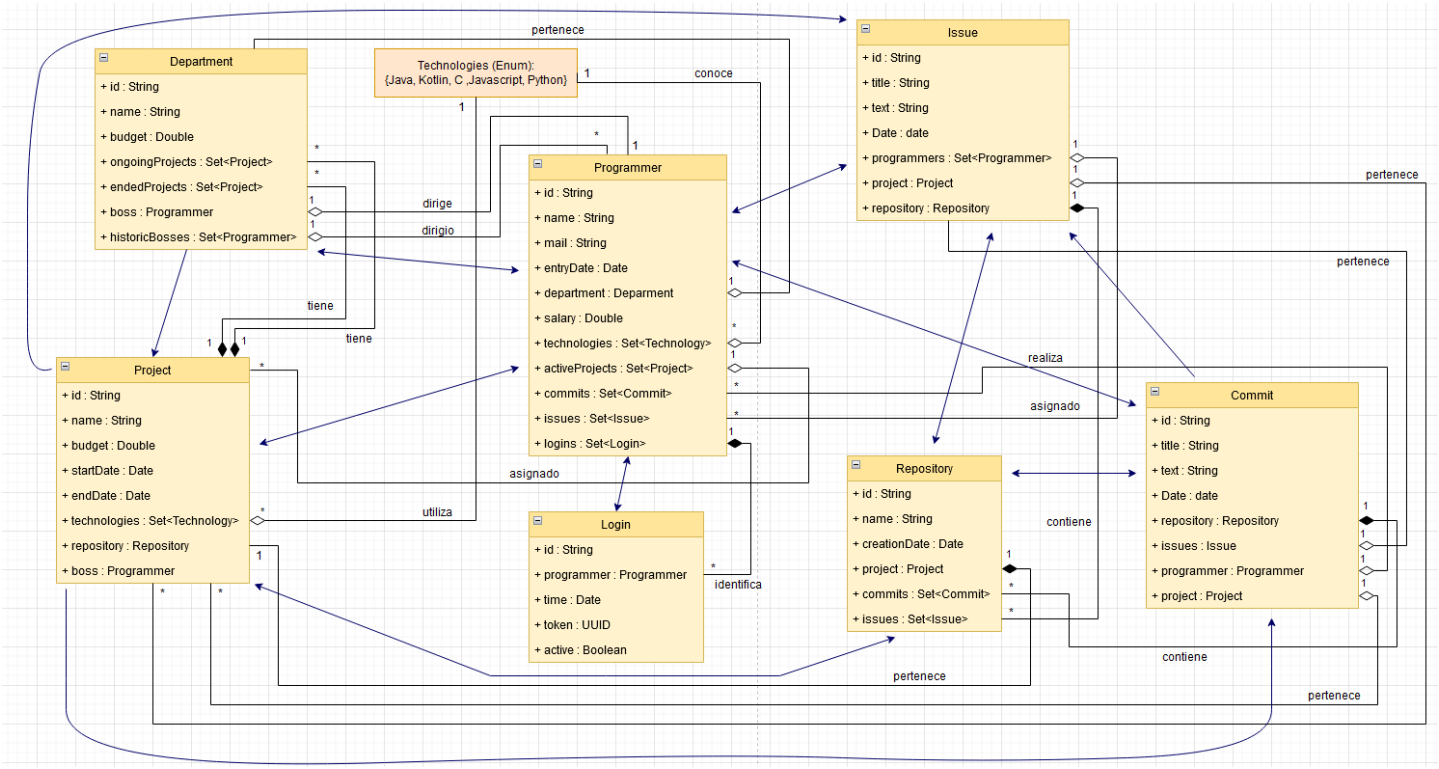
SERGIO PEREZ Y FEDERICO TOLEDO

2DAM – Acceso a Datos

Contenido

Diagrama de Clases 3
Análisis de propuestas NoSQL: 3
Entidades: 5
Programador: 5
Departamento: 6
Proyecto: 6
Login: 7
Issue: 7
Commit: 7
Repositorio: 8
Patrones de diseño implementados: 9

DIAGRAMA DE CLASES



ANÁLISIS DE PROPUESTAS NOSQL:

Decidimos intentar mantener un diagrama similar al de la practica anterior eliminando las entidades intermedias de las relaciones de muchos a muchos para mantener la bidireccionalidad dentro de lo razonable. Dejando así las entidades principales con colecciones y/o unidad atómicas de las otras entidades relacionadas pudiendo acceder y consultar las inmediatamente relacionadas entre sí.

En esta practica buscamos implementar un diseño propio de bases de datos NoSQL. Las opciones que barajamos incluyen:

- Un diseño en árbol por nodos y herencia.
- Un diseño con documentos embebidos.
- Un diseño segmentado en documentos y subdocumentos.
- Y por último diseño de referencias a claves primarias.

Dada la premisa de que el proyecto original estaba basado en SQL encontramos muchas dificultades para adaptar la solución a diseño por arboles jerárquico. El principal motivo para ello es el exceso de relaciones entre entidades y la ausencia de herencia entre ellas en todo el problema. Este es el motivo por el que descartamos el primer diseño.

La solución de documentos embebidos parecía bastante atractiva sobre el papel puesto que nos permitía hacer consultas muy amplias y dinámicas. Sin embargo, tras intentar implementarlo nos percatamos de que gran parte de las relaciones de la practica multiplicarían la información y además descubrimos que tenemos mas peso en escritura que en lectura. Si bien las ventajas en Lectura eran increíbles, los esfuerzos que implicarían en las operaciones de escritura nos llevaron a explorar otros diseños.

Un diseño similar a la de documentos embebidos fue la de segmentación en documentos y subdocumentos. Ofrecía una solución embebida que se adecuaba más a nuestro problema puesto

que en un comienzo no buscábamos borrar colecciones. Sin embargo, conforme avanzo la practica este deseo mostro ser contrario a los requerimientos del problema puesto que sí buscamos borrar cualquier entidad del diagrama. Esto deshizo la ventaja principal de este modelo lo que nos llevó a descartarlo.

Por ultimo se presenta el modelo mas cercano a las bases de datos relacionales. Desde un principio la interfaz de JPA nos premiaba al trabajar con este modelo. Adicionalmente, el problema originalmente diseñado para una base de datos SQL se mostraba especialmente afín a la propagación de claves entre entidades.
De esta manera, tras descartar la opción de embeber documentos optamos por implementar este ultimo diseño. La desventaja principal con la que nos hemos enfrentado es la lectura de datos, puesto que teníamos que ensamblar muchos datos para ofrecer una salida adecuada.
Esta desventaja es notablemente mas llevadera en consonancia con la facilidad que nos ofrece al tener relaciones muchos a muchos y no tener que mantener la integridad de todos los campos sincronizada.

Una vez explicado el motivo de nuestra elección pasamos a explicar el modelo de datos elegido:

ENTIDADES:

Programador:

El programador es la unidad atómica de este problema. Idealmente en una empresa solo se borra programadores al despedirlo y se relaciona con prácticamente todas las otras entidades. De esta forma la entidad programador presenta bidireccionalidad con muchas de las entidades puesto que va a ser un punto de entrada muy común. También hay que destacar el hecho de que tanto de la entidad Departamento como de la entidad Proyecto, conviven en la misma colección tanto jefes (programadores que no participan como trabajadores de este) como programadores normales complicando mucho las relaciones y la sincronización de la entidad.

Un programador tiene datos básicos como nombre, correo electrónico, fecha de ingreso y salario. También tiene contraseña que no debe aparecer en consultas.

Las relaciones que presenta incluyen:

- De jefes a Departamento y a Proyecto
- Además, con el histórico de jefes en Departamento
- Departamento al que pertenece (Muchos a Uno)
- Tecnologías que domina (Uno a Muchos)
- Proyectos en los que participa (Muchos a Muchos)
- Commits realizados (Uno a Muchos)
- Issues a los que se encuentra asignado (Muchos a Muchos)
- Y historial de accesos mediante identificación en Login (Uno a Muchos)

Presenta bidireccionalidad con:

- Departamento en cuestiones de jefes
- También en Departamento con histórico de jefes
- Proyecto en jefe de proyecto
- En historial de accesos en Login
- En Issues tanto como jefe siendo autor del Issue como los programadores asignados
- En Commit como el autor del Commit

Las operaciones en cascada de Programador solo se propagan al historial de accesos puestos que las demás entidades no están compuestas por programador.

Departamento:

Es la entidad superior en la jerarquía, puesto que de ella dependen otras, pero ella no depende de nadie. De un departamento solo nos interesa saber los proyectos (tanto activos como finalizados), sus programadores y los jefes (tanto el actúa como aquellos que lo han sido con anterioridad). Sin embargo, los programadores decidimos no incluirlos en esta entidad ya que no nos era muy costoso ensamblar estos datos a la salida, mientras que su inclusión suponía un gran problema en las demás operaciones.

Los datos básicos que posee son su nombre y su presupuesto anual.

Las relaciones que presenta son:

- Proyectos activos con Proyectos (Uno a Muchos)
- Proyectos terminados con Proyectos (Uno a Muchos)
- Jefe de Departamento con programador (Uno a Uno)
- Histórico de jefes de Departamento con Programador (Muchos a Muchos)
- Programadores que pertenecen a un Departamento (Uno a Muchos)

Presenta bidireccionalidad con:

- Como jefe con Programador

Las operaciones en cascada de Departamento se propagan a Proyecto, no deseamos tener un Proyecto sin Departamento. Sin embargo, si queremos Programadores que puedan no pertenecer a departamento de modo que no propagamos las operaciones hacia ellos.

Proyecto:

Entidad directamente relacionada con Departamento, por lo que depende ella. Al igual que esta última presenta un jefe que es un Programador.

Los datos básicos que posee son su nombre, su presupuesto anual, fecha de inicio y fecha de fin.

Las relaciones que presenta son:

- Proyectos activos con Departamento (Muchos a Uno)
- Proyectos terminados con Departamento (Muchos a Uno)
- Jefe de proyecto con Programador (Uno a Uno)
- Repositorio propio del Proyecto (Uno a Uno)
- Tecnologías que usadas (Uno a Muchos)
- Programadores que participan en un Proyecto activo (Uno a Muchos)
- Issues de un Proyecto (Uno a Muchos)
- Commits de un Proyecto (Uno a Muchos)

Presenta bidireccionalidad con:

- Con Repositorio
- Con Programador

Las operaciones en cascada de Proyecto se propagan a Repositorio.

Login:

Es la entidad nueva introducido en esta practica respecto a la anterior. Depende solo y directamente de programador. Presenta un Token generado por el programa que debe estar autenticado para su validación junto a el mail y contraseña (encriptada en SHA256) del programador.

Los datos básicos que posee son su fecha en la que se produjo, la UUID del Token, y si el programador esta activo o no.

Las relaciones que presenta son:

- Historial de acceso de Programadores que se identifican (Uno a Muchos)

Presenta bidireccionalidad con:

- Historial de acceso de Programadores

Las operaciones en cascada de Login no se propagan.

Issue:

Entidad que presenta propiedades especiales puesto que solo puede ser insertada por un Programador que es jefe de un Proyecto. Además, los Commit se relacionan con Issues puesto que las cierran.

Los datos básicos que posee son el título del Issue, el texto del cuerpo y su fecha.

Las relaciones que presenta son:

- Proyecto al que pertenece (Uno a Muchos)
- Programadores asignados (Muchos a Uno)
- Repositorio al que pertenece (Uno a Muchos)
- Commit del Issue (Uno a Uno)

Presenta bidireccionalidad con:

- Con Repositorio
- Con Programador

Las operaciones en cascada de Issue no se propagan.

Commit:

Esta entidad no genera dependencias a ninguna de las otras entidades. Sin embargo, esta compuesta por varias entidades, convirtiéndose en la entidad más débil. En su inserción cierra una Issue

Los datos básicos que posee son el título del Commit, el texto del cuerpo y su fecha.

Las relaciones que presenta son:

- Proyecto al que pertenece (Uno a Muchos)
- Programador autor (Uno a Muchos)
- Repositorio al que pertenece (Uno a Muchos)
- Issue del Commit (Uno a Uno)

Presenta bidireccionalidad con:

- Con Repositorio
- Con Programador

Las operaciones en cascada de Commit no se propagan.

Repositorio:

Entidad intermediaria entre las entidades Proyecto y Commits e Issues.

Los datos básicos que posee son su nombre y su fecha de creación.

Las relaciones que presenta son:

- Proyecto al que pertenece (Uno a Uno)
- Con Commits (Muchos a Uno)
- Con Issue (Muchos a Uno)

Presenta bidireccionalidad con:

- Con Commit
- Con Issue
- Con Proyecto

Las operaciones en cascada de Repositorio se propagan a Commit e Issue.

PATRONES DE DISEÑO IMPLEMENTADOS:

Decidimos para esta práctica usar Docker de MongoDB con JPA e Hibernate OGM JTA.
Hemos optado por utilizar clases controller para separar y reutilizarlas, por ejemplo, para la conexión con Hibernate.
En esta práctica hemos aprendido a utilizar la inyección de dependencias, patrón muy importante para librerías como Mockito y en general el desacoplamiento del código.
También hemos utilizado una arquitectura de tres niveles para gestión, transporte y conversión de la base de datos.
Para nuestra clase principal hemos implementado un patrón Fachada, lo que nos permite simplificar el código para rápida lectura y separarlo.
Por último, hemos utilizado el patrón Singleton para asegurar que los controladores no se duplican y que no pueda haber más de una instancia de clase principal.