https://drive.google.com/drive/folders/1he33tsxXwarZBp1tJ68TZsPT-Y8ikEoz?usp=drive_link

Punto 1 - Modelo de Datos (30 puntos)

1. (5 puntos) Enumere las entidades identificadas y mencione, de forma concisa, qué representa cada una de ellas.

- Monopatinador: Usuario registrado en el sistema que utiliza los monopatines y abona su costo. Incluye sus datos personales y debe validar su identidad.
    - Medio de contacto: Dirección del monopatinador. Email o teléfono.

- Monopatines: Representa al monopatín físico, permitiendo trazar su estado y disponibilidad.
    - Modelo
    - Marca
    - Estado: Estado actual en el que se encuentra (disponible, no disponible, en reparación, en uso, etc). Se hace una tabla intermedia para tener trazabilidad de los estados por los que pasa.

- Reparación: Evento histórico que describe una intervención sobre un monopatín (fecha, tarea realizada, observaciones).

- Estación: Lugar geográfico donde se pueden almacenar monopatines. Incluye ubicación.

- Viaje: Representa el recorrido puntual de una persona al tomar un monopatín. Tiene un costo, tiempo y distancia.

- Tarifa: Determina el valor del uso del monopatín, costo anual y adicional (ajustable por diversos factores y costos).

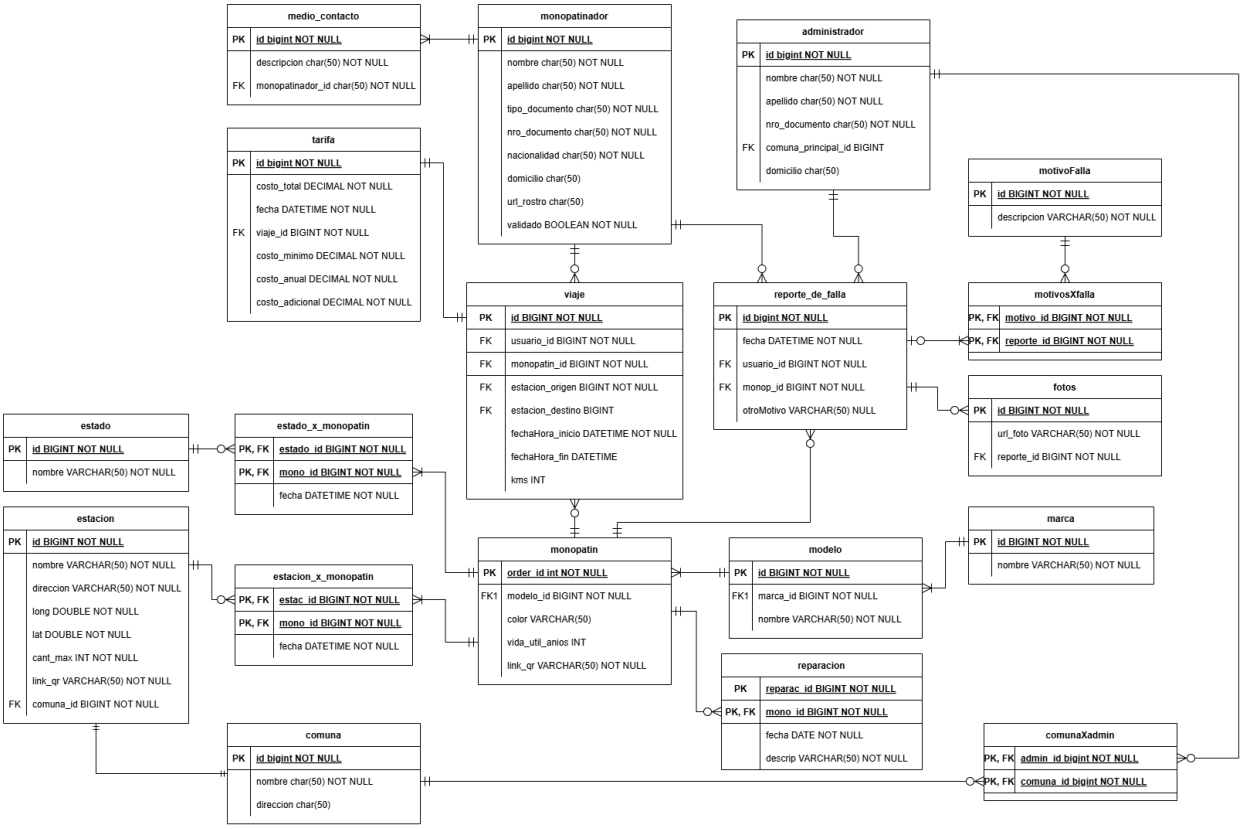
- Administrador: Encargado de registrar y modificar las tarifas del sistema, reportar fallas. Incluye datos personales.
    - Comuna: Zonas de la ciudad. Gestionadas por un administrador a cargo.

- Falla Reportada: Incluye datos del fallo y descripción del mismo. El reporte está asociado al usuario que lo realizó.
    - Motivo: Enumerado que permite ofrecer diferentes tipos de fallas precargadas.

- Factura: Documento que refleja el pago por el uso del servicio (particularmente para usuarios extranjeros). Incluye monto, fecha y concepto.

2. (25 puntos) Realice el modelo de datos del dominio presentado. Indique los supuestos que crea necesario considerar.

https://drive.google.com/file/d/1yxCiiUbVL-aPKS16O4yJiOob1x_YIW2c/view?usp=sharing



Arquitectura (40 puntos)

1. (10 puntos) Entendiendo que nuestro Sistema debe poseer una capa de presentación gráfica, ¿qué tipo de interfaz propondría implementar para los monopatinadores? Justifique adecuadamente su respuesta entendiendo que se poseen 5 meses en total para lanzar el Sistema a producción.

b. Aplicación mobile híbrida.

Lo que más interesa es la usabilidad, portabilidad y disponibilidad. Respecto a la usabilidad, se trata de un sistema que será usado siempre en el exterior. Esto nos lleva a una aplicación mobile. Debido al poco tiempo de desarrollo, se recomienda que sea híbrida (ganar portabilidad), ya que la nativa implica realizarla en los dos SO más usados.

Al ser híbrida, permite instalarse desde una tienda tanto Android como iOS, y puede usar las capacidades del dispositivo como la cámara o el GPS, gracias al shell nativo. Esto es fundamental para las fotografías, el análisis facial, los kilómetros recorridos y conocer la ubicación de las estaciones. La principal ventaja es que se escribe un solo código que funciona en múltiples plataformas, lo que agiliza el desarrollo y reduce costes.

2. (5 puntos) Se cuenta con un componente de Backoffice para que los administradores (empleados del Gobierno de la Ciudad), gestionen las tarifas, y demás parámetros de configuración, de forma mensual. El cliente nos comenta que la alta disponibilidad es prioritario para este componente. ¿Está de acuerdo? ¿Por qué? Justifique adecuadamente.

No. El Backoffice probablemente se use solo por administradores, una vez al mes. No es crítico 24/7 como el módulo de uso público (reservas, viajes, pagos). Si el Backoffice cae temporalmente, los usuarios (monopatinadores) siguen pudiendo usar el servicio, solo que los administradores no pueden modificar tarifas o parámetros en ese momento.

3. (15 puntos) La Ciudad nos solicitó visualizar en tiempo real el estado de las estaciones y sus variables ambientales (% de monopatines disponibles, temperatura, nivel de ruido, calidad del aire). Para ello ubicará monitores en diferentes edificios del Gobierno para visualizar estos datos. ¿Cómo diseñaría esta arquitectura? Justifique su respuesta. Puede utilizar un Diagrama de Despliegue para comunicar su propuesta.

Para resolver el requerimiento de visualizar en tiempo real el estado de las estaciones y sus variables ambientales, propongo una arquitectura basada en el Patrón Broker (Publicador–Suscriptor).

Desacoplamiento total: los publicadores (estaciones) y suscriptores (monitores) no se conocen entre sí. Esto permite escalar fácilmente el sistema agregando nuevas estaciones o monitores sin modificar los demás componentes.

Asincronismo: el flujo de datos no depende del estado o disponibilidad del consumidor; el broker actúa como buffer temporal.

Escalabilidad: el modelo soporta múltiples fuentes de datos simultáneas.

Bajo acoplamiento y alta flexibilidad: si mañana se desea agregar un sistema de alertas o almacenamiento histórico, solo se suscribe al broker sin alterar las estaciones.

Arquitectura:

- Cada estación envía periódicamente sus datos (porcentaje de monopatines disponibles, temperatura, nivel de ruido, calidad del aire, etc.) a un servidor intermedio de mensajería. Estos datos se publican en tópicos específicos, por ejemplo:
    - /estaciones/{id}/monopatines
    - /estaciones/{id}/ambiente/temperatura
    - /estaciones/{id}/ambiente/ruido

- Broker de Mensajería: Es el componente central del patrón. Actúa como intermediario entre los productores (estaciones) y los consumidores (monitores). Puede implementarse con tecnologías como MQTT Broker (ej. Mosquitto) o Kafka, dependiendo del volumen y frecuencia de los datos.

    - Recibir las publicaciones de las estaciones.
    - Almacenar temporalmente los mensajes.

    - Distribuirlos a los suscriptores interesados.
    - Garantizar bajo acoplamiento y comunicación asincrónica.

- Monitores (Subscribers): Los monitores ubicados en los edificios del Gobierno se suscriben a los tópicos de interés (por estación o variable ambiental). Cada monitor recibe en tiempo real las actualizaciones, que se visualizan mediante una interfaz web o dashboard (por ejemplo, implementado con WebSockets o MQTT Web Client).

4. (10 puntos) Para poder realizar la reparación de los monopatines debemos integrarnos con el Sistema de reparación de la Ciudad. Este Sistema ofrece la posibilidad de integración sincrónica, mediante llamada a una API REST para el envío de las órdenes. Cuando el área acepta la orden, luego de una revisión manual (algunos días después), debe avisarnos. ¿qué mecanismo de integración escogería? ¿Por qué? Justifique adecuadamente su respuesta.

Primero, una llamada sincrónica a la API REST. Nuestro sistema debe crear una orden de reparación (HTTP) y mandarla al sistema externo. Se espera la confirmación de su recepción en el momento.

Para recibir la aceptación, debe ser push based. Una vez aceptada la orden, nos envían de forma asincrónica, un **webhook**. Esto nos ahorra tener que estar esperando una respuesta.

Ventajas:

- Menor carga y costos: evitamos realizar peticiones repetitivas (polling) que consumiría recursos de red y procesamiento en ambos sistemas.

- Mayor eficiencia y escalabilidad: el sistema reacciona automáticamente ante los eventos, permitiendo procesar grandes volúmenes de órdenes sin bloqueos.

- Acoplamiento mínimo: el sistema de reparación no necesita conocer detalles internos del nuestro, solo la URL del webhook.

- Trazabilidad y resiliencia: se pueden registrar los eventos entrantes y reenviar en caso de fallas de red (los webhooks suelen incluir mecanismos de reintento).

Persistencia (20 puntos)

```
@Entity @Table(name="deportista")
class Deportista {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @Column(name = "altura")
  private Double altura;

  @Column(name = "apellido")
  private String apellido;

  @ElementCollection
  @CollectionTable(name = "deportista_contactos",
        joinsColumns= @JoinColumn(name=”deportista_id”) )
  private List<String> contactos;

  @Convert(converter = MotivacionConverter.class)
  @Column(name = “motivacion_principal")
  private Motivacion motivacionPrincipal;

  @Column(name = "nombre")
  private String nombre;

  @Column(name = "peso_inicial")
  private Double pesoInicial;
 }

@Entity @Table(name="rutina")
class Rutina {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @ManyToOne
  @JoinColumn(name = "deportista_id", referencedColumnName=”id”)
  private Deportista deportista;

  @ManyToMany @JoinTable(name = "rutinaXdia"
   joinColumns= @JoinColumn(name = "rutina_id",  referencedColumnName=”id”),
 inverseJoinColumns = @JoinColumn(name = "dia_id", referencedColumnName=”id”)
)
  private List<DiaEntrenamiento> dias;

  @OneToOne
  @JoinColumn(name = "rutina_anterior_id", referencedColumnName=”id”)
   private Rutina rutinaAnterior;
}
```

```
@Entity @Table(name="diaEntrenamiento")
class DiaEntrenamiento {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @ManyToMany
  @JoinTable(name = "ejercicioXdia"
   joinColumns= @JoinColumn(name = "dia_id",  referencedColumnName=”id”),
 inverseJoinColumns = @JoinColumn(name = "ejercicio_id",
referencedColumnName=”id”) )
  private List<Ejercicio> ejercicios;

  @Column(name = "numero")
  private Int numero;

  @OneToOne
  @JoinColumn(name = "siguiente_dia_id", referencedColumnName=”id”)
   private DiaEntrenamiento siguienteDia;
 }

@Entity @Table(name="ejercicio")
class Ejercicio{
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @Column(name = "detalle")
  private String detalle;

  @Column(name = "nombre")
  private String nombre;
}
```

Se podría mapear la Motivación como interfaz stateless. La herencia no tiene sentido, ya que es rígida y en este caso se usa sólo para métodos. Se aplica un converter.

```java
@Converter(autoApply="true”)
public class MotivacionConverter implements AttributeConverter<Motivacion,String>{
   @Override
   public String convertToDatabaseColumn(Motivacion tipo) {
      if (tipo == null) return null;
      if (tipo instanceof BajarDePeso) return "BAJAR_PESO";
      if (tipo instanceof Tonificar) return "TONIFICAR";
      if (tipo instanceof Mantener) return "MANTENER";
   throw new IllegalArgumentException("Motivación desconocida: " + tipo.getClass());
   }

   @Override
   public Motivacion convertToEntityAttribute(String valor) {
      if (valor == null) return null;
      return switch (valor) {
         case "BAJAR_PESO" -> new BajarDePeso();
         case "TONIFICAR" -> new Tonificar();
         case "MANTENER" -> new Mantener();
         default -> throw new IllegalArgumentException("Valor inválido: " + valor);
      };
   }
 }
```

Teoría (10 puntos)

Responder V (verdadero) o F (Falso) Además, justifique brevemente su respuesta solamente en caso de responder por la opción de “falso”.

1) Hibernate es un ORM del tipo Active Record.

FALSO. Hibernate implementa el patrón **Data Mapper**, no Active Record.
Active Record es un patrón de diseño diferente utilizado en marcos como Ruby on Rails.

Ambos son patrones ORM, pero difieren en su enfoque:

- Active Record: Es un patrón de diseño que vincula estrechamente un objeto a una fila de una tabla de la base de datos.
- Hibernate: Es un framework (ORM) para Java que proporciona una capa de abstracción sobre JDBC. Su enfoque principal es mapear objetos de Java a tablas de bases de datos

2) Una cookie es generada por el servidor y almacenada en el cliente.

VERDADERO. El servidor envía la cookie en la cabecera Set-Cookie de la respuesta HTTP, y luego el navegador la almacena y la reenvía automáticamente en las siguientes peticiones al mismo dominio.

3) GraphQL es un mecanismo de integración asincrónica.

GraphQL es un lenguaje de consulta y runtime para APIs, generalmente usado sobre HTTP sincrónico, donde el cliente realiza una petición y el servidor responde con los datos solicitados.

Aunque puede usarse en contextos asincrónicos (por ejemplo, subscriptions sobre WebSocket), su modelo base es sincrónico (request/response).

4) En un estilo Call and Return implementado por un Cliente-Servidor, el primero en iniciar la comunicación siempre es el Servidor.

FALSO. En una arquitectura cliente-servidor, el cliente realiza peticiones y el servidor responde a ellas.

5) El principal componente de una arquitectura del tipo de microservicios es el ESB.

Falso. El ESB (Enterprise Service Bus) es característico de SOA (Service Oriented Architecture).

En Microservicios, la comunicación es descentralizada y liviana, generalmente mediante APIs REST, colas o mensajería (event-driven, no un bus centralizado).

Enumerar

1) Dos motivos en los que puede desnormalizar una base de datos relacional.

- Performance. Ahorrar consultas costosas, reduciendo JOINS de tablas
- Integridad y consistencia. Lógica de negocio.

2) Dos ejemplos de algoritmos utilizados en un Load Balancer.

- Round Robin: distribuye las solicitudes de forma secuencial entre los servidores disponibles.
- Least Connections: asigna la nueva solicitud al servidor con menos conexiones activas en ese momento.

3) Un ejemplo de base de datos compartida

Base de datos central única utilizada por varios servicios o aplicaciones (por ejemplo, varios microservicios accediendo a la misma base relacional compartida).

Ejemplo concreto: una base de datos PostgreSQL compartida por los módulos de usuarios, pagos y pedidos dentro de un sistema monolítico desacoplado parcialmente.