Exámen Backend

Sección 1: Conceptos Generales

¿Qué es una API y cuál es su propósito principal?

Una API como sus siglas lo dicen es una APPLICATION PROGRAMMING INTERFACE son un conjunto de reglas y prótocolos que permiten la comunicación de distintos softwares, fácilitando el acceso a funciones o datos de aplicaciones.

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

El ciclo inicia cuando un cliente envía una solicitud HTTP a la API. El servidor recibe la petición, la enruta al controlador correspondiente, procesa la lógica (incluyendo validaciones y acceso a la base de datos), genera una respuesta (generalmente en JSON) y la devuelve al cliente con el código de estado HTTP adecuado.

¿Qué es la inyección de dependencias y cómo se implementa en una API?

La inyección de dependencias (DI) es un patrón que permite pasar objetos (dependencias) a una clase en lugar de que esta los cree. Mejora la escalabilidad y testeo del código. En una API, se implementa pasando servicios o repositorios a los controladores a través del constructor o de un contenedor IoC (Inversión de Control), como en Laravel o AdonisJS.

¿Cuáles son las diferencias entre REST y SOAP?

REST usa HTTP estándar, es ligero, flexible y devuelve datos en formatos como JSON.

SOAP es un protocolo más estricto, basado en XML, con reglas rígidas y mayor seguridad incorporada. REST es más común hoy en día en APIs web por su simplicidad y compatibilidad con múltiples plataformas.

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

Una API RESTful sigue los principios de REST, un estilo arquitectónico que usa HTTP para operar sobre recursos. Sus principios clave son:

Uso de métodos HTTP estándar (GET, POST, PUT, DELETE)

Acceso a recursos a través de URLs únicas

Sin estado: cada solicitud es independiente

Formato uniforme de respuestas (generalmente JSON)

Uso de códigos de estado HTTP adecuados

Sección 2: Desarrollo de APIs

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

Los controladores en la API son fundamentales para estas mismas, ya que aquí dentro es donde se realizan todas las operaciones que se solicitan y se reciben las solicitudes del cliente, desde actualizar, crear y eliminar recursos. Se definen como una clase, también ayuda a organizar de mejor manera el código. Por estándar se suele nombrar como controller un ejemplo sería UsersController, fácilitando así la manera de saber si es un controlador o no.

¿Cuál es la diferencia entre GET, POST, PUT y DELETE? Da un ejemplo de cada uno.

La diferencia de estos verbos rádica en como su mismo nombre lo índica, uno sirve para obtener, otro para guardar, otro para actualizar y otro para eliminar.

GET: El verbo GET se usa para solicitar datos del servidor sin modificar nada. Por ejemplo, si tenemos una lista de usuarios en una base de datos, al realizar una petición GET al servidor, este nos enviará la lista completa de usuarios. También se puede utilizar para obtener un único registro.

POST: El verbo POST se utiliza para enviar datos al servidor para la creación de recursos. Por ejemplo, si queremos registrar un nuevo usuario realizamos una solicitud POST enviando la información necesaria (como nombre, correo electrónico y contraseña) en el cuerpo de la solicitud. Si la operación es exitosa, el servidor suele responder con el nuevo recurso creado o con un mensaje de confirmación.

PUT: El verbo PUT se utiliza para actualizar datos que ya existen en el servidor. Por ejemplo, si queremos editar a un usuario que ya existe en nuestra base de datos, realizamos la solicitud al servidor con el verbo PUT y el identificador del usuario en la URL con la información necesaria (nombre, email, etc) el servidor procesa la información y nos responde con el recurso actualizado o un mensaje de confirmación.

DELETE: El verbo DELETE se utiliza para solicitar eliminar un registro. Por ejemplo, si queremos eliminar un usuario, al realizar la petición DELETE incluyendo el identificador del usuario en la URL al servidor con el usuario que deseamos eliminar el mismo servidor lo eliminará y nos enviará una respuesta de confirmación.

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

En el siguiente código podemos observar cómo manejar errores en nuestra API, esté es el método que recibe la solicitud desde el cliente, el cuál, es para registrar un usuario, vemos que si el nombre del usuario es nulo o está vacío retornamos un mensaje de error, así mismo si el email no es válido o si es nulo o vacío. Este código es para un controller con Java y Springboot. También es clave mencionar que tenemos que hacer el uso de los Status Code cuando se trata de un prótocolo HTTP.

```
@PostMapping()
public ResponseEntity<String> guardarUsuario(@RequestBody Usuario usuario) {
    // Validar nombre
    if (usuario.getNombre() == null || usuario.getNombre().isEmpty()) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("El nombre es
obligatorio.");
    }
    // Validar email
    if (usuario.getEmail() == null || usuario.getEmail().isEmpty()) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("El email es
```

```
obligatorio.");
    }
    // Validar formato del email
    String emailRegex = "^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$";
    if (!usuario.getEmail().matches(emailRegex)) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("El email no tiene un
formato válido.");
    }
    // Verificar si el email ya está registrado
    Optional<Usuario> usuarioExistente = usuarioRepository.findByEmail(usuario.getEmail());
    if (usuarioExistente.isPresent()) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("El email ya está
registrado.");
    }
    // Establecer la fecha de registro
    usuario.setFechaRegistro(new Date());
    // Guardar el usuario
    usuarioService.guardarUsuario(usuario);
    return ResponseEntity.status(HttpStatus.CREATED).body("Usuario creado exitosamente.");
}
```

Sección 3: Seguridad y Buenas Prácticas

Explica cómo implementar autenticación y autorización en una API.

Para implementar autenticación en una API es necesario el uso de tokens, como JWT, Sanctum, entre otros. Estos tokens funcionan como credenciales que permiten acceder a rutas seguras. Para obtener un token, es necesario realizar un login, donde el servidor valida las credenciales proporcionadas y, si son correctas, otorga un token de acceso. En cuanto a la autorización, depende si la API la necesita, pero generalmente es esencial.
Para implementarla, se pueden crear campos como roles o permisos en la base de datos (por ejemplo: admin, usuario, invitado, etc.). Luego, mediante el uso de middlewares, se verifica si el usuario tiene acceso a determinadas rutas o si cuenta con los permisos necesarios para realizar ciertas acciones.

¿Qué es JWT y cómo se usa en una API?

JWT (JSON WEB TOKEN), se utiliza principalmente para la autenticación y autorización en APIs. Cuándo un usuario inicia sesión correctamente, el servidor genera un JWT y se entrega al cliente. Después en cada solicitud el cliente debe enviar este token (normalmente en el encabezado) para acceder a las rutas seguras. Después el servidor verifica que el token sea válido y procesa la solicitud.

¿Cómo evitar ataques de inyección SQL en una API?

Para evitar inyecciones SQL en una API, se deben usar consultas preparadas o un ORM, que manejan los parámetros de forma segura. También es importante validar y sanitizar los datos recibidos, evitar concatenar strings en consultas SQL y aplicar principios de seguridad como el mínimo privilegio en la base de datos.

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

Los filtros en una API permiten al cliente obtener recursos específicos según ciertos criterios. Se utilizan para refinar los resultados de una consulta sin necesidad de recuperar todos los datos. Por ejemplo, se puede filtrar por campos como id, nombre, teléfono, fecha, entre otros.

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

Para manejar la paginación en una API, es común utilizar métodos nativos del framework o biblioteca utilizada (por ejemplo, paginate() en Laravel o Model.paginate() en AdonisJS). La paginación permite dividir grandes conjuntos de datos en páginas más pequeñas, lo que mejora el rendimiento y facilita el consumo de la API. Puede configurarse especificando el número de elementos por página (limit) y la página actual (page), ya sea mediante parámetros en la URL o configuraciones internas.

Por otro lado, el cacheo se utiliza para almacenar temporalmente respuestas o resultados de consultas costosas, reduciendo la carga sobre la base de datos y mejorando el tiempo de respuesta. Dependiendo de la tecnología, se puede implementar usando soluciones como Redis, Memcached o sistemas de archivo. Es importante definir una estrategia de expiración y actualización del caché para garantizar que los datos estén actualizados cuando sea necesario.

Sección 4: Base de Datos y ORM

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

El ORM (Object Relational Mapping) funciona para poder relacionar los datos del modelo con la base de datos. La diferencia rádica en que no tenemos que construir una consulta como tal para acceder a la base de datos y hacer las operaciones, sino que este mismo ya cuenta con lo necesario para realizarlo. Cabe recalcar que los ORM también tienen estas relaciones de uno a muchos, muchos a muchos y muchos a uno. Importante mencionar también que de esta manera evitamos escribir la sentencia SQL y podemos realizar las operaciones con los métodos y las propiedades del modelo.

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

Para configurar una conexión a una base de datos en una API necesitamos usar una líbreria o motor (Eloquent, knex, etc) se requieren las credenciales que se nos soliciten si estamos usando un framework, pueden ser desde el usuario, la contraseña, el puerto y el nombre de las bases de datos. Como mencione previamente esto va variando dependiendo el entorno en el que estemos desarrollando, pero esto es lo básico para realizar la conexión a una base de datos en una API. Normalmente estas credenciales se definen en el archivo .ENV.

¿Cómo se realizan consultas asincrónicas en una API?

Para realizar consultas asincrónicas en una API, es requerido colocar la palabra async en la función que procesa la solicitud y dentro de esta misma await para esperar la respuesta de

las operaciones que son asincrónicas como consultas a bases de datos o llamados a otros servicios.

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

Las migraciones en las bases de datos son archivos que contienen instrucciones para crear, editar o eliminar la estructura de las bases de datos, como las tablas, columnas, índices o relaciones. Se utilizan para facilitar la gestión de cambios de manera controlada permitiendo que los cambios puedan ser aplicados y revertidos fácilmente en distintos entornos.

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

Para implementar una relación uno a muchos en una base de datos es necesario utilizar una clave foránea en la tabla que representa el "muchos". Con esta clave foránea hacemos referencia a la clave primaria de la tabla que representa el "uno" y de esta manera la clave primaria de la tabla principal aparecerá múltiples veces en la tabla relacionada, estableciendo así la relación de uno a muchos.

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"

}
```

- **Implementa una conexión a una base de datos y la migración inicial.**
- **Crea un controlador con los métodos CRUD para Usuario.**
- **Implementa validaciones básicas en el modelo.**

Explica cómo proteger la API con autenticación JWT.

1. Primero necesitamos comprender el flujo.

- JWT se usa para autenticar usuarios de forma segura sin mantener sesiones en el servidor. El flujo básico es:
- El usuario envía sus credenciales (usuario/contraseña).
- Si las credenciales son válidas, el servidor genera un token JWT y se lo devuelve al cliente.
- El cliente guarda ese token (usualmente en el navegador o app) y lo envía en cada petición.
- El servidor verifica el token antes de responder a cualquier solicitud protegida.

2. Autenticación inicial del usuario

- Debemos tener un endpoint (por ejemplo, /login) que reciba las credenciales del usuario. En este punto se validan con los datos que hay en nuestras bases de datos.

3. Generación del JWT

Cuando las credenciales son correctas, generas un JWT. El token incluye:

- Información del usuario (como el ID o roles) en el payload.
- Una firma generada con una clave secreta para garantizar la integridad del token.
- Un tiempo de expiración.

4. Devolución del token al cliente

- Una vez generado el JWT, se envía como respuesta al cliente. El cliente lo debe almacenar (usualmente en localStorage o en memoria).

5. Protección de rutas

En los endpoints protegidos de nuestra API, debemos interceptar cada petición y verificar si incluye un token válido:

- Extraesmos el token del encabezado Authorization (usualmente en el formato Bearer <token>).
- Lo verificamos usando la misma clave secreta.
- Si es válido y no ha expirado, permitiremos la ejecución del endpoint.
- Si no es válido, devolvemos un error 401 (no autorizado).

6. Decodificación y uso de datos del token

Al validar el token, puedes acceder a los datos que colocaste en el payload (como el userId) y usarlos para aplicar lógica de negocio, como identificar al usuario que hizo la solicitud.

7. Manejo de expiración y renovación del token

El token debe tener una duración limitada (por ejemplo, 15 minutos o 1 hora).

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