UNIVERSIDAD TECNOLÓGICA NACIONAL

PROYECTO INTEGRADOR

Programación 2

Título del proyecto: Sistema Usuario – Credencial de Acceso

Alumnos: Pablo Molinari, Nicolás Olima, Nicolás Pannunzio, Leonel Mercorelli.

Materia: Programación 2

Comisiones: 4, 15 y 17

Fecha: 17/11/2025

Carrera: Tecnicatura en Programación a Distancia.

Índice

Integrantes y roles

Para la realización de este proyecto, el equipo se organizó dividiendo las responsabilidades principales de la arquitectura del sistema de la siguiente manera:

Pablo Molinari: Desarrollo de la Capa de Persistencia (Desarrollo DAO)

Nicolás Olima: Diseño y Estructura de la Base de Datos (Diseño BD)

Leonel Mercorelli: Desarrollo de la Lógica de Negocio (Desarrollo Service)

Nicolás Pannunzio: Desarrollo de la Interfaz de Consola (Frontend AppMenu)

Elección del dominio y justificación

Se eligió como dominio la gestión de usuarios y credenciales de acceso, dado que representa un escenario común en sistemas informáticos donde se requiere controlar la creación, actualización y la baja lógica de usuarios con sus respectivas credenciales.

El dominio permite aplicar los principales conceptos de la materia, como persistencia transaccional, validaciones, relaciones 1 a 1, manejo de excepciones y arquitectura multicapa, manteniendo un alcance acotado y fácilmente verificable.

Además, facilita el uso de transacciones JDBC y validaciones SQL (triggers), por lo que resulta un excelente entorno para demostrar comprensión de los fundamentos de programación orientada a objetos y persistencia relacional que aprendimos durante la cursada tanto de la materia Programación II, como también la implementación, configuración y sincronización con los servicios de las bases de datos que aprendimos en la materia Bases de datos I .

Diseño: decisiones clave (1 → 1, FK única vs PK compartida) + UML

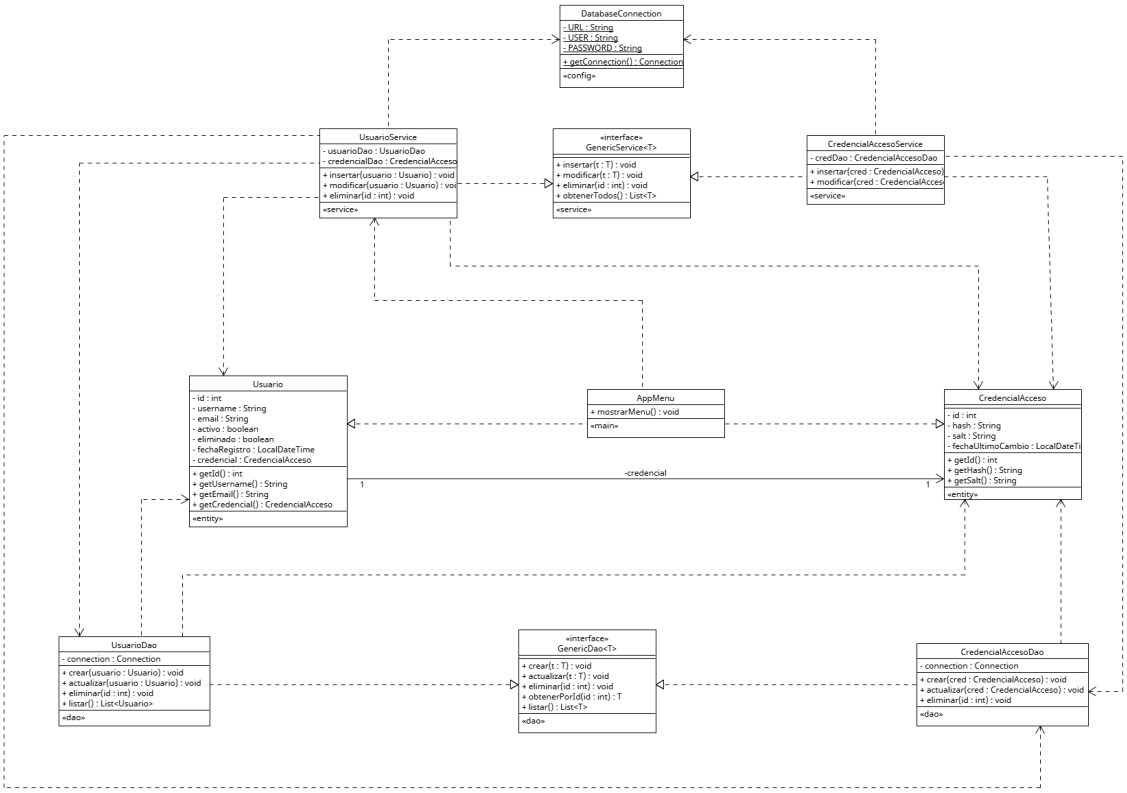
En el diseño se estableció una relación 1 → 1 entre Usuario y CredencialAcceso, dado que cada usuario posee una única credencial y cada credencial pertenece a un único usuario.

Se optó por implementar la relación mediante una clave foránea única (FK) en la tabla usuarios, referenciando la tabla credencialesacceso.

Esta elección se debe a que la credencial no depende existencialmente del usuario (puede persistir temporalmente), por lo que no corresponde usar PK compartida ni composición estricta.

Ventaja: mejora la independencia entre tablas y simplifica operaciones de mantenimiento.

A continuación, se presenta el diagrama UML final del sistema, donde se visualiza la arquitectura multicapa y las relaciones entre clases e interfaces:



Arquitectura por capas

El sistema adopta una arquitectura en capas, que favorece la separación de responsabilidades y la mantenibilidad del código:

- **config:** Maneja la configuración de conexión con la base de datos (DatabaseConnection).
- **entities:** Define las entidades del dominio (Usuario, CredencialAcceso).
- **dao:** Encapsula la persistencia (CRUD sobre las tablas) y la conexión directa con JDBC.
- **service:** Implementa la lógica de negocio, transacciones y validaciones intermedias.
- **main:** Contiene el punto de entrada (Main) y el menú de interacción (AppMenu).

Cada capa **sólo conoce a la capa inferior**, siguiendo el principio de independencia jerárquica:

Persistencia: estructura, orden de operaciones y transacciones

Estructura de la base de datos

La base de datos tpi-bd-i está formada por dos tablas principales:
Usuario y CredencialAcceso, vinculadas entre sí en una relación 1 a 1**.**
Cada usuario tiene una sola credencial, y cada credencial pertenece a un único usuario.

La gestión de transacciones y el orden de las operaciones son responsabilidad exclusiva de la capa de **servicio**, para garantizar la atomicidad de las operaciones que involucran a más de una entidad. Los DAO no manejan transacciones; solo ejecutan operaciones SQL recibiendo una conexión externa.

El flujo de una operación transaccional, como **"Crear un nuevo usuario"**, sigue este orden:

1. El método en UsuarioService obtiene una única conexión a la base de datos.
2. Se deshabilita la confirmación automática (<code>conn.setAutoCommit(false)</code>) para iniciar la transacción.
3. Se sigue un orden estricto para respetar la integridad referencial (dado que usuarios tiene la FK a credencialesacceso):
    - **Paso A:** Se inserta la entidad CredencialAcceso en su tabla, utilizando CredencialAccesoDao y pasando la conexión.
    - **Paso B:** Si el Paso A tiene éxito, se obtiene el ID autogenerado de la credencial recién creada.
    - **Paso C:** Se inserta la entidad Usuario (asignándole la FK del ID obtenido en el Paso B), utilizando UsuarioDao y pasando la *misma* conexión.
4. **Commit (Éxito):** Si ambos pasos (A y C) se completan sin errores, el Service ejecuta <code>conn.commit()</code>, persistiendo todos los cambios en la base de datos de forma atómica.
5. **Rollback (Error):** Si ocurre *cualquier* error durante la transacción (ej. un username duplicado, un email inválido que viola un TRIGGER , o cualquier SQLException), el Service captura la excepción e invoca <code>conn.rollback()</code>. Esto deshace todos los cambios realizados, incluyendo la inserción de la CredencialAcceso (Paso A), asegurando que no queden datos inconsistentes en la base.
6. **Limpieza:** Finalmente, en un bloque finally, se restaura el modo autoCommit(true) y se cierra la conexión.

Este mecanismo garantiza que o se crean ambas entidades (Usuario y Credencial) o no se crea ninguna, cumpliendo con el principio de atomicidad requerido.

Tabla Usuario

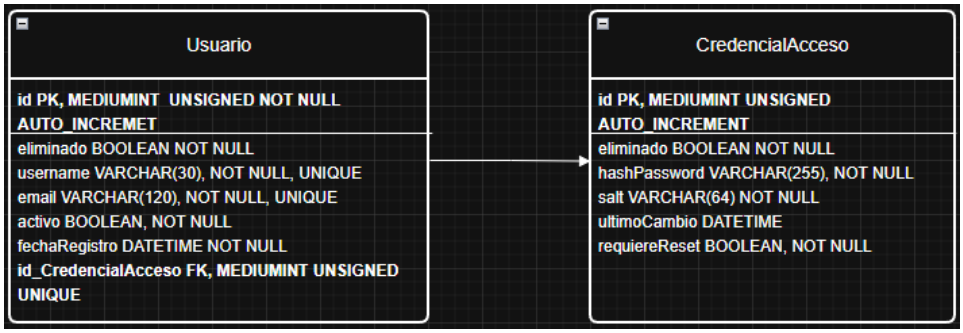
- id: clave primaria autoincremental
- username, email: datos básicos del usuario
- activo: indica si está habilitado
- eliminado: marca de baja lógica

- fechaRegistro: fecha de creación
- id_CredencialAcceso: clave foránea única que apunta a la credencial del usuario

Tabla CredencialAcceso

- id: clave primaria autoincremental
- hashPassword y salt: datos de seguridad simulados
- ultimoCambio: fecha del último cambio de contraseña
- requiereReset: indica si el usuario debe restablecer su clave
- eliminado: marca de baja lógica

Se decidió usar una FK única en lugar de compartir la misma PK, ya que la credencial puede seguir existiendo aunque el usuario sea eliminado.
Esto da más flexibilidad para el mantenimiento o recuperación de datos.



Validaciones y reglas de negocio

- El campo email se valida en la base de datos mediante un trigger SQL, que impide insertar usuarios sin símbolo @.
- Se aplica baja lógica: los registros no se eliminan físicamente.
- Los campos activo y eliminado se inicializan automáticamente al crear el usuario.
- Las fechas (fechaRegistro, ultimoCambio) se actualizan automáticamente en cada inserción o modificación.

- La lógica criptográfica de contraseñas se simula manualmente (no se implementa hashing real), dado que el objetivo del TFI es demostrar persistencia y transacciones, no seguridad avanzada.

Pruebas realizadas

Se verificó el correcto funcionamiento del menú principal (AppMenu), realizando las siguientes pruebas:

- Inserción de un nuevo usuario y credencial (transacción exitosa).
- Listado completo de usuarios no eliminados.
- Búsqueda por ID.
- Actualización de datos de usuario.
- Eliminación lógica.

También se verificó la validación SQL del email intentando ingresar un nuevo usuario con email = 'correo_invalido.com' (sin @), verificando que funcione el rollback.

A. **Verificación al crear nuevo usuario**:

```
===== MENU PRINCIPAL =====
1. Crear nuevo usuario
2. Listar usuarios
3. Buscar usuario por ID
4. Actualizar usuario
5. Eliminar usuario (baja logica)
0. Salir
Seleccione una opcion: 1

--- Crear nuevo usuario ---
Username: Fulanito
Email: correo_invalido.com
Hash Password: CONTRASENIA
Salt: CODIGO
Error al crear usuario: El email debe contener @
```

```
===== MENU PRINCIPAL =====
1. Crear nuevo usuario
2. Listar usuarios
3. Buscar usuario por ID
4. Actualizar usuario
5. Eliminar usuario (baja logica)
0. Salir
Seleccione una opcion: 1

--- Crear nuevo usuario ---
Username: Fulanito
Email: correo_valido@hotmail.com
Hash Password: CONTRASENIA
Salt: CODIGO
Usuario creado con exito.
```

B. Verificación al actualizar un usuario existente:

```
===== MENU PRINCIPAL =====
1. Crear nuevo usuario
2. Listar usuarios
3. Buscar usuario por ID
4. Actualizar usuario
5. Eliminar usuario (baja logica)
0. Salir
Seleccione una opcion: 4

Ingrese el ID del usuario a actualizar: 5
Nuevo email: correo_invalido.com
Activo? (true/false): true
Error al actualizar usuario: El email debe contener @
```

Conclusiones y mejoras futuras

El sistema logró cumplir todos los objetivos propuestos: modelado POO, persistencia, manejo transaccional y validaciones SQL.

Como mejora futura, se podrían implementar:

- Cifrado real de contraseñas (hashing con SHA-256).
- Un sistema de roles y permisos.
- Interfaz gráfica con JavaFX o integración con una API REST.
- Pruebas unitarias automatizadas (JUnit).

Fuentes y herramientas utilizadas

- **IDE:** Apache NetBeans 21
- **DBMS:** MySQL Workbench 8.0
- **Modelado UML:** UMLetino (formato .uxf)
- **IA:** ChatGPT (OpenAI) utilizada para análisis técnico, revisión de código y redacción documental.
- **Otros:** Creately, Paint.NET (edición de diagramas).