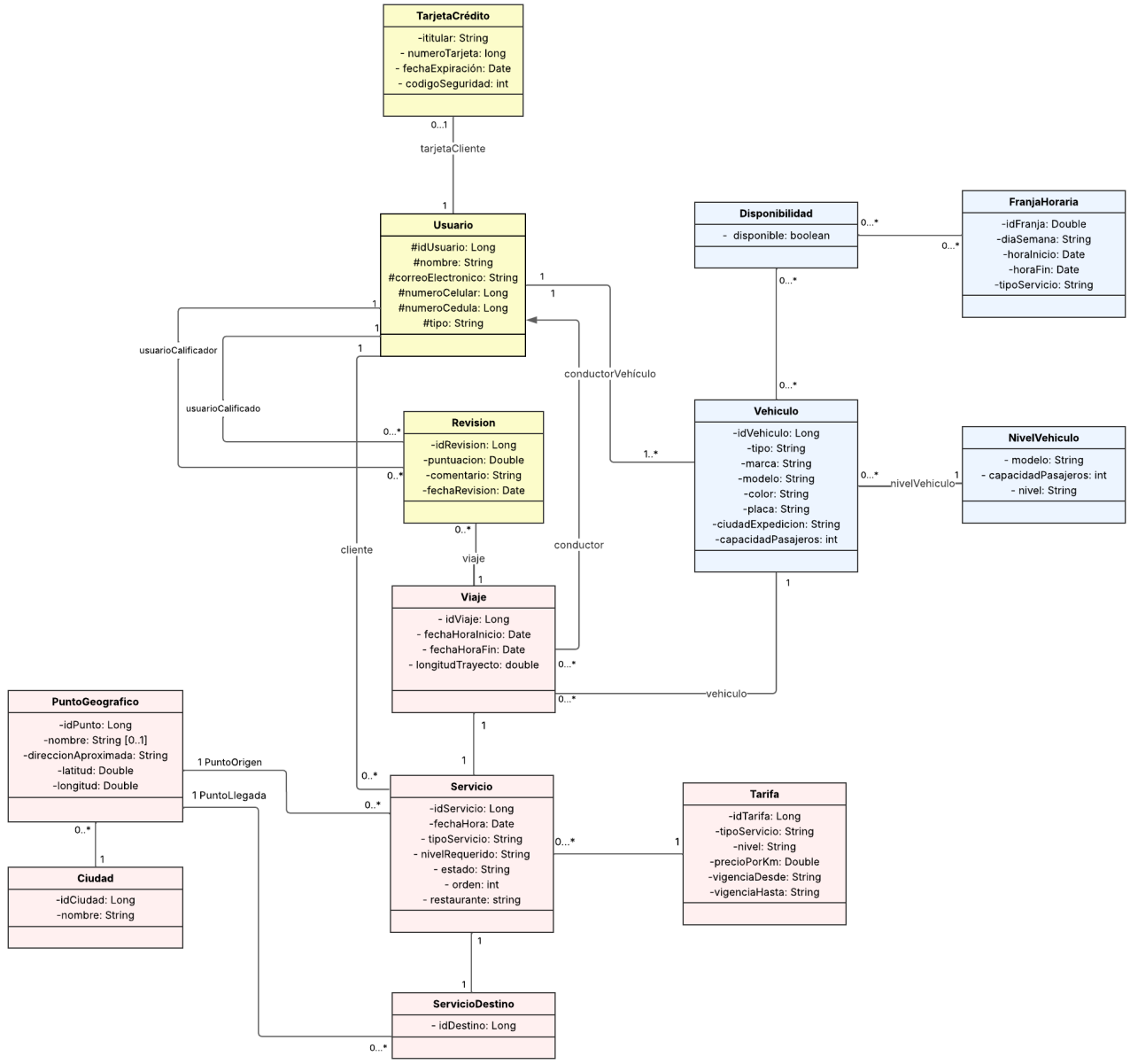
Entrega 3 AlpesCab

Sergio Arias | 202412370
Sara González | 202210908
Laura Martinez-Galindo | 202125643

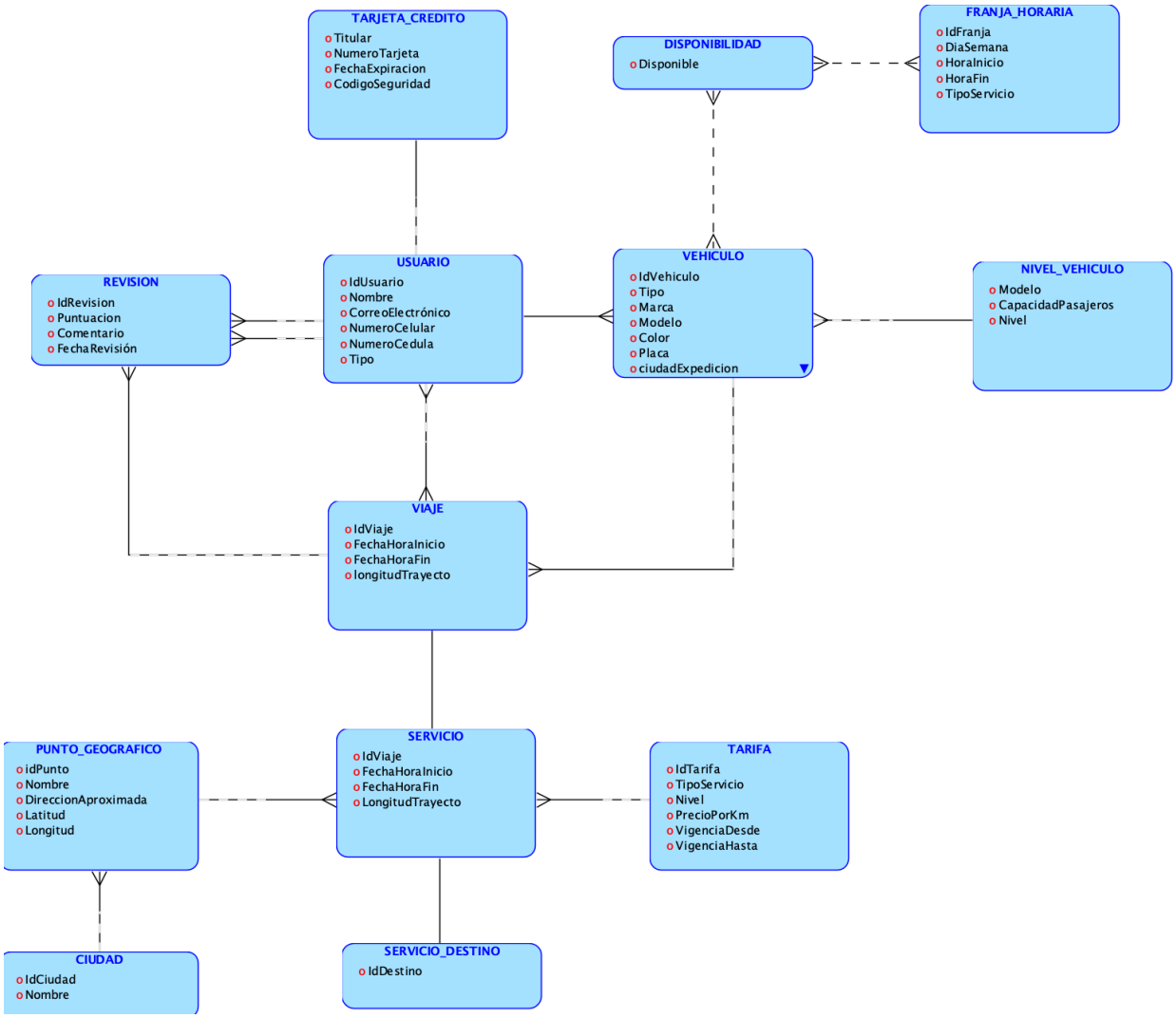
1. Diseño Base de Datos

a. Modelo Conceptual UML



[Modelo en LucidChart para mejor visualización]

a. Modelo Conceptual E/R



b. Entidades y Relaciones

En el modelo conceptual propuesto, la entidad Usuario representa a cualquier persona registrada en la plataforma. A partir de ella se distinguen dos roles: el UsuarioServicio, que es quien solicita los servicios y puede registrar medios de pago, y el UsuarioConductor, que es quien presta los servicios de transporte y registra sus vehículos. De manera coherente con esto, la entidad TarjetaCredito modela los medios de pago asociados a los usuarios de servicio, mientras que la entidad Vehiculo corresponde a los vehículos registrados por los conductores, incluyendo atributos como tipo, marca, modelo, color, placa, ciudad de expedición y capacidad, además de un nivel asignado (Estándar, Confort o Large). Por su parte, la entidad Disponibilidad describe la disponibilidad temporal de un vehículo para prestar servicios y se descompone en varias franjas horarias representadas por la entidad FranjaHoraria, que detalla días y rangos de atención. En complemento, la entidad Tarifa describe el precio fijo por kilómetro calculado según la combinación de tipo de servicio y nivel del vehículo.

En cuanto a la operación de AlpesCab, la entidad Servicio representa la solicitud inicial realizada por un usuario de servicios, incluyendo el punto de origen, los puntos de destino y la tarifa asociada. A partir de esta solicitud, la entidad Viaje registra el servicio efectivamente prestado, incorporando información como la hora de inicio, la hora de fin, la longitud recorrida, el costo final, el conductor asignado y el vehículo utilizado. Para capturar la información espacial, la entidad PuntoGeografico representa los puntos utilizados en los servicios, incluyendo dirección aproximada, latitud, longitud y

la Ciudad, que se modela como una entidad adicional. Finalmente, la entidad Review representa la calificación y comentario que un usuario deja sobre otro después de un servicio.
Las relaciones entre entidades y su cardinalidad se presentan en la siguiente tabla:

Relación	Cardinalidad	Descripción
UsuarioServicio – TarjetaCredito	Uno a muchos	Un UsuarioServicio puede registrar varias TarjetaCredito; cada TarjetaCredito pertenece a un único UsuarioServicio
UsuarioConductor – Vehiculo	Uno a muchos	Un UsuarioConductor puede registrar varios Vehiculo; cada Vehiculo está asociado exactamente a un UsuarioConductor
NivelVehiculo – Vehiculo	Uno a muchos	Un NivelVehiculo (Estándar, Confort, Large) puede clasific ar muchos Vehiculo; cada Vehiculo tiene exactamente un NivelVehiculo
Vehiculo – Disponibilidad	Uno a muchos	Un Vehiculo puede tener registradas varias Disponibilidad a lo largo del tiempo; cada Disponibilidad corresponde a un único Vehiculo
Disponibilidad – FranjaHoraria	Uno a muchos	Cada Disponibilidad se descompone en varias FranjaHoraria (día y rango de horas); cada FranjaHoraria pertenece a una sola Disponibilidad
Vehiculo – Viaje	Uno a muchos	Un Vehiculo puede ser utilizado en muchos Viaje; cada Viaje se realiza usando exactamente un Vehiculo
Viaje – Review	Uno a muchos	Un Viaje puede generar varias Review asociadas (por ejemplo, del usuario hacia el conductor y viceversa); cada Review está ligada a un único Viaje
UsuarioServicio – Review	Uno a muchos	Un UsuarioServicio puede escribir muchas Review y también puede recibir muchas
UsuarioConductor – Review	Uno a muchos	Un UsuarioConductor puede recibir muchas Review y, cuando corresponde, también puede escribirlas
Servicio – Viaje	Uno a uno (0 o 1)	Cada Servicio solicitado puede generar a lo sumo un Viaje cuando se presta; cada Viaje proviene de un único Servicio
Tarifa – Servicio	Uno a muchos	Una Tarifa (definida por tipo de servicio y nivel) puede aplicarse a muchos Servicio; cada Servicio se calcula usando exactamente una Tarifa vigente
PuntoGeografico – Servicio	Uno a muchos	Un mismo PuntoGeografico puede ser utilizado como origen en muchos Servicio; cada Servicio tiene un Punto de origen
Servicio – ServicioDestino	Uno a muchos	Un Servicio tiene uno o varios ServicioDestino que representan sus destinos intermedios o finales; cada ServicioDestino pertenece a un único Servicio

PuntoGeografico – ServicioDestino	Uno a muchos	Un PuntoGeografico puede ser usado como destino en muchos ServicioDestino; cada ServicioDestino se asocia a un único PuntoGeografico
Ciudad – PuntoGeografico	Uno a muchos	Una Ciudad contiene muchos PuntoGeografico dentro de su área; cada PuntoGeografico está localizado en una única Ciudad

c. Análisis de Selección Esquema

En el paso del modelo conceptual al modelo documental, partimos de todas las entidades originales: Usuario, UsuarioServicio, UsuarioConductor, TarjetaCredito, Vehiculo, NivelVehiculo, Disponibilidad, FranjaHoraria, Tarifa, Servicio, ServicioDestino, Viaje, PuntoGeografico, Ciudad y Review. Luego, en el modelo NoSQL propuesto, todas las entidades que representan "actores" o "hechos" principales del negocio, y que además aparecen en los requerimientos RF y RFC, se mantienen como colecciones: USUARIOS, VEHICULOS, DISPONIBILIDADES, TARIFAS y VIAJES. Otras pasan a ser subdocumentos embebidos (por ejemplo TarjetaCredito, FranjaHoraria, ServicioDestino, PuntoGeografico y Review) y algunas se simplifican a atributos categóricos (como NivelVehiculo y Ciudad).

USUARIOS se mantiene como colección porque la aplicación tiene alrededor de 1.200.000 usuarios activos, con crecimiento anual del 15 %, y son la base de los RF1, RF2 y RF6. Dentro de esta colección se unifican UsuarioServicio y UsuarioConductor usando un atributo tipoUsuario, ya que desde el punto de vista de almacenamiento no justifica tener dos colecciones casi iguales. Sobre ese mismo documento se embeben las TarjetaCredito como un arreglo, porque las tarjetas sólo se usan para cobrarle a ese usuario en RF6 y, según la carga de trabajo, los usuarios sólo actualizan el medio de pago aproximadamente una vez al año. Es decir, tenemos muchísimos usuarios, pero con pocas tarjetas cada uno y con muy pocas actualizaciones; en ese escenario, una colección aparte de tarjetas solo introduciría más consultas y más joins, sin aportar nada relevante a RFC1, RFC2 o RFC3. Embebidas en USUARIOS se leen directo cuando se va a cobrar y no hacen ruido el resto del tiempo.

Para VEHICULOS, la razón de tenerlos como colección independiente es que hay alrededor de 100.000 conductores, con un promedio de 1,1 coches por conductor, así que hablamos de unas 110.000 entradas de vehículo. Estos vehículos intervienen en RF3 (registro), RF4 y RF5 (configuración y modificación de disponibilidad) y en RF7 (registro de viajes), y además son necesarios para las consultas RFC2 y RFC3 cuando se quiere saber cuántos servicios prestó cada conductor y cuánto se ganó por vehículo. Si los vehículos estuvieran embebidos directamente dentro del documento del conductor, cada vez que se los relaciona con disponibilidades o viajes habría que cargar documentos grandes de usuario, recorrer arreglos internos y actualizar estructuras embebidas mucho más pesadas. Al mantener VEHICULOS como colección normalizada y referenciar al conductor con un idConductor simple, se pueden consultar y actualizar vehículos de manera más directa, y sobre todo se pueden hacer RFC2 y RFC3 con filtros por vehículo sin pasar por los documentos de usuario. En cambio, NivelVehiculo, que en el modelo relacional podía ser otra tabla, aquí se simplifica a un atributo categórico dentro del vehículo (nivel = "Estandar", "Confort", "Large"), porque el conjunto de niveles es muy pequeño y estable, y no hay requerimientos que pidan consultar "la entidad nivel" de manera independiente; basta con poder filtrar vehículos o tarifas por ese valor.

Por otro lado, la parte de disponibilidad está fuertemente condicionada por la carga de trabajo. Cada conductor actualiza sus disponibilidades aproximadamente una vez al mes, mientras que la aplicación hace búsquedas de conductores disponibles una vez por minuto durante unos cinco minutos en cada sesión de búsqueda. Es decir, se escriben disponibilidades muy poco, pero se leen muchísimas veces para RF4, RF5 y especialmente para RF6, cuando hay que asignar un servicio a un conductor que esté libre, en una ciudad y franja horaria determinadas. Con unos 100.000 conductores, 70 % activos durante el día, 20 % en la noche y 10 % en ambas jornadas, más o menos la mitad tiene disponibilidades

relevantes para cualquier búsqueda. Por eso, la entidad Disponibilidad se convierte en una colección propia DISPONIBILIDADES, con referencias a idVehiculo e idConductor, de forma que se puedan indexar directamente ciudad, tipos de servicio y campos de franjas horarias sin tener que entrar al documento de vehículo. Dentro de cada disponibilidad, las FranjaHoraria sí se embeben, porque no existe ningún requerimiento que pida tratarlas individualmente: siempre se definen, validan y modifican como un conjunto asociado a una disponibilidad concreta. Así, la relación Vehiculo–Disponibilidad se vuelve normalizada (referenciada) y la relación Disponibilidad–FranjaHoraria se vuelve embebida.

En cuanto a la parte transaccional, la entidad Servicio (la solicitud) y la entidad Viaje (el servicio realmente prestado) se colapsan en una sola colección VIAJES. Esta decisión está muy alineada con RF6 y RF7: primero se solicita un servicio (con un punto de origen y destinos, nivel requerido, tipo de servicio, tarifa calculada) y luego, si se presta, se registra el viaje con hora de inicio, hora de finalización, longitud y costo. En lugar de hacer una colección SERVICIOS y otra VIAJES con una relación uno a uno entre ellas, se agrupan todos estos datos en un sólo documento por viaje/servicio. Esto simplifica muchísimo RFC1, que pide mostrar el histórico de todos los servicios de un usuario "con toda su información relevante", porque basta con consultar VIAJES filtrando por idUsuarioServicio y se obtiene de una sola vez tanto la parte de solicitud como la parte de ejecución. Además, según la carga de trabajo, en ciudades grandes los conductores realizan entre 7 y 8 viajes diarios, y en ciudades intermedias 6, con alrededor del 70 % de los conductores activos en el día. Eso significa que se están creando cientos de miles de documentos VIAJES a lo largo del tiempo y que cada viaje sufre al menos dos escrituras (una al crearlo, otra al marcarlo como terminado), e incluso un tercio de ellos puede sufrir modificaciones adicionales por cambios de ruta o de puntos de llegada. Con ese volumen de escrituras, es mejor que los viajes sean una colección separada y muy optimizada para inserts/updates, y que ni usuarios ni vehículos crezcan demasiado guardando grandes arrays embebidos de viajes.

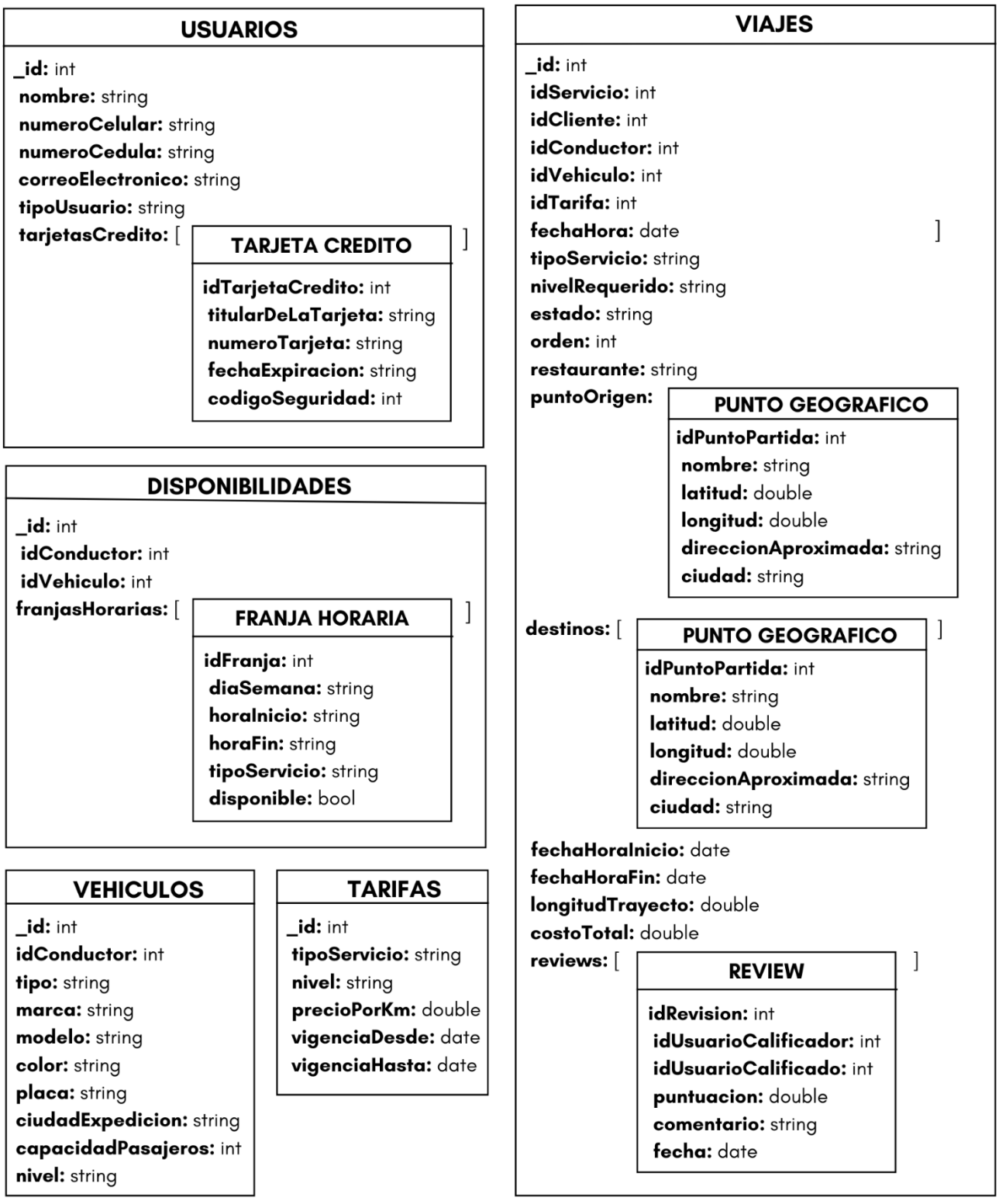
La relación entre vehículo y viaje se mantiene normalizada: VIAJES guarda idVehiculo, y se relaciona con VEHICULOS por referencia. Esta decisión se justifica porque a lo largo del tiempo un solo vehículo puede acumular miles de viajes, dada la frecuencia de viajes por día, embebirlos dentro del documento del vehículo haría que este creciera sin control y fuera pesado de leer y actualizar. Mientras que al normalizar, es fácil implementar RFC3, que pide la utilización de servicios de ALPESCAB en una ciudad durante un rango de fechas, incluyendo descomposición por nivel y tipo de servicio: basta con agrupar por tipo, nivel y ciudad a partir de VIAJES, usando las referencias a vehículo solo cuando se necesite información adicional.

En el caso de las reviews y su relación con viajes y usuarios, el modelo elegido es mixto: la relación Viaje–Review se maneja como embebida (las reviews se guardan dentro del documento de VIAJES), mientras que las relaciones Review–UsuarioServicio y Review–UsuarioConductor se representan como referencias dentro de cada subdocumento review. El motivo es que cada viaje genera como mucho un par de reviews y no hay requerimientos que pidan analizar grandes volúmenes de reviews de forma independiente a los viajes; más bien interesa verlas junto con el viaje en el histórico. Además, embebidas en VIAJES no afectan tanto el tamaño del documento, y no generan un patrón de escritura más complejo que el que ya existe para marcar el viaje como terminado.

Para las relaciones Servicio–PuntoGeografico, Servicio–ServicioDestino y ServicioDestino–PuntoGeografico, la decisión es tratar ServicioDestino y PuntoGeografico como subdocumentos embebidos dentro de VIAJES. El punto de partida y los destinos de un servicio son datos propios de esa solicitud en particular: se definen cada vez que un usuario pide un viaje y pueden cambiar de una carrera a otra e incluso durante la misma carrera. La carga de trabajo indica que alrededor de un tercio de los viajes sufre modificaciones (se añaden nuevos puntos de llegada, se cambia la ruta o se actualiza el precio), de modo que estos puntos se crean y se modifican junto con el viaje y no forman una lista fija de lugares reutilizables que valga la pena mantener en una colección separada.

La relación PuntoGeografico–Ciudad se simplifica a un atributo ciudad dentro de cada punto, en lugar de una colección CIUDADES, porque las operaciones que se realizan sobre ciudades se limitan a filtrar o agrupar por nombre, algo que se puede hacer directamente almacenando la ciudad como cadena.
Finalmente está la relación Servicio–Tarifa. Conceptualmente, muchas instancias de Servicio se asocian a una Tarifa que depende de tipo de servicio y nivel del vehículo. Dado que el número de combinaciones posibles de tipoServicio y nivel es pequeño y la estructura de Tarifa es sencilla, tiene sentido mantener una colección TARIFAS normalizada, donde cada documento representa una combinación tipoServicio–nivel con su precio por kilómetro actualizado. En los documentos de VIAJES se guarda un campo idTarifa que referencia a la tarifa específica usada para ese servicio, y, además, se guarda el costoTotal que realmente se cobró en ese viaje. De esta forma, cuando se registra el viaje (RF7) se toma la tarifa vigente en TARIFAS, se calcula el valor, se almacena el resultado en el propio viaje, y no es necesario duplicar toda la definición de la tarifa dentro del documento. Si en el futuro cambian las tarifas, los nuevos servicios usarán los nuevos valores de TARIFAS, mientras que los viajes históricos seguirán mostrando el costoTotal que se guardó en su momento, sin necesidad de modificar nada. Desde el punto de vista de carga de trabajo, las tarifas cambian muy rara vez, mientras que se crean y actualizan viajes constantemente; por eso es más razonable mantener TARIFAS pequeña y estática, y que los viajes solo apunten a ella con un idTarifa y un costoTotal ya calculado, en vez de replicar la misma información de tarifa en cientos de miles de documentos.
En conjunto, este análisis tiene en cuenta dos cosas: por un lado, qué entidades es útil poder consultar y modificar de manera independiente, y por otro, cómo se espera que se use realmente la aplicación (cantidad de usuarios y conductores, número de vehículos, frecuencia de viajes, frecuencia de búsquedas de disponibilidad y cuántas veces se actualiza cada tipo de dato). Con esa información se decide qué entidades permanecen como colecciones separadas, cuáles se incluyen como subdocumentos embebidos dentro de otra y cuáles se representan simplemente como atributos dentro de los documentos.

d. Modelo Documental


USUARIOS
_id: int
nombre: string
numeroCelular: string
numeroCedula: string
correoElectronico: string
tipoUsuario: string
tarjetasCredito: [
TARJETA CREDITO
idTarjetaCredito: int
titularDeLaTarjeta: string
numeroTarjeta: string
fechaExpiracion: string
codigoSeguridad: int
]
DISPONIBILIDADES
_id: int
idConductor: int
idVehiculo: int
franjasHorarias: [
FRANJA HORARIA
idFranja: int
diaSemana: string
horaInicio: string
horaFin: string
tipoServicio: string
disponible: bool
]
VEHICULOS
_id: int
idConductor: int
tipo: string
marca: string
modelo: string
color: string
placa: string
ciudadExpedicion: string
capacidadPasajeros: int
nivel: string
TARIFAS
_id: int
tipoServicio: string
nivel: string
precioPorKm: double
vigenciaDesde: date
vigenciaHasta: date
VIAJES
_id: int
idServicio: int
idCliente: int
idConductor: int
idVehiculo: int
idTarifa: int
fechaHora: date
tipoServicio: string
nivelRequerido: string
estado: string
orden: int
restaurante: string
puntoOrigen:
PUNTO GEOGRAFICO
idPuntoPartida: int
nombre: string
latitud: double
longitud: double
direccionAproximada: string
ciudad: string
destinos: [
PUNTO GEOGRAFICO
idPuntoPartida: int
nombre: string
latitud: double
longitud: double
direccionAproximada: string
ciudad: string
]
fechaHoraInicio: date
fechaHoraFin: date
longitudTrayecto: double
costoTotal: double
reviews: [
REVIEW
idRevision: int
idUsuarioCalificador: int
idUsuarioCalificado: int
puntuacion: double
comentario: string
fecha: date
]

Ejemplo:

Colección USUARIOS


```
{
  "_id": 1001,
  "nombre": "Usuario A",
  "numeroCelular": "3000000000",
  "numeroCedula": "11111111",
  "correoElectronico": "u@correo.com",
  "tipoUsuario": "Cliente",
  "tarjetasCredito": [
    {
      "idTarjetaCredito": 5001,
      "titularDeLaTarjeta": "Usuario A",
      "numeroTarjeta": "**** 1111",
      "fechaExpiracion": "2027-08",
      "codigoSeguridad": 123
    }
  ]
}
```


Colección USUARIOS


```
{
  "_id": 2001,
  "nombre": "Conductor X",
  "numeroCelular": "3100000000",
  "numeroCedula": "22222222",
  "correoElectronico": "c@correo.com",
  "tipoUsuario": "Conductor",
  "tarjetasCredito": []
}
```


Colección TARIFAS


```
{
  "_id": 6001,
  "tipoServicio": "Transporte Pasajeros",
  "nivel": "Estandar",
  "precioPorKm": 2000.0,
  "vigenciaDesde": "2025-01-01",
  "vigenciaHasta": null
}
```


Colección DISPONIBILIDADES


```
{  "_id": 4001,
  "idConductor": 2001,
  "idVehiculo": 3001,
  "franjasHorarias": [
    { "idFranja": 1,
      "diaSemana": "Lunes",
      "horaInicio": "09:30",
      "horaFin": "11:30",
      "tipoServicio": "Transporte Pasajeros",
      "disponible": true
    },
    { "idFranja": 2,
      "diaSemana": "Lunes",
      "horaInicio": "14:00",
      "horaFin": "19:00",
      "tipoServicio": "Domicilio Comida",
      "disponible": true
    } ]
}
```


Colección VIAJES


```
{
  "_id": 5001,
  "idServicio": 7001,
  "idCliente": 1001,
  "idConductor": 2001,
  "idVehiculo": 3001,
  "idTarifa": 6001,
  "fechaHora": "2025-09-29T09:05:32Z",
  "tipoServicio": "Transporte Pasajeros",
  "nivelRequerido": "Estandar",
  "estado": "Finalizado",
  "orden": null,
  "restaurante": null,
  "puntoOrigen": {
    "idPuntoPartida": 8001,
    "nombre": "Punto Origen 1",
    "latitud": 4.6000,
    "longitud": -74.0800,
    "direccionAproximada": "Dir 1",
    "ciudad": "Ciudad1" },
  "destinos": [
    {
      "nombre": "Punto Destino 1",
      "latitud": 4.6500,
      "longitud": -74.0500,
      "direccionAproximada": "Dir 2",
      "ciudad": "Ciudad1" }
    ],
  "fechaHoraInicio": "2025-09-29T09:15:00Z",
  "fechaHoraFin": "2025-09-29T09:45:00Z",
  "longitudTrayecto": 12.5,
  "costoTotal": 25000.0,
  "reviews": [
  {
    "idRevision": 10001,
    "idUsuarioCalificador": 1001,
    "idUsuarioCalificado": 2001,
    "puntuacion": 5,
    "comentario": "Comentario 1",
    "fecha": "2025-09-29T10:00:00Z"} ] }
```


Colección VEHICULOS


```
{
  "_id": 3001,
  "idConductor": 2001,
  "tipo": "Carro",
  "marca": "Marca1",
  "modelo": "Modelo1",
  "color": "Rojo",
  "placa": "ABC001",
  "ciudadExpedicion": "Ciudad1",
  "capacidadPasajeros": 4,
  "nivel": "Estandar"
}
```


Como se observa en el diagrama documental, se tienen las cinco colecciones principales, cada una con sus respectivos objetos embebidos y referenciados, así como clases que se volvieron atributos dentro de estos (ciudad, nivel). En rojo se indican todas las relaciones de referencia, en azul las embebidas y en verde las clases convertidas en atributos.

2. Implementación Base de Datos

El primer paso es conectarse al cluster definido, el cual está en el connection string: mongodb://ISIS2304A01202520:kuuuULHdgiAZ@157.253.236.88:8087. Una vez allí, se crean las colecciones junto con sus respectivas validaciones ejecutando el script que se encuentra en la ruta "src\db\colecciones.js".

Comando para correrlo en consola:

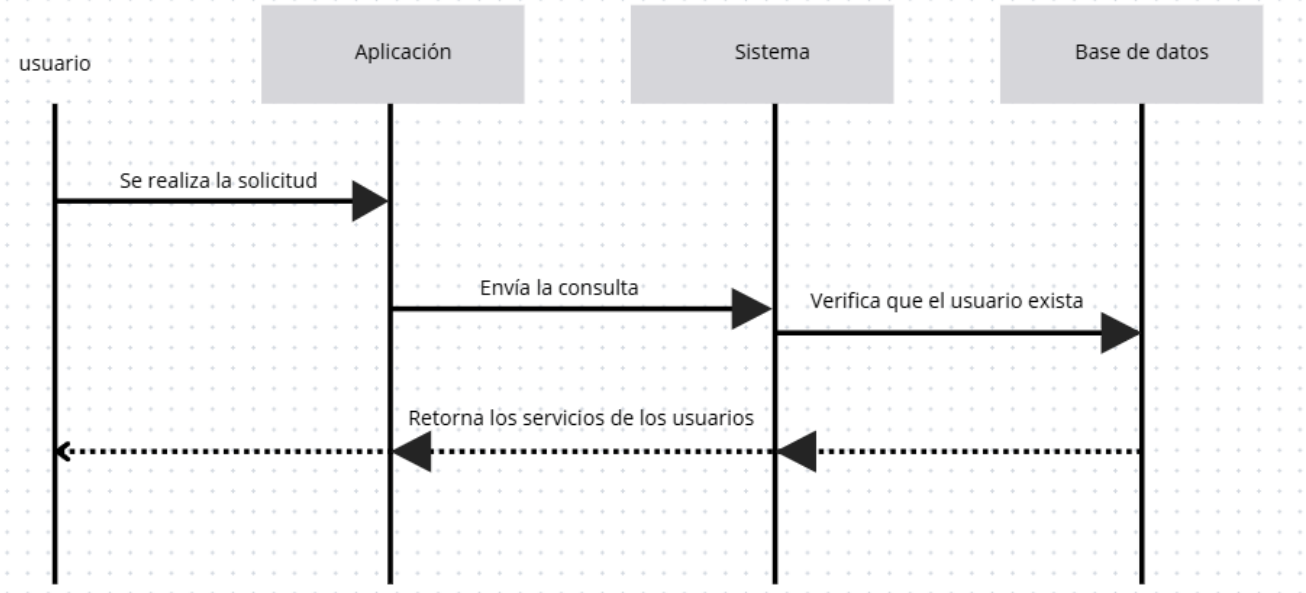
mongosh "mongodb://ISIS2304A01202520:kuuuULHdgiAZ@157.253.236.88:8087/ISIS2304A01202520" --file .\src\db\colecciones.js

También se puede dirigir a MongoDB Compass, conectarsecon el connection string: “mongodb://ISIS2304A01202520:kuuuULHdgiAZ@157.253.236.88:8087/ISIS2304A01202520” y pegar el contenido del script en MongoDB Shell y ejecutarlo.

3. Escenarios de Prueba

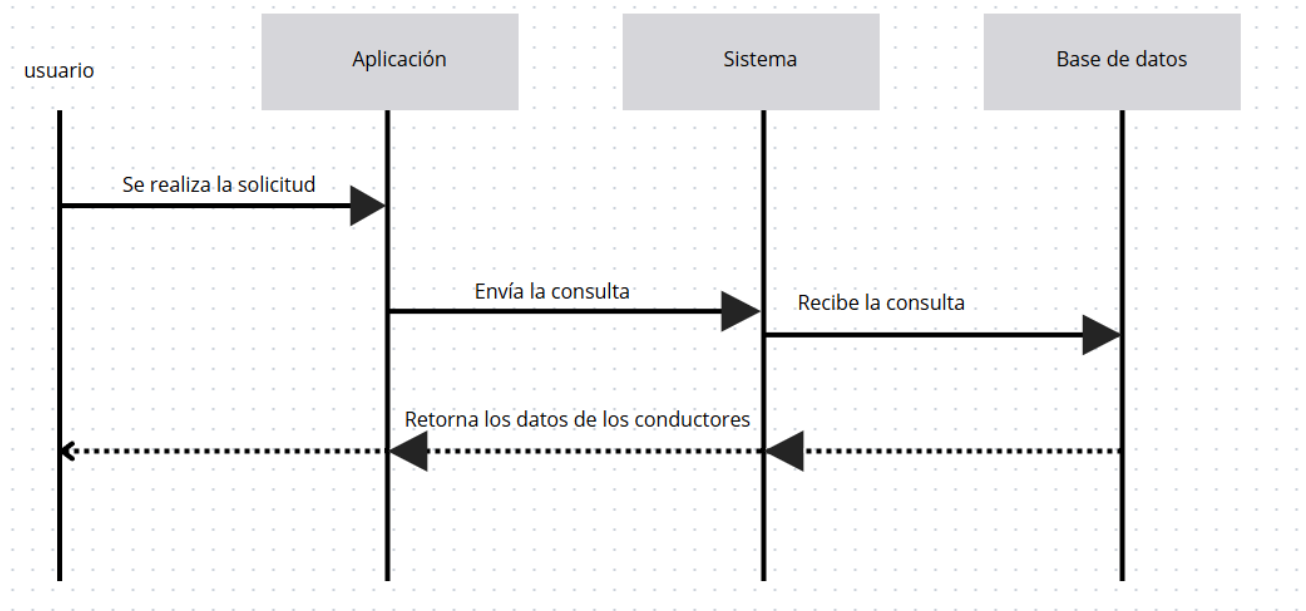
RFC1: Se desea conocer el historial de viajes solicitados por un cliente. Para realizar esto, únicamente hace falta que se ingrese el id del cliente y esto retornara un arreglo con la información de los viajes. Esto se realiza al hacer una consulta GET al endpoint /usuarios/{id}/viajes


```
"idViaje": 32,
"idServicio": 10031,
"idCliente": 1,
"idConductor": 237,
"idVehiculo": 41,
"idTarifa": 1,
"fechaHora": "2025-09-22T09:11:00.000+00:00",
"tipoServicio": "Transporte Pasajeros",
"nivelRequerido": "Estandar",
"estado": "Finalizado",
"orden": null,
"restaurante": null,
"puntoOrigen": {
    "idPunto": 8071,
    "nombre": "Origen 8071",
    "latitud": 4.644575389820116,
    "longitud": -73.9640436254024,
    "direccionAproximada": "Dir 8071",
    "ciudad": "Bogotá"
},
"destinos": [
    {
        "idPunto": 8072,
        "nombre": "Destino 8072",
        "latitud": 4.7482115413785095,
        "longitud": -73.8100283176063,
        "direccionAproximada": "Dir 8072",
        "ciudad": "Bogotá"
    },
    {
        "idPunto": 8073,
        "nombre": "Destino 8073",
        "latitud": 4.632017777552641,
        "longitud": -73.9468901379962,
        "direccionAproximada": "Dir 8073",
        "ciudad": "Bogotá"
    }
```

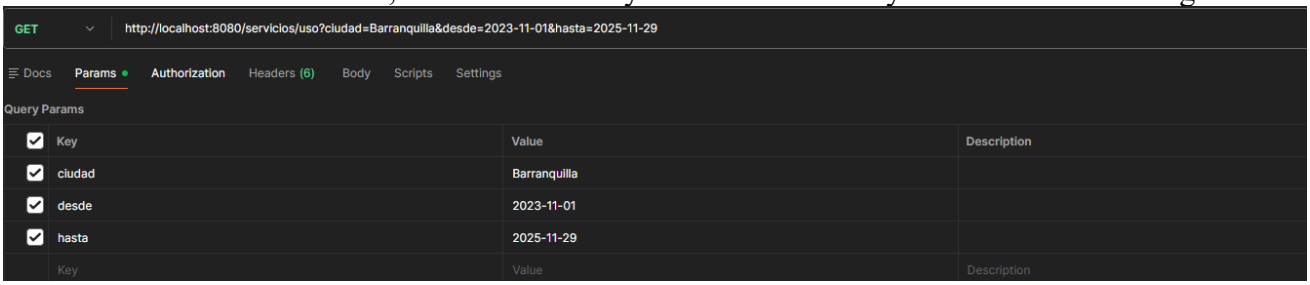



RFC2: Se desea conocer a los 20 conductores que más viajes han realizado, para esto simplemente se hace una consulta GET a la base de datos al endpoint /usuarios/top20Conductores y este traera como resultado un arreglo con el número de viajes realizados por el conductor, su id y nombre.

```
[
    {
        "numeroViajes": 16,
        "idConductor": 250,
        "nombreConductor": "Conductor 250"
    },
    {
        "numeroViajes": 15,
        "idConductor": 222,
        "nombreConductor": "Conductor 222"
    },
    {
        "numeroViajes": 12,
        "idConductor": 212,
        "nombreConductor": "Conductor 212"
    },
    {
        "numeroViajes": 12,
        "idConductor": 225,
        "nombreConductor": "Conductor 225"
    },
    {
        "numeroViajes": 11,
        "idConductor": 235,
        "nombreConductor": "Conductor 235"
    },
    {
        "numeroViajes": 11,
        "idConductor": 245,
        "nombreConductor": "Conductor 245"
```

RFC3: En este requerimiento funcional de consulta se desea consultar las estadísticas de los servicios prestados y el nivel de estos en una ciudad en específico. Para poder realizar esta consulta se necesita brindar una ciudad, una fecha inicial y una fecha final tal y como se ve en la imagen:



Y el resultado de esta consulta son los datos con la cantidad, porcentaje y tipos de servicios:

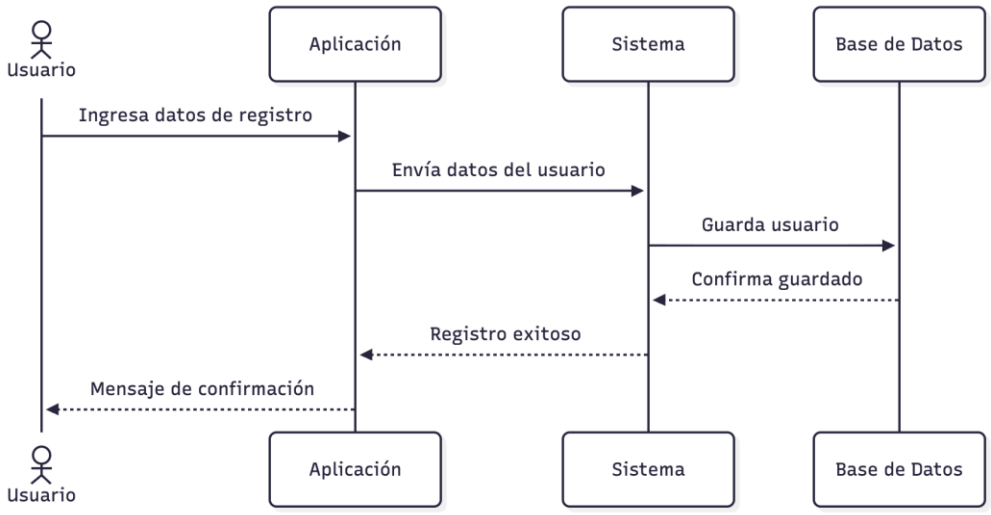
```
[
    {
        "cantidad": 19,
        "nivel": "Confort",
        "tipoServicio": "Transporte Mercancías",
        "porcentaje": 23.75
    },
    {
        "cantidad": 16,
        "nivel": "Confort",
        "tipoServicio": "Transporte Pasajeros",
        "porcentaje": 20.0
    },
    {
        "cantidad": 11,
        "nivel": "Confort",
        "tipoServicio": "Domicilio Comida",
        "porcentaje": 13.75
    },
```


RF1: Se desea registrar un usuario de tipo cliente. Para ello, es necesario que se ingresen sus datos personales y la información de la tarjeta de crédito para realizar cobros en caso de que solicite algún servicio. Estos parámetros se evidencian mejor en el ejemplo a continuación:


```
{
 "nombre": "Carlos López",
 "numeroCelular": "3001112233",
 "numeroCedula": "1012345678",
 "correoElectronico": "carlos.lopez@correo.com",
 "tipoUsuario": "Cliente",
 "tarjetasCredito": [
  {
   "idTarjetaCredito": 1,
   "titularDeLaTarjeta": "Carlos López",
   "numeroTarjeta": "4111111111111111",
   "fechaExpiracion": "2027-04",
   "codigoSeguridad": 377
  }
 ]
}
```


A

Con estos, se hace una consulta POST al endpoint /usuarios/new/save, con estos parámetros. En respuesta, se esperaría un mensaje de registro exitoso (201 created) que indica que las validaciones que se hicieron sobre la información del usuario fueron correctas. El diagrama de secuencia que ilustra lo anteriormente descrito se encuentra en la figura a continuación:



RF2: Se desea registrar un usuario de tipo conductor. Para ello, es necesario que se ingrese la información personal del usuario, y que este sea de tipo conductor. Contrario al caso anterior, no se

debe incluir información de la tarjeta de crédito, ya que esto solo es necesario para los clientes. Un ejemplo de un usuario conductor se ilustra a continuación:

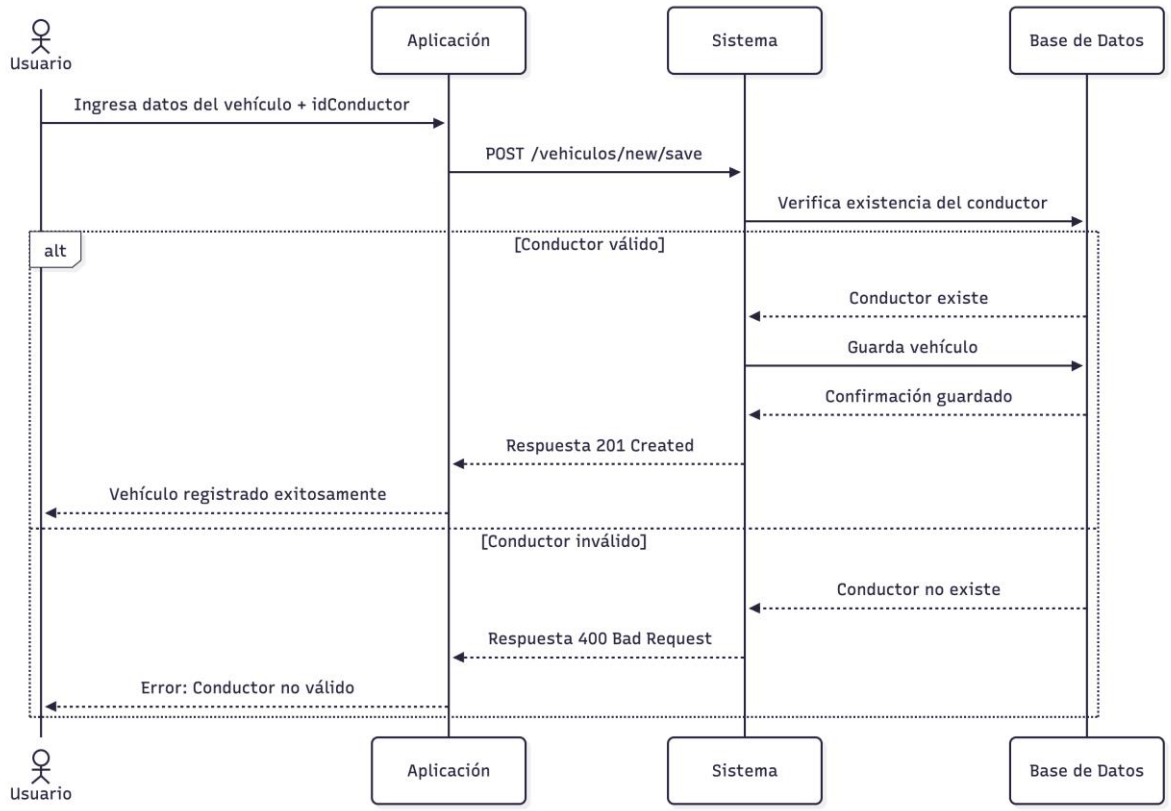
```
{
 "nombre": "Conductor Uno",
 "numeroCelular": "3002223344",
 "numeroCedula": "1023456789",
 "correoElectronico": "conductor.uno@correo.com",
 "tipoUsuario": "Conductor"
}
```

El diagrama de secuencia para este caso es idéntico al del RF 1.

RF3: Se desea verificar que el sistema permita registrar un vehículo correctamente y efectuar dicho registro en la base de datos. Se debe ingresar toda la información relevante del vehículo (placa, modelo, color, etc) y adicionalmente se debe ingresar el identificador del conductor al cual se asocia. El id del conductor que se ingrese debe existir en el sistema, como en el ejemplo a continuación:

```
{
 "idConductor": 267
 "tipo": "Automovil",
 "marca": "Toyota",
 "modelo": "Corolla",
 "color": "Rojo",
 "placa": "ABC123",
 "ciudadExpedicion": "Bogotá",
 "capacidadPasajeros": 4,
 "nivel": "Estandar"
}
```

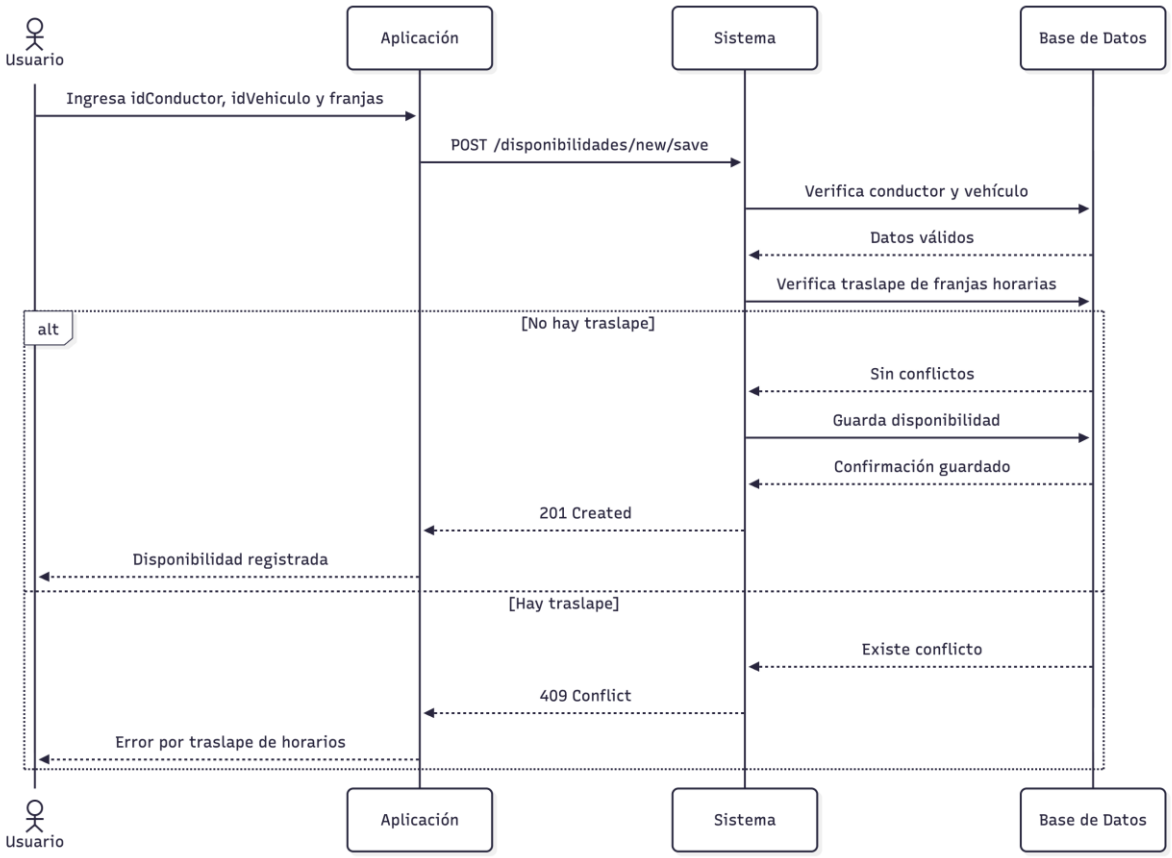
Esta información se envía mediante una consulta POST al endpoint /vehiculos/new/save y se esperaría una respuesta de 201 Created, lo que indicaría que el proceso fue exitoso y que la información se almacenó en la base de datos. En caso de que el identificador para el usuario conductor no sea válido, se esperaría una respuesta 400 Bad Request. El diagrama de secuencia asociado se puede observar en la Figura a continuación:



RF4: Se desea registrar la disponibilidad de un usuario conductor para cierto vehículo. Para esto, se deben ingresar los id del conductor y el vehículo del que se quiere registrar la disponibilidad; ambos deben coincidir con objetos en la base de datos. Adicionalmente, se debe ingresar un arreglo de franjas horarias en las que el conductor estará disponible para ese vehículo; cada una tendrá la información del día de la semana, el tipo de servicio a proveer, la hora de inicio y la hora de fin de esa franja. Un ejemplo de ello se ve así:

```
{
 "idConductor": 201,
 "idVehiculo": 99,
 "franjasHorarias": [
  {
   "diaSemana": "Miércoles",
   "horaInicio": "13:00",
   "horaFin": "14:00",
   "tipoServicio": "Transporte Pasajeros",
   "disponible": true
  }
 ]
}
```

Con estos parámetros, se hace la consulta al backend mediante una consulta POST al endpoint /disponibilidades/new/save, y se evidenciará una respuesta 201 Created, indicando que fue exitosa. En caso de que el conductor tenga traslape con otras franjas horarias ya registradas a su nombre, se evidenciará una respuesta 409 Conflict, ya que no puede tener dos franjas horarias simultáneamente. El diagrama de secuencia se evidencia a continuación:

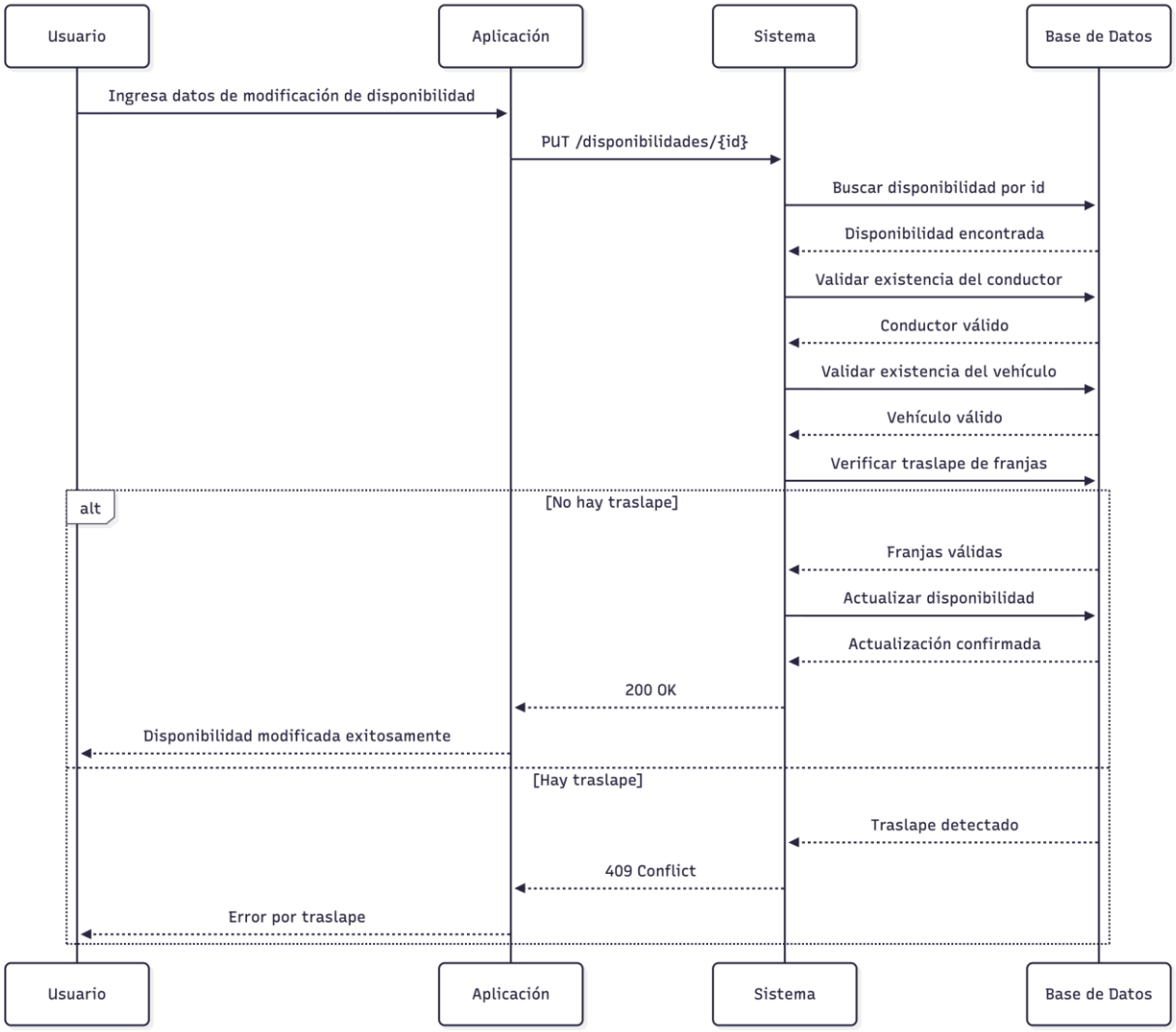


RF5: Se desea modificar la disponibilidad de un vehículo. Para esto, se deben ingresar los id del conductor y el vehículo del que se quiere registrar la disponibilidad; ambos deben coincidir con objetos en la base de datos. Adicionalmente, se debe ingresar un arreglo de franjas horarias en las que el conductor estará disponible para ese vehículo; cada una tendrá la información del día de la semana, el tipo de servicio a proveer, la hora de inicio y la hora de fin de esa franja. Obsérvese que esto es altamente similar al anterior, solo que en este caso se debe sobrescribir la disponibilidad de un vehículo que ya tenía una disponibilidad asignada para ese conductor. Por lo tanto el ejemplo que se anexará a continuación es altamente similar al anterior.

```
{
 "idConductor": 201,
 "idVehiculo": 99,
 "franjasHorarias": [
  {
   "idFranja": 1000,
   "diaSemana": "Lunes",
   "horaInicio": "01:00",
   "horaFin": "01:10",
   "tipoServicio": "Transporte Pasajeros",
   "disponible": true
  }
 ]
}
```

Con los parámetros anteriores, se hace una consulta al backend con un método PUT al endpoint /disponibilidades/{{id}}, donde {{id}} corresponde al id de la disponibilidad que se desea modificar. Si todo está en orden, no hay traslape y los ids ingresados son correctos, se recibe una respuesta de tipo 200 OK. Mientras que, si hay algún traslape con las nuevas franjas y las que el conductor ya

tenía asignadas para ese vehículo o para otros vehículos, se obtiene una respuesta 409 Conflict. Lo anterior, se puede evidenciar mejor en el siguiente diagrama de secuencia:

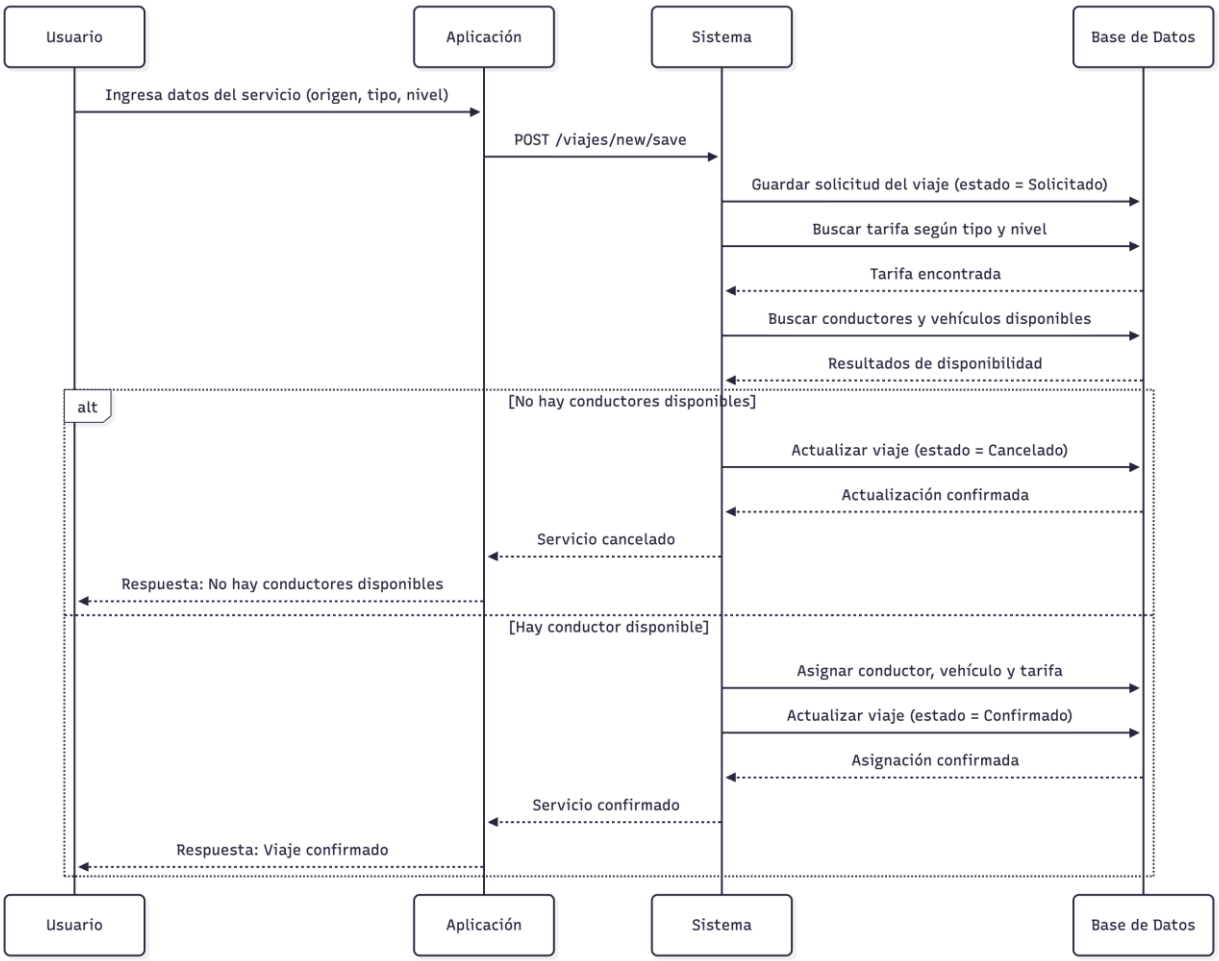


RF6: En este escenario de prueba, es el usuario quien desea solicitar un servicio. Para ello el usuario debe ingresar la información del punto de partida (coordenadas, dirección aproximada, ciudad), el tipo de servicio que requiere y el nivel del vehículo que desea. Con esta información, se decide la tarifa que aplicará al servicio y se hace una búsqueda de los vehiculos que cumplan con las especificaciones y cuyo conductor esté disponible en el momento para proveer ese servicio. En caso de que no se encuentre ningún conductor en el momento, se da por “Cancelado” el servicio; de lo contrario, se le asigna un conductor, un vehículo y se le modifica el estado a “Confirmado”. Un ejemplo del script que se debe ingresar se observa a continuación:

```
{
    "idViaje": 1,
    "idServicio": 10000,
    "idCliente": 262,
    "idConductor": -1,
    "idVehiculo": -1,
    "idTarifa": -1,
    "fechaHora": "2025-09-24T11:12:00.000+00:00",
    "tipoServicio": "Transporte Pasajeros",
```

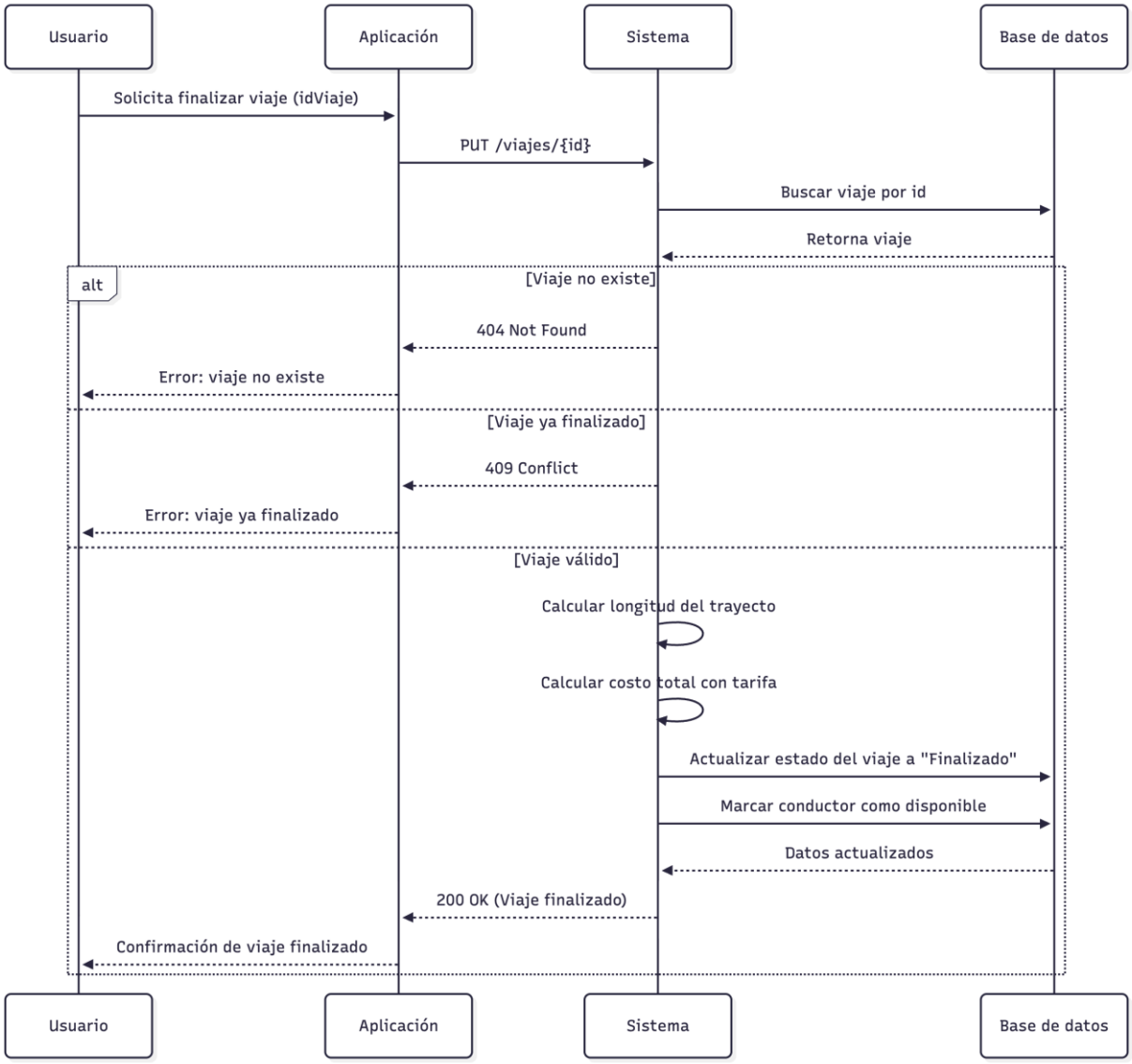
```
    "nivelRequerido": "Estandar",
    "estado": "Solicitado",
    "orden": null,
    "restaurante": null,
    "puntoOrigen": {
      "idPunto": 8000,
      "nombre": "Origen 8000",
      "latitud": 4.843452133743705,
      "longitud": -74.04382328072059,
      "direccionAproximada": "Dir 8000",
      "ciudad": "Barranquilla"
    },
    "destinos": [
      {
        "idPunto": 8001,
        "nombre": "Destino 8001",
        "latitud": 4.6249609913270255,
        "longitud": -73.86793803042919,
        "direccionAproximada": "Dir 8001",
        "ciudad": "Barranquilla"
      }
    ],
    "fechaHoraInicio": "2025-09-24T11:13:00.000+00:00",
    "fechaHoraFin": "2025-09-24T11:23:00.000+00:00",
    "longitudTrayecto": 0,
    "costoTotal": 0,
    "reviews": null
  }
```


Como se puede observar, las variables idConductor, idVehiculo, idTarifa se instancian en -1, ya que estas se asignan en la lógica dependiendo de los parámetros que ingrese el usuario. Asimismo, las variables de fechaHora, fechaHoraInicio, fechaHoraFin, se instancian con una fecha aleatoria, ya que estas se modifican en la lógica dependiendo de los momentos en que se hace la solicitud, se consigue el vehículo y posteriormente cuando se culmina un servicio. Este script se envía mediante una consulta POST al endpoint /viajes/new/save, y se esperaría una respuesta 200 OK cuando se encuentre a alguien que cumpla con el servicio y cuente con la disponibilidad; mientras que, se espearía una respuesta 400 Bad Request si no hay ningún conductor que pueda tomar ese servicio. El diagrama de secuencia asociado se evidencia a continuación:



RF7: Se desea registara un viaje para un usuario, cuando el servicio ya ha sido prestado por parte de un conductor. En este caso, solo se necesita el identificador del viaje que se creó en el paso anterior, para que en el Sistema se pueda poner el estado de “Finalizado”, se pueda calcular la longitud total del trayecto (esto mediante los puntos geográficos que ya se tenían) y el costo total. Este úlimo, calculado con la tarifa por kilómetro y la longitud total del trayecto. Al culminar, se vuelve a registrar al usuario conductor como disponible.

Para ello, se hace una consulta PUT al endpoint /viajes/{{id}}, donde {{id}} corresponde al identificador del viaje que se creó en el requerimiento anterior, que estaba “Confirmado”. Se esperaría un mensaje indicando que el viaje se finalizó exitosamente, con una respuesta 200 OK; en caso de que ya se hubiese finalizado, se esperaría un error 409 Conflict, y si se ingresa un id no válido, es decir que no coincida con ningún viaje, se arroja una respuesta 404 Not Found. El diagrama de secuencia asociado a este requerimiento se puede evidenciar a continuación:



4. Población Base de Datos

Para poblar las diferentes colecciones, se generó un script con todos los datos a insertar. Este archivo se encuentra en la ruta “src\db\poblar.js”.

Comando para correrlo en consola:

mongosh "mongodb://ISIS2304A01202520:kuuuULHdgiAZ@157.253.236.88:8087/ISIS2304A01202520" --file .\src\db\poblar.js

También se puede dirigir a MongoDB Compass, conectarsecon el connection string: “mongodb://ISIS2304A01202520:kuuuULHdgiAZ@157.253.236.88:8087/ISIS2304A01202520” y pegar el contenido del script en MongoDB Shell y ejecutarlo.

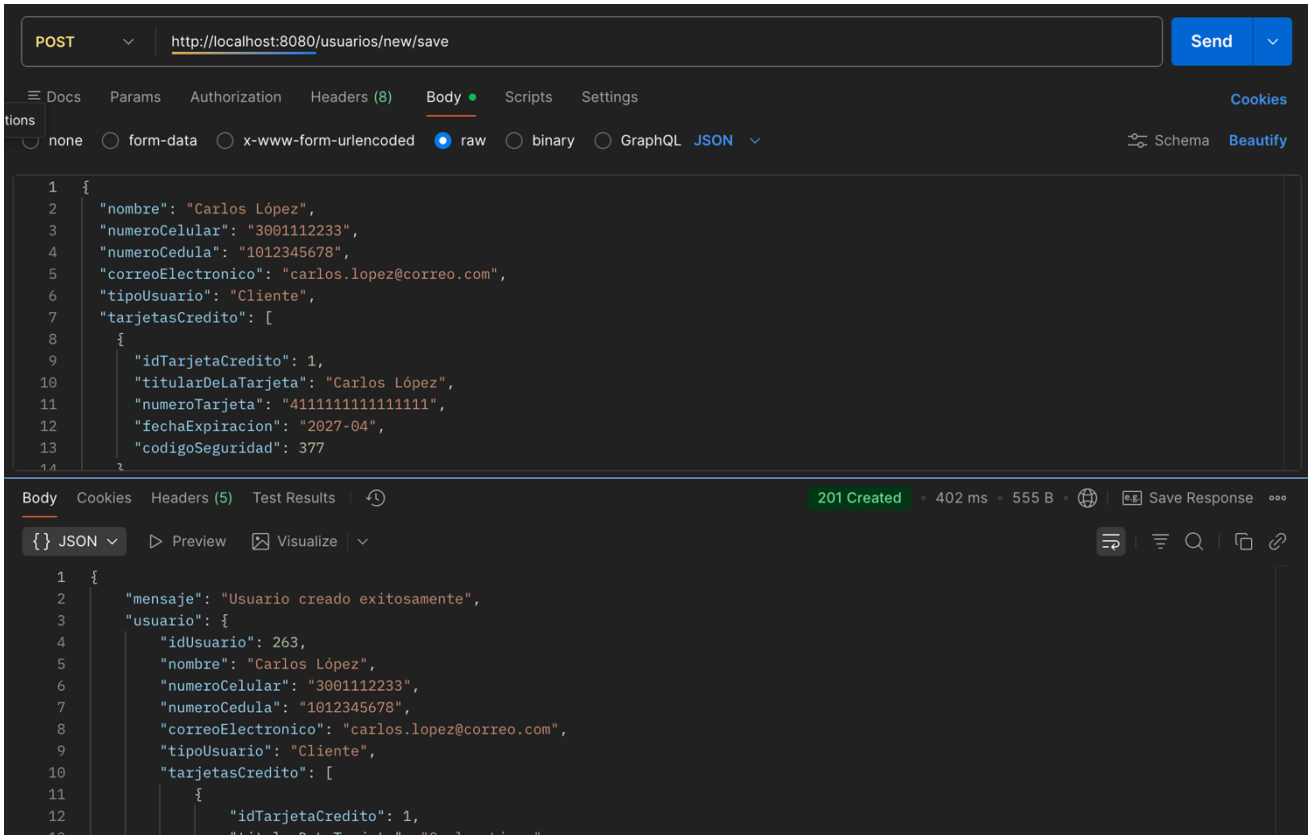
5. Implementación Requerimientos

RFC1:

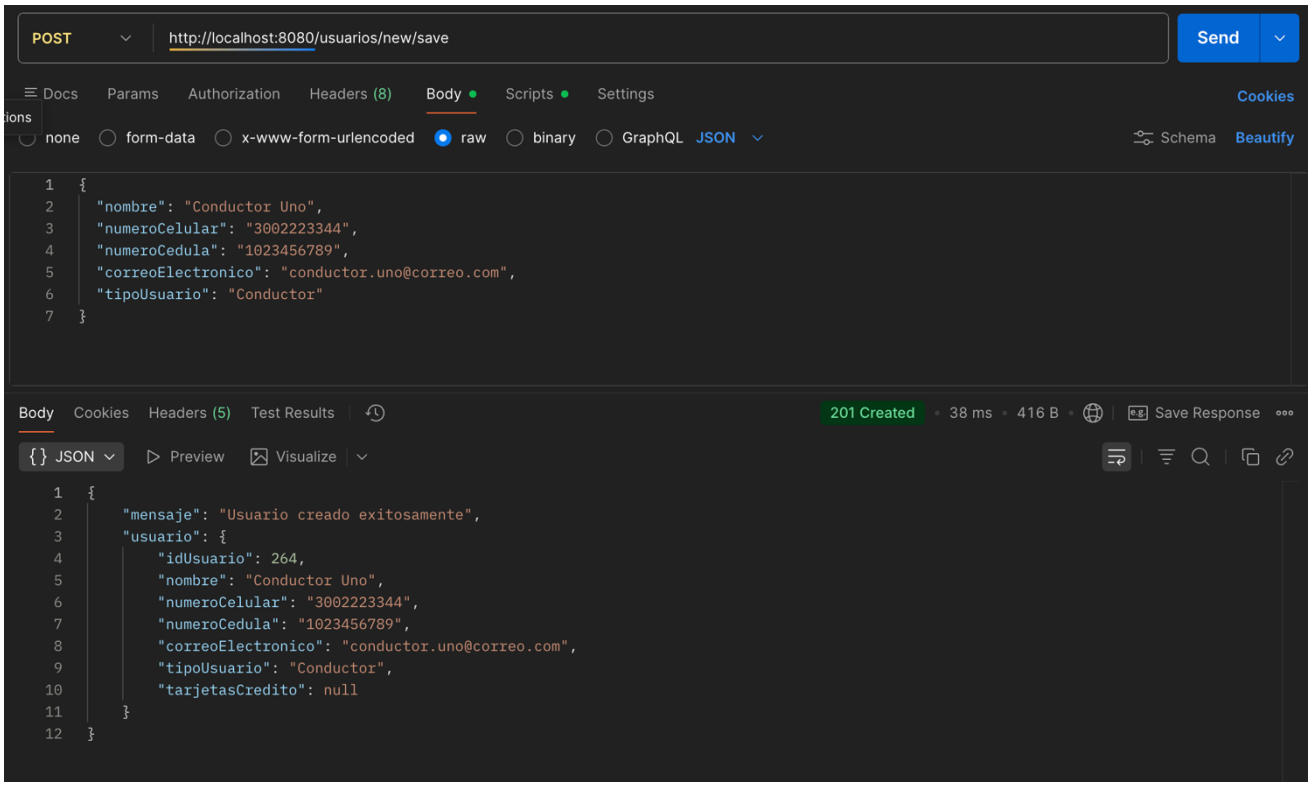
RFC2:

RFC3:

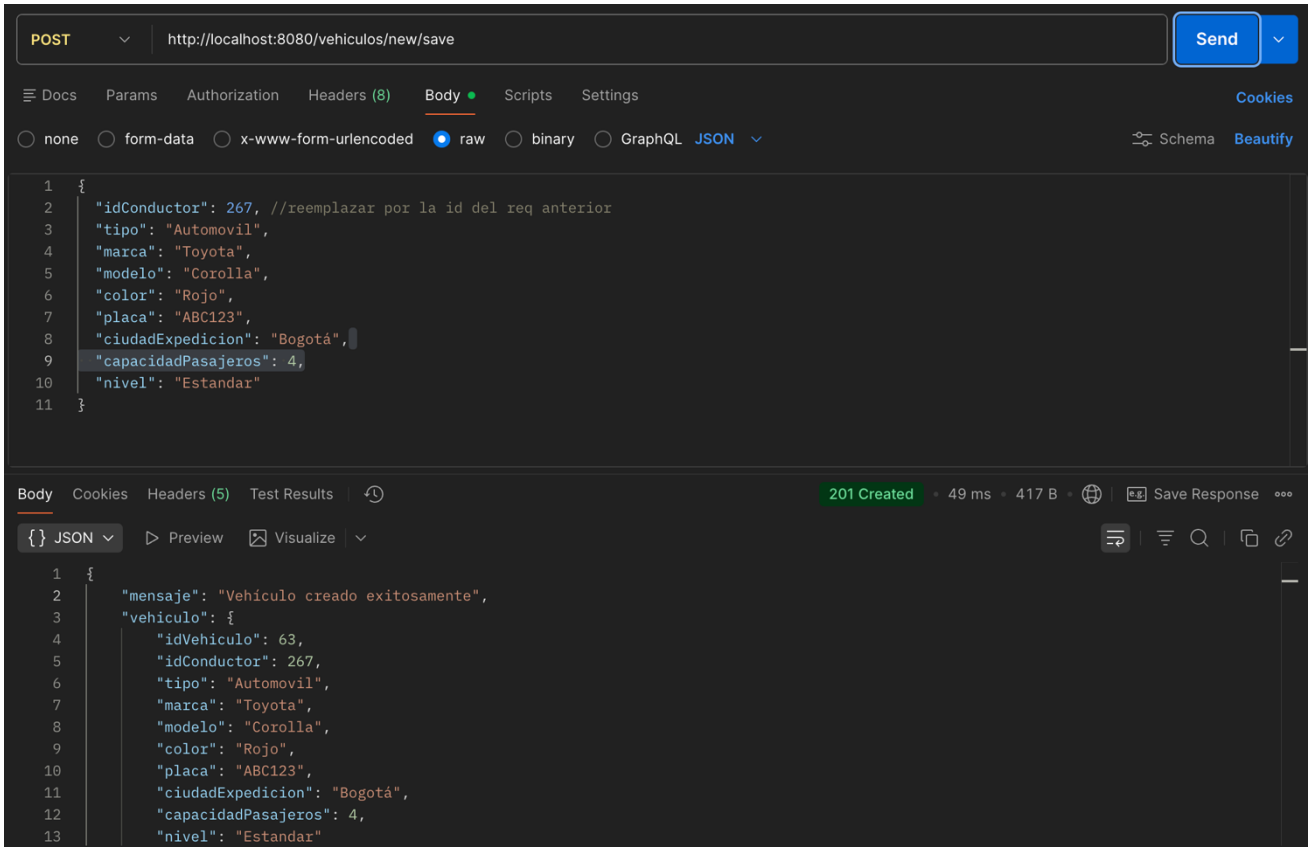
RF1: Se crea un usuario con las credenciales que se ingresaron en el caso de prueba. Se obtiene una respuesta 201 Created ya que no hubo ningún problema.



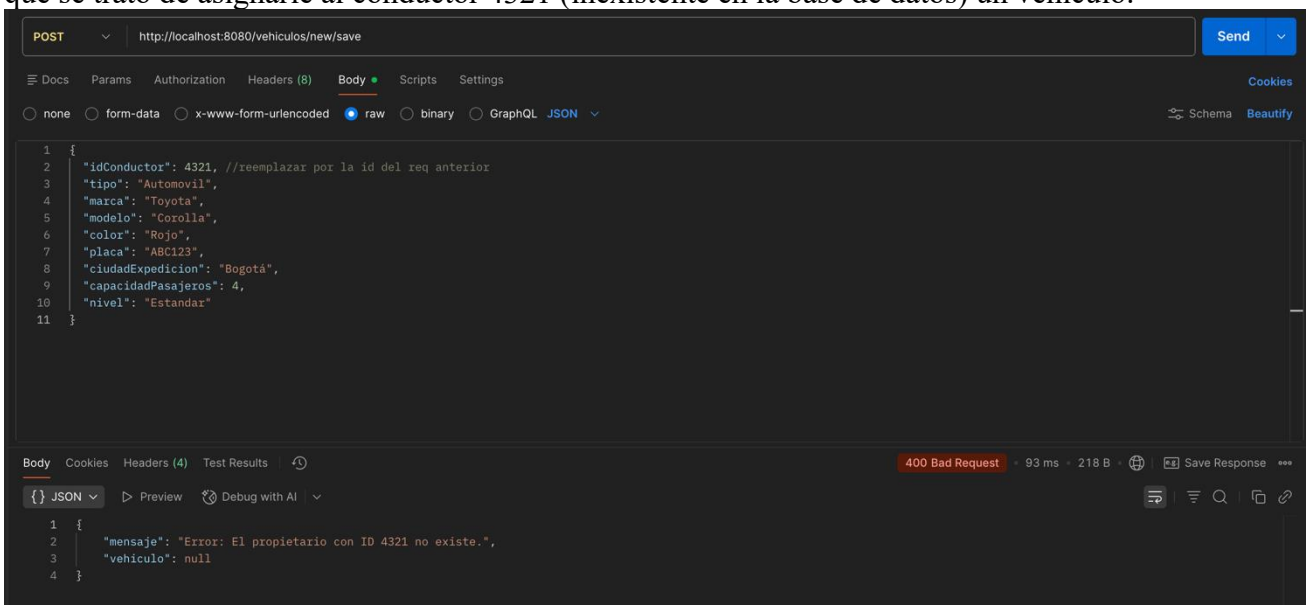
RF2: Se crea un usuario conductor con las credenciales que se plantearon en el caso de prueba. Se obtiene una respuesta 201 Created, indicando que fue exitoso y que este usuario ya se encuentra en la base de datos.



RF3: Se crea un vehículo con las credenciales planteadas en el caso de prueba. Se obtiene una respuesta 201 Created, lo que indica que fue exitoso y que el id ingresado para el conductor coincide con un conductor en la base de datos.

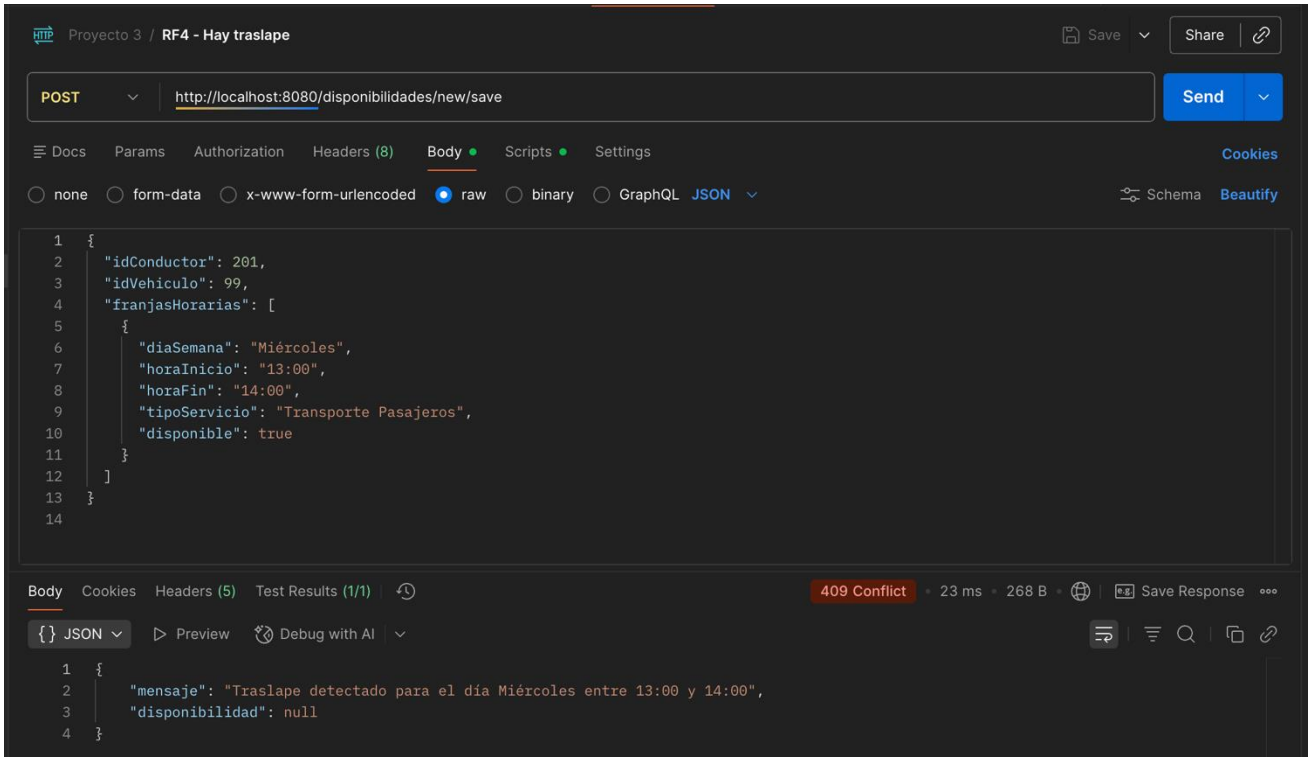


En caso de que se ingrese un id para conductor no válido, se obtiene un error 400 Bad Request, junto con el mensaje que indica lo que se acabó de mencionar. En el ejemplo anexo, se puede evidenciar que se trató de asignarle al conductor 4321 (inexistente en la base de datos) un vehículo.

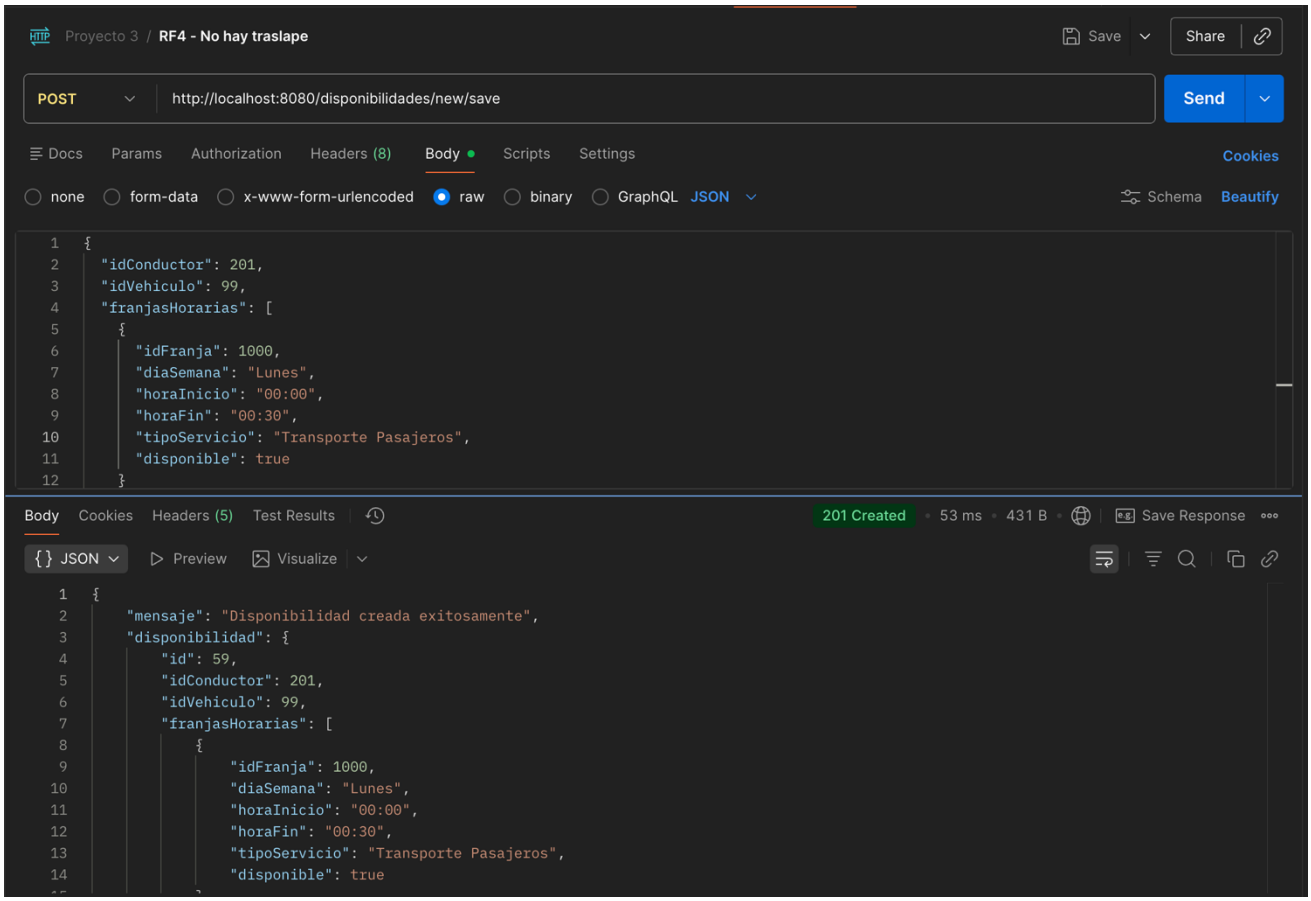


RF4:

En el siguiente ejemplo, se puede evidenciar que se intenta crear una disponibilidad para el conductor, pero las franjas horarias ingresadas coinciden con otras franjas que este tiene para otros vehículos. Por eso es que se evidencia el error 409 Conflict.

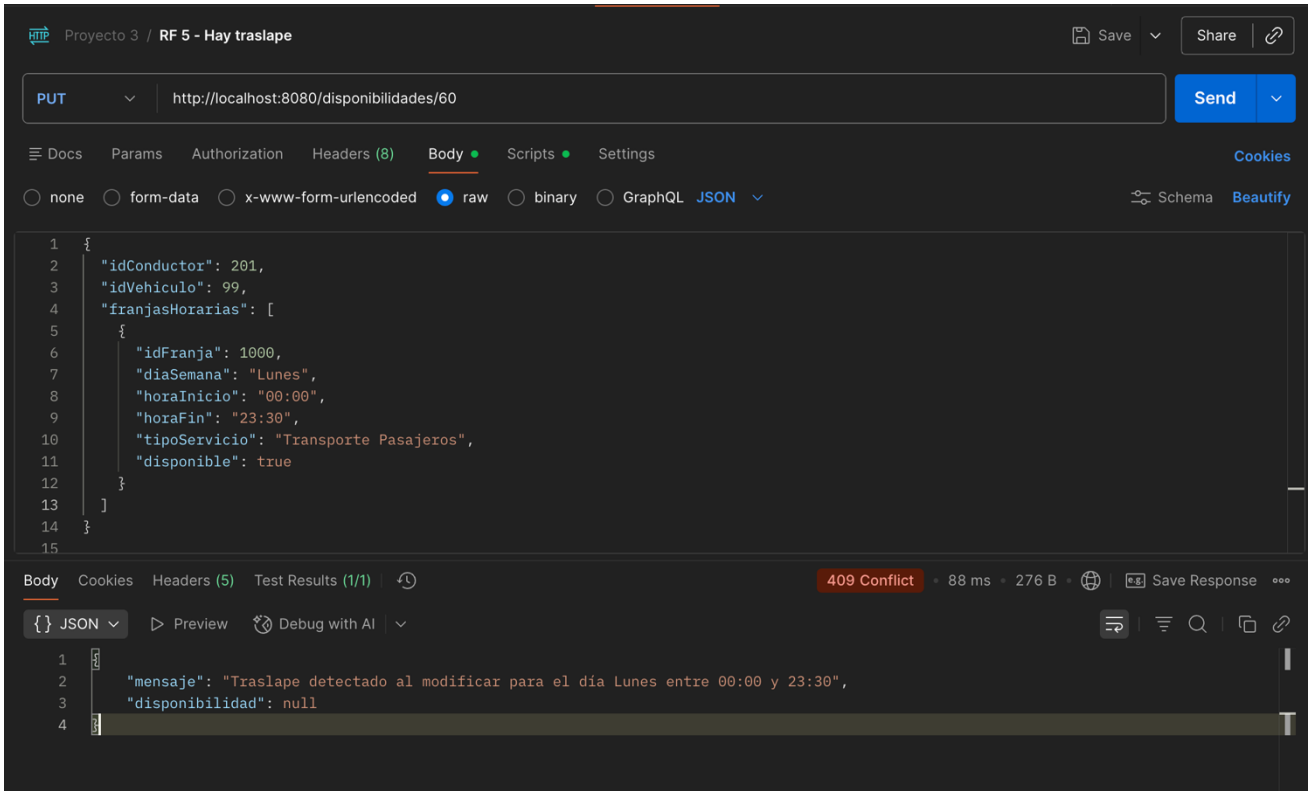


Ahora bien, para el caso correcto en el que ya no hay traslape con otras franjas, se puede evidenciar la respuesta 201 Created.

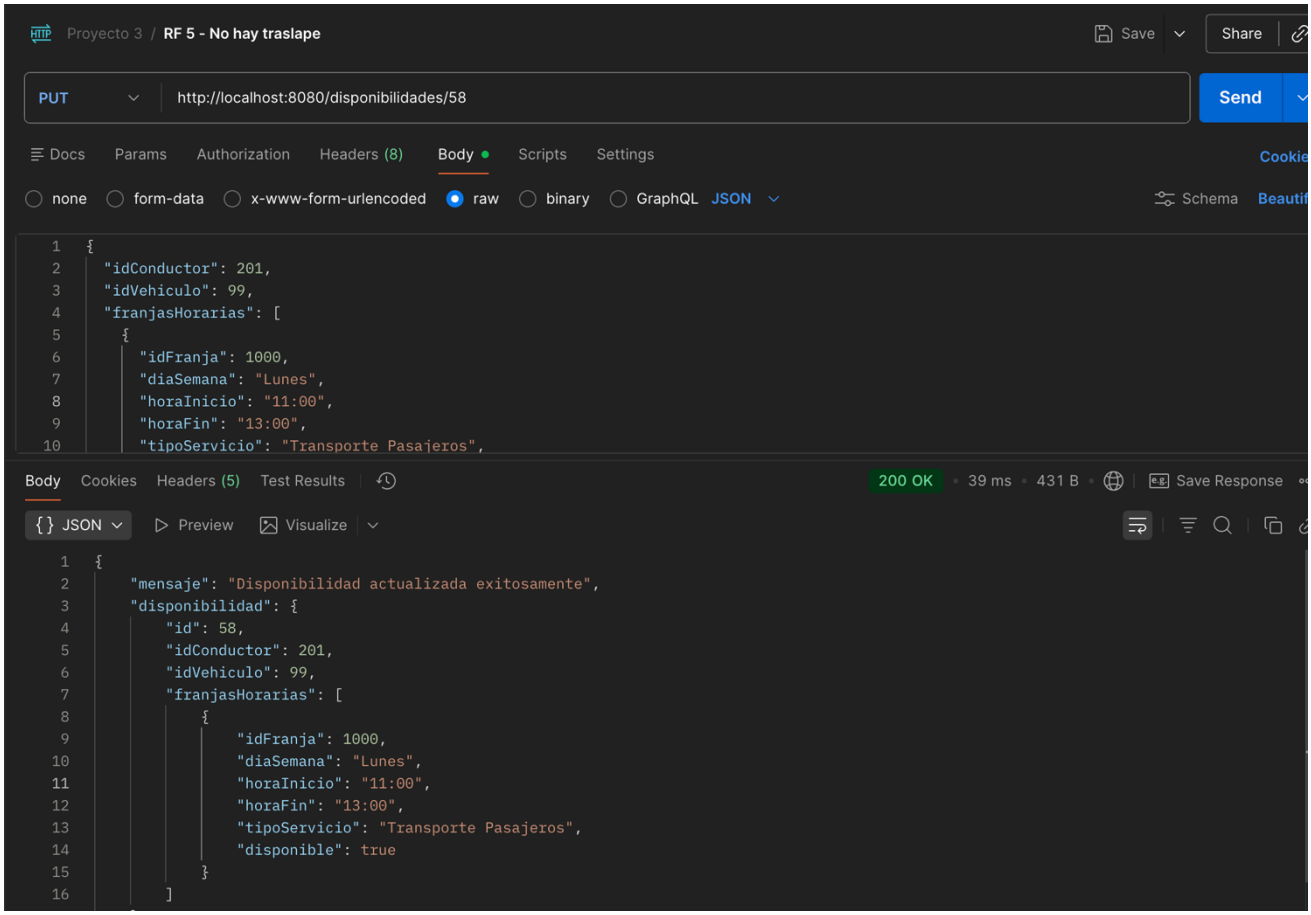


RF5:

En el siguiente ejemplo, se puede evidenciar que se intenta modificar una disponibilidad para el conductor, pero las franjas horarias ingresadas coinciden con otras franjas que este tiene para otros vehículos que no sean las que corresponden a la disponibilidad que se desea cambiar. Por eso es que se evidencia el error 409 Conflict.

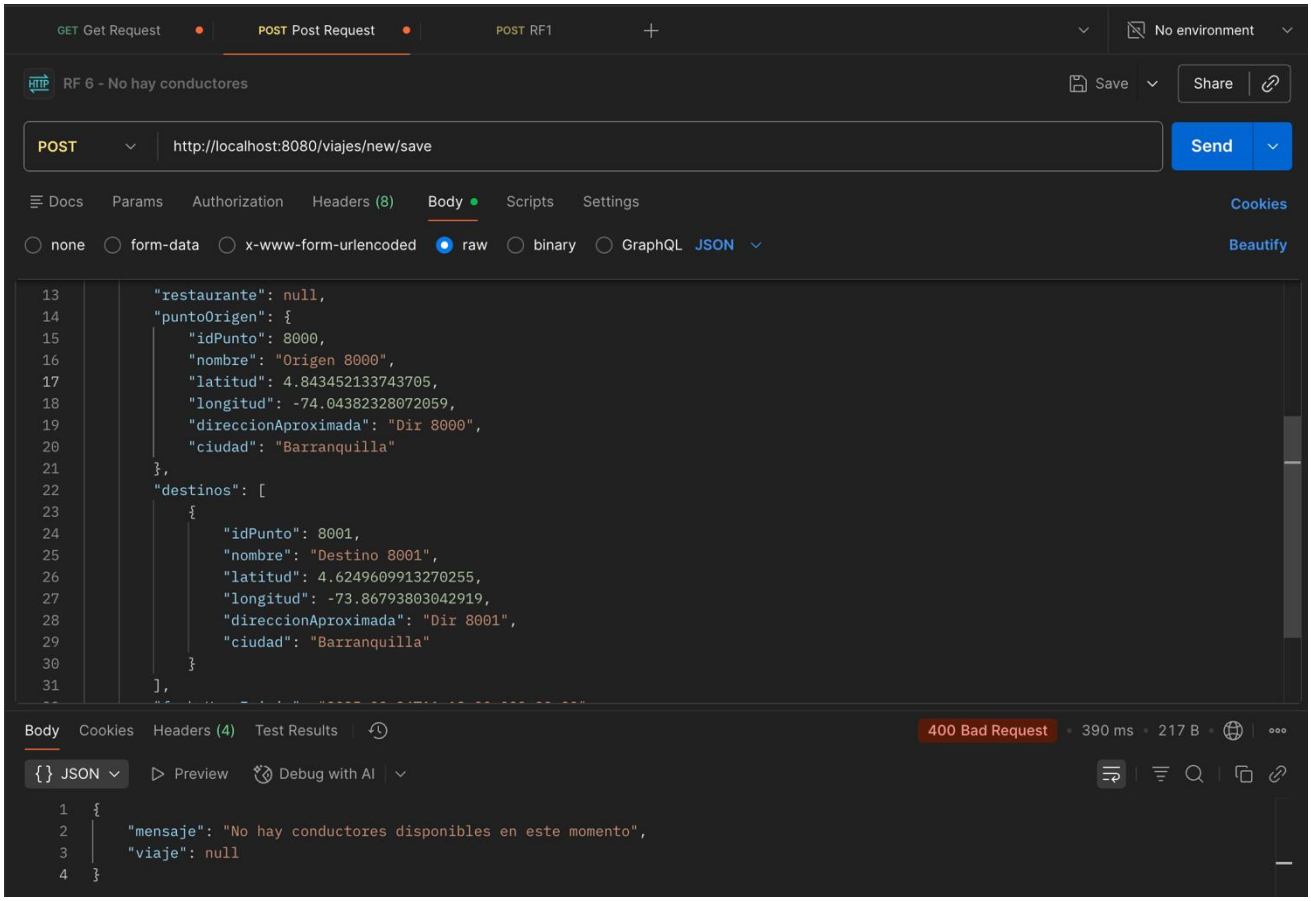


Ahora bien, para el caso correcto en el que ya no hay traslape con otras franjas, se puede evidenciar la respuesta 201 Created.

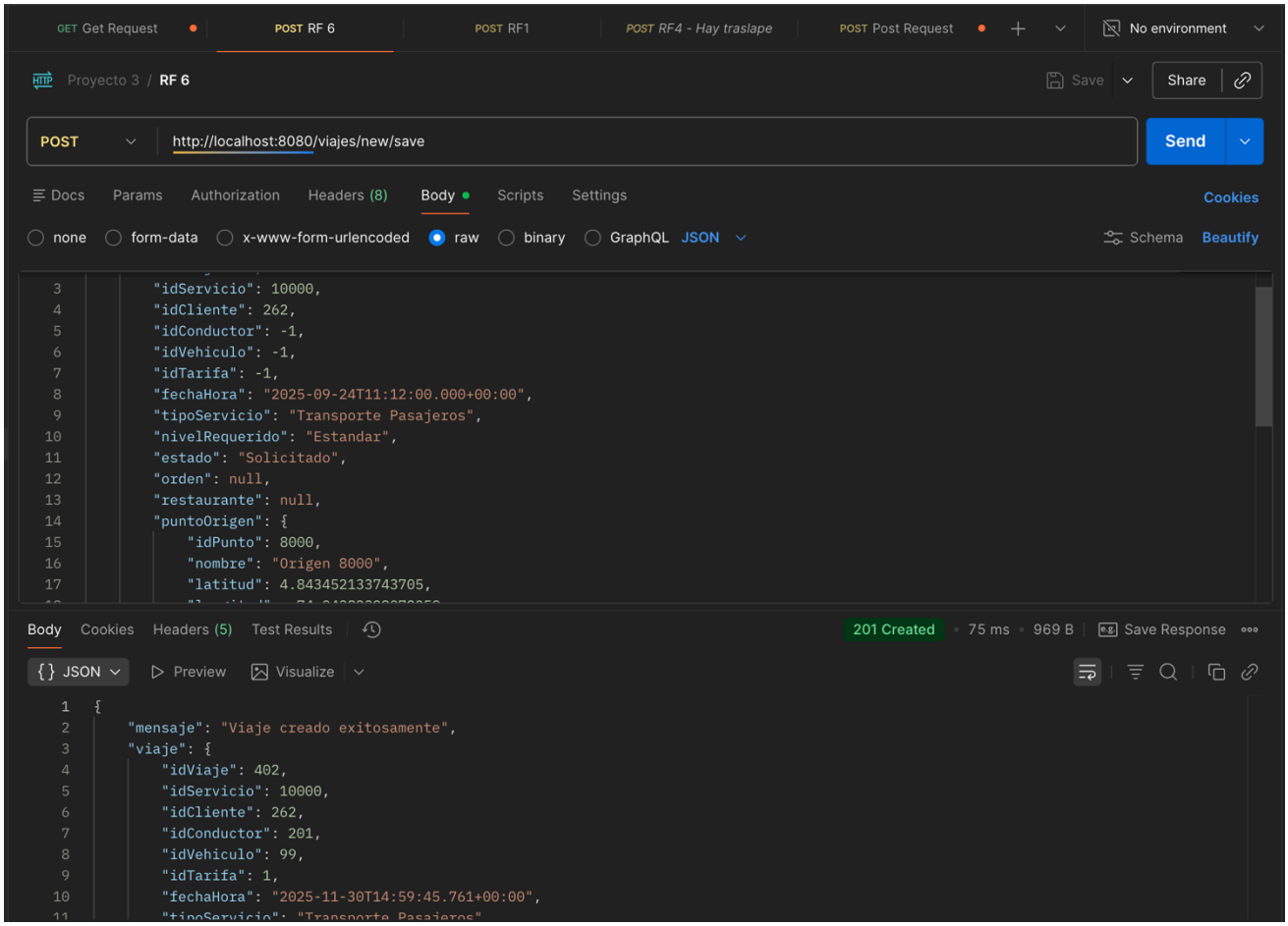


RF6:

Se intenta solicitar un servicio con los parámetros que se plantearon en el caso de prueba y para un usuario activo. Para el primer caso, se puede evidenciar que no hay ningún conductor disponible para proveer el servicio; por lo tanto, se obtiene una respuesta 400 Bad Request.

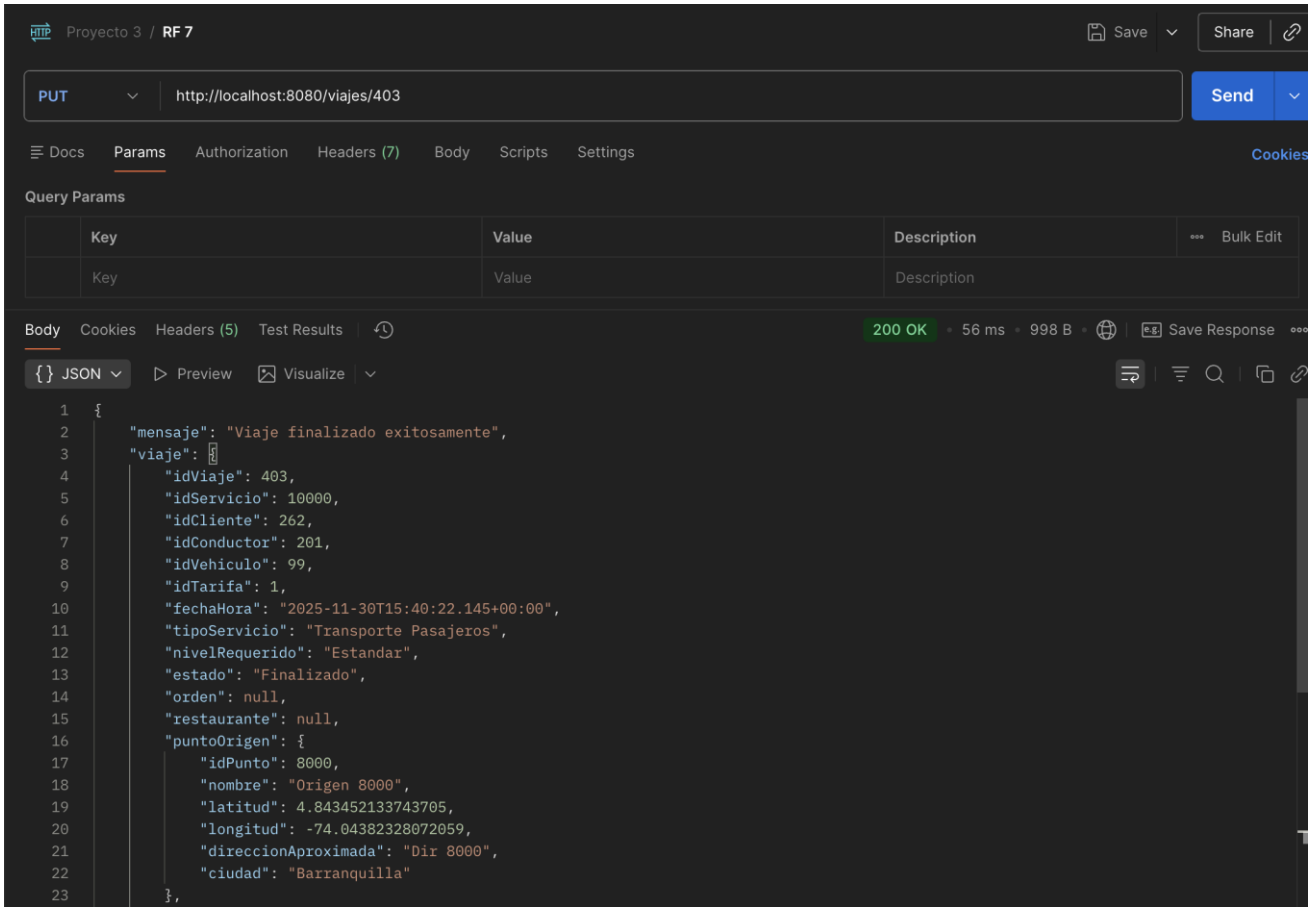


Ahora bien, en caso de que si haya disponibilidad de algún conductor en el momento de la solicitud, se evidencia una respuesta 201 Created que indica que el proceso fue exitoso.

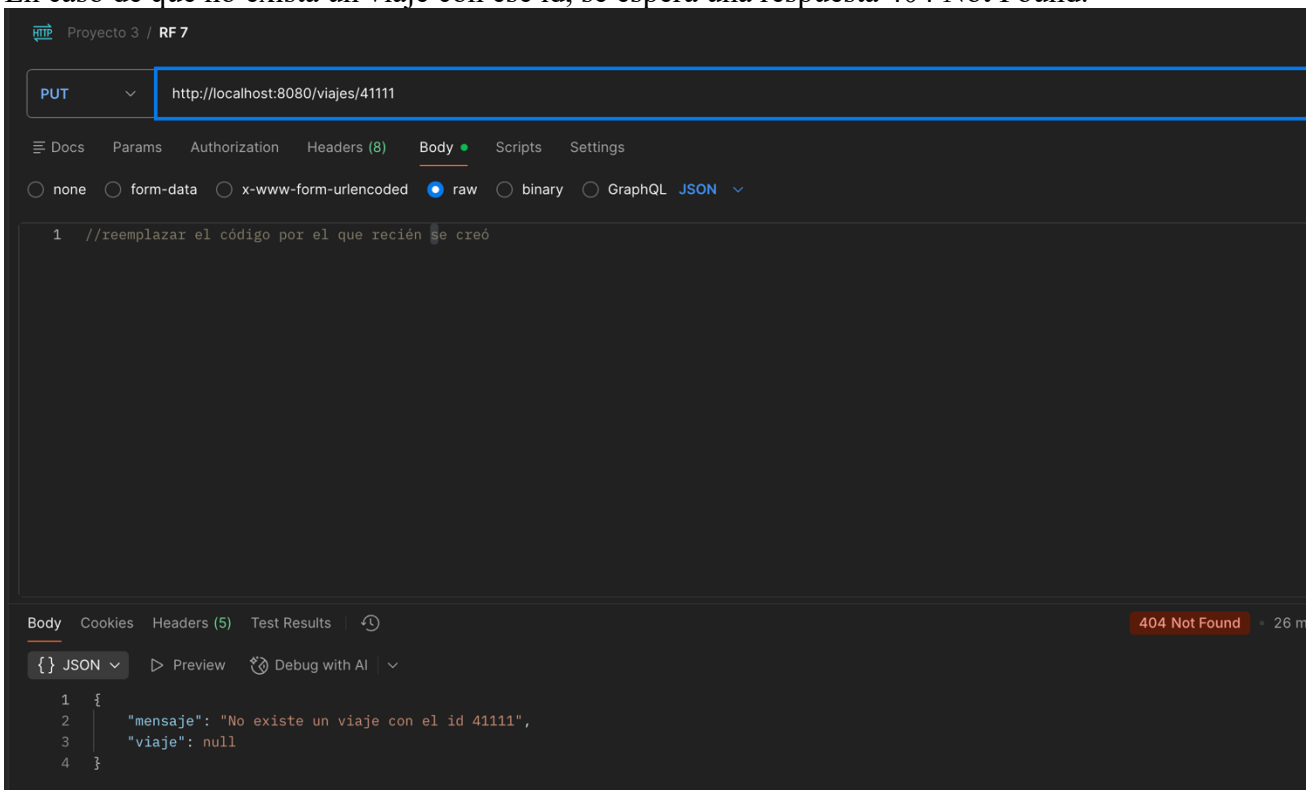


RF7:

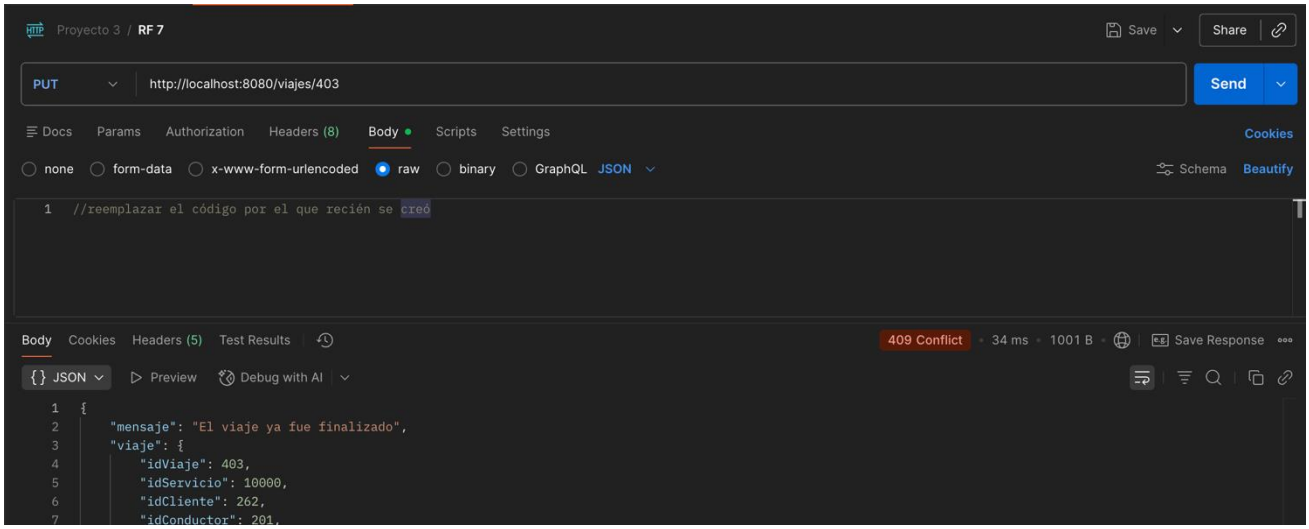
Con el id del viaje que se creó en el paso anterior, este se inserta como parte de la consulta. En este caso, se puede evidenciar que la petición fue exitosa y que el viaje ya se dio por finalizado y que se hicieron las respectivas modificaciones a sus parámetros.



En caso de que no exista un viaje con ese id, se espera una respuesta 404 Not Found.



Y finalmente, si ese viaje ya existe pero si este ya se finalizó, se espera una respuesta 409 Conflict.


Proyecto 3 / RF 7
Save
Share
PUT
http://localhost:8080/viajes/403
Send
Docs
Params
Authorization
Headers (8)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
1 //reemplazar el código por el que recién se creó
Body
Cookies
Headers (5)
Test Results
409 Conflict
34 ms
1001 B
Save Response
JSON
Preview
Debug with AI
1 {
2 "mensaje": "El viaje ya fue finalizado",
3 "viaje": {
4 "idViaje": 403,
5 "idServicio": 10000,
6 "idCliente": 262,
7 "idConductor": 201,