Preguntero Técnico - TaskFlow API

ÍNDICE

TEÓRICAS SIMPLES

1 ¿Qué es una API REST?

Una API REST expone recursos a través de HTTP usando métodos como GET, POST, PUT, PATCH y DELETE, y responde con formatos como JSON. Sigue el modelo cliente-servidor y es stateless.

2 ¿Qué hace Spring Boot?

Simplifica la creación de aplicaciones Spring al autoconfigurar componentes y permitir arrancar con poca configuración. Incluye servidor embebido y facilita el desarrollo rápido.

3 ¿Qué es JPA?

Una especificación para mapear objetos Java a tablas de base de datos y manejar persistencia con anotaciones. Define un estándar para ORM.

4 ¿Qué es Hibernate?

Es la implementación más común de JPA que gestiona el mapeo ORM y las operaciones SQL automáticamente. Abstrae los detalles de la BD.

5 ¿Qué significa ORM?

Object-Relational Mapping: mapeo entre clases Java y tablas SQL. Permite trabajar con objetos en lugar de escribir SQL directamente.

6 ¿Qué es una entidad en JPA?

Una clase anotada con `@Entity` que representa una tabla de base de datos. Está gestionada por el persistence context de JPA.

7 ¿Qué hace `@Id` y `@GeneratedValue`?

`@Id` marca la clave primaria de la entidad; `@GeneratedValue` indica que se auto-genera (por ejemplo, con autoincremental).

8 ¿Qué diferencia hay entre PUT y PATCH?

PUT reemplaza el recurso completo; PATCH actualiza parcialmente uno o más campos. PUT requiere enviar todo; PATCH solo lo que cambia.

9 ¿Qué es un DTO y para qué sirve?

Un DTO (Data Transfer Object) es un objeto para transportar datos entre capas o cliente-servidor, evitando exponer entidades directamente.

10 ¿Qué es BCrypt y por qué se usa?

Un algoritmo de hashing para contraseñas, seguro porque incluye salt y es lento a propósito, dificultando ataques de fuerza bruta.

TEÓRICAS MEDIAS

1 ¿Qué es Inversión de Control (IoC) en Spring?

Es el principio donde el framework crea y gestiona objetos (beans) y los inyecta donde se necesitan, en lugar de que el código maneje instancias.

2 ¿Qué es la Inyección de Dependencias (DI)?

Es el mecanismo de pasar dependencias a una clase en lugar de que la clase las cree por sí misma, mejorando desacoplamiento y testabilidad.

3 ¿Qué es un Bean en Spring?

Un objeto gestionado por el contenedor de Spring, creado y configurado por él. Se registra con anotaciones como `@Component`, `@Service`.

4 ¿Qué diferencia hay entre `@Component`, `@Service` y `@Repository`?

Todas registran beans, pero `@Service` y `@Repository` agregan semántica específica (lógica de negocio y acceso a datos) y manejo especial de excepciones.

5 ¿Qué es una transacción en JPA?

Un conjunto de operaciones que se ejecutan como una unidad atómica: o se completan todas o ninguna. Garantiza consistencia.

6 ¿Para qué sirve `@Transactional`?

Marca un método/clase para que Spring gestione transacciones automáticamente, abriendo, commiteando y rollbackeando según corresponda.

7 ¿Qué es la paginación en Spring Data?

Es la forma de obtener resultados en páginas usando `Pageable` y `Page`. Mejora performance y UX al dividir grandes datasets.

8 ¿Qué es un `Pageable` y un `Page`?

`Pageable` define la página solicitada, tamaño y orden; `Page` contiene los resultados y metadatos (total de elementos, páginas).

9 ¿Qué es una excepción controlada vs no controlada?

Controlada (`checked`): heredan de `Exception` y deben manejarse o declararse. No controlada: `RuntimeException`, se propagan automáticamente.

10 ¿Qué es una capa de servicio y por qué es importante?

Contiene la lógica de negocio y coordina repositorios. Separa controllers de repositorios, mejorando mantenibilidad y testabilidad.

TEÓRICAS COMPLEJAS

1 ¿Qué es el patrón Unit of Work en JPA?

JPA agrupa cambios en el `Persistence Context` y los sincroniza con la BD al commit, evitando múltiples writes y garantizando consistencia.

2 ¿Qué es el `Persistence Context`?

Es el primer nivel de caché de JPA donde se gestionan las entidades durante una transacción. Rastrea cambios y genera SQL automáticamente.

3 ¿Qué diferencia hay entre `merge`, `persist`, `detach` y `remove`?

- `persist`: crea una entidad nueva bajo gestión - `merge`: sincroniza un objeto fuera de contexto - `detach`: saca del contexto sin persistir cambios - `remove`: marca para borrar al commit

4 ¿Qué implica Lazy Loading y cuáles son sus riesgos?

Carga relaciones bajo demanda en lugar de eagerly. Riesgo: `LazyInitializationException` si se accede a relaciones fuera de la sesión.

5 ¿Qué es N+1 Problem y cómo mitigarlo?

Se generan muchas queries pequeñas (N+1) al acceder relaciones. Se mitiga con `fetch join`, `EntityGraph` o batch fetching.

6 ¿Qué es el principio de idempotencia en HTTP?

Una operación idempotente puede ejecutarse varias veces con el mismo resultado (GET, PUT, DELETE son idempotentes; POST no).

7 ¿Qué es el patrón Repository y cuál es su objetivo?

Abstrae el acceso a datos, desacoplando lógica de negocio de la persistencia. Permite cambiar BD sin afectar el resto del código.

8 ¿Qué diferencia hay entre `@OneToMany` y `@ManyToMany` en términos de diseño?

`@OneToMany` implica un dueño único (foreign key en tabla secundaria); `@ManyToMany` crea tabla intermedia y complica el modelo significativamente.

9 ¿Qué es el problema de concurrencia optimista y cómo se resuelve?

Conflictos cuando múltiples transacciones actualizan el mismo recurso. Se resuelve con `@Version` y control de versiones (OptimisticLockException).

10 ¿Por qué es buena práctica no exponer entidades directamente en el API?

Evita acoplamiento, fuga de datos sensibles, permite versionar el contrato sin cambiar la BD, y facilita transformaciones.

DISEÑO SIMPLES

1 ¿Qué es separación de capas y por qué se usa?

Dividir la aplicación en controller, service y repository. Mejora mantenibilidad, testabilidad y permite cambios independientes en cada capa.

2 ¿Qué responsabilidad tiene el Controller?

Recibir HTTP, mapear requests a DTOs, validación básica y delegar lógica al service. No debe contener lógica de negocio.

3 ¿Qué responsabilidad tiene el Service?

Contener la lógica de negocio, coordinar repositorios, realizar validaciones y coordinar transacciones.

4 ¿Qué responsabilidad tiene el Repository?

Acceder a la base de datos y encapsular consultas. Abstrae detalles de JPA y SQL del resto de la aplicación.

5 ¿Por qué usar DTOs en vez de entidades?

Para controlar el contrato de la API y evitar exponer campos internos o relaciones que no deben ser públicas.

6 ¿Qué es un contrato de API?

El formato y reglas de request/response que el cliente espera. Debe ser estable y versionable.

7 ¿Qué es un endpoint REST?

Una URL que representa un recurso y permite operar sobre él (GET, POST, PUT, PATCH, DELETE).

8 ¿Por qué es importante usar códigos HTTP correctos?

Mejora la semántica, facilita integración con clientes y herramientas de debugging. Permite diferenciar entre 200, 201, 204, 400, 404, 409, 500.

9 ¿Qué significa "resource-oriented design"?

Diseñar la API alrededor de recursos (`/tasks`, `/users`) y sus acciones, no alrededor de verbos de acciones.

10 ¿Qué es un CRUD?

Create, Read, Update, Delete: cuatro operaciones básicas sobre un recurso, correspondientes a POST, GET, PUT/PATCH, DELETE.

DISEÑO MEDIAS

1 ¿Cómo decidir entre PUT y PATCH en diseño de API?

PUT reemplaza el recurso completo; PATCH actualiza campos específicos. Usa PATCH si solo cambias 1-2 campos; PUT si cambias todo.

2 ¿Cuándo conviene crear un endpoint de stats separado?

Cuando el dashboard necesita agregaciones (conteos, promedios) y resulta ineficiente/complejo en el endpoint de listado.

3 ¿Cómo diseñar filtros en endpoints de listado?

Usando query params opcionales (`status`, `priority`, `assigneeId`) y combinaciones controladas. Validar valores permitidos en el service.

4 ¿Por qué incluir paginación en listados?

Evita respuestas gigantes, mejora rendimiento al limitar queries, y es mejor UX.

5 ¿Cómo manejar validaciones en la arquitectura?

Validaciones de formato en DTOs (JSR-380) y reglas de negocio complejas en el service.

6 ¿Qué significa "single responsibility" en capas?

Cada capa tiene un rol claro y único: controller maneja HTTP, service lógica, repository datos. Facilita testing y reuso.

7 ¿Por qué usar un GlobalExceptionHandler?

Centraliza manejo de errores y asegura respuestas consistentes. Evita duplicar `try-catch` en todo el código.

8 ¿Cómo justificar el uso de enums para `status` y `priority`?

Asegura valores válidos, reduce errores, facilita queries y es más eficiente que strings en la BD.

9 ¿Qué es el "contract-first thinking"?

Diseñar primero el contrato de API (DTOs, endpoints) y luego implementar lógica. Alinea equipo frontend-backend desde el inicio.

10 ¿Qué riesgo hay si el controller maneja lógica de negocio?

Se vuelve difícil de mantener, testear y reutilizar. Viola SRP y complica cambios futuros.

DISEÑO COMPLEJAS

1 ¿Cómo diseñarías la API para soportar multi-tenant?

Agregar `tenant_id` a entidades y filtrar automáticamente por tenant en repositorios o con filtros globales de Hibernate.

2 ¿Cómo evitar "chatty APIs" en listados complejos?

Incluyendo campos necesarios en una sola respuesta, usando endpoints agregados, proyecciones o GraphQL si aplica.

3 ¿Qué estrategia usarías para versionar la API?

Versionado en URL (`/v1`, `/v2`) o por header. Mantener compatibilidad hacia atrás en cambios no-breaking.

4 ¿Cómo diseñarías un sistema de permisos por roles?

RBAC (Role-Based Access Control) con roles y permisos por endpoint. Validar en filtros de seguridad o `@PreAuthorize`.

5 ¿Qué harías si las estadísticas se vuelven costosas?

Pre-agregar datos en tablas dedicadas, usar caché con TTL, o actualizarlas via eventos asincronos.

6 ¿Cómo manejarías auditoría y trazabilidad?

Agregar campos auditables (`created_by`, `updated_by`, `created_at`, `updated_at`) y eventos de dominio para cambios importantes.

7 ¿Qué problema resuelve la paginación basada en cursor?

Evita inconsistencias cuando se insertan/borran filas entre requests y mejora performance en datasets masivos.

8 ¿Cómo diseñarías un "bulk update" de tareas?

Endpoint específico (`POST /tasks/bulk-update`) con validación y transacción controlada. Retornar detalle de éxitos/fallos.

9 ¿Qué es un "bounded context" y por qué importa?

Separar dominios de negocio en límites claros para reducir acoplamiento, facilitar escalado y claridad de modelos.

10 ¿Cuándo pasarías a arquitectura de microservicios?

Cuando hay equipos independientes, límites claros de dominio, necesidad de escalado separado y tolerancia a latencia de red.

TÉCNICAS SIMPLES

1 ¿Qué hace `@RestController`?

Marca la clase como controller REST y retorna JSON por defecto (implica `@Controller` + `@ResponseBody`).

2 ¿Qué diferencia hay entre `@RequestParam` y `@PathVariable`?

`@RequestParam` toma parámetros de query (`?status=DONE`); `@PathVariable` toma valores de la URL (`/tasks/123`).

3 ¿Qué hace `@RequestBody`?

Convierte el JSON del request en un objeto Java automáticamente (deserialización).

4 ¿Qué hace `@Valid`?

Ejecuta validaciones definidas en el DTO (anotaciones como `@NotNull`, `@Email`, etc.).

5 ¿Qué es Lombok y qué hace `@Data`?

Lombok es una librería que genera código boilerplate. `@Data` genera getters, setters, `equals`, `hashCode`, `toString`.

6 ¿Para qué sirve `ResponseEntity`?

Permite definir el cuerpo y el status HTTP explícitamente en la respuesta.

7 ¿Qué hace `@GetMapping`?

Mapea un endpoint HTTP GET a un método del controller. Shortcut para `@RequestMapping(method = RequestMethod.GET)`.

8 ¿Qué hace `@Autowired`?

Inyecta automáticamente una dependencia desde el contenedor de Spring.

9 ¿Qué es un `PageRequest`?

Un objeto que define página, tamaño y orden para paginación. Se pasa como `Pageable` al repository.

10 ¿Qué devuelve un método del repository que retorna `Optional`?

Un contenedor que puede tener valor o estar vacío. Evita `null` y obliga a manejar el caso "no encontrado".

TÉCNICAS MEDIAS

1 ¿Cómo funciona la validación de DTOs en Spring?

Con anotaciones JSR-380 en los DTOs y `@Valid` en el controller. Los errores se capturan por el `GlobalExceptionHandler`.

2 ¿Qué es `@Enumerated(EnumType.STRING)` y por qué usarlo?

Guarda el nombre del enum en la BD en lugar del ordinal. Evita problemas si cambia el orden de los enums.

3 ¿Qué diferencia hay entre `findById` y `getById` en JPA?

`findById` retorna `Optional`; `getById` retorna proxy lazy-loadable y lanza `EntityNotFoundException` si no existe.

4 ¿Qué es `@JsonIgnore` y cuándo usarlo?

Evita serializar un campo en JSON. Útil para evitar ciclos, datos sensibles o campos internos.

5 ¿Qué es el DTO `UpdateTaskStatusRequest` y por qué separarlo?

Permite actualización parcial específica del estado sin mezclar/requerir otros campos. Define contrato claro.

6 ¿Cómo implementarías búsqueda por título?

Usando un método repository `findByTitleContainingIgnoreCase(String search)` or `findByTitleLikeIgnoreCase`.

7 ¿Cómo se manejan los errores de validación?

Se captura `MethodArgumentNotValidException` en el handler global y se devuelve un `ErrorResponse` con lista de errores de campos.

8 ¿Qué hace `@Column(nullable = false)`?

Genera restricción NOT NULL en la columna. Valida a nivel BD y JPA.

9 ¿Qué es `@Builder` en Lombok y cuándo conviene?

Genera un builder pattern para construcción fluida de objetos. Útil en DTOs, tests y cuando el constructor tiene muchos parámetros.

10 ¿Qué pasa si se usa `cascade` mal en relaciones?

Se pueden eliminar o persistir entidades no deseadas automáticamente, causando data corruption o errores inesperados.

TÉCNICAS COMPLEJAS

1 ¿Cómo controlarías el rendimiento de un endpoint con múltiples filtros?

Usando índices adecuados en BD, combinando filtros en query derivada o `Specification`, y paginación obligatoria.

2 ¿Qué es `EntityGraph` y cuándo lo usarías?

Define qué relaciones cargar de forma eager en una consulta específica para evitar N+1 sin hardcodear fetch joins.

3 ¿Qué es el "dirty checking" en Hibernate?

Hibernate detecta cambios en entidades gestionadas comparándolas y genera SQL automáticamente al commit.

4 ¿Cómo evitarías un `LazyInitializationException`?

Accediendo a relaciones dentro de la transacción o usando `fetch join`/`EntityGraph` para cargar eagerly.

5 ¿Qué estrategia usarías para soft deletes?

Agregar campo `deleted_at` o `is_deleted` y filtrar en todas las queries o usar `@Where(clause = "deleted_at IS NULL")`.

6 ¿Cómo manejarías concurrencia en actualizaciones de tareas?

Con `@Version` para optimistic locking. Si hay conflicto, se lanza `OptimisticLockException` y el cliente reintenta.

7 ¿Qué implica usar proyecciones en Spring Data?

Traer solo los campos necesarios, reduciendo payload de respuesta y costo de consulta en la BD.

8 ¿Cómo manejarías validaciones cross-field en DTOs?

Con validadores custom (`@Constraint`) que implementan `ConstraintValidator` y acceden múltiples fields.

9 ¿Cómo diseñarías un rate limit para login?

Con filtros o interceptores, usando Redis o bucket en memoria por IP/usuario. Retornar 429 Too Many Requests.

10 ¿Qué ventaja tiene usar `Specification` sobre métodos derivados?

Permite combinar filtros dinámicamente sin explotar el número de métodos. Escala bien con múltiples filtros opcionales.

CUESTIONES DE CÓDIGO SIMPLES

1 ¿Por qué `@RequestBody` en el login?

Porque los datos (email, password) vienen en JSON en el body y deben mapearse al DTO `LoginRequest`.

2 ¿Por qué usar `Optional` al buscar por ID?

Evita `NullPointerException` y obliga a manejar explícitamente el caso "no encontrado" con `.orElseThrow()` o `.ifPresent()`.

3 ¿Por qué `ResponseEntity` en los controllers?

Permite devolver status HTTP correctos (201, 204, 404) junto con el cuerpo, en lugar de solo retornar el objeto.

4 ¿Qué pasa si no uso `@Valid` en un DTO?

Las validaciones anotadas (`@NotNull`, `@Email`) no se ejecutan y entran datos inválidos sin error.

5 ¿Por qué separar `CreateTaskRequest` de `UpdateTaskRequest`?

Porque los campos requeridos pueden ser distintos. En creación, `status` es opcional; en actualización, puede serlo también.

6 ¿Por qué `@Enumerated(EnumType.STRING)` en status y priority?

Evita errores cuando cambias el orden de los enums o agregas valores nuevos en el medio.

7 ¿Qué hace `@JsonFormat` en fechas?

Define el formato de serialización/deserialización. Ejemplo: `@JsonFormat(pattern = "yyyy-MM-dd")` para que cliente envíe `"2025-03-15"`.

8 ¿Por qué `@Column(unique = true)` en email?

Evita duplicados a nivel BD y agrega una segunda línea de defensa además de la validación lógica en el service.

9 ¿Por qué devolver 201 en POST?

Porque se creó un recurso nuevo. Es más semánticamente correcta que 200 y comunica al cliente que hubo creación.

10 ¿Qué ocurre si no manejo `ResourceNotFoundException`?

El cliente recibe un 500 Internal Server Error genérico en lugar de 404, confundiendo si fue error del servidor o dato no encontrado.

CUESTIONES DE CÓDIGO MEDIAS

1 ¿Por qué en `updateTaskStatus` solo aceptamos un DTO específico?

Para limitar el cambio únicamente al estado y evitar que se actualicen accidentalmente otros campos como `title` o `priority`.

2 ¿Qué ventaja tiene usar `Page` en lugar de `List`?

`Page` incluye metadatos: total de elementos, número de páginas, página actual. Útil para UI que necesita info de paginación.

3 ¿Por qué se hace `taskRepository.existsByEmail` antes de crear usuario?

Para validar unicidad tempranamente y retornar 409 Conflict antes de que intente insertar y falle a nivel BD.

4 ¿Por qué el endpoint de stats está separado?

Porque devuelve agregaciones (conteos por estado/prioridad), no tareas individuales. Se optimiza distinto y lógicamente es diferente.

5 ¿Por qué usamos BCrypt y no MD5/SHA?

BCrypt es lento por diseño e incluye salt automático, resistiendo ataques de fuerza bruta. MD5/SHA son rápidos y vulnerables.

6 ¿Por qué `@ManyToOne` en Task→User?

Una tarea tiene un solo usuario asignado, pero un usuario puede tener muchas tareas. Es el lado "many" de la relación.

7 ¿Qué problema evita `@JsonIgnore` en relaciones bidireccionales?

Evita ciclos infinitos al serializar JSON. Si Task tiene User y User tiene List, circula infinito sin marcas de ignore.

8 ¿Por qué en `deleteTask` devolvemos 204?

Porque no hay contenido en la respuesta y 204 No Content es el estándar para operaciones exitosas sin body.

9 ¿Qué riesgo hay si no validamos `status` o `priority`?

La BD puede recibir valores inválidos (strings aleatorios en lugar de enum) y romper la integridad del dominio.

10 ¿Por qué el service centraliza la lógica de filtros?

Para mantener el controller limpio y permitir reutilización de filtros en múltiples endpoints o contextos.

CUESTIONES DE CÓDIGO COMPLEJAS

1 ¿Cómo evitarías que la lógica de filtros escale mal a medida que crecen los parámetros?

Usando `Specification` o Criteria API para construir queries dinámicas y combinables sin explotar el número de métodos.

2 ¿Cómo resolverías un N+1 si agregas relación Task→Comments?

Con `fetch join` en JPQL, `EntityGraph`, batch fetching en Hibernate o lazy collections con batch size.

3 ¿Cómo manejarías cambios concurrentes en una tarea?

Con `@Version` (optimistic locking). Si dos usuarios editan simultáneamente, uno recibe `OptimisticLockException` y reintenta.

4 ¿Cómo desacoplarías el mapeo Entity→DTO?

Usando mappers dedicados o librerías como MapStruct para evitar lógica de mapeo duplicada o en controladores.

5 ¿Cómo garantizarías consistencia en un update parcial?

Validando reglas de negocio en el service y usando DTOs específicos por operación para definir qué campos son permitidos.

6 ¿Cómo optimizarías el endpoint `/tasks/stats` si crece la data?

Con queries agregadas directas en SQL o HQL, índices adecuados y, si aplica, caché con TTL o pre-agregaciones en tablas.

7 ¿Qué harías si el `Page` es muy costoso en conteo total?

Usar `Slice` que no cuenta total, o paginación por cursor para evitar `COUNT(*)` costoso en tablas gigantes.

8 ¿Cómo manejarías validaciones dependientes del estado actual?

Ejemplo: no permitir DONE si falta assignee. Se valida en el service con lógica contextual que consulta estado actual.

9 ¿Qué harías si el frontend pide orden por múltiples campos?

Permitir `Sort.by` con varios criterios (`order_by=created_at,desc■_by=priority,asc`), validando que campos existan para evitar inyección.

10 ¿Cómo manejarías logs y trazas para debug de requests?

Agregar logging estructurado en service/controller, generar request ID único y correlacionar en filtros/interceptores para rastrear flujo.

NOTAS FINALES