1. En el método Crear de la clase JugadorService, ¿por qué se utiliza SCOPE_IDENTITY() en la consulta SQL y qué beneficio aporta al código?

Su función es mejorar la eficiencia del código al momento de insertar datos y recuperar el ID generado de una forma segura.

2. En el método Eliminar del servicio de jugadores, ¿por qué se verifica la existencia de elementos en el inventario antes de eliminar un jugador y qué problema está previniendo esta comprobación?

Esta verificación asegura que se mantenga la integridad de los datos y evita los problemas que se derivan de la base de datos

3. Qué ventaja ofrece la línea using var connection = _dbManager.GetConnection(); frente a crear y cerrar la conexión manualmente? Menciona un posible problema que podría ocurrir si no se usara esta estructura?

Permite la gestión automática de los recursos y un posible problema al no utilizarlo es una sobrecarga del servidor de base de datos.

4. En la clase DatabaseManager, ¿por qué la variable _connectionString está marcada como readonly y qué implicaciones tendría para la seguridad si no tuviera este modificador?

Permite proteger información sensible y en caso de no utilizar la variable se pueden producir riesgos de seguridad y problemas de mantenimiento.

5. Si quisieras agregar un sistema de logros para los jugadores, ¿qué cambios realizarías en el modelo de datos actual y qué nuevos métodos deberías implementar en los servicios existentes?

Para agregar logros, haría lo siguiente:

Modelo de datos: Crear una tabla Logros con columnas como Id, Nombre, Descripcion, y FechaCreacion. También una tabla intermedia JugadorLogros para relacionar jugadores con sus logros.

Métodos en el servicio:

AgregarLogro(int jugadorId, int logroId)

ObtenerLogrosPorJugador(int jugadorId)

EliminarLogro(int jugadorId, int logroId)

6. Qué sucede con la conexión a la base de datos cuando ocurre una excepción dentro de un bloque using como el que se utiliza en los métodos del JugadorService?

Cuando ocurre una excepción dentro de un bloque **using**, la conexión se cierra automáticamente al salir del bloque, gracias a que **SqlConnection** implementa **IDisposable**. Esto evita fugas de recursos, incluso si algo falla.

7. En el método ObtenerTodos() del JugadorService, ¿qué ocurre si la consulta SQL no devuelve ningún jugador? ¿Devuelve null o una lista vacía? ¿Por qué crees que se diseñó de esta manera?

Si no hay jugadores, devuelve una lista vacía (**new List<Jugador>()**). Esto es mejor que devolver **null** porque evita errores al intentar iterar sobre la lista. Se diseño de esa manera porque es más seguro y práctico para el consumidor del método.

8. Si necesitaras implementar una funcionalidad para registrar el tiempo jugado por cada jugador, ¿qué cambios harías en la clase Jugador y cómo modificarías los métodos del servicio para mantener actualizada esta información?

Cambios en Jugador: Agregar una propiedad **TiempoJugado** (por ejemplo, en minutos).

Cambios en el servicio: Actualizar el tiempo jugado en métodos como Actualizar o crear uno nuevo como **RegistrarTiempo(int jugadorId, int minutos)**.

9. En el método TestConnection() de la clase DatabaseManager, ¿qué propósito cumple el bloque try-catch y por qué es importante devolver un valor booleano en lugar de simplemente lanzar la excepción?

El **try-catch** permite capturar errores al probar la conexión y devolver **false** en lugar de lanzar una excepción. Esto es útil para manejar el estado de la conexión de forma más controlada y evitar que el programa se detenga.

10. Si observas el patrón de diseño utilizado en este proyecto, ¿por qué crees que se separaron las clases en carpetas como Models, Services y Utils? ¿Qué ventajas ofrece esta estructura para el mantenimiento y evolución del proyecto?

Separar en **Models**, **Services** y **Utils** organiza el código por responsabilidad. Esto facilita el mantenimiento, la lectura y la escalabilidad del proyecto.

11. En la clase InventarioService, cuando se llama el método AgregarItem, ¿por qué es necesario usar una transacción SQL? ¿Qué problemas podría causar si no se implementara una transacción en este caso?

Las transacciones aseguran que todas las operaciones relacionadas (como agregar un ítem y actualizar el inventario) se completen juntas o no se realicen en absoluto. Sin transacciones puede terminar con datos inconsistentes si algo falla a mitad del proceso.

12. Observa el constructor de JugadorService: ¿Por qué recibe un DatabaseManager como parámetro en lugar de crearlo internamente? ¿Qué patrón de diseño se está aplicando y qué ventajas proporciona?

Recibir DatabaseManager como parámetro aplica el patrón **Inversión de Dependencias**. Esto facilita pruebas unitarias y hace que el código sea más flexible y desacoplado.

13. En el método ObtenerPorId de JugadorService, ¿qué ocurre cuando se busca un ID que no existe en la base de datos? ¿Cuál podría ser una forma alternativa de manejar esta situación?

Si el ID no existe, devuelve **null**. Una alternativa sería lanzar una excepción personalizada como **JugadorNoEncontradoException** o devolver un objeto con un estado que indique que no se encontró.

14. Si necesitas implementar un sistema de "amigos" donde los jugadores puedan conectarse entre sí, ¿cómo modificarías el modelo de datos y qué nuevos métodos agregarías a los servicios existentes?

Modelo de datos: Crear una tabla **Amigos** con columnas JugadorId y **AmigoId**.

Métodos en el servicio:

AgregarAmigo(int jugadorId, int amigoId)

ObtenerAmigos(int jugadorId)

EliminarAmigo(int jugadorId, int amigoId)

15. En la implementación actual del proyecto, ¿cómo se maneja la fecha de creación de un jugador? ¿Se establece desde el código o se delega esta responsabilidad a la base de datos? ¿Cuáles son las ventajas del enfoque utilizado?

La fecha de creación parece venir de la base de datos (FechaCreacion). Esto es bueno porque asegura que sea consistente y precisa, sin depender del reloj del servidor de la aplicación.

16. ¿Por qué en el método GetConnection() de DatabaseManager se crea una nueva instancia de SqlConnection cada vez en lugar de reutilizar una conexión existente? ¿Qué implicaciones tendría para el rendimiento y la concurrencia?

Cada vez que se llama a **GetConnection()**, se crea una nueva conexión. Esto es necesario porque las conexiones no son seguras para múltiples hilos. Reutilizar conexiones podría causar problemas de concurrencia.

17. Cuando se actualiza un recurso en el inventario, ¿qué ocurriría si dos usuarios intentan modificar el mismo recurso simultáneamente? ¿Cómo podrías mejorar el código para manejar este escenario?

Si dos usuarios modifican el mismo recurso al mismo tiempo, podría haber conflictos. Para evitarlo, usaría bloqueos optimistas (con un campo **Version**) o bloqueos pesimistas (transacciones con **SELECT ... FOR UPDATE**).

18. En el método Actualizar de JugadorService, ¿por qué es importante verificar el valor de rowsAffected después de ejecutar la consulta? ¿Qué información adicional proporciona al usuario?

Sirve para confirmar si la operación afectó alguna fila. Si no, se puede informar al usuario que el jugador no existe o que no se realizaron cambios.

19. Si quisieras implementar un sistema de registro (logging) para seguir todas las operaciones realizadas en la base de datos, ¿dónde colocarías este código y cómo lo implementarías para afectar mínimamente la estructura actual?

Colocaría el código de logging en un middleware o en un decorador alrededor de los métodos del servicio. Usaría una librería como **Serilog** para registrar las operaciones en un archivo o base de datos.

20. Observa cómo se maneja la relación entre jugadores e inventario en el proyecto. Si necesitaras agregar una nueva entidad "Mundo" donde cada jugador puede existir en múltiples mundos, ¿cómo modificarías el esquema de la base de datos y la estructura del código para implementar esta funcionalidad?

Modelo de datos: Crear una tabla **Mundos** y una tabla intermedia

JugadorMundos: para relacionar jugadores con mundos.

Métodos en el servicio:

AgregarJugadorAMundo(int jugadorId, int mundoId)

ObtenerMundosPorJugador(int jugadorId)

21. Qué es un SqlConnection y cómo se usa?

Es una clase que representa una conexión a una base de datos SQL Server. Se usa para ejecutar comandos y consultas.

22. Para qué sirven los SqlParameter?

Se usan para evitar inyecciones SQL al pasar valores a las consultas de forma seguray mejora el rendimiento al permitir que el servidor reutilice planes de ejecución.