Máster Universitario en Tecnologías Web,
Computación en la Nube y
Aplicaciones Móviles

Asignatura:

Persistencia relacional y no relacional de datos

30 de mayo de 2023

Trabajo final

Andoni Salcedo Navarro

1. SQL

1.1. Modelo Conceptual

A continuación se muestra el esquema entidad relación de los datos del trabajo final (ver figura 1)

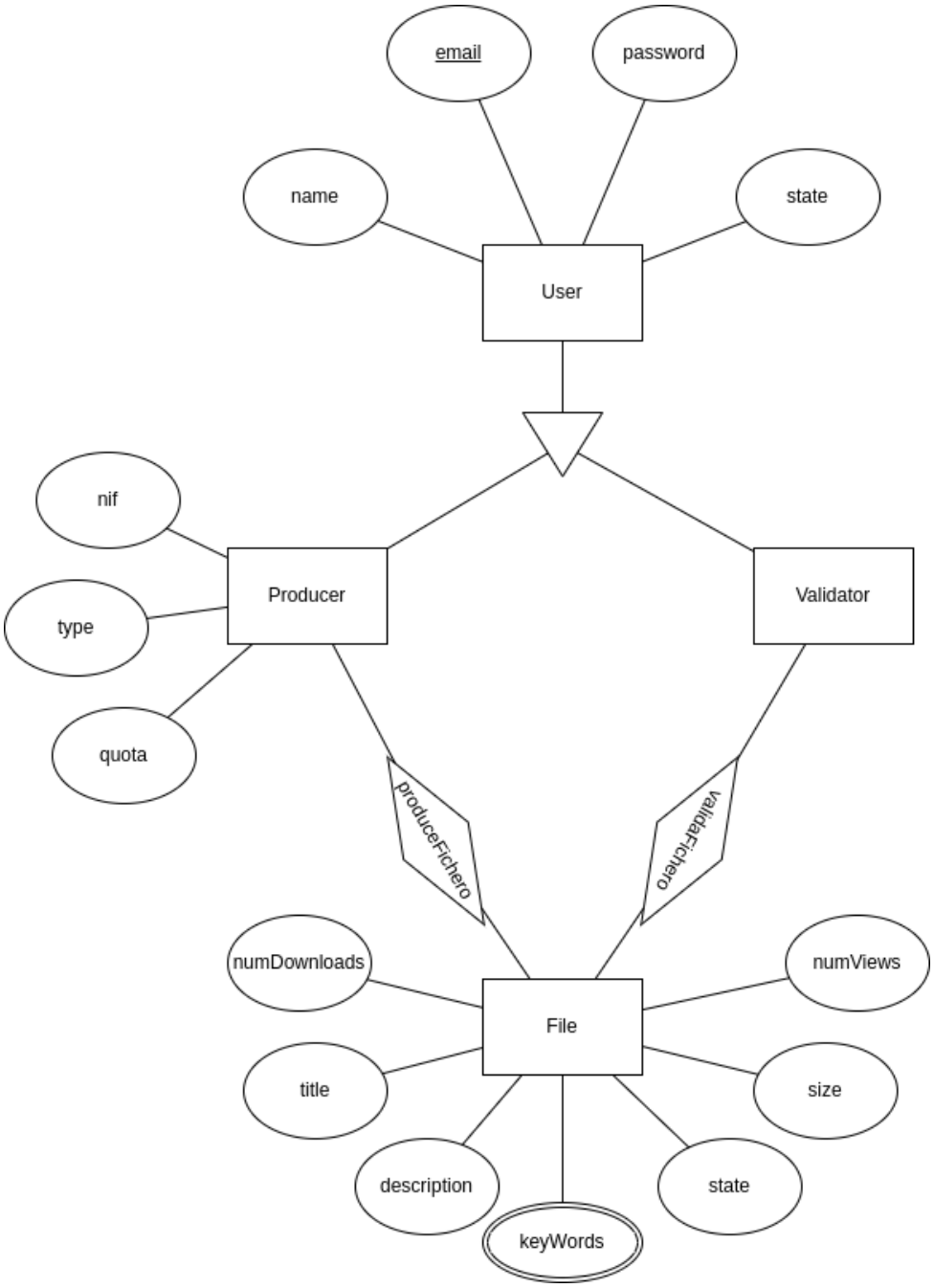

Figura 1: Modelo Conceptual

1.2. Modelo lógico

A continuación se muestra el modelo lógico generado por mysql workbench (ver figura 2).

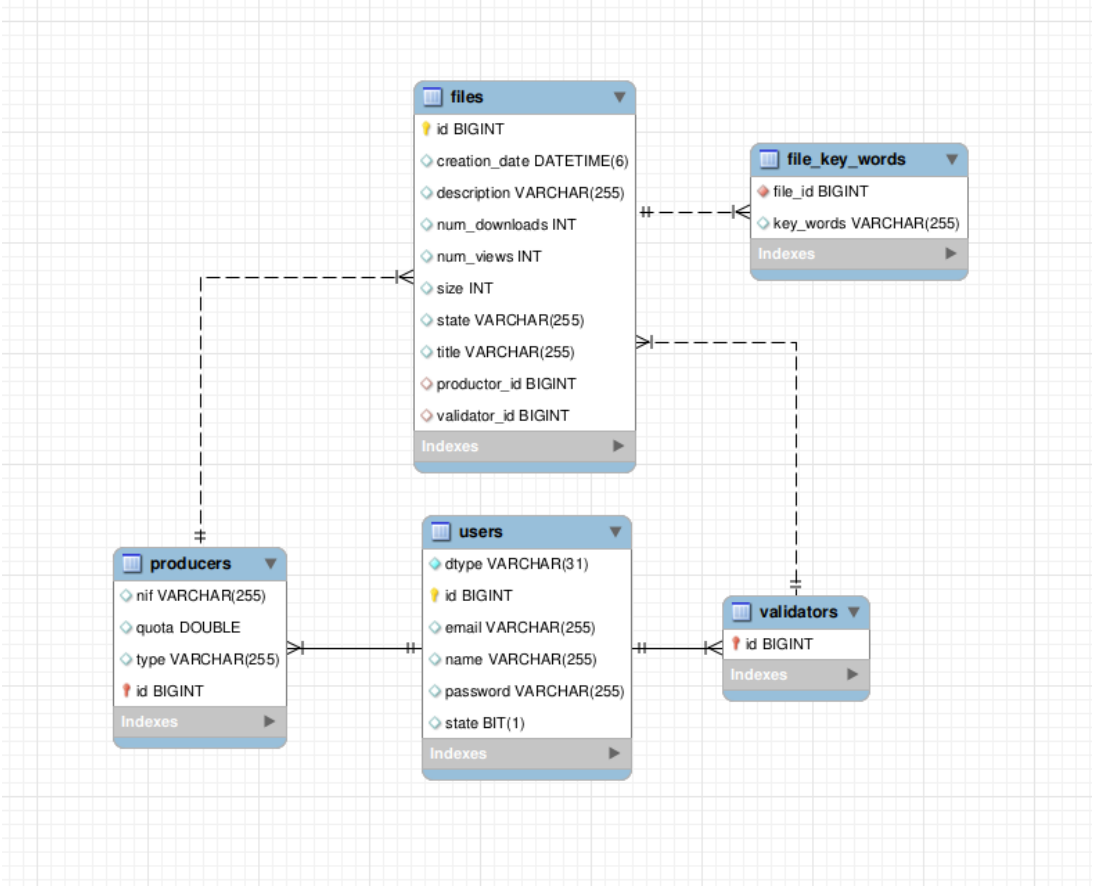

Figura 2: Modelo Lógico

- users: Esta tabla almacena información sobre los usuarios. Cada usuario tiene un identificador único (id), nombre (name), correo electrónico (email), contraseña (password), un campo de estado que indica si el usuario está activo o no (state) y un dtype que se especifica el tipo de usuario almacenados en la fila.
- producers: Esta tabla parece contener información específica de los usuarios que son productores. Incluye un identificador de usuario (id), un número de identificación fiscal (nif), tipo (type) y cuota (quota). El id es una clave foránea que referencia el id en la tabla users.
- validators: Esta tabla parece ser una tabla de enlace que simplemente lista los usuarios que son validadores. Sólo tiene un campo, el identificador de usuario (id), que también es una clave foránea que referencia el id en la tabla users.
- files: Esta tabla almacena información sobre los archivos. Cada archivo tiene un identificador único (id), fecha de creación (creationDate), título (title), descripción (description), estado (state), tamaño (size), número de visualizaciones (numViews), número de descargas (numDownloads), productor (producer_id) y validador (validator_id). Los identificadores de productor y validador son claves foráneas que referencia el id en las tablas producers y validators respectivamente.

- files_keyWords: Esta tabla parece ser una tabla de enlace que almacena palabras clave asociadas a los archivos. Tiene un identificador de archivo (File_id) y una palabra clave (keyWords). El File_id es una clave foránea que referencia el id en la tabla files.

1.2.1. Decisiones de diseño:

Se ha decidido mantener las tablas producers y validators separadas de la tabla users a pesar de que cada fila en producers y validators corresponde a una fila en users. Esto se debe a la necesidad de separar las funciones de productor y validador, y permite a un usuario tener ambos roles.

Los archivos pueden ser asociados tanto a un productor como a un validador, lo que podría sugerir que los productores son los responsables de la creación de archivos, mientras que los validadores los revisan o aprueban. La tabla files_keyWords está diseñada para permitir una relación de muchos a muchos entre archivos y palabras clave. Esto significa que un archivo puede tener varias palabras clave y una palabra clave puede estar asociada a varios archivos.

1.3. Modelos de datos

```
@Entity
@Table(name = "files")
public class File {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @CreationTimestamp
  @Column(updatable = false)
  private Date creationDate;

  private String title;

  private String description;

  @ElementCollection
  private List<String> keyWords;

  private String state;

  private Integer size;

  private Integer numViews;

  private Integer numDownloads;

  @ManyToOne
  @JoinColumn(name = "productor_id")
  private Producer producer;

  @ManyToOne
  @JoinColumn(name = "validator_id")
  private Validator validator;

...
}
```

```
Entity
@Table(name = "users")
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "dtype")
public class User {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  protected Long id;

  protected String name;

  @Column(unique = true)
  protected String email;

  protected String password;

  protected Boolean state;

  @Column(insertable = false, updatable = false)
  private String dtype;
...
}


@Entity
@Table(name = "producers")
@DiscriminatorValue("Producer")
public class Producer extends User {

  private String nif;

  private String type;

  private Double quota;
...
}


@Entity
@Table(name = "validators")
@DiscriminatorValue("Validator")
public class Validator extends User {
...
}
```

1.4. Consultas realizadas en SQL

1.4.1. Repositorio de files

```
 @Query(
    "SELECT f FROM File f WHERE f.state = 'READY' AND :keyword MEMBER OF f.keyWords"
  )
List<File> findPublishedFilesByKeywordSorted(
    @Param("keyword") String keyword,
    Sort sort
);
```

Encuentra todos los archivos cuyo estado es 'READY'y que contienen la palabra clave proporcionada en su lista de palabras clave. Los resultados se devuelven ordenados de acuerdo a un criterio de ordenación proporcionado.

```
 List<File> findAllByProducerId(Long producerId);
```

Encuentra todos los archivos que han sido producidos por el productor con el ID dado.

```
Optional<File> findByIdAndProducerId(Long id, Long producerId);
```

Encuentra un archivo específico (dado por el ID) que ha sido producido por el productor con el ID dado.

```
int deleteByIdAndProducerId(Long id, Long producerId);
```

Borra un archivo específico (dado por el ID) que ha sido producido por el productor con el ID dado.

```
List<File> findAllByState(String state);
```

Encuentra todos los archivos que tienen un estado específico.

```
List<File> findByStateAndProducerName(String state, String producerName);
```

Encuentra todos los archivos con un estado específico que han sido producidos por el productor con el nombre dado.

```
@Modifying
@Query("UPDATE File f SET f.numViews = f.numViews + 1 WHERE f.id = :id")
void incrementNumViews(@Param("id") Long id);
```

Incrementa en 1 el número de vistas de un archivo específico (dado por el ID).

```
@Modifying
@Query(
    "UPDATE File f SET f.numDownloads = f.numDownloads + 1 WHERE f.id = :id"
)
void incrementNumDownloads(@Param("id") Long id);
```

Incrementa en 1 el número de descargas de un archivo específico (dado por el ID).

```
//Q1
@Query(
    "SELECT f FROM File f JOIN FETCH f.producer ORDER BY f.numViews + f.numDownloads DESC"
)
List<File> findTop10ByOrderByNumViewsDescNumDownloadsDesc(Pageable pageable);
```

Encuentra los 10 archivos más populares ordenados por número de vistas y descargas, en orden descendente.

```
//Q2
List<File> findByValidatorAndProducer(Validator validator, Producer producer);
```

Encuentra todos los archivos que han sido validados por un validador específico y producidos por un productor específico. En JPA no se puede limitar el número de resultados que devuelve una consulta por ello se le tiene que pasar un pageable y configurar desde el service el número de resultados a 10.

1.4.2. Repositorio de producers

```
List<Producer> findByStateFalse();
```

Encuentra todos los productores cuyo estado es falso.

```
@Query("SELECT p FROM Producer p WHERE p.quota <= 0.0")
List<Producer> findByQuotaExceeded();
```

Encuentra todos los productores cuya cuota es menor o igual a 0.

```
@Query("SELECT DISTINCT f.producer FROM File f WHERE f.state = 'ERROR'")
List<Producer> findProducersWithErrors();
```

Encuentra todos los productores que han producido archivos con estado ÉRROR'.

```
//Q3
@Query("SELECT p.state, COUNT(p) FROM Producer p GROUP BY p.state")
List<Object[]> countProducersByState();
```

Cuenta el número de productores por cada estado. Te devuelve un array de objetos en la primera posición te devuelve el estado y en la segunda posición te devuelve el número. En el servicio se parsea para mostrar los datos.

```
//Q4
@Query(
    "SELECT p FROM Producer p JOIN File f ON p.id = f.producer.id GROUP BY p HAVING COUNT(f) > 5"
)
List<Producer> findProducersWithMoreThan5Files();
```

Encuentra todos los productores que han producido más de 5 archivos.

1.4.3. Repositorio de usuarios

```
@Query("SELECT u FROM User u WHERE u.email = :email AND u.state = true")
Optional<User> findUserByEmail(@Param("email") String email);
```

Encuentra un usuario por su correo electrónico, siempre y cuando el estado del usuario sea verdadero.

1.5. Pruebas

Para comprobar las consultas sql que no son del sistema se ha creado un sistema que genera usuarios y ficheros aleatoriamente dentro de unos parámetros que se le otorga. Para ello al inicio de la aplicación se puebla la base de datos con estas entidades generadas aleatoriamente y se realizan las consultas para comprobar su funcionamiento.

Se inicia la aplicación para ello se extiende de la clase CommandLineRunner en primer lugar se generan aleatoriamente los datos y luego se lanzan los test de las consultas.

```
@SpringBootApplication
public class SQLDataApplication implements CommandLineRunner {

  @Autowired
  DataInitService ds;

  @Autowired
  TestService ts;

  public static void main(String[] args) {
    SpringApplication.run(SQLDataApplication.class, args);
  }

  @Override
  public void run(String... args) throws Exception {
    ds.load();
    ts.test();
  }
}
```

Clase que se ha creado para generar entidades al inicio de la aplicación:

```
@Service
public class DataInitService {

  @Autowired
  ProducerRepository pr;

  @Autowired
  FileRepository fr;

  @Autowired
  UserRepository us;

  List<Producer> producers = new ArrayList<>();
  List<Validator> validators = new ArrayList<>();
  List<File> files = new ArrayList<>();

  private static final Random RANDOM = new Random();
  private static final String[] STATES = { "ERROR", "PENDING", "READY" };

  private static final List<String> palabras = Arrays.asList( "manzana",  "perro", "casa", "amor",
  ↪  "juego", "coche", "futbol"...);

  public static List<String> getTresPalabrasAleatorias() {
    Collections.shuffle(palabras);
    return palabras.subList(0, 3);
  }

  public void load() {
    this.validators = generateValidators(5);
    this.producers = generateProducers(10);
    this.files = generateFiles(52, producers, validators);
  }

  private List<Validator> generateValidators(int count) {
    List<Validator> validators = new ArrayList<>();
    for (int i = 0; i < count; i++) {
      Validator validator = new Validator();
      validator.setName("Validator " + (i + 1));
      validator.setEmail("validator" + (i + 1) + "@example.com");
      validator.setPassword("password");
      validator.setState(true);
      validators.add(validator);

      us.save(validator);
    }
    return validators;
  }

  private List<Producer> generateProducers(int count) {
    List<Producer> producers = new ArrayList<>();
    IntStream
```

```
      .range(0, count)
      .forEach(i -> {
        Producer producer = new Producer();
        producer.setName("Producer " + (i + 1));
        producer.setEmail("producer" + (i + 1) + "@example.com");
        producer.setPassword("password");
        producer.setState(i < 8);
        producer.setNif("NIF" + (i + 1));
        producer.setType("Type " + (i + 1));
        producer.setQuota(100.0 + i);
        producers.add(producer);

        pr.save(producer);
      });
    return producers;
  }

  private List<File> generateFiles(
    int count,
    List<Producer> producers,
    List<Validator> validators
  ) {
    List<File> files = new ArrayList<>();
    for (int i = 0; i < count; i++) {
      File file = new File();
      file.setTitle("File " + (i + 1));
      file.setDescription("Description for File " + (i + 1));
      file.setState(STATES[i % STATES.length]);
      file.setSize(RANDOM.nextInt(1000) + 500);
      file.setNumViews(RANDOM.nextInt(1000));
      file.setNumDownloads(RANDOM.nextInt(500));
      file.setProducer(producers.get(i % producers.size()));
      file.setKeyWords(getTresPalabrasAleatorias());
      file.setCreationDate(new Date());
      file.setValidator(validators.get(0));
      files.add(file);
      fr.save(file);
    }
    return files;
  }

  public List<Producer> getProducers() {
    return producers;
  }

  public List<Validator> getValidators() {
    return validators;
  }

  public List<File> getFiles() {
    return files;
  }
}
```

Clase que prueba que las consultas funcionan correctamente:

```
@Service
public class TestService {

  @Autowired
  DataInitService dataInitService;

  @Autowired
  PRPNRService prpnrService;

  public void test() {
    // Generar datos
    List<Validator> validators = dataInitService.getValidators();
    List<Producer> producers = dataInitService.getProducers();
    dataInitService.getFiles();

    // Prueba Q1
    List<File> topFiles = prpnrService.findTop10ByViewsAndDownloads();
    System.out.println("Top 10 Files:");
    for (File file : topFiles) {
      System.out.println(file.getTitle());
    }

    // Prueba Q2
    Validator validator = validators.get(0);
    Producer producer = producers.get(0);
    List<File> validatorFiles = prpnrService.findByValidatorAndProducer(
      validator,
      producer
    );
    System.out.println(
      "\nFiles validated by validator and produced by producer:"
    );
    for (File file : validatorFiles) {
      System.out.println(file.getTitle());
    }
    Map<Boolean, Long> producerStateCounts = prpnrService.countProducersByState();
    System.out.println("\nNumber of Producers by state:");
    for (Map.Entry<Boolean, Long> entry : producerStateCounts.entrySet()) {
      System.out.println(
        "State: " +
        (entry.getKey() ? "Active" : "Inactive") +
        ", Count: " +
        entry.getValue()
      );
    }

    // Prueba Q4
    List<Producer> activeProducers = prpnrService.findProducersWithMoreThan5Files();
    System.out.println("\nProducers with more than 5 files:");
    for (Producer activeProducer : activeProducers) {
      System.out.println(activeProducer.getName());
    }
  }
}
```

Servicio que se ha creado para acceder a los repositorios de los datos y hacer las pruebas para las consultas SQL de la Q1 a la Q4:

```
@Service
public class PRPNRService {

  @Autowired
  private FileRepository fileRepository;

  @Autowired
  private ProducerRepository producerRepository;

  public List<File> findTop10ByViewsAndDownloads() {
    return fileRepository.findTop10ByOrderByNumViewsDescNumDownloadsDesc(
      PageRequest.of(0, 10)
    );
  }

  public List<File> findByValidatorAndProducer(
    Validator validator,
    Producer producer
  ) {
    return fileRepository.findByValidatorAndProducer(validator, producer);
  }

  public Map<Boolean, Long> countProducersByState() {
    List<Object[]> results = producerRepository.countProducersByState();
    Map<Boolean, Long> resultMap = new HashMap<>();
    for (Object[] result : results) {
      resultMap.put((Boolean) result[0], (Long) result[1]);
    }
    return resultMap;
  }

  public List<Producer> findProducersWithMoreThan5Files() {
    return producerRepository.findProducersWithMoreThan5Files();
  }
}
```

El resto de consultas se han verificado utilizando llamadas a los endpoints que llaman directamente a través de los servicios a las consultas de la base de datos.

1.5.1. Resultados:

```
Top 10 Files:
File 23
File 29
File 36
File 18
File 34
File 22
File 24
File 47
File 30
File 10

Files validated by validator and produced by producer:
File 1
File 11
File 21
File 31
File 41
File 51
```

```
Number of Producers by state:
State: Inactive, Count: 2
State: Active, Count: 8

Producers with more than 5 files:
Producer 1
Producer 2
```

2. NO SQL

2.1. Descripción del modelo de datos

El modelo no relacional solo tiene dos campos:

- fileId: Es el identificador único de cada documento File. Se anota con @Id, lo que indica que este campo se utiliza como la clave primaria en MongoDB.
- data: Es una lista de objetos, lo que en MongoDB se traduciría en un array de documentos embebidos. Cada objeto en data es un objeto JSON, lo que significa que puede contener múltiples campos y valores de forma estructurada, aunque la estructura exacta no está definida en este modelo. Este campo podría usarse para almacenar una variedad de datos diferentes dentro de un solo documento File.

Se ha añadido a la entrega un fichero de ejemplo (file.txt) que se ha utilizado para pruebas de la parte no relacional.

2.2. Consultas

Este es el repositorio que se ha utilizado para acceder a los ficheros desde Java

```
public interface FileRepository extends MongoRepository<File, Long> {}
```

Las consultas que se han utilizado vienen definidas en la interfaz propia de MongoRepository

```
@Service
public class FileService {

  @Autowired
  FileRepository fr;

  public List<File> findAll() {
    return this.fr.findAll();
  }

  public File create(File file) {
    return this.fr.save(file);
  }

  public File findById(Long id) {
    return this.fr.findById(id).orElse(null);
  }

  public void deleteById(Long id) {
    this.fr.deleteById(id);
  }
}
```

Las pruebas que se han realizado han sido efectuadas a través de llamadas a la API de este componente

Se ha creado un script que puebla la base de datos de ficheros, este script se carga en un contenedor de docker y cuando la aplicación está iniciada se ejecuta poblando la base de datos no relacional de ficheros de prueba.

```
#!/bin/bash

for ((i=1; i<=50; i++))
do
    filename="file$i.txt"
    size=$((i * 1000))

    curl -F "fileData=@$(pwd)/$filename" -F "title=Prueba" -F "description=Probando la API" -F
    ↪  "keyWords=uno,dos,tres,cuatro" -F "size=$size" http://localhost:8321/api/v1/producers/files/1

    echo "Llamada $i completada."
done
```

La parte relacional se ha hecho desde java y en esta parte desde fuera para mostrar distintas posibilidades de inicialización de bases de datos.

3. Problemas con la consistencia de los datos

Cuando se utilizan dos bases de datos diferentes, una relacional y una no relacional, para hacer referencia al mismo objeto, en este caso un archivo, puede surgir un problema de integridad de los datos. Por ejemplo, la base de datos relacional podría contener metadatos sobre el archivo, como su ID, título, descripción, fecha de creación, etc., mientras que la base de datos no relacional almacena el archivo en sí. El problema se presenta cuando intentamos asegurar que los datos en ambas bases de datos sean coherentes y estén sincronizados, es decir, que describan y se refieran al mismo archivo.

La de sincronización puede ocurrir por diversas razones, como las fallas durante la escritura en una de las bases de datos, las actualizaciones que no se replican en ambas bases de datos, o simplemente los errores humanos durante la entrada de datos. Este tipo de inconsistencias pueden llevar a problemas serios, como referencias a archivos que ya no existen, archivos que se supone que existen, pero no están en la base de datos no relacional. Este problema es un desafío importante en la gestión de datos distribuidos y requiere una cuidadosa consideración del diseño de la base de datos y la aplicación.

Una solución implementada para mitigar el problema de integridad de datos entre la base de datos relacional y la no relacional es la ejecución periódica de una aplicación de validación de consistencia. Esta aplicación se encarga de comparar los identificadores de los archivos en ambas bases de datos. Si descubre un identificador en una base de datos que no tiene correspondencia en la otra, procede a eliminar la entrada en la base de datos donde sí existe. Esto se hace tanto para la base de datos relacional como para la no relacional, garantizando así que solo los archivos con identificadores presentes en ambas bases de datos se mantengan.

Se trata de una aplicación springboot montada sobre un CronJob de kubernetes que consulta los repositorios de datos de ambas bases de datos.

```
@Service
public class DataIntegrityService {

  @Value("${url.data.service}")
  private String DATA_SERVICE_URL;

  @Value("${url.process.service}")
  private String PROCESS_SERVICE_URL;

  private final RestTemplate restTemplate = new RestTemplate();

  public void checkDataIntegrity() {
    List<Long> dataServiceIds = fetchIds(DATA_SERVICE_URL);
    List<Long> processServiceIds = fetchIds(PROCESS_SERVICE_URL);

    dataServiceIds
      .stream()
      .filter(id -> !processServiceIds.contains(id))
      .forEach(id -> deleteId(DATA_SERVICE_URL, id));

    processServiceIds
      .stream()
      .filter(id -> !dataServiceIds.contains(id))
      .forEach(id -> deleteId(PROCESS_SERVICE_URL, id));
  }

  private List<Long> fetchIds(String serviceUrl) {
    ResponseEntity<List<Long>> response = restTemplate.exchange(
      serviceUrl + "/files",
      HttpMethod.GET,
      null,
      new ParameterizedTypeReference<List<Long>>() {}
    );
    return response.getBody();
  }

  private void deleteId(String serviceUrl, Long id) {
    restTemplate.execute(
      serviceUrl + "/files/" + id,
      HttpMethod.DELETE,
      null,
      (ResponseExtractor<Void>) null
    );
  }

  @EventListener
  public void onApplicationEvent(ContextRefreshedEvent event) {
    checkDataIntegrity();
  }
}
```