Índice

Manual Técnico

Manual Técnico – Sistema de Gestión de Turismo

Introducción técnica

El manual técnico explica la estructura interna del sistema, los componentes de software utilizados, la forma en que el código se organiza y se comunica con Oracle XE, así como los lineamientos para mantenimiento, pruebas y mejoras futuras. El objetivo es facilitar a desarrolladores y responsables técnicos la comprensión del diseño y la ejecución del proyecto.

Arquitectura general y flujo

La aplicación sigue una estructura modular sencilla. El punto de entrada (main.py) presenta el menú y orquesta las operaciones. Un módulo de gestión de base de datos (db_manager.py) concentra la lógica de conexión y ejecución de sentencias SQL. Un archivo de configuración (oracledb_config.py) mantiene las credenciales y parámetros de DSN. Este esquema se asemeja a un patrón MVC simplificado: Modelo (BD/SQL), Controlador (db_manager.py), Vista (consola).

Flujo general: el usuario elige una opción; main.py valida y delega; db_manager.py arma y ejecuta la consulta o transacción contra Oracle; los resultados o mensajes vuelven a main.py para mostrarse en consola.

Componentes y responsabilidades

main.py: inicializa la conexión, presenta el menú principal, captura entradas del usuario, maneja validaciones básicas de opciones y delega las operaciones al gestor de BD.

db_manager.py: implementa funciones CRUD, llamadas centralizadas a oracledb.connect, ejecución de consultas con cursor, manejo de commits/rollbacks y cierre seguro de recursos.

oracledb_config.py: variables de conexión (usuario, contraseña, host, puerto, servicio) y función de obtención de conexión; puede ampliarse para leer variables de entorno y mejorar portabilidad.

Conexión a Oracle y manejo de recursos

Se utiliza la librería oracledb, que permite conexiones con DSN del tipo host:puerto/servicio. La apertura y cierre de conexiones debe ser explícita; cada operación crítica debe envolver la ejecución en bloques try/except para capturar errores ORA-*. Es recomendable usar context managers para cursores y asegurar el cierre en finally.

Scripts SQL del proyecto

1) creacion de tablas paso 1.sql: contiene DDL para crear tablas principales del dominio de turismo. Debe respetar claves primarias, foráneas y restricciones de integridad. Sugiere 3FN para evitar redundancias.

2) triggers paso 2.sql: define disparadores que auditan inserciones, actualizaciones y eliminaciones lógicas. Los triggers deben ser idempotentes al recrearse, o necesitar limpieza previa.

3) insercion de datos paso 3.sql: carga datos iniciales o de prueba para facilitar la verificación del sistema.

4) eliminado logico paso 4.sql: establece campos o tablas auxiliares para marcar registros como inactivos en lugar de borrarlos, preservando trazabilidad y permitiendo potencial restauración.

Modelo de datos y normalización (visión general)

Las tablas deben describir entidades del dominio (por ejemplo, turistas, ubicaciones, reservas o equivalentes definidos en el script real) con claves primarias numéricas o naturales según el diseño, y referencias foráneas para relaciones. Se recomienda mantener 3FN como base, además de índices en campos de búsqueda frecuente y restricciones NOT NULL en atributos obligatorios.

Auditoría y triggers

Objetivo y alcance

La auditoría se implementa **dentro de la base de datos**, de modo que el registro de cambios **no depende del cliente** (Python). Esto asegura trazabilidad incluso si en el futuro hay otras aplicaciones o procesos que escriben en las mismas tablas. La estrategia estándar es: por cada operación relevante (INSERT, UPDATE, “DELETE” lógico), un **trigger** inserta un registro en una **tabla de bitácora**.

Diseño recomendado de la bitácora

- **Tabla**: AUDITORIA_* (una general o una por entidad crítica).
- **Campos mínimos**:
    - id_evento (PK)
    - tabla_afectada (VARCHAR2)
    - operacion (INSERT/UPDATE/DELETE_LOGICO)
    - id_registro (identificador de la fila afectada)
    - usuario_bd (quién ejecuta: SYS_CONTEXT('USERENV','SESSION_USER'))
    - fecha_hora (SYSTIMESTAMP)
    - valores_antes (CLOB) — opcional en INSERT
    - valores_despues (CLOB) — opcional en DELETE lógico
    - origen (opcional: cliente, IP, módulo)
- **Buenas prácticas**:
    - Usar **SYSTIMESTAMP** (con zona horaria) en lugar de SYSDATE.
    - Capturar el usuario con **SYS_CONTEXT** para evitar depender de variables del cliente.
    - Si valores_antes/valores_despues son pesados, serializar a JSON o registrar **solo columnas clave**.

Tipos de triggers y consideraciones

- **Nivel**: preferir **row-level** (FOR EACH ROW) para obtener :OLD y :NEW.
- **Momento**:
    - AFTER INSERT: registra creación.
    - AFTER UPDATE: registra modificación (idealmente solo si cambió algo significativo).
    - AFTER UPDATE OF estado: útil para **eliminado lógico** (p. ej., cambio activo = 'N').
- **Rendimiento**:
    - Mantener el trigger **ligero**: solo construye el registro de auditoría y hace INSERT.
    - Indexar tabla_afectada e id_registro en la bitácora si se hacen consultas frecuentes.
- **Confiabilidad**:
    - Evitar DML masivo dentro de triggers.
    - Manejar errores dentro del trigger con cuidado: un fallo no debe "romper" la transacción, salvo que la política lo exija.

Reporte y revisión

- Consultar la bitácora por **rango de fechas**, tabla_afectada, id_registro o operacion.
- Opcional: vistas de apoyo VW_AUDITORIA_RESUMEN con columnas calculadas (día, usuario, conteos por operación).
- Si el volumen crece, definir **retención** (por ejemplo, rotar/archivar registros antiguos).

Compatibilidad multiplataforma y despliegue

Código único, entorno diferente

- El código Python es **el mismo** en macOS, Linux y Windows.

- Lo que **cambia** es: activación del **entorno virtual**, instalación/arranque del **motor de contenedores** y detalles de **shell** (zsh/bash/PowerShell).

Ruta de despliegue local recomendada

1. **Docker en marcha** (Docker Desktop en macOS/Windows; docker/systemd en Linux).
2. **Levantar Oracle XE** en 1521 con servicio XE:
    - Opcional: asignar **volúmenes** para persistir datos (-v oracle-data:/opt/oracle/oradata) y evitar perderlos al recrear el contenedor.
3. **Crear usuario de app** (p. ej., turismo) con **privilegios mínimos** y **cuota** en tablespace.
4. **Cargar scripts SQL** (creación → triggers → inserción → eliminado lógico).
5. **Configurar DSN** en Python (localhost:1521/XE) y **variables de entorno** para credenciales.
6. **Ejecutar main.py** con el **venv** activo (python3 en macOS/Linux, python en Windows).

Verificaciones clave por SO

- **macOS**: python3 --version, which python3, docker ps.
- **Linux**: pertenecer al grupo docker (usermod -aG docker $USER y re-login), systemctl status docker.
- **Windows**: Docker Desktop con **WSL2** habilitado; PowerShell con permisos para ejecutar el **Activate.ps1** del venv.

Diagnóstico rápido

- docker logs oracle-xe si no puedes conectar.
- tnsping XE (si lo tienes) para validar listener/servicio.

- Revisar firewall/antivirus si el puerto 1521 está bloqueado.

Seguridad, credenciales y respaldo

Principio de mínimo privilegio

- Asignar **solo** lo necesario al usuario de aplicación: CREATE SESSION + privilegios de objetos requeridos (p. ej., CREATE TABLE/VIEW/SEQUENCE/TRIGGER/PROCEDURE si el diseño lo necesita).
- Evitar otorgar **DBA** o roles globales en entornos compartidos.

Gestión de secretos

- No commitear contraseñas al repo.
- Preferir **variables de entorno** (o un .env no versionado).
- Rotar contraseñas periódicamente y al finalizar el curso/proyecto.

Red y superficie de ataque

- Limitar el acceso de red a la BD (puerto 1521) al host local cuando sea posible.
- Deshabilitar cuentas por defecto si no se usan.

Respaldos (backup) y restauración

- En Docker, usar **volúmenes** para persistir datos y poder **snapshotear**.
- Probar un **ciclo de restauración** (no basta con “tener backup”; hay que validar que vuelve a levantar).
- Mantener al menos un **backup frío** antes de cambios de esquema o de pruebas “destructivas”.

Auditoría y retención

- Definir una **política de retención** para tablas de auditoría (por ejemplo, 3–6 meses) o movidas periódicas a una tabla histórica.

Mantenimiento y mejoras futuras

Código y organización

- Separar en **paquetes** (por ejemplo, app/cli.py, app/db.py, app/config.py…) con **docstrings** y **type hints**.
- Añadir formateo y linting (black, flake8) para uniformidad.

Configuración y despliegue

- Centralizar configuración en **variables de entorno** y un config.py que las lea con defaults seguros.
- Añadir requirements.txt o pyproject.toml para reproducibilidad.
- Scripts de **migraciones** simples (tabla schema_version + archivos DDL con número incremental).

Observabilidad

- Logging con niveles (INFO/WARN/ERROR) y **mensajes claros** ante ORA-*.
- Contadores simples (cuántas operaciones CRUD por sesión) para diagnóstico.

Rendimiento y datos

- Índices en columnas de **búsqueda frecuente**.
- **Paginación** en listados si la tabla crece.

- Revisar estadísticas y planes de ejecución si aparecen consultas lentas.

Experiencia de usuario

- Opcional: interfaz de **consola enriquecida** (p. ej., rich) o una futura GUI/Web.
- Mensajes de error/confirmación **consistentes** y específicos.

Cierre del manual técnico

El sistema se apoya en una arquitectura **simple y extensible** (Python + Oracle XE), con responsabilidades claras entre módulos y la base de datos. La auditoría basada en triggers otorga **trazabilidad** sin depender del cliente; la compatibilidad multiplataforma permite ejecutar el mismo código en macOS, Linux y Windows cambiando solo el **entorno**.

Con un conjunto mínimo de **pruebas de humo e integración**, políticas de **seguridad y respaldo**, y tareas periódicas de **mantenimiento**, el proyecto queda listo para crecimiento futuro (enriquecer la interfaz, mejorar la configuración y escalar el esquema con migraciones controladas).