TRABAJO FINAL INTEGRADOR PROGRAMACIÓN II

Usuario-CredencialAcceso

Presentado por:

- Joana Gamarra - DNI: 42867054 - yoanagamarra09@gmail.com - Comisión 9

- Emilce Noemí Spital - DNI: 40448362 - emilcespital@gmail.com - Comisión 9

- Gabriela Machin - DNI: 37048164 - gabrielamachin.gm@gmail.com - Comisión 16

- Virginia Pérez - DNI: 37236238 - virginia.m.perez@gmail.com - Comisión 9

Fecha de Entrega: 18 De Noviembre de 2025

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN A DISTANCIA

Índice:

Integrantes y Roles **3**
Dominio seleccionado y justificación **3**
Diseño (Decisiones clave + UML) **4**
Relación 1→1 unidireccional **4**
Estructura del Modelo 6
Arquitectura por capas **7**
Persistencia **8**
Mecanismo de Operaciones y Transacciones Delegadas 9
Control Transaccional en la Capa Service 9
Validaciones y reglas de negocio **10**
1. Detalle de Validaciones Implementadas en la Capa Service 10
2. Relación 1 → 1 11
Mecanismo de Implementación (Impedir más de un B por A): 11
3. Validaciones Implementadas en la Capa Service (Usuario) 12
4. Unicidad de Campos 12
Mecanismo de Implementación (Unicidad de Usuario/Email): 12
5. Coordinación de Servicios (Preparación para Rollback) 13
Pruebas realizadas **13**
Conclusiones y mejoras futuras **13**
Bibliografía **14**

Integrantes y Roles

- Emilce Noemí Spital - **Responsable de la conexión, scripts SQL y estructura relacional**
- Joana Gamarra - **Responsable del modelado de clases y diagrama UML**
- Virginia Pérez - **Responsable de la persistencia con JDBC y PreparedStatement**
- Gabriela Machin - **Responsable de la lógica de negocio, transacciones y consola**

Dominio seleccionado y justificación

El dominio elegido para el Trabajo Final Integrador es **Usuario → CredencialAcceso**, un modelo ampliamente utilizado en sistemas de autenticación y gestión de acceso. Esta selección se fundamenta en su relevancia práctica y su capacidad para representar con claridad una relación **1→1 unidireccional**, donde cada usuario posee exactamente una credencial asociada.

Las razones principales que respaldan esta elección son:

• ***Relevancia en sistemas reales***

Las aplicaciones modernas requieren gestionar usuarios y credenciales; este dominio refleja un caso real y cotidiano en la industria del software.

• ***Claridad conceptual del modelo 1→1***

La relación es simple, directa y adecuada para implementar las restricciones de unicidad requeridas en el proyecto.

• ***Atributos técnicos significativos***

Permite trabajar con atributos relevantes como `username`, `email`, `hashPassword`, `salt` y `requiereReset`, que introducen conceptos de seguridad y buenas prácticas en el manejo de contraseñas.

• ***Buena integración con el patrón DAO***

La separación entre Usuario (A) y CredencialAcceso (B) encaja naturalmente en una arquitectura basada en DAOs y servicios con transacciones.

Este dominio aporta complejidad suficiente para aplicar todas las técnicas solicitadas sin perder claridad en la estructura del sistema.

Diseño (Decisiones clave + UML)

El diseño de la base de datos se elaboró siguiendo los principios formales de la ingeniería de datos, priorizando la **integridad referencial**, la **normalización** y la **separación clara de responsabilidades** entre entidades.

Siguiendo estas buenas prácticas, se definieron dos entidades principales:

- usuario
- credencial_acceso

Ambas tablas se construyeron respetando las características fundamentales de un modelo relacional bien diseñado:

- Clave primaria (id)

Cada entidad del modelo debe poseer un identificador único para asegurar su distinción dentro del conjunto de registros.
Por ello, cada tabla cuenta con un campo `id BIGINT AUTO_INCREMENT PRIMARY KEY`.

- Campo eliminado para bajas lógicas

La estrategia de "baja lógica" o *soft delete* es ampliamente utilizada en sistemas modernos (Fowler, 2003 – *Patterns of Enterprise Application Architecture*).
Permite mantener la trazabilidad histórica sin eliminar físicamente un registro.

```
eliminado BOOLEAN NOT NULL DEFAULT 0
```

- Restricciones NOT NULL y UNIQUE

Siguiendo las definiciones de integridad de dominio establecidas en SQL estándar, se aplicaron:

- NOT NULL para impedir datos incompletos.
- UNIQUE para mantener la unicidad de atributos como email y username, asegurando consistencia y evitando duplicidades.

• Tipos de datos adecuados
- VARCHAR para cadenas de tamaño variable
- BOOLEAN para flags
- BIGINT para claves primarias
- DATETIME para campos temporales

Relación 1→1 unidireccional

Para modelar la relación entre *Usuario* y *CredencialAcceso*, se estudiaron las dos estrategias formales para implementar una relación **uno a uno** en bases de datos relacionales

Modelo A — Clave primaria compartida (PK = FK)

En este diseño, la tabla secundaria utiliza como clave primaria la misma clave primaria de la tabla principal.
Ventajas:

- Garantiza la relación 1→1 estricta.

Desventajas:

- Alto acoplamiento entre entidades.
- Complejiza la inserción y mantenimiento de registros.
- Limita la independencia estructural.

Modelo B — Clave foránea única en la tabla secundaria (FK UNIQUE)

En esta estrategia, la tabla secundaria tiene su propio `id`, pero incluye una columna con restricción `UNIQUE` que hace referencia al `id` de la tabla principal.
Es recomendada en sistemas reales por su mayor flexibilidad y separación conceptual.

Se adoptó el **Modelo B – FK única** por las siguientes razones:

- Mantiene independencia entre las entidades. El usuario existe como entidad autónoma, mientras que la credencial depende de él, sin necesidad de compartir claves primarias. Esta independencia mejora la claridad del diseño y cumple con los principios de la tercera forma normal (3FN).
- Reduce el acoplamiento entre tablas.
- Facilita mantenimiento, depuración y escalabilidad. Permite agregar atributos adicionales a `credencial_acceso` sin afectar la tabla `usuario`.
- Coincide con el enunciado del TPI. El lineamiento del trabajo pide explícitamente una FK única para garantizar la relación 1→1.

La relación se implementó con la siguiente definición SQL:

usuario_id BIGINT UNIQUE,
FOREIGN KEY (id_usuario) REFERENCES usuario(id) ON DELETE CASCADE

Esto garantiza:

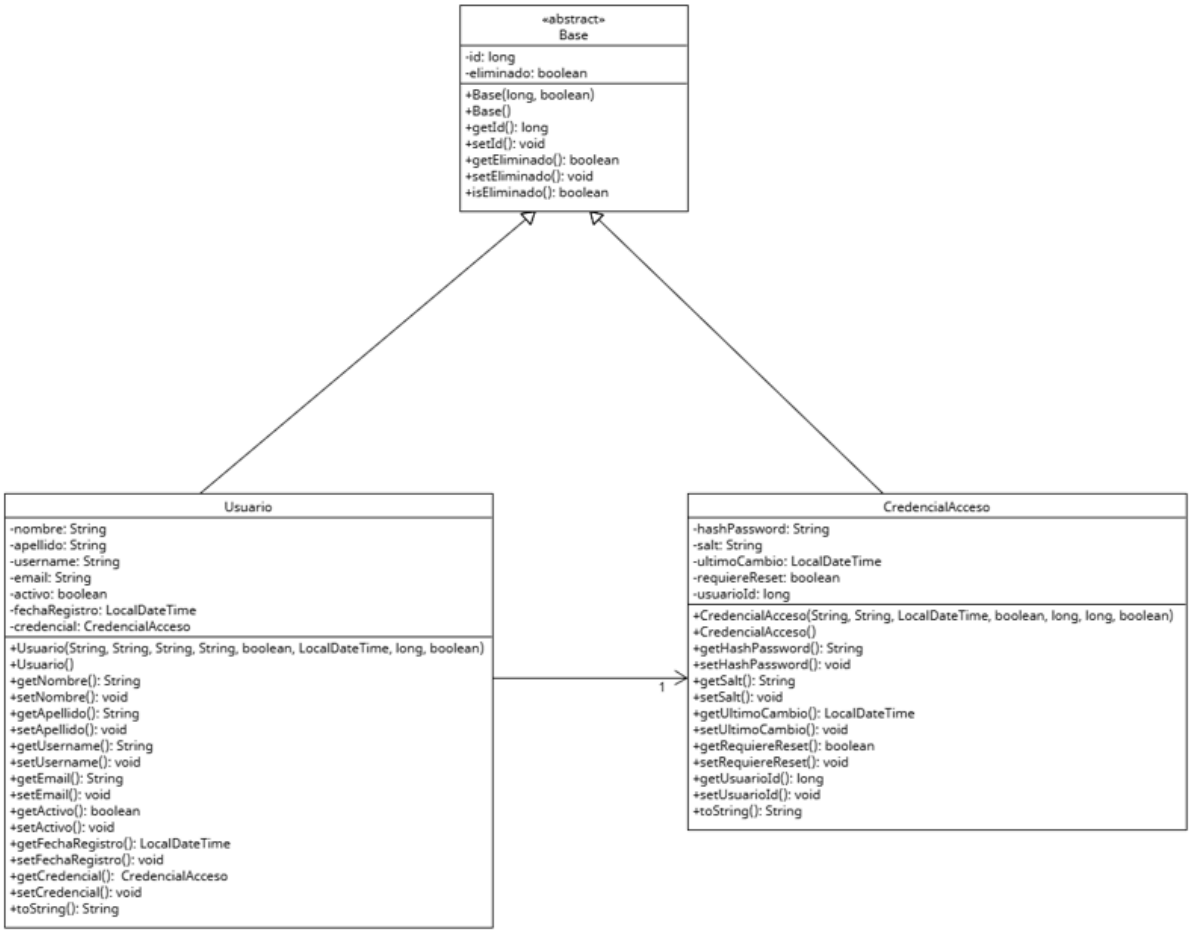
•Asociaciones válidas (Integridad referencial): Cada credencial debe apuntar a un usuario existente. Si se intenta insertar una credencial con id_usuario inexistente, MySQL lo rechaza.
• Relación verdaderamente 1→1: La restricción UNIQUE evita que un usuario tenga múltiples credenciales.
• Comportamiento coherente en eliminaciones: ON DELETE CASCADE garantiza que:

- Si un usuario se elimina, su credencial se elimina automáticamente.
- Evita credenciales huérfanas.
- Cumple el comportamiento esperado en sistemas de autenticación reales.

Estructura del Modelo

1. **Base (Clase Abstracta):** Actúa como la raíz de la jerarquía, definiendo los atributos esenciales comunes a todas las entidades persistentes: el **ID** y el estado de **eliminación lógica** (`eliminado`).
2. **Usuario:** Hereda de **Base**. Esta clase se centra en los datos personales (`nombre`, `apellido`, `username`, `email`) y el estado de la cuenta (`activo`, `fechaRegistro`).
3. **CredencialAcceso:** También hereda de **Base**. Se enfoca en la seguridad y autenticación, almacenando la información sensible como el **hash** y el **salt** de la contraseña, el `ultimoCambio` y si `requiereReset`. Además, incluye el `usuarioId` como clave foránea para la persistencia.

El diagrama muestra una estructura jerárquica robusta y una clara asociación entre el modelo de datos personales (**`Usuario`**) y el modelo de seguridad (**`CredencialAcceso`**).



Arquitectura por capas

Para cumplir con las consignas del Trabajo Final Integrador y garantizar un código limpio, legible y escalable, implementamos una arquitectura por capas basadas en los patrones DAO y Service. Esta estructura asegura la separación de intereses al asignar una responsabilidad única a cada paquete, como se detalla a continuación:

1. **Paquete Models/**

Esta es la capa más baja y contiene las clases de dominio que representan el estado de la información de negocio. Dicha capa está conformada por:

- Una ***clase base abstracta***, la cual contiene los atributos *ID* y *eliminado*. Estos serán heredados a todas las clases presentes que van a formar parte de nuestro dominio.
- La entidad ***Usuario*** (Clase A) y la entidad ***CredencialAcceso*** (Clase B). Cada una de estas entidades incluye sus atributos propios, y heredan desde la clase base abstracta, garantizando la reutilización del código y la estandarización de atributos comunes.

Se estableció la asociación unidireccional 1 → 1 desde *Usuario* hacia *CredencialAcceso* (Usuario tiene una referencia a CredencialAcceso).
Esta capa no contiene lógica de negocio ni dependencias externas, cumpliendo con el principio de independencia de dominio. Su propósito es modelar de forma clara y consistente los conceptos principales del sistema, además sirve como base para las operaciones que se irán haciendo en las capas de aplicación y persistencia.

2. **Paquete config/**

Este paquete se encarga de aislar la lógica necesaria para establecer la conectividad con el motor de base de datos. Proporciona un punto centralizado y estático para lograr la conexión con la base de datos MySQL. Contiene una clase clave, que es la ***DatabaseConnection***, la cual encapsula los detalles del driver JDBC y el DriverManager, devolviendo un objeto java.sql.Connection. Una de sus ventajas es el manejo de las credenciales y URLs de conexión en un solo lugar, esto hace que la configuración y el mantenimiento sean más sencillos.

3. **Paquete dao/**

Esta es la capa de persistencia que se encarga de interactuar con la base de datos usando JDBC, aislando al resto de la aplicación de las sentencias SQL. Su responsabilidad es implementar las operaciones CRUD (crear, leer, actualizar, eliminar) sobres las entidades de dominio.
Se utiliza la interfaz **GenericDAO<T>** para definir el contrato CRUD, luego se implementa en los DAOs concretos ***UsuarioDAO*** y ***CredencialAccesoDAO***, utilizando obligatoriamente ***PreparedStatement***.

Los métodos del DAO fueron diseñados para aceptar una Connection externa como parámetro, lo que les permite participar en las transacciones que seran organizadas por la capa Service.

4. **Paquete service/**

Esta es la capa de lógica de negocio, control de flujo y validación, que actúa como el coordinador de las operaciones. Su responsabilidad es contener las reglas de negocio, realizar validaciones y gestionar la lógica de las transacciones.
La capa Service es la única responsable de gestionar transacciones compuestas. Esto se logra a través de:

- Deshabilitación del *AutoCommit* sobre una conexión compartida.
- Llamada a múltiples métodos del DAO.
- Ejecución de *commit()* en el caso de que todas las operaciones sean exitosas o *rollback()* si se produce una excepción.
- Restablecimiento de *autoCommit(true)* al finalizar.

5. **Paquete main/**

Este paquete representa el punto de entrada y la capa de presentación de la aplicación. Su responsabilidad es contener la lógica de arranque de la aplicación e implementar el menú interactivo de consola (`AppMenu`). Tiene como función recibir las entradas del usuario, interactúa con la capa Service (para mantener la separación de intereses), y se encarga del manejo robusto de entradas inválidas, IDs inexistentes y la impresión de mensajes claros de éxito o error.

Debe permitir el CRUD completo de ambas entidades (`Usuario` y `CredencialAcceso`), además de la búsqueda especializada por un campo clave como `username`.

Persistencia

Al implementarse los Data Access Objects (DAOs), se puede articular la capa de persistencia. Se utilizaron ***UsuarioDAO*** y ***CredencialAccesoDAO***, las cuales adhieren a la interfaz genérica ***GenericDAO<T>***. Esta arquitectura, lo que hace es aislar la lógica de negocio de las complejidades y particularidades de la base de datos subyacente. Los DAOs utilizan de forma consistente objetos ***java.sql.PreparedStatement*** para ejecutar todas las operaciones, por lo que es vital para prevenir la inyección SQL y optimizar la ejecución de queries repetitivas.

Mecanismo de Operaciones y Transacciones Delegadas

Dentro de un DAO, se sigue un flujo operativo bien definido que permite generar una operación CRUD. Primero se obtienen las *queries* SQL estáticas predefinidas en la clase. Luego, el método establece la conexión a través de ***`DatabaseConnection.getConnection()`***. Una vez que se realiza lo anterior, se crea el ***PreparedStatement***. Finalmente, se ejecuta la operación (***`executeUpdate`*** o ***`executeQuery`***). Es crucial notar que el uso del bloque **Try-with-Resources** asegura que todos los recursos de JDBC (`Connection`, `PreparedStatement`, `ResultSet`) se cierren automáticamente, liberando recursos sin necesidad de manejo manual en un bloque `finally`.

Algo sumamente importante en este diseño es la participación transaccional. Los DAOs brindan métodos diferenciados:
Métodos "Self-Contained" (`insertar`, `actualizar`): Estos métodos gestionan y cierran su propia conexión, operando en su propia transacción implícita si el *autocommit* está activo.
Métodos Transaccionales (`...Tx`): Estos métodos aceptan una **`Connection` externa** como argumento. Al aceptar esta conexión, el DAO delega por completo la responsabilidad del control transaccional a la capa llamante, ejecutando solo las sentencias SQL sin llamar a `commit()` o `rollback()`.

Control Transaccional en la Capa Service

El verdadero **control transaccional** se realiza obligatoriamente en la **Capa Service**. Esto garantiza la atomicidad de las operaciones de negocio que implican la modificación de múltiples entidades (por ejemplo, registrar un usuario y, de forma dependiente, su credencial de acceso). La lógica de la capa Service sigue un estricto protocolo de flujo:

1. **Apertura de Transacción:** Se obtiene una conexión (`Connection conn = ...`), e inmediatamente se deshabilita el *autocommit* de la conexión: ***`conn.setAutoCommit(false);`***.
2. **Ejecución:** Se invocan todos los métodos transaccionales (`...Tx`) necesarios en los DAOs, pasándoles la conexión abierta (`conn`).
3. **Control de Flujo:**
    - **Éxito:** Si todas las llamadas se completan sin lanzar excepciones, se invoca ***`conn.commit()`*** para hacer permanentes todos los cambios.
    - **Fallo:** Si ocurre cualquier excepción (`catch`), se ejecuta ***`conn.rollback()`*** para deshacer todos los cambios realizados por la conexión hasta ese momento, garantizando la consistencia de los datos.
4. **Cierre:** Finalmente, en el bloque `finally`, se asegura que la conexión sea restablecida a su estado original (***`conn.setAutoCommit(true);`***) y posteriormente cerrada, liberando los recursos del sistema. Este patrón garantiza que, ante cualquier resultado, los recursos se limpien correctamente.

Validaciones y reglas de negocio

La capa de Servicio (`CredencialAccesoServiceImpl`) es el componente central para la aplicación de las reglas funcionales y la garantía de la integridad de los datos en el sistema. A continuación, se detallan las validaciones y la regla de negocio más crítica implementadas para la entidad `CredencialAcceso`.

1. Detalle de Validaciones Implementadas en la Capa Service

Todas las validaciones previas a la persistencia se gestionan a través del método privado `validateCredencial(CredencialAcceso credencial)` dentro de `CredencialAccesoServiceImpl`. Este enfoque asegura que cualquier intento de guardar o actualizar una credencial cumpla con los estándares de seguridad y consistencia definidos. Las validaciones implementadas son las siguientes:

Validación	Condición Evaluada	Excepción Lanzada	Propósito
Objeto Nulo	Verifica que el objeto CredencialAcceso no sea nulo.	IllegalArgumentException	Prevenir errores de referencia nula.
Longitud Contraseña	Valida que el hashPassword (contraseña) **no sea nulo ni vacío**, y que su longitud sea de **al menos 8 caracteres**.	IllegalArgumentException	Asegurar un nivel básico de complejidad y seguridad en el hash de la contraseña.
Obligatoriedad del Salt	Valida que el salt (valor único para la encriptación) **no sea nulo ni vacío**.	IllegalArgumentException	Garantizar que el proceso de *hashing* sea seguro contra ataques de *rainbow table*.

ID de Usuario	Valida que el usuarioId asociado (FK) sea **mayor a 0**.	IllegalArgumentException	Asegurar que la credencial esté vinculada a un usuario existente y válido.

Aunque en este contexto no se manejan campos como email, DNI o CUIT, la estructura del método `validateCredencial` permite incorporar fácilmente validaciones de formato (por ejemplo, expresiones regulares para email o longitud fija para DNI) si la entidad lo requiriera, manteniendo la responsabilidad de la coherencia de datos dentro de la capa Service.

2. Relación 1 → 1

La regla de negocio más importante implementada en la capa Service es la que garantiza la **cardinalidad de uno a uno (1 → 1)** entre un `Usuario` (entidad A) y una `CredencialAcceso` (entidad B). Esto significa que un usuario solo puede tener una única credencial de acceso asociada.

Mecanismo de Implementación (Impedir más de un B por A):

Esta regla se aplica dentro del método `save(CredencialAcceso entity)` de la siguiente manera:

1. Validación Previa: Primero, se invoca `validateCredencial(entity)` para asegurar que los datos son válidos.

2. Verificación de Existencia: Se realiza una consulta específica al DAO utilizando el ID del usuario: CredencialAcceso existente = credencialDAO.getByIdUsuario((int) entity.getUsuarioId());

3. Aplicación de la Regla: Si la consulta devuelve un resultado (`existente != null`), significa que ya hay una credencial asociada a ese usuario. En este caso, el Service **impide la inserción** y lanza una excepción controlada: throw new ServiceException("Ya existe una credencial para el usuario con ID: " + entity.getUsuarioId());

Este mecanismo **impide activamente** la creación de credenciales duplicadas para un mismo usuario, reforzando la restricción **UNIQUE** a nivel de base de datos (`id_usuario` en la tabla `credencial`) con una capa de lógica de negocio que ofrece un manejo de errores más informativo y amigable para la aplicación.

Para un sistema completo, la entidad Usuario es la base, y su capa de servicio (UserServiceImpl) es responsable de mantener la coherencia de la información personal.

3. Validaciones Implementadas en la Capa Service (Usuario)

Las validaciones se centran en la información personal y de acceso del usuario, típicamente dentro del método validateUsuario(Usuario usuario).

Validación	Condición Evaluada	Excepción Lanzada	Propósito
Campos de Identificación	Valida que nombre y apellido **no sean nulos ni vacíos**.	IllegalArgumentException	Asegurar la presencia de datos básicos de identificación.
Formato de Contacto	Valida que el email y username **no sean nulos** y, opcionalmente, que el email cumpla con un **patrón Regex** estándar.	IllegalArgumentException	Garantizar que los campos de contacto sean utilizables.
Consistencia de Datos	Verifica que el ID del usuario (usuario.getId()) sea válido (mayor a 0) antes de cualquier operación de update o delete.	ServiceException	Prevenir modificaciones o eliminaciones accidentales de registros inexistentes.

4. Unicidad de Campos

Una regla crítica en la creación de cuentas de usuario es evitar duplicados que puedan causar conflictos en el inicio de sesión o la identidad.

Mecanismo de Implementación (Unicidad de Usuario/Email):

Esta regla se aplica dentro del método save(Usuario entity) de UserServiceImpl para validar campos que deben ser únicos globalmente:

1. **Validación Previa:** Se invoca validateUsuario(entity) para validar formatos y presencia de datos.
2. **Verificación de Unicidad:** Antes de la inserción, el Service consulta al UsuarioDAO si ya existe un registro con el mismo username o email:

```
Java
Usuario existentePorUsername = usuarioDAO.getByUsername(entity.getUsername());
Usuario existentePorEmail = usuarioDAO.getByEmail(entity.getEmail());
```

3. **Aplicación de la Regla:** Si cualquiera de las consultas devuelve un resultado, se lanza una excepción específica:

```
Java
if (existentePorUsername != null) {
    throw new ServiceException("El nombre de usuario ya está registrado.");
}
```

Este control en la capa Service garantiza que la identidad de cada usuario sea única en el sistema, siendo una protección complementaria a las restricciones UNIQUE definidas en la base de datos.

5. Coordinación de Servicios (Preparación para Rollback)

La existencia de ambos servicios (UserServiceImpl y CredencialAccesoServiceImpl) permite la demostración de **Transacciones de Negocio Complejas**.

- El proceso de **Registro de un Nuevo Usuario** requiere coordinar las operaciones de ambos servicios:
    1. Guardar la entidad Usuario (usando UserServiceImpl).
    2. Guardar la entidad CredencialAcceso (usando CredencialAccesoServiceImpl).
- En la demostración de **Rollback** se simula una transacción que intenta ejecutar estas dos operaciones. Si la **segunda operación (guardar la Credencial)** falla por una validación o restricción, el servicio coordinador se encargará de invocar el mecanismo de **rollback** de la base de datos, asegurando que la **primera operación (el Usuario ya insertado)** también se revierta, manteniendo la atomicidad de la operación de negocio de registro completo.

Pruebas realizadas

Las pruebas manuales ejecutadas y sus resultados se adjuntan en el archivo **Reporte de casos de prueba manuales.pdf**. Se puede acceder de forma directa en el siguiente link: https://github.com/Joygamarra09/Trabajo_Integrador_Programaci-n2/blob/main/Reporte%20de%20casos%20de%20prueba%20manuales.pdf

Conclusiones y mejoras futuras

El desarrollo de este Trabajo Final Integrador permitió consolidar de manera efectiva los contenidos abordados durante la cursada de Programación 2, integrando aspectos teóricos y prácticos en un proyecto completo y funcional. A través del diseño de una base de datos relacional, la implementación de una arquitectura por capas y el uso de JDBC para la persistencia, se logró construir una aplicación que respeta principios de claridad, modularidad y separación de responsabilidades.

La relación uno a uno entre las entidades seleccionadas se implementó correctamente tanto a nivel de modelo como de base de datos, garantizando integridad referencial mediante restricciones y el uso adecuado de claves primarias y foráneas. El manejo transaccional mediante commit y rollback aportó robustez al sistema, asegurando operaciones consistentes ante posibles fallos.

Asimismo, la utilización del patrón DAO, junto con una capa de servicios encargada de las validaciones y la coordinación de acciones, permitió lograr un código más mantenible, organizado y alineado con buenas prácticas de programación orientada a objetos. La capa de presentación, a través del menú de consola, ofreció una interfaz simple pero completa para interactuar con la aplicación y verificar el funcionamiento de todas las funcionalidades solicitadas.

En conjunto, el proyecto constituye una experiencia valiosa que integró habilidades técnicas, diseño estructurado y trabajo colaborativo. Además, deja sentadas las bases para posibles ampliaciones futuras, como la incorporación de nuevas entidades, mejoras en la interfaz o mecanismos adicionales de seguridad. En síntesis, el sistema desarrollado cumple satisfactoriamente con los requisitos planteados y evidencia un proceso de aprendizaje sólido y significativo.

Este proyecto también marcó una evolución significativa en nuestra metodología de trabajo colaborativo. Inicialmente realizamos commits y pushes directamente a la rama main, pero identificamos la necesidad de adoptar un flujo de trabajo más ordenado y profesional. La implementación de branches con pull requests para revisión, seguida de la integración controlada a main, nos permitió coordinar mejor los cambios y mejorar la trazabilidad de nuestro proyecto. De esta manera ganamos mayor experiencia práctica en el uso de Git y GitHub en un entorno de desarrollo real.

En cuanto a mejoras futuras, el sistema podría ampliarse en varias direcciones. A nivel de experiencia de usuario, se podrían implementar validaciones en tiempo real en la capa de presentación, permitiendo que el usuario reciba feedback inmediato sobre datos incorrectos antes de que estos lleguen al service o a la base de datos. Esto mejoraría significativamente la usabilidad y reduciría errores.

En el ámbito técnico, se contempla la incorporación de una interfaz gráfica para una experiencia más intuitiva, algoritmos de seguridad más avanzados para el manejo de contraseñas, un sistema de logs robusto para registrar errores y actividades, y pruebas automáticas con JUnit para garantizar la calidad del código.

En cuanto a escalabilidad, sería posible extender el modelo con nuevas entidades o roles de usuario, e incluso migrar a un framework de persistencia más avanzado como JPA o Hibernate. Estas mejoras permitirían escalar el sistema y alinearlo aún más con estándares profesionales de desarrollo.

Bibliografía

[Programación 2]. (2025). *Clases Abstractas, Herencia, Interfaces y Excepciones* [Vídeos de apoyo y material didáctico].

- HERENCIA 👪 en la POO y UML 📐
- HERENCIA 👪 (extends) en Java ☕
- SUPER en Java ☕Los constructores NO se heredan ❌
- ABSTRACT en Java ☕ CLASES abstractas 💭
- INTERFACES 📜 (implements) en Java ☕ Acá lo vas a entender ✅
- Qué es una EXCEPCIÓN 💥❓ Manejo de errores al programar
- TRY-CATCH-FINALLY 🥏 en Java ☕ Capturar EXCEPCIONES 💥
- EXCEPCIONES 💥 en Java ☕ Cuáles hay ❓
- MULTICATCH en Java ☕ Cómo Discriminar EXCEPCIONES 💥

- **Investigación Complementaria:** Consultas de apoyo y contraste de conceptos (ej., diseño de *PreparedStatement*, patrones transaccionales) utilizando **Inteligencia Artificial Generativa (IA).**