Informe Proyecto Entrega 2

Cambios realizados por cada RF y RFC

RF1: REGISTRAR UNA CIUDAD

Viejo: CityController era @RestController con endpoint /api/cities/register que usaba CityService para cityRepository.save(city) (API JSON).
Nuevo: CityController es @Controller MVC con rutas /cities, /cities/new y POST cities/new/save que devuelven vistas Thymeleaf (cities, cityNew) y usa cityRepository directamente (inyección por constructor). Se añadió funcionalidad de listar y eliminar (eliminarCiudad).
Cómo funciona ahora: el usuario administra ciudades desde UI: formulario nuevo → submit → cityRepository.save(). Se muestra lista de ciudades en /cities.

RF2: REGISTRAR UN USUARIO DE SERVICIOS

Viejo: UsuarioController REST con registerServiceUser que guardaba Usuario y creaba UsuarioCliente (mapsId).
Nuevo: UsuarioController MVC con /users, /users/new y /users/new/save que maneja la creación desde formularios y guarda Usuario vía usuarioRepository.save(usuario).
Cómo funciona ahora: UI simple para crear usuarios de servicio; persistencia estándar JPA en tabla Usuario y (según modelo) entrada en UsuarioCliente si aplica.

RF3: REGISTRAR UN USUARIO CONDUCTOR

Viejo: UsuarioService.registerConductorUser creaba Usuario y UsuarioConductor.
Nuevo: en la UI se registra conductor via /users/new (mismo endpoint general). Internamente existen UsuarioConductor y repositorios; controller guarda Usuario.
Cómo funciona ahora: formulario → usuarioRepository.save(usuario); si se requiere, se crea fila en subtabla conductor (lógica de negocio en servicio o en repositorios).

RF4: REGISTRAR UN VEHÍCULO PARA UN USUARIO CONDUCTOR

Viejo: UsuarioService.registerVehicleForConductor revisaba existencia del conductor y guardaba Vehiculo.
Nuevo: VehiculoController con /vehicles/new y POST /new/save que valida conductor (por cedula), asigna vehiculo.setConductor(...), opcional ciudad de expedición y guarda con vehiculoRepository.save(vehiculo).
Cómo funciona ahora: UI para crear vehículos; valida conductor y enlaza entidad.

RF5: REGISTRAR LA DISPONIBILIDAD (y RF6 modificar)

Viejo: DisponibilidadRepository tenía métodos existsByVehiculoAndFechaAndHoraInicioLessThanAndHoraFinGreaterThan(...) y ...AndIdDisponibilidadNot(...) y UsuarioService validaba solapamiento y guardaba. Disponibilidades se guardaban y podían solaparse prevenido por la lógica en Java.
Nuevo: DisponibilidadController con formularios (/disponibilidades/new, /disponibilidades/{id}/edit) valida solapamiento localmente buscando disponibilidades por

conductor (findByConductorCedula) o usando query findOverlapping(...). Además, se introdujo el campo ASIGNADA para reservar. Al crear/editar se verifica solapamientos y se retorna error en UI si existe conflicto.
Cómo funciona ahora: el conductor crea disponibilidad; el backend verifica solapamientos entre sus disponibilidades y evita superposición; al asignar un servicio el registro ya no se borra, sino que se marca ASIGNADA = 1.

RF6: MODIFICAR DISPONIBILIDAD
Viejo: UsuarioService.modifyDisponibilidad validaba solapamientos excluyendo la propia entrada por id.
Nuevo: DisponibilidadController implementa el formulario de edición, valida horas y solapamientos con disponibilidadRepository.findOverlapping, actualiza y guarda.
Cómo funciona ahora: edición por UI con validación, evita solapamiento con demás disponibilidades del mismo conductor.

RF7: REGISTRAR UN PUNTO GEOGRÁFICO
Viejo: Existía PuntoGeografico model y endpoints mínimos.
Nuevo: PuntoGeograficoController con /puntos, /puntos/new y POST /new/save que valida ciudad y guarda punto con puntoGeograficoRepository.save(punto). Se muestran listas y formularios en UI.
Cómo funciona ahora: cualquier usuario añade puntos con nombre/dirección/coordenadas y se asocia a City.

RF8: SOLICITAR UN SERVICIO (detalle en Parte 2)
Viejo: casi sin implementación completa
Nuevo: implementado como transacción atómica @Transactional en ServicioController.guardarServicioTransactional(...) que realiza verificación tarjeta, búsqueda de disponibilidad (query nativa que filtra ASIGNADA=0), marca asignada=1, genera IDs (via nextId()), crea Servicio y ServicioPunto(s) y guarda todo; si algo falla la tx hace rollback y libera la disponibilidad. Ver Parte 2 para explicación técnica completa.

RF9: REGISTRAR EL FINAL DE UN VIAJE
Viejo: no completamente implementado.
Nuevo: POST /servicios/{id}/finish en ServicioController dentro de @Transactional que:

- Busca Servicio por id,
- Actualiza horaFin = now,
- Guarda servicioRepository.save(s),
- Busca la Disponibilidad reservada (por placa, fecha y asignada = 1) y pone asignada = 0 (libera la disponibilidad),
- Redirige a vista de servicio finalizado.
  Cómo funciona ahora: el conductor finaliza → sistema grava hora fin y libera la disponibilidad que había sido marcada al asignar el servicio.

RF10 y RF11: DEJAR REVISIÓN (usuario y conductor)

Viejo: existía la tabla Rating (pero no UI completa). Usuario tenía además calificacion y comentario en su entidad.
Nuevo: RatingRepository con nextId() y métodos findByRevisado_CedulaOrderByFechaDesc, avgRatingForRevisado(...). Templates comentarios.html, comentarioDetail.html se añadieron. Se implementó repositorio y endpoints para listar comentarios por usuario y crear nuevos (detalle en templates y controller de comentarios).
Cómo funciona ahora: se puede listar comentarios de un usuario y añadir una nueva revisión (se genera idComentario con nextId() y se guarda); promedia de ratings calculado por repo.

RFC1: HISTÓRICO DE SERVICIOS DE UN USUARIO
Viejo: SQL en archivo RFC1.sql (consulta con LEFT JOINs).
Nuevo: StatsController.rfc1 usa JdbcTemplate para ejecutar la consulta (con parámetro clienteId) y mostrar rows en rfc1.html. Además se añadió StatsIsolationController que implementa RFC1 como transacción con aislamiento (ver Parte 3 & 4).
Cómo funciona ahora: UI con formulario para seleccionar cliente y ver su historial; existe versión transaccional para pruebas de aislamiento.

RFC2: TOP 20 CONDUCTORES
Viejo: SQL en RFC2.sql.
Nuevo: StatsController.rfc2 ejecuta la consulta via JdbcTemplate y muestra rfc2.html.
Cómo funciona ahora: muestra cedula, nombre y conteo de servicios, ordenado desc, top20.

RFC3: DINERO OBTENIDO POR CONDUCTORES POR VEHÍCULO
Viejo: SQL en RFC3.sql.
Nuevo: StatsController.rfc3 con JdbcTemplate que ejecuta consulta y presenta rfc3.html.
Cómo funciona ahora: aggregated por conductorCedula y placa con suma de valorServicio.

RFC4: UTILIZACIÓN POR CIUDAD EN RANGO DE FECHAS
Viejo: SQL en RFC4.sql.
Nuevo: StatsController.rfc4 con formulario para ciudad, desde y hasta; convierte fechas y ejecuta la consulta (with servicios_ciudad...). Resultado mostrado en rfc4.html.
Cómo funciona ahora: el usuario selecciona ciudad y rango de fechas y obtiene una tabla con tipo, nivelTransporte, cantidad y % del total.

Implementación del RF8

Objetivo del RF8
Solicitar un servicio: verificar medio de pago, buscar conductor disponible, reservarlo, registrar servicio (inicio, costo, conductor/cliente), y garantizar atomicidad (todo o nada).
Dónde está implementado

ServicioController.guardarServicioTransactional(...) (anotado con @Transactional(rollbackFor = Exception.class)).

Pasos que ejecuta la transacción (orden, SQL/JPA equivalente y comentarios)

1. **Recepción de parámetros** (desde formulario): tipo, distancia, tarifa, puntoPartidaId, puntoDestinoId, clienteId, tarjetaNumero.
    - No SQL aún.
2. **Validación de medio de pago**:
    - tarjetaCreditoRepository.findByUsuario_Cedula(clienteId) (JPA query/native), devuelve tarjetas del usuario.
    - Verifica que tarjetaNumero exista y pertenezca al clienteId.
    - Si falla: redirect con error (no commit).
3. **Buscar disponibilidad (conductor disponible)**:
    - Llama disponibilidadRepository.findFirstAvailableForTipoAndFechaAndTime(tipoNormalized, today, now) que ejecuta query nativa:
    - SELECT * FROM Disponibilidad d
    - WHERE UPPER(d.TIPOSERVICIO) = ?1
    - AND d.FECHA = ?2
    - AND ?3 BETWEEN d.HORAINICIO AND d.HORAFIN
    - AND NVL(d.ASIGNADA,0) = 0
    - FETCH FIRST 1 ROWS ONLY
    - Resultado: Optional<Disponibilidad> con Vehiculo y conductor.
4. **Reservar la disponibilidad**:
    - En vez de borrar la fila: disp.setAsignada(1); disponibilidadRepository.save(disp);
    - Esto evita la pérdida de historial y permite liberarla luego.
5. **Crear entidad Servicio**:
    - Generar id manual con servicioRepository.nextId() (consulta nativa SELECT COALESCE(MAX(id),0)+1).
    - Construir Servicio con tipo, distancia, tarifa, horaInicio(now), vehiculo, conductor, cliente.
    - servicioRepository.save(servicio);
6. **Crear puntos de servicio (ServicioPunto)**:
    - Crear dos ServicioPunto (partida orden=1, destino orden=2).
    - Generar idServicioPunto con servicioPuntoRepository.nextId() (igual patrón MAX+1).
    - servicioPuntoRepository.save(sp1) y save(sp2).
7. **Simular cobro / finalizar la transacción**:
    - Mensaje de “cobro simulado”.
    - redirect /servicios con success.
    - Si en cualquier paso ocurre error (p. ej. nextId() devuelve null, falta punto, falta tarjeta), la transacción falla y Spring hace rollback (revirtiendo el disp.setAsignada(1) y cualquier save()).

Notas técnicas importantes

- **Uso de ASIGNADA en Disponibilidad**: evita delete y permite manejo más explícito de reserva y liberación.
- **nextId() (MAX+1)**: funciona pero tiene riesgo de *race conditions* en alta concurrencia; en entornos concurrentes se recomienda secuencia DB (SEQUENCE.NEXTVAL) o bloqueo explícito.
- **Atomicidad**: @Transactional garantiza rollback automático si ocurre una excepción no capturada o se lanza RuntimeException. Aquí se indica rollbackFor = Exception.class para atrapar excepciones chequeadas también.
- **Order of operations**: se marca disponibilidad **antes** de crear el servicio; esto evita asignaciones dobles a dos peticiones concurrentes (si la búsqueda de disponibilidad filtra ASIGNADA=0 y ambas transacciones se ejecutan exactamente al mismo tiempo, puede existir una ventana de race, por eso el uso de transacción y/o bloqueo DB es importante).
- **Persistencia y flush**: se llama servicioPuntoRepository.flush() después del primer save(sp1) en el código final, útil para forzar la ejecución inmediata y detectar errores de insert temprano.

Escenario de concurrencia: nivel de aislamiento SERIALIZABLE

Objetivo del escenario: ejecutar primero RFC1 transaccional en aislamiento SERIALIZABLE (lectura histórica + sleep 30s) y durante su espera ejecutar RF8 (crear servicio). Observar si RFC1 ve la nueva orden y si hubo bloqueo/espera.

Implementación usada para la prueba

- **RFC1 transaccional**: StatsIsolationController.runSerializable(...):
- @Transactional(isolation = Isolation.SERIALIZABLE)
- List<Servicio> before = servicioRepository.findByCliente_CedulaOrderByHoraInicioDesc(clienteId);
- Thread.sleep(30_000);
- List<Servicio> after = servicioRepository.findByCliente_CedulaOrderByHoraInicioDesc(clienteId);
- **RF8**: guardarServicioTransactional(...) en ServicioController (ve Parte 2).

Pasos (línea de tiempo), cómo ejecutar la prueba manualmente

1. **Cliente A (RFC1)**: Abrir navegador A → formulario /stats/rfc1/isolation → seleccionar cliente X → pulsar “serializable” (envía POST a /stats/rfc1/isolation/serializable).
    - La tx A comienza, ejecuta la primera lectura (before) y entra en sleep(30s) con la transacción aún abierta en aislamiento SERIALIZABLE.
2. **Entre t=0s y t=30s**: Mientras A está en sleep, ejecutar **Cliente B (RF8)**: desde otro navegador o curl hacer el POST /servicios/new/save con datos válidos para crear un servicio nuevo para el mismo o distinto cliente (según el escenario). RF8 inicia su propia transacción B, valida tarjeta, busca disponibilidad y, si la encuentra, realiza disp.setAsignada(1), save(disp) y crea el nuevo Servicio y ServicioPunto y commit.
3. **t≈30s**: la tx A despierta y realiza la segunda lectura (after) y luego hace commit/fin.

Qué sucede (descripción)

- **Visibilidad en SERIALIZABLE**: en la mayoría de bases de datos SQL que implementan serializable (ej. Oracle en modo SERIALIZABLE o DBs con snapshot isolation que emulan serializability), una transacción en SERIALIZABLE **ve un snapshot consistente** del momento en que empezó la transacción. Eso implica:
    - La segunda lectura dentro de la misma transacción A **no** verá las filas que fueron insertadas y committeadas por B durante el sleep. Por tanto, after contendrá *los mismos resultados* que before (la nueva orden creada por B no aparece en la vista de A).
- **Bloqueo/espera entre A y B**:
    - RFC1 (A) es *lectora*; RF8 (B) realiza inserts y updates (marca asignada = 1). En este caso **B no tiene que esperar** por A (A sólo leyó), a menos que la base de datos use locks más restrictivos. De forma general:
        - No hay bloqueo: B ejecuta, inserta y commit sin esperar a A.
        - A no bloquea a B porque A no adquirió locks excluyentes para las filas que B modifica.
    - Sin embargo, si A hubiese hecho operaciones que colocaran locks FOR UPDATE o hubiese modificado las mismas filas, B podría verse bloqueado o generar error de serialización al intentar commit.
- **Resultado presentado por RFC1**:
    - before: lista original sin la orden creada por B.
    - after: **igual** que before (no incluye la orden creada concurrentemente por B).
    - Conclusión: en SERIALIZABLE, RFC1 **no** vio la orden creada por RF8 durante su ejecución; tampoco se exigió que B esperara a que A terminase.

Posibles efectos secundarios

- En algunos RDBMS el commit de B podría fallar si se detecta que la combinación de operaciones violaría la serializabilidad (lanzando error que obliga a reintento). En nuestro caso, como A solo lee y B inserta, típicamente no ocurre error de serialización.
- Si nextId() está implementado como MAX+1, en alta concurrencia puede haber colisiones; la transacción B que hace nextId() podría obtener el mismo valor si dos transacciones calculan MAX+1 simultáneamente y luego intentan insert, causando PK violation en commit. Ese riesgo es independiente del aislamiento elegido.

Escenario de concurrencia: nivel de aislamiento READ COMMITTED

Objetivo: igual que el anterior, pero RFC1 corre con aislamiento READ_COMMITTED. Se observa si RFC1 ve la orden creada por RF8 en la segunda lectura.

Implementación usada

- StatsIsolationController.runReadCommitted(...) anotado:
- @Transactional(isolation = Isolation.READ_COMMITTED)
- List<Servicio> before = servicioRepository.findByCliente_CedulaOrderByHoraInicioDesc(clienteId);

- Thread.sleep(30_000);
- List<Servicio> after = servicioRepository.findByCliente_CedulaOrderByHoraInicioDesc(clienteId);
- RF8 igual que en Parte 2.

Pasos (línea de tiempo): cómo ejecutar la prueba manualmente

1. **Cliente A (RFC1)**: iniciar POST /stats/rfc1/isolation/read_committed → tx A ejecuta primera lectura y sleep(30s).
2. **Cliente B (RF8)**: durante el sleep de A, ejecutar POST /servicios/new/save creando y commiteando un nuevo servicio.
3. **t≈30s**: tx A hace la segunda lectura y luego finaliza.

Qué sucede (descripción)

- **Visibilidad en READ COMMITTED**:
  - En READ_COMMITTED cada sentencia ve el estado committed al momento en que se ejecuta esa sentencia.
  - Por tanto, la **segunda lectura** en A, que ocurre después de que B haya hecho commit, **sí verá** la nueva fila insertada por B.
- **Bloqueo/espera**:
  - Igual que en SERIALIZABLE: A no bloquea a B por ser lectora; B no debería esperar salvo que intente modificar exactamente las mismas filas que están bloqueadas por A (no aplica aquí).
- **Resultado presentado por RFC1**:
  - before: listado antes de la inserción de B (sin la nueva orden).
  - after: listado **incluye** la nueva orden creada por RF8 (porque la sentencia se ejecutó tras el commit de B).
  - Conclusión: en READ_COMMITTED RFC1 **sí** puede ver el resultado de transacciones concurrentes si esas transacciones committearon antes de la segunda sentencia de lectura dentro de RFC1.

Implicación práctica

- READ_COMMITTED muestra datos *actualizados* por otras transacciones entre sentencias; SERIALIZABLE no lo hace (snapshots consistentes). Para tus pruebas de concurrencia y explicación en el informe, estos son los comportamientos a documentar y demostrar con capturas/outputs.

Conclusiones y recomendaciones

Conclusiones resumen

1. **Funcionalidad**: la aplicación avanzó desde una API parcial a una aplicación MVC transaccional completa con implementación de RF1..RF11 y RFC1..RFC4. RF8 y RF9 ahora son transaccionales y manejan reserva/liberación de disponibilidades correctamente mediante ASIGNADA (cambio de diseño robusto frente a delete).
2. **Estadísticas y aislamiento**: se implementó RFC1 en dos modos de aislamiento (SERIALIZABLE y READ_COMMITTED) con Thread.sleep(30s) para permitir experimentación de concurrencia. Esto permite demostrar diferencias visibles en before y after de la consulta.

3. **Consistencia y atomicidad**: @Transactional en RF8 garantiza rollback en caso de fallo; sin embargo, el uso de nextId() (MAX+1) es un punto débil potencial para concurrencia alta.
4. **Interfaz y usabilidad**: se añadieron vistas (Thymeleaf) para gestionar ciudades, usuarios, vehículos, disponibilidades, servicios, estadísticas y comentarios, lo que facilita pruebas manuales.