Sección 1: Conceptos Generales

1. **¿Qué es una API y cuál es su propósito principal?**

Una api es un conjunto de definiciones y protocolos que permiten conectar dos sistemas entre sí. El propósito principal de una api es establecer los servicios necesarios para una aplicación de forma segura y eficiente, también permitir la integración de servicios de terceros y permitir que sistemas con distintos lenguajes o tecnologías puedan trabajar juntos.

2. **Explica el ciclo de vida de una solicitud HTTP en una API.**

1. **Cliente envía la solicitud HTTP**
- **Método HTTP**: GET, POST, PUT, DELETE, etc.
- **Endpoint**: Ruta específica (ej: /api/users).
- **Headers**:
    - Content-Type: Tipo de dato enviado (ej: application/json).
    - Authorization: Credenciales (ej: API Key, JWT).
- **Body** (opcional): Datos en formato JSON, XML, etc. (ej: {"name": "Alice"}).

2. **Servidor web recibe la petición**
- **Servidores como Nginx/Apache** actúan como intermediarios:
    - Redirigen la petición al backend (ej: una app en Node.js o Laravel).

3. **Procesamiento en la API**
- **Routing**: El framework (Laravel, Spring, etc.) identifica la ruta y método solicitado.
- **Middlewares**: Capas que ejecutan lógica antes de llegar al controlador:
    - Autenticación (ej: verificar un token JWT).
    - Permisos (ej: saber si el cliente que hizo la petición tiene los suficientes permisos para realizarla).
- **Controlador**:
    - Consulta a bases de datos (MySQL, MongoDB).
    - Lógica de negocio (cálculos, llamadas a otras APIs).

4. **Generación de la respuesta**
- **Status Code**:
    - 2xx (Éxito): 200 OK, 201 Created.
    - 4xx (Error cliente): 400 Bad Request, 404 Not Found.
    - 5xx (Error servidor): 500 Internal Server Error.
- **Headers**:
    - Content-Type: application/json.
- **Body**: Datos solicitados (ej: lista de usuarios en JSON).

5. **Envío al cliente y cierre de conexión**
- La respuesta viaja al cliente a través de la red.

3. **¿Qué es la inyección de dependencias y cómo se implementa en una API?**
La inyección de dependencias es un patron de diseño que permite separar responsabilidades, separando la lógica de negocio, acceso a datos y el manejo de peticiones, nos facilita el cambio de implementaciones, por ejemplo:

al querer cambiar una base de datos sql a una base de datos no sql simplemente tienes que hacer el cambio en el apartado que se encarga de el acceso a los datos y no tener que cambiarlo en multiples partes de nuestro proyecto.

Para poder implementar la inyección de dependencias necesitamos separar nuestro proyecto en varias partes:

1. Modelo/Entidad *(Clases que representan tablas/colecciones)*
- **Propósito**:
    - Definir la estructura de datos de la aplicación (ej: tablas de BD, documentos JSON).
    - Mapear objetos a la base de datos (usando anotaciones de JPA o MongoDB).

2. Repositorio (@Repository**)** *(Acceso a datos)*
- **Propósito**:
    - Gestionar operaciones CRUD (crear, leer, actualizar, eliminar) con la base de datos.
    - Spring Data JPA/MongoDB implementa los métodos automáticamente.

3. Servicio (@Service**)** *(Lógica de negocio)*
- **Propósito**:
    - Contener reglas de negocio (validaciones, cálculos, etc.).
    - **No interactúa directamente con la BD**, usa el repositorio inyectado.

4. Controlador (@Controller***/***@RestController**)** *(Manejo de peticiones HTTP)*
- **Propósito**:
    - Recibir peticiones HTTP y devolver respuestas (REST API).
    - **No contiene lógica de negocio**, la delega al servicio inyectado.

Con la inyección de dependencias ya separamos responsabilidades entre esos apartados, estos apartados hacen uso uno del otro, pero cada quien hace un trabajo diferente, el repositorio hace uso de el modelo para trabajar con el mismo en la base de datos y proporcionar el acceso a los datos, el Service o servicio hace uso del repositorio para poder trabajar con los datos y manejar la lógica de negocio, por ultimo el controlador maneja las peticiones http y hace uso del service para poder responder al cliente que hizo la solicitud http.

4. **¿Cuáles son las diferencias entre REST y SOAP?**
   REST

   - Basado en HTTP/HTTPS y sigue principios de arquitectuca RESTful
   - Usa métodos HTTP estándar GET, POST, PUT, DELETE.
   - Más ligero y rápido (JSON es menos verboso que XML).
   - Ideal para APIs públicas, aplicaciones móviles y web.
   - No tiene soporte nativo para transacciones complejas (ACID).

   SOAP

   - Basado en XML y sigue un protocolo estricto con estructura definida (WSDL)
   - Usa únicamente POST para todas las operaciones
   - Más lento debido al peso del XML y el procesamiento adicional.
   - Inclute WS-Security (encriptación y formas digitales dentro del XML).
   - Mas robusto para datos sensibles (ej: transacciones financieras)

5. **¿Qué es una API RESTful y cuáles son sus principios fundamentales?**

Una API RESTful es una interfaz que sigue los principios de REST (Representational State Transfer), un estilo arquitectónico para diseñar servicios web escalables, flexibles y stateless. Se comunica mediante HTTP/HTTPS y usa formatos como JSON o XML para intercambiar datos.

Principios fundamentales:

1. Stateless (Sin estado)
   - Cada petición HTTP debe contener toda la información necesaria para ser procesada.
   - El servidor no guarda contexto entre peticiones (ej: sin sesiones).
   - Ejemplo:
   - El token de autenticación va en el header de cada petición, no se almacena en el servidor.

2. Client-Server (Cliente-Servidor Separados)

   - El cliente (frontend) y el servidor (backend) son independientes.
   - El cliente solo necesita conocer los endpoints de la API, no cómo se implementa la lógica.

3. Cacheable (Cacheable)

   - Las respuestas deben indicar si pueden ser almacenadas en caché (mejora el rendimiento).

4. Uniform Interface (Interfaz Uniforme)
   - URIs consistentes: Identifican recursos de forma única (ej: /api/users, /api/users/1).
   - Métodos HTTP estándar:
     - GET: Obtener un recurso.
     - POST: Crear un recurso.
     - PUT/PATCH: Actualizar un recurso.
     - DELETE: Eliminar un recurso.

5. Buenas Prácticas en APIs RESTful
   1. Usar sustantivos en URIs (no verbos):
      - /api/users (correcto).
      - /api/getUsers (incorrecto).
   2. Versionar la API:
      - /api/v1/users, /api/v2/users.
   3. Usar códigos HTTP adecuados:
      - 200 OK: Éxito.
      - 201 Created: Recurso creado.
      - 404 Not Found: Recurso no existe.
      - 401 Unauthorized: No autenticado.

Sección 2: Desarrollo de APIs

6. **¿Qué son los controladores en una API y cómo se definen?**

   Los controladores (o *Controllers*) en una API son componentes que:

   - Reciben peticiones HTTP (desde clientes como navegadores, apps móviles, etc.).
   - Procesan la lógica (validaciones, llamadas a servicios).
   - Devuelven respuestas (JSON, XML, códigos HTTP).

   Como se definen (Spring boot Java):

```java
@RestController
@RequestMapping("/api/usuario")
public class UsuarioController {
    private final UsuarioService usuarioService;

    @GetMapping  Redaserf
    public ResponseEntity<List<Usuario>> getAllUsuarios(){
        return new ResponseEntity<>(usuarioService.getAllUsuarios(), HttpStatus.OK);
    }

    @PostMapping  Redaserf
    public ResponseEntity<Usuario> postUsuario(@RequestBody @Valid Usuario usuario) throws UniqueException {
        return new ResponseEntity<>(usuarioService.postUsuario(usuario), HttpStatus.CREATED);
```

7. **¿Cuál es la diferencia entre `GET`, `POST`, `PUT` y `DELETE`? Da un ejemplo de cada uno.**
    - GET: obtener recursos
    - POST: Crear recursos
    - PUT: Actualizar recursos
    - DELETE: Eliminar recursos

GET:

Un ejemplo de get puede ser por ejemplo cuando quieres obtener los usuarios que hay en una aplicación o por ejemplo cuando quieres saber cuales son los perfiles de tus amigos en facebook.

POST:

Registrarte en una aplicación, subes tu información para que se guarde en una aplicación.

PUT:

Actualizar los datos de tu perfil en facebook, tu nombre, tu email, etc.

DELETE:

Eliminar tu cuenta de facebook.

8. **¿Cómo manejas errores en una API? Explica con código.**

Crear una excepción personalizada para manejar un error como puede ser cuando un registro solicitado no fue encontrado:

```
public class NotFoundExcpetion extends Exception{  10 usages  Redaserf
    public NotFoundExcpetion(String message) {  2 usages  Redaserf
        super(message);
    }
}
```

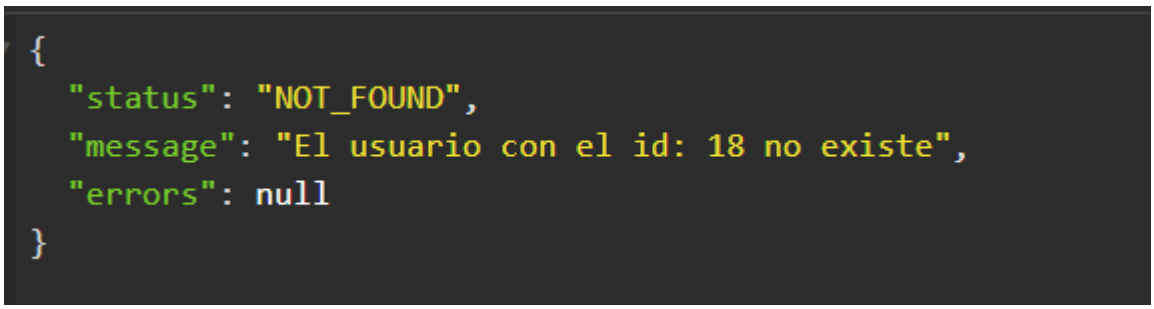
Tener un manejador de errores globales incluyendo las excepciones personalizadas que creas:

```
@ControllerAdvice  Redaserf *
public class RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(NotFoundExcpetion.class)  new *
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ResponseEntity<ErrorMessage> handleNotFoundExcpetion(NotFoundExcpetion ex){
        ErrorMessage errorMessage = new ErrorMessage(HttpStatus.NOT_FOUND, ex.getMessage(),  errors: null);

        return new ResponseEntity<>(errorMessage, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(UniqueException.class)  Redaserf
    @ResponseStatus(HttpStatus.CONFLICT)
    public ResponseEntity<ErrorMessage> handleUniqueException(UniqueException ex){
        ErrorMessage errorMessage = new ErrorMessage(HttpStatus.CONFLICT, ex.getMessage(),  errors: null);

        return new ResponseEntity<>(errorMessage, HttpStatus.CONFLICT);
    }
```

Arrojar la excepción una vez haya llegado un resultado que no se esperaba:

```
Usuario usuarioBD = usuarioRepository.findById(id)
.orElseThrow(() -> new NotFoundExcpetion("El usuario con el id: " + id + " no existe"));
```

Una vez arrojada la excepción se muestra el mensaje al cliente para que este enterado de el error:



Sección 3: Seguridad y Buenas Prácticas

11. **Explica cómo implementar autenticación y autorización en una API.**

La autenticación verifica la identidad del usuario. En APIs modernas, el método más común es mediante tokens (como JWT o SACTUM en laravel).

Proceso:

1. Login: El usuario envía credenciales (email + contraseña).
2. Validación: El servidor verifica que coincidan con la base de datos.
3. Generación de token: Si son correctas, se crea un token único (ej: JWT, sanctum, etc.) que contiene:
    - Identificador del usuario (ej: userId, email).
    - Tiempo de expiración (ej: 1 hora).
    - Firma digital (para evitar manipulaciones).

La autorización define qué recursos o acciones puede realizar el usuario autenticado.

Mecanismos clave:

- Roles: ADMIN, USER, GUEST (normalmente usados).
- Permisos: obtener usuarios, eliminar usuarios.

En Spring Boot, se usan filtros o interceptores similares a los middlewares en Laravel para:

1. Interceptar peticiones antes de llegar al controlador.
2. Validar el token (ej: verificar firma, expiración).
3. Verificar permisos:
    - Extraer el userId o roles del token.
    - Consultar en BD si el usuario tiene acceso al recurso solicitado.

12. **¿Qué es JWT y cómo se usa en una API?**

    Un JWT (JSON Web Token) es un token de acceso estandarizado que permite el intercambio seguro de datos entre dos partes es como una credencial única que te da el acceso a ciertas partes en este caso a una aplicación. Se define en el RFC 7519 y es una cadena codificada que puede contener una cantidad ilimitada de datos, firmada criptográficamente para verificar su autenticidad.

    ¿Cómo se usa JWT en una API?

    1. Cliente inicia sesión: Envía usuario/contraseña.
    2. API valida credenciales: Si son correctas, genera un JWT y lo devuelve.
    3. Cliente (aplicación web, aplicación móvil, etc.) almacena el token:
       En localStorage, cookies o memoria.
    4. Cliente envía el token: En el header Authorization de cada petición

13. **¿Cómo evitar ataques de inyección SQL en una API?**

Validar las entradas con expresiones regulares o librerías como OWASP ESAPI, usar prepared statements (sentencias preparadas), restringir lo mas posible el acceso a la base de datos, por ejemplo crear un usuario para la api el cual solo tenga permisos para realizar operaciones select, insert, update y delete, sin los permisos para crear usuarios con máximos privilegios.

14. **¿Qué son los filtros en una API y para qué se usan?**

En Spring Boot, un Filtro (Filter) es una clase que intercepta las peticiones y respuestas HTTP antes o después de que lleguen a un controlador.
Se usan, por ejemplo, para:

- Validar tokens de autenticación (JWT).
- Registrar (loggear) información de las peticiones.
- Modificar las respuestas.
- Implementar medidas de seguridad.

15. **¿Cómo manejas la paginación y el cacheo en una API?**

Paginación:
La paginación se usa para dividir grandes cantidades de datos en partes más pequeñas y manejables, mejorando el rendimiento y la experiencia del usuario.
Se maneja agregando parámetros en la URL, como page (página) y size (cantidad de resultados por página).

Cacheo:
El cacheo sirve para guardar temporalmente respuestas de la API, reduciendo la carga del servidor y mejorando la velocidad.
Se puede manejar de dos formas:

- Cacheo a nivel de cliente o proxy: usando cabeceras HTTP como Cache-Control, ETag, Last-Modified.

- Cacheo a nivel de servidor: guardando en memoria (por ejemplo, usando Redis, Caffeine, o el caché de Spring Boot con @Cacheable).

Sección 4: Base de Datos y ORM

16. **¿Qué es un ORM y cómo se diferencia de las consultas SQL tradicionales?**

Un ORM (Object-Relational Mapping) es una herramienta que permite trabajar con bases de datos usando objetos del lenguaje de programación en lugar de escribir SQL directamente. Traduce operaciones con objetos a consultas SQL automáticamente.

ORM: utiliza clases y metodos, es mas legible y reutilizable, abstrae detalles del motor de base de datos.

SQL:

Usa sentencias sql tradicionales, mas control y rendimiento, se necesita escribir toda la query completa.

17. **Explica cómo configurar una conexión a una base de datos en una API.**

En una api se necesitan varias configuraciones para establecer conexión con una base de datos, por ejemplo la dirección en la que se encuentra el servidor puede ser de forma local algo así: localhost:3306/base_datos, se necesita el username y la contraseña para poder acceder a la base de datos, el driver de la base de datos que se esta usando y con eso podría conectarte a una base de datos.

La configuración se vería así:

```
spring.datasource.url=jdbc:mysql://localhost:3306/prueba_tecnica
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

18. **¿Cómo se realizan consultas asincrónicas en una API?**

Bueno este depende en que lenguaje se este haciendo la API, pero por ejemplo en java con spring boot se realiza con la anotación de @Async

19. **¿Qué son las migraciones en bases de datos y cómo se usan?**

Las migraciones son archivos que definen cambios en la estructura de la base de datos (crear tablas, modificar columnas, etc.) de forma controlada y versionada.

Flyway en Spring Boot:

- Agregas un archivo V1__crear_tabla_usuario.sql en la carpeta resources/db/migration.
- Flyway ejecuta ese archivo automáticamente al iniciar la app.

20. **¿Cómo implementar relaciones uno a muchos en una base de datos?**

En sql al crear las tablas necesitas definir sus primary keys, para en la tabla donde se va hacer referencia a la otra se declare el cambo que viene siendo la foreign key, y hacia que tabla y la primary key con la cual hacer referencia la foreign key:

```
CREATE TABLE clientes (
  id INT PRIMARY KEY
);

CREATE TABLE ordenes (
  id INT PRIMARY KEY,
  cliente_id INT,
  FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);
```

ORM en spring boot con jpa:

En una entidad se pone la anotación oneToMany y en la otra entidad la misma relación pero a la inversa ManyToOne y la columna con la cual hace referencia a la primary key de la otra entidad:

```
@Entity
public class Cliente {
    @OneToMany(mappedBy = "cliente")
    private List<Orden> ordenes;
}

@Entity
public class Orden {
    @ManyToOne
    @JoinColumn(name = "cliente_id")
    private Cliente cliente;}
```

Sección 5: Desafío Práctico

Escenario: Se requiere construir una API que maneje una entidad `Usuario` con las siguientes propiedades: `Id`, `Nombre`, `Email` y `FechaRegistro`.

Tareas:

Define el modelo de `Usuario`.

```
{
   "Id": 1,
   "Nombre": "Juan Pérez",
   "Email": "juan@example.com",
   "FechaRegistro": "2024-01-01T12:00:00Z"

}
```

1. **Implementa una conexión a una base de datos y la migración inicial.**
2. **Crea un controlador con los métodos CRUD para `Usuario`.**
3. **Implementa validaciones básicas en el modelo.**
4. **Explica cómo proteger la API con autenticación JWT.**

Nota: el desafío práctico se puede realizar en cualquier lenguaje, pero de preferencia usar java.

Explicación como proteger la API con autenticación JWT

Para proteger la api con jwt, necesitas implementar las dependencias necesarias, vienen siendo spring security y jjwt-api, jjwt-impl, jjwt-jackson, estas dependencias son necesarias para poder llevar acabo la autenticación de la api.

1. **Modelo Usuario**
Debes crear una clase Usuario que implemente UserDetails. Esta clase representa los datos del usuario y debe incluir campos como:

- email o username: será el identificador único.
- password: la contraseña encriptada.
- rol o lista de roles.
- Implementaciones de los métodos isAccountNonExpired(), isAccountNonLocked(), etc., que generalmente devuelven true si no usas restricciones adicionales.

2. Modelo rol:

- Este modelo es necesario si se va a manejar control de permisos en la aplicación.

3. **Modelo Rol (opcional)**
Este modelo puede ser un enum o una entidad, es útil si deseas controlar permisos con más detalle, por ejemplo: el usuario con rol empleado no podrá acceder a ciertos endpoint de nuestra API, como ver usuarios.

4. **Servicio JWT**
Es necesario un servicio que se encargue de generar y validar tokens JWT, usando una clave secreta. Este servicio implementa métodos como:

- generateToken(UserDetails userDetails)
- isTokenValid(String token, UserDetails userDetails)
- getUserNameFromToken(String token)

Este servicio utiliza la biblioteca JJWT para construir y parsear tokens.

5. **Filtro JwtAuthenticationFilter**
Este filtro intercepta cada petición, extrae el token del encabezado Authorization, y si es válido, autentica al usuario en el contexto de Spring, si no es valido no deja seguir el ciclo de vida del request.

6. **Configuración de Seguridad (SecurityConfig)**
Aquí defines:

- Qué rutas son públicas (permitAll). Por ejemplo login y registro, si no fueran publicas nadie se podría autenticar.
- Qué rutas requieren autenticación (authenticated)
- Tu JwtAuthenticationFilter personalizado
- Tu AuthenticationEntryPoint para manejar errores (como tokens inválidos)

7. **Autenticación y generación del token**
Se recomienda tener un endpoint como /auth/login donde recibes email y password, autenticas con AuthenticationManager, y si es correcto, devuelves el token JWT al cliente.

8. Enviar el token por cada peticion a la API:
El cliente debe guardar el token que recibe al logearse, para poder enviarlo a cada peticion que hace a nuestra API y ser correctamente autenticado.