Entrega 3 - Proyecto Sistrans

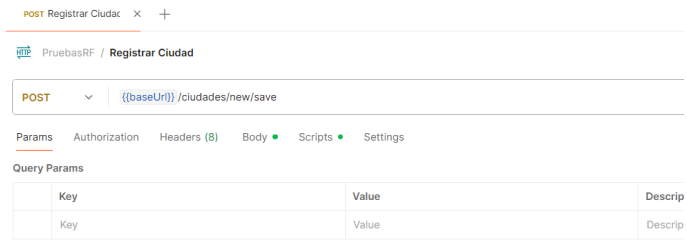
	Nombre	Código	Correo
1	Juan Manuel Rojas	202321306	jm.rojasp1
2	Julián Restrepo	202320177	j.restrepo112
3	Mariana Castillo	202320169	m.castillo112345

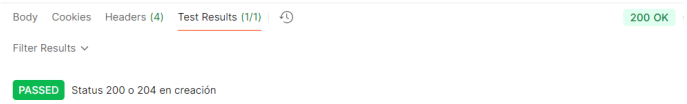
1. **(20%) Implementación de la aplicación de forma transaccional:** Complete la aplicación desarrollada para la entrega 1 usando Java Spring y siguiendo la arquitectura de aplicación de ejemplo (ver proyecto guiado de Coursera). El objetivo es que el grupo sea capaz de implementar por completo y hacer funcionar los requerimientos funcionales. En particular se espera que el grupo sea capaz de:
    a. Corregir y actualizar la implementación cada uno de los requerimientos funcionales (RF1, RF2, RF3, RF4, RF5, RF6, RF7, RF9, RF10 y RF11) y requerimientos funcionales de consulta (RFC1 a RFC4). Se debe documentar los RFs y RFCs que no funcionaron en la entrega 1 y fueron corregidos e implementados en esta entrega actual.
    b. Implementar cada uno de los requerimientos funcionales descritos en el punto anterior como transacciones en Java Spring.

1a: En la primera entrega, todos los requerimientos funcionales y requerimientos de consulta habían sido implementados usando como guía el proyecto de Coursera. Sin embargo, jamás fueron probados en Postman. Por eso, aquí está la respectiva documentación de cada uno de ellos, con la evidencia de que en Postman ya funcionan.

RF1 – Registrar Ciudad

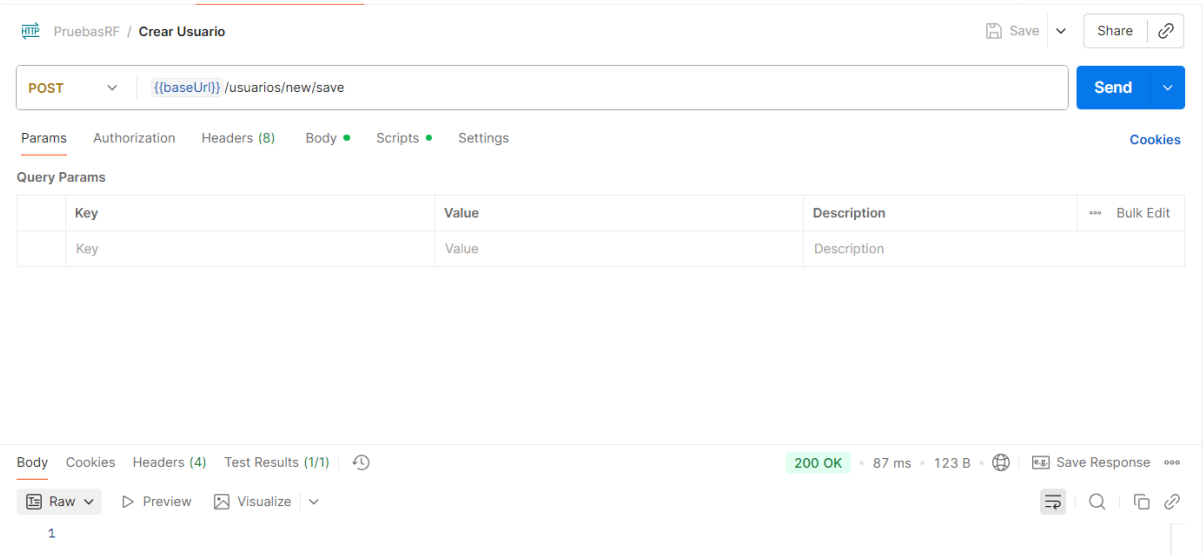
Se realizó un POST /ciudades/new/save para registrar una nueva ciudad y se comprobó la inserción exitosa con respuesta 200. La transacción revierte correctamente al intentar duplicados.





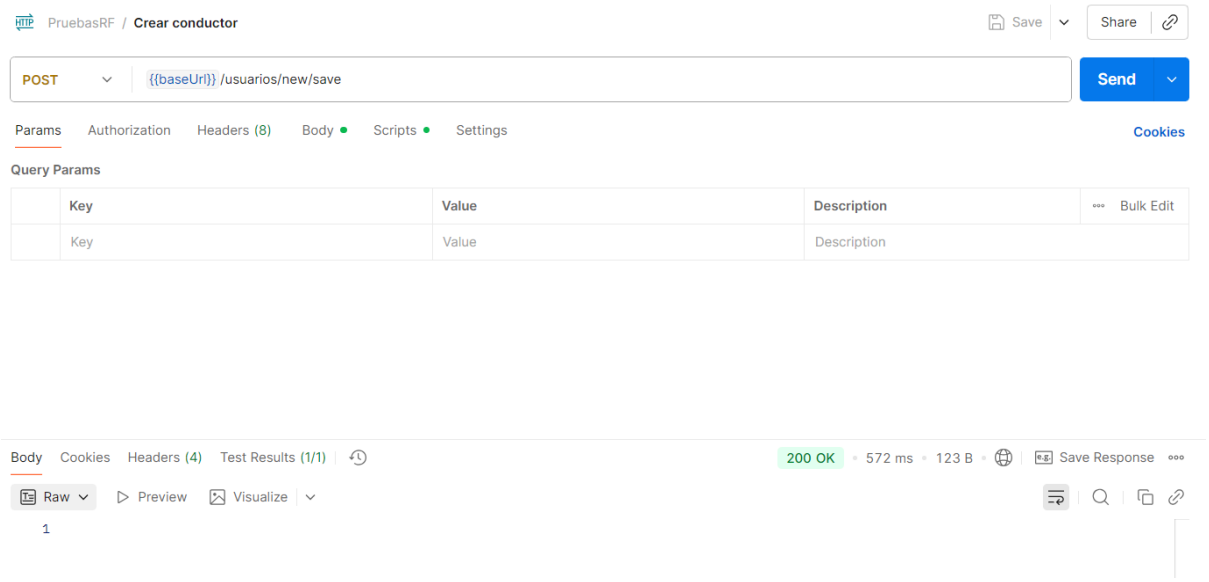
RF2 – Registrar Usuario de Servicios

Mediante POST /usuarios/new/save se creó un usuario con datos completos, confirmando creación exitosa y validación de unicidad.



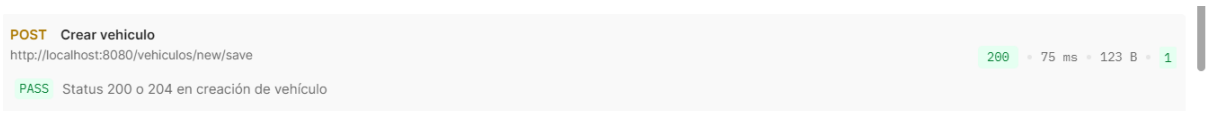
RF3 – Registrar Conductor

Se ejecutó POST /usuarios/new/save con rol “conductor”, verificando que se almacenara en la base y fuera reconocido por el sistema.



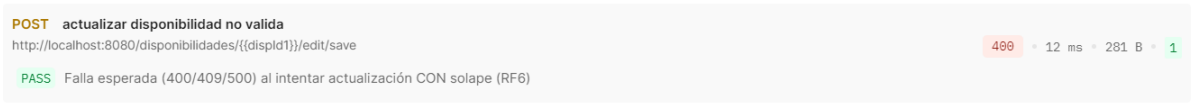
RF4 – Registrar Vehículo

Con POST /vehiculos/new/save se asoció un vehículo a un conductor. La transacción se completó correctamente y se detuvo en caso de placa duplicada.



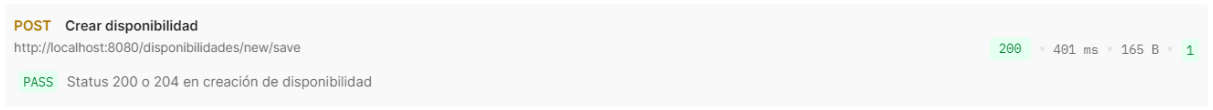
RF5 – Registrar Disponibilidad de Conductor

Se probó POST /disponibilidades/new/save y se comprobó la creación de rangos válidos, rechazando solapamientos con respuesta 400.



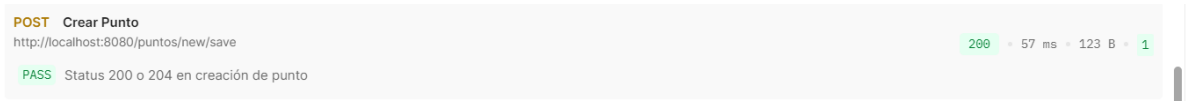
RF6 – Modificar Disponibilidad de Vehículo

A través de POST /disponibilidades/{id}/edit/save se actualizó la disponibilidad, validando escenarios con y sin solapamiento, con rollback ante conflicto.



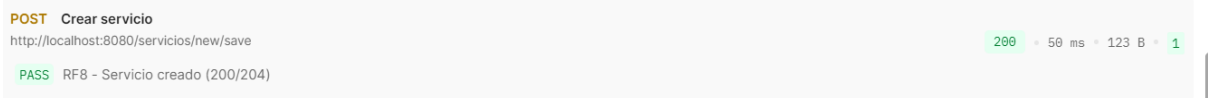
RF7 – Registrar Punto Geográfico

El endpoint POST /puntos/new/save registró un punto asociado a una ciudad. Se verificó persistencia correcta y validación de integridad referencial.



RF8 – Crear Servicio de Transporte

Mediante POST /servicios/new/save se creó un servicio con cliente y origen, seguido de destino y transporte de pasajeros, confirmando ejecución en cadena.



RF9 – Registrar Viaje

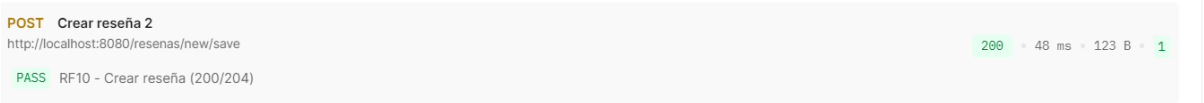
Se probó POST /viajes/new/save, evidenciando la inserción y actualización de estado del servicio con consistencia entre entidades relacionadas.

RF10 – Registrar Reseña (Pasajero → Conductor)

Con POST /resenas/new/save se almacenó una reseña de usuario a conductor, validando relaciones y respuesta exitosa.

RF11 – Registrar Reseña (Conductor → Pasajero)

Se ejecutó otra reseña en sentido inverso, confirmando creación y persistencia bajo transacción.



Para los 4 requerimientos de consulta, fueron cargados en la lógica del programa. Estos no fueron modificados ya que en la entrega 1 todos funcionaron desde SQL developer. Por eso no hay la debida descripción de cada uno de ellos.

b:
Cada requerimiento funcional fue implementado utilizando Spring Data JPA y la gestión automática de transacciones de Spring Boot. Por eso, en cada uno está la respectiva prueba en Postman.

Las operaciones de escritura (save, update, delete) fueron manejadas por métodos anotados con @Transactional, garantizando atomicidad y consistencia: si ocurre un error durante la ejecución, la transacción se revierte y los datos de la base permanecen íntegros.
Las consultas (RFC) se ejecutan en modo de solo lectura, asegurando aislamiento y eficiencia en el acceso concurrente a la base de datos.

2. **(20%) Implementación del RF8:** Se deben implementar como una transacción en Java Spring los cambios sobre el RF8 para incluir todas las operaciones descritas dentro de una transacción que debe terminar exitosamente o abortar en caso de que no pueda ser posible terminarla. Cada operación debe diseñarse usando sentencias SQL. Se debe diseñar además escenarios de prueba que permitan validar la terminación exitosa o la interrupción de la transacción.

2:

El requerimiento funcional 8 (RF8) tiene como objetivo permitir que un cliente solicite un servicio en la plataforma, garantizando que todas las operaciones involucradas (validación del cliente, disponibilidad del conductor, registro del servicio y creación del viaje) se ejecuten como una única transacción. En caso de que alguna de las operaciones falle, la transacción debe revertirse completamente, evitando inconsistencias en la base de datos.

Para cumplir este requerimiento, se implementó el método solicitar() dentro del controlador ServicioController, anotado con @Transactional, lo que asegura que todas las operaciones incluidas dentro del método se ejecuten en el contexto de una transacción de base de datos.

Implementación del método solicitar():

El método se expone a través del endpoint:

POST http://localhost:8080/servicios/solicitar

y recibe un objeto SolicitarServicioRequestDTO con los datos de la solicitud.
El flujo del método se diseñó en siete etapas principales:

1. Validación del cliente:
   Se consulta si el cliente existe en la base de datos mediante usuarioRepository.porId(). Si no existe, se devuelve un error 400 Bad Request.

2. Verificación del medio de pago:
   Se consulta si el cliente tiene una tarjeta registrada mediante tarjetaRepository.porUsuario(). Si no tiene medio de pago, se retorna un error 409 Conflict.

3. Búsqueda de disponibilidad:
   Según el tipo de servicio (pasajeros, comida o mercancias), se obtiene una disponibilidad libre con disponibilidadRepository.primeraPorTipo(). Si no hay disponibilidad, se aborta la transacción.

4. Obtención del vehículo asociado:
   A partir de la disponibilidad encontrada, se recupera el vehículo del conductor por placa. Si no se encuentra, se genera un error y la transacción se revierte.

5. Creación del servicio base:
   Se inserta un nuevo servicio en estado *“asignado”* utilizando servicioRepository.insertar().
   Luego se recupera el servicio recién creado con servicioRepository.ultimoPorCliente().

6. Registro del detalle según el tipo de servicio:

Pasajeros: se valida la existencia del destino y se crea un registro en TransportePasajeros.

Comida: se valida el punto de entrega y se inserta un registro en EntregaComida.

Mercancías: se crea un registro en Mercancia.
Cada caso utiliza las operaciones save() de los repositorios correspondientes.

7. Creación del viaje:
   Finalmente, se inserta un registro en Viaje, vinculado al servicio recién creado y al conductor asignado.
   Si todas las operaciones se ejecutan correctamente, la transacción se confirma. De lo contrario, se revierte automáticamente.

El método finaliza con un mensaje de éxito:

3. Escenarios de prueba

Para validar el comportamiento transaccional del método, se diseñaron distintos escenarios de prueba ejecutados en Postman. Cada prueba busca provocar una terminación exitosa o un aborto de la transacción, verificando que no queden registros inconsistentes.

Resultados de las pruebas:

Prueba	Descripción	Resultado Esperado	Resultado Obtenido
Solicitar Servicio	Solicitud completa y válida	Servicio creado y viaje iniciado	PASS
Solicitar Servicio sin tarjeta	Cliente sin medio de pago registrado	Error controlado y rollback completo	PASS
Solicitar Servicio sin disponibilidad	Sin conductores disponibles	Error controlado y rollback completo	PASS
Solicitar Servicio sin destino	Falta destino para tipo “pasajeros”	Error controlado y rollback completo	PASS

POST Solicitar Servicio
http://localhost:8080/servicios/solicitar

PASS Servicio creado y viaje iniciado

POST Solicitar Servicio sin tarjeta
http://localhost:8080/servicios/solicitar

PASS Falla esperada porque cliente no tiene tarjeta para pago

POST Solicitar Servicio sin dispinibilidad
http://localhost:8080/servicios/solicitar

PASS Falla esperada porque no hay disponibilidad

POST Solicitar servicio sin destino
http://localhost:8080/servicios/solicitar

PASS Falla esperada porque no hay disponibilidad

3. **(20%) Implementación del RFC1 en diferentes niveles de aislamiento:** Implemente ahora una versión adicional del RFC1, como una transacción en Java Spring, usando dos niveles de aislamiento diferente: read_commited y serializable. Añada a la transacción un temporizador de 30 segundos para poder observar la interacción con otras consultas para los escenarios de prueba. Para propósitos de los escenarios de concunrrencia en nivel de aislamiento serializable (punto 4) y en nivel de aislamiento read commited (punto 5), asegúrese de incluir dos veces la sentencia que realiza la consulta de todos los viajes históricas, una antes del temporizador y otra después.

3:
Implementamos una versión adicional del RFC1 como transacción en Spring que se puede ejecutar en dos modos de aislamiento: READ_COMMITTED y SERIALIZABLE. Para ambos casos expusimos un endpoint (/rfc1/read-committed/{idUsuario} y /rfc1/serializable/{idUsuario}) que realiza dos lecturas de la misma consulta histórica del usuario, una antes y otra después de una pausa de 30 segundos. Ese temporizador nos permite provocar y observar la interacción con otras operaciones concurrentes (en particular, cuando durante esa ventana se ejecuta el RF8 (solicitar servicio)).

En el código implementamos una transacción en Spring Boot que ejecuta el RFC1 directamente desde el backend. Para esto creamos un repositorio (RFCrepository) con una consulta SQL nativa que obtiene la información de los servicios asociados a un cliente, incluyendo datos de la ciudad de origen, el conductor y el viaje. Luego construimos un controlador REST (RfcController) que expone dos endpoints: uno con el nivel de aislamiento READ_COMMITTED y otro con SERIALIZABLE. En cada método, la transacción ejecuta la

misma consulta dos veces: una antes y otra después de una pausa de 30 segundos, lo que permite observar si aparecen o no nuevos servicios creados durante ese intervalo. La lógica convierte los resultados en un formato JSON legible, mostrando los datos “antes” y “después”, junto con el tipo de aislamiento utilizado.

4. **(20%) Implementación del escenario de concurrencia en nivel de aislamiento serializable:** se evalúa el escenario de prueba en el cual se ejecuta primero el RFC1 en nivel de aislamiento serializable, y antes de que pasen los 30 segundos de ejecución de esta consulta, el usuario ejecuta de manera concurrente el componente que implementa RF8. Este escenario de prueba debe contener:
    - Los pasos para la ejecución concurrente de RFC1 y RF8 a través de la línea de tiempo.
    - Descripción de lo sucedido: ¿acaso el componente que implementa RFC1 debió esperar a que terminara la ejecución de la consulta RF8 para poder registrar la orden de servicio?
    - El resultado presentado por RFC1: presente el resultado de esta consulta. Diga si allí apareció la orden de servicio ingresada al ejecutar el componente que implementa RF8 de manera simultánea.

4: Se ejecutaron todas las pruebas de postman

a.

Time	Instruccion	Observación
t0	{{baseUrl}}/rfc1/serializable/1, se ejecuta la primera lectura de antes	Se pone serializable
t1	{{baseUrl}}/servicios/solicitar	Se solicita el servicio mientras se esperan los 30 segundos
t2	Se ejecuta la lectura de despues	Se obtienen los resultados

b.

Descripción de lo sucedido:

En **t0**, FRC1 se abre la sesión serializable y se realiza la lectura de “antes”.
En **t1**, se ejecuta POST /servicios/solicitar para el mismo cliente y se confirma.
En **t2**, FRC1 hace la lectura de “despues”, en la que se devuelve lo mismo de la lectura de “antes”, no ve la orden creada mientras se esperaban los 30 segundos.

Esto ocurre porque en serializable se usa una lectura consistente, esto significa que aunque las transacciones estén en paralelo, se ejecutan como si fueran una después de la otra. Las lecturas no bloquean las escrituras por lo que no cambia la lectura. Si se tiene que esperar a que se complete la transacción de FRC 1 para que quede registrada la de FRC 8. En el antes y después, se tiene un snapshot que es el mismo por lo que el resultado de las dos es el mismo.

c. **Resultados**


```
{
    "antes": [
        {
            "ID_SERVICIO": 2,
            "TIPO_SERVICIO": "pasajeros",
            "ESTADO": "solicitado",
            "CIUDAD_ORIGEN": "Bogotá",
            "HORA_INICIO": null,
            "HORA_FIN": null,
            "DISTANCIA_KM": null,
            "COSTO_TOTAL": null,
            "CONDUCTOR": null
        },
        {
            "ID_SERVICIO": 1,
            "TIPO_SERVICIO": "pasajeros",
            "ESTADO": "solicitado",
            "CIUDAD_ORIGEN": "Bogotá",
            "HORA_INICIO": "08:00",
            "HORA_FIN": "08:45",
            "DISTANCIA_KM": 12.4,
            "COSTO_TOTAL": 31000,
            "CONDUCTOR": "Juan Rojas"
        }
    ],
```

```
"despues": [
    {
        "ID_SERVICIO": 2,
        "TIPO_SERVICIO": "pasajeros",
        "ESTADO": "solicitado",
        "CIUDAD_ORIGEN": "Bogotá",
        "HORA_INICIO": null,
        "HORA_FIN": null,
        "DISTANCIA_KM": null,
        "COSTO_TOTAL": null,
        "CONDUCTOR": null
    },
    {
        "ID_SERVICIO": 1,
        "TIPO_SERVICIO": "pasajeros",
        "ESTADO": "solicitado",
        "CIUDAD_ORIGEN": "Bogotá",
        "HORA_INICIO": "08:00",
        "HORA_FIN": "08:45",
        "DISTANCIA_KM": 12.4,
        "COSTO_TOTAL": 31000,
        "CONDUCTOR": "Juan Rojas"
    }
],
"aislamiento": "SERIALIZABLE"
```


Antes habían dos servicios y despues tambien habian dos.
No apareció la orden que fue creada por RF8.
Podemos ver que al ser serializable, no se vio lo que se hizo en la otra transacción ya que se toman como si fueran consecutivas,

5. **(20%) Implementación del escenario de concurrencia en nivel de aislamiento read commited:** se evalúa el escenario de prueba en el cual se ejecuta primero el RFC1 en nivel de aislamiento serializable, y antes de que pasen los 30 segundos de ejecución de esta consulta, el usuario ejecuta de manera concurrente el componente que implementa RF8. Este escenario de prueba debe contener:
    - Los pasos para la ejecución concurrente de RFC1 y RF8 a través de la línea de tiempo
    - Descripción de lo sucedido: ¿acaso el componente que implementa RFC1 debió esperar a que terminara la ejecución de la consulta RF8 para poder registrar la orden de servicio?
    - El resultado presentado por RFC1: presente el resultado de esta consulta. Diga si allí apareció la orden de servicio ingresada al ejecutar el componente que implementa RF8 de manera simultánea.

5:

a.

Time	**Instruccion**	**Observación**
t0	GET /rfc1/read-committed/{clientId}, se ejecuta la primera lectura de antes	Se pone read commited
t1	{{baseUrl}}/servicios/solicitar	Se solicita el servicio mientras se esperan los 30 segundos y se hace commit
t2	Se ejecuta la lectura de despues	Se obtienen los resultados

b. No, en read commited RFC1 lee los datos que ya fueron confirmados en el momento que se hace la lectura. En este caso, se hizo la primera lectura, con 3 filas, mientras se esperaba RFC 8 inserto otro servicio y después se hizo la última lectura después de los 30 segundos y se obtuvo la nueva lectura que se había insertado. No hubo bloqueos entre las transacciones.

c.


```
{
    "antes": [
        {
            "ID_SERVICIO": 3,
            "TIPO_SERVICIO": "pasajeros",
            "ESTADO": "asignado",
            "CIUDAD_ORIGEN": "Bogotá",
            "HORA_INICIO": "10:30",
            "HORA_FIN": "10:30",
            "DISTANCIA_KM": 5,
            "COSTO_TOTAL": 20000,
            "CONDUCTOR": "Juan Rojas"
        },
        {
            "ID_SERVICIO": 2,
            "TIPO_SERVICIO": "pasajeros",
            "ESTADO": "solicitado",
            "CIUDAD_ORIGEN": "Bogotá",
            "HORA_INICIO": null,
            "HORA_FIN": null,
            "DISTANCIA_KM": null,
            "COSTO_TOTAL": null,
            "CONDUCTOR": null
        },
        {
            "ID_SERVICIO": 1,
            "TIPO_SERVICIO": "pasajeros",
            "ESTADO": "solicitado",
            "CIUDAD_ORIGEN": "Bogotá",
            "HORA_INICIO": "08:00",
            "HORA_FIN": "08:45",
            "DISTANCIA_KM": 12.4,
            "COSTO_TOTAL": 31000,
            "CONDUCTOR": "Juan Rojas"
        }
    ],
```

```
],
"despues": [
    {
        "ID_SERVICIO": 21,
        "TIPO_SERVICIO": "pasajeros",
        "ESTADO": "asignado",
        "CIUDAD_ORIGEN": "Bogotá",
        "HORA_INICIO": "10:30",
        "HORA_FIN": "10:30",
        "DISTANCIA_KM": 5,
        "COSTO_TOTAL": 20000,
        "CONDUCTOR": "Juan Rojas"
    },
    {
        "ID_SERVICIO": 3,
        "TIPO_SERVICIO": "pasajeros",
        "ESTADO": "asignado",
        "CIUDAD_ORIGEN": "Bogotá",
        "HORA_INICIO": "10:30",
        "HORA_FIN": "10:30",
        "DISTANCIA_KM": 5,
        "COSTO_TOTAL": 20000,
        "CONDUCTOR": "Juan Rojas"
    },
    {
```

```
            "CONDUCTOR": "Juan Rojas"
        },
        {
            "ID_SERVICIO": 2,
            "TIPO_SERVICIO": "pasajeros",
            "ESTADO": "solicitado",
            "CIUDAD_ORIGEN": "Bogotá",
            "HORA_INICIO": null,
            "HORA_FIN": null,
            "DISTANCIA_KM": null,
            "COSTO_TOTAL": null,
            "CONDUCTOR": null
        },
        {
            "ID_SERVICIO": 1,
            "TIPO_SERVICIO": "pasajeros",
            "ESTADO": "solicitado",
            "CIUDAD_ORIGEN": "Bogotá",
            "HORA_INICIO": "08:00",
            "HORA_FIN": "08:45",
            "DISTANCIA_KM": 12.4,
            "COSTO_TOTAL": 31000,
            "CONDUCTOR": "Juan Rojas"
        }
    ],
    "aislamiento": "READ_COMMITTED"
}
```

Antes había 3 servicios, después había 4 servicios. Apareció la orden que fue creada por RF 8 en la segunda lectura. Se puede ver que en el antes solamente hay 3 servicios mientras que en él después hay 4 lecturas. En read commited se ven los cambios que fueron confirmados.