Ejercicio planteado

1. Crear una aplicación integradora usando:
    a. Frontend: Angular + HTML
    b. Backend: Java + Spring Boot + Hibernate
    c. Base de datos: a elección (Ej: Postgresql)

Esta aplicación deberá contener una tabla de base de datos (ej Persona, que modele a una persona, 3 o 4 columnas). El backend deberá exponer un servicio rest que liste todas las personas de la base de datos. El frontend deberá consumir el servicio rest y mostrar los datos en una tabla. Tratar de usar una arquitectura en capas.

Plus 1: Si lo anterior sale fácil sería bueno agregar a la pantalla 2 filtros (ej: Nombre, Apellido) y que la api rest soporte esos filtros de modo que esos filtros lleguen al backend y se deleguen a la BD para filtrar los datos en usando el motor de base de datos. Pensar que diferencia hay entre eso y hacer el filtro en frontend sin resolverlo a nivel BD.
Plus 2: Formulario para alta de Personas con validaciones de campos
Plus 3: Entender si esta usando Inyección de dependencias en el backend. Si es asi explicar quien se encarga de esto y que es la inyección de dependencias.

2. Determinar que patrones de diseño podría usar y porque para cada capa.
3: Conceptos:
    Que es una Single Page Application, en donde se ejecuta?, como se comunica con el backend?
    Que es un cliente de bases de datos? Ejemplos
    Que es un motor de base de datos? Ejemplos?
    Que es un driver de conexión de base de datos? Ejemplos en Java

Respuestas

Filtros

Backend vs frontend

La ventaja de hacer los filtros en el backend es principalmente la mejor performance ya que tiene que enviar menos datos al frontend. Si tuviésemos muchas personas y sólo queremos mostrar alguna con determinado ID, es más performante aplicar el filtro en el backend y que al cliente sólo se envíe ese resultado a mostrar; en lugar de enviar todos y después filtrar en el frontend.

Una opción frente a esto podría ser también cachear cierta información para evitar ciertos problemas de performance pero ahí habría que manejar los riesgos de que la información pueda quedar desactualizada.

Validaciones de campos

Se realizaron tanto del lado del frontend como del backend. Al hacerlas en el frontend limité los ingresos erróneos para que no tengan que todos llegar al backend si pueden validarse antes. De todas formas, del lado del backend también se realizaron al recibir los datos y antes de persistirlos.

Patrones de diseño

Inyección de dependencias

Sí, se está usando inyección de dependencias en el Backend. En particular se lo utiliza en el servicio JugadoresServ para *inyectar* la implementación del repositorio jugadoresRepo. Quien se encarga de esta inyección de la dependencia es Spring y se le indica por medio de @Autowired y @Service. Mediante el uso de estos decoradores se inyecta la dependencia en el constructor del servicio.

La Inyección de dependencias es un patrón utilizado para parametrizar aquello de lo que se depende. Es decir, dejar que las dependencias sean inyectadas (por un tercero) en el componente que las necesita. Se utiliza para evitar el acoplamiento a otros componentes.

Utilizados capa por capa

Capa de presentación

Cliente-servidor entre la computadora que se conecta y la aplicación

Inyección de dependencias (del service en el component)

MVC

Capa lógica de negocio

Inyección de dependencias (del repo en el service). Para desacoplarse de qué implementación se está usando en el repositorio, se inyectó la implementación del repo en el serv. Esto se hizo con Spring con @Autowired y @Service.

Arquitectura en capas

Singleton en el uso de @bean para las configuraciones y en cada clase que maneja spring que lo hace como una sola instancia. Por ejemplo el serv del que se habló anteriormente es manejado por spring como única instancia.

Posibles futuros patrones al complejizar la lógica de negocio:

- **Patron state** = Sumar otro campo que sea categoría a cada persona, la cual puede ser una clase que tenga un método ascender() para cambiar a otra categoría.

- **Template method** = Refactorizar la posición como una clase con el método jugar(). Donde, para cada posición, jugar implica diferentes pasos y, por tal motivo, tengan que tener otros métodos como entradaEnCalor(), partido() y posibilidadDeLesiones().

Capa Persistencia

Cliente-servidor entre la aplicación y la base de datos

Hibernate cumple con el standard de JPA. Se está utilizando JPA para no quedar acoplado a Hibernate, el día de mañana podría cambiarse por otro. JPA es una api para consultar a los datos que nos provee directamente Java. Hibernate cumple con el estándar de JPA. A bajo nivel, Hibernate implementa JPA como si fuese un **adapter** y Hibernate haría uso de todo eso eventualmente.

Single Page Application

Una SPA es una página donde todo el contenido está dentro de un solo archivo. Es decir, sólo se carga un archivo html y todo el contenido esté ahí. Esto hace que sea más rápido porque todo se carga de una vez. Si los contenidos son mostrados dinámicamente, sólo cambia ese contenido, no toda la página.

Dentro de una sola página se pueden tener diferentes vistas pero en cada una estamos en la misma página, sólo se sustituye su contenido. Esto a su vez crea una página reactiva y con una mejor experiencia de usuario en comparación con una SSR (Server Side Rendering).

Estas SPA se ejecutan del lado del cliente, en el navegador web y ahí sólo podemos utilizar Javascipt html y css. Ahí Angular es un framework que ayuda a crear esas SPA.

Las SPA requieren que nuestro backend tenga una API que les proporcione el contenido para mostrar y no le importa cómo ese fue hecho. Esto además permite desacoplar el frontend del backend.

Base de datos

Cliente

El cliente de base de datos es un programa que me permite interactuar con la base de datos. Mediante el uso de un cliente de base de datos, se puede acceder a la información que se guarda en las diferentes bases de datos.

Motor

El motor o servidor de base de datos es quien realmente a bajo nivel está persistiendo las cosas

Ejemplo

- Servidor de Base de datos relacional: mysql server o mariadb
- Cliente sql: mysql workbench para poder ver las tablas y demás.

Driver

Un driver es un programa que funciona como adaptador para conectar una interfaz con una base de datos particular. Este driver es el que nos permite comunicarnos con el motor de base de datos, internamente estos programas, implementan los protocolos de comunicación necesarios para ejecutar operaciones sobre la base de datos. Estos drivers a su vez exponen una API con la que nos podemos comunicar con la base de datos.

Java drivers

Para conectar con una base de datos a una aplicación de Java, necesitamos un JDBC (Java Database Connectivity API) driver específico para la base de datos a la que se quiera conectar. Este permite, entre otras cosas, conectar programas escritos en Java con los datos de las bases de datos, en conjunto con el driver adecuado para esa fuente de datos.

Es entonces una API para Java que define cómo un cliente podrá acceder a los datos (especialmente a bases de datos relacionales). Es parte del estandar de Java y actúa como una capa intermedia entre la aplicación y la base de datos.

Específicamente permite que la aplicación pueda:

- Estableces una conexión con la fuente de datos
- Enviar queries y modificar estados de la fuente de datos
- Procesar los resultados

Los JDBC drivers son adapters del lado del cliente que convierten las peticiones de los programas de java a un protocolo que el motor de base de datos pueda entender. Hay cuatro tipos de drivers de JDBC.