OBLIGATORIO: SISTEMA DE RESERVA DE SALAS

Santiago Blanco, Facundo Martinez, y Felipe Paladino

Universidad Católica del Uruguay

Bases de Datos I

Docentes:

Christian Dinardi, Leandro Barral

23/11/2025

Índice

Introducción

El presente informe tiene como objetivo documentar y justificar el proceso de diseño y desarrollo del proyecto que comprende un sistema de reservas de salas de estudio como parte del obligatorio del curso de Bases de Datos I.

Decisiones de diseño

Estructura de la base de datos

Aquí se describe la estructura general de la base de datos, tablas utilizadas, relaciones principales y motivaciones del modelo final adoptado.

Evolución de diseño de la base de datos

A continuación se explican las modificaciones que se produjeron durante el desarrollo del proyecto respecto al diseño inicial planteado en el documento del obligatorio, justificando las decisiones.

Sistema de usuarios y sesión

Originalmente:

- Únicamente existía la tabla login(correo, contrasena) asociada al participante por su correo.
- No existía manejo de sesiones ni tokens.

Sin embargo, la base de datos sufrió un rediseño importante:

- login ahora incluye un campo isAdmin para distinguir roles.
- Se creó la nueva tabla sesion_login con:
    - UUID como identificador de sesión.
    - Timestamps de creación y expiración.
    - Campo booleano revocado.

Este cambio habilita el control de roles administradores y permite implementar autenticación mediante tokens revocables y expirables, haciendo posible la implementación de un logout real y tracking de sesiones activas.

Modelo de programas académicos y roles

En el diseño original:

- Tabla participante_programa_academico con roles deficientes: alumno o docente.

  En el nuevo diseño:

- Rol ampliado y normalizado utilizando ENUM:
  - estudiante_grado
  - estudiante_posgrado
  - docente
- Se agrega restricción:

```
UNIQUE(ci_participante, nombre_programa, rol)
```

- La tabla hace referencia al participante y programa mediante claves foráneas con políticas claras:
  - ON UPDATE CASCADE garantiza que cualquier actualización en los identificadores primarios se vea reflejada automáticamente.
  - ON DELETE RESTRICT evita que se borren participantes o programas que estén asociados a registros, evitando perder información relevante y manteniendo **integridad referencial**. (Silberschatz, 2006)
  - Permite distinguir claramente estudiantes de grado y posgrado y hace más simple la consulta.

Nuevas reglas de integridad para turnos

Original

- turno(id_turno, hora_inicio, hora_fin) sin restricciones adicionales.

Nuevo diseño

Se incorpora un CHECK estrictamente definido:

```
CHECK (TIMESTAMPDIFF(MINUTE, hora_inicio, hora_fin) = 60)
```

- Garantiza que todos los turnos duran exactamente 1 hora (Divididos en bloques) y evita horarios inconsistentes.

Nuevos estados para salas y VIEW del estado

Original

- sala no tenía campo estado.
- No existía mecanismo para reportar inconvenientes o calcular disponibilidad real.

Nuevo diseño

- Se agrega estado en sala:
    - operativa, con_inconvenientes, fuera_de_servicio.
- Se crea incidencia_sala para reportar problemas.
- Se agrega alerta_reserva para generar alertas a los usuarios.
- Se crea la vista vista_estado_sala, que:
    - Detecta automáticamente salas fuera de servicio o con inconvenientes según incidencias abiertas. (Ej: Rata en la pared del Mullin, proceda con precaución)
- Permite soporte a un sistema real de alertas e incidencias.
- Aporta información en tiempo real para deshabilitar salas dañadas.

  →Esto se desarrolla más adelante

Nueva funcionalidad: Sistema de alertas

La funcionalidad permite que los participantes que reservan una sala puedan reportar incidencias asociadas a esa sala (por ejemplo, un proyector roto, problemas de ratas, suciedad, etc.). Cada incidencia se registra con un tipo y una gravedad, y a partir de esa información el sistema:

Registra la incidencia ligada a:

- la sala afectada,
- la reserva desde la cual se reporta (si corresponde),
- y el participante que reporta el problema.

Evalúa el impacto de la incidencia sobre reservas futuras:

- Si la incidencia es de gravedad alta y existen reservas futuras activas para esa misma sala (a partir de la fecha actual), el sistema genera alertas asociadas a esas reservas.

Estas alertas cumplen dos funciones:

- Informar al usuario que la sala presenta un inconveniente.
- Impedir o permitir la reserva, según el tipo de alerta.

Cálculo del estado operativo de la sala

El sistema distingue entre:

- Estado calculado: se deriva automáticamente de las incidencias registradas en incidencia_sala. Si existe al menos una incidencia no resuelta de gravedad alta para una sala (ej: fuga de gas), el estado calculado de esa sala es fuera_de_servicio. Si existen incidencias abiertas de gravedad media o baja, el estado calculado es con_inconvenientes. En ausencia de incidencias abiertas, el estado calculado es operativa.
- Estado manual: se almacena en la tabla sala y puede ser modificado por un usuario administrador. Este estado representa la decisión de la administración sobre la disponibilidad de la sala.

El estado manual tiene prioridad sobre el estado calculado: es decir, un administrador puede forzar que una sala figure como operativa o fuera de servicio, independientemente de lo que sugiera el estado calculado.

Impacto sobre el flujo de reservas

Dependiendo del estado (calculado y/o manual), se aplica la siguiente lógica:

- Si la sala está fuera_de_servicio (por incidencia grave o por decisión manual):
- No se permiten nuevas reservas sobre esa sala mientras dure esa condición.

Si la sala está con_inconvenientes:

- Se permiten nuevas reservas, pero al usuario se le muestra una alerta indicando que la sala presenta problemas. El usuario debe aceptar explícitamente que está al tanto del inconveniente para confirmar la reserva.

Si la sala está operativa:

- El flujo de reservas es normal, sin advertencias adicionales.

Roles y permisos

Usuario (rol participante):

- Puede crear incidencias sobre salas donde efectivamente tiene una reserva activa (según la validación realizada entre reservas y participante).
- Puede marcar como "resuelta" su propia incidencia (es decir, darla por finalizada), lo cual impacta en el cálculo del estado de la sala y en las alertas asociadas.
- Puede consultar el estado de una sala y las alertas que existan sobre sus reservas.
- No puede eliminar incidencias.

Administrador:

- Puede ver el listado completo de incidencias registradas en el sistema, independientemente de quién las haya reportado o en qué sala.
- Puede crear incidencias (por ejemplo, a partir de inspecciones internas).
- Puede eliminar incidencias. Al eliminar una incidencia, se eliminan también las alertas asociadas.
- Puede modificar el estado manual de las salas, con el fin de corregir situaciones donde el estado calculado no refleja la realidad, anticipar mantenimientos o cierres programados.

De esta forma, la funcionalidad extra integra:

- la experiencia del usuario (que reporta problemas y recibe alertas),
- la operación administrativa (gestión manual del estado de salas),
- y una lógica automática de propagación de información a reservas futuras.

Desarrollo del backend

El backend del sistema fue desarrollado utilizando Flask, que según su sitio web, es un micro–framework ligero de Python que se caracteriza por ser simple y modular. (*Welcome to Flask — Flask Documentation*, s. f.) El equipo de producción lo eligió por contar con experiencia previa y para reducir al mínimo las dependencias.

Descripción de la API

La API del sistema de reservas fue diseñada siguiendo un estilo RESTful, modular y orientado a servicios. Cada conjunto de funcionalidades se organiza en blueprints indepen-

dientes, lo que permite un código más escalable. Todas las rutas de esta magnifica API se encuentran bajo el prefijo general /api/v1/.

Estructura general de la API

- /api/v1/auth → Autenticación, manejo de sesiones, cambio de contraseña.
- /api/v1/participante → Gestión de participantes y sus datos.
- /api/v1/reserva → Creación, consulta, cancelación y administración de reservas.
- /api/v1/sala → Gestión de salas, estados y disponibilidad.
- /api/v1/edificio → Consulta de edificios.
- /api/v1/sancion → Asignación y consulta de sanciones.
- /api/v1/incidencia → Registro de incidencias en salas.
- /api/v1/alerta → Alertas generadas por incidencias.
- /api/v1/reportes → Consultas administrativas y estadísticas.

Autenticación y sesiones

La autenticación se basa en un sistema de tokens unidos a sesiones persistentes en la base de datos.

- POST /login: valida credenciales y crea una sesión en sesion_login.
- POST /logout: revoca únicamente la sesión activa.
- PATCH /cambiar_contrasena: permite actualizar la contraseña del usuario autenticado.

Autenticación con JWT

El sistema implementa autenticación basada en JSON Web Tokens (JWT) usando la librería PyJWT, complementada con un registro persistente de sesiones en la base de datos. Cuando un usuario inicia sesión, el servidor genera un token que incluye el correo, el rol (is_admin), y un UUID de sesión. Antes de devolver el token, la sesión se guarda en la tabla sesion_login, con su fecha de expiración y un flag revocado.

Cada endpoint protegido utiliza el decorador required_token, que:

- Extrae y decodifica el token del header Authorization.
- Verifica que la sesión correspondiente exista, no esté vencida y no esté revocada.

- Expone al handler información del usuari .
- Para rutas administrativas, el decorador admin_required exige además que el campo is_admin sea verdadero.

El acceso a la mayoría de funcionalidades requiere un token válido vía @required_token. Las operaciones administrativas usan @admin_required.

Gestión de participantes

- Listado general.
- Creación, modificación y eliminación (solo administradores).
- Consulta de datos propios mediante /me.

La creación de un participante crea simultáneamente su entrada de acceso en la tabla login.

Gestión de reservas

Los usuarios pueden:

- Crear reservas.
- Consultar sus reservas.
- Cancelar reservas propias.

Los administradores pueden eliminar reservas. El sistema aplica validaciones estrictas:

- No permite doble reserva para el mismo turno y sala.
- No permite que un usuario reserve dos salas simultáneamente.

Salas y estados

Las operaciones incluyen:

- Listar, crear, actualizar y eliminar salas.
- Ajustar manualmente el estado operativo.
- Consultar estados calculados dinámicamente mediante vista_estado_sala.

El estado final combina incidencias registradas y ajustes manuales.

Incidencias y alertas

Los usuarios pueden reportar incidencias en salas. El sistema:

1. Registra la incidencia.
2. Genera alertas en reservas futuras dependiendo de la gravedad.
3. Permite a administradores gestionar el estado de cada incidencia.

El módulo /alerta expone las alertas asociadas al usuario autenticado.

Seguridad general

La API implementa:

- Tokens firmados y sesiones revocables.
- Validación estricta de roles.
- Parametrización de todas las consultas SQL.
- Validación de entrada en cada endpoint.

Conexión con la base de datos

La aplicación utiliza la librería mysql-connector-python para conectarse a MySQL mediante la función get_connection(). Esta función recibe un parámetro booleano (is_admin) que determina si se deben usar credenciales con permisos extendidos o las del usuario restringido app_user. En cada service, se pasa por parámetro true si se trata de una operación que debe realizarse únicamente por un administrador.

Las credenciales se obtienen desde variables de entorno, lo que permite una configuración segura. El uso de dos usuarios distintos implementa el **principio de mínimos privilegios**, que establece que «un usuario solo debe tener acceso a lo que necesita estrictamente para desempeñar sus responsabilidades, y nada más»(Cloudflare, 2025):

- admin: posee permisos completos y se utiliza únicamente en operaciones propias de un administrador del sistema (creación de participantes, eliminación de reservas, gestión de salas, sanciones, etc.).
- app_user: es el usuario por defecto para la mayoría de las operaciones. Tiene permisos limitados y no puede modificar la estructura de la base de datos.

Esta separación protege al sistema, ya que incluso si existiera un error en la lógica de la API, los daños quedarían acotados por los permisos del usuario limitado.

Parametrización de consultas

La parametrización de las consultas es una de las partes fundamentales. Todas las consultas SQL se ejecutan mediante funciones centralizadas (execute_query y execute_returning_id) que aplican de manera estricta consultas parametrizadas. Los valores ingresados por el usuario nunca se concatenan dentro del SQL, lo cual elimina la posibilidad de inyecciones SQL (Probá tranquilo, flaco).

Ejemplo:

```
cursor.execute("SELECT * FROM sala WHERE edificio = %s", (edificio,))
```

El valor es enviado al driver de MySQL por separado del comando SQL, garantizando seguridad y evitando que un pérfido rufián pueda alterar la estructura de la consulta.

Además, todas las funciones ejecutan:

- commit al finalizar correctamente,
- rollback ante errores,
- y un cierre seguro de la conexión y el cursor en un bloque finally.

Esto asegura consistencia en las transacciones y evita que queden conexiones abiertas. En resumen, todo opera maravillosamente y se puede expresar sin ninguna duda que la seguridad de esta aplicación es sólida y proxima a la infaliabilidad.

Uso de los usuarios app_user y admin

La API divide explícitamente las responsabilidades según la sensibilidad de cada operación:

Operaciones con admin

- Crear participantes y sus credenciales.
- Crear, actualizar o eliminar salas.
- Eliminar reservas.
- Registrar sanciones.
- Actualizar estados manuales.
- Ejecutar reportes administrativos.

Estas operaciones requieren permisos elevados y afectan múltiples tablas o reglas del sistema.

Operaciones con app_user

- Crear reservas.
- Listar salas y reservas propias.
- Reportar incidencias.
- Consultar edificios.
- Obtener alertas y sanciones personales.

Este usuario tiene permisos muy restringidos, adecuados para las operaciones del usuario común.

Frontend

Tecnologías utilizadas

Para el desarrollo del frontend, el equipo de desarrollo determinó poner a efecto una aplicación React con la siguiente estructura:

El punto de entrada main.jsx monta el componente App dentro de los proveedores de contexto AuthProvider.

Enrutado y layout

Enrutado implementado con react-router-dom dentro de App.

- La interfaz utiliza un layout principal (AppLayout) que gestiona la barra lateral, el menú y la lógica de logout.

Estado y autenticación

- La autenticación se centraliza en el contexto AuthProvider.
- El hook useAuth gestiona el estado del usuario (user) y el token, almacenados en localStorage.
- Al hacer login, la aplicación realiza un fetch de los datos del usuario en el endpoint /participante/me y lo guarda como objeto JSON en una variable user en el contexto de

AuthContext. Esto permite que todos los componentes puedan acceder a los datos del usuario logueado (correo, is_admin, etc.) y evitar realizar llamadas adicionales a la API.

Capa de comunicación con la API

- Todas las solicitudes al backend pasan por api.js. Este archivo define funciones como getSalas, getReservas, createReserva, cancelReserva y deleteReserva, que encapsulan las llamadas fetch y la interacción con la API en general, encargandose también de integrar el token de sesión automáticamente en todos los request.

Cada pantalla se organiza dentro de la carpeta pages (Reservas, Salas, Reportes, Sanciones, Usuarios, Perfil, Login, entre otras).

Diferenciación de interfaz para admin y usuario común

- El login guarda el token en localStorage y actualiza el contexto global.
- Las rutas y elementos de UI se muestran u ocultan según el estado del usuario y su rol (user.is_admin).
- Esto permite controles como botones exclusivos de administrador y restricciones de acceso a páginas protegidas.

Validación de campos

- Validaciones se realizan principalmente en el propio componente antes de enviar una request: comprobaciones simples con JavaScript (campos requeridos, formato de fecha, conversión a Number, coherencia entre campos, p. ej. turnoExtra distinto del turno principal).
- Validación en HTML valida también (atributos como required, min, pattern cuando aplica).
- El backend también valida y devuelve errores: la capa de servicios (src/services/api.js) captura respuestas no OK y propaga mensajes para mostrarlos en la UI.
- UX: errores se muestran en el componente (estado local errors) y se evita el envío si hay errores.

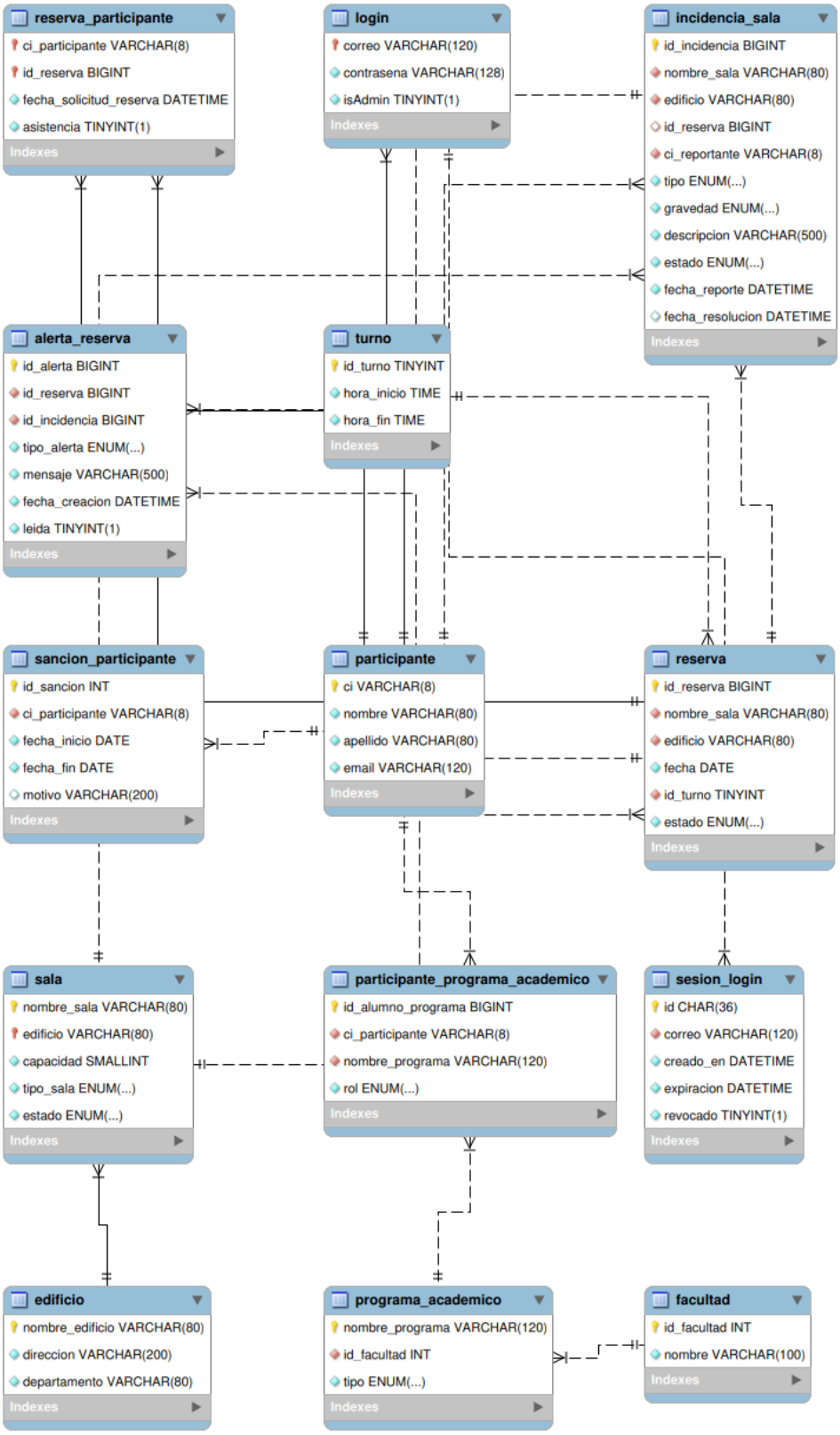
Ejemplo: función de validación en Reservas.jsx

```
function validateReserva({ salaId, fecha, turno, turnoExtra }) {
    const errors = [];
    if (!salaId) errors.push("Seleccione una sala.");
    if (!fecha) errors.push("Seleccione una fecha.");
    else {
        const d = new Date(fecha);
        if (isNaN(d.getTime())) errors.push("Fecha inválida.");
        // fecha anterior a hoy
        const today = new Date();
        today.setHours(0,0,0,0);
        if (d < today) errors.push("La fecha debe ser hoy o futura.");
    }
    if (!turno) errors.push("Seleccione un turno.");
    if (turno && turnoExtra && Number(turno) === Number(turnoExtra)) {
        errors.push("El turno adicional debe ser distinto al turno principal.");
    }
    return errors;
}
```

Anexo

Figura 1

Diagrama de la base de datos generada por MySQL workbench



Bibliografía

Cloudflare. (2025,). *¿Qué Es El Principio de Mínimos Privilegios?*. https://www.cloudflare.com/es-es/learning/access-management/principle-of-least-privilege/

Create check constraints - SQL Server. Recuperado 22 de noviembre de 2025, de https://learn.microsoft.com/en-us/sql/relational-databases/tables/create-check-constraints?view=sql-server-ver17

Silberschatz, A. (2006). *Fundamentos de bases de datos* (5a ed.). McGraw-Hill.

Welcome to Flask — Flask Documentation. https://flask.palletsprojects.com/

Welcome to PyJWT — PyJWT 2.6.0 documentation. https://pyjwt.readthedocs.io/en/stable/