

Wheels es una aplicación/plataforma de movilidad (muy parecida a Uber, Beat o Didi) que funciona con **carros compartidos, vans o buses pequeños** para transportar personas.

📌 Generalmente:

- Se usa para **llover grupos de personas** hacia un destino común (por ejemplo, universidades, oficinas, eventos).
- Puede ser un **servicio privado contratado por instituciones** (como rutas universitarias o empresariales) o un servicio abierto en el que estudiantes se inscriben y comparten el viaje.
- Ayuda a **reducir costos**, la huella ambiental y el tráfico, porque en lugar de que cada estudiante vaya en su carro, todos comparten un mismo vehículo.

Es básicamente una mezcla entre:

🚌 Ruta escolar/universitaria + 🚗 app tipo Uber.

💡 Idea central

Una aplicación donde los estudiantes puedan:

- Registrar su ruta hacia/desde la universidad.
- Ver en un mapa a otros estudiantes con trayectos similares.
- Coordinar viajes compartidos (conductor/pasajero).
- Calcular tiempos de llegada y rutas óptimas.

🔑 APIs que puedes usar

1. Google Maps API

- a. Geolocalización (saber dónde está el usuario).
- b. Autocompletar direcciones.
- c. Calcular rutas y tiempo estimado.
- d. Mostrar marcadores en el mapa.

2. Waze API (Waze Transport SDK o Waze Deep Links)

- a. Para abrir directamente la navegación en Waze.

- b. También puedes usar Waze for Broadcasters si quieres mostrar tráfico en tiempo real.

3. API de tipo Uber/Didi (Ride-hailing)

- a. No todas son públicas, pero puedes usar **Uber API**:
 - i. Cotizar viajes.
 - ii. Estimar tiempo de llegada.
 - iii. Integrar botón de pedir Uber directamente.
- b. Alternativa: si no te dan acceso a APIs privadas, puedes **simular la lógica** (matching de pasajeros y conductores dentro de tu app).

Tecnologías recomendadas

- **Frontend:** React, Next.js o Vue (para una interfaz moderna con mapas).
- **Backend:** Node.js (Express) o Python (Django/FastAPI) para manejar usuarios y viajes.
- **Base de datos:** PostgreSQL / Firebase (para almacenar usuarios, rutas, viajes).
- **Autenticación:** Google OAuth o sistema con correo institucional @unisabana.edu.co.

Flujo de la aplicación

1. Registro/Login

- a. Solo estudiantes con correo institucional.

2. Crear viaje (como conductor)

- a. Selecciona origen/destino.
- b. Escoge hora de salida.
- c. Se publica en la app.

3. Unirse a viaje (como pasajero)

- a. Busca viajes disponibles en el mapa.
- b. Solicita unirse.

4. Matching automático

- a. La app sugiere viajes con rutas similares.

5. Integración de APIs

- a. Mostrar mapa con rutas en **Google Maps**.
- b. Botón de abrir ruta en **Waze**.
- c. (Opcional) Comparación con precio estimado de Uber/Didi.

Funcionalidades extra (para destacarte)

- Chat interno (para coordinar recogida).
- Calificación de conductores/pasajeros.
- Sistema de puntos/recompensas (gamificación).
- Seguridad: compartir viaje con un contacto de emergencia.

 En resumen:

Tu proyecto sería un **carpool universitario tipo Wheels**, con **Google Maps para rutas**, **Waze para navegación** y **Uber API como referencia/complemento**.

Wheels universitario para La Sabana, pero con un enfoque **simple y seguro**:

- **Rutas fijas/pautadas** (como si fueran minibuses o colectivos, pero organizados entre estudiantes).
- **Reserva de cupos** según la ubicación del usuario y los puntos predefinidos.
- **Sin pagos dentro de la app** (por ahora), solo coordinación entre estudiantes → efectivo o Nequi directo.

Flujo actualizado de la app

1. Registro/Login (solo estudiantes de La Sabana)

- a. Con correo institucional @unisabana.edu.co o código estudiantil.
- b. Perfil con foto + placa del carro (si es conductor).

2. Rutas fijas/pautadas (PRs)

- a. El administrador define rutas principales → Ejemplo:
 - i. Bogotá Norte → Sabana
 - ii. Chía → Sabana
 - iii. Cajicá → Sabana

- b. Se muestran en un mapa con puntos de recogida.

3. Reservar cupo

- a. El estudiante elige:
 - i. Ruta fija (PR).
 - ii. Punto de recogida cercano.
 - iii. Hora programada.
- b. Confirma la reserva (notificación al conductor).

4. Conductor

- a. Ve la lista de pasajeros en su ruta.
- b. Puede confirmar/rechazar reservas.
- c. Usa la integración con **Waze/Google Maps** para seguir la ruta optimizada.

5. Viaje en curso

- a. Pasajeros reciben notificación cuando el conductor está cerca.
- b. Se abre la navegación en **Waze o Google Maps**.

6. Pago

- a. No en la app.
- b. Se acuerda: **efectivo o Nequi** al llegar.

Tecnologías / Integraciones

- **Google Maps API** → mostrar mapa, puntos de recogida y rutas.
- **Waze Deep Links** → abrir directamente la navegación para el conductor.
- **Backend (Node.js o Django)** → gestionar rutas, reservas y usuarios.
- **Firebase o PostgreSQL** → guardar rutas y reservas en tiempo real.

Seguridad (muy importante en una universidad)

- Solo estudiantes registrados.
- Verificación con correo institucional.
- Opción de compartir viaje con un contacto de confianza.
- Calificaciones (conductor ↔ pasajero).

Posibles mejoras futuras

- Integrar pagos en línea (Stripe, Nequi API, MercadoPago).
- Algoritmo de matching dinámico (si alguien crea su ruta personalizada).
- Estadísticas de ahorro de CO₂ (impacto ambiental positivo).

Ejemplos para tu app de Wheels Sabana

1. Registro

- Como estudiante de la Universidad de La Sabana, quiero registrarme con mi correo institucional, para poder usar la aplicación de forma segura y solo entre la comunidad.

2. Rutas fijas (PRs)

- Como pasajero, quiero ver las rutas pautadas disponibles en el mapa, para poder elegir la que más me convenga según mi ubicación.

3. Reserva de cupo

- Como estudiante pasajero, quiero reservar un cupo en una ruta y punto de recogida, para poder asegurarme transporte hacia la universidad.

4. Conductor

- Como conductor estudiante, quiero ver quiénes se han registrado en mi ruta y confirmar o rechazar reservas, para poder organizar mi viaje con anticipación.

5. Navegación

- Como conductor, quiero abrir la ruta directamente en Waze o Google Maps, para poder seguir la mejor ruta sin desviarme.

6. Notificaciones

- Como pasajero, quiero recibir una notificación cuando el conductor esté cerca, para poder prepararme y no retrasar el viaje.

7. Pagos

- Como *usuario pasajero*,
quiero acordar el pago directamente con el conductor en efectivo o Nequi,
para poder *pagar de forma simple sin necesidad de la app*.

💡 Cómo usar esto en Figma

- Cada **pantalla** (login, mapa, reserva, notificaciones) debe responder a una historia de usuario.
- Ejemplo:
 - Historia de usuario → *Reserva de cupo*.
 - Pantalla en Figma → Lista de rutas + botón "Reservar".

De esta manera, cada diseño en Figma se justifica con un “Como, quiero, para poder”.

❖ ¿Qué es un ticket?

- Es un **ítem** en el backlog o tablero.
- Representa una tarea concreta que alguien del equipo debe hacer.
- Suele contener:
 - **Título** → lo que se va a hacer.
 - **Descripción** → detalles o criterios de aceptación.
 - **Responsable** → quién lo ejecuta.
 - **Estado** → pendiente, en progreso, hecho.

👉 Ejemplo en Wheels Sabana:

- Ticket: “Diseñar pantalla de reservas en Figma”.
- Ticket: “Integrar API de Google Maps para mostrar rutas”.

❖ Agrupación de tickets

Los tickets **no viven solos**, se agrupan jerárquicamente:

1. Subtask (Subtarea):

- a. Trabajo más pequeño, técnico.
- b. Ejemplo: “*Crear endpoint /api/reservas*”.

2. Story / Task (Historia de usuario o tarea):

- a. Conjunto de tickets relacionados que entregan valor al usuario.
- b. Ejemplo: “*Como estudiante quiero reservar cupo en una ruta para asegurar transporte*”.
- c. Tickets dentro: diseño de pantalla, backend de reservas, pruebas de reservas.

3. Epic:

- a. Agrupación grande de **varias historias (y sus tickets)**.
- b. Ejemplo: “*Epic: Plataforma de reservas de rutas*”.
- c. Dentro: historias de ver rutas, reservar cupo, confirmar reservas.

4. Initiative (Iniciativa):

- a. Agrupación de varios **epics** que persiguen un objetivo estratégico.
- b. Ejemplo: “*Mejorar la movilidad de los estudiantes de la Sabana*”.

📌 Entonces:

- Un **ticket** es lo más pequeño (lo que alguien hace en 1–2 días).
- Una **agrupación de tickets** puede ser una **historia de usuario**.
- Varias historias forman un **epic**.
- Varios epics forman una **initiative**.

❖ ¿Qué es Swagger UI?

- Es una interfaz gráfica para probar y documentar tu **API REST**.
- Se genera a partir de un archivo **OpenAPI Specification (YAML o JSON)**.

- Permite que cualquiera (profes, testers, compañeros) pueda abrir un navegador y probar los endpoints: login, rutas, reservas, etc.

❖ Aplicado a Wheels Sabana

Tus **endpoints principales** (lo que ya definimos en el backlog) se verían así:

Auth

- POST /auth/register → Crear cuenta de estudiante.
- POST /auth/login → Iniciar sesión.
- GET /auth/me → Ver perfil autenticado.

Rutas (PRs)

- GET /rutas → Listar rutas fijas disponibles.
- POST /rutas → Crear nueva ruta (solo admin).
- GET /rutas/{id} → Ver detalles de una ruta.

Reservas

- POST /reservas → Crear una reserva (pasajero).
- GET /reservas/{id} → Ver reserva específica.
- PUT /reservas/{id}/confirmar → Conductor confirma reserva.
- PUT /reservas/{id}/rechazar → Conductor rechaza reserva.

Viajes

- GET /viajes/activos → Ver viajes en curso.
- PUT /viajes/{id}/iniciar → Conductor inicia viaje.
- PUT /viajes/{id}/finalizar → Conductor finaliza viaje.

Pagos (simulados)

- POST /pagos/{id} → Registrar pago (Efectivo/Nequi, solo confirmación).

Ejemplo de Swagger (OpenAPI YAML simplificado)

```
openapi: 3.0.0
info:
  title: Wheels Sabana API
  version: 1.0.0
paths:
  /auth/register:
    post:
      summary: Registrar usuario
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                email:
                  type: string
                password:
                  type: string
      responses:
        "201":
          description: Usuario registrado correctamente
  /rutas:
    get:
      summary: Listar rutas disponibles
      responses:
        "200":
          description: Lista de rutas
    post:
      summary: Crear ruta (solo admin)
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                origen:
                  type: string
                destino:
                  type: string
```

```
        type: string
    horario:
        type: string
responses:
    "201":
        description: Ruta creada
```

Beneficios de usar Swagger UI

- Tus profes verán que la API está **bien estructurada y documentada**.
- Los testers pueden probar cada endpoint sin Postman.
- Es más profesional: si luego escalas el proyecto, Swagger ya sirve como **manual técnico**.

Datos del usuario Pasajero

Obligatorios

1. **Nombre** (string) → Ej: *Maria José*
2. **Apellido** (string) → Ej: *Ramírez*
3. **ID Universidad** (string/int) → Código estudiantil asignado por La Sabana.
4. **Correo corporativo** (string) → Debe terminar en *@unisabana.edu.co*.
5. **Número de contacto** (string) → Para llamadas o WhatsApp en caso de emergencia.

Opcionales

6. **Foto de perfil** (URL o archivo) → Para confianza entre estudiantes.
7. **Contacto de emergencia** (nombre + teléfono) → Recomendado para seguridad.
8. **Método de pago preferido** (enum: *Efectivo, Nequi*) → Solo referencia, no transacción real.

Ejemplo de esquema en JSON (Swagger/OpenAPI style)

```
{  
    "nombre": "María José",  
    "apellido": "Ramírez",  
    "id_universidad": "20251234",  
    "correo_corporativo": "maria.ramirez@unisabana.edu.co",  
    "numero_contacto": "+57 3214567890",  
    "foto_perfil": "https://wheels-sabana.com/uploads/profiles/maria.jpg",  
    "contacto_emergencia": {  
        "nombre": "Laura Ramírez",  
        "telefono": "+57 3209876543"  
    },  
    "metodo_pago_preferido": "Nequi"  
}
```

Consideraciones de seguridad

- **Correo corporativo** → obligatorio para asegurar que solo entren estudiantes reales.
- **ID universidad** → permite validar contra una base institucional.
- **Número de contacto + emergencia** → crucial en caso de incidentes en viajes.

Datos del usuario Conductor

◊ Datos personales (los mismos del pasajero)

1. **Nombre** (string)
2. **Apellido** (string)
3. **ID Universidad** (string/int)
4. **Correo corporativo** (string, @unisabana.edu.co)
5. **Número de contacto** (string)
6. **Foto de perfil** (URL, opcional)
7. **Contacto de emergencia** (objeto: nombre + teléfono, opcional)
8. **Método de pago preferido** (*Efectivo, Nequi*)

◊ Datos del vehículo (obligatorios para conductor)

9. **Placa** (string) → Ej: ABC123.
10. **Foto del vehículo** (URL) → Para reconocimiento.
11. **Capacidad de pasajeros** (int) → Número de asientos disponibles.
12. **Marca** (string) → Ej: Mazda.
13. **Modelo** (string) → Ej: Mazda 3 2018.
14. **SOAT vigente** (boolean o fecha de vencimiento) → Para seguridad.

Ejemplo de esquema en JSON (Swagger/OpenAPI style)

```
{  
  "nombre": "Juan",  
  "apellido": "Pérez",  
  "id_universidad": "20253456",  
  "correo_corporativo": "juan.perez@unisabana.edu.co",  
  "numero_contacto": "+57 3104567890",  
  "foto_perfil": "https://wheels-sabana.com/uploads/profiles/juan.jpg",  
  "contacto_emergencia": {  
    "nombre": "Carlos Pérez",  
    "telefono": "+57 3206547890"
```

```
},
"metodo_pago_preferido": "Efectivo",
"vehiculo": {
    "placa": "ABC123",
    "foto_vehiculo": "https://wheels-sabana.com/uploads/vehiculos/juan\_carro.jpg",
    "capacidad": 4,
    "marca": "Mazda",
    "modelo": "Mazda 3 2018",
    "soat_vigente": "2025-12-31"
}
}
```

✓ Con esto, en la base de datos o en Swagger, puedes tener:

- **Rol pasajero** → solo info personal.
- **Rol conductor** → info personal + info vehículo.

❖ Lógica de roles en Wheels Sabana

- **Usuario (base)** → siempre existe (nombre, apellido, correo, id universidad, contacto, etc.).
- **Vehículo (opcional)** → si el usuario tiene un registro en esta tabla, automáticamente también es **conductor**.
- Si no tiene vehículo → solo puede actuar como **pasajero**.
- En una misma persona:
 - Puede reservar viajes como pasajero.
 - Puede ofrecer viajes como conductor (si tiene vehículo).

Modelo Entidad-Relación (simplificado)

Usuario

- id_usuario (PK)
- nombre
- apellido
- id_universidad
- correo_corporativo
- numero_contacto
- foto_perfil (opcional)
- contacto_emergencia (opcional)
- metodo_pago_preferido

Vehículo

- id_vehiculo (PK)
- id_usuario (FK → Usuario)
- placa
- foto_vehiculo
- capacidad
- marca
- modelo
- soat_vigente

Reserva

- id_reserva (PK)
- id_usuario (FK → Usuario, pasajero)
- id_viaje (FK → Viaje)
- estado (pendiente, confirmada, rechazada)

Viaje

- id_viaje (PK)
- id_conductor (FK → Usuario)
- id_vehiculo (FK → Vehículo)
- origen
- destino
- horario
- cupos_disponibles

- estado (activo, finalizado, cancelado)

👉 En este modelo:

- Todos los **usuarios son pasajeros** por defecto.
- Cuando un usuario tiene un **vehículo registrado**, también puede ser **conductor**.

🔧 Ajustes en el modelo

Usuario

- id_usuario (PK)
- nombre
- apellido
- id_universidad
- correo_corporativo
- numero_contacto
- foto_perfil (opcional)
- contacto_emergencia (opcional)
- metodo_pago_preferido

Vehículo

- id_vehiculo (PK)
- id_usuario (FK → Usuario)
- placa
- foto_vehiculo
- capacidad

- marca
- modelo
- soat_vigente

Relación Usuario ↔ Vehículo

- **1 Usuario puede tener varios Vehículos (1..N).**
- **Cada Vehículo pertenece a un único Usuario.**
- Restricción de negocio: **para ser conductor debe existir al menos 1 vehículo asociado.**

 Esto te da flexibilidad:

- Hay estudiantes que solo tendrán 1 carro.
- Pero si alguien tiene moto + carro, también puede registrar ambos.
- La app siempre validará: “*¿tienes al menos un vehículo registrado?*” antes de dejarlo crear un viaje como conductor.

Reglas de negocio importantes

◊ Usuarios

1. Todo usuario debe estar registrado con **correo corporativo** (@unisabana.edu.co) → validación para evitar externos.
2. Un **usuario siempre es pasajero por defecto**.
3. Para ser **conductor** el usuario debe tener al menos **un vehículo registrado y SOAT vigente**.
4. Un usuario puede tener varios vehículos, pero **solo puede usar uno por viaje activo**.

◊ Vehículos

5. Cada vehículo debe tener: placa única, marca, modelo, capacidad y SOAT válido.
6. La **capacidad mínima** es 1 pasajero extra (además del conductor).
7. No se pueden crear viajes con vehículos que tengan **SOAT vencido**.

◊ Rutas y Viajes

8. Las rutas son **predefinidas** (ej: desde ciertos barrios hacia la Universidad o viceversa).
9. Solo se puede **crear un viaje** en una ruta disponible.
10. Cada viaje tiene **horario fijo de salida** → no se puede cambiar una vez creado.
11. Un viaje solo se puede **iniciar** cuando el conductor confirma que está en el punto de salida.
12. Un conductor **no puede tener dos viajes activos al mismo tiempo**.
13. Los viajes tienen cupos limitados según la capacidad del vehículo.

◊ Reservas

14. Un pasajero puede **reservar un solo cupo por viaje**.
15. Una reserva tiene estados: **pendiente, confirmada, rechazada**.
16. Solo el **conductor** puede confirmar/rechazar reservas.
17. Si un pasajero cancela antes del viaje, el cupo se libera.

◊ Pagos

18. Los pagos se hacen **fuera de la plataforma** (efectivo o Nequi).
19. La app solo registra el método de pago como **informativo**, no procesa transacciones.

◊ Seguridad y Confianza

20. Todos los perfiles deben mostrar: nombre, foto y contacto.

21. Pasajeros y conductores deben poder **calificarse mutuamente** (1–5 estrellas o “ / ”).
22. Si un usuario recibe muchas malas calificaciones, puede ser **bloqueado**.

Con estas reglas, tu app no solo tiene un backend funcional, sino que muestra que pensaste en la **operación real y segura** en un entorno universitario.

2. Roles de Usuario en Wheels Sabana

◊ Registro

1. **Todos los usuarios** al registrarse entran como **Pasajeros** por defecto.
 - a. Datos: nombre, apellido, id_universidad, correo corporativo, número de contacto, etc.
2. Para convertirse en **Conductor**, el usuario debe:
 - a. Registrar al menos **un vehículo** con datos válidos (placa, marca, modelo, capacidad).
 - b. Tener **SOAT vigente**.

◊ Alternar roles

3. En la app, un usuario puede alternar entre:
 - a. **Modo Pasajero** → buscar rutas y reservar cupos.
 - b. **Modo Conductor** → crear y gestionar viajes en una ruta.
4. La opción de cambiar de rol se controla con un **switch en el perfil** o menú de usuario:
 - a. Si no tiene vehículo → el switch a *Conductor* debe estar bloqueado.
 - b. Si tiene vehículo → puede activarlo/desactivarlo libremente.

❖ **Restricciones**

5. Un usuario **no puede estar en ambos roles al mismo tiempo en un mismo viaje.**
 - a. Ejemplo: si es conductor en un viaje activo, no puede reservarse a sí mismo como pasajero.
6. Un usuario **sí puede usar distintos roles en diferentes momentos.**
 - a. Ejemplo: puede ser pasajero en la mañana y conductor en la tarde.

📋 **Ejemplo visual (en la app)**

- Perfil de Juan Pérez → **Rol actual: Pasajero**
- Botón: “*Cambiar a Conductor*”
 - Si tiene vehículo registrado → se habilita.
 - Si no → aparece mensaje: “*Registra tu vehículo para activar el modo Conductor*”.

✓ Con esto, en tu modelo de datos solo necesitas **un tipo de usuario**, y el rol depende de si tiene vehículo y qué acción está ejecutando en ese momento.

❖ **Regla de Roles en Wheels Sabana**

1. **Todos los usuarios registrados** son **Pasajeros** por defecto.
 - a. Pueden buscar rutas y reservar cupos.
 - b. No necesitan vehículo registrado.
2. **Un usuario solo puede ser Conductor si tiene al menos un vehículo registrado** (placa, marca, modelo, capacidad, SOAT vigente).
3. **Todo Conductor es automáticamente también Pasajero.**
 - a. Puede usar la app en ambos roles:
 - i. Conductor → crear y gestionar viajes.
 - ii. Pasajero → reservar viajes de otros conductores.
4. **No todos los Pasajeros son Conductores.**

- a. Si un usuario no tiene vehículo, nunca podrá crear un viaje.
5. Restricción lógica:
- a. En un **viaje activo**, el usuario no puede estar en doble rol (ej: ser el conductor de un viaje y al mismo tiempo pasajero de ese mismo viaje).
 - b. Sí puede alternar roles en diferentes viajes.

Ejemplo

- **Juan** → Tiene vehículo registrado → Rol dinámico (Conductor + Pasajero).
- **María** → No tiene vehículo registrado → Solo puede ser Pasajera.

 Con este enfoque, en tu modelo de datos solo necesitas **una tabla de Usuario**, y el rol depende de si tiene **vehículo asociado o no**.

Registro de Viajes (Conductor)

Datos obligatorios del viaje

1. **id_viaje** → Identificador único (PK).
2. **id_conductor** → Usuario que publica el viaje (FK a Usuario).
3. **id_vehiculo** → Vehículo asociado al viaje (FK a Vehículo).
4. **punto_inicio** → Dirección o coordenadas de salida.
5. **punto_final** → Dirección o coordenadas de llegada.
6. **ruta** → Puede ser:
 - a. Predefinida por la universidad (ej: *Cajicá* → *UniSabana*)
 - b. O generada por Google Maps/Waze.
7. **hora_salida** → Fecha y hora programada.
8. **tarifa_por_pasajero** → Valor que cada pasajero debe pagar (efectivo o Nequi).
9. **puestos_disponibles** → Cupos totales disponibles (\leq capacidad del vehículo).

Datosopcionales

10. **descripcion** → Comentarios extra del conductor (ej: “*Paso por la Av. 80*”).

11. **estado** → pendiente, activo, finalizado, cancelado.



Ejemplo en JSON (Swagger/OpenAPI style)

```
{  
    "id_viaje": 101,  
    "id_conductor": 15,  
    "id_vehiculo": 3,  
    "punto_inicio": "Calle 10 #25-30, Cajicá",  
    "punto_final": "Universidad de La Sabana, Chía",  
    "ruta": "Cajicá - Chía (via Variante Cota)",  
    "hora_salida": "2025-09-10T07:00:00",  
    "tarifa_por_pasajero": 5000,  
    "puestos_disponibles": 3,  
    "descripcion": "Paso por el centro comercial Fontanar",  
    "estado": "pendiente"  
}
```

❖ Reglas de negocio específicas

1. El **número de puestos disponibles** no puede superar la capacidad registrada del vehículo.
2. La **hora de salida** debe ser mayor a la fecha y hora actual (no se crean viajes en el pasado).
3. El **estado inicial** de un viaje es *pendiente*.
4. Cuando el conductor inicia el recorrido, el viaje pasa a *activo*.
5. Cuando finaliza, el viaje pasa a *finalizado*.
6. Un conductor no puede tener **dos viajes activos** al mismo tiempo.

Con esto ya tienes bien definido el **modelo de viajes** que se registran en tu app.

❖ 1. Tarifa definida por el conductor

- El conductor decide cuánto cobrar por pasajero.
- Pros:
 - Flexible.
 - Se ajusta a costos reales (gasolina, peajes).
- Contras:
 - Riesgo de que algunos cobren demasiado.
 - Inconsistencia entre viajes.

👉 Regla de negocio:

- Se podría poner un **mínimo y máximo autorizado por la universidad** (ej: \$3.000 – \$10.000 COP por trayecto).
- Así evitas abusos y mantienes un estándar.

❖ 2. Tarifa automatizada (calculada por la app)

- La app calcula la tarifa usando variables como:
 - **Distancia** (Google Maps / Waze API).
 - **Tiempo estimado de viaje**.
 - **Capacidad del vehículo** (para dividir costos).
- Fórmula ejemplo:

$$\text{Tarifa_por_pasajero} = \frac{\text{Costo_estimado_viaje} \times \text{Número_de_puestos}}{\text{Número_de_puestos}}$$
$$\text{Tarifa_por_pasajero} = \frac{\text{Costo_estimado_viaje}}{\text{Número_de_puestos}}$$

Donde **Costo_estimado_viaje** = (distancia * costo_por_km).

👉 Ejemplo:

- 12 km * \$500/km = \$6.000.
- Vehículo con 3 cupos → tarifa sugerida = \$2.000 por pasajero.

Estrategia híbrida (recomendada)

1. La **app propone una tarifa automática** (basada en distancia).
2. El conductor puede **ajustarla dentro de un rango permitido** (ej: ±20%).
3. Esto da:
 - a. Transparencia para pasajeros.
 - b. Flexibilidad para conductores.
 - c. Control institucional para evitar abusos.

 Entonces:

- Sí, **se puede automatizar** con Google Maps/Waze.
- Pero lo más práctico para universidad es un **sistema híbrido**: sugerencia automática + confirmación del conductor dentro de límites establecidos.

Uber-style (API Uber / Lyft)

- Pensado para **trayectos punto a punto**.
- Puedes añadir **stops intermedios**, pero no se manejan como “paradas oficiales”, sino como *waypoints* (puntos de pausa dentro del viaje).
- Ejemplo: Cajicá → UniSabana, con un stop en Fontanar.

 En este modelo:

- El viaje es **personalizado** para los pasajeros que suban.
- No existe una ruta fija ni “estaciones oficiales”.

TransMilenio-style (API TransMilenio / GTFS feeds)

- Pensado para **rutas fijas con estaciones/paradas predefinidas**.
- Cada ruta tiene:
 - **Troncales** (líneas principales).
 - **Paradas oficiales** (puntos donde los usuarios pueden subir/bajar).
- Ejemplo:
 - Ruta Cajicá → UniSabana.
 - Paradas: *Centro Cajicá → Fontanar → Variante Cota → Puerta UniSabana*.

 En este modelo:

- Los viajes siempre siguen el **mismo recorrido**.
- Los pasajeros pueden reservar desde un punto de origen hasta un destino **dentro de las paradas oficiales**.

¿Cómo aplicarlo en Wheels Sabana?

1. **Si usas estilo Uber** → conductor define libremente su viaje y puede añadir stops (flexible, pero caótico a gran escala).
2. **Si usas estilo TransMilenio** → definen **rutas fijas oficiales** (PRs que mencionabas) y todos los viajes deben adherirse a una de esas rutas.
 - a. Ejemplo:
 - i. **Ruta 1:** Cajicá – Fontanar – UniSabana.

ii. **Ruta 2:** Chía Centro – Centro Comercial Sabana Norte – UniSabana.

👉 Esto último **es ideal para universidad** porque:

- Aumenta la seguridad.
- Evita viajes improvisados.
- Facilita reservas, porque los pasajeros saben dónde pueden subirse.

✓ Mi recomendación:

- **Combinar ambos modelos** →
 - Base fija tipo **TransMilenio** (rutas + paradas oficiales).
 - Con opción de que el conductor **agregue un par de stops adicionales** si los pasajeros lo acuerdan (Uber style).

❖ 4. Listado de Conductores y Viajes Disponibles

◊ Reglas de negocio

1. El pasajero debe poder ver **todos los viajes activos y futuros** publicados por conductores.
2. Cada viaje debe mostrar los siguientes datos:
 - a. **Conductor:** nombre + foto + calificación promedio.
 - b. **Punto de inicio** (ej: *Cajicá Centro*).
 - c. **Punto final** (ej: *Universidad de La Sabana*).
 - d. **Ruta** (ej: *Cajicá → Fontanar → UniSabana*).
 - e. **Cupos disponibles** (ej: 2/4).
 - f. **Hora de salida** (ej: 07:00 AM).
 - g. **Tarifa por pasajero** (ej: \$5.000 COP).
3. Los viajes deben filtrarse por:
 - a. **Fecha y hora de salida**.
 - b. **Origen y destino** (paradas oficiales o cercanas).
 - c. Opcional: **tarifa máxima que el pasajero esté dispuesto a pagar**.
4. Solo se muestran viajes que **todavía tienen cupos disponibles**.

◊ Ejemplo en JSON (respuesta API)

```
[  
  {  
    "id_viaje": 101,  
    "conductor": {  
      "nombre": "Juan Pérez",  
      "foto": "https://wheels.com/perfiles/juan.jpg",  
      "calificacion": 4.8  
    },  
    "punto_inicio": "Cajicá Centro",  
    "punto_final": "Universidad de La Sabana",  
    "ruta": "Cajicá - Fontanar - UniSabana",  
    "hora_salida": "2025-09-10T07:00:00",  
    "cupos_disponibles": 2,  
    "tarifa_por_pasajero": 5000  
  },  
  {  
    "id_viaje": 102,
```

```

"conductor": {
    "nombre": "María Ramírez",
    "foto": "https://wheels.com/perfiles/maria.jpg",
    "calificacion": 4.5
},
"punto_inicio": "Chía Centro",
"punto_final": "Universidad de La Sabana",
"ruta": "Chía - Variante Cota - UniSabana",
"hora_salida": "2025-09-10T06:30:00",
"cupos_disponibles": 1,
"tarifa_por_pasajero": 4000
}
]

```

❖ Vista en la aplicación (para Figma)

- **Listado tipo cards o tabla:**
 - Foto y nombre del conductor.
 - Ruta resumida (*Inicio → Final*).
 - Icono  para vehículo.
 - Reloj  para hora de salida.
 -  Cupos disponibles.
 -  Tarifa.
- Botón “**Reservar cupo**” en cada card.
- Filtros arriba: *Fecha | Origen | Destino | Tarifa máxima*.

 Con esto, el pasajero tiene toda la info que necesita para tomar la decisión.

❖ Sistema de Calificación y Reseñas

◊ Reglas de negocio

1. **Escala de calificación:** 1 a 5 estrellas ★.
2. Solo los pasajeros que **tomaron un viaje** pueden calificar al conductor.
3. Opcional: los conductores también pueden calificar a los pasajeros (como en Uber/Didi).
4. Cada calificación puede tener:
 - a. Puntaje (ej: ★ ★ ★ ★ ☆ = 4).
 - b. Comentario opcional (ej: “*Muy puntual y amable*”).
5. El promedio de calificaciones se muestra en el **perfil del conductor** y en el **listado de viajes**.
6. Para garantizar transparencia:
 - a. Los comentarios no se pueden editar, solo eliminar si hay reporte.
 - b. La universidad podría tener un **moderador** que revise quejas o reseñas ofensivas.

◊ Ejemplo en JSON (calificación)

```
{  
  "id_calificacion": 202,  
  "viaje_id": 101,  
  "pasajero_id": 301,  
  "conductor_id": 501,  
  "puntaje": 5,  
  "comentario": "Excelente viaje, conductor puntual y amable.",  
  "fecha": "2025-09-09T09:00:00"  
}
```

◊ Ejemplo de visualización (UI/Figma)

- En el **listado de viajes**:
 - Foto del conductor.
 - Nombre.
 - ★ 4.8 (basado en 56 reseñas).

- En el **perfil del conductor**:
 - Promedio de calificación.
 - Historial de comentarios (paginado).
 - Botón “**Reportar**” si el comentario es ofensivo.

 Beneficio:

- Genera confianza → los pasajeros eligen conductores bien calificados.
- Motiva a los conductores a dar buen servicio.
- Permite a la universidad detectar problemas.

❖ 5. Selección de Viaje para Pasajeros

◊ Reglas de negocio

1. El pasajero puede **elegir un viaje disponible** del listado.
2. Al abrir el viaje, el sistema debe mostrar:
 - a. Datos del conductor (nombre, foto, calificación).
 - b. Vehículo (marca, modelo, placa, capacidad).
 - c. Ruta completa en el mapa.
 - d. Punto de inicio y punto final.
 - e. Hora de salida.
 - f. Cupos disponibles.
 - g. Tarifa por pasajero.
3. El pasajero puede:
 - a. Seleccionar **cuántos cupos reservar** (ej: 1, 2 o más si hay disponibles).
 - b. Definir el **punto de recogida** para cada cupo (ej: *Portería Fontanar, Chía Centro*).
 - c. Confirmar la reserva.
4. Regla de validación:
 - a. No se puede reservar más cupos de los disponibles.
 - b. Si se reservan varios cupos, se deben asociar a diferentes pasajeros (ej: si alguien reserva para un amigo).
5. Una vez confirmada la reserva:
 - a. Se descuenta el número de cupos del viaje.
 - b. El pasajero recibe un **ticket de reserva** con los datos del viaje.
 - c. El conductor recibe una **notificación** con los pasajeros y puntos de recogida.

◊ Ejemplo en JSON (reserva de viaje)

```
{  
  "id_reserva": 601,  
  "viaje_id": 101,  
  "pasajero_id": 301,  
  "cupos_reservados": 2,  
  "puntos_recogida": [  
    "Parque principal de Cajicá",  
    "Fontanar Centro Comercial"  
  ],  
}
```

```
"estado": "confirmada",
"fecha_reserva": "2025-09-09T10:15:00"
}
```

❖ Ejemplo UX/UI (pantalla Figma)

1. Detalle del viaje (card o pantalla):

- a. Foto del conductor.
- b. Ruta en mapa.
- c. Hora de salida.
- d. Cupos disponibles.
- e. Precio por pasajero.

2. Sección de selección:

- a. Dropdown o input numérico: “*¿Cuántos cupos quieres reservar?*”.
 - b. Para cada cupo → campo: “*Selecciona punto de recogida*” (lista de paradas predefinidas en la ruta).
3. Botón CTA: “Confirmar reserva 

Beneficio:

- Flexibilidad para que los pasajeros elijan dónde los recogen.
- Claridad en la cantidad de cupos reservados.
- Transparencia entre pasajeros y conductores.

🔗 6. Bloqueo de Viajes Llenos

◊ Reglas de negocio

1. Cada viaje tiene un **campo de cupos disponibles**.
2. Cuando un pasajero reserva, el sistema:
 - a. Resta el número de cupos reservados al total disponible.
 - b. Actualiza el viaje en tiempo real.
3. Si los cupos = **0** → el viaje cambia de estado a:
 - a. “**Lleno**” (visible, pero ya no se puede reservar).
 - b. O “**No disponible**” (se oculta de la lista pública, dependiendo de la política de UX).
4. Una vez marcado como lleno:
 - a. Se bloquea cualquier nueva reserva.
 - b. El botón “**Reservar**” queda deshabilitado.
 - c. O se muestra un mensaje: “*Viaje lleno, no hay cupos disponibles*”.
5. Notificación automática al conductor:
 - a. “*Tu viaje ya está lleno. No se aceptan más pasajeros.*”
6. Opcional: permitir lista de espera.
 - a. Pasajeros interesados pueden inscribirse.
 - b. Si alguien cancela, el primer inscrito recibe notificación.

◊ Ejemplo en JSON (viaje lleno)

```
{  
  "id_viaje": 101,  
  "conductor_id": 501,  
  "estado": "lleno",  
  "cupos_totales": 4,  
  "cupos_disponibles": 0,  
  "pasajeros_reservados": [301, 302, 303, 304]  
}
```

◊ Ejemplo UX/UI (pantalla Figma)

- En la **lista de viajes**:
 - Card con etiqueta  “**Lleno**”.

- Botón de reservar → deshabilitado.
- En el **detalle del viaje**:
 - Mensaje: “Este viaje ya no tiene cupos disponibles”.
 - Opción: “Unirme a la lista de espera” (si se implementa).

 Beneficio:

- Evita sobreventa.
- Transparencia para el pasajero.
- Menos conflictos con el conductor.

❖ 7. Filtros de Búsqueda para Pasajeros

◊ Reglas de negocio

1. El sistema debe permitir a los pasajeros **buscar y filtrar viajes** según:
 - a. **Punto de salida** (ej: Cajicá, Chía, Zipaquirá, Fontanar, etc.).
 - b. **Cupos disponibles** (ej: mínimo 1, 2 o más cupos).
 - c. Opcional:
 - i. **Hora de salida** (ej: mañana, tarde, noche).
 - ii. **Tarifa máxima** (ej: hasta \$5.000).
 - iii. **Calificación mínima del conductor** (ej: 4.0+).
2. La búsqueda debe ejecutarse en tiempo real sobre la base de datos de viajes activos.
3. Los filtros se pueden **combinar** (ej: “*Viajes desde Chía con mínimo 2 cupos disponibles antes de las 8:00 AM*”).
4. Si no hay resultados:
 - a. Mostrar mensaje: “*No se encontraron viajes con esos filtros*”.
 - b. Ofrecer opción: “*Unirme a una lista de espera*” o “*Recibir notificación cuando se publique un viaje similar*”.

◊ Ejemplo en JSON (consulta de filtros)

👉 Petición del pasajero:

```
{  
  "punto_salida": "Cajicá",  
  "cupos_minimos": 2,  
  "hora_maxima": "2025-09-09T08:00:00"  
}
```

👉 Respuesta del sistema:

```
[  
  {  
    "id_viaje": 202,  
    "conductor": "Laura Gómez",  
    "punto_inicio": "Cajicá Centro",  
    "punto_final": "Universidad de La Sabana",  
    "hora_salida": "2025-09-09T08:00:00",  
    "tarifa_maxima": 5000  
  }]
```

```
        "hora_salida": "2025-09-09T07:30:00",
        "cupos_disponibles": 3,
        "tarifa_por_pasajero": 4000
    }
]
```

❖ Ejemplo UX/UI (pantalla Figma)

- Barra de búsqueda + filtros arriba del listado:
 - **Dropdown “Punto de salida”** (lugares predefinidos).
 - **Selector de cupos** (ej: “Al menos 1 / 2 / 3+ cupos”).
 - Opcionales:
 - **Slider de tarifa máxima** 💰.
 - **Filtro de calificación mínima del conductor** ⭐.
 - **Selector de rango de hora de salida** 🕒.
- Resultados actualizados dinámicamente al aplicar filtros.

Beneficio:

- Los pasajeros encuentran más fácil el viaje que se ajusta a su necesidad.
- Se optimiza la ocupación de cupos en viajes cercanos.
- El conductor recibe pasajeros que realmente se ajustan a su ruta/horario.

❖ Sistema de Notificaciones en Wheels Sabana

◊ Reglas de negocio

1. Eventos que generan notificaciones

- a. **Confirmación de reserva** → pasajero recibe ticket digital.
- b. **Viaje lleno** → conductor recibe aviso y el botón de reservar se bloquea.
- c. **Cancelación de viaje por el conductor** → todos los pasajeros reciben alerta inmediata.
- d. **Cancelación de reserva por el pasajero** → conductor recibe notificación (y se libera cupo).
- e. **Recordatorio de viaje** → pasajeros y conductor reciben aviso 30–60 min antes.
- f. **Cambio en ruta u hora** (si el conductor actualiza detalles).
- g. **Llegada al punto de recogida** (opcional, con GPS/Google Maps API).

◊ Tipos de notificaciones

- **Push notifications** (si la app se convierte en móvil).
- **Notificaciones dentro de la web app** (campanita  con historial).
- **Correo electrónico corporativo** (ej: confirmación de reserva).
- **SMS/WhatsApp opcional** (para emergencias o recordatorios rápidos).

◊ Ejemplo en JSON (notificación API)

```
{  
  "id_notificacion": 701,  
  "usuario_id": 301,  
  "tipo": "cancelacion_viaje",  
  "mensaje": "El viaje de Juan Pérez programado para 07:30 AM ha  
  sido cancelado.",  
  "fecha": "2025-09-09T11:00:00",  
  "estado": "no_leida"  
}
```

◊ Ejemplo UX/UI (pantalla Figma)

- Ícono de campana  en la barra superior.
- Lista de notificaciones con iconos:
 -  Confirmación.
 -  Cancelación.
 -  Recordatorio.
- Colores según tipo (verde = confirmación, rojo = cancelación, azul = recordatorio).
- Botón: “*Marcar como leídas*”.

◊ Beneficio

- Evita confusiones entre pasajeros y conductores.
- Mejora la seguridad (todos están al tanto de cambios).
- Genera confianza y profesionalismo en la app.

❖ Puntos de salida/entrada en la Universidad

❖ Accesos principales

1. **Puente Madera** → salida hacia Bogotá / Autopista Norte.
2. **Ad Portas** → salida hacia Chía y Cajicá.
3. (Opcional) **Parqueadero principal / interno** → para viajes cortos dentro de la U o recogida puntual.

❖ Reglas de negocio

1. El **conductor**, al registrar un viaje, debe seleccionar **desde qué portería de la U** sale o entra.
2. Los **pasajeros**, al reservar, deben poder filtrar también por **punto de salida específico de la U**.
 - a. Ejemplo: “Quiero un viaje que salga de Ad Portas, no de Puente Madera”.
3. En el **detalle del viaje**, debe mostrarse claramente:
 - a. “Punto de salida: Universidad de La Sabana – Ad Portas”.
4. Para integrarlo con **Google Maps / Waze**, cada punto debe tener coordenadas predefinidas:
 - a. **Puente Madera**: Lat, Long.
 - b. **Ad Portas**: Lat, Long.

❖ Ejemplo en JSON (puntos de salida de la U)

```
{  
  "puntos_entrada_universidad": [  
    {  
      "id": 1,  
      "nombre": "Puente Madera",  
      "lat": 4.864201,  
      "lng": -74.033512  
    },  
    {  
      "id": 2,  
      "nombre": "Ad Portas",  
      "lat": 4.862730,  
      "long": -74.033512  
    }  
  ]}
```

```
        "lng": -74.029841
    }
]
}
```

❖ UX/UI en Figma

- En la **creación de viaje** (para conductor):
 - Dropdown: “*Selecciona portería de salida/entrada*” → Puente Madera / Ad Portas.
- En la **búsqueda de viajes** (para pasajero):
 - Filtro adicional: “*Portería de salida*”.
- En el **detalle del viaje**:
 -  *Salida: Universidad de La Sabana – Puente Madera.*

 Esto evita confusiones (porque un pasajero que espera en Ad Portas puede perder el viaje si el carro sale por Puente Madera).

Requisitos no funcionales (Wheels Sabana)

1) Usabilidad / UI

- **Interfaz:** Intuitiva y fácil de usar; flujo claro para registrar>buscar>reservar.
- **Responsive:** Soporte completo desde 320px (móviles) hasta 1920px (escritorio).
- **Accesibilidad básica:** contraste legible, etiquetas ARIA en componentes críticos, navegación por teclado.
- **PWA:** considerar versión PWA para experiencia casi nativa (offline básico, push).

2) Rendimiento

- **Objetivo de carga:**
 - **Interactividad inicial (Time to Interactive, TTI):** < 2 s en condiciones móviles típicas; ideal < 1 s en conexiones decentes.
 - **Primer contenido visible (LCP):** < 2.5 s; objetivo < 1.5 s para páginas críticas (home, lista de viajes).
 - **API response (páginas/datos):** endpoints críticos (listar viajes, reservar) < 200–500 ms medianos; picos ≤ 1 s.
- **Técnicas:**
 - Server-Side Rendering (SSR) para páginas principales (Next.js/ Nuxt).
 - Lazy-load de mapas y marcadores; cargar mapa ligero y agregar marcadores on-demand o clusters.
 - CDN (Cloudflare/Cloud CDN) para assets y caché de recursos estáticos.
 - Caching a nivel API (Redis) y HTTP caching (Cache-Control, ETags).
 - Minimizar requests: bundling, tree-shaking, compresión gzip/ Brotli, imágenes optimizadas (AVIF/WebP).
 - DB: consultas optimizadas e índices; paginación en listados.
 - Monitorización de performance (RUM + synthetic tests).

3) Escalabilidad y disponibilidad

- **Disponibilidad objetivo (SLA):** 99% uptime mínimo.
- **Escalado:**

- Infraestructura en la nube con auto-scaling (AWS/GCP/Azure) y despliegue multi-AZ.
- Separación de capas: frontend (CDN + app servers), API (stateless), bases de datos (replicación read replicas), cache (Redis), storage (object storage).
- **Resiliencia:**
 - Health checks y auto-recovery.
 - Circuit breakers y retries en llamadas a terceros (Google Maps, Waze, etc.).
- **Mantenimiento mínimo:**
 - Ventanas de mantenimiento programadas fuera de horas pico; degradación gracia (modo solo lectura si DB principal no disponible).

4) Seguridad y privacidad

- **Protección de datos personales:**
 - Cumplir con normativa aplicable (Colombia: Habeas Data / buenas prácticas) — almacenar solo lo necesario.
 - Política de retención y eliminación de datos.
- **Autenticación & Contraseñas:**
 - Contraseñas: hashing con **bcrypt** o **Argon2** (parámetros fuertes).
 - Autenticación: OAuth2 / JWT con expiración y refresh tokens seguros; forzar verificación por correo institucional.
- **Transmisión y almacenamiento:**
 - TLS 1.2+ obligatorio para todo tráfico.
 - Encriptación en reposo para datos sensibles (DB columnas: teléfono, documento).
- **Control de acceso:**
 - Roles y permisos (usuario, admin, moderador).
 - Registro de auditoría para acciones críticas (cancelaciones, cambios en viajes).
- **Protección frente a ataques:**
 - Rate limiting por IP / usuario en endpoints sensibles (login, reservas).
 - WAF (Web Application Firewall) y protección contra bots.
 - Escaneo SAST/DAST en pipelines (Snyk, OWASP tools).
- **Backups & recuperación:**
 - Backups automáticos diarios, con retención ≥ 30 días y pruebas de restore periódicas.
 - Plan DR (disaster recovery) con RTO/RPO definidos.

5) Compatibilidad

- **Browsers:** Chrome, Firefox, Edge, Safari (últimas 2 versiones).
- **Dispositivos:** iOS y Android (navegadores móviles).
- **Integraciones de terceros:** tolerancia a errores y degradación cuando APIs externas (Google Maps, Waze) fallen.

6) Observabilidad y operación

- **Logging:** trazabilidad estructurada (JSON) con request IDs.
- **Monitorización:**
 - Métricas infra + app (Prometheus / CloudWatch), dashboards (Grafana).
 - Error tracking (Sentry).
 - Uptime & synthetic tests (Pingdom / NewRelic / Datadog).
- **Alertas:** umbrales claros (latencia API, error rate, DB CPU, Redis memory).
- **Testing:**
 - Unit, integration, end-to-end (Cypress / Playwright).
 - Pruebas de carga y estrés pre-release (k6, JMeter).
- **CI/CD:**
 - Pipelines automáticos con pruebas y despliegue canary/blue-green para minimizar downtime.

7) Operación con APIs de mapas y terceros

- **Uso responsable:** cachear resultados geocoding / rutas para consultas repetidas.
- **Rate limits:** anticipar cuotas y manejar errores; plan de pago/ escala para Google Maps cuando suba tráfico.
- **Latencia:** reducir llamadas síncronas bloquantes; usar llamadas asíncronas y pre-calculadas si es posible (tarifas sugeridas, ETA).

8) Rendimiento en mapas / listados (específico)

- Clustering de marcadores cuando hay muchos viajes en área.
- Limitar número de marcadores iniciales y paginar o cargar por viewport.
- Precalcular rutas y distancias para la tarificación automatizada.

KPIs y metas medibles

- TTFB promedio de la API: < 200–500 ms.
- P90 API latency: < 1 s.
- LCP página crítica: < 2.5 s (ideal < 1.5 s).
- Error rate (5xx): < 0.1% en producción.
- Uptime: ≥ 99% mensual.
- Tiempo de recuperación de incidente crítico (RTO): < 2 horas.

Recomendaciones tecnológicas (rápido)

- **Frontend:** React + Next.js (SSR/SSG), TypeScript, Tailwind. PWA.
- **Backend:** Node.js (NestJS/Express) o Python (FastAPI/Django Rest) — TypeScript o Python según equipo.
- **DB:** PostgreSQL (ACID) + Redis (cache, locks).
- **Auth:** Firebase Auth (si quieres simplificar) o Auth0 / OAuth2 + JWT.
- **Hosting:** AWS (ECS/EKS/Elastic Beanstalk) o GCP (Cloud Run/GKE) con autoscaling.
- **CDN:** Cloudflare / Cloud CDN.
- **Observability:** Prometheus + Grafana, Sentry, ELK/Cloud logging.

Checklist práctico (para poner en Trello / Sprint)

- Definir SLAs y métricas de rendimiento.
- Implementar SSR y lazy-load de mapas.
- Integrar Redis para cache de rutas y listados.
- Implementar hashing de contraseñas (Argon2/bcrypt).
- TLS en todos los endpoints; configurar secretos en vault.
- Añadir rate limiting y WAF básico.
- Configurar backups automáticos y pruebas de restore.
- Crear pipelines CI/CD con pruebas unit/integration/e2e.

- Configurar monitorización y alertas (latencia, errors, health).
- Realizar pruebas de carga (k6) y ajustar autoscaling.
- Preparar plan de degradación (sin mapas funciona el listado).

❖ Integraciones de APIs para Wheels Sabana

◊ 1. Google Maps Platform

- **Usos:**
 - Geocoding (convertir direcciones ↔ coordenadas).
 - Distance Matrix (cálculo de distancias y tiempos entre puntos).
 - Directions API (rutas optimizadas con paradas).
 - Mapas embebidos en la app.
- **Beneficio:** rutas confiables, mapas detallados y cálculo de tarifas base automático.

◊ 2. Waze API (Waze for Broadcasters / Waze Deep Links)

- **Usos:**
 - Mostrar rutas en tiempo real con tráfico actualizado.
 - Redirigir a la app de Waze para navegación del conductor.
- **Beneficio:** viajes más precisos y puntuales, considerando tránsitos.

◊ 3. Uber API (Ride Request & Trip Experiences)

- **Usos:**
 - Inspirarse en funcionalidades de multi-stop (paradas intermedias).
 - Calcular tiempos estimados de llegada (ETA).
 - Opcional: integración para comparar costo de Uber vs Wheels.
- **Beneficio:** mejores estimaciones de viaje y referencia de calidad.

◊ 4. API TransMilenio / SITP (Datos abiertos Bogotá)

- **Usos:**
 - Integrar troncales, estaciones y rutas oficiales.
 - Permitir a pasajeros combinar *Wheels + TransMilenio* (ej: bajarse en un punto intermedio y seguir en bus).
- **Beneficio:** complementa la movilidad universitaria con transporte público.

◊ 5. Nequi / Daviplata (Pagos)

- **Usos:**
 - Registrar pagos o confirmar transferencias (si en el futuro quieres formalizar).
 - APIs de billeteras digitales colombianas.
- **Beneficio:** en el futuro puedes automatizar el pago en vez de depender solo de efectivo.

◊ 6. Firebase (Notificaciones push + Auth)

- **Usos:**
 - Autenticación segura con correo institucional.
 - Notificaciones en tiempo real (cancelación, viaje lleno, recordatorios).
- **Beneficio:** reduce tiempo de implementación, confiable y escalable.

◊ 7. Twilio / WhatsApp API (opcional)

- **Usos:**
 - Enviar confirmaciones o emergencias por WhatsApp/SMS.
- **Beneficio:** comunicación directa en caso de cancelación urgente.

◊ 8. Weather API (OpenWeather / IDEAM)

- **Usos:**
 - Mostrar clima en ruta (ej: “lluvia en Bogotá, lleva sombrilla”).
- **Beneficio:** experiencia extra para el pasajero.

◊ 9. Universidad de La Sabana API interna (si existe / SSO)

- **Usos:**
 - Validar identidad con el correo institucional.

- Integrar con carnet digital / base de estudiantes y profesores.
- **Beneficio:** seguridad y evitar registros falsos.

 Con estas integraciones, tu app sería:

- **Inteligente en rutas** (Google + Waze + Uber).
- **Compatible con transporte público** (TransMilenio).
- **Confiable y segura** (Firebase, auth institucional).
- **Escalable a futuro con pagos digitales** (Nequi/Daviplata).
- **Adaptada al contexto local** (clima, notificaciones por WhatsApp).

🔗 Bases de datos NoSQL recomendadas

◊ 1. MongoDB Atlas (en la nube)

- **Modelo:** Documentos (JSON-like).
- **Ventajas:**
 - Flexible: puedes guardar usuarios, viajes, vehículos, etc. en colecciones diferentes.
 - Escalable horizontalmente (sharding → soporta miles de usuarios).
 - Integración directa con Node.js, Express y APIs REST/GraphQL.
 - Tiene **MongoDB Realm** para sincronización en tiempo real.
- **Uso en tu caso:**
 - Colección usuarios (datos básicos, rol pasajero/conductor).
 - Colección vehiculos (asociados a un conductor).
 - Colección viajes (rutas, cupos, tarifas, estado).
 - Colección reservas (qué pasajero reservó qué viaje).

◊ 2. Firebase Firestore

- **Modelo:** Documentos / colecciones.
- **Ventajas:**
 - Tiempo real nativo (ideal para notificaciones de cambios en viajes/cupos).
 - Autenticación integrada (con correo institucional, Google, etc.).
 - Escalable sin configuración complicada.
 - SDK fácil de integrar en frontend web y móvil.
- **Uso en tu caso:**
 - Perfecto para sincronizar **cupos disponibles en tiempo real**.
 - Notificaciones automáticas cuando un viaje se cancela o se llena.

◊ 3. Cassandra / DataStax Astra (opcional, si crece mucho)

- **Modelo:** Wide-column store.
- **Ventajas:**
 - Excelente para sistemas que requieren **altísima disponibilidad** (99.99%).

- Usada por apps grandes como Uber y Netflix.
- **Uso en tu caso:**
 - Si en el futuro la app escala a **miles de viajes/segundo**, es una opción de nivel empresarial.

💡 Recomendación práctica para Wheels Sabana

- 👉 Para arrancar en universidad:
 - ✓ **Firestore** (por su facilidad + tiempo real + autenticación).
- 👉 Si luego quieres algo más robusto:
 - ✓ **MongoDB Atlas** (te da más control y flexibilidad de modelo de datos).

📁 Ejemplo de modelo de datos (MongoDB / Firestore)

```
// Usuario
{
  "id": "u123",
  "nombre": "Juan",
  "apellido": "Pérez",
  "correo": "juan.perez@unisabana.edu.co",
  "rol": "conductor",
  "telefono": "3001234567",
  "vehiculos": ["v001", "v002"]
}

// Vehículo
{
  "id": "v001",
  "placa": "ABC123",
  "marca": "Kia",
  "modelo": "Picanto 2019",
  "capacidad": 4,
  "soat": "2025-10-01",
  "id_conductor": "u123"
}
```

```
// Viaje
{
    "id": "viaje001",
    "id_conductor": "u123",
    "vehiculo": "v001",
    "origen": "Universidad de La Sabana - Puente Madera",
    "destino": "Portal Norte",
    "hora_salida": "2025-09-09T07:30:00",
    "tarifa": 4000,
    "cupos_disponibles": 3,
    "estado": "activo"
}

// Reserva
{
    "id": "r001",
    "id_viaje": "viaje001",
    "id_pasajero": "u567",
    "cupos_reservados": 2,
    "punto_recogida": "Chía - Centro Comercial Fontanar"
}
```

 Así tienes todo separado, escalable, y fácil de consultar.

Modelo de colecciones en MongoDB (Wheels Sabana)

Te propongo algo así:

Usuarios

```
{  
    "_id": "u123",  
    "nombre": "Juan",  
    "apellido": "Pérez",  
    "correo": "juan.perez@unisabana.edu.co",  
    "telefono": "3001234567",  
    "rol": "pasajero",  
    "foto": "url_foto",  
    "vehiculos": ["v001"] // si tiene  
}
```

Vehículos

```
{  
    "_id": "v001",  
    "placa": "ABC123",  
    "marca": "Kia",  
    "modelo": "Picanto",  
    "capacidad": 4,  
    "soat_vigencia": "2025-10-01",  
    "foto": "url_foto_carro",  
    "id_conductor": "u123"  
}
```

Viajes

```
{  
    "_id": "viaje001",  
    "id_conductor": "u123",  
    "vehiculo": "v001",
```

```
"origen": {
    "nombre": "Universidad de La Sabana - Puente Madera",
    "lat": 4.864201,
    "lng": -74.033512
},
"destino": {
    "nombre": "Portal Norte",
    "lat": 4.757983,
    "lng": -74.046761
},
"ruta": ["Ad Portas", "Fontanar", "Peaje Autopista"],
"hora_salida": "2025-09-09T07:30:00",
"tarifa": 4000,
"cupos_disponibles": 3,
"estado": "activo"
}
```

Reservas

```
{
    "_id": "r001",
    "id_viaje": "viaje001",
    "id_pasajero": "u567",
    "cupos_reservados": 2,
    "punto_recogida": "Chía - Fontanar",
    "estado": "confirmada"
}
```

Calificaciones (opcional)

```
{
    "_id": "c001",
    "id_viaje": "viaje001",
    "id_pasajero": "u567",
    "id_conductor": "u123",
    "calificacion": 5,
    "comentario": "Puntual y amable"
```

}

Beneficios de MongoDB aquí:

1. **Documentos embebidos** → puedes guardar vehículos dentro de un usuario o separarlos en colección (flexible).
2. **Escalabilidad** → si mañana hay 10.000 viajes diarios, Mongo lo aguanta.
3. **Geospatial Queries** → puedes buscar viajes cerca de un punto (*ejemplo: "muéstrame viajes que salgan en 2km alrededor de Fontanar"*).

Solo necesitas 4 colecciones para empezar Wheels Sabana:

1. Usuarios

- a. Guardas info de estudiantes (pasajeros o conductores).
- b. Si un usuario tiene un carro → se le activa el rol de conductor.

2. Vehículos

- a. Datos de cada carro (placa, modelo, capacidad, etc.).
- b. Siempre asociado a un conductor.

3. Viajes

- a. Info de los viajes creados por conductores:
 - i. Origen, destino, hora, cupos, tarifa.

4. Reservas

- a. Relación entre pasajero y viaje.
- b. Dice: “*Pepita reservó 2 cupos en el viaje de Juan*”.

Cómo se conectan (explicado fácil)

- **Usuario** puede ser pasajero  o conductor .
- **Conductor** tiene al menos 1 **vehículo**.
- **Conductor + vehículo** crean un **viaje**.
- **Pasajeros** reservan cupos en ese **viaje** (reservas).

Ejemplo práctico

- Juan (usuario) → tiene un carro Picanto (vehículo).
- Juan crea un viaje (viaje001) de *Sabana – Portal Norte*.
- María (usuario) → reserva 1 cupo en viaje001 (reserva).

Tip para que no te enredes

Imagina que cada colección es como una **tabla de Excel**:

- Usuarios.xlsx → lista de estudiantes.
- Vehículos.xlsx → lista de carros.
- Viajes.xlsx → lista de viajes creados.
- Reservas.xlsx → lista de quién va en cada viaje.

👉 Eso mismo, pero guardado en **MongoDB** en lugar de Excel.

❖ Herramientas que mencionaste

◊ Firebase

- 🔑 Autenticación de usuarios (login con correo institucional, Google, etc.).
 - 📁 Base de datos en tiempo real (Firestore o Realtime DB).
 - 📲 Notificaciones push (cuando un viaje se cancela o cambia).
 - 💡 Seguridad y hosting escalable.
- 👉 **Muy recomendado** si quieres ir rápido y seguro sin manejar servidores propios.

◊ Google Maps

- 🗺 Mapas interactivos en tu app (para ver viajes, rutas, puntos de recogida).
- 🚕 Directions API (calcula rutas optimizadas).

-  Geocoding API (transforma direcciones ↔ coordenadas).
 **Casi obligatorio** para Wheels Sabana porque tu app depende de rutas y ubicaciones.

 Nota: **no necesitas Firebase para usar Google Maps**, pero **Firebase + Maps** se complementan bien (ej: guardar coordenadas de viajes en Firestore y luego mostrarlas en Maps).

◊ Leaflet.js

- Librería open source para mapas (alternativa a Google Maps).
 - Ventaja: Gratis y ligera.
 - Desventaja: Menos datos en tiempo real que Google (no tiene tráfico en vivo como Waze/Maps).
-  **Úsalo si no quieres pagar Google Maps** a futuro.

◊ SweetAlert

- Librería de **alertas bonitas y modales** (ej: confirmar reserva, error de login, viaje lleno).
-  Es un **extra visual** para mejorar UX, pero no es esencial.

◊ Swagger UI

- Herramienta para documentar y probar APIs.
 - Si vas a crear un backend (ej: Node.js con Express), Swagger te ayuda a:
 - Probar endpoints (login, crear viaje, reservar, etc.).
 - Documentar tu API para que todo el equipo entienda cómo consumirla.
-  **Muy recomendado** si tu profe pide documentación técnica o si trabajas en equipo.

💡 Entonces, ¿qué es necesario?

💡 Necesario sí o sí

- **Firebase** → auth, DB, notificaciones.
- **Google Maps** → rutas, ubicaciones.
- **Swagger** → documentar y probar tu API.

❖ Opcional (mejora experiencia)

- **SweetAlert** → para interfaz más bonita.
- **Leaflet** → solo si no quieras Google Maps (pero perderías tráfico en tiempo real).

💡 Recomendación final para Wheels Sabana

- **Backend ligero** con Firebase (no te complicas con servidores).
- **Swagger** para documentar la API REST que expongas (ej: /viajes, /usuarios, /reservas).
- **Google Maps** para rutas y geolocalización (sí o sí).
- **SweetAlert** solo como “maquillaje” de la UI.

💡 ¿Qué es Socket.IO?

Es una librería de **Node.js** que facilita el uso de **WebSockets** (comunicación en tiempo real).

Funciona como un "chat abierto" entre servidor y clientes, para que todos reciban datos en vivo sin refrescar la página.

💡 ¿Para qué podrías usarlo en Wheels Sabana?

1. Actualización de cupos en tiempo real 🚙

- Ejemplo: Un pasajero reserva → todos los demás ven de inmediato que quedan menos puestos.
- Si no hay Socket.IO → habría que refrescar o consultar la base de datos otra vez.

2. **Cancelación de viajes instantánea** 
 - a. El conductor cancela → pasajeros conectados reciben la notificación al instante.
3. **Ubicación del conductor en vivo (tracking)** 
 - a. Los pasajeros pueden ver cómo el carro se acerca, parecido a Uber.
 - b. Socket.IO envía las coordenadas cada pocos segundos.
4. **Chat entre pasajero y conductor (opcional)** 
 - a. Pasajero: “Estoy en Ad Portas, al lado de la portería”.
 - b. Conductor: “Listo, ya voy llegando”.

¿Conviene usarlo en tu caso?

- **Si usas Firebase Firestore:** ya tienes actualizaciones en tiempo real (sin configurar sockets).
- **Si usas MongoDB:** Mongo como tal no es tiempo real → ahí sí **Socket.IO** te ayuda mucho.

 Entonces:

- **MVP rápido y simple** → Firebase (no necesitas Socket.IO).
- **Versión pro y escalable con MongoDB** → sí conviene meter Socket.IO para tiempo real (cupos, viajes, tracking).

 En palabras simples:

- **Firebase = tiempo real fácil, ya hecho.**
- **Socket.IO = tiempo real personalizable y más poderoso.**

💡 Lo que necesitas según lo que hemos hablado

- Autenticación segura con correo institucional.
- Guardar usuarios, vehículos, viajes y reservas.
- Mostrar viajes disponibles y cupos.
- Bloquear viajes cuando se llenen.
- Notificaciones si cancelan un viaje.
- Escalabilidad (soportar muchos estudiantes en la U).
- Que sea fácil de implementar (tienes poco tiempo y es proyecto de clase).

❖ Opción 1: Firebase (Firestore + Auth + FCM)

✓ Ventajas:

- Tiempo real nativo sin configurar nada extra.
- Autenticación lista (email institucional, Google, etc.).
- Notificaciones push con Firebase Cloud Messaging.
- Hosting y backend serverless (no necesitas montar un servidor Node).
- Rápido de implementar → MVP listo en semanas.

✗ Desventajas:

- Estructura un poco menos flexible que Mongo.
- Escalabilidad pro = \$\$\$ (tienes que pagar si hay muchísimos usuarios).
- Menos control sobre el backend (Firebase es más “caja negra”).

👉 Ideal si: quieres algo **rápido, simple y que funcione bien en demo/universidad**.

❖ Opción 2: MongoDB + Socket.IO + Node.js

✓ Ventajas:

- Control total del backend (puedes definir tus reglas de negocio como quieras).

- MongoDB es muy flexible y escalable (documentos JSON).
- Socket.IO te da tiempo real personalizado (cupos, cancelaciones, tracking en vivo).
- Más parecido a lo que usan apps como Uber.

 Desventajas:

- Requiere más tiempo: montar servidor en Node, configurar sockets, documentar con Swagger, etc.
- Más mantenimiento técnico.
- La curva de aprendizaje es un poco más alta.

 Ideal si: quieres un **proyecto más pro, escalable y cercano a producción real**.

Mi recomendación para Wheels Sabana (según lo que hablamos)

- Si lo que buscas es un **proyecto funcional, rápido y defendible en la U** →
 **Firebase + Google Maps + Swagger + SweetAlert**.
- Si lo que quieres es algo más **robusto y demostrable como “mini Uber universitario”** →
 **MongoDB + Node.js + Socket.IO + Google Maps + Swagger**.

 Truco: puedes empezar con **Firebase (rápido)** y luego migrar a **Mongo + Socket.IO (robusto)** si te queda tiempo o si el profe pide más complejidad.

💡 Cómo se vería un “combo completo” para Wheels Sabana

◊ 1. Firebase (rápido y seguro)

- Autenticación de usuarios (correo institucional, Google).
- Notificaciones push (viajes cancelados, recordatorios).
- Hosting del frontend (si no usas otro).

👉 Lo usas como **puerta de entrada** para seguridad y mensajería.

◊ 2. MongoDB Atlas (backend flexible)

- Base de datos principal: usuarios, vehículos, viajes, reservas.
- Más control para reglas de negocio complejas (roles, cupos, historial).

👉 Lo usas como **cerebro del proyecto** para datos.

◊ 3. Node.js + Express + Swagger

- API REST documentada y probada.
- Controla la lógica de negocio:
 - Crear viaje, reservar, cancelar.
 - Validar cupos.
- Swagger para documentar endpoints.

👉 Lo usas como **ponte** entre frontend ↔ MongoDB ↔ Firebase.

◊ 4. Socket.IO

- Comunicación en tiempo real:
 - Cupos bajan automáticamente.
 - Viaje cancelado se notifica instantáneo.
 - Ubicación en vivo del conductor (tracking).

👉 Lo usas como **canal en vivo** para actualizaciones.

◊ 5. Google Maps API

- Mostrar rutas, calcular tiempos, ubicar portería de salida (Puente Madera / Ad Portas).
- Geocoding para transformar direcciones.

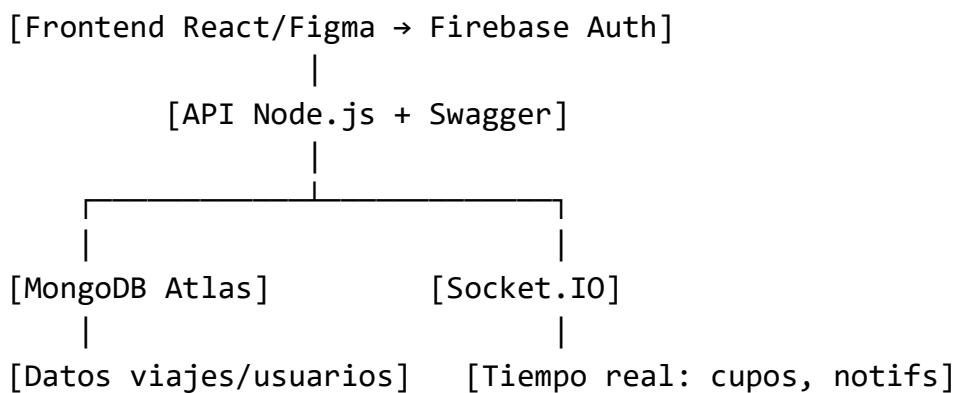
👉 Lo usas como **mapa y calculadora de rutas**.

◊ 6. SweetAlert

- UX bonita para alertas, reservas confirmadas o errores.

👉 Solo es **maquillaje visual**, pero ayuda a defender el proyecto.

ડ Arquitectura híbrida (simplificada)



Firebase entra solo en 2 puntos: **Auth + Notifs**. El resto vive en **MongoDB + Node**.

⌚ Entonces, ¿se puede todo?

✓ Sí, pero:

- Si lo haces todo, el proyecto se vuelve **nivel producción** (muy pro, pero más trabajo).
- Si es solo para **clase/prototipo**, tal vez no necesitas tanto.

👉 Lo mejor es un **MVP con Firebase + Maps + Swagger** y luego, si hay tiempo, le añades **Mongo + Socket.IO** para hacerlo “full Uber”.

Excelente que lo traigas de nuevo 🚀 porque esas APIs le pueden dar mucho valor a tu app, pero también hay que ver **qué tan viables son** según el alcance de tu proyecto (MVP vs proyecto full).

◊ 1. Uber API

- **Qué ofrece:** tarifas estimadas, tiempos de espera, cálculo de rutas dinámicas, disponibilidad de autos.
- **Problema:** ya no está tan abierta, Uber cerró mucho de su API pública. Para integrarte de verdad toca pasar por acuerdos con ellos.
- **Alternativa realista:**
 - Usar **Google Maps Directions API** para calcular tiempo de llegada y rutas.
 - Simular tarifas (definidas por conductor o fórmula propia).

👉 Para un **prototipo**: mejor Google Maps + tu lógica.

◊ 2. TransMilenio API

- Bogotá tiene datos abiertos en el **portal de datos** (<https://datosabiertos.bogota.gov.co>).
- **Qué puedes usar:**
 - Rutas, paraderos, horarios.
 - Información de estaciones y troncales.
- **Cómo aplicarlo:**
 - Mostrar paraderos cercanos como referencia.
 - Hacer que tus rutas compartidas (Wheels) se integren con la red troncal → ej:

“Viaje en Wheels hasta Portal Norte, luego conexión a Troncal TransMilenio.”

👉 Esto **sí lo puedes usar**, y queda muy innovador porque integra transporte público y privado.

❖ 3. Waze API

- Waze ofrece el **Waze for Developers (Waze Deep Links, Waze Transport SDK)**.
- **Qué permite:**
 - Abrir rutas en Waze desde tu app.
 - Compartir ubicación en vivo usando Waze.
 - Redirigir al conductor a la mejor ruta según tráfico real.
- **Limitación:** no es tan abierta como Maps, pero sirve como **complemento** (ej: botón “Ver en Waze”).

👉 Ideal para dar al conductor la opción de navegación “Google Maps” o “Waze”.

💡 Entonces, ¿cómo encajan en Wheels Sabana?

- **Obligatorios (sí o sí):**
 - Google Maps API (mapas, rutas, geocoding).
 - Firebase/Mongo (usuarios, viajes).
- **Opcionales (para dar plus):**
 - Waze (botón de navegación alterna para conductores).
 - TransMilenio (paraderos y rutas como referencia → multimodalidad).
- **Difícil en MVP:**
 - Uber API (cerrada, mejor simular con Maps).

⌚ Recomendación para ti

Si lo que quieres es **impactar en la clase**, integra **Google Maps + Waze + TransMilenio**.

Con eso muestras innovación y un sistema multimodal: Wheels + Transporte Público.

Uber no lo pongas porque puede complicar la entrega y no agrega mucho valor real en prototipo.

🚗 Cómo calcular tarifas sin Uber API

❖ 1. Usar Google Maps Distance Matrix API

- Te da la **distancia** y el **tiempo estimado** entre origen y destino.
- Con eso puedes armar tu propia fórmula, ejemplo:

```
tarifa = tarifa_base + (costo_km * distancia) + (costo_minuto * tiempo)
```

- Si el viaje es compartido, la tarifa por pasajero = tarifa_total / número_de_pasajeros.

👉 Esto simula el cálculo de Uber, pero lo controlas tú.

❖ 2. Inspirarse en Uber pero adaptado

Uber usa algo así:

- Tarifa base
- Costo por minuto
- Costo por kilómetro
- Multiplicador (según demanda)

Tú podrías simplificarlo:

- ✓ Tarifa base = \$2.000
- ✓ Costo por km = \$1.000
- ✓ Costo por minuto = \$200
- ✓ Sin multiplicador (para MVP)

Ejemplo:

- Viaje de 8 km, 20 minutos
- Tarifa = $2.000 + (8 \cdot 1.000) + (20 \cdot 200) = 2.000 + 8.000 + 4.000 = \mathbf{14.000}$
- Si son 4 pasajeros → cada uno paga \$3.500

◊ 3. El conductor puede ajustar

- El sistema calcula un valor sugerido (como hace Uber).
- El conductor puede dejarlo igual o ajustarlo dentro de un rango permitido (ej: ±20%).

❖ Conclusión

No necesitas la API de Uber.

- **Google Maps** te da distancias y tiempos.
- Tu backend hace el cálculo con una fórmula inspirada en Uber.
- Al final, decides si:
 - **Automatizas** (tarifa sugerida por sistema).
 - **Dejas flexible** (el conductor ajusta dentro de un rango).

Tipografías (claras, modernas, legibles)

Piensa en algo **universitario, tecnológico y confiable**:

- **Primaria (títulos, headers):**
 - *Poppins* → moderna, limpia y amigable.
 - Alternativa: *Montserrat*.
- **Secundaria (textos, descripciones):**
 - *Roboto* → estándar en apps móviles y muy legible.
 - Alternativa: *Inter*.

 Ejemplo:

- H1 (Pantallas principales): Poppins Bold 24–28px
- H2 (Subtítulos): Poppins SemiBold 20–22px
- Body text: Roboto Regular 14–16px
- Labels / Inputs: Roboto Medium 12–14px

Paleta de Colores (inspirada en movilidad + universidad)

Base

- **Azul Sabana** → #003366 (confianza, identidad académica)
- **Verde Movilidad** → #4CAF50 (ecología, transporte compartido)
- **Amarillo Resaltado** → #FFC107 (alertas, puntos de recogida)

Secundarios

- **Gris Claro (fondos neutros)** → #F5F5F5
- **Gris Oscuro (texto secundario)** → #555555
- **Rojo Error / Cancelar** → #E53935

 Juega con contrastes: azul (identidad) + verde (acción positiva) + amarillo (interacción).

Componentes Reutilizables en Figma

1. Botones (Primary / Secondary / Disabled)

- a. Primary: fondo azul (#003366), texto blanco.
- b. Secondary: fondo blanco, borde azul, texto azul.
- c. Disabled: gris claro con texto gris oscuro.

2. Inputs (formularios)

- a. Campo con label arriba, placeholder gris claro.
- b. Icono opcional (ej:  para dirección).

3. Cards (para viajes disponibles)

- a. Foto conductor + nombre.
- b. Ruta resumida (inicio → fin).
- c. Info: cupos, hora, tarifa.
- d. Botón “Reservar”.

4. Navbar inferior (App móvil)

- a. Home () , Buscar () , Mis Viajes () , Perfil () .

5. Modales / Alertas (SweetAlert style)

- a. Confirmar reserva.
- b. Viaje cancelado.
- c. Error (cupos agotados).

6. Listados con filtros

- a. Filtros arriba (cupo disponible, salida, destino).
- b. Lista de cards con scroll.

7. Mapa embebido

- a. Basado en Google Maps.
- b. Iconos personalizados ( conductor,  puntos recogida).

Recomendación de flujo visual

- **Home** → rutas destacadas / próximos viajes.
- **Buscar viaje** → filtros + mapa.
- **Detalle del viaje** → conductor, ruta, cupos, precio.
- **Reserva** → seleccionar cupos y punto de recogida.
- **Perfil** → alternar pasajero ↔ conductor, ver historial.

Reglas de negocio Wheels Sabana

1. Reservas de cupos

- a. Un pasajero puede reservar **más de un cupo** en un viaje (ej: para amigos).
- b. Cada cupo reservado requiere especificar el punto de recogida.

2. Cálculo de tarifas

- a. El sistema sugiere un precio automático con base en **distancia + tiempo (Google Maps)**.
- b. El conductor puede ajustar la tarifa de acuerdo a la inflación, pero dentro de un rango permitido (ej: ±20%).

3. Puntos de salida de la Universidad

- a. Los viajes deben tener puntos de inicio predefinidos para evitar confusiones:
 - i. **Puente Madera**
 - ii. **AdPortas**
- b. Estos se muestran al pasajero en el momento de buscar viajes.

4. Notificaciones

- a. El sistema debe notificar al pasajero y al conductor en caso de cambios o cancelaciones.
- b. Se priorizan **push notifications**, con respaldo por **correo institucional**.

5. Validación de conductores

- a. Para que un pasajero pueda convertirse en conductor, debe registrar al menos un vehículo.
- b. El vehículo y el conductor deben pasar una validación con:
 - i. **SOAT vigente**
 - ii. **Licencia de conducción válida**
 - iii. **Placa, marca, modelo y capacidad del vehículo**

Historias de Usuario – Wheels Sabana

Epic 1: Registro y Autenticación

1. Como estudiante, quiero registrarme con mi **correo institucional** para acceder a la plataforma.
2. Como usuario, quiero iniciar sesión con **correo y contraseña** para entrar a mi cuenta.
3. Como usuario, quiero poder **cerrar sesión** para proteger mi cuenta.
4. Como usuario, quiero **recuperar mi contraseña** para acceder en caso de olvidarla.
5. Como usuario, quiero **ver y editar mi perfil** (nombre, apellido, ID, número, foto) para mantenerlo actualizado.
6. Como sistema, debo validar que solo **correos @unisabana.edu.co** puedan registrarse.

Epic 2: Gestión de Conductores y Vehículos

7. Como usuario pasajero, quiero **registrar un vehículo** con sus datos (placa, marca, modelo, capacidad, SOAT, licencia) para poder convertirme en conductor.
8. Como usuario, quiero **alternar entre pasajero y conductor** para usar la app de ambas formas.
9. Como conductor, quiero **gestionar mis vehículos** (añadir, editar, eliminar) para mantener mi información actualizada.
10. Como sistema, debo validar que cada conductor tenga al menos **un vehículo registrado**.
11. Como sistema, debo validar que el vehículo tenga documentos al día (**SOAT y licencia vigentes**).



Epic 3: Creación y Gestión de Viajes

12. Como conductor, quiero **crear un viaje** indicando: punto de inicio, destino, ruta, hora de salida, tarifa y cupos disponibles.
13. Como conductor, quiero **definir puntos de recogida intermedios** para que pasajeros sepan dónde pueden subirse.
14. Como sistema, quiero **calcular la distancia y tiempo estimado** de un viaje usando Google Maps/Waze.
15. Como sistema, quiero sugerir una **tarifa mínima sugerida** según distancia e inflación (pero que el conductor pueda ajustarla).
16. Como pasajero, quiero **ver un listado de viajes disponibles** con detalles de ruta, tarifa, cupos, hora y conductor.
17. Como pasajero, quiero **reservar uno o más cupos** para asegurar mi lugar en el viaje.
18. Como pasajero, quiero **seleccionar el punto de recogida** para cada reserva.
19. Como sistema, quiero **bloquear automáticamente un viaje lleno** para que no se sobreventa.
20. Como conductor, quiero **ver quiénes reservaron** mi viaje y sus puntos de recogida.



Epic 4: Búsqueda y Filtros

21. Como pasajero, quiero **filtrar viajes por punto de salida** (ej. Puente Madera o AdPortas).
22. Como pasajero, quiero **filtrar viajes por número de cupos disponibles**.
23. Como pasajero, quiero **filtrar viajes por hora de salida**.
24. Como pasajero, quiero **filtrar viajes por precio máximo por pasajero**.



Epic 5: Notificaciones y Comunicación

25. Como pasajero, quiero **recibir notificación si el viaje se cancela** para no quedarme esperando.
26. Como pasajero, quiero **recibir notificación si el conductor cambia la hora de salida** para estar actualizado.
27. Como conductor, quiero **recibir notificación cuando alguien reserve un cupo** en mi viaje.

28. Como pasajero, quiero poder **cancelar mi reserva** para liberar el cupo a otro estudiante.
29. Como conductor, quiero **cancelar un viaje** en caso de que no pueda realizarlo.
30. Como sistema, quiero enviar **recordatorios automáticos** antes de la hora de salida.

★ Epic 6: Reputación y Seguridad

31. Como pasajero, quiero **calificar al conductor** al final del viaje.
32. Como conductor, quiero **calificar a los pasajeros** para mejorar la confianza.
33. Como sistema, quiero mostrar un **promedio de calificaciones** en los perfiles.
34. Como sistema, debo **encriptar contraseñas y proteger datos personales** para garantizar la seguridad.
35. Como sistema, debo asegurar que la app esté disponible al menos **99% del tiempo**.

✉ Epic 7: Pagos (Futuro, pero lo incluyo)

36. Como pasajero, quiero **pagar en efectivo o Nequi** directamente al conductor.
37. Como conductor, quiero **ver un historial de reservas y pagos recibidos**.
38. Como sistema, quiero permitir en el futuro **pagos electrónicos integrados** (API Nequi, PSE, etc.).

🌐 Epic 8: Infraestructura y Rendimiento

39. Como sistema, quiero ser **responsive** para que funcione en celular, tablet y PC.
40. Como sistema, quiero que las pantallas carguen en **menos de 2 segundos**.
41. Como sistema, quiero ser **escalable** para soportar un gran número de usuarios.
42. Como sistema, quiero integrarme con **Google Maps, Waze, Uber API y TransMilenio API** para mejorar rutas, precios y tiempos.
43. Como sistema, quiero tener **sockets para actualizaciones en tiempo real** (ej. cupos ocupados o cambios de ruta).