**DISEÑO DE SOFTWARE**

**JULIETH VANESSA MENA ORTEGA**

**INGENIERIA DE SOFTWARE, UNIVERSIDAD COOPERATIVA DE COLOMBIA**

**PROFESOR: MIGUEL TOVAR**

**PASTO-NARIÑO**

**17 DE SEPTIEMBRE DE 2025**

**Ejercicio 1- Centro médico**

1. Funcionalidades (Requisitos funcionales)

* Pacientes: registrarse e iniciar sesión, gestionar perfil, buscar médicos por especialidad y disponibilidad, reservar/modificar/cancelar citas, recibir recordatorios y consultar historial.
* Médicos: gestionar agenda, definir horarios, aceptar/rechazar citas, ver lista de pacientes y actualizar estado de las consultas.
* Administradores: administrar usuarios, especialidades y recursos, configurar políticas de uso, y generar reportes del sistema.
* Sistema: manejar autenticación y roles, seguridad, envío de notificaciones, almacenamiento de datos y posibles integraciones con calendarios o pasarelas de pago.

2. Modularización (Componentes)

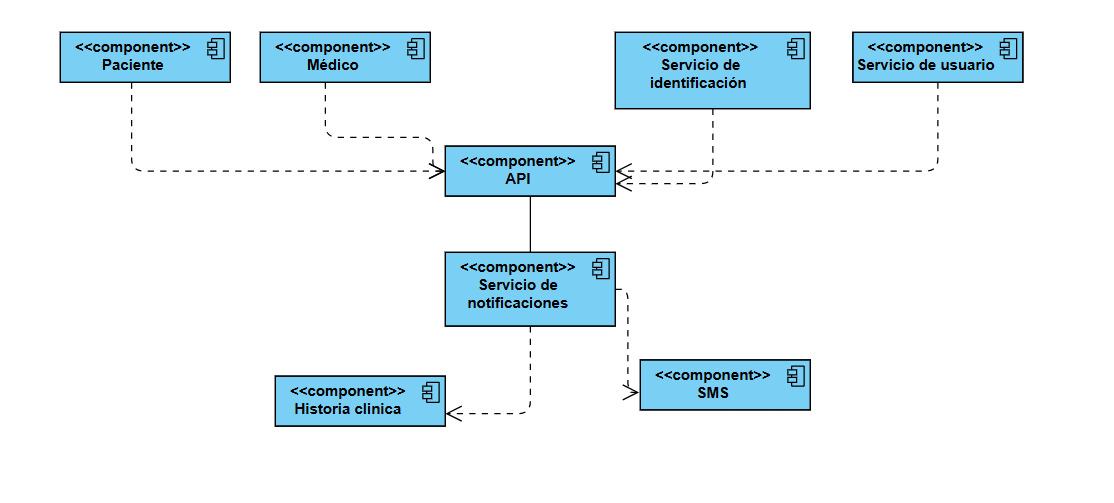
* Frontend: portales diferenciados para pacientes, médicos y administradores.
* Auth Service: autenticación, registro y control de roles.
* User/Profile Service: gestión de datos de pacientes y médicos.
* Scheduling Service: motor de reservas y disponibilidad de citas.
* Notification Service: envío de correos, SMS y recordatorios automáticos.
* Admin & Reporting Service: reportes, métricas y gestión del sistema.
* Storage: base de datos para citas y usuarios, almacenamiento de archivos y logs.

3. Interfaces – Conectores

* Frontend ↔ Backend: comunicación vía REST/HTTPS o GraphQL.
* Servicios internos: comunicación con APIs REST o gRPC.
* Eventos asincrónicos: mediante un bus de mensajes (Kafka/RabbitMQ) para notificaciones y actualizaciones.
* Integraciones externas: conexión a proveedores de SMS, correo, calendarios y pagos.

4. Análisis de diseño

El sistema se divide en módulos con responsabilidades claras, lo que facilita el mantenimiento y la escalabilidad. Las reservas de citas requieren comunicación síncrona para garantizar la disponibilidad en tiempo real, mientras que notificaciones y reportes funcionan mejor con procesos asíncronos. La seguridad se refuerza con roles y cifrado de datos. Además, la arquitectura modular permite empezar con un monolito organizado y migrar hacia microservicios si el centro médico crece.



**Ejercicio 2 – App tipo Rappi/Uber Eats**

1. Funcionalidades

* Usuario: registro, búsqueda de restaurantes/comida, realizar pedidos, pagar en línea, seguir estado de entrega, calificar servicio.
* Restaurante: gestionar menú, recibir y aceptar pedidos, actualizar estado de preparación, administrar horarios.
* Repartidor: recibir pedidos asignados, ver ruta de entrega, actualizar estado de la entrega (en camino, entregado).
* Administrador: supervisar usuarios, restaurantes y repartidores, ver métricas y generar reportes.

2. Modularización (Componentes)

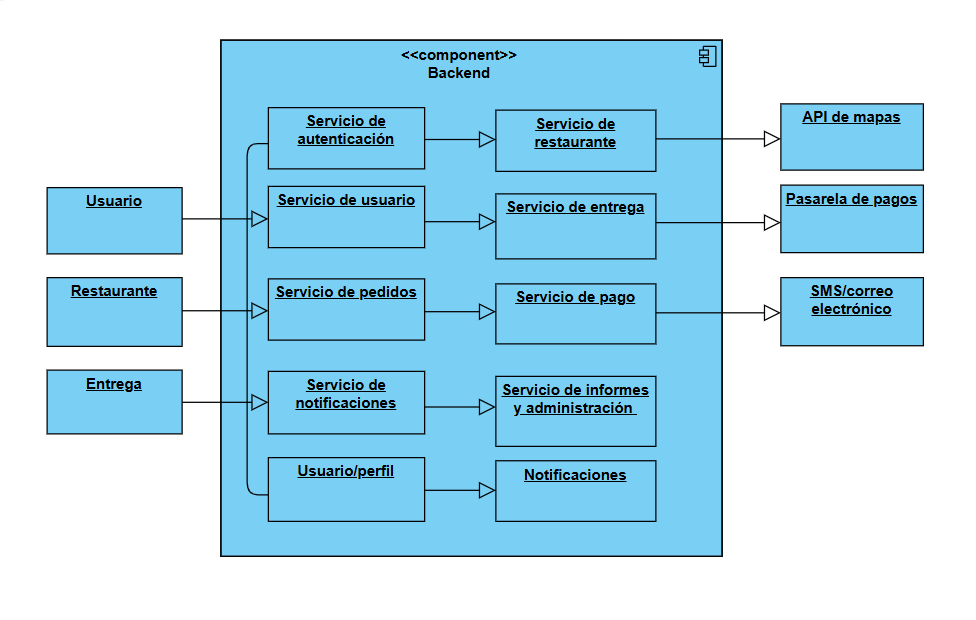
* Frontend: app móvil/web para clientes, restaurantes y repartidores.
* Auth Service: login, roles (usuario, restaurante, repartidor, admin).
* User/Profile Service: gestión de datos y calificaciones.
* Order Service: gestión de pedidos y estados.
* Restaurant Service: menús, horarios y disponibilidad.
* Delivery Service: asignación de pedidos y seguimiento en mapa (GPS).
* Payment Service: integración con pasarelas de pago.
* Notification Service: confirmaciones, cambios de estado, alertas.
* Reporting & Admin Service: estadísticas de ventas, desempeño de repartidores, control del sistema.

3. Interfaces – Conectores

* REST/GraphQL para comunicación cliente-backend.
* APIs internas (REST/gRPC) para coordinar pedidos, pagos y entregas.
* Event bus (Kafka/RabbitMQ): order.created, order.accepted, delivery.assigned.
* Integraciones externas: GPS/Maps API, pasarelas de pago, SMS/email/push.

4. Análisis de diseño

Se separan los módulos principales (Pedidos, Restaurante, Entregas, Pagos) para reducir acoplamiento. Los pedidos requieren procesos síncronos en la creación y confirmación, mientras que notificaciones y tracking funcionan mejor con eventos asincrónicos. El diseño modular permite escalar componentes críticos (ej. Order Service) sin afectar el resto.



**Ejercicio 3 – Sistema de Biblioteca**

1. Funcionalidades

* Usuarios: registrarse, consultar catálogo, pedir préstamo, renovar, devolver, ver historial.
* Bibliotecario: gestionar usuarios, préstamos, devoluciones y sanciones.
* Administrador: configurar catálogo, generar reportes de uso y disponibilidad.
* Sistema: manejar autenticación, reglas de préstamo, alertas de vencimiento.

2. Modularización (Componentes)

* Frontend: portal para usuarios, bibliotecarios y administradores.
* Auth Service: control de acceso y roles.
* User Service: datos de estudiantes y personal.
* Catalog Service: gestión de libros, autores, categorías, disponibilidad.
* Loan Service: préstamos, renovaciones, devoluciones, sanciones.
* Notification Service: recordatorios de vencimiento y avisos.
* Reporting Service: métricas de uso, estadísticas de préstamos.

3. Interfaces – Conectores

* Frontend ↔ Backend: via REST/GraphQL.
* Servicios internos: comunicación REST/gRPC.
* Eventos asincrónicos: loan. created loan. overdue, book. returned.
* Integraciones externas: sistemas académicos, correo institucional.

4. Análisis de diseño

El sistema pasa de un monolito a una arquitectura modular, separando catálogo, préstamos y usuarios. Esto facilita mantenimiento, evita dependencias rígidas y permite escalar servicios críticos como el de préstamos. Las notificaciones se manejan de forma asincrónica, mejorando la experiencia del usuario y la eficiencia del sistema.

