Justificaciones de Diseño

Fuente Dinámica

1. Diseño de DTOs

- Uso de `idCategoria`, `idEtiqueta`, `idContribuyente`, etc. en los DTOs en lugar de objetos completos
- Aplicamos el principio de encapsulamiento y el patrón DTO. Los DTO contienen solo los datos esenciales para la comunicación entre capas o servicios.
- Si incluimos objetos de dominio generariamos un acoplamiento innecesario entre la representación externa y el modelo interno, y podría exponer lógica o estructuras que deben permanecer ocultas.

2. Diseño de entidades

- El ID no se incluye en el constructor de `Hecho`, es asignado por el repositorio, que es el responsable de la persistencia. En esto se involucran los principios:
    - Responsabilidad única (SRP): El repositorio es el único responsable de la generación de IDs.
    - Encapsulamiento: La entidad no debe saber cómo se asignan sus IDs.

3. Simulación de persistencia

- Implementamos repositorios en memoria con generación de ID autoincremental
- Dado que no usamos bases de datos reales, simulamos la persistencia manteniendo las entidades en listas en memoria.
- El método `generarNuevoId()` encuentra el máximo ID actual y suma 1.

4. Carga de hechos sin usuario registrado

- Permitimos poder subir hechos sin usuario registrado pero sin posibilidad de edición
- En los DTO, el `idContribuyente` puede ser `null`. El campo `idContribuyente` se usa para determinar si se puede editar el hecho.

5. Datos de prueba

- Dado que aún no desarrollamos completamente el manejo de etiquetas desde el frontend o DTOs, agregamos temporalmente etiquetas “de prueba” para que el código no rompa.

6. Patrones y principios aplicados

- **SRP (Single Responsibility Principle):** Cada clase o componente cumple una función bien definida. Ej: los servicios no acceden directamente al repositorio de otras entidades.
- **DTO Pattern:** Aplicado para evitar exponer entidades.
- **Repository Pattern:** Usado para aislar la lógica de almacenamiento y acceso a los datos.
- **Encapsulamiento y bajo acoplamiento:** Todas las decisiones de diseño intentan minimizar las dependencias entre módulos: los servicios interactúan a través de interfaces claras y específicas, se evita exponer las entidades directamente mediante el uso de DTOs y se encapsula la lógica de persistencia dentro de los repositorios. Estas prácticas favorecen el bajo acoplamiento y la extensibilidad del sistema.

Agregador

1. Flujo de solicitudes de eliminación

1. Una persona crea una solicitudDeEliminacion desde fuenteDinamica, e inmediatamente se llama al agregador al método filtrarSpam.
    a. Si es Spam se rechaza automáticamente, se cambia el estado de solicitud y se setea la fecha de atención.
    b. Sino esSpam la deja en pendiente.

    Las rechace o las deje en pendiente, de todas maneras las guarda en un repositorio de solicitudes.

2. Luego, un administrador va a acceder al repositorio de solicitudes (solo a las pendientes) y va a aceptarlas o rechazarlas.
    a. Si acepta, cambia el estado a aceptada, setea fecha de atención y llama al método ocultarHecho del agregador.
        i. En ese método se actualiza el repositorio de colecciones, ya que se le cambiara un atributo a un hecho de una colección (hecho.setFueEliminado(true)).
    b. Si la rechaza, cambia el estado a rechazada, setea fecha de atención.

    En ambos casos se vuelve a guardar en el repositorio.

2. Colección

- Decidimos que las colecciones se creen desde el service de colección dentro del agregador (método crearColeccion), y no que se creen en la propia entidad de domino Colección.
  ¿Por qué? Porque creemos que el llamado al método coleccion.getFuentes() y

luego el llamado a getHechos() que se le hace a cada fuente (se le piden sus hechos), es una conexión del agregador con las fuentes mediante una API, por lo que sería correcto que esté dentro de la estructura de capas, y no en la de dominio.

- Decidimos implementar un handle que se genera automáticamente a partir del título en ColeccionRepository.save(), eliminando espacios y garantizando unicidad con un sufijo numérico.

3. Preguntas disparadoras

- **Incorporar un string alfanumérico, sin espacios, único por Colección. En ColeccionRepository.save(), se genera un handle:**
  - Se eliminan los espacios del título.
  - Si ya existe ese handle, se le agrega un sufijo numérico para garantizar unicidad.

- **¿Cómo se actualizan automáticamente los hechos de las colecciones?**
  Nuestro método actualizarColecciones() en ColeccionService itera todas las colecciones y para cada una llama a filtrarHechos(), en donde se le piden los hechos a las fuentes y así se rearma la colección.
  Este método se ejecuta periódicamente mediante @Scheduled(fixedRate = 3600000) en ColeccionesScheduler.

- **No deben actualizarse hechos de fuentes Proxy de tipo MetaMapa (son en tiempo real).**

- **¿Cómo rechazar automáticamente solicitudes de eliminación si son Spam?**
  En SolicitudService, en el método filtrarSpams(Solicitud solicitud) si el hecho es null o el detector dice que es spam, se marca como rechazada. Luego se guarda en el repositorio.

- **¿Analizamos el Spam al crear la solicitud o después?**
  Nosotros decidimos que en el momento de creación de la solicitud, inmediatamente se analiza si es spam. Esto se hace antes de que intervenga el administrador.

- **¿Cómo avanzar sin tener el detector de Spam implementado?**
  Con una clase abstracta. SolicitudService es abstract e implementa la interfaz IDetectorDeSpam. Esto permite separar responsabilidades, que nuestro módulo pueda seguir funcionando sin estar acoplado directamente a su implementación (que se delega al detector de spam) y permite mockear el detectorDeSpam para tests.

Fuente Proxy

- Utilizamos un patrón Adapter para unificar el acceso a múltiples servicios externos (cada uno con su propia estructura y lógica de obtención de datos). Con esto logramos que el agregador no necesite saber si una fuente está trayendo hechos de la API de la cátedra, de la de Metamapa o de cualquier otra. También, si cambia la estructura de una API, solo se modifica el adapter.

- **MOCKEO API METAMAPA:**
    1. Creamos un servidor mockeado y ponemos un ejemplo de respuesta.
    2. Levantamos nuestro sistema
    3. Mandamos request de /hechos y devuelve el ejemplo que pusimos antes.