BITÁCORA COMPLETA DEL PROYECTO – SISTEMA DE RESERVA DE SALAS

Felipe Paladino, Santiago Blanco y Facundo Martínez

Materia: Bases de Datos I – Universidad Católica del Uruguay

Período: Segundo Semestre 2025

1. Inicio del Proyecto

Comenzamos analizando la letra del obligatorio y definimos las tres areas más importantes:

1) **Diseño de la base de datos.**

2) **Backend** en Python (obligatorio).

3) **Frontend** libre (optamos por React).

Detectamos la primera tristeza con el requisito de usar Python para el backend (considerándolo lento y poco expresivo comparado con C++), pero decidimos mitigar esto utilizando **tipado estático** (*type hints*) para mantener la sanidad mental del equipo.

1.1. Planificación de Estructura Inicial

Se planificó la siguiente estructura de directorios:

- `/api`
- `/services`
- `/models`
- `/enums`
- `/db`
- otras cosas

Se inició el trabajo en el script `creacionTablas.sql`, que sería modificado consistentemente a lo largo del proyecto.

2. Diseño Inicial de la Base de Datos

Se crearon las tablas básicas, reflejando directamente los requisitos del obligatorio:

- `login`, `participante`, `sala`, `reserva`, `reserva_participante`, `turno`, `facultad`, `edificio`, `programa_académico`, `participante_programa_académico`.

En esta etapa se generó el archivo `inserciones.sql` con datos maestros para testeo, que se mantuvo con ajustes menores hasta el final.

3. Primera Implementación del Backend (Flask)

Santiago y Felipe se enfocaron en el desarrollo del backend.

3.1. Decisiones Técnicas

- **Framework:** Se eligió *Flask* por simplicidad y experiencia previa (se consideró *fastAPI*, pero se descartó).
- **Estructura:** Modular, utilizando *blueprints* por dominio (`auth`, `participante`, `sala`, `reserva`, etc.).
- **Acceso a Datos:** Lógica centralizada en la carpeta *services*.
- **Modelos:** Entidades de la BD representadas en la carpeta *models* con *type hints*.
- **Conexión:** Uso directo de *mysql-connector-python* (sin ORM).

Se definió la regla estricta de usar **consultas parametrizadas** desde el inicio para prevenir *SQL Injection*.

3.2. Herramientas

- **Postman** se usó intensivamente para probar *endpoints*, verificar *payloads*, y asegurar que el contrato JSON fuera estable.
- Facundo comenzó a desarrollar las consultas avanzadas de reportes.

---

4. Sistema de Autenticación y Sesiones

El diseño inicial de `login(correo, contrasena)` fue *drásticamente* revisado para incluir seguridad y gestión de roles.

4.1. Rediseño de la Base de Datos

- Se agregó el campo `isAdmin` a la tabla `login`.
- Se creó la nueva tabla `sesion_login` para soportar:
  - UUID único por sesión
  - Fechas de creación y expiración
  - Flag `revocado` (para **logout real**).

4.2. Implementación de JWT

Se adoptó *PyJWT*:

- Al iniciar sesión, se genera un token (con correo, `is_admin` y UUID de sesión) y se registra la sesión en `sesion_login`.
- Todos los *endpoints* protegidos usan el decorador `@required_token`, que verifica la validez, no expiración y no revocación de la sesión.
- El decorador `@admin_required` se implementó para proteger las operaciones administrativas.

---

5. Consultas Avanzadas y Reportes

Se desarrollaron las consultas avanzadas solicitadas, que fueron integradas al módulo `/api/v1/reportes` para consumo del frontend. Las consultas incluyen métricas como salas más reservadas, porcentaje de ocupación, cantidad de reservas por carrera/facultad, y sanciones por rol (profesor/alumno).

6. Evolución de la Funcionalidad Extra

6.1. Primer Intento Descartado

Inicialmente se propuso un sistema simple de alertas "técnicas"(humo, fuego, inundación). Tras crear prototipos de tablas y código, se sugirió que la funcionalidad extra debía ser más útil y consistente con el uso real del sistema. Esta aproximación inicial fue **descartada**.

6.2. Rediseño: Sistema de Incidencias y Alertas (Diseño Final)

Se implementó una funcionalidad extra robusta: un sistema de incidencias y alertas ligado al estado operativo de las salas y a las reservas futuras.

- **Objetivo:** Permitir a usuarios reportar problemas reales e impactar en reservas futuras.
- **Nuevas Tablas:** `incidencia_sala` y `alerta_reserva`.
- **Regla de Propagación:** Si una incidencia es de gravedad alta, se crean alertas para reservas futuras activas de esa sala.

6.2.1. Estados Operativos Híbridos

Se definieron dos conceptos de estado de sala, donde el *Estado Manual* tiene prioridad sobre el *Estado Calculado*:

Estado Calculado	Estado Manual (Admin)
Derivado automáticamente de *incidencias abiertas*: • **Grave abierta** → `fuera_de_servicio` • **Media/Baja abierta** → `con_inconvenientes` • **Ninguna abierta** → `operativa`	Campo almacenado en `sala`: Puede ser modificado por un administrador para forzar el estado (ej. mantenimiento programado o reapertura rápida).

La lógica híbrida se implementó para integrar la información automática (incidencias) con la decisión administrativa explícita (estado manual).

7. Implementación del Sistema de Incidencias y Alertas

Se implementó la lógica completa, incluyendo:

- **Validación en Creación:** Solo se permiten crear incidencias si el usuario tiene una reserva activa en la sala afectada.
- **Endpoints de Incidencias:** Rutas para creación (`POST /`), listado personal (`GET /mias`), resolución por el usuario (`PATCH /{id}/resolver`) y eliminación por el administrador (`DELETE /{id}`).
- **Endpoints de Alertas:** `GET /mias` para consultar alertas que afectan las reservas futuras del usuario.

Se definieron los permisos de roles estrictos: el **Participante** no puede eliminar incidencias y el **Administrador** puede ver todas las incidencias y modificar el estado manual de las salas.

8. Desarrollo del Frontend (React)

Facundo se encargó principalmente del frontend.

8.1. Estructura y Tecnologías

- **Tecnología:** React con React Router.
- **Estado:** `AuthContext` para gestionar el estado del usuario (`user`, `token`, `is_admin`) y almacenarlo en `localStorage`.
- **Comunicación:** Módulo `api.js` que centraliza todas las llamadas `fetch` e inyecta automáticamente el token de sesión.

8.2. Diferenciación de Interfaz

La interfaz utiliza el rol (`user.is_admin`) para mostrar u ocultar rutas y elementos (como botones de gestión), creando dos experiencias de usuario distintas.

9. Debugging

Se realizaron pruebas de punta a punta en todos los flujos (autenticación, reservas, incidencias) y se verificó la seguridad (protección de roles y uso de dos usuarios de BD).

- **Problemas Encontrados:** Desajustes de rutas, errores 401/403 de token, problemas de CORS, cambios en credenciales de Docker y ajustes finales en consultas SQL.
- **Resolución:** Ensayo y error, uso constante de Postman, logs y revisiones conjuntas de código.

10. Última Semana: Integración Final y Cierre

Esta fue la fase más intensa, centrada en la integración y la documentación.

- Se configuró docker compose para levantar el proyecto completo junto con una BD MySQL con scripts iniciales de creación de tablas e inserción de datos.
- Se terminó de implementar el frontend y se revisó la protección por roles del backend.
- Se verificó la coherencia entre `creacionTablas.sql`, `inserciones.sql`, el modelo real y la descripción del informe.
- Completamos informe, bitácora, instrucciones de uso.
- Encontramos errores en la BD: Había un ON UPDATE RESTRICT que no permitía eliminar un usuario, eso tuvimos que cambiarlo.

A pesar de los desafíos, se logró un backend estable y seguro, un frontend funcional y una funcionalidad extra que marchó joya.

10.1. Tarea Pendiente (no pudimos)

Se optó por no implementar filtros opcionales en el módulo de reportes, ya que las consultas requeridas eran muy específicas y agregar filtros habría implicado reescribir consultas complejas, lo que se consideró un riesgo dado el tiempo límite. Esta funcionalidad quedó trunca y desordenada, por lo cual el equipo se disculpa con los profesores a cargo.

Fin de la bitácora.