Desarrollo del Backend para "Buena Vida"

El objetivo es desarrollar el backend del sistema para la tienda ecológica 'Buena Vida' siguiendo la arquitectura hexagonal con TypeScript, Node.js y Express. El sistema se estructurará en capas para garantizar modularidad y facilitar la integración con el frontend en el segundo parcial.

Puntos Claves del Desarrollo

1. Modelado del dominio: Definir entidades y sus relaciones basadas en los requisitos del negocio.
2. Arquitectura: Implementar la estructura hexagonal con capas bien definidas (dominio, aplicación, infraestructura).
3. Base de datos: Uso de PostgreSQL para almacenamiento de datos.
4. Desarrollo de la API: Implementación de endpoints REST para gestión de productos, usuarios, carrito de compras, autenticación y pedidos.
5. Patrones de diseño: Aplicación de Factory, Strategy, Singleton y Repositorio.
6. Pruebas: Implementación de pruebas unitarias y de integración.
7. Documentación: Diagramas de clases, base de datos y API REST.

1. Modelado del Dominio

Se han definido las siguientes entidades principales:
- Producto: Representa un artículo de la tienda (nombre, descripción, precio, stock, categoría).
- Usuario: Cliente o administrador del sistema (nombre, email, contraseña encriptada, dirección).
- Carrito: Lista de productos seleccionados antes de realizar un pedido (relación 1:1 con usuario).
- Pedido: Orden de compra confirmada (fecha, estado, total, usuario relacionado, productos).

Relaciones:
- Un Usuario puede tener múltiples Pedidos (1:N).
- Un Pedido puede contener múltiples Productos (N:M con tabla intermedia `pedido_productos`).
- Un Carrito pertenece a un Usuario (1:1) y contiene múltiples Productos (N:M).

2. Arquitectura Hexagonal

La aplicación se divide en tres capas:

Capa de Dominio:
- Contiene las entidades del negocio y sus reglas.
- Define interfaces que otras capas deben implementar.
- No depende de infraestructura ni frameworks externos.

Capa de Aplicación:
- Contiene la lógica de negocio a través de casos de uso.
- Coordina entidades y usa los repositorios definidos en Dominio.
- Ejemplo: `PedidoService` valida stock, calcula el total y persiste el pedido.

Capa de Infraestructura:
- Implementa la persistencia con PostgreSQL.
- Incluye controladores Express para manejar las solicitudes HTTP.
- Separa adaptadores de entrada (API) y salida (BD).

3. Base de Datos

Se ha implementado una base de datos PostgreSQL con las siguientes tablas:
- `usuarios` (id, nombre, email, contraseña, dirección).
- `productos` (id, nombre, descripción, precio, stock, categoría).
- `pedidos` (id, usuario_id, fecha, estado, total).
- `pedido_productos` (pedido_id, producto_id, cantidad, precio_unitario).
- `carritos` (id, usuario_id).
- `carrito_productos` (carrito_id, producto_id, cantidad).

La conexión se maneja con un Singleton (`DatabaseSingleton.ts`) para optimizar el uso de conexiones a la base de datos.

4. API REST

Se han implementado los siguientes endpoints REST:

Gestión de Productos:
- `GET /productos`: Lista de productos con paginación y filtros.
- `GET /productos/{id}`: Obtiene detalles de un producto.
- `POST /productos`: Crea un producto (admin).
- `PUT /productos/{id}`: Actualiza un producto (admin).
- `DELETE /productos/{id}`: Elimina un producto (admin).

Gestión de Usuarios:

- `POST /usuarios`: Registro de un nuevo usuario.
- `POST /auth/login`: Autenticación y generación de token JWT.
- `GET /perfil`: Obtiene información del usuario autenticado.

Gestión de Carrito:
- `GET /carrito`: Obtiene el carrito del usuario.
- `POST /carrito`: Agrega un producto al carrito.
- `PUT /carrito`: Actualiza la cantidad de productos en el carrito.
- `DELETE /carrito/{productoId}`: Elimina un producto del carrito.

Gestión de Pedidos:
- `POST /pedidos`: Crea un pedido basado en el carrito.
- `GET /pedidos`: Lista los pedidos del usuario.
- `GET /pedidos/{id}`: Obtiene detalles de un pedido.

5. Patrones de Diseño Aplicados

- Repositorio: Interfaces de acceso a datos implementadas en `infraestructura/`.
- Factory: `PedidoFactory.ts` para creación flexible de pedidos.
- Strategy: `PagoStrategy.ts` para manejo de diferentes métodos de pago.
- Singleton: `DatabaseSingleton.ts` para optimizar la conexión a PostgreSQL.

6. Pruebas Unitarias e Integración

Se han implementado pruebas en la carpeta `tests/`:

✅ Dominio: Prueba creación de pedidos.
✅ Aplicación: Pruebas de lógica de negocio y validaciones.
✅ Infraestructura: Verificación de repositorios y consultas a BD.
✅ Presentación (API): Test de endpoints con `fetch`.
✅ Manejo de errores: Validación de respuestas adecuadas en errores.

7. Documentación

- Diagramas UML de clases y base de datos.
- Documentación de la API en formato Markdown.
- Instrucciones detalladas para instalación y ejecución.

Conclusión

El backend de 'Buena Vida' se ha desarrollado con una arquitectura modular y escalable, asegurando desacoplamiento, testabilidad y facilidad de mantenimiento. Está listo para integrarse con el frontend en el siguiente parcial.