**README**

Información Técnica

El proyecto se realiza utilizando .NET Core 5 e implementando asincronía en cada una de las capas. A continuación, se lista cada requerimiento de negocio con su respectiva función desarrollada.

* Consulta si un producto X existe en el inventario: Se crea un método de filtro por Id en el controlador de productos. Si el producto existe, se devolverá un objeto con los datos del producto. Sin el producto no existe se devolverá un código de error 404.
* Listar las bodegas existentes: Se crea un método para traer las bodegas existentes en la base de datos. Se crea un mecanismo básico de paginación y por defecto, cuando no se proporcionan parámetros, se toma la página con índice cero y el tamaño de la página es diez. No es permitida una consulta mayor a 100 registros, por lo que el sistema cambiará el tamaño automáticamente si el número parametrizado excede esta cantidad.
* Cargar productos a una bodega, mover productos de una bodega a otra y sacar productos de una bodega: Se resolvieron estos tres puntos bajo la creación de nuevos movimientos. Los movimientos deben ir ligados a un tipo de movimiento que determinará si se restan o se suman las cantidades o si es necesario afectar una bodega de destino, en caso de ser un traslado.
* Mostrar la cantidad total de una mercancía X y su valor acumulado: En este punto se creó un nuevo endpoint en el controlador de Productos. Desde swagger se podrá visualizar mediante la URL /api/v1/Product/stock/{id}.

**Patrones de diseño aplicados**

Para la realización de la prueba se implementaron los patrones de diseño siguientes:

* Inyección de dependencias.

Se implementa con el objetivo de disminuir la dependencia fuerte entre las capas, desacoplándolas para facilitar el mantenimiento y la escalabilidad, pues la comunicación entre las capas se lleva a cabo por medio de interfaces. Ya por defecto, la implementación de este patrón lleva a la práctica el principio SOLID correspondiente a la inversión de dependencias.

* Mediador con mappers.

Este patrón, utilizado especialmente en la capa media de la arquitectura de una aplicación, fue implementado en el proyecto para aumentar el desacople de la capa de interfaz y de aplicación. Adaptándonos al patrón de inyección de dependencias implementado, registramos el mediador en el archivo Startup.cs. El paquete NuGet instalado para efectos de la implementación del patrón se llama “MediatR”. Desde el controlador, se utiliza la interfaz IMediatR para encapsular el objeto de la clase principal del patrón, y desde los endpoint se ejecutan los comandos que tienen lugar en la capa Application. Desde la capa media se asocian a los comandos las clases Handlers que implementan la lógica tomando como base los datos enviados en las propiedades del comando instanciado. Desde las clases Handlers se realiza el mapeo entre la entridad y el DTO, o viceversa según sea el caso de uso.

* Repositorio genérico.

Este patrón es comúnmente utilizado en la capa de datos y se complementa especialmente con EntityFramework. Desde allí se construye una clase repositorio como base abstracta de los métodos CRUD. De allí, extienden las clases de cada entidad para escalar la funcionalidad conforme a sus particularidades funcionales. Esto disminuye la cantidad de código fuente que se debe digitar y la redundancia en él; como resultado, tenemos un código más limpio y organizado.

**Utilización de Healthchecks**

Los Healthchecks fueron publicados mediante la URL: <https://localhost:44324/health> como se ve en la siguiente imagen:

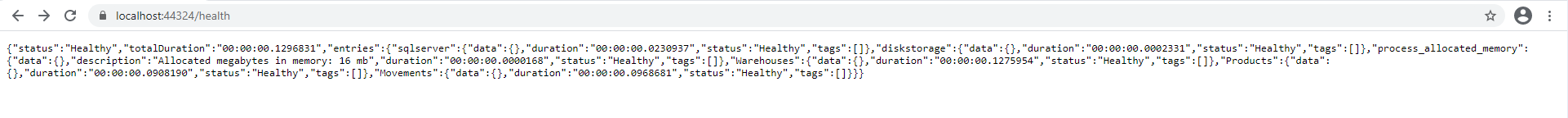


Ilustración 1. Vista desde el navegador de los resultados de los healthchecks implementados.

Se utilizan healthchecks que evalúan la disponibilidad de memoria, disco duro, conexión a la base de datos SQL Server, y de los endpoints del API.

Formateando el JSON del resultado de la evaluación de salud de la aplicación se vería así:



Ilustración 2. JSON formateado de los resultados de los healthchecks.