Implementación de persistencia en un modelo relacional aplicado a un sistema de Seguimiento de Items

Objetivos del trabajo práctico

El desarrollo de este trabajo práctico reúne, bajo mi perspectiva, dos tipos de objetivos: uno tecnológico y otro personal.

Objetivo tecnológico

El objetivo tecnológico de este trabajo práctico es implementar un mapeo entre la aplicación orientada a objetos y la persistencia de los datos que esta administra, en forma tal que sea lo más transparente posible a la estrategia de persistencia elegida. En este caso, la estrategia elegida es mapeo objeto – relacional. Este objetivo podría especificarse según el siguiente detalle:

- Que pueda cambiarse el paradigma de persistencia utilizado con el menor impacto posible en el código, sin mayores costos.
- Al elegir el mapeo objeto – relacional para este desarrollo, que la aplicación sea transparente – también – a distintos motores de bases de datos relacionales, o sea, que pueda cambiarse de DBMS sin generar retrabajo.
- Los dos puntos anteriores implican diseñar la aplicación siguiendo el principio de design for change.
- Tratar de minimizar – en lo posible no usar – código SQL embebido en la aplicación desarrollada bajo el paradigma objetos. Esto implica minimizar su uso incluso en annotations (@Query).
- Que la aplicación provea mecanismos de control de concurrencia y que estos sean transparentes respecto al código orientado a objetos.
- Que el desarrollo se base en uso de frameworks ampliamente conocidos y validados para bajar los tiempos de desarrollo y asegurar calidad y entendibilidad del código.
- Que el código resulte ordenado, legible, entendible y extendible, o sea mantenible a bajo costo.

Objetivo personal

La elección del objetivo tecnológico referido al mapeo objeto-relacional, conlleva también un objetivo personal. En los proyectos en los que intervengo en la vida profesional, la persistencia se realiza preferentemente en DBMS relacionales. Por lo tanto, otro objetivo de este trabajo, es profundizar sobre técnicas y frameworks que permitan mejorar la arquitectura de los proyectos de la industria en los que participo.

Problema

Objetivo:

Diseñar un modelo que permita soportar la funcionalidad básica de una herramienta de tipo workflow para el seguimiento y control de ítems (pedidos de cambios y requerimientos) de un proyecto de software.

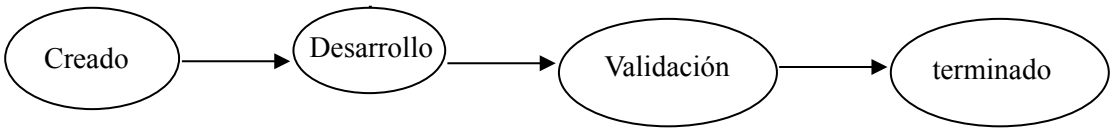
Introducción:

El propósito de esta herramienta es el de ser utilizada en proyectos de desarrollo de software para formalizar la manera en la que se informan y manipulan los diferentes ítems que se van creando a lo largo de la vida de un proyecto.

Los ítems se clasifican utilizando un "tipo" (reporte de bug, ampliación, mejora, nuevo requerimiento, etc.) para poder asignarle el equipo correcto. Por ejemplo, en caso de ser un reporte de bug, permitirá asignar un equipo que haya participado en el proceso de desarrollo. La herramienta debe permitir la creación de nuevos tipos de items. Estos tipos varían de proyecto en proyecto.

El ciclo de vida de un ítem está determinado por la secuencia de estados por los que puede pasar, por ejemplo: Creado, Análisis, En Desarrollo, Testing, Evaluación Usuario, Aceptado, Finalizado. Los estados deben ser configurados por el líder de proyecto teniendo en cuenta los diferentes tipos de ítems y los canales más eficientes para su resolución.

De cada tipo de ítem se conocen las posibles secuencias que puede seguir un ítem clasificado con este tipo. Por ejemplo el siguiente gráfico describe las posibles secuencias de estados que puede seguir un ítem clasificado como "reporte de bug".



Teniendo en cuenta la figura anterior, cabe destacar que no todos los ítems clasificados con el tipo "reporte de bug" necesariamente recorrerán la misma secuencia de estados. El camino que "realice" cada ítem estará definido en última instancia por el responsable del ítem.

De cada ítem, se conoce su prioridad, su tipo, su estado actual, y la secuencia de estados por la que pasó. El conjunto de posibles estados a los que puede pasar un ítem en un momento dado depende del tipo de ítem y del estado actual.

Cuando un ítem pasa por un estado, se le debe asignar un responsable. Puede ocurrir que el responsable de un ítem en un estado sea reasignado, con lo cual otro miembro del proyecto deberá ocupar su puesto. Es necesario poder conocer quien fue el responsable de un ítem en un estado dada una fecha. Los responsables de los ítems siempre serán miembros del proyecto.

Análisis de Requerimientos

Alcance

El propósito de esta herramienta es dar soporte al seguimiento y control de ítems de un proyecto de software. Básicamente, los ítems involucrados son pedidos de cambios y requerimientos. La herramienta permitirá crear y configurar mecanismos de control y seguimiento generales para estos tipos de ítems como así también efectuar el control y seguimiento de instancias específicas de los mismos.

Lista de requerimientos funcionales

Para el problema planteado, se han detectado los siguientes requerimientos funcionales:

1) Administración de tipos de ítems.
2) Administración de tipos de estados
3) Configuración de estados y sus transiciones para cada tipo de ítem. Determinación de estado inicial y estados finales para un tipo de ítem.
4) Administración de los usuarios del sistema
5) Creación, actualización y eliminación de proyectos.
6) Administración de miembros de proyectos.
7) Administración de ítems particulares de un proyecto, con su correspondiente asociación a un tipo de ítem.
8) Asignación de miembros a un proyecto.
9) Cambio del estado de un ítem específico de un proyecto.
10) Asignación de un miembro del proyecto como responsable de un estado de un item específico de un proyecto.
11) Reasignación de un miembro del proyecto como responsable de un estado de un item específico de un proyecto.

Lista de QAs

Para el problema planteado se han detectado, como prioritarios, los siguientes requerimientos de atributos de calidad:

1) Modificabilidad: el sistema debe ser diseñado para adaptarse fácilmente y a bajo costo, a posibles cambios de requerimientos o mejoras.
2) Portabilidad: el sistema debe ser portable entre distintos motores de bases de datos relacionales, orientados a objetos u otra forma de almacenar la información.
3) Performance sin compromisos.

Casos de Uso

La complejidad o importancia para el sistema de ciertos requerimientos funcionales detectados, requiere de un análisis detallado de los mismos. Para lograrlo, se usarán casos de uso, de forma tal de especificar la interacción posible entre usuario y sistema.

Casos de Uso de Configuración de estados y sus transiciones para cada tipo de ítem

La configuración de estados y sus transiciones para un tipo de item, queda determinado por varias interacciones:

- Inicio del workflow de un tipo de ítem.
- Agregado de un nuevo estado para un workflow de un tipo de ítem
- Determinación del estado inicial.
- Determinación de los estados finales.

Caso de Uso: Inicio de workflow de un tipo de ítem

Actor: Configurador

Precondiciones: El tipo de ítem ya fue creado.

Curso Normal	Curso Alternativo
1. El configurador selecciona el tipo de item involucrado.	
2. El configurador inicia el workflow para el tipo de item, indicando el nombre del estado inicial.	

3. El sistema crea el estado, lo define como inicial y lo relaciona al tipo de item.	

Caso de Uso: Agregado de un nuevo estado para un workflow de un tipo de ítem

Actor: Configurador

Precondiciones: El tipo de ítem ya fue creado y su workflow iniciado (al menos, contiene un tipo de estado asociado)

Curso Normal	Curso Alternativo
1. El configurador selecciona el tipo de item involucrado	
2. El configurador le agrega un nuevo tipo de estado al workflow, indicando su nombre y si es final. Además selecciona, entre los tipos de estado del workflow, el que será el anterior al nuevo tipo de item	2. Si para el workflow, existe un solo tipo de estado, el sistema no le pedirá que lo seleccione.
3. El sistema crea el estado, lo agrega como estado siguiente al tipo de estado elegido y y lo relaciona al tipo de item	

Caso de Uso: Determinación del Estado Inicial

Actor: Configurador

Precondiciones: El tipo de ítem y su workflow ya han sido creados.

Curso Normal	Curso Alternativo
1. El configurador selecciona el tipo de item involucrado	
2. El configurador selecciona un tipo de estado como inicial.	2. El configurador selecciona un tipo de estado ya definido como inicial. El sistema indica el error.
3. El sistema controla si el estado elegido puede ser inicial, o sea que no tenga definidos estados anteriores a él.	
4. Si puede ser inicial, el sistema lo	4. Si no puede ser inicial, el sistema

define como inicial.	informa el error.

Caso de Uso: Determinación del Estado Final

Actor: Configurador

Precondiciones: El tipo de ítem y su workflow ya han sido creados.

Curso Normal	Curso Alternativo
1. El configurador selecciona el tipo de item involucrado	
2. El configurador selecciona un tipo de estado como final.	2. El configurador selecciona un tipo de estado ya definido como final. El sistema indica el error.
3. El sistema controla si el estado elegido puede ser final, o sea que no tenga definidos estados siguientes a él.	
4. Si puede ser final, el sistema lo define como final.	4. Si no puede ser final, el sistema informa el error

Casos de Uso de Seguimiento de un proyecto

El seguimiento de los ítems de un proyecto queda determinado por varias interacciones. La más compleja que requiere la especificación mediante un caso de uso es:

- Cambio del estado de un ítem específico de un proyecto

Casos de Uso: Cambio del estado de un ítem específico de un proyecto.

Actor: responsable del estado de un ítem ó responsable del proyecto

Precondiciones: el ítem está creado, está relacionado a un proyecto y asociado a algún tipo de ítem. El responsable del estado del ítem es un miembro del proyecto.

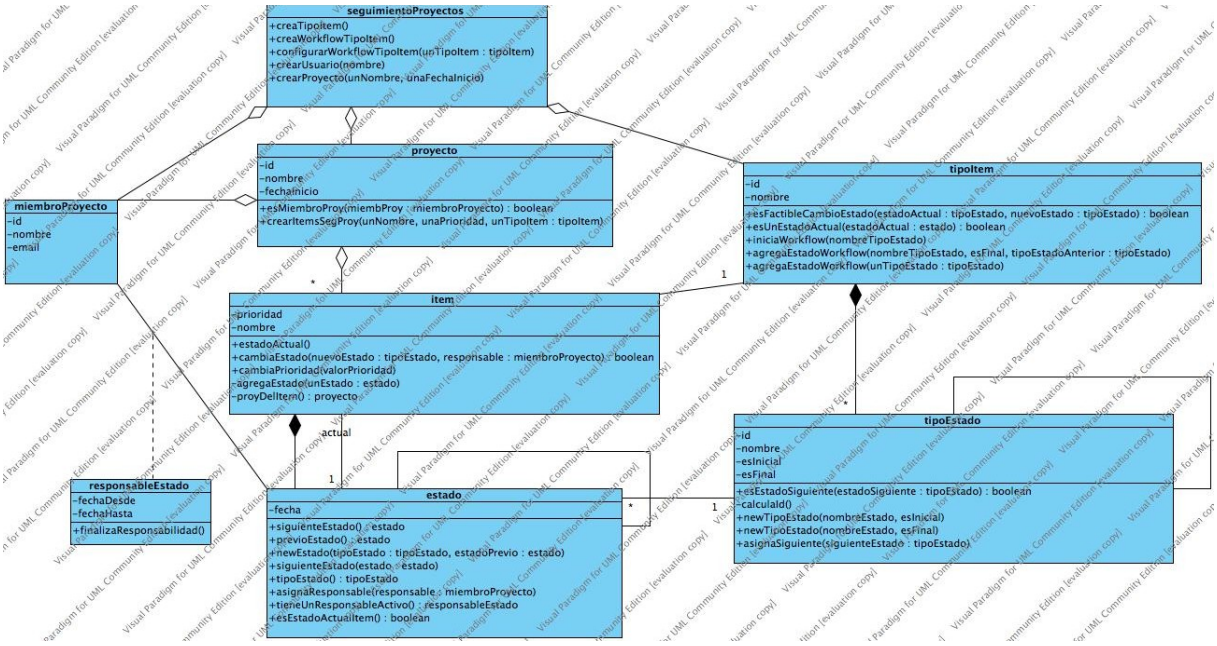
Curso Normal	Curso Alternativo
1. El responsable indica que va a cambiar el estado actual de un item	
2. El sistema controla que el actor sea responsable del ítem ó responsable del proyecto.	
3. Si es responsable del ítem o del proyecto. el sistema le ofrece los tipos de estado posibles (los tipos de estado posteriores al actual) , según la configuración de estados del tipo de item asociado al item elegido.	3. Si el actor no es responsable del ítem ni del proyecto, el sistema informa error.
4. El responsable elige el nuevo estado entre los posibles y le asigna un responsable entre los miembros del proyecto.	
5. El sistema controla que pueda darse el cambio de estado según el workflow del tipo de item.	
6. Si es factible el cambio, crea el nuevo estado, lo define como siguiente a su estado previo y lo agrega como estado del item.	6. Si no es factible, informa el error.
7. El sistema controla que el responsable del nuevo estado sea un miembro del proyecto.	
8. Si es miembro del proyecto, lo asigna como responsable del nuevo estado.	8. Si no es miembro del proyecto, informa el error y deshace la operación de creación del nuevo estado.

Diseño conceptual del sistema

El diseño conceptual – bajo el paradigma objetos – del problema planteado consta de un diagrama de clases y de los diagramas de secuencias de las funcionalidades más complejas o de importancia. Se utilizaron los diagramas de secuencia para profundizar y validar la elaboración del diseño conceptual, tanto en las clases y como en los métodos.

Diagrama de Clases

El diagrama de clases del diseño conceptual para el sistema es el siguiente:



Decisiones de diseño conceptual adoptadas

Para la elaboración del diseño conceptual se tomaron las siguientes decisiones:

a) Los servicios modelados contienen validaciones que deberían también ser realizadas por la interfaz de usuario. Como ejemplo de este aspecto, se puede citar la validación del usuario responsable asignado a un nuevo estado de un ítem. Este punto debe ser controlado y propuesto por el front-end y por una capa de seguridad que administre usuarios, menúes y permisos, pero de todas formas, el servicio de cambio de estado realiza este control nuevamente. El objetivo tras esta decisión, es independizar el modelo de interfaz de usuario de los servicios presentados y minimizar los defectos de una posible implementación.
b) La propuesta no modela el aspecto de persistencia de los datos porque el mismo, según requerimiento de la cátedra (reflejado en uno de los QAs presentados) debe ser lo más independiente posible del modelo que resuelve el problema planteado.
c) El modelo propuesto no incluye constructores, getters ni setters que no sean significativos para entender y diseñar los aspectos complejos o de peso del sistema, que es el objetivo de este trabajo.
d) Este modelo constituye un marco para la implementación por tratarse del diseño conceptual.

Responsabilidades de las clases

Para una mejor lectura y comprensión del modelo que se presenta, se efectúa una breve descripción de las responsabilidades de cada clase.

Una aclaración: en la descripción de cada una no se indica textualmente que se encarga de crear el/los objeto/s de su clase para evitar repetir este concepto en todos los casos, sino que se presenta una descripción del objetivo de cada objeto que crea.

- miembroProyecto:
  - Objetivo: modela el rol de miembro del proyecto.
  - Responsabilidades principales:
    - Identificar los datos de los usuarios.

- seguimientoProyectos:
  - Objetivo: modela las principales reglas de negocio del sistema.
  - Responsabilidades principales:
    - Administrar nuevos tipos de ítems.

        - Configurar el workflow de un tipo de ítem.
        - Administrar los miembros del proyecto.
        - Administrar los proyectos

- tipoItem
    - Objetivo: modela los tipos de ítems que podrán utilizarse para efectuar el seguimiento de ítems específicos de un proyecto.
    - Responsabilidades principales:
        - Administrar y configurar su posible workflow de estados.
        - Informar acerca del workflow de un tipo de ítem

- tipoEstado
    - Objetivo: modela los tipos de estado posibles para un tipo de ítem.
    - Responsabilidades principales:
        - Registrar e informar su/sus tipo de estado/estados siguiente/siguientes.
        - Registrar e informar su tipo de estado anterior.

- proyecto:
    - Objetivo: modela los proyectos del sistema
    - Responsabilidades principales:
        - Administrar los ítems específicos de un proyecto
        - Administrar los miembros de su proyecto.

- item:
    - Objetivo: modela los ítems específicos de cada proyecto
    - Responsabilidades principales:
        - Registrar sus cambios de estado.
        - Permitir el seguimiento de sus estados.
        - Registrar los responsables de cada estado.

- estado:
  - Objetivo: modela los estados de un ítem específico
  - Responsabilidades principales:
    - Registrar el estado
    - Registrar e informar los miembros del proyecto responsables por él.
    - Registrar e informar su/sus estado/estados siguiente/siguientes.
    - Registrar e informar su estado anterior.

- responsableEstado
  - Objetivo: modela la responsabilidad de un miembro del proyecto por un estado de un ítem de proyecto.
  - Responsabilidades principales:
    - Registrar e informar el rango de fechas en la cual un miembro del proyecto es responsable por el estado.

Diagramas de Secuencias

Para validar y completar el diseño conceptual, se elaboraron los diagramas de secuencias de las funcionalidades complejas o core del problema planteado.

Diagrama de secuencias del inicio de la creación del workflow para un tipo de ítem

Este diagrama de secuencias modela la creación del workflow para un tipo de ítem, representado mediante la creación de su tipo de estado inicial.

Diagrama de Secuencias del agregado de un tipo de estado en el workflow de un tipo de ítem

Este diagrama de secuencias modela el agregado de un tipo de estado como posterior a otro preexistente para el workflow de un tipo de ítem.

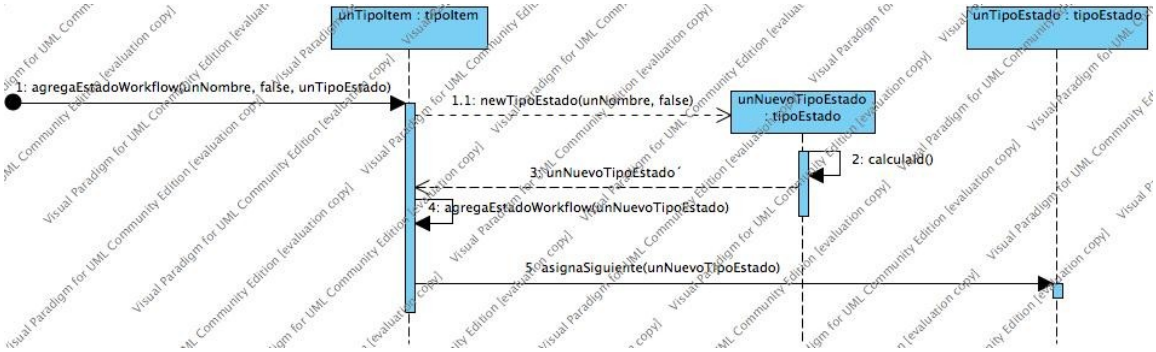
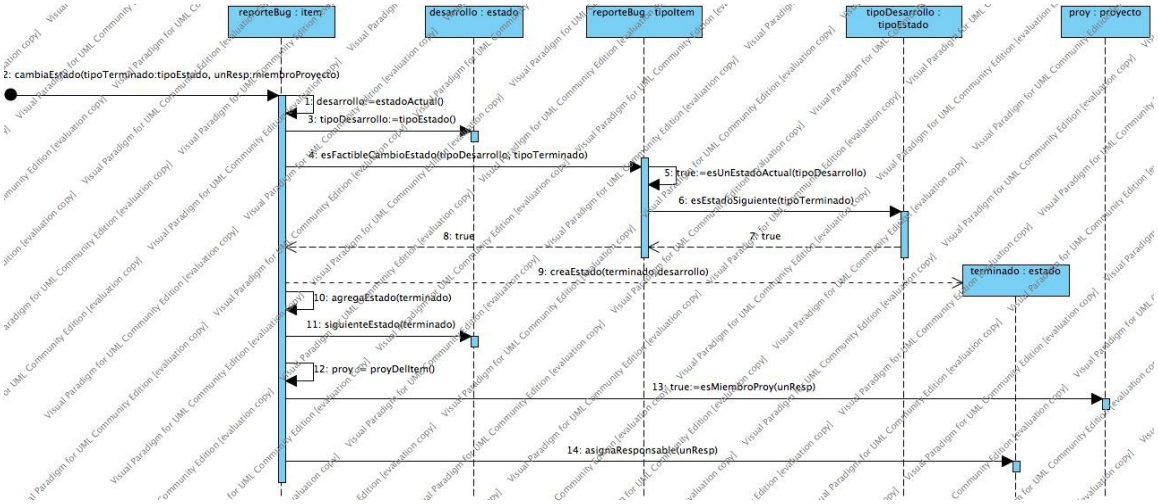


Diagrama de Secuencias del cambio de estado de un ítem específico de un proyecto

Este diagrama de secuencias modela el requerimiento de mayor complejidad, que es el cambio de estado de un ítem de un proyecto. Este diagrama modela el escenario donde un ítem “Reporte de Bug” pasa del estado “desarrollo” al estado “finalizado”. Para efectuarlo, verifica la factibilidad de dicho cambio de estado a través de la configuración del workflow perteneciente a la tipificación del ítem. Cada ítem específico de un proyecto está tipificado

mediante su asociación un tipo de ítem, el que, a su vez, tiene definido su workflow posible.



Construcción - Consideraciones iniciales

Introducción

La construcción del sistema se realizó utilizando un mapeador modelo de objetos – modelo relacional, bajo un arquitectura orientada a servicios, con el siguiente detalle:

- Java como lenguaje de programación
- Postgres como base de datos relacional
- JPA como estandard de persistencia para Java
- Hibernate como mapeador modelo de objetos – modelo relacional. Hibernate es la implementación de referencia del estándard JPA.
- Maven como herramienta para la construcción del proyecto y definición de dependencias de componentes externos.
- Spring Boot como herramienta para construir y configurar aplicaciones Spring en forma ágil y sencilla. Reduce notablemente la complejidad del desarrollo. PD: ¡Es magia pura! ;-)
- Spring Boot incluye un servidor web embebido, que se pone en marcha al lanzar la aplicación.
- Eclipse como entorno de programación Java
- DBeaver como cliente SQL para administrar la base de datos en el motor Postgres.

Prueba de Concepto

Se desarrolló una prueba de concepto (a partir de ahora, PoC) del back end de una aplicación que resuelve el problema elegido.

Como el objetivo de este trabajo práctico es generar un mapeo objeto – relacional transparente a la estrategia de persistencia y al motor de base de datos relacional elegido, la PoC se centra en funcionalidades de consulta, alta, actualización y baja de datos, con concurrencia.

Se desarrollaron varias consultas: de tipo de estados, de tipos de items, de proyectos creados, último estado de un item, items de un proyecto, estados

de un item, por ejemplo.

Respecto a altas, se desarrollaron alguna sencillas como alta de tipo de estado, de tipo de item, de proyectos y algunas más complejas como alta de un item de un proyecto ó alta de un estado para un item de un proyecto.

Relacionadas con las actualizaciones de datos, se pueden considerar una funcionalidad directa como la modificación del nombre de un proyecto, aunque otras funcionalidades traen aparejadas modificaciones en las listas de objetos asociados, como al dar de alta un item de un proyecto (actualiza su colección de items).

Relacionado con baja o eliminación de datos, se desarrolló la baja del último estado de un ítem, la cual aplica persistencia por alcance, ya que no se indica el borrado directo sobre la base de datos ni sobre el objeto, sino que se indica la eliminción del objeto estado de la lista de estados de un item de un proyecto.

En esta PoC, no se abordaron reglas de negocio que sean algoritmos complejos, por no estar dentro de los objetivos de este trabajo, resolver algoritmos. De todas formas, se implementaron reglas de negocio no complejas como controlar que un estado inicial de un ítem, sólo puede estar tipificado como tipo de estado inicial ó que cuando el último estado de un item es de tipo final, no se puedan agregar más estados.

Como cliente usé Posman, con carpetas y funcionalidades disponibles para testear los servicios implementados para la prueba de concepto del back end.

Quedaron fuera del alcance de la PoC los siguientes aspectos:

- capa de seguridad y acceso: usuarios y roles
- funcionalidades relacionadas con “miembroProyecto” del modelo conceptual
- asignación de “miembroProyecto” como responsables de estados de items de un proyecto
- generación del workflow de tipos de estado posibles para un tipo de item.
- algoritmo de control de cambio de estado de un item según el workflow o patrón de estados posibles asociado a su tipo de item.

Construcción - Diseño del esquema de Base de Datos

Elección del esquema de persistencia - razones

La elección del esquema de persistencia adoptado para el trabajo se basó en un mapeo del modelo de objetos con el modelo relacional. ¿Por qué elegí persistencia en una base de datos relacional cuando podría haber elegido una base de datos orientada a objetos, por ejemplo? Las causas principales fueron dos: la primera es porque en la industria aún es el modelo de base de datos más utilizado y la segunda, es debido a que en los proyectos en los que intervengo la persistencia se realiza en DBMS relacionales. Como expresé en la sección de *Objetivos del Trabajo Práctico*, uno de ellos es profundizar sobre técnicas y frameworks para mejorar la arquitectura y el código de los proyectos en los que trabajo.

Decisiones adoptadas para el esquema de persistencia

Bajo una arquitectura general del sistema orientada a servicios, la construcción del mapeo modelo objetos – modelo relacional se realizó utilizando el framework Hibernate y JPA, como estandard de persistencia de Java. Cabe destacar que Hibernate fue utilizado como framework y no como librería. El motor de base de datos relacional elegido fue Postgres.

¿Archivos externos ó annotions?

El framework Hibernate provee dos esquemas para la implementación del mapeo: mediante archivos externos hbm.xml ó mediante annotations. En la prueba de concepto (PoC) elegí usar annotations porque provee un mecanismo ágil de mapeo y como la PoC tiene un alcance acotado, su implementación en las clases del modelo no interfirió en su legibilidad, o sea, no bajó su entendibilidad. No se generó un exceso de annotations. Sin embargo para una aplicación real, en la industria, elegiría implementar el mapeo mediante archivos externos para bajar el acoplamiento con la capa del modelo y que las clases de esta capa queden mucho más legibles. Pero la razón más fuerte sería que, al bajar notablemente el acoplamiento con la capa del modelo, se facilita el cambiar la aplicación a alguna otra plataforma o paradigma de persistencia, reduciendo los costos de retrabajo; o sea, sin necesidad de recodificar una gran

cantidad de clases de la aplicación: el uso de archivos externos para el mapeo baja notablemente el impacto de futuros cambios en el esquema de persistencia.

Generación de claves primarias

Las claves primarias de las tablas mapeadas se especificaron usando la annotation @Id aplicada a los atributos id de las clases del modelo. También decidí que los valores de las claves se generen automáticamente, para lo cual usé la annotation @GeneratedValue.

Control de concurrencia

Para implementar control de concurrencia elegí un esquema de lockeo optimista, o sea, no realicé bloqueo de registros de la base de datos. Lockear en forma optimista afecta la performance, no es lo más aconsejable en aplicaciones web y hasta podría traer aparejado problemas de inanición. Pero, al usar un mecanismo optimista de lockeo pueden existir varias transacciones que acceden concurrentemente a un mismo recurso, elevando la posibilidad de tener conflictos con los cambios que se realizan.

La forma de manejar estos conflictos mediante JPA, es agregar un atributo denominado **version** para cada tabla sobre la cual necesito implementar lockeo optimista. En la capa modelo también se agrega, se nota con @Version y no debe ser manipulado por la aplicación. Su uso es exclusivo de JPA que, conceptualmente, utiliza el siguiente mecanismo: cuando se accede (lee) a un recurso, toma su versión. Al momento del commit, compara la versión que había leído contra la versión actual del registro en la base de datos. Si difieren, ejecuta un rollback y da aviso al usuario del conflicto. Si no difieren, realiza el commit e incrementa – automáticamente - el valor del atributo version.

Motor de Base de Datos elegido

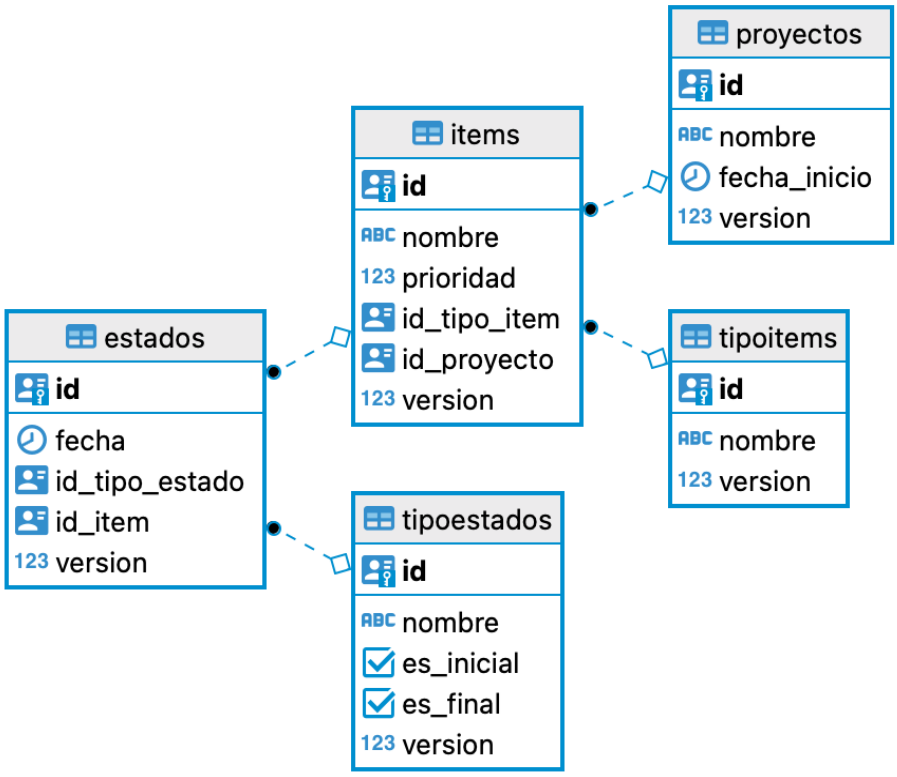
La prueba de concepto se desarrolló utilizando el motor relacional PostgreSQL versión 13.1.

Las tablas se crearon en una base de datos denominada vivianaortiz, en el esquema de nombre es tplp.

Construcción - Diseño del esquema de Base de Datos

Diagrama Entidad-Relación

Las tablas implementadas en el motor para la PoC son las que modelan tipos de items, tipos de estados, proyectos, items de proyectos y sus estados, y las relaciones entre estos conceptos, según el modelo de objetos. El siguiente diagrama de entidad-relación muestra las tablas y sus relaciones.



El mapeo entre atributos del modelo y atributos del modelo relacional, fue el habitual, mapeando uno a uno con el mismo naming. La resolución de las relaciones entre clases del modelo, por ser son todas agregaciones y de una cardinalidad 1:n, se mapeó al modelo relacional con relaciones 1:n entre las tablas mapeadas.

No tuve que persistir composiciones, por lo cual no fue necesario implementar triggers para resolver este tipo de asociación entre clases. Tampoco tuve necesidad de persistir jerarquías.

Cabe destacar que todos los atributos de las clases del modelo orientado a objetos fueron persistidos. Por lo tanto, no tuve necesidad de utilizar la annotation @Transient, que indica que el atributo de la clase de la capa modelo,

no es persistido.

Configuración de la conexión a la base de datos

En la construcción de la PoC, utilicé Maven para gestionar y construir el proyecto y como manejador de librerías de la aplicación. Maven utiliza un Proyect Object Model (archivo pom.xml), donde se especifican las dependencias de otros módulos/componentes externos. En el caso del motor de base de datos, registré la dependencia en este archivo de la siguiente forma:

```
<dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.2.2.jre7</version>
</dependency>
```

Respecto a la configuración de la conexión con la base de datos, la registré en el archivo application.yml del proyecto de la siguiente forma:

```
datasource:
    url: jdbc:postgresql://localhost:5432/vivianaortiz
    driverClassName: org.postgresql.Driver
    username: vivianaortiz
    password: vivianaortiz
```

En una aplicación de la industria, buscaría la forma de registrar la configuración de la conexión en un archivo externo, para aumentar la portabilidad de la aplicación.

Construcción - Arquitectura y Diseño de la aplicación

Decisiones adoptadas para la arquitectura

Elección de Spring Boot – Razones

La elección del framework Spring Boot para la construcción es claro: resulta una herramienta extremadamente útil, ágil y sencilla de utilizar para generar el esquema de trabajo y la arquitectura de la aplicación. En este caso, una aplicación web. Pone a nuestro alcance desde la estructura de trabajo, hasta el servidor web que escucha las peticiones, pasando por inyección de dependencias y manejo transparente de transacciones. Todo está incluído en el proyecto generado por este framework. Además, cuando se lo utiliza con Maven, genera todas las dependencias en el archivo pom.xml.

Spring Boot es el framework de Java más utilizado y es gratis. Simplifica enormemente el trabajo, ahorrando tiempo y costos de desarrollo, sin por ello sacrificar control sobre el código ni rendimiento.

Pero no todo es positivo: Spring Boot es pesado, usa mucha memoria. Si bien para esta PoC puedo suponer que el servidor está arriba siempre, en otras aplicaciones que no lo permitan (por costos, disponibilidad u otras razones) o requieran, por ejemplo, actualizaciones frecuentes, podría resultar complejo de administrar. De todas formas, Spring Boot también permite generar un arquitectura de microservicios, que daría una solución a este tipo de problemas. Otra solución a pesar ante el desarrollo de este tipo de sistemas, es utilizar el framework Quarkus, que permite reducir el tiempo de start up y el consumo de memoria y recursos de aplicaciones orientadas a microservicios.

Decisiones en las capas de la arquitectura

En el siguiente punto del informe, se aborda la arquitectura y diseño por capas. Relacionado con este punto, otra decisión tomada fue respecto a la capa de los DTOs (Data Transfer Object): decidí no utilizar un mapping (Map<String,String>) porque si bien otorga gran flexibilidad, se pierde legibilidad y, por lo tanto, calidad. Siempre prefiero un código legible y entendible; y, por lo tanto, mantenible.

En la capa de servicios, generé una clase ServiceBroker que inyecta instancias de las interfases de servicios, para que éstas puedan colaborar entre sí en la implementación de alguno de sus métodos.

Manejo de errores

Relacionado con el manejo de errores y mensajes al usuario por controles de reglas de negocio, decidí usar un esquema simple: la gestión la realicé mediante *ResponseStatusException*, capturé el error 409 (HttpStatus.Conflict) y a través de éste, envío los mensajes de excepciones. Además, configuré el manejo de errores en *application.yml*. Para una PoC, creo que es un manejo adecuado. Si tendría que implementar la aplicación completa en la industria, estimo que cambiaría de estrategia de manejo de unificado de errores mediante @ControllerAdvice y @RestControllerAdvice.

Arquitectura

La prueba de concepto (PoC) es una aplicación web, orientada a servicios, que cumple el patrón arquitectónico MVC. La capa de persistencia de datos se implementó en un motor de base de datos relacional. El mapeo objeto-relacional es resuelto mediante Hibernate, como implementación de referencia del estándard JPA.

No fue necesario desplegar la aplicación en un servidor web ya que el framework Spring Boot provee uno, integrado al .jar de la aplicación.

Diseño

El código se organizó por capas implementadas a través de paquetes; cada una con responsabilidades definidas, claras y acotadas, siguiendo el principio de single responsibility. Las capas, por orden alfabético, son las siguientes:

Capa api:

La responsabilidad de esta capa es brindar respuesta a las peticiones e invocar a la capa de servicios para que resuelva la petición. Como se trata de una aplicación web, define su API REST.

Las principales annotations usadas son:

- @RestControler: crea servicios web RESTFul, indica que desde la web puedo invocar los servicios de la clase.
- @Inyect: inyecta una instancia, en este caso, de cada interfaz de la capa de servicios. No existe un new, no se necesita saber de qué tipo es el colaborador. Es un ejemplo de inyección de dependencias.
- @RequestMapping: especifica el path o ruta desde la cual se escuchará el servicio que los clientes invoquen, y qué método le responde.

Esta capa tiene una clase denominada PocController, en la cual – para mejor legibilidad – organicé los servicios por tema, agrupándolos. Los métodos de esta capa sólo invocan a servicios, a excepción de una única operación que realizan: generar los UUID a partir de un string recibido en la petición, cuando es necesario.

Capa dto:

La responsabilidad de esta capa es crear objetos DTO, que cumplen el patrón del mismo nombre (Data Transfer Object). Estos objetos DTO se usan para transferir datos entre capas. Particulamente, en este proyecto se usan para mostrar, fuera de las transacciones, los objetos obtenidos o modicados por ésta.

Los objetos de esta capa no tienen comportamiento, sólo poseen los atributos que necesito para transferir datos entre capas ó para mostrarlos como respuesta a servicios invocados. Por lo tanto como respuesta a las peticiones transaccionales no se retornan objetos del modelo sino representaciones de estos objetos.

Las principal annotation usada es:

- @Component: que indica que la clase en un componente o Bean de Spring

Esta capa tiene una clase de DTOFactory que se utiliza para crear los distintos objetos DTO en forma centralizada. También contiene las clases que definen los diferentes objetos DTO necesarios para resolver los casos de uso implementados en la PoC.

Capa model:

La responsabilidad de esta capa es la de representar los objetos del modelo y su comportamiento. La idea es que esta capa quede lo más independiente de la estrategia de persistencia elegida.

En mi caso, utilicé annnotations en las clases de esta capa, que da indicios del esquema de persistencia elegido. Como indico en la sección "*Construcción – Diseño del esquema de Base de Datos*" de este informe, si tuviese que desarrollar esta aplicación para la industria, preferiría elegir la opción de implementar el esquema de persistencia mediante archivos externos hbm.xml para bajar el acoplamiento entre el modelo y la estrategia de persistencia. Para esa PoC, usé annotations por ser una forma ágil y que cumple los objetivos de este trabajo práctico sin perder legibilidad en las clases involucradas.

Las principales annotations usadas son:

- @Entity: indica que se trata de una entidad JPA
- @Table: indica el nombre de esquema y tabla que persiste el objeto de la clase.
- @Id: especifica que es la clave primaria de la entidad
- @GeneratedValue: autogenera los valores de los ids
- @Version: utilizada por JPA para administrar concurrencia.
- @ManyToOne, @OneToMany: especifica el tipo de relación con otras entidades.
- @JsonIgnore, @JsonManagedReference: utilizada para que Json lo omita en el proceso de serialización.

Esta capa contiene las clases del modelo de objetos implementado en la PoC: tipoItem, tipoEstado, item, estado y proyecto.

Capa repository:

La responsabilidad de esta capa es ser un colaborador que permite recuperar objetos del modelo persistidos en la base de datos. Implementa el patrón Repository.

Si bien los repositorios pueden gestionar todos los accesos a los datos, ya que disponen de los métodos básicos de CRUD, en esta implementación se restringió su responsabilidad a la recuperación (R en CRUD). La creación, modificación y borrado de los objetos de la base de datos se generan por modelo y por persistencia por alcance.

Las principal annotation usada es:

- @Repository: indica que es una especialización de @Component, de tipo repositorio que encapsula el acceso a los datos persistidos.

Esta capa contiene las interfaces que definen el acceso a los datos persistidos. Extienden a JpaRepository. Definí una clase interfaz por cada tipo de objeto persistido. No tuve necesidad de definir mediante @Query ninguna consulta sql. Tampoco tuve la necesidad de implementar ninguna extensión customizada de estas clases, ya que no necesité programar algún método

específico para resolver consultas complejas o llamar a un stored procedure, por ejemplo.

Capa service:

La responsabilidad de esta capa es implementar los servicios o reglas de negocio de alto nivel de la aplicación. Contiene una interfaz con su implementación por cada objeto de relevancia del modelo. Los métodos de servicio, básicamente, utilizan la capa repository, la capa modelo y la capa dto. La primera, para recuperar los datos necesarios para realizar su tarea; la segunda para resolver reglas de modelo y la tercera, para retornar, cuando es necesario un dto representante del objeto del modelo. Los servicios son transaccionales, para lo cual usé @Transactional.

Las principales annotations usadas son:

- @Service: indica que es una especialización de @Component, de tipo servicio que encapsula las reglas de negocio de alto nivel.
- @Transactional: decora los servicios con una transacción: la abre, ejecuta el servicio y luego cierra la transacción. Es una implementación del patrón Decorator.
- @Inject: inyecta una instancia, en este caso, de las interfaces de la capa repository y de la clase DTOFactory.

Esta capa contiene las interfaces que definen los servicios de la aplicación. Definí una clase interface para cada tema de relevancia del dominio. Fue necesario crear una clase ServiceBroker que inyecta instancias de las interfases de los distintos servicios, para que puedan colaborar en la implementación de algunos servicios. También fue necesario crear una clase abstracta en la capa de implentación de las interfases de servicio, denominada AbstractServiceImpl. Las clases que implementan los servicios, heredan de esta última. Tanto en ServiceBroker como en AbstractServiceImpl, decidí usar @Getter de Lombok. Para usarlo, agregué la dependencia en pom.xml.

Despliegue

Spring Boot permite compilar las aplicaciones web como un archivo .jar y ejecutarlas como una aplicación Java habitual, ya que integra en servidor de

aplicaciones propio en el .jar y lo levanta cuando arranca la aplicación.

De esta forma se puede desplegar la aplicación de una forma muy sencilla. Para esta PoC resulta una solución adecuada.

Por supuesto, además tenemos que tener desplegado el DBMS que se utilizará para persistir la información.

Código

El código completo de la aplicación se presenta en un repositorio Git y en un archivo POC.zip en una carpeta compartida de Google Drive.

Repositorio Git: github.com/vivortiz/PoC

Carpeta Google Drive: https://drive.google.com/drive/folders/1jQwZn7mxruQF0Y6N_HXBmOPcgD3T4aDC?usp=sharing

Conceptos teóricos aplicados en la construcción

Los principales conceptos teóricos aplicados en la construcción de la prueba de concepto fueron los siguientes:

- Principios SOLID: para mejorar la calidad en cuanto a portabilidad, legibilidad y diseño para el cambio
  - Single Responsability: cada clase diseñada y programada tiene una responsabilidad definida y clara. Si bien no tienen un solo motivo para cambiar, la responsabilidad de cada clase a nivel general es única y claramente definida.
  - Open Close: el diseño de este trabajo está abierto para la extensión y busca estar cerrado para la modificación. La idea es no modificar, por ejemplo, la firma de los servicios para los clientes que hagan uso de ellos.
  - Liskov: este principio se aplicó, por ejemplo, para el DBMS utilizado; en este trabajo podría cambiarse el motor de persistencia sin cambiar el código. Otro ejemplo de aplicación de este principio es que siempre que pude usé tipos de la superclase, por ejemplo usé Collection habitualmente. Generalizar es siempre un buen criterio para extensión

y modificación de aplicaciones aunque puede entrar en tensión con algunos requerimientos de calidad (performance, por ejemplo), por lo que debe balancearse y determinar hasta qué profundidad se aplica. Diría, en términos de ingeniería de software, aplicarlo bajo el principio de “Good Enough”, siempre que existan tensiones.

- Segregación de interfaces: cada interface que diseñé tiene una única responsabilidad ya que es preferible tener muchas interfaces pequeñas que contengan pocos métodos que tener pocas con muchos métodos. En este último caso, un cambio en la interface puede producir muchos dolores de cabeza.
- Inversión de control: se trató de alcanzar el objetivo que una clase interactúe con otras sin necesidad de conocerlas directamente, o sea tratando de bajar el acoplamiento entre las clases. Un ejemplo que utilicé es inyección de dependencias. Al inyectar una instancia, por ejemplo, en cada interfaz de la capa de servicios, no se necesita saber de qué tipo es el colaborador.

- Patrón DTO: lo utilicé en la capa dto del diseño, para transferir información entre capas, específicamente para la capa externa. Lo usé intensamente para mostrar datos fuera de una transaccion con el objeto de minimizar su duración y tratar de evitar – en lo posible – conflictos por caídas de servicio o concurrencia.

- Patrón Repositorio: lo utilicé en la capa repository del diseño, para encapsular la lógica de acceso (lo restringí a la recuperación) a los datos. No necesitamos saber si el objeto que me devuelve está en memoria o en la base de datos o conocer el metodo de persistencia. De esto se encarga el patrón, mediante diferentes mecanismos.

- Patron Decorator: agrega comportamiento dinámicamente a los objetos. En este trabajo se aplica, por ejemplo, al manejo de transacciones. El decorador abre la transaccion, llama al método de la capa de servicios y luego cierra la transacción. Por suerte ¡no tuve q escribir el decorador!, ya que lo provee Spring.

- ¿Librería o Framework?: trato de no invocar a Hibernate como librería, lo uso como framework. Otro ejemplo es el uso de Spring para administrar transacciones, no invoco servicios para abrir o cerrar sesiones, le dejo la responsabilidad al framework el saber cúando y cómo manejarlas.
- Persistencia por alcance: la utilicé en la PoC ya que el alta de objetos al vincularlo con otro existente se persiste o guarda por persistencia por alcance. La actualizacion de un objeto se hace obteniendo el objeto y enviandole los setters que corresponden y el borrado se realiza recuperando los objetos que lo conocen y eliminándolo de las referencias.

En todos los casos se aplica persistencia por alcance.

Configuración de la aplicación y pruebas

Para instalar y configurar la aplicación y la base de datos, se elaboró un documento denominado *BD-TPVOrtizInformeFinal Configuración POC.* El mismo contiene una breve guía con los pasos necesarios para el fin indicado.

Según fue expresado en este informe, se utilizó la herramienta Postman para testear la aplicación POC. Se elaboró un documento detallado de los servicios implementados y cómo ejecutarlos para probarlos mediante Postman, denominado *BD-TPVOrtizInformeFinal Instructivo de uso POC.*

Ambos documentos forman parte de este informe.

Conclusiones

El desarrollo del trabajo práctico resultó interesante y desafiante desde el punto de vista tecnológico. Si bien en la industria se usa Hibernate, la transparencia en el mapeo objeto-relacional dista – habitualmente – mucho de los aspectos tratados durante la materia.

Respecto a los objetivos planteados para este trabajo, creo que logré alcanzar el objetivo de elaborar una prueba de concepto de una aplicación programada en el paradigma objetos con persistencia de su información implementada en un motor de base de datos relacional, mediante un mapeo objeto-racional transparente respecto al motor elegido. El código resulta limpio y con una independiencia muy satisfactoria – en mi opinión – de la estrategia de persistencia elegido. No embebí código SQL en la aplicación, no abrí ni cerré sesiones, no administré concurrencia, no instalé ni configuré un servidor donde corre la PoC. El uso del framework Spring con la instrucción dada en la materia, me resultó muy interesante, útil, diría que casi “mágico”. La arquitectura y el diseño realizados resultan en una aplicación – en realidad prueba de concepto – entendible, con división clara de responsabildades, extendible, preparada para el cambio.

Respecto a los objetivos personales planteados, creo que fueron logrados. Espero tener la oportunidad de aplicar los conceptos y tecnologías aprendidas durante la materia y el desarrollo de este trabajo, en la vida profesional.