Documento Implementación Entrega Proyecto 2 Sistrans

Tomás Velásquez – 202311016

Daniel Vargas – 202123892

Alejandro Parada – 202313816

Descripción del RF9: Agendar un servicio de salud por afiliado (Transaccional)

El RF9 le permite a un afiliado agendar un servicio de salud de manera transaccional, asegurando la consistencia de los datos y la disponibilidad del servicio en un entorno concurrente dado su aislamiento. Este requerimiento se implementó utilizando el método agendarServicioPorAfiliado en la clase AgendarCitaService. El proceso comienza verificando la disponibilidad del servicio solicitado mediante el método consultarDisponibilidadServicio lo que referencia el RF7.1 y retorna una lista de horarios disponibles para el servicio. Luego, se valida que la fecha y hora solicitadas por el afiliado coincidan con uno de los horarios disponibles, ajustando la hora al inicio de la hora (minuto 00, segundo 00) para mantener un formato estándar. Si la fecha no está disponible, se lanza una excepción con un mensaje claro para el usuario. Finalmente, el método delega la creación de la cita al método agendarServicio lo que referencia al RF7.2 y este realiza las validaciones adicionales y persiste la cita en la base de datos. Nuestro proceso se ejecuta dentro de una transacción con aislamiento SERIALIZABLE y rollBack en caso de fallos por elección del grupo y para mantener la concordancia y proteger la información, lo que garantiza que no haya conflictos de concurrencia y que los datos sean consistentes incluso en escenarios de alta concurrencia.

```java
//RF9 - Agendar un servicio de salud por afiliado- RF7 transaccional
@Transactional(isolation = Isolation.SERIALIZABLE, rollbackFor = Exception.class)
public AgendarCita agendarServicioPorAfiliado(AgendarCita cita) {
    Long servicioId = cita.getServicioDeSalud().getIdServicio();
    List<Object[]> disponibilidad = consultarDisponibilidadServicio(servicioId);

    if (disponibilidad.isEmpty()) {
        logger.warn(format:"No hay disponibilidad para el servicio con ID: {}", servicioId);
        throw new RuntimeException("No hay disponibilidad para el servicio con ID: " + servicioId);
    }

    // Obtener solo la fecha y hora de la cita solicitada (ignorando minutos y segundos)
    String fechaCitaStr = cita.getFechaHora().format(java.time.format.DateTimeFormatter.ofPattern(pattern:"yyyy-MM-dd HH"));

    boolean fechaDisponible = disponibilidad.stream()
        .map(obj -> ((String) obj[2]).substring(beginIndex:0, endIndex:13)) // Obtener solo año-mes-día hora
        .anyMatch(fecha -> fecha.equals(fechaCitaStr));

    if (!fechaDisponible) {
        logger.warn(format:"La fecha {} no está disponible para el servicio {}. Las citas deben ser al inicio de cada hora",
                cita.getFechaHora(), servicioId);
        throw new RuntimeException(message:"La fecha solicitada no está disponible. Por favor seleccione una hora disponible (al inicio de cada hora, XX:00:00)");
    }

    // Ajustar la hora de la cita al inicio de la hora (minuto 00, segundo 00)
    LocalDateTime horaAjustada = cita.getFechaHora()
        .withMinute(minute:0)
        .withSecond(second:0)
        .withNano(nanoOfSecond:0);
    cita.setFechaHora(horaAjustada);

    return agendarServicio(cita);
}
```

AgendarCitaController y un nuevo PostMapping como lo es "…/agendar" y se le entregará la información que consumirá el repositorio y servicio a través del Body el cual es un objeto de tipo AgendarCita.

```java
//RF9 - Agendar un servicio de salud por afiliado- RF7 transaccional
@PostMapping("/agendar")
@Transactional(isolation = Isolation.SERIALIZABLE)
public ResponseEntity<?> agendarServicioPorAfiliado(@RequestBody AgendarCita cita) {
    logger.debug(format:"POST /api/citas/agendar - Body: {}", cita);
    try {
        AgendarCita nuevaCita = citaService.agendarServicioPorAfiliado(cita);

        // Create a simplified response
        Map<String, Object> response = new HashMap<>();
        response.put(key:"idCita", nuevaCita.getIdCita());
        response.put(key:"fechaHora", nuevaCita.getFechaHora());
        response.put(key:"afiliadoId", nuevaCita.getAfiliado().getNumeroDocumento());
        response.put(key:"medicoId", nuevaCita.getMedico().getNumeroDocumento());
        response.put(key:"servicioId", nuevaCita.getServicioDeSalud().getIdServicio());
        if (nuevaCita.getOrdenDeServicio() != null) {
            response.put(key:"ordenId", nuevaCita.getOrdenDeServicio().getIdOrden());
        }

        logger.info(format:"Cita agendada exitosamente: {}", nuevaCita);
        return new ResponseEntity<>(response, HttpStatus.CREATED);
    } catch (RuntimeException e) {
        logger.error(format:"Error al agendar cita: {}", e.getMessage(), e);
        return ResponseEntity.badRequest().body(Collections.singletonMap(key:"error", e.getMessage()));
    }
}
```

Descripción del RFC5:

Consultar disponibilidad de un servicio de salud - SERIALIZABLE

El RFC5 permite consultar la disponibilidad de un servicio de salud específico, mostrando los horarios disponibles para agendar citas. Este requerimiento se implementó mediante el método consultarDisponibilidadServicio en la clase AgendarCitaService. El método utiliza el repositorio AgendarCitaRepository para ejecutar una consulta que retorna los horarios disponibles para el servicio solicitado. Los resultados incluyen información relevante como el médico, la IPS y la fecha del horario disponible. Además, se utiliza el logger para registrar la cantidad de horarios disponibles y los detalles de cada uno, lo que facilita la depuración y el monitoreo del sistema.

```
@Transactional(readOnly = true, isolation = Isolation.SERIALIZABLE)
public List<Object[]> DisponibilidadServicio(Long servicioId, Long medicoId, String fechaInicio, String fechaFin) {
try {
    Thread.sleep(millis:30000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    throw new RuntimeException(message:"Thread was interrupted", e);
}
return citaRepository.findDisponibilidadServicioTransaccional(servicioId, medicoId, fechaInicio, fechaFin);
}
```

Para este RFC se le pide tener un temporizador de 30 segundos previo a mandar la sentencia SQL por lo cual se usa el thread.sleep en milisegundos de 30000, además de ellos se exige el nivel de aislamiento como SERIALIZABLE y el rollBack en caso de no completar en su totalidad la transacción (atomicidad) . Y para este RFC así mismo como para el RFC6 se construyó una nueva sentencia SQL para consultar la disponibilidad sin considerar las 4 semanas de gracia pedidas anteriormente, por ende el repositorio al que responde es:

```
// Buscar disponibilidad de citas para un servicio en un rango de fechas específico
@Query(value = "SELECT ac.IDCITA, ac.FECHA_HORA, m.NOMBRE, s.NOMBRE " +
           "FROM AGENDARCITA ac " +
           "JOIN MEDICO m ON ac.MEDICO_NUMERODOCUMENTO = m.NUMERODOCUMENTO " +
           "JOIN SERVICIODESALUD s ON ac.SERVICIODESALUD_ID = s.ID_SERVICIO " +
           "WHERE ac.SERVICIODESALUD_ID = :servicioId " +
           "AND ac.MEDICO_NUMERODOCUMENTO = :medicoId " +
           "AND TO_CHAR(ac.FECHA_HORA, 'YYYY-MM-DD HH24:MI:SS') BETWEEN :fechaInicio AND :fechaFin",
    nativeQuery = true)
List<Object[]> findDisponibilidadServicioTransaccional(@Param("servicioId") Long servicioId,
                                                       @Param("medicoId") Long medicoId,
                                                       @Param("fechaInicio") String fechaInicio,
                                                       @Param("fechaFin") String fechaFin);
```

La forma mas sencilla para evitar transformaciones de tiempos, strings y datetimes es pasar directamente como @param las fechas ya en el formato deseado por ende se hace uso de un nuevo @GetMapping como (“…/disponibilidad/serializable”) el cual no requiere un body sino el paso de parámetros desde la propia url siendo concatenada con % y comas.

```
// Consultar disponibilidad de servicio con aislamiento SERIALIZABLE
@GetMapping("/disponibilidad/serializable")
public ResponseEntity<List<Object[]>> consultarDisponibilidadServicioSerializable(
        @RequestParam Long servicioId,
        @RequestParam Long medicoId,
        @RequestParam String fechaInicio,
        @RequestParam String fechaFin) {
    try {
        List<Object[]> disponibilidad = citaService.DisponibilidadServicio(servicioId, medicoId, fechaInicio, fechaFin);
        return new ResponseEntity<>(disponibilidad, HttpStatus.OK);
    } catch (Exception e) {
        logger.error(format:"Error al consultar disponibilidad (Serializable): {}", e.getMessage(), e);
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Descripción del RFC6:

Consultar disponibilidad de un servicio de salud - READ COMMITTED

El RFC6 permite consultar la disponibilidad de un servicio de salud así mismo como el anterior RFC, asegurando que se cumplan todas las condiciones necesarias para completar el proceso como el intervalo de tiempos, el id del servicio y del médico. Este requerimiento se implementó de igual manera al otro requerimiento por ende no se va a hondar en su estructura, sencillamente la diferencia es que ahora el servicio hace uso de un nivel de aislamiento distinto de READ COMMITTED y para este, aunque se usa la misma sentencia SQL, si cambia el @GetMapping puesto que ahora es ("…/disponibilidad/read-committed")

```
@Transactional(readOnly = true, isolation = Isolation.READ_COMMITTED)
public List<Object[]> DisponibilidadServicioRead(Long servicioId, Long medicoId, String fechaInicio, String fechaFin) {
    try {
        Thread.sleep(millis:30000);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException(message:"Thread was interrupted", e);
    }
    return citaRepository.findDisponibilidadServicioTransaccional(servicioId, medicoId, fechaInicio, fechaFin);
}
```

Y la forma de hacer su llamado en Postman será:

```
// Consultar disponibilidad de servicio con aislamiento READ_COMMITTED
@GetMapping("/disponibilidad/read-committed")
public ResponseEntity<List<Object[]>> consultarDisponibilidadServicioReadCommitted(
        @RequestParam Long servicioId,
        @RequestParam Long medicoId,
        @RequestParam String fechaInicio,
        @RequestParam String fechaFin) {
    try {
        List<Object[]> disponibilidad = citaService.DisponibilidadServicioRead(servicioId, medicoId, fechaInicio, fechaFin);
        return new ResponseEntity<>(disponibilidad, HttpStatus.OK);
    } catch (Exception e) {
        logger.error(format:"Error al consultar disponibilidad (Read Committed): {}", e.getMessage(), e);
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

ESCENARIOS DE PRUEBA:

Escenario 1

En este caso de prueba se va a hacer uso de una solicitud SERIALIZABLE para la cual se espera y así mismo se hace uso de nativequery= True por lo que no se hará uso de los locks sino de un protocolo de aislamiento donde se respetan los bloqueos a nivel de fila o tabla impuestos por la base de datos. Gracias a lo dicho anteriormente podemos generar una línea de tiempo sobre la transacción y este nos permite conocer con mayor claridad lo sucedido.

Tiempo	Acción	Descripción
t=0s	Usuario A ejecuta RFC5	Se lanza una consulta a la agenda usando nivel de aislamiento ***serializable***. Se inicia una transacción que accede a filas de disponibilidad de agenda. La base de datos bloquea estas filas para asegurar consistencia.
t=10s	Usuario B ejecuta RF9	El usuario intenta registrar una orden de servicio de salud, afectando las mismas filas de disponibilidad que RFC5 está leyendo (e.g., cupos, fechas, turnos). Se ejecuta mediante nativeQuery=true, así que las instrucciones SQL intentan realizar la inserción y actualización directamente.
t=10s-t=30s	RF9 entra en espera o es bloqueada	La base de datos detecta conflicto de concurrencia y bloquea la ejecución de RF9. Dado que RFC5 tiene bloqueos activos (lectura serializable), RF9 ***no puede realizar escrituras en esas filas*** hasta que finalice la transacción de RFC5.
t=30s	RFC5 finaliza (commit o rollback)	Se liberan los bloqueos sobre las filas consultadas.
t=31s	RF9 se desbloquea y ejecuta	La inserción y actualización de la orden de servicio se realiza correctamente.

En este escenario, se ejecutó primero la consulta RFC5, la cual accede a la agenda de disponibilidad de un servicio de salud utilizando un nivel de aislamiento serializable, el más restrictivo. Dado que este tipo de aislamiento simula la ejecución completamente secuencial de las transacciones, la base de datos garantiza que ningún otro proceso pueda modificar los datos leídos durante la ejecución de RFC5. Esta consulta se mantuvo activa durante 30 segundos y, antes de su finalización, se inició de forma concurrente la ejecución del componente RF9, encargado de registrar una orden de servicio de salud, usando SQL directo (nativeQuery=true) y por ende sin intervención de una capa de persistencia (como JPA). Dado que RF9 intentaba modificar los mismos datos que RFC5 estaba consultando (disponibilidad de agenda), la base de datos bloqueó la ejecución de RF9 hasta que la transacción de RFC5 finalizara. Esto se debe a que, bajo el aislamiento serializable, cualquier intento de escritura sobre datos leídos por otra transacción activa genera un

conflicto que impide el acceso concurrente para mantener la integridad de la "imagen actual" de datos que RFC5 está utilizando.

Una vez que la transacción de RFC5 terminó y liberó los bloqueos (mediante commit o rollback), RF9 pudo continuar con normalidad, registrando la orden de servicio en la base de datos. Como resultado, la orden registrada por RF9 no apareció en los resultados devueltos por RFC5, ya que esta consulta accedió a una imagen consistente de la agenda anterior al momento de la ejecución de RF9. El nivel serializable impidió que RFC5 "viera" cualquier modificación concurrente mientras estaba activa, garantizando un aislamiento total de los cambios realizados por RF9.

El escenario que se probo fue:

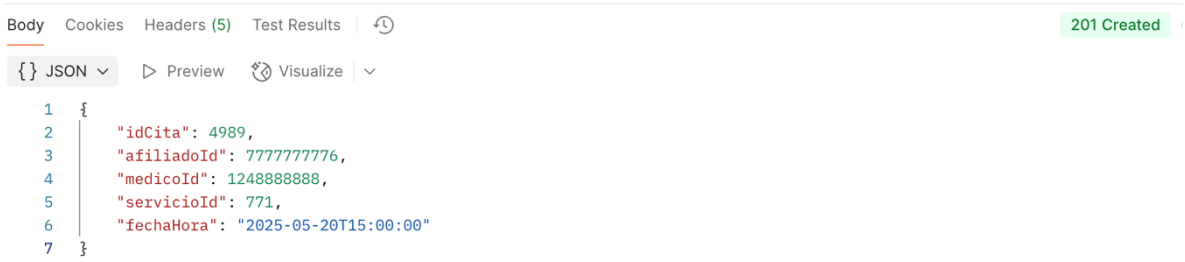
URL del RFC5:

http://localhost:8080/api/citas/disponibilidad/serializable?servicioId=771&medicoId=1248888888&fechaInicio=2025-05-18%2008:00:00&fechaFin=2025-05-22%2018:00:00

Body del RF9:

```
{
  "idCita": 4989,
  "fechaHora": "2025-05-20T15:00:00",
  "afiliado": {
    "numeroDocumento": 7777777776
  },
  "medico": {
    "numeroDocumento": 1248888888
  },
  "servicioDeSalud": {
    "idServicio": 771
  }
}
```

El resultado final es que se logró crear el servicio:



Escenario 2

En este caso de prueba se va a hacer uso de una solicitud READ COMMITED para la cual se espera y así mismo se hace uso de nativequery= True por lo que no se hará uso de los locks sino de un protocolo de aislamiento donde se respetan los bloqueos a nivel de fila o tabla impuestos por la base de datos. Gracias a lo dicho anteriormente podemos generar una línea de tiempo sobre la transacción y este nos permite conocer con mayor claridad lo sucedido.

Tiempo	Acción	Descripción
t=0s	Usuario A ejecuta RFC6	Se inicia la consulta a la agenda bajo aislamiento **read committed**. La transacción accede a los datos disponibles en ese momento y continúa viva durante 30 segundos.
t=10s	Usuario B ejecuta RF9	Se inicia la ejecución directa de SQL para registrar una orden de servicio. La operación inserta la orden y actualiza la agenda (e.g., decrementa disponibilidad).
t=15s	RF9 finaliza (commit)	Los cambios en la base de datos se confirman.
t=16s	RFC6 accede a la agenda (segunda mitad de la consulta)	Dependiendo del momento en que RFC6 accede a las filas modificadas, puede leer datos **ya modificados** por RF9.
t=30s	RFC6 finaliza	La consulta completa y devuelve resultados.

En este segundo escenario, la ejecución inició con la consulta RFC6, que también accede a la agenda de disponibilidad de un servicio de salud, pero esta vez utilizando el nivel de aislamiento read committed, el cual permite leer únicamente los datos que han sido confirmados (committed) por otras transacciones. A diferencia del primer escenario, este nivel de aislamiento no impide que otras transacciones realicen escrituras sobre los datos leídos. Por lo tanto, mientras RFC6 permanecía activa, se ejecutó de forma concurrente el componente RF9, encargado de registrar una orden de servicio de salud. Este componente, al igual que antes, utilizó ejecución directa con nativeQuery=true, por lo que se aplicaron directamente las instrucciones SQL sobre las tablas de la base de datos. RF9 logró registrar la orden exitosamente y realizó el commit de la transacción antes de que RFC6 completara la consulta.

Dado que RFC6 estaba utilizando el nivel read committed, sí fue capaz de ver los cambios realizados por RF9, siempre y cuando accediera a las filas modificadas después del commit de RF9. Es decir, si la consulta RFC6 leyó los datos afectados por RF9 luego de que este terminara su operación (por ejemplo, una agenda con cupos ya reducidos tras la creación

de la orden), entonces esos cambios aparecieron reflejados en los resultados. Como consecuencia, la orden de servicio registrada por RF9 sí pudo aparecer reflejada en los datos devueltos por RFC6, evidenciando que este nivel de aislamiento permite una mayor concurrencia, aunque con menos control sobre la consistencia en comparación con el modo serializable.

URL del RFC5:

http://localhost:8080/api/citas/disponibilidad/read-committed?servicioId=771&medicoId=1248888888&fechaInicio=2025-05-18%2008:00:00&fechaFin=2025-05-22%2018:00:00

Body del RF9:


```
{
  "idCita": 4333,
  "fechaHora": "2025-05-20T19:00:00",
  "afiliado": {
    "numeroDocumento": 7777777776
  },
  "medico": {
    "numeroDocumento": 1248888888
  },
  "servicioDeSalud": {
    "idServicio": 771
  }
}
```


El resultado final es que se logró crear el servicio:

```
[
    4987,
    "2025-05-20T21:00:00.000+00:00",
    "Dr. Prueba Nuevo",
    "Servicio Prueba 2"
],
[
    4333,
    "2025-05-20T19:00:00.000+00:00",
    "Dr. Prueba Nuevo",
    "Servicio Prueba 2"
],
[
    4966,
    "2025-05-20T23:00:00.000+00:00",
    "Dr. Prueba Nuevo",
    "Servicio Prueba 2"
],
[
    4989,
    "2025-05-20T20:00:00.000+00:00",
    "Dr. Prueba Nuevo"
```