

# Documento de Diseño

## Proyecto 2: MasterTicket Marketplace de Reventa

### Índice

1.	Contexto	del	problema
1.1.	Descripción	general	sistema
1.2.		Problemática	principal
1.3.	Actores Principales		
2.	Alcance de la solución		
2.1.	Funcionalidades incluidas		
2.2.	Funcionalidades futuras		
2.3.	Supuestos Generales		
3.	Restricciones y limitaciones		
3.1.	Restricciones/limitaciones técnicas y operativas		
4.	Objetivos		
4.1.	Objetivo general		
4.2.	Objetivos específicos		
5.	No objetivos		
6.	Elementos de la solución		
6.1.	Descripción general del diseño		
6.2.	Clases principales del proyecto		
6.3.	Métodos y atributos relevantes		
7.	Diagrama de clases de alto nivel		
7.1.	Estructura general		
7.2.	Relaciones principales		
8.	UML		
8.1.	Descripción del diagrama		
9.	Diagramas de secuencia		
9.1.	Diagrama de Secuencia del Cliente		

9.2. Diagrama de Secuencia del Organizador

9.3. Diagrama de Secuencia del Administrador

9.4. Diagrama de Secuencia: Publicación y venta de oferta en el Marketplace

9.5. Diagrama de Secuencia: Consulta del LogRegistro por el Administrador

## **10. Información sobre la persistencia**

10.1. Estructura de carpetas

10.2. Formato de los archivos

10.3. Esquema de datos

10.4. Ejemplo de persistencia

## **11. Alternativas de solución consideradas**

11.1. Opciones evaluadas

11.2. Razones para no implementarlas

## **12. Preocupaciones y riesgos de la solución**

12.1. Riesgos técnicos

12.2. Riesgos funcionales

12.3. Riesgos de mantenimiento

## **13. Conclusiones**

# **1. Contexto del Problema**

## *1.1 Descripción General del Sistema*

El proyecto consiste en el diseño e implementación de una plataforma para la compra, gestión y administración de eventos de diferentes tipos. El sistema debe permitir la interacción de tres actores principales:

- Clientes naturales, quienes ingresan al sistema para buscar, cotizar y comprar tickets de eventos específicos.
- Organizadores, responsables de la creación y administración de eventos, manejo de venues, definición de precios y configuración de ofertas.
- Administradores, encargados de la gestión general del sistema, incluyendo control financiero, sobrecargos, cancelaciones y supervisión global de las operaciones.

La función principal del sistema es la venta de tickets, ya que constituye la base operativa y la funcionalidad más utilizada. El sistema debe manejar diferentes tipos de tickets y paquetes,

adaptándose a los distintos eventos. Toda la información del sistema debe ser persistente, almacenándose en archivos planos (JSON) para garantizar la recuperación de datos y trazabilidad entre ejecuciones.

### *1.2 Problemática Principal*

El principal reto es construir una plataforma integral y flexible capaz de manejar la compra de distintos tipos de tickets para eventos diversos, administrados por diferentes organizadores, y bajo la supervisión centralizada de un administrador. Además, la solución debe garantizar:

- Una estructura lógica y modular para las relaciones entre eventos, usuarios y tickets.
- Persistencia eficiente y confiable sin uso de bases de datos externas.
- Capacidad de escalar a nuevas funcionalidades sin comprometer el diseño base.
- Mantenimiento sencillo y extensible para futuras versiones.

### *1.3 Actores Principales*

- **Cliente:** actor principal del sistema. Compra, visualiza y gestiona sus tickets dentro de su cuenta personal.
- **Organizador:** genera los eventos, define precios, localidades y administra el venue.
- **Administrador:** supervisa la plataforma, aprueba eventos, gestiona finanzas y ejecuta cancelaciones o reembolsos.
- **Programadores:** responsables de la arquitectura, desarrollo y mantenimiento técnico del sistema.

## **2. Alcance de la Solución**

### *2.1 Funcionalidades incluidas*

La versión actual del sistema permite:

- Registro y gestión de usuarios con diferentes roles: clientes, organizadores y administradores.
- Compra directa de tickets desde la plataforma.
- Historial de tickets adquiridos o revendidos, accesible para cada usuario.
- Gestión de saldo virtual, que funciona como un sistema de puntos y reembolsos automáticos o manuales, según tres escenarios:
  - Cancelación del evento por parte del administrador.
  - Cancelación por parte del organizador.
  - Solicitud individual del usuario.
- Reventa de tickets entre usuarios mediante el Marketplace, que permite publicar ofertas, realizar contraofertas y concretar ventas de forma segura dentro del sistema.

- Registro automático de todas las operaciones en el LogRegistro, lo que garantiza trazabilidad.
- Supervisión del administrador, quien puede eliminar ofertas y consultar el historial de transacciones.

## *2.2 Funcionalidades Futuras*

En el futuro se espera:

- Integrar una base de datos relacional para optimizar la gestión de usuarios, eventos y transacciones del Marketplace.
- Incorporar autenticación externa para mejorar la seguridad y simplificar el acceso.
- Desarrollar una interfaz gráfica (UI/UX) más intuitiva que permita la navegación visual de eventos, compras y reventas.
- Agregar análisis de actividad y reportes automáticos, utilizando la información del LogRegistro para generar métricas de uso, ventas y confiabilidad.
- Incluir nuevos tipos de tickets y usuarios (como Premium o VIP), con beneficios personalizados y funciones exclusivas dentro del sistema.

## *2.3 Supuestos Generales*

- Solo la aplicación accede a los archivos de datos.
- Todas las fechas son válidas y dentro del mismo uso horario.
- Los usuarios ingresan credenciales correctas.
- No existe concurrencia ni edición simultánea.
- Los archivos de persistencia siempre existen o se crean al iniciar.
- Toda la información financiera se maneja en una sola moneda (COP).

# **3. Restricciones y limitaciones**

## *3.1 Restricciones/ limitaciones técnicas y operativas*

### **Restricciones técnicas y operativas**

1. Lenguaje y entorno de desarrollo: La solución está completamente desarrollada en Java, sin dependencias externas ni integración con APIs. Esto garantiza independencia del entorno y control total sobre la estructura del código.
2. Persistencia basada en JSON: El sistema no utiliza una base de datos tradicional. La información se almacena en archivos planos con formato JSON, lo que permite mantener la simplicidad, legibilidad y dominio total sobre los datos registrados.

3. Ejecución en entorno local: El backend opera únicamente en modo local, sin conexión a servicios web o servidores en la nube. Por tanto, la plataforma no puede ejecutarse en línea ni realizar comunicación externa.
4. Interfaz de usuario mínima: El proyecto se centra en la lógica de negocio y la arquitectura de clases. No incorpora una interfaz gráfica avanzada ni elementos visuales de interacción. La funcionalidad es completamente operativa desde la lógica interna.
5. Sin integración de pagos ni manejo de divisas: La plataforma no incluye pasarelas de pago ni gestión de monedas. El proceso de compra de tiquetes es puramente funcional y no contempla transacciones económicas reales, lo que elimina la necesidad de integrar APIs externas.

### **Limitaciones Funcionales**

1. Tipos de usuarios: El sistema admite únicamente tres roles: Cliente, Organizador y Administrador. Todos los clientes se tratan de manera uniforme; no se contemplan perfiles diferenciados por edad, condición o accesibilidad.
2. Eventos estáticos: Los eventos son simulados y no están vinculados con plataformas o proveedores reales. Las actualizaciones solo se reflejan al recargar los datos, sin sincronización en tiempo real.
3. Ausencia de control físico de acceso: El sistema no administra inventarios físicos ni mecanismos de validación en espacios reales (como torniquetes o códigos QR). Se asume que el ID del tiquete garantiza su autenticidad dentro del entorno lógico. No se contemplan medidas contra la reventa o duplicación fuera del sistema.

## **4. Objetivos**

### *4.1 Objetivo General*

Desarrollar una plataforma para la creación, gestión y venta de tiquetes, que asegure la persistencia de los datos y el control adecuado de las operaciones por cada tipo de usuario.

### *4.2 Objetivos Específicos*

- Modelar un dominio orientado a objetos que represente con precisión las entidades del sistema de boletería, incluyendo usuarios, eventos, tiquetes, ofertas y registros de operaciones.
- Desarrollar una lógica de negocio coherente, capaz de gestionar procesos clave como la creación de eventos, la venta y reventa de tiquetes, las transferencias, cancelaciones y reembolsos, manteniendo las reglas definidas para cada tipo de usuario.
- Diseñar y extender el mecanismo de persistencia mediante archivos JSON estructurados, incorporando nuevas entidades como el Marketplace y el LogRegistro, para asegurar la trazabilidad de la información entre ejecuciones.

- Implementar casos de prueba y programas de demostración funcional que validen el comportamiento del sistema desde la consola, mostrando el flujo de compra, reventa y registro de eventos de manera clara y verificable.
- Garantizar la escalabilidad del diseño, de modo que la arquitectura actual sirva como base para futuras ampliaciones, incluyendo integración con bases de datos, pasarelas de pago, autenticación externa y reportes automáticos basados en los datos del LogRegistro.

## 5. No objetivos

El presente proyecto se enfoca en la implementación funcional del sistema de boletería y gestión de eventos, por lo cual se han definido las características que no forman parte del alcance actual. Estas exclusiones permiten mantener los objetivos realistas y priorizar el desarrollo de la lógica del sistema.

1. No se busca integrar pagos reales: El sistema no contempla la conexión con plataformas externas de pago (como PayU, Stripe o PayPal). Las transacciones son simuladas de manera interna, sin involucrar dinero real ni validación bancaria.
2. No se implementan sistemas avanzados de autenticación y seguridad: El sistema no incluye cifrado de contraseñas, verificación en dos pasos, ni manejo de sesiones seguras. Se asume que los usuarios que ingresan son legítimos y que las credenciales son válidas.
3. No se desarrolla una interfaz gráfica profesional (UI/UX): La interacción con el sistema se realiza únicamente por consola. No se consideran elementos visuales, diseño de experiencia de usuario o componentes gráficos.
4. No se contempla el manejo de múltiples divisas, idiomas o regiones: Todas las operaciones se ejecutan en una sola moneda (COP) y en idioma español. No se incluye conversión automática de valores ni localización del contenido.
5. No se integra control físico o validación presencial de tiquetes.: La verificación de acceso a eventos no depende de códigos QR, torniquetes ni dispositivos externos.

## 6. Elementos de la Solución

### 6.1 Descripción general del diseño

La solución está compuesta por 9 clases principales que conforman la lógica, complementadas por clases auxiliares dedicadas a la persistencia de datos y a la consola.

Entre las más relevantes se encuentran Usuario, Organizador, Evento, Tiquete y Administrador. Cada entidad asume responsabilidades específicas: el organizador administra los eventos, el usuario adquiere tiquetes y el administrador supervisa el funcionamiento general de la plataforma.

En la Entrega Única del Proyecto 2, el modelo se amplía con nuevas clases: Marketplace, Oferta y LogRegistro, que permiten la reventa de boletas entre usuarios, la gestión de contraofertas y la

trazabilidad completa de las operaciones mediante un registro de eventos. Con ello, los usuarios pueden comprar, vender y negociar tiquetes directamente dentro de MasterTicket.

## 6.2 Clases Principales del proyecto

Cada clase cumple una función específica dentro de la lógica del programa, permitiendo una interacción ordenada y eficiente entre los diferentes componentes. Estas son las clases actuales:

- **Usuario:** Representa al cliente del sistema. Puede registrarse, iniciar sesión, comprar tiquetes y consultar sus compras.
- **Organizador:** Es el encargado de crear y administrar los eventos; y puede definir la información de cada evento, modificarla o eliminarla.
- **Evento:** Contiene todos los datos relacionados con un evento, como nombre, fecha, lugar, capacidad y tiquetes disponibles.
- **Tiquete:** Registra la información de una entrada específica, incluyendo su número, precio, estado y el evento al que pertenece.
- **Administrador:** Supervisa el funcionamiento general del sistema. Puede revisar la información de usuarios, eventos y tiquetes, y realizar ajustes si es necesario.
- **PersistenciaJSON:** Se encarga de guardar y cargar la información de las clases principales en archivos JSON.
- **Marketplace:** administra la reventa de boletas entre clientes, gestionando las ofertas activas, las contraofertas y las transacciones aceptadas o canceladas.
- **Oferta:** representa una boleta listada en el Marketplace. Contiene la información del tiquete, su vendedor, precio y estado.
- **LogRegistro:** Lleva una bitácora completa de todas las operaciones realizadas dentro del Marketplace. *Nota:* solo puede ser consultado por el administrador.

## 6.3 Métodos y atributos Relevantes

Cada clase del sistema cuenta con un conjunto de atributos y métodos que permiten representar su comportamiento y las relaciones con las demás entidades. A continuación, se describen los más importantes:

### Clase Usuario

- Atributos:
  - idUsuario: Identificador único del usuario.
  - nombre: Nombre completo del usuario.
  - correo: Correo electrónico asociado a la cuenta.
  - saldoVirtual: Saldo disponible para compras o reembolsos.
  - tiquetesComprados: Lista de tiquetes adquiridos por el usuario.
- Métodos:

- comprarTiquete(Evento evento, TipoTiquete tipo): Permite al usuario adquirir un tiquete.
- consultarHistorial(): Muestra el historial de compras del usuario.
- solicitarReembolso(Tiquete tiquete): Envía una solicitud de devolución al administrador.
- aumentarSaldoVirtual(double monto): Se utiliza cuando el usuario recibe un pago por la venta de una boleta en el Marketplace o un reembolso autorizado por el administrador.
- disminuirSaldoVirtual(double monto): Se utiliza cuando el usuario realiza una compra de boleta o una contraoferta aceptada dentro del Marketplace.

### Clase Organizador

- Atributos:
  - idOrganizador: Identificador único.
  - nombre: Nombre del organizador.
  - eventosCreados: Lista de eventos que administra.
- Métodos:
  - crearEvento(String nombre, Date fecha, Venue venue): Registra un nuevo evento en el sistema.
  - cancelarEvento(Evento evento): Permite cancelar eventos propios.
  - modificarEvento(Evento evento): Actualiza información de un evento existente.

### Clase Evento

- Atributos:
  - idEvento: Identificador del evento.
  - nombreEvento: Nombre o título del evento.
  - fecha: Fecha del evento.
  - hora: Hora de inicio.
  - venue: Lugar donde se realizará.
  - tiquetesDisponibles: Lista de tiquetes que pueden ser adquiridos.
- Métodos:
  - generarTiquetes(int cantidad, double precio): Crea los tiquetes asociados al evento.
  - obtenerDisponibilidad(): Retorna el número de tiquetes aún disponibles.
  - calcularIngresos(): Calcula el total recaudado por el evento.

### Clase Tiquete

- Atributos:
  - idTiquete: Código único del tiquete.



- precioBase: Valor original del tickete.
- cargoServicio: Tarifa adicional por emisión.
- evento: Referencia al evento asociado.
- estado: Indica si el tickete está activo, usado o reembolsado.
- Métodos:
  - calcularPrecioFinal(): Retorna el precio total (base + cargos).
  - transferir (Usuario nuevoPropietario): Transfiere el tickete a otro usuario.
  - reembolsar (): Gestiona el proceso de devolución del valor del tickete.

## **Clase Administrador**

- Atributos:
  - idAdministrador: Identificador del administrador.
  - nombre: Nombre del administrador.
  - eventosSupervisados: Lista de eventos bajo su control.
- Métodos:
  - aprobarReembolso(Tickete tickete): autoriza o rechaza una devolución solicitada por un usuario.
  - cancelarEvento(Evento evento): cancela un evento y gestiona automáticamente los reembolsos correspondientes.
  - generarReporteGeneral(): genera informes consolidados sobre ventas, cancelaciones, reembolsos y actividad del Marketplace.
  - consultarLog(): accede al historial completo de operaciones registradas en el sistema (LogRegistro).
  - eliminarOferta(int idOferta): elimina una oferta activa del Marketplace, registrando la acción en el LogRegistro.

## **Clase PersistenciaJSON**

- Atributos:
  - rutaArchivo: Ubicación donde se guarda la información.
- Métodos:
  - guardarDatos(Object objeto, String tipo): Almacena los datos de una clase en formato JSON.
  - cargarDatos(String tipo): Recupera la información almacenada en los archivos.
  - actualizarArchivo(): Reescribe la información para mantenerla sincronizada.
  - guardarMarketplace(Marketplace marketplace): guarda las ofertas, contraofertas y transacciones del Marketplace.
  - cargarMarketplace(): carga el estado actual del Marketplace.
  - guardarLog(LogRegistro log): registra en log.json los eventos del sistema (ofertas creadas, ventas, cancelaciones, eliminaciones).

- cargarLog(): recupera el historial completo de eventos registrados en el log.

## Clase Marketplace

- Atributos:
  - ofertasActivas: lista de objetos Oferta disponibles.
  - transacciones: lista de operaciones completadas.
  - logRegistro: historial general del sistema.
- Métodos:
  - publicarOferta(Tiquete tiquete, double precio): permite que un cliente ponga una boleta a la venta.
  - eliminarOferta(int idOferta): retira una oferta del mercado.
  - contraofertar(int idOferta, double nuevoPrecio, Usuario comprador): genera una contraoferta.
  - aceptarContraoferta(int idOferta): concreta la venta y transfiere el tiquete.
  - registrarTransaccion(): actualiza saldo y escribe en el log.

## Clase Oferta

- Atributos:
  - idOferta, tiquete, vendedor, precioActual, estado, historialCambios.
- Métodos:
  - actualizarPrecio(double nuevoPrecio)
  - marcarVendida()
  - cancelarOferta()

## Clase LogRegistro

- Atributos:
  - fechaHora, tipoEvento, usuario, detalle.
- Métodos:
  - registrarEvento(String tipo, Usuario actor, String detalle)
  - mostrarHistorial()
  - exportarLog()

# 7. Diagrama de Clases de Alto Nivel

## 7.1 Estructura General

El diagrama de clases está separado entre la lógica de negocio y la persistencia de datos. En la lógica de negocio se encuentran las clases que gestionan las operaciones principales del sistema:

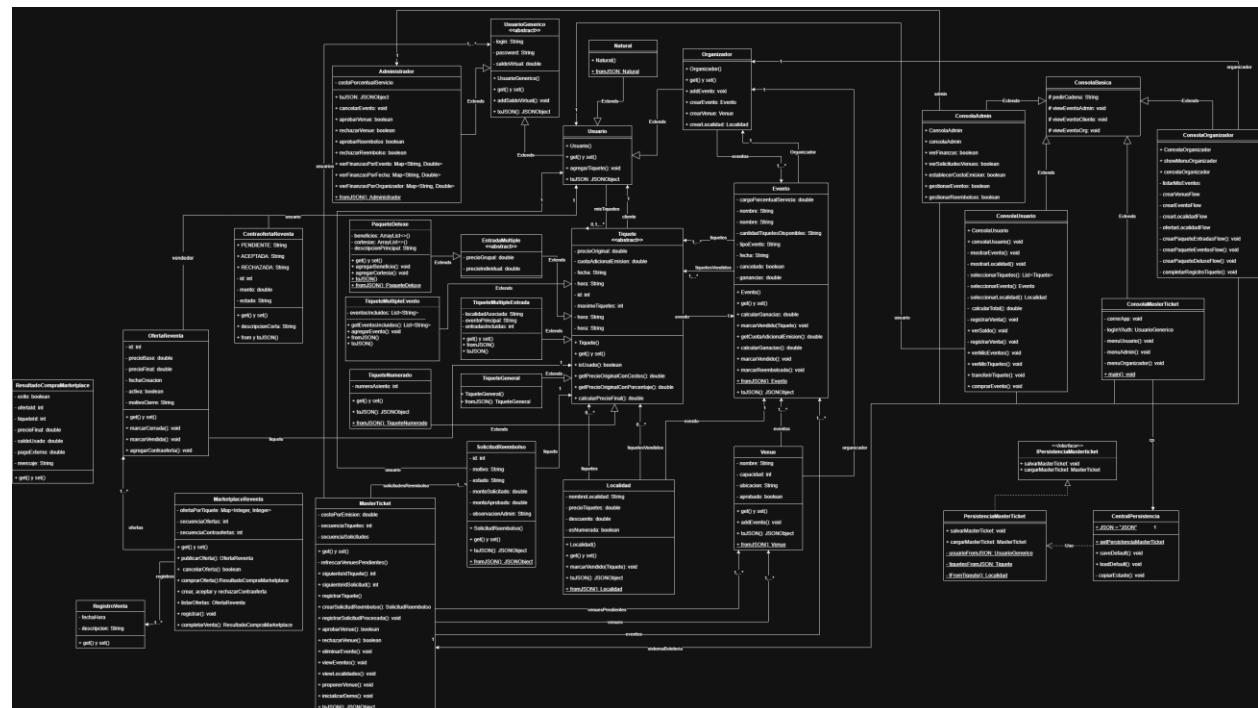
- Usuario, clase base de la cual derivan los roles Organizador y Administrador.
- Evento y Tiquete, encargadas de representar la oferta y compra de entradas.
- Las clases Marketplace, Oferta y LogRegistro amplían la funcionalidad para permitir reventa de boletas entre usuarios, contraofertas y registro de transacciones.

En la persistencia, la clase PersistenciaJSON se mantiene como componente central, responsable de almacenar y recuperar la información del sistema en archivos locales (.json).

## *7.2 Relaciones Principales*

El diseño del sistema establece un conjunto de relaciones que reflejan tanto la estructura lógica del modelo como el flujo operativo de la plataforma MasterTicket.

- **Herencia:** Las clases Organizador y Administrador heredan de Usuario, compartiendo atributos como nombre, correo y saldoVirtual.
- **Relación Usuario-Tiquete:** Un usuario puede poseer múltiples tiquetes. Cada tiquete pertenece a un único usuario, lo que permite rastrear sus compras, transferencias o ventas dentro del sistema.
- **Relación Organizador-Evento:** Cada organizador puede crear y administrar varios eventos, mientras que cada evento cuenta con un único organizador responsable.
- **Relación Evento-Tiquete:** Un evento contiene una lista de tiquetes asociados. Cada tiquete referencia directamente el evento al que pertenece.
- **Relación Administrador-Sistema:** El administrador mantiene una relación jerárquica con las demás clases. Supervisa eventos, aprueba o rechaza reembolsos, puede eliminar ofertas y consulta el registro general de operaciones.
- **Relación Usuario-Marketplace:** Los usuarios interactúan con el Marketplace para publicar, modificar o eliminar ofertas de reventa. Esta relación es mediada por la clase Oferta, que representa cada publicación individual.
- **Relación Marketplace-LogRegistro:** Cada acción realizada en el Marketplace genera automáticamente una entrada en el LogRegistro.
- **Dependencia Administrador-LogRegistro:** El administrador puede consultar, revisar y gestionar el log de registros.
- **Dependencia PersistenciaJSON-Clases principales:** La clase PersistenciaJSON actúa como un módulo de soporte que mantiene una relación de dependencia con todas las clases del dominio, ya que es la encargada de guardar los datos en archivos JSON.



El UML anterior muestra el funcionamiento total de la solución, incluyendo las clases más importantes y sus relaciones (1), atributos y métodos empleados (2), el modelo de la consola/interacción con el usuario (3) y el manejo de la persistencia (4).

## 9. Diagramas de Secuencia

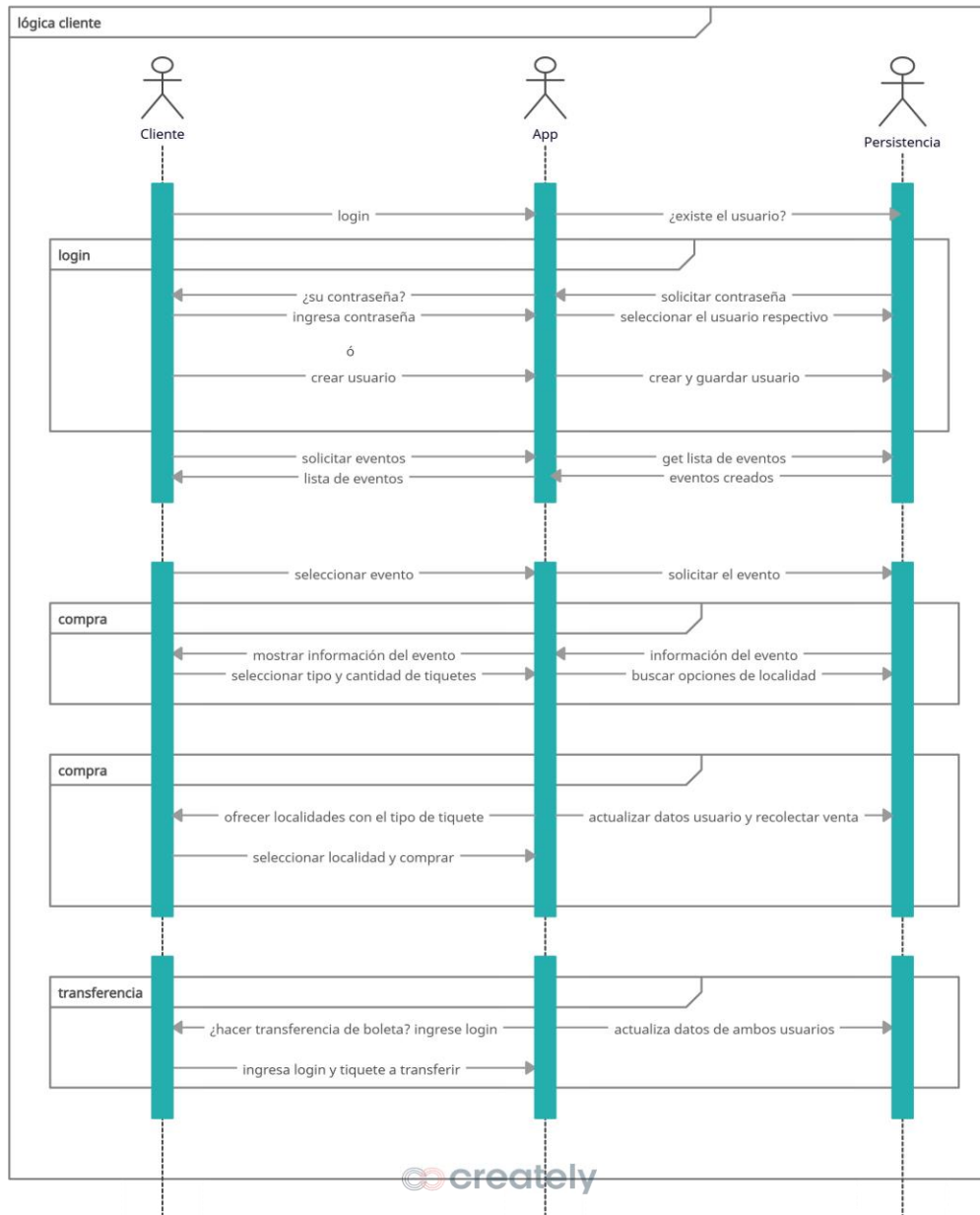


Figura 1: Diagrama de Secuencia del Cliente

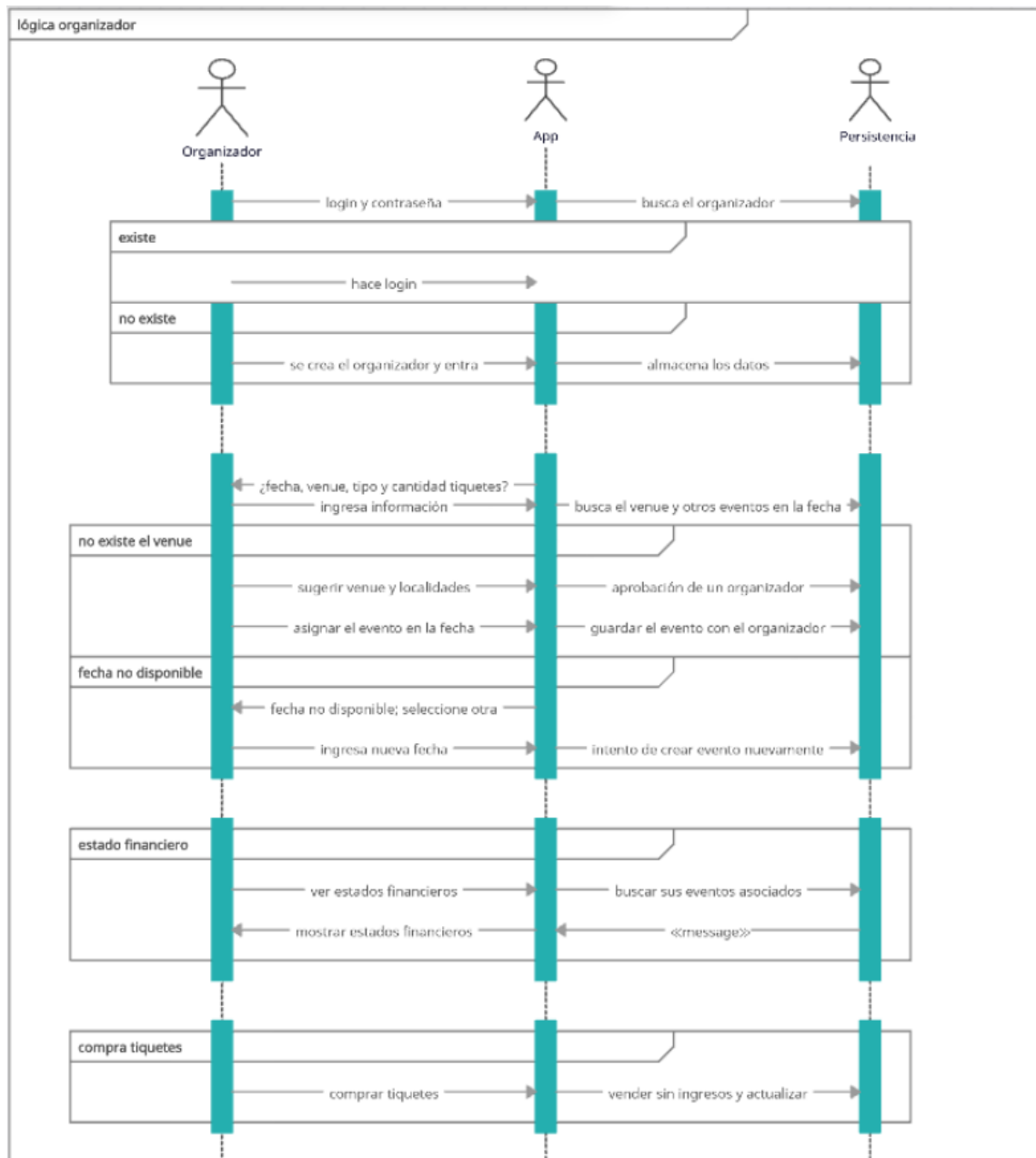


Figura 2: Diagrama de Secuencia del Organizador

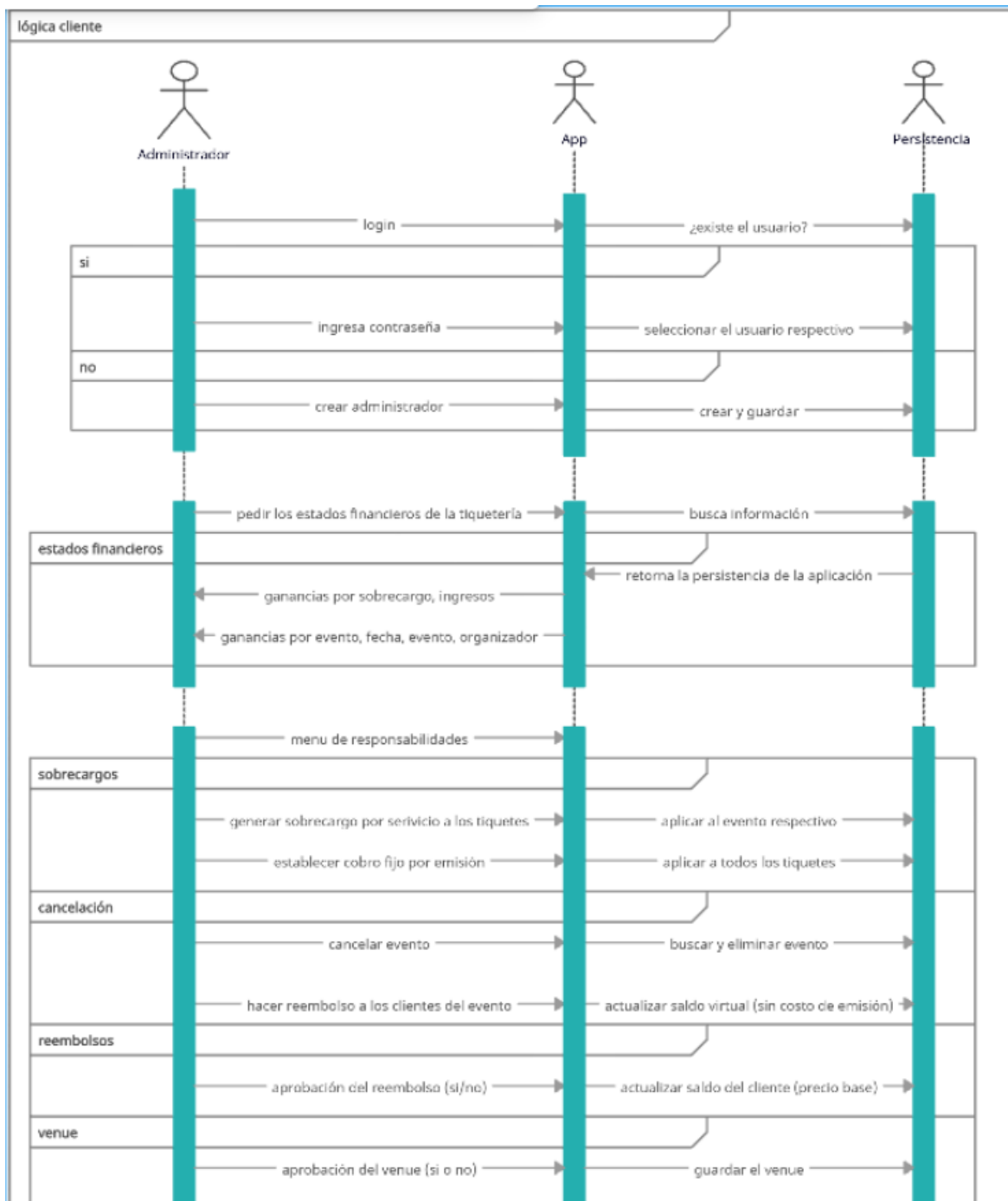


Figura 3: Diagrama de Secuencia del Adinistrador

## 10. Información sobre la persistencia.

### 10.1. Estructura de carpetas

La aplicación utiliza un esquema de persistencia local basado en archivos JSON. Toda la información se almacena en una carpeta denominada `/datos`, ubicada al mismo nivel del código fuente, con el fin de mantener una separación clara entre la lógica del programa y los datos persistentes.

Dentro de esta carpeta se generan los siguientes archivos:

- `usuarios.json`, `eventos.json`, `tiquetes.json` → almacenan la información base del sistema.
- `marketplace.json` → registra las ofertas, contraofertas y transacciones de la reventa de boletas.
- `log.json` → conserva el historial cronológico de todas las operaciones realizadas en el Marketplace.

Si la carpeta o alguno de los archivos no existen al momento de ejecutar el programa, la clase `PersistenciaJSON` los crea automáticamente mediante sus métodos de inicialización.

### 10.2. Formato de los archivos

El sistema utiliza el formato JSON para almacenar toda la información del dominio. La serialización se realiza con la biblioteca `org.json`, usando las clases `JSONObject` y `JSONArray`.

El formato permite representar jerárquicamente todos los objetos del sistema, manteniendo los datos legibles y fáciles de recuperar. Los archivos se guardan con sangría (pretty print) para facilitar su lectura y el control de versiones en Git.

Los principales tipos de datos almacenados son:

- **Strings:** nombres, correos, identificadores y fechas (ISO 8601, ej. "2025-11-06T10:30").
- **Numbers:** valores numéricos para precios, saldos y contadores.
- **Booleans:** campos de estado (cancelado, vendido, activo...).

### 10.3 Esquema de datos

Cada archivo JSON sigue una estructura donde las entidades se vinculan por sus identificadores únicos. Por ejemplo:

- En `marketplace.json`, cada objeto `Oferta` referencia el `idTiquete` y el `idUsuario` del vendedor, permitiendo reconstruir la relación entre usuarios y tiquetes revendidos.



- En log.json, cada registro incluye los campos fechaHora, tipoEvento, usuario y detalle, lo que asegura trazabilidad completa de las operaciones.

#### *10.4 Ejemplo de persistencia*

Al cerrar la aplicación, el sistema guarda automáticamente:

- La lista actualizada de usuarios, eventos y tiquetes.
- Las ofertas activas del Marketplace, junto con su estado y precio.
- El historial completo del LogRegistro.

## **11. Alternativas de solución consideradas**

### *11.1 Opciones Evaluadas*

Durante el diseño se analizaron varias alternativas para la persistencia y la estructura del sistema. La principal fue implementar una base de datos relacional para almacenar usuarios, eventos y tiquetes, opción que ofrecía mayor robustez y escalabilidad. Sin embargo, se descartó por requerir configuración de servidores externos, lo que excedía el alcance local del proyecto. También se consideró usar archivos binarios o serialización de objetos, pero se prefirió mantener el formato JSON por su simplicidad, legibilidad y compatibilidad con el modelo actual.

### *11.2 Razones para no implementarlas*

Las alternativas se descartaron por criterios de simplicidad y alcance. El uso de bases de datos relacionales implicaba dependencias externas y mayor complejidad de configuración, mientras que los archivos binarios reducían la legibilidad y trazabilidad de los datos. Por ello, se mantuvo el formato JSON, que ofrece un equilibrio adecuado entre claridad, facilidad de depuración y compatibilidad con la arquitectura del sistema.

## **12. Preocupaciones y riesgos de la solución**

### *12.1 Riesgos Técnicos*

- Corrupción de archivos JSON: Si el archivo de persistencia se modifica manualmente o no se cierra correctamente durante la escritura, el sistema podría no poder cargar los datos.
- Escalabilidad limitada: Al no usar una base de datos, el sistema podría ralentizarse si el volumen de datos crece significativamente.

### *12.2 Riesgos Funcionales*

- Errores en transferencias o reembolsos: Si no se validan correctamente las reglas de negocio, podría haber pérdidas de coherencia en los saldos o en la propiedad de tiquetes.

- Dependencia en la precisión del usuario: El sistema asume que los datos ingresados son válidos, por lo que un error humano puede generar resultados inconsistentes.

### *12.3 Riesgos de Mantenimiento*

- Dependencia de archivos locales: El manejo manual de archivos puede complicar las pruebas y la depuración en entornos distribuidos.
- Falta de automatización en pruebas: La validación manual desde consola dificulta la integración continua y el aseguramiento de calidad.

## **13. Conclusiones**

Durante esta segunda parte del proyecto logramos que MasterTicket pasara de ser solo una plataforma de venta de tiquetes a convertirse en un sistema mucho más completo, donde los usuarios también pueden revender y negociar sus boletas dentro del mismo entorno.

Agregar el Marketplace y el LogRegistro fue un reto interesante, porque implicó pensar no solo en nuevas clases, sino en cómo mantener la coherencia con el diseño que ya teníamos. El resultado fue un modelo más vivo, pues los usuarios ahora pueden interactuar entre sí, crear ofertas, hacer contraofertas y ver cómo sus acciones quedan registradas. Todo esto permitió reforzar la idea de trazabilidad, sin perder la claridad del código ni romper la estructura original.

También fue valioso extender la clase PersistenciaJSON para guardar las nuevas partes del sistema. Aunque seguimos trabajando con archivos locales, el hecho de poder mantener la información del Marketplace y del Log entre ejecuciones demuestra que la aplicación puede crecer sin depender todavía de una base de datos externa.

En general, esta entrega nos ayudó a entender mejor cómo evolucionar un proyecto sin empezar desde cero: cómo agregar nuevas funcionalidades, conservar la lógica base y garantizar que todo siga funcionando. El sistema ahora es más completo, modular y realista, y deja abierta la posibilidad de seguir avanzando en cosas como una interfaz más intuitiva o incluso la integración con un sistema de pagos real.