**Modularización y componetización.**

Elabore para cada uno de los siguientes enunciados:

1. Identifique las funcionalidades (requisitos funcionales)
2. Modularice el sistema que dará solución a las funcionalidades identificando componentes
3. Identifique las interfaces – conectores que permitirán comunicar los componentes
4. Agregue un análisis breve que justifique las decisiones de diseño para el modelo propuesto.
5. Socialice en clase sus propuestas de diseño.

**Ejercicio 1.** Un centro médico requiere un sistema web para que pacientes puedan reservar consultas, médicos gestionen su agenda y los administradores controlen el sistema.

**Ejercicio 2.** Una startup quiere desarrollar una aplicación móvil y web tipo Rappi o Uber Eats, donde los usuarios pueden pedir comida, los restaurantes gestionan pedidos y los repartidores hacen las entregas.

**Ejercicio 3.** Un colegio tiene un sistema heredado de biblioteca que mezcla lógica de préstamo, usuarios, catálogo y reportes en un solo módulo monolítico. El sistema debe ser **refactorizado y rediseñado** usando principios modernos de diseño.

DESARROLLO:

Nombre del estudiante: Brayan Alejandro Rojas Sanchez.

Código: 817148

Fecha: 16/09/2025

**Ejercicio 1 · Centro médico (reservas de citas)**

**1) Requisitos funcionales**

* **RF-001:** Registro/Login de usuarios (pacientes, médicos, administradores).
* **RF-002:** Gestión de perfiles (datos personales, especialidades, horarios médicos).
* **RF-003:** Gestión de agenda médica (cupos, bloqueos, ausencias).
* **RF-004:** Búsqueda de disponibilidad por médico/especialidad/fecha.
* **RF-005:** Reserva de citas médicas.
* **RF-006:** Reprogramación y cancelación de citas.
* **RF-007:** Recordatorios automáticos (correo, SMS, push).
* **RF-008:** Check-in/confirmación de asistencia.
* **RF-009:** Administración de usuarios, catálogos y auditoría.
* **RF-010:** Reportes de ocupación, tiempos de espera y no-shows.

**2) Modularización (componentes)**

* Frontend Paciente
* Frontend Médico
* Frontend Administrador
* API Gateway / BFF
* Auth & Identity
* Agenda & Disponibilidad
* Citas
* Notificaciones
* Reportes & Analytics

**3) Interfaces – conectores**

* **REST API:** /auth/login, /agenda/disponibilidad, /citas, /reportes.
* **Eventos (pub/sub):** AppointmentBooked, AppointmentCancelled, ReminderDue.
* **Externos:** Proveedor SMS/email, FHIR/HL7 (opcional).

**4) Justificación**

El sistema se divide en módulos claros para que cada parte cumpla una función específica: la agenda organiza la disponibilidad, el módulo de citas gestiona reservas y cancelaciones, y notificaciones envía recordatorios. Esto facilita el mantenimiento y evita que un cambio afecte a todo el sistema.  
Además, usar eventos (como cuando alguien reserva una cita) permite que el sistema mande recordatorios o actualice reportes automáticamente, sin interrumpir al usuario.  
Así se logra un sistema **más ordenado, fácil de manejar y que mejora la experiencia de pacientes y médicos**.

**Ejercicio 2 · App tipo Rappi/UberEats**

**1) Requisitos funcionales**

* **RF-001:** Registro/Login (usuarios, restaurantes, repartidores).
* **RF-002:** Gestión de catálogos y menús por restaurante.
* **RF-003:** Creación de pedidos con carrito y direcciones.
* **RF-004:** Procesamiento de pagos (tarjeta, PSE, contraentrega).
* **RF-005:** Asignación de repartidor al pedido.
* **RF-006:** Tracking en tiempo real del pedido.
* **RF-007:** Notificaciones push/SMS de estado del pedido.
* **RF-008:** Calificaciones y reseñas.
* **RF-009:** Panel de gestión para restaurantes.
* **RF-010:** Panel de gestión para repartidores.
* **RF-011:** Administración general (tarifas, comisiones, reportes).

**2) Modularización (componentes)**

* Frontend Usuario (móvil/web)
* Frontend Restaurante (web)
* Frontend Repartidor (móvil)
* Admin Console
* API Gateway / BFF
* Catálogo & Menús
* Pedidos & Carrito
* Pagos & Facturación
* Dispatch/Matching
* Geolocalización & Routing
* Tracking & WebSockets
* Notificaciones
* Ratings & Reviews
* Pricing/Fees & Promos
* Reportes & Fraude

**3) Interfaces – conectores**

* **REST API:** /restaurantes, /menu, /pedidos, /pagos.
* **WebSockets:** /ws/tracking (estado en tiempo real).
* **Eventos:** OrderPlaced, RiderAssigned, OrderDelivered.
* **Externos:** Pasarela de pagos, mapas, push.

**4) Justificación**

El sistema se organiza por módulos para que cada actor (usuario, restaurante, repartidor) tenga su espacio bien definido.  
El uso de WebSockets permite que el cliente vea el pedido en vivo en el mapa, algo esencial en este tipo de apps.  
Separar procesos como pagos, pedidos y asignación de repartidores hace que si un módulo falla, no colapse todo el sistema.  
Este diseño permite que la aplicación sea **rápida, confiable y fácil de escalar** a medida que crezca el número de usuarios.

**Ejercicio 3 · Biblioteca escolar (refactor de monolito)**

**1) Requisitos funcionales**

* **RF-001:** Gestión de usuarios (estudiantes, docentes, admins).
* **RF-002:** Control de roles y permisos.
* **RF-003:** Gestión del catálogo de libros (títulos, autores, ejemplares).
* **RF-004:** Préstamos de libros.
* **RF-005:** Renovaciones y devoluciones.
* **RF-006:** Reservas de ejemplares.
* **RF-007:** Políticas de préstamo (tiempos, sanciones, límites).
* **RF-008:** Cálculo y gestión de multas.
* **RF-009:** Generación de reportes (libros más prestados, morosidad).
* **RF-010:** Notificaciones de vencimiento y atraso.
* **RF-011:** Integración con directorio académico/SIS.

**2) Modularización (componentes)**

* Auth & Directory
* Usuarios & Roles
* Catálogo
* Circulación (préstamos, devoluciones, reservas)
* Políticas
* Multas & Cobranzas
* Reportes & Analytics
* Notificaciones
* Capa de Integración con sistema legado

**3) Interfaces – conectores**

* **REST API:** /catalogo/search, /prestamos, /reservas, /reportes.
* **Eventos:** LoanStarted, LoanOverdue, LoanReturned.
* **Externos:** LDAP/SIS, correo, pagos en línea.

**4) Justificación**

El sistema original era un monolito difícil de mantener. Con la nueva división en módulos (catálogo, circulación, usuarios, políticas) cada parte se puede actualizar sin afectar el resto.  
Se usa una capa de integración para migrar poco a poco y no dejar de prestar el servicio mientras se moderniza.  
Además, definir las políticas como configurables evita tener que cambiar código cada vez que se modifica una regla.  
Esto hace que la biblioteca tenga un sistema **más flexible, seguro y fácil de mejorar a futuro**.