Trabajo Práctico Anual - Grupo 24

Diseño de Sistemas de Información

Índice

Justificación Entrega N°1

Para esta primera entrega realizamos un diseño que incluye las clases Usuario, SolicitudEliminacion, Hecho, Coleccion y FileReader.

En la clase Usuario implementamos las funcionalidades necesarias tanto para administradores como para contribuyentes, quienes a su vez pueden ser visualizadores o editores. Estas funcionalidades varían de acuerdo a los permisos (roles) asignados a cada usuario, que se detallará en próximas entregas.
Entre las responsabilidades principales de esta clase se encuentran:

- solicitarEliminacion, que permite a un contribuyente generar una nueva solicitud para eliminar un hecho.
- agregarSolicitud, que agrega dicha solicitud a la lista llamada *Spam* si cumple con el mínimo de caracteres requeridos en el motivo. En caso contrario, se lanza una excepción y no se guarda.
- crearColeccion, que instancia una nueva colección.
- asignarEtiqueta, que permite agregar una etiqueta a un hecho llamando al método correspondiente de la clase Hecho.
- evaluarSolicitud, usada por administradores para revisar solicitudes en la lista *Spam*. Este método incluye un bloque try-catch para capturar posibles excepciones si la solicitud no se encuentra, y anticipa futuras extensiones en la lógica de evaluación.

La clase Usuario también cuenta con atributos como nombre, apellido, edad, una lista de solicitudes de eliminación pendientes (*Spam*), y otra lista de solicitudes ya respondidas, lo que permite llevar un registro de las acciones realizadas por cada usuario.

La clase SolicitudEliminacion representa el pedido de eliminar un hecho de una colección. Incluye los métodos validarSolicitud, para verificar que el motivo cumpla con el mínimo requerido, y actualizarFechaModificacionSolicitud, que actualiza la fecha de última modificación.
Entre sus atributos se encuentran:

- un número identificador único para cada solicitud.
- el hecho a eliminar.
- el motivo de la solicitud.
- el estado actual (PENDIENTE, ACEPTADO o RECHAZADO), definidos mediante un enum, ya que son valores fijos sin comportamiento asociado.
- la fecha de creación y de última modificación (por defecto, el día actual).
- el mínimo de caracteres para el motivo, definido como un atributo estático para poder modificarse fácilmente en todas las instancias si fuera necesario.
- y un contador estático para asignar el número de solicitud de forma incremental cada vez que se crea una nueva.

La clase Coleccion tiene como objetivo agrupar hechos que comparten ciertas características. Para esto cuenta con una lista de criterios de pertenencia, representada como filtros.
Estos filtros se modelaron utilizando el patrón Strategy, mediante una interfaz llamada Filtro, que define el método filtrar. Las clases que implementan esta interfaz son FiltroPorCategoria, FiltroPorFecha, FiltroPorTitulo y FiltroPorUbicacion, cada una evaluando un atributo distinto del hecho y devolviendo un valor booleano según si coincide o no con el criterio definido.
Dentro de la colección, el método correspondiente evalúa todos los filtros sobre los hechos y retorna solo aquellos que los cumplen todos. Esto permite no solo definir la pertenencia de un hecho a una colección, sino también realizar búsquedas dentro de la misma.
Además de los filtros, la colección tiene atributos como título, descripción, lista de hechos y una fuente, que puede ser:

- PROXY (datos cargados por servicios externos)
- ESTÁTICA (cargados por archivos .csv)
- DINÁMICA (cargados por usuarios)

También definidos mediante un enum.

Entre los métodos destacados de la clase se encuentran:

- importarCSV, que lee un archivo .csv y convierte cada línea en un hecho.
- agregarHechos, que carga nuevos hechos a la colección usando el método agregarOReemplazar, el cual detecta duplicados por título y reemplaza el hecho existente o lo agrega como uno nuevo, según corresponda. Esto se basa en el método buscarIndicePorTitulo.
- recorrer, que permite aplicar distintos filtros para buscar hechos específicos dentro de la colección.

La clase Hecho representa un evento con todos sus datos relevantes. Entre sus atributos se encuentran:

- título y descripción.
- categoría.
- latitud y longitud.
- fecha en que ocurrió el hecho.
- fecha en que fue cargado (por defecto, el día actual).
- origen del hecho.
- etiquetas asociadas.

Como funcionalidades, incluye el método agregarEtiqueta, que permite clasificar un hecho, y mostrarse, que imprime por pantalla su título y descripción.

Por último, la clase FileReader se encarga de leer archivos .csv para cargar hechos al sistema. Su función principal es transformar cada línea del archivo en un objeto de tipo Hecho, que luego puede usarse dentro de las colecciones.

Tiene como atributo principal el path, que es la ruta del archivo a leer. A partir de ahí, con el método leerHechosDesdeCSV, abre el archivo, salta la primera línea (que son los títulos de las columnas) y empieza a procesar cada línea.

Para que esto funcione, hay un método llamado parsearLineaCSV, el cuál separa los distintos campos de la línea. Está pensado para manejar casos donde un campo puede tener comas y venir entre comillas, entonces lo recorre carácter por carácter y arma los campos de forma correcta. Una vez que se tienen los campos bien separados, se extraen los datos importantes como el título, la descripción, la categoría, latitud, longitud, la fecha en que ocurrió el hecho, y si hay un campo extra, también se toma. Con todo eso se arma un nuevo hecho, al que se le indica que su origen es CSV porque viene de un archivo. Si hay algún error en la lectura del archivo, se muestra un mensaje avisando que no se pudo leer.

CORRECCIONES ENTREGA 1

- El origen de los hechos no esta bueno que sea un enum porque no es extensible y es poco mantenible.

- Les hago una consulta para que piensen, relativa a la categoría de los hechos. Que sucederia en caso de que alguien escriba "Categoria A" y otro escriba "categoria a". Seria sencillo filtrar por esta categoría? Ven alguna forma mejor de modelarlo?

- La ubicación me parece que estaría buena plantearla como una clase aparte y sumarle un par de datos mas, mas que nada para simplificar a la hora de filtrar. Que les parece? Por que decidieron incluirlo dentro del hecho?

- Teniendo en cuenta que a futuro van a aparecer nuevas fuentes, les parece que lo mas adecuado es modelarlo como un enum?

- De los estados de la solicitud no les parece que estaria bueno guardar algún que otro dato mas?

- Me gustaría que a la clase usuario le pongan un nombre mas descriptivo, relacionado con el modelo que están desarrollando.

- Ademas, revisen los métodos de los usuarios, recuerden separar aquello que hace un usuario fisico de aquello que realiza la clase usuario. Esto es lo único que me parece importante que corrijan y entiendan muy bien el porque, el resto son correcciones pequeñas.

Justificación Entrega N°2 - Grupo 24

Realizamos una arquitectura basada en capas: controladores, servicios, repositorios y modelo de dominio. Esta separación nos permitió:

- Desacoplar la lógica de negocio de la exposición a través de APIs REST.
- Favorecer la reutilización del dominio y el testeo unitario.
- Facilitar la integración futura con nuevas interfaces o fuentes de datos (por ejemplo, nuevas APIs proxy).

Colecciones y Hechos

El objeto Hecho tiene un atributo Origen, que indica cómo fue creado (por CSV, por un usuario o desde un servicio externo).
Esta decisión evita acoplar la colección con una fuente específica, permitiendo que las colecciones sean más dinámicas y extensibles.

Origen y Fuente

Origen y Fuente se modelaron como interfaces con implementaciones concretas (CSV, Usuario, etc.).

Se definió Origen como una interfaz con tres implementaciones:

- ➔ CSV (fuente estática)
- ➔ Usuario (fuente dinámica)
- ➔ Servicio_Externo (fuente proxy)

El concepto de Fuente representa el canal por el cual se incorporaron los hechos al sistema:

- ➔ Estática (por archivos CSV)
- ➔ Dinámica (por usuarios)
- ➔ Proxy (por servicios externos)

Las fuentes proxy, en particular, se modelaron como servicios que consumen APIs externas, aplicando el patrón Adapter para encapsular tanto el formato como el consumo de esas APIs. Esto nos permitió incorporar hechos sin acoplar el sistema a la implementación concreta de cada fuente externa.

Solicitudes de Eliminación

La SolicitudEliminacion encapsula el motivo, el hecho a eliminar y su estado, el cual se modela usando una clase polimórfica (por ejemplo: Pendiente, Aceptada, Rechazada). Esta decisión sigue el patrón **State**, lo que permite extender fácilmente el comportamiento según el estado, sin depender de condicionales distribuidos en el código.
La detección automática de spam se implementó como una validación inicial del estado, manteniendo el control de flujo encapsulado dentro del servicio de creación.
Se utilizó un servicio DetectorDeSpam, lo que nos permite mejorarlo en el futuro sin alterar el resto del sistema. Esto favorece el bajo acoplamiento y una evolución progresiva del sistema.

Servicio de Agregación

Encapsula la lógica periódica de actualización de colecciones, permitiendo unificar hechos de múltiples fuentes (estáticas, dinámicas, externas) en una misma colección. Se implementó como un componente @Scheduled de Spring Boot, que se ejecuta cada hora. Esta tarea consulta únicamente las fuentes no MetaMapa (que se consultan en tiempo real) y agrega los hechos nuevos. Esta decisión busca un equilibrio entre consistencia y eficiencia, evitando recargar continuamente fuentes estáticas.
Además, ayuda a mantener la separación de responsabilidades: las fuentes se encargan solo de entregar hechos, mientras que el servicio decide cómo y cuándo agregarlos. También mejora la reutilización del código, evitando repetir la lógica de filtrado en distintos puntos del sistema.

Roles de Usuario

En el diseño del sistema definimos tres roles principales: **Contribuyente** (registrado o anónimo), **Visualizador** y **Administrador**, según los distintos tipos de interacción que permite la plataforma.
Esta separación facilita aplicar controles de acceso claros: los contribuyentes pueden subir hechos (y editarlos por un tiempo si están registrados), los visualizadores solo tienen permisos para pedir solicitudes de eliminación, y se contempla el rol de administrador, los cuales tienen permisos para realizar todo tipo de cambios en el sistema y además, son quienes aceptan o rechazan las solicitudes de eliminación de los hechos y los únicos responsables de la creación de colecciones. Todas las validaciones de acciones de cada rol fueron declaradas junto a los permisos, explicados debajo.

Validación y Permisos

En lugar de verificar cada acción directamente por rol, se diseñó una estructura flexible de **permisos**. Esto permite asociar acciones habilitadas directamente al usuario, como una lista de Permiso (como ACEPTAR_SOLICITUD_ELIMINACION, RECHAZAR_COLECCION, entre otros). Esta lógica sigue el **Principio de Responsabilidad Única**, ya que centraliza el control de acceso dentro de la lógica de negocio y no en los controladores.

El uso de un modelo basado en permisos hace que el sistema sea más flexible, escalable y fácil de extender a medida que evoluciona.

Servicios y Repositorios por Tipo de Fuente

Se implementó un servicio específico para cada tipo de fuente:

- EstaticaService
- DinamicaService

- ProxyService

Cada uno se encarga de:

- Obtener hechos desde su fuente correspondiente.
- Registrar las colecciones que contienen hechos originados desde esa fuente.

Para lograrlo, cada servicio cuenta con su propio repositorio (por ejemplo, estaticaRepository) que almacena los handleID de las colecciones asociadas.
Este diseño evita que las colecciones se acoplen directamente a las fuentes, pero mantiene la trazabilidad: sabemos qué colecciones dependen de qué fuentes.
También favorece la **separación de responsabilidades** (cada servicio maneja su propia lógica), facilita el **escalado futuro** del sistema si se agregan nuevos tipos de fuentes, y simplifica la lógica del ServicioAgregacion, ya que puede consultar esos registros para saber qué colecciones actualizar.

Justificación del tipo de cambio

Todo lo trabajado en esta entrega se considera principalmente un **refactor**, ya que se respetó la estructura general del sistema previamente desarrollado, pero se aplicaron mejoras en la organización del código, se incorporaron buenas prácticas de diseño orientado a objetos, y se agregaron nuevas funcionalidades, como la moderación de solicitudes de eliminación con validación de permisos.
Además, se ampliaron servicios existentes sin romper la arquitectura original ni rehacer el sistema desde cero, lo que marca la diferencia frente a un proceso de reingeniería.