Práctica: Motor de Juego de Rol con Hibernate, DAO/DTO, Excepciones, Logs y Web

Contenido

Práctica: Motor de Juego de Rol con Hibernate, DAO/DTO, Excepciones y Logs 1

1. Contexto de la práctica 2
2. Objetivos 2
3. Descripción funcional del juego 3
   - 3.1. Raza 3
   - 3.2. Criatura 3
   - 3.3. Equipamiento: armas, escudos, pociones 4
   - 3.4. Historia, capítulos y eventos 4
   - 3.5. Reglas de grupo 4
   - 3.6. Límites de ataque y defensa 5
4. Arquitectura por capas y patrones 5
   - 4.1. Singleton para la SessionFactory (Hibernate) 5
   - 4.2. Patrón DAO 5
   - 4.3. Patrón DTO 6
   - 4.4. Capa de servicios / lógica de negocio 6
5. Excepciones personalizadas 7
6. Logs 7
7. Sistema de usuarios, roles e imagen de perfil 8
   - 7.1. Entidad Usuario 8
   - 7.2. Roles de usuario 8
   - 7.3. Relaciones entre Usuario y Criatura 9
   - 7.4. Funcionalidad de login y permisos 9
   - 7.5. DAO, DTO, servicios y excepciones para Usuario 10
   - 7.6. Logs relacionados con usuarios 11
8. Parte web 11
   - 8.1 Página de login 11

1. Contexto de la práctica

Desarrollar el **motor de un juego de rol**.

El juego se basa en **criaturas** que pertenecen a una **raza**, equipadas con **armas**, **escudos** y **pociones**, que avanzan por una **historia dividida en capítulos** con eventos aleatorios y combates contra un "monstruo final".

La práctica se diseñará pensando en que, más adelante, este motor pueda reutilizarse en una **aplicación web**, con sistema de login y gestión de partidas.

Toda la información del juego (razas, criaturas, armas, escudos, pociones, capítulos, etc.) se almacenará en una **base de datos** y se accederá a ella usando **Hibernate**.

2. Objetivos

1. Modelar el dominio del juego de rol mediante **entidades JPA/Hibernate**.
2. Acceder a la base de datos usando:
    - **Patrón Singleton** para gestionar la SessionFactory de Hibernate.
    - **Patrón DAO** para separar la lógica de acceso a datos.
    - **Patrón DTO** para desacoplar las entidades de la capa de presentación.
3. Implementar la lógica básica del juego:
    - Cálculo de ataque y defensa de las criaturas.
    - Desarrollo de capítulos con eventos y combates.
    - Subida de nivel / ganancia de puntos al superar capítulos.
4. Controlar errores mediante **excepciones personalizadas**, incluyendo:
    - Límites máximos de ataque y defensa.
    - Errores de negocio (por ejemplo, grupo con más de 3 criaturas).
5. Configurar un sistema de **logs** para registrar:
    - Eventos importantes del juego.
    - Errores y excepciones.
    - Información útil para depuración.

3. Descripción funcional del juego

El juego tendrá, como mínimo, los siguientes elementos:

3.1. Raza

Una **raza** define las características base de una criatura.

Atributos mínimos:

- id
- tipo (nombre de la raza, por ejemplo: Humano, Elfo, Orco...)
- fuerza
- resistencia
- velocidad
- magia

Estas características podrán influir en el cálculo del **ataque** y la **defensa**.

3.2. Criatura

La **criatura** es el personaje jugable o enemigo en el juego.

Atributos mínimos:

- id
- nombre
- nivel o experiencia (podéis elegir uno de los dos modelos, pero debe estar claro)
- Colecciones de:
    - escudos
    - armas
    - pociones
- puntosVida
- Relación con **Raza** (muchas criaturas pueden compartir la misma raza).

Métodos lógicos (en la capa de dominio o de servicio):

- atacar()
  Fórmula base sugerida:
  resultado = dado10 * nivel + puntosArmas + modificadorRaza

- defender()
  Fórmula base sugerida:
  resultado = dado10 * nivel + puntosEscudos + modificadorRaza

Nota: el valor concreto de modificadorRaza y cómo se calculan puntosArmas y puntosEscudos forma parte del diseño del alumno.

3.3. Equipamiento: armas, escudos, pociones

Deberán existir entidades (o interfaces + clases concretas, si queréis ir más lejos) para:

- **Arma**: nombre, poder de ataque, tipo...
- **Escudo**: nombre, poder de defensa, tipo...
- **Poción**: nombre, efecto (subir vida, subir magia, etc.), cantidad de efecto.

Las criaturas tendrán **colecciones** de estos objetos.

3.4. Historia, capítulos y eventos

El juego tendrá una **historia** dividida en **N capítulos**.
Cada capítulo contiene:

- Una lista de **eventos**.
- Un **monstruo final** (otra criatura).

En cada capítulo, se producirán varios eventos. Ejemplos:

- Una trampa que quita puntos de vida.
- Un cofre que da puntos adicionales o una poción.
- Un evento neutro donde no pasa nada.

Para determinar el resultado de un evento, se puede simular tirando un **dado de 20** (valor aleatorio entre 1 y 20) y aplicando reglas simples.

Al final del capítulo, el **grupo de criaturas del jugador** se enfrenta al monstruo final del capítulo.
Si **al menos una criatura del grupo sobrevive**, se pasa al siguiente capítulo y las criaturas **ganan puntos** (por ejemplo, nivel, experiencia o vida).

3.5. Reglas de grupo

- El juego deberá permitir iniciar una partida con de **2 a 5 criaturas** máximo.
- Si se intenta iniciar el juego con más de 5 criaturas o menos de 2, se deberá lanzar una **excepción de negocio**.

3.6. Límites de ataque y defensa

Se deben definir **valores máximos permitidos** para:

- Ataque total de una criatura.
- Defensa total de una criatura.

Si al calcular el ataque o la defensa se supera el límite definido, se deberá:

- Lanzar una **excepción personalizada** (por ejemplo, LimiteCombateException).
- Registrar el error en los **logs**.

4. Arquitectura por capas y patrones

4.1. Singleton para la SessionFactory (Hibernate)

- Implementar una clase tipo HibernateUtil que:
    - Gestione una única instancia de SessionFactory usando **patrón Singleton**.
    - Tenga constructor privado.
    - Tenga un método público estático getSessionFactory() que devuelva siempre la misma instancia.
- Todos los DAOs deberán usar esta clase para obtener la sesión.

4.2. Patrón DAO

Para las entidades principales (al menos):

- Raza
- Criatura
- Arma
- Escudo
- Pocion
- (Opcional: Capitulo, Evento, Usuario si se diseña ya)

Deberá existir:

- Una **interfaz DAO** (por ejemplo, CriaturaDAO, RazaDAO, etc.) que defina operaciones CRUD básicas.

- Una **implementación Hibernate** (por ejemplo, CriaturaDAOImpl).

Responsabilidades de los DAOs:

- Gestionar sesiones y transacciones.
- Realizar las operaciones de persistencia.
- Lanzar **excepciones DAO personalizadas** si ocurre algún problema de acceso a datos.

4.3. Patrón DTO

Crear **DTOs** para las entidades que vayan a usarse en la capa de presentación o en una futura capa web, por ejemplo:

- CriaturaDTO
- RazaDTO
- ArmaDTO
- EscudoDTO
- PocionDTO
- (Opcional más adelante: UsuarioDTO, CapituloDTO, etc.)

Los DTOs se usarán:

- En la capa de presentación por consola (menús, datos que se muestran al usuario).
- En la capa de servicios.

Debe existir lógica clara para convertir **Entidad ⇄ DTO** (en la capa de servicios o en clases auxiliares).

4.4. Capa de servicios / lógica de negocio

- Implementar una capa de servicios que:
    - Coordine las llamadas a los DAOs.
    - Trabaje con DTOs.
    - Implemente la lógica del juego:
        - Creación de criaturas con equipamiento.
        - Cálculo de ataque y defensa.
        - Avance por capítulos.

- Aplicación de eventos.
- Comprobación de límites de ataque/defensa.
- Validación del grupo (máximo 3 criaturas).

5. Excepciones personalizadas

Definir, como mínimo:

1. DaoException (o similar):
    - Para errores relacionados con la base de datos / Hibernate.
2. JuegoException o BusinessException:
    - Para errores de reglas del juego (más de 3 criaturas, valores inválidos, etc.).
3. LimiteCombateException (o nombre similar):
    - Cuando el ataque o la defensa superen el máximo permitido.

Todas estas excepciones deben tener:

- Mensajes descriptivos.
- Posibilidad de encadenar la causa (Throwable cause) cuando venga de Hibernate u otra capa.

La capa de presentación **no debe ver directamente** excepciones técnicas de Hibernate.

6. Logs

Configurar un sistema de logs (Log4j2, SLF4J+Logback, etc.) para:

- Registrar el **inicio y fin del juego**.
- Registrar la **creación y carga** de criaturas, razas, armas, etc.
- Registrar el **inicio y fin de cada capítulo**.
- Registrar el **resultado de combates importantes**.
- Registrar los **errores y excepciones** con nivel ERROR.

Al menos deben verse los logs en consola. Opcionalmente, se pueden guardar en un fichero.

7. Sistema de usuarios, roles e imagen de perfil

Además de las entidades del juego (Raza, Criatura, Arma, Escudo, Poción, Capítulo, etc.), se deberá implementar un **sistema de usuarios** almacenado en base de datos, que permita diferenciar entre distintos tipos de rol y que guarde también una **imagen de perfil** del usuario.

7.1. Entidad Usuario

Se deberá crear la entidad **Usuario** con, al menos, los siguientes campos:

- id (clave primaria, generado automáticamente).
- username (nombre de usuario para el login, único).
- email (correo electrónico, único).
- password (contraseña para el acceso; inicialmente se puede guardar en texto plano, pero debe estar en un campo propio, preparado para sustituir después por un hash).
- rol (tipo de usuario dentro del sistema).
- fechaAlta (fecha de creación/registro del usuario).
- activo (booleano que indique si el usuario está activo o bloqueado).
- **Imagen de perfil del usuario**, que se podrá modelar de una de estas formas (a elegir y justificar en la memoria):
    - Como un campo String que guarde la **ruta** de la imagen (en disco o en un servidor web).
    - Como un campo binario (BLOB) que almacene la imagen directamente en la base de datos.

El diseño concreto (ruta vs BLOB) forma parte de la decisión de diseño del alumno y deberá explicarse en el documento de entrega.

7.2. Roles de usuario

El sistema deberá contemplar como mínimo dos tipos de rol:

1. **ADMINISTRADOR**
    - Puede dar de alta nuevos usuarios (crear otros usuarios en el sistema).
    - Puede listar usuarios existentes.

- Puede activar/desactivar usuarios.
- Opcionalmente, puede modificar su rol (por ejemplo, convertir un usuario jugador en administrador o viceversa).

2. **JUGADOR**
   - Puede **crear y gestionar sus propias criaturas** (altas, modificaciones y, si se desea, bajas lógicas).
   - Puede **iniciar partidas** y jugar utilizando las criaturas que tenga asociadas.
   - No puede dar de alta otros usuarios ni gestionar usuarios existentes.

El rol se recomienda modelarlo como un **enum** (por ejemplo, RolUsuario { ADMIN, JUGADOR }) o como una tabla de roles relacionada con la entidad Usuario.

7.3. Relaciones entre Usuario y Criatura

Cada usuario de tipo **JUGADOR** podrá tener asociadas una o varias criaturas que él mismo haya creado.

- Se deberá definir una relación entre Usuario y Criatura, por ejemplo:
  - Un **Usuario JUGADOR** puede tener **N criaturas**.
  - Cada **Criatura** pertenece a un **único usuario propietario**.
- Esta relación permitirá:
  - Cargar las criaturas de un jugador al iniciar sesión.
  - Restringir la edición/eliminación de criaturas a su propietario.

7.4. Funcionalidad de login y permisos

El flujo de la aplicación deberá ampliarse para incorporar el sistema de usuarios:

1. **Menú inicial** (antes de jugar):
   - Iniciar sesión.
   - Registrarse como nuevo usuario.
   - Salir.
2. **Registro de usuario**:
   - Pedirá como mínimo: username, email, password y tipo de rol inicial.

- Por defecto, se recomienda que los usuarios creados desde el menú público sean de tipo **JUGADOR**.
- La creación de **usuarios ADMIN** deberá limitarse (por ejemplo, solo desde base de datos o a través de un usuario administrador ya existente).

3. **Inicio de sesión**:
   - Se validará el username o email junto con la password.
   - Si las credenciales son correctas y el usuario está activo:
     - Se mostrará un menú según su rol:
       - **ADMINISTRADOR**: menú de gestión de usuarios (alta/baja/listado) y, opcionalmente, acceso a estadísticas generales.
       - **JUGADOR**: menú para crear criaturas, gestionar sus criaturas e iniciar el juego.
   - Si las credenciales son incorrectas o el usuario está inactivo:
     - Se mostrará un mensaje de error apropiado.
     - Se registrará el intento fallido en los **logs**.

7.5. DAO, DTO, servicios y excepciones para Usuario

- Deberá crearse el **UsuarioDAO** y su implementación correspondiente, siguiendo el mismo patrón DAO que el resto de entidades.
- Se deberá definir un **UsuarioDTO** que se utilice en:
  - La capa de presentación por consola.
  - La capa de servicios para realizar operaciones de login, alta de usuarios, etc.
- La capa de servicios deberá incluir métodos como:
  - registrarUsuario(UsuarioDTO usuarioDTO)
  - login(String usernameOrEmail, String password)
  - listarUsuarios(), activarUsuario(), desactivarUsuario(), etc. (al menos para el rol administrador).
- Cualquier problema de acceso a datos asociado a usuarios deberá lanzar una **DaoException** (o similar).

- Cualquier regla de negocio incumplida (por ejemplo, intento de crear un usuario con un email duplicado) deberá lanzar una **excepción de negocio** (BusinessException, UsuarioException, etc.).

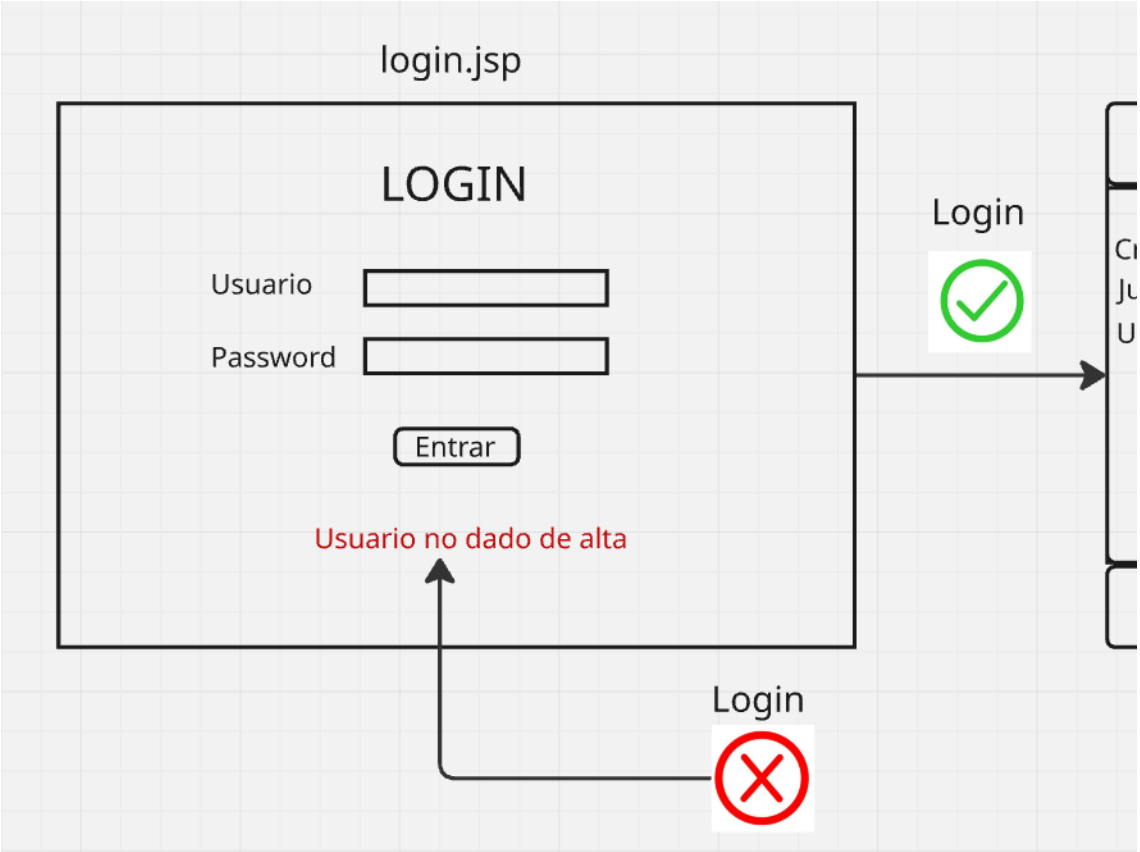
7.6. Logs relacionados con usuarios

El sistema de logs deberá registrar, como mínimo:

- Altas de usuarios (info).
- Cambios de estado (activar/desactivar) o de rol (info/warn).
- Intentos de inicio de sesión:
  - Éxito (info).
  - Fallo (warn o error, según el caso).
- Errores relacionados con la gestión de usuarios (error).

8. Parte web

8.1 Página de login



(login.jsp + servlet/controlador)

Objetivo

Implementar la pantalla de **inicio de sesión** para acceder a la parte privada de la aplicación. El login debe validar un usuario contra la **tabla de usuarios** existente, utilizando la arquitectura por capas (**DAO → Service → Web**), sin acceder a BD directamente desde JSP o Servlet.

1) Vista login.jsp

Crear la vista login.jsp con:

- Título o cabecera: **LOGIN**
- Formulario con:
    - Campo **Usuario** (name=usuario)
    - Campo **Password** (name=password, type=password)
    - Botón **Entrar**
- El formulario debe enviarse por **POST** al controlador/servlet de login (por ejemplo: /LoginServlet o /login).

Mensajes de error

En la misma página se deben mostrar mensajes según el resultado:

- Si el usuario **no existe**: mostrar en rojo Usuario no dado de alta
- Si la contraseña **no es correcta**: mostrar en rojo Contraseña incorrecta
- Si faltan campos: mostrar en rojo Debe introducir usuario y contraseña (**en el html se puede poner required los campos**)

Los mensajes deben mostrarse mediante un atributo (por ejemplo request.setAttribute("error", "...")) y pintarlos en login.jsp.

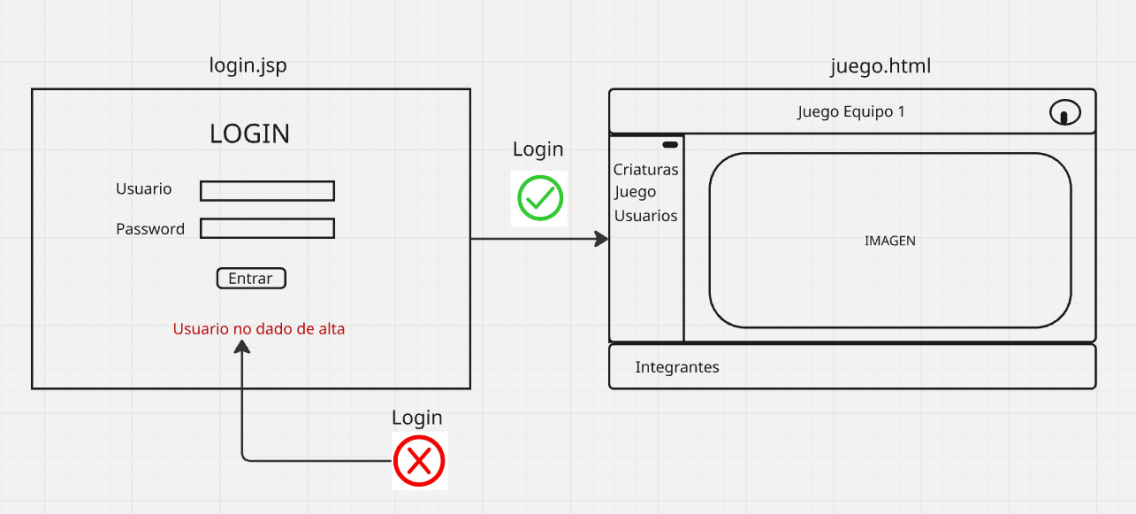
2) Controlador (Servlet) de Login

Crear un servlet que gestione el login:

doPost

1. Recoger usuario y password del formulario y cargarlo en DTO.
2. Llamar a la **capa de servicios** para comprobar credenciales pasándole el DTO.

3. Según el resultado:



   - **Login OK**:
     - Redirigir a la zona privada (juego.html).

   - **Login KO**:
     - Volver a login.jsp con el mensaje de error correspondiente

3) Capa de servicios (Service)

En la capa de servicios debéis implementar un método del estilo:

- login(UsuarioDTO usuario)

La lógica mínima en Service:

- Validar usuario y password correctos
- Buscar el usuario por nombre (a través del DAO).
- Si no existe → error “Usuario no dado de alta”.
- Si existe, pero password no coincide → error “Contraseña incorrecta”.
- Si coincide → devolver UsuarioDTO (sin exponer datos innecesarios).

4) DAO (Acceso a datos)

En DAO debéis disponer (o crear) operaciones como:

- findByUsuario(String usuario) (devuelve Entity o DTO según vuestro diseño)

- (Opcional) existsByUsuario(...)

El DAO se encarga únicamente de la consulta a la BD, sin lógica de navegación ni de sesión.

5) DTO / Entity

- **Entity** representa la tabla usuarios.
- **DTO** será el objeto que viajará entre capas.