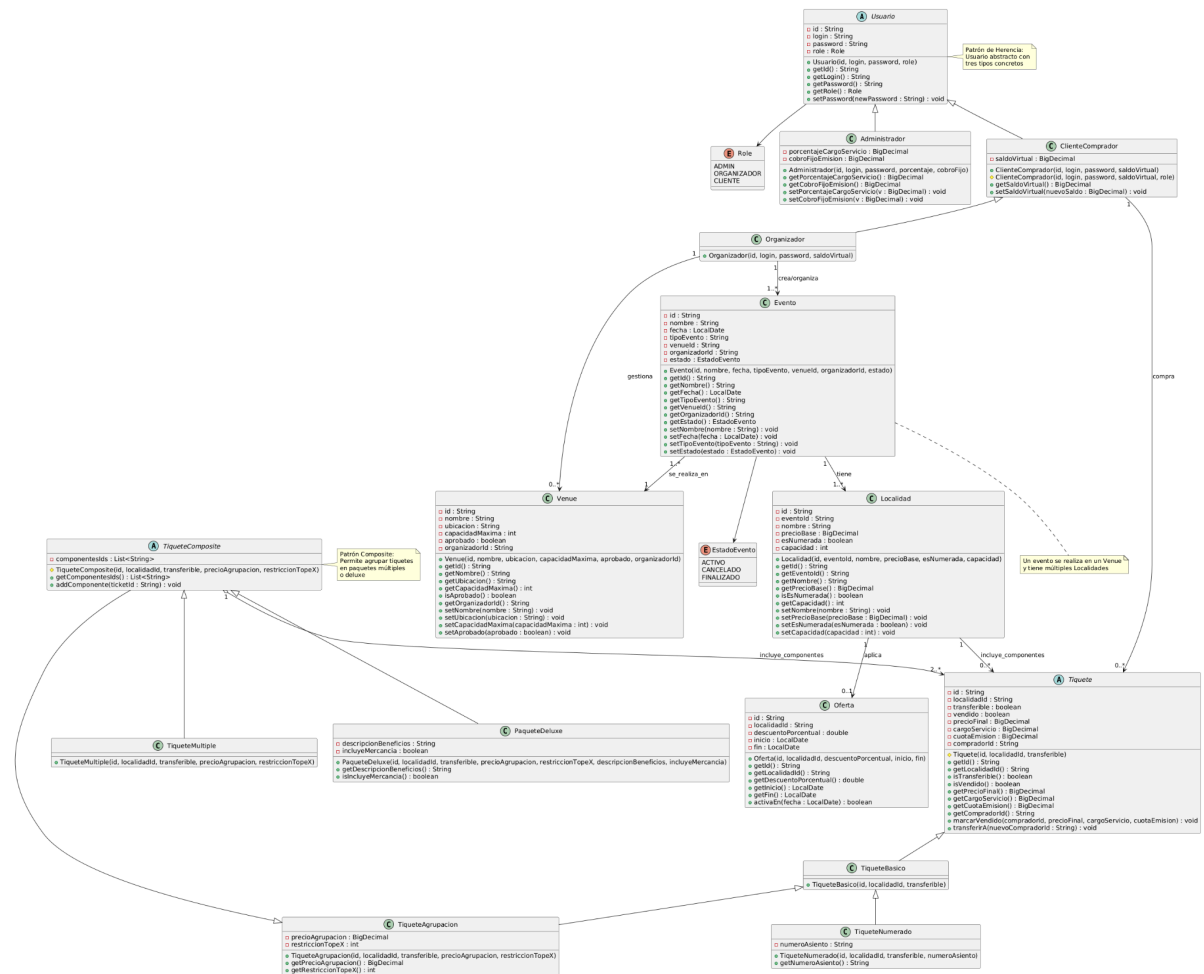


Proyecto DP00

Entrega 1:

1. Diagrama



2. Restricciones Funcionales:

a. Compra de Tiquetes:

Un cliente puede comprar varios tiquetes para cualquier evento.

Dependiendo del evento puede existir un límite de tiquetes que se pueden comprar por transacción.

Las restricciones de los tiquetes múltiples como palcos o pases de temporada aplican para todo el paquete.

b. Tipos de Tiquetes:

1. Tiquetes simples: acceso a un evento a una localidad específica.
2. Tickets numerados: acceso a un evento con derecho a un asiento único.
3. Paquetes Deluxe: incluyen beneficios adicionales pero no son transferibles.

c. Transferencias:

1. Los tiquetes pueden transferirse entre clientes.
2. Los tiquetes de paquetes deluxe no pueden ser transferidos por cuestión de seguridad.
3. Para transferir se requieren los datos de ingreso tanto del receptor como del emisor.

d. Eventos y Localidades:

1. Cada evento debe tener un organizador.
2. Cada evento debe estar asociado a un lugar aprobado.
3. Cada lugar no puede tener más de un evento por fecha.
4. Las localidades definen el precio y las características de cada tiquete.

e. Reembolsos:

1. El administrador del evento puede cancelar y autorizar reembolsos.
2. Los reembolsos sólo se hacen al saldo virtual del usuario.
3. En cancelaciones por el organizador solo se reembolsa el precio base.

3. Restricciones de dominio

Usuarios

- Login único y contraseña obligatoria.
- Saldo inicial = 0 (o 500 en ambiente demo).
- Solo **clientes y organizadores** pueden comprar; el administrador **no**.
- Transferencias requieren autenticación y sólo aplican a tiquetes no Deluxe.

Eventos y venues

- Un evento por **venue y fecha**.
- Los venues deben estar **aprobados** por el administrador.
- Capacidad > 0 y localidades coherentes con los asientos.

Localidades y precios

- Todas las boletas de una misma localidad comparten precio base.
- Descuentos u ofertas se expresan como porcentaje (0–100 %).
- Se aplica siempre **el mejor descuento disponible**.
- Cargos globales: porcentaje de servicio + cuota fija de emisión.

Tiquetes

- ID único por tiquete.
- Tipos: simples, numerados, múltiples, deluxe.
- Los **Deluxe** no se transfieren.
- Las agrupaciones respetan límite máximo **por agrupación**, no por tiquete.

Reembolsos y cancelaciones

- Cancelación por admin → reembolso total – cuota de emisión.
- Cancelación por organizador → solo valor base.
- Los reembolsos se abonan al **saldo virtual**.

4. Programas de Prueba:

a. Creación de usuarios:

1. Registro efectivo de clientes, organizadores y administradores.
2. Validación de login y contraseña.

b. Creación de eventos:

1. Asociación con un lugar aprobado.
2. Configuración de fechas, localidades y tipos de tiquetes.

c. Compra de tiquetes:

1. Compra efectiva de todos los tipos de tiquetes.
2. Aplicación efectiva de las restricciones por transacción.

d. Transferencia de tiquetes:

1. Transferencia entre usuarios con validación.
2. Aplicación efectiva de las restricciones.

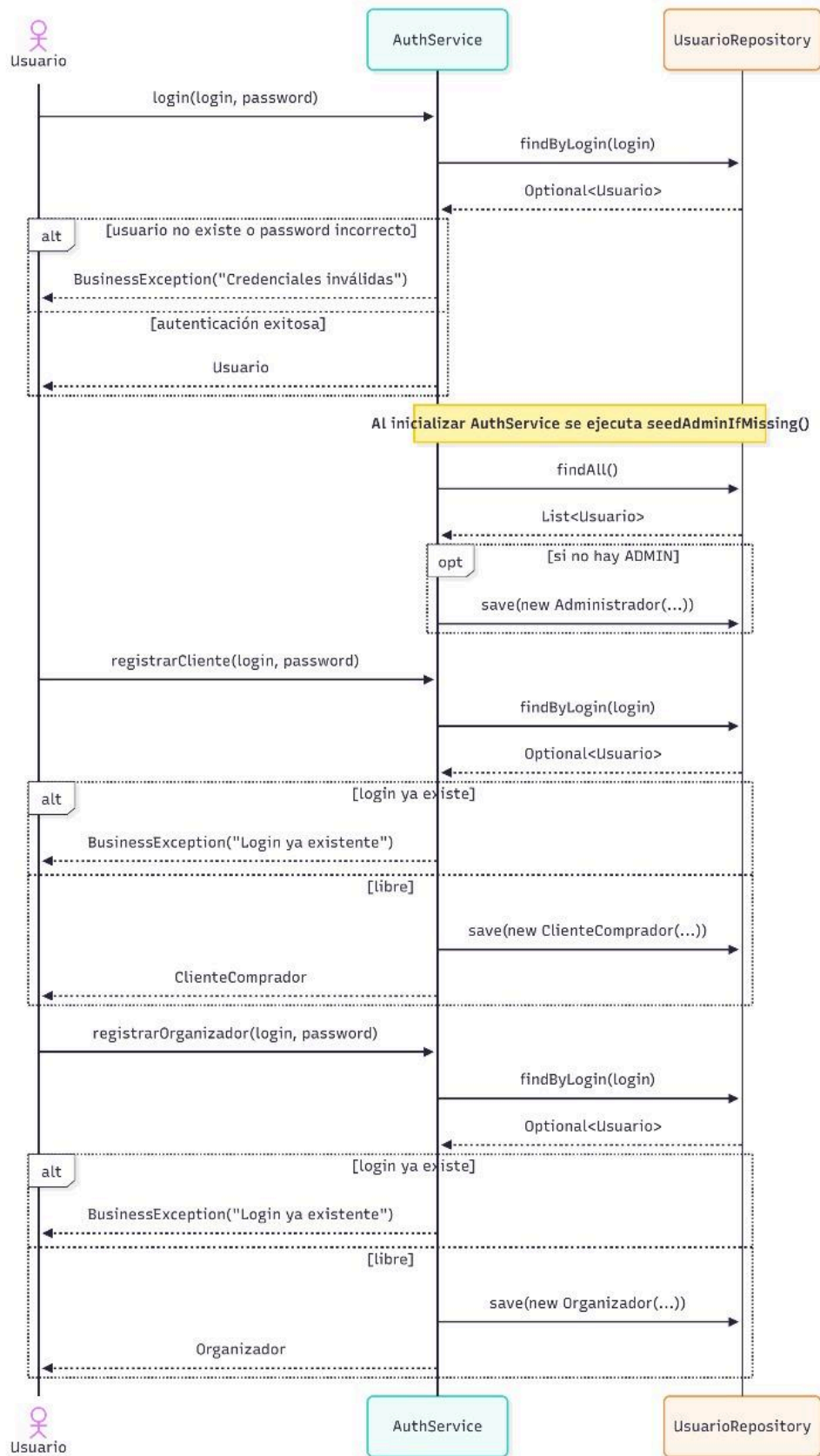
e. Reembolsos:

1. Simulación de cancelación de eventos por administrador.
2. Reembolso a saldo virtual de cada usuario.

f. Consultas financieras:

1. Visualización de ganancias por organizadores y eventos.
2. Cálculo de ingresos totales.

5. Diagramas de secuencia



1. Proceso de login

Secuencia:

1. El usuario invoca login(login, password) en AuthService.
2. AuthService consulta UsuarioRepository.findByLogin(login), que devuelve un Optional<Usuario>.
3. Existen dos caminos alternativos:
 - [Usuario no existe o password incorrecto]:
Se lanza una BusinessException("Credenciales inválidas").
 - [Autenticación exitosa]:
Se retorna el objeto Usuario correspondiente.

Puntos clave:

- El manejo de errores se hace con un bloque alt (alternativa).
- La validación de credenciales se realiza dentro de AuthService.
- UsuarioRepository solo recupera la información, no válida contraseñas.

2. Inicialización del AuthService

Evento:

Al instalar AuthService, se ejecuta automáticamente el método seedAdminIfMissing().

Flujo:

1. AuthService llama a UsuarioRepository.findAll(), obteniendo una lista de usuarios.
2. Si no existe un usuario con rol ADMIN, entonces:
 - Se ejecuta save(new Administrador(...)) para crear el administrador por defecto.

Propósito:

Garantizar que siempre exista un usuario administrador en el sistema.

3. Registro de cliente (registrarCliente)

Secuencia:

1. El usuario llama registrarCliente(login, password) en AuthService.
2. Se valida si el login ya existe con UsuarioRepository.findByLogin(login).
3. Dos caminos posibles:
 - [Login ya existe]:
Se lanza BusinessException("Login ya existente").
 - [Libre]:
Se guarda un nuevo usuario tipo ClienteComprador mediante UsuarioRepository.save(new ClienteComprador(...)).

Resultado:

Devuelve el nuevo objeto ClienteComprador.

4. Registro de organizador (registrarOrganizador)

Secuencia:

1. Similar al caso anterior:
AuthService llama findByLogin(login).
2. Si ya existe un usuario con ese login → BusinessException("Login ya existente").
3. Si no existe → save(new Organizador(...)).

Resultado:

Devuelve el nuevo objeto Organizador.

6. Justificación

Arquitectura (capas y justificación)

- Estilo: Arquitectura en capas (Layered Architecture) con separación estricta de responsabilidades.
- Presentación: *AppCli* es la interfaz de usuario. Solo I/O y delegación a servicios de aplicación.
- Aplicación: servicios orquestadores (*AuthService*, *AdminService*, *OrganizerService*, *CatalogService*, *PurchaseService*, *TransferService*, *OfferService*) coordinan transacciones y reglas de caso de uso; no contienen lógica de persistencia ni detalles de UI.
- Dominio: entidades y comportamiento del negocio:
Entidades: *Usuario* y subtipos, *Venue*, *Evento*, *Localidad*, *Asiento*, *Oferta*, *Tiquete*.
Servicios de dominio: *PricingPolicy* / *DefaultPricingPolicy*, *OfferEngine*, *TicketFactory*, *PriceBreakdown*.
Invariantes: venta y transferencia de *Tiquete*, cálculo de precios y descuentos.
- Infraestructura: adaptaciones a CSV: *infrastructure.csv.repository.*, *infrastructure.csv.mappers.*, utilidades *CsvUtil*, *CsvFormat*, *CsvPaths*.
- Reglas de dependencia: Presentación → Aplicación → Dominio. Infraestructura implementa interfaces del dominio y se ensambla desde *AppContext* (anti-corrupción entre capas y wiring centralizado).
- Diagramas de apoyo: *arquitectura-componentes.mmd*, *paquetes.mmd*, *despliegue.mmd*.

Patrones de diseño (concretos)

- Repository: interfaces en *domain.repository.* e implementaciones en *infrastructure.csv.repository.* (p. ej., *UsuarioRepository* → *CsvUsuarioRepository*). Permite cambiar CSV por otra tecnología sin tocar aplicación/dominio.
- Strategy: *PricingPolicy* define el contrato de cálculo de precio; *DefaultPricingPolicy* es la estrategia por defecto. Facilita nuevas políticas (promos, impuestos, etc.).
- Factory: *TicketFactory* crea *TiqueteBasico*, *TiqueteNumerado*, *TiqueteAgrupacion*, *TiqueteMultiple*, *PaqueteDeluxe* centralizando la construcción y generación de ids.
- Composite: *TiqueteCompuesto* = (*TiqueteMultiple*, *PaqueteDeluxe*). Permite tratar paquetes y agrupaciones como tiquetes compuestos.

- Service Layer: *application.service*. encapsula casos de uso y coordina transacciones.
- Value Object: *PriceBreakdown* es inmutable y describe el resultado de precio.
- (Infraestructura) Mapper: *infrastructure.csv.mappers*. actúa como traductor a/desde líneas CSV.

Por qué estos patrones

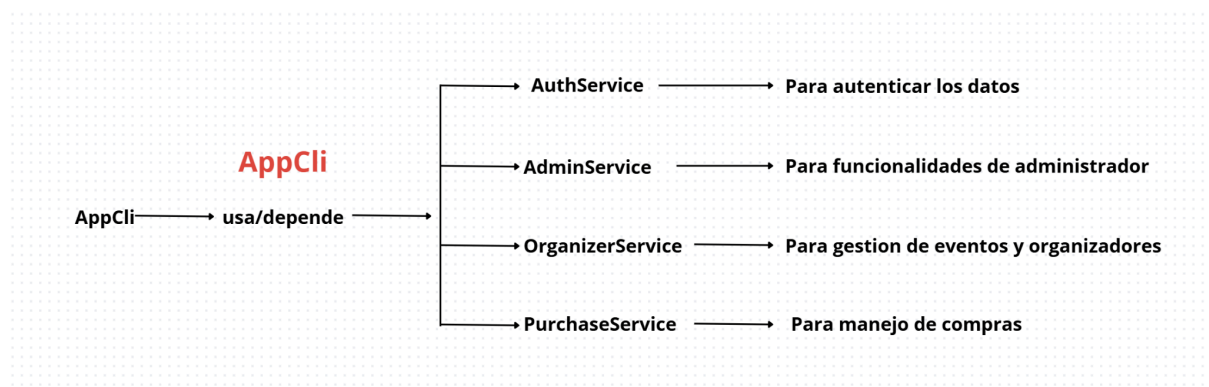
- Aíslan el dominio de detalles tecnológicos (CSV hoy, DB mañana) y evitan el acoplamiento.
- Habilitan la extensibilidad: nuevas estrategias de precio, nuevos repositorios.
- Mejor testabilidad: dominios y servicios se prueban con dobles de *Repository* / *PricingPolicy*.

7. Diagrama de clases de Alto nivel

Una clase de Alto nivel se refiere a las clases que encapsulan comportamientos generales, coordinan múltiples componentes o representan conceptos centrales del dominio sin depender en detalles de implementación. Debido a esto principalmente las clases de alto nivel en nuestro proyecto hacen parte principalmente del paquete de “service”. Sin embargo consideramos que la clase *App.Cli* también hace parte de las clases de alto nivel.

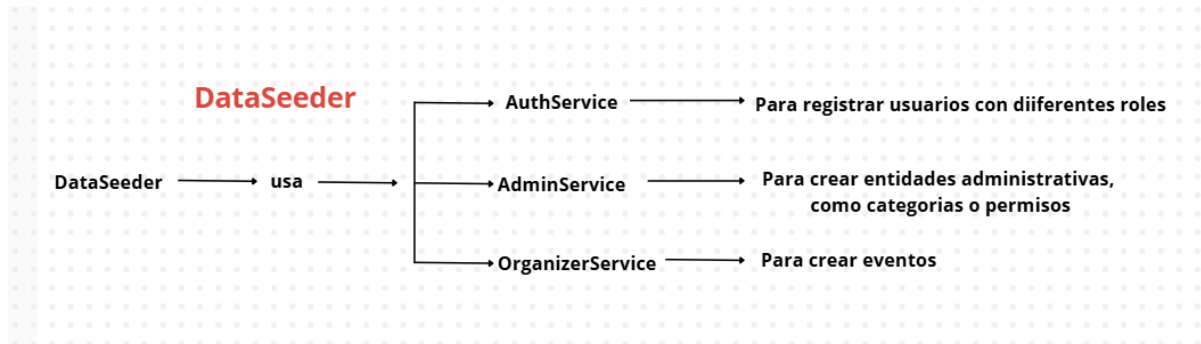
Appcli:

Es la clase principal del sistema ya que esta interactúa con las diferentes clases del paquete “service”. Su función principal es recibir comandos o peticiones del usuario y delegar la lógica al servicio correspondiente.



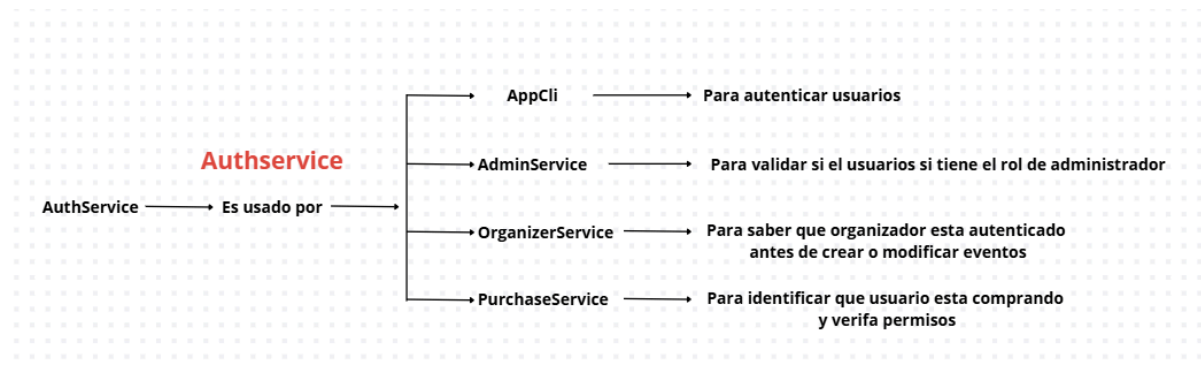
DataSeeder:

Es la clase encargada de inicializar datos en el sistema (usuarios, eventos, venues, organizadores, etc). Esta clase tiene como objetivo cargar el sistema con la información inicial o de prueba. Esta clase usa diferentes clases en el paquete “service” para crear y registrar los datos.



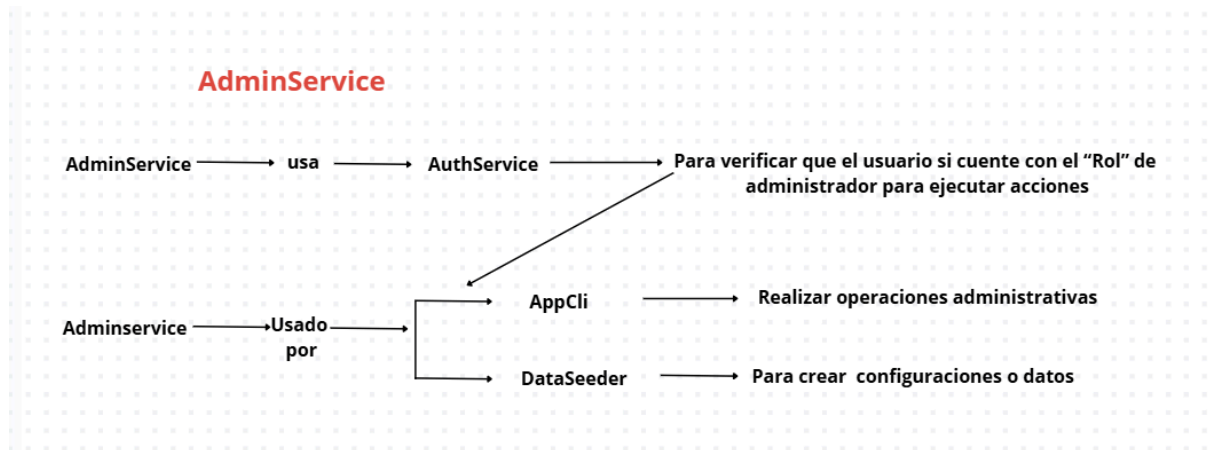
AuthService:

Es la clase encargada de manejar la autenticación y registro de usuarios. Esta define los roles del usuario como “Admin”, “Organizador”, “Cliente”. Esta clase es un servicio central que provee autenticación y los roles de los usuarios lo que desbloquea diferentes funcionalidades a través de la implementación.



AdminService:

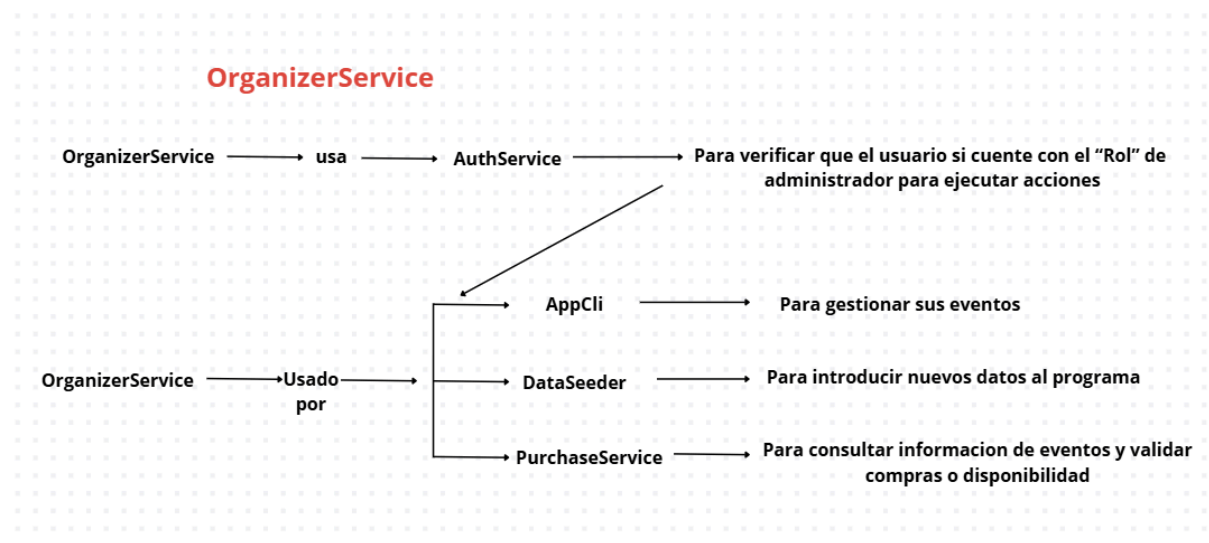
Esta clase maneja operaciones que son exclusivas del administrador como crear usuarios, gestionar eventos o revisar estadísticas.



Esta clase depende de AuthService para validar el acceso y es usado por AppCli y por DataSeeder para ejecutar tareas administrativas.

OrganizerService:

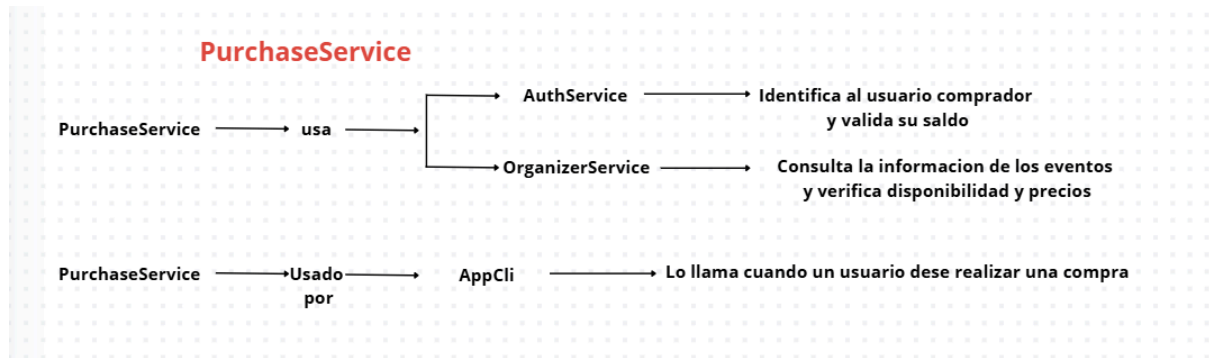
Esta clase está encargada de la gestión de eventos o actividades por parte de los organizadores. Esta permite crear, modificar o eliminar eventos.



OrganizerService depende de Authservice para reconocer al organizador y es utilizado por AppCli, DataSeeder, y PurchaseService

PurchaseService:

Esta clase maneja la compra de entradas o tickets dentro del sistema. Usa los datos de los usuarios autenticados y eventos creados por organizadores.



PurchaseService depende de AuthService y OrganizerService para procesar correctamente una compra. Es utilizado directamente por AppCli.