1) **¿Qué dependencias utiliza el proyecto? ¿Dónde se encuentran explicitadas?**

Las dependencias que utiliza el proyecto son:

"@hapi/boom": "^9.1.4",
"cors": "^2.8.5",
"crypto": "^1.0.1",
"dotenv": "^16.0.0",
"express": "^4.17.2",
"joi": "^17.6.0",
"jsonwebtoken": "^8.5.1",
"mssql": "^8.0.1"

Se encuentran explicitadas en el archivo de configuración “package.json”.

2) **¿Qué scripts de inicio posee?**

El script de inicio que posee es "start": "node index.js".

3) **¿Cuál es el objetivo que posee la línea const dotenv = require('dotenv').config({ path: path.resolve(__dirname, '../.env') }); en el archivo database/database.js?**

El objetivo que posee esta línea es cargar variables de entorno de un archivo .env ubicado en el directorio raíz del proyecto.
El path.resolve que se utiliza en esta línea de código es para obtener la ruta absoluta del archivo .env y así asegurarse de que se esté utilizando la ubicación correcta del archivo.
Una vez cargadas las variables de entorno, estas pueden ser accedidas en el archivo “database.js” y utilizadas para establecer las credenciales de acceso a la base de datos.

4) **La estructura de la aplicación contempla los directorios routes, controllers, services, middlewares y schemas. ¿Qué función cumple cada uno?**

Routes: Este directorio contiene los archivos que definen las rutas para la aplicación. Cada ruta se define en un archivo separado para facilitar el mantenimiento del código. En estos archivos se importan los controladores adecuados para cada ruta.

Controllers: Este directorio contiene los archivos que definen la lógica para cada ruta. Los controladores son responsables de procesar las solicitudes del cliente y devolver una respuesta adecuada.

Services: Este directorio contiene los archivos que definen la lógica que no está directamente relacionada con las rutas de la aplicación.

Middlewares: Este directorio contiene los archivos que definen los middlewares utilizados en la aplicación. Los middlewares son funciones que se ejecutan antes o después de una solicitud para realizar tareas adicionales, como la autenticación, validación de datos, y gestión de errores.

Schemas: Este directorio contiene los archivos que definen los modelos y esquemas de datos utilizados en la aplicación. Los modelos y esquemas son estructuras de datos que se utilizan para definir la estructura y los tipos de datos que se esperan en una solicitud o en una respuesta HTTP.

5) Explicar el funcionamiento del método get de la clase ProductService.

Se utiliza para obtener información de uno o varios productos de una base de datos. Primero se recibe el id como parámetro en la función. Luego se define una consulta SQL para obtener la información del producto o de todos los productos si id es null. Después se establece una conexión a la base de datos utilizando el método connect(). Si se han encontrado productos en la base de datos, se devuelve un arreglo de objetos JSON que representan los productos encontrados en la tabla 'vwGetProduct's y si no se han encontrado productos en la base de datos, se devuelve un arreglo vacío.
Por último, si ocurre algún error al ejecutar la consulta SQL, se lanzará una excepción y se devolverá un mensaje de error "throw 'Error inesperado'".

6) Explicar los pasos que se realizan al ejecutarse la función delete del archivo movement.service.js.

Los pasos de la función delete son los siguientes:

- Se recibe el id como parámetro en la función.
- Se establece una conexión a la base de datos utilizando el método connect(), esta devuelve un objeto 'pool' que se utilizará para realizar las consultas a la base de datos.
- Se ejecuta una consulta SQL para obtener el identificador del producto y la cantidad de productos del movimiento correspondiente.
- Una vez obtenida la información del movimiento, se ejecuta otra consulta SQL para actualizar la cantidad de stock del producto en la tabla products. La consulta utiliza el identificador del producto (idproduct) y la cantidad de productos (quantity) obtenidos en el paso anterior.
- Finalmente, se ejecuta una última consulta SQL para eliminar el movimiento correspondiente en la tabla movements.
- Si todo ha sido exitoso, se devuelve el resultado de la consulta SQL de eliminación (result).
- Si ocurre algún error en cualquiera de las consultas SQL, se lanzará una excepción y se devolverá un mensaje de error.

7) En auth.service.js se generan mediante throw dos excepciones, boom.unauthorized("invalid access") y boom.badRequest(err) ¿Por qué se generan mediante boom? ¿Desde dónde se recuperan o capturan? ¿En qué colabora la implementación de dicha clase?

Las excepciones boom.unauthorized("invalid access") y boom.badRequest(err) se generan mediante Boom en el archivo auth.service.js porque esta biblioteca da utilidades para manejar errores en Node.js facilita la creación de errores HTTP y devuelve al cliente un mensaje y un código de estado HTTP adecuado.

En el archivo auth.controller.js, se capturan estas excepciones mediante el uso de un bloque try/catch para manejar los errores generados por el servicio de autenticación. En caso de que se genere una excepción, el controlador captura la excepción y devuelve una respuesta HTTP con un mensaje de error y un código de estado HTTP correspondiente.

La implementación de la clase AuthService colabora en la modularización y organización del código relacionado con la autenticación en la aplicación. Al tener un servicio separado para manejar la lógica de autenticación, es posible reutilizar el código en diferentes partes de la aplicación.

8) ¿Qué función cumple el token que se genera con jsonwebtoken? En el ejemplo ¿De qué archivo depende su generación? ¿En qué archivo se crea y cuándo y mediante qué, se valida nuevamente?

El token generado con jsonwebtoken es un objeto JSON que contiene información codificada que se utiliza para autenticar y autorizar solicitudes entre el cliente y el servidor. El token se utiliza para mantener el estado de la sesión en la aplicación y para verificar que el usuario que realiza la solicitud es auténtico y tiene los permisos necesarios para acceder a los recursos protegidos.

El archivo auth.service.js se encarga de generar el token y el archivo validator.login.js se encarga de validar el token en las rutas protegidas.

9) ¿Qué verificaciones lleva adelante ProductSchema?

Las verificaciones que lleva a cabo son:

idcategory: valida que sea un número entero con Joi.number().integer(), y que esté dentro del rango de 0 a 255 con .min(0).max(255). Además, se establece que es obligatorio con .required().

denomination: valida que sea una cadena de caracteres con .string(), con una longitud mínima de 10 y máxima de 255 con .min(10).max(255). También se establece que es obligatorio con .required().

additional_info: valida que sea una cadena de caracteres con .string(), con una longitud mínima de 0 y máxima de 100 con .min(0).max(100). Además, se establece que es opcional con .optional().

price: valida que sea un número con .number(), y que tenga una precisión de 2 decimales con .precision(2). No se establece que sea obligatorio ni un rango específico.

stock: valida que sea un número entero con .number().integer(), y que esté dentro del rango de 0 a 100 con .min(0).max(100). No se establece que sea obligatorio.

10) ¿Qué utilidad implica la implementación de la instrucción next()?

La instrucción next() se utiliza cuando una solicitud HTTP llega a una función middleware, ésta puede realizar alguna lógica y luego decidir si quiere pasar el control de la solicitud a la siguiente función middleware en la cadena de middleware o enviar una respuesta al cliente y finalizar la solicitud. Si se llama a la función next(), el control de la solicitud se pasa a la siguiente función middleware en la cadena y si no se llama a next(), la solicitud se termina sin pasar el control a la siguiente función middleware.