

# Documento de Arquitectura - Sistema Carpooling

## Trabajo Final - Arquitectura de Software II - Universidad Católica de Córdoba

---

### Información del Proyecto

**Dominio:** Sistema de carpooling para compartir viajes entre ciudades

**Objetivo:** Permitir que usuarios publiquen o se unan a viajes para compartir gastos

**Fecha Primera Entrega:** 7-14 de Noviembre 2024

**Equipo:** 4 desarrolladores

---

### Arquitectura de Microservicios

#### Servicios Definidos (4 servicios independientes)

1. **users-api** (Puerto 8001) - MySQL
2. **trips-api** (Puerto 8002) - MongoDB
3. **reservations-api** (Puerto 8003) - MySQL
4. **search-api** (Puerto 8004) - Solr + Cache

#### Decisión Clave: Separación trips-api y reservations-api

- **Justificación:** Permite evolución independiente, mejor escalabilidad y separación de responsabilidades
  - **trips-api:** Gestión del ciclo de vida del viaje (durante y post-viaje)
  - **reservations-api:** Flujo transaccional de reserva (pre-viaje)
- 

### Modelo de Datos

#### MySQL - Base de Datos Relacional

**Schema:** carpooling\_users

#### Tabla users

```
sql
```

```

users (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  email_verified BOOLEAN DEFAULT FALSE,
  name VARCHAR(100) NOT NULL,
  lastname VARCHAR(100) NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  role ENUM('user', 'admin') NOT NULL DEFAULT 'user',
  phone VARCHAR(20) NOT NULL,
  street VARCHAR(255) NOT NULL,
  number INT NOT NULL,
  photo_url VARCHAR(255),
  sex ENUM('hombre', 'mujer', 'otro') NOT NULL,
  avg_driver_rating DECIMAL(3,2) DEFAULT 0.00,
  avg_passenger_rating DECIMAL(3,2) DEFAULT 0.00,
  total_trips_passenger INT DEFAULT 0,
  total_trips_driver INT DEFAULT 0,
  birthdate DATE NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)

```

## Tabla ratings

```

sql

ratings (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  rater_id BIGINT NOT NULL,
  rated_user_id BIGINT NOT NULL,
  trip_id VARCHAR(24) NOT NULL, -- ObjectId de MongoDB
  roleRated ENUM('conductor', 'pasajero') NOT NULL,
  score TINYINT NOT NULL CHECK (score >= 1 AND score <= 5),
  comment TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  INDEX idxRatedUser (rated_user_id),
  INDEX idxTrip (trip_id)
)

```

## Schema: carpooling\_reservations

### Tabla reservations

```

sql

```

```
reservations (  
  id VARCHAR(36) PRIMARY KEY DEFAULT (UUID()),  
  trip_id VARCHAR(24) NOT NULL,  
  passenger_id BIGINT NOT NULL,  
  driver_id BIGINT NOT NULL,  
  seats_reserved INT NOT NULL,  
  price_per_seat DECIMAL(10,2) NOT NULL,  
  total_amount DECIMAL(10,2) NOT NULL,  
  status ENUM('pending','confirmed','completed','cancelled') DEFAULT 'pending',  
  payment_status ENUM('pending','paid','refunded') DEFAULT 'pending',  
  arrived_safely BOOLEAN DEFAULT FALSE,  
  arrival_confirmed_at TIMESTAMP NULL,  
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  updated_at TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,  
  UNIQUE KEY unique_trip_passenger (trip_id, passenger_id)  
)
```

## MongoDB - Base de Datos NoSQL

### Colección: trips

javascript

```

{
  _id: ObjectId(),
  driver_id: Number, // ID del usuario conductor
  origin: {
    city: String,
    province: String,
    coordinates: { lat: Number, lng: Number }
  },
  destination: {
    city: String,
    province: String,
    coordinates: { lat: Number, lng: Number }
  },
  departure_date: Date,
  price_per_seat: Number,
  total_seats: Number,
  available_seats: Number,
  description: String,
  car: {
    model: String,
    color: String,
    plate: String
  },
  preferences: {
    pets_allowed: Boolean,
    smoking_allowed: Boolean,
    music_allowed: Boolean
  },
  status: String, // 'published', 'in_progress', 'waiting_confirmations', 'completed', 'cancelled'
  created_at: Date,
  updated_at: Date
}

```

## Responsabilidades de Cada Servicio

### users-api (MySQL)

#### Endpoints principales:

- `POST /users` - Crear usuario
- `GET /users/:id` - Obtener usuario por ID
- `POST /login` - Autenticación con JWT
- `PUT /users/:id` - Actualizar perfil

- `POST /users/:id/rate` - Calificar usuario

### **Responsabilidades:**

- Autenticación y autorización (JWT)
- Gestión de usuarios
- Sistema de calificaciones
- Hash de contraseñas (bcrypt)

## **trips-api (MongoDB) - ENTIDAD PRINCIPAL**

### **Endpoints principales:**

- `POST /trips` - Crear viaje
- `GET /trips/:id` - Obtener viaje
- `PUT /trips/:id` - Actualizar viaje
- `DELETE /trips/:id` - Eliminar viaje
- `POST /trips/:id/reserve` - Endpoint de acción (delega a reservations-api)

### **Responsabilidades:**

- CRUD de viajes
- Validación de owner contra users-api
- Publicar eventos en RabbitMQ
- Gestión del ciclo de vida del viaje

## **reservations-api (MySQL)**

### **Endpoints principales:**

- `POST /reservations` - Crear reserva
- `GET /reservations/:id` - Obtener reserva
- `PUT /reservations/:id/cancel` - Cancelar reserva
- `GET /reservations/user/:userId` - Reservas de un usuario
- `POST /reservations/:id/confirm-arrival` - Confirmar llegada

### **Responsabilidades:**

- Gestión transaccional de reservas (ACID)
- Validación de disponibilidad

- Proceso concurrente de reserva
- Sistema de confirmación de llegada

## search-api (Solr)

### Endpoints principales:

- `GET /search` - Búsqueda paginada con filtros
- `GET /search/suggestions` - Sugerencias de búsqueda

### Responsabilidades:

- Indexación de viajes en Solr
  - Consumidor RabbitMQ para sincronización
  - Cache de dos capas (CCache local + Memcached distribuido)
  - Búsqueda y filtrado optimizado
- 

## Implementación de Tecnologías Requeridas

### JWT (JSON Web Tokens)

- Implementado en users-api
- Expiración: 24 horas
- Contiene: user\_id, email, role
- Middleware para validación en todos los servicios

### RabbitMQ - Sistema de Mensajería

#### Exchanges y Colas:

- `trips.events` → trip.created, trip.updated, trip.deleted
- `reservations.events` → reservation.created, reservation.cancelled
- Consumer en search-api para sincronización con Solr

### Proceso Concurrente (Goroutines + Channels)

#### Implementación en reservations-api al crear reserva:

1. Verificación de disponibilidad (trips-api)
2. Validación de usuario (users-api)
3. Verificación de conflictos de horario

#### 4. Cálculo de precio óptimo

Uso de `sync.WaitGroup` para sincronización y `channels` para comunicación.

## Solr - Motor de Búsqueda

### Campos indexados:

- origen, destino (texto)
- fecha\_salida (fecha)
- precio (numérico)
- lugares\_disponibles (numérico)
- conductor\_rating (numérico)

## Cache Multinivel

1. **CCache** (local): TTL 5 minutos
2. **Memcached** (distribuido): TTL 30 minutos
3. Estrategia: Cache-aside pattern

---

## Docker y Deployment

### Estructura Docker

```
yaml

# Un único docker-compose.yml en la raíz
services:
  mysql:      # Para users y reservations
  mongodb:    # Para trips
  rabbitmq:   # Message broker
  solr:       # Search engine
  memcached:  # Distributed cache
  users-api:  # Servicio usuarios
  trips-api:  # Servicio viajes
  reservations-api: # Servicio reservas
  search-api: # Servicio búsqueda
  frontend:   # React app
```

### Estructura de Carpetas

```
carpooling/
├── docker-compose.yml
```

```
├── services/
│   ├── users-api/
│   │   ├── cmd/
│   │   ├── internal/
│   │   ├── dao/
│   │   ├── domain/
│   │   ├── repository/
│   │   ├── services/
│   │   ├── controllers/
│   │   └── middleware/
│   └── Dockerfile
├── trips-api/
├── reservations-api/
├── search-api/
├── frontend/
├── scripts/
└── init-db.sql
```

## Decisiones Técnicas Clave

### 1. IDs: AutoIncrement vs UUID

- **Decisión:** AutoIncrement (BIGINT) para simplicidad
- **Consideración:** En producción se usarían UUIDs

### 2. Reservas en MySQL vs MongoDB

- **Decisión:** MySQL para reservas
- **Justificación:** Propiedades ACID fundamentales para transacciones

### 3. Rating Dual (Conductor/Pasajero)

- **Decisión:** Ratings separados por rol
- **Implementación:** Tabla específica de ratings con rol Rated

### 4. Usuario Puede ser Conductor Y Pasajero

- **Decisión:** Sí, en diferentes viajes
- **Implementación:** Ratings y contadores separados

### 5. Sistema de Confirmación de Llegada

- **Decisión:** Ambos (conductor y pasajero) confirman
- **Implementación:** Campo booleano en reservations



## 6. Búsqueda Simple vs Geolocalización

- **Decisión:** Búsqueda simple por ciudad/provincia
  - **Futuro:** Posible expansión a búsqueda geográfica
- 

### División del Equipo (4 personas)

#### Persona 1: Backend Core + Users

- users-api completo
- JWT y autenticación
- Docker compose principal
- MySQL setup

#### Persona 2: Trips Service

- trips-api completo
- MongoDB setup
- Integración con RabbitMQ
- Validación de owner

#### Persona 3: Reservations + Concurrencia

- reservations-api completo
- Proceso concurrente
- Lógica de validación
- Tests del servicio

#### Persona 4: Search + Frontend Base

- search-api completo
  - Configuración Solr
  - Cache implementation
  - Estructura base React
-

## ✓ Checklist Primera Entrega (7-14/11)

### Requerido

- ✓ Flujo: Login → Búsqueda → Detalle → Reserva → Confirmación
- ✓ Backend en Go con patrón MVC
- ✓ Docker y Docker Compose funcional
- ✓ JWT implementado
- ✓ RabbitMQ básico
- ✓ MongoDB para trips
- ✓ MySQL para users
- ☐ Al menos un test de servicio

### NO Requerido (Primera Entrega)

- Usuarios administradores
  - Pantalla de administración
  - Vista de registro
  - Vista "Mis Acciones"
  - Cálculo concurrente complejo (versión simple está OK)
- 

## 🚀 Flujos Principales del Sistema

### Flujo de Reserva

1. Usuario busca viajes (search-api → Solr)
2. Selecciona viaje (trips-api → MongoDB)
3. Crea reserva (reservations-api con validaciones concurrentes)
4. Actualiza disponibilidad (trips-api)
5. Publica evento (RabbitMQ)
6. Sincroniza búsqueda (search-api consume evento)

### Flujo de Autenticación

1. Usuario hace login (users-api)
2. Valida credenciales (bcrypt)
3. Genera JWT
4. Token usado en headers para otros servicios

## Flujo de Calificación

1. Viaje finalizado
  2. Usuarios confirman llegada
  3. Sistema habilita calificaciones
  4. Actualiza ratings promedio
- 



## Notas Importantes

1. **Patrón MVC obligatorio** en todos los microservicios
  2. **HTTP + JSON** para comunicación entre servicios
  3. **Validación de errores** en todas las capas con códigos HTTP correctos
  4. **Tests requeridos** en al menos un servicio (`{entidad}_service_test.go`)
  5. **GitHub** con commits de todos los integrantes
- 



## Endpoints HTTP y Códigos de Estado

### Códigos de Respuesta Estándar

- `200 OK` - Operación exitosa
- `201 Created` - Recurso creado
- `400 Bad Request` - Error de validación
- `401 Unauthorized` - Sin autenticación
- `403 Forbidden` - Sin autorización
- `404 Not Found` - Recurso no encontrado
- `409 Conflict` - Conflicto (ej: email duplicado)
- `500 Internal Server Error` - Error del servidor

### Formato de Respuesta Estándar

json

```
{  
  "success": true/false,  
  "data": {},  
  "error": "mensaje de error",  
  "pagination": {  
    "page": 1,  
    "limit": 10,  
    "total": 100  
  }  
}
```

---

*Última actualización: Octubre 2024*

*Documento consolidado de decisiones arquitectónicas para el Trabajo Final*