Entrega 2: Exposición de APIs + Persistencia

Fecha: 09/10/2025

Entregables

1. ***Implementación*** de la API REST del Sistema, que incluya todos los endpoints necesarios para dar solución a los requerimientos listados.
2. ***Implementación*** de la persistencia de las entidades de dominio en una base de datos NoSQL Documental.
3. ***Implementación*** de los Test Unitarios de la capa de servicios y de la capa de dominio.
4. ***Documentación*** de la API REST.

Tecnologías a utilizar

- Todas las mencionadas en la primera iteración.
- MongoDB como Base de Datos Documental NoSQL.
- Jest como framework de Testing Unitario.
- Swagger como Herramienta de Documentación de APIs. Está permitido la utilización de alguna dependencia que genere el contenido de Swagger.

Justificaciones iniciales

1. Introducción

Esta documentación presenta la justificación de diseño para la API REST de la plataforma "Tienda Sol", desarrollada en la segunda iteración del proyecto. La API se enfoca en la gestión de pedidos, la búsqueda y visualización de productos, y la visualización de notificaciones, cumpliendo con los requerimientos funcionales especificados. Todos los datos manipulados son persistentes, asumiendo el uso de una base de datos como MongoDB para almacenar información de usuarios, productos, pedidos y notificaciones de forma no relacional

El diseño sigue el enfoque REST, utilizando métodos HTTP estándar (GET, POST, PUT, PATCH, DELETE) para operaciones CRUD, con énfasis en distintos atributos de calidad como performance y escalabilidad. Se incorpora paginación para consultas eficientes, filtros y ordenamientos para mejorar la usabilidad, y validaciones en los esquemas para garantizar la integridad de los datos. Además, se prevé la implementación de tests unitarios en la capa de servicios para validar la lógica de negocio.

La especificación OpenAPI 3.0.3 define la estructura de la API, incluyendo paths, parámetros, request/response bodies y esquemas, facilitando la generación de documentación automática (con Swagger UI) y la integración con clientes.

2. Enfoque General del Diseño

2.1 Principios REST

- Recursos y URIs: Cada recurso (pedidos, productos, notificaciones) se representa como un endpoint lógico. Por ejemplo, /pedidos para la colección de pedidos y /pedidos/{pedidoId} para instancias específicas. Esto promueve la uniformidad y la reutilización de lógica.
- Métodos HTTP:
    - POST para creación (crear pedido o producto).
    - GET para consultas (listar productos con filtros).
    - PATCH para actualizaciones parciales (cancelar o enviar un pedido, marcar notificación como leída), ya que no siempre se modifica todo el recurso.
    - PUT para actualizaciones completas (actualizar producto).
    - DELETE para eliminación (producto).
- Códigos de Respuesta HTTP: Se usan códigos estándar (200 OK, 201 Created, 400 Bad Request, 404 Not Found) para indicar el resultado de las operaciones, con cuerpos de respuesta en JSON que incluyen mensajes de error o datos relevantes.
- Persistencia: Todos los endpoints interactúan con una base de datos persistente. Por ejemplo, la creación de un pedido valida el stock de productos y actualiza el inventario atómicamente para evitar inconsistencias.
- Formato de Datos: JSON como único formato, con esquemas validados (usando bibliotecas como Zod en el backend).

2.2 Paginación, Filtros y Ordenamiento

Para manejar grandes volúmenes de datos (listas de productos o notificaciones), se implementa paginación obligatoria en endpoints de consulta múltiple:

- Parámetros page (default: 1) y limit (default: 10, max: 100) para dividir resultados.
- Respuestas incluyen metadatos como totalColecciones, totalPaginas para navegación frontend. Esto mejora la eficiencia, reduce la carga en el servidor y previene timeouts.

Los filtros y ordenamientos se aplican vía query params, permitiendo búsquedas dinámicas sin endpoints adicionales, alineado con REST.

2.3 Manejo de Errores implementados

- Errores de negocio (stock insuficiente al crear pedido) devuelven 400 con un objeto { error: "mensaje descriptivo" }.
- Errores del servidor (500) incluyen detalles para debugging, pero en producción se ocultan.

- Validaciones en request bodies aseguran datos requeridos y tipos correctos.

2.4 Tests Unitarios

Se implementarán tests unitarios en la capa de servicios (usando Jest) para cubrir:

- Lógica de validación de stock en creación de pedidos.
- Aplicación de filtros y paginación en servicios de productos.
- Transiciones de estado en pedidos (ej: de PENDIENTE a CANCELADO).
- Generación y marcado de notificaciones. Esto garantiza robustez y facilita el mantenimiento.
- Y otras cuestiones que tienen que ver con la implementación de dominio

3. Diseño de Endpoints por Módulo

3.1 Gestión de Pedidos

Los requerimientos exigen el ciclo de vida completo de un pedido: creación (con validación de stock), cancelación (antes de envío), consulta de historial por usuario, y marcado como enviado por el vendedor.

- **Creación (POST /pedidos)**:
  - Justificación: Se encarga de validar el stock disponible por ítem (restando del stock del producto si es suficiente), actualiza totalVendido en productos e inicializa historial de estados con PENDIENTE. Requiere compradorId, items, moneda y direccionEntrega para completitud. Respuesta incluye pedidoId para tracking inmediato.
  - Alineación: Cumple con "Creación de un pedido, validando el stock disponible".
- **Consulta General (GET /pedidos) y por Usuario (GET /pedidos/usuario/{usuarioId})**:
  - Justificación: El endpoint general lo pensamos para admins o vendedores; el GET por usuario para historial personal (resumido para privacidad). Usa PedidoResumido para ocultar detalles sensibles como la dirección completa.
  - Alineación: Cumple con "Consulta del historial de pedidos de un usuario".
- **Cancelación (PATCH /pedidos/{pedidoId}/cancelar)**:
  - Justificación: Sólo se permite si el estado es PENDIENTE o CONFIRMADO (no enviado). Requiere compradorId para autorización. Actualiza historial con motivo y restaura stock. PATCH para cambio parcial de estado.
  - Alineación: Cumple con "Cancelación de un pedido antes de que haya sido enviado".

- **Marcado como Enviado (PATCH /pedidos/{pedidoId}/enviar)**:
    - Justificación: Sólo se permite si no está cancelado. Requiere vendedorId para autorización. Transición a ENVIADO en historial. PATCH para actualización mínima.
    - Alineación: Cumple con "Marcado de un pedido como enviado por parte del vendedor".

Además, para permitir la persistencia del pedido en MongoDB se realizó un esquema Pedido que incluye historialEstados para auditar cambios, y enums para estados estandarizados, asegurando consistencia.

3.2 Búsqueda y Visualización de Productos

Se requiere listar productos de un vendedor con filtros (nombre, categoría, descripción, rango de precios), paginación y ordenamiento (precio asc/desc, más vendido).

- **Creación (POST /productos), Actualización (PUT /productos/{id}) y Eliminación (DELETE /productos/{id})**:
    - Justificación: Están asociados a usuarioId (vendedor). Incluyen stock, totalVendido (inicial 0, actualizado en pedidos) y activo para visibilidad. PUT reemplaza completamente; DELETE soft-delete o delete lógico (set activo: false) para preservar datos históricos.
    - Alineación: Soporte base para gestión de inventario.
- **Listado General (GET /productos) y por Vendedor (GET /productos/vendedor/{vendedorId})**:
    - Justificación: Esta permite los siguientes filtros via query: nombre, descripcion (búsqueda parcial con LIKE o regex), categoría (exacta por ID), precioMin/Max (rango numérico). Ordenamiento: sortParam con enums para SQL/Mongo sort (e.g., {precio: 1} para asc). Paginación con offset/limit. mas_vendidos ordena por totalVendido descendente. Respuesta paginada con ProductoRespuesta (incluye timestamps para auditoría).
    - Alineación: Cumple con "Listar los productos de un vendedor en particular, con filtros (término de búsqueda, rango de precios), paginación y ordenamiento".

Esto permite búsquedas eficientes, indexando campos como título, descripción y precio en la BD.

3.3 Visualización de Notificaciones

Se implementan notificaciones para confirmación/cancelación/envío de pedidos, con endpoints para sin leer, leídas y marcadas.

- **Sin Leer (GET /notificaciones/unread/{usuarioId}) y Leídas (GET /notificaciones/read/{usuarioId})**:
    - Justificación: Son filtradas por usuarioId y leída: false/true. Utilizan paginación para listas largas. tipo enum para categorizar (como "pedido" para confirmaciones).
    - Alineación: Cumple con "Endpoint para obtener la lista de notificaciones sin leer/leídas de un usuario".
- **Marcado como Leída (PATCH /notificaciones/{id}/read)**:
    - Justificación: Actualiza el campo de leída: true y fechaLectura. Se utiliza PATCH para cambio atómico. No requiere body, solo ID.
    - Alineación: Cumple con "Endpoint para marcar una notificación como leída".

El esquema Notificación incluye título, mensaje y timestamps. Se generan automáticamente en eventos

4. Esquemas y Modelos de Datos

Los esquemas de la API se implementan utilizando Mongoose para MongoDB, lo que permite una modelación flexible de documentos NoSQL con soporte para referencias (populate), validaciones integradas y optimizaciones de rendimiento mediante índices. Esta elección se justifica por la naturaleza no relacional de MongoDB, que facilita el manejo de arrays embebidos (como ítems de pedidos, fotos de productos) y subdocumentos (como historial de estados), reduciendo joins complejos y mejorando la velocidad de consultas.

4.1 Modelo de Usuario

El modelo **Usuario** representa a compradores y vendedores, sirviendo como base para referencias en otros modelos (vendedor en productos, comprador en pedidos).

- Estructura Principal:
    - **nombre**: String requerido y trimmeado para limpieza de datos.
    - **email**: String único, requerido, trimmeado, en minúsculas y validado con regex para formato estándar (e.g., evita emails inválidos como "user@invalid").
    - **telefono**: String opcional, trimmeado para normalización.
    - **dirección**: Subobjeto embebido con calle, ciudad y codigoPostal (todos opcionales y trimmeados), permitiendo almacenamiento flexible de datos geográficos sin un modelo separado.
    - **Activo**: Booleano default true, para soft-delete o desactivación de cuentas.
    - **fechaRegistro**: Date default Date.now, para tracking de onboarding.
- Justificación: Este esquema es minimalista pero robusto, enfocándose en datos esenciales para autenticación y perfiles. La validación de email

previene errores comunes, y el subobjeto dirección simplifica el modelo sin sacrificar extensibilidad (puede expandirse para lat/lon en futuras iteraciones).

4.2 Modelo de Categoría

El modelo **Categoría** es un esquema simple para clasificar productos, permitiendo referencias múltiples.

- Estructura Principal:
    - **nombre**: String requerido y trimmeado, para un identificador único y legible.
- Justificación: Como catálogo estático, se mantiene liviano para facilitar la creación y mantenimiento manual/administrativo. Se usa en arrays de referencias en productos (categorías: [ObjectId ref 'Categoria']), permitiendo productos multi-categorías sin duplicación de datos. Esto soporta filtros por categoría en endpoints de productos, con queries eficientes vía populate o agregaciones.

4.3 Modelo de Producto

El modelo **Producto** gestiona el inventario, con énfasis en visibilidad, ventas y filtros de búsqueda.

- Estructura Principal:
    - **vendedor**: ObjectId ref 'Usuario', requerido, para asociar productos a un vendedor específico.
    - **titulo**: String requerido y trimmeado, para búsquedas por nombre.
    - **descripción**: String opcional y trimmeado, para búsquedas textuales detalladas.
    - **Categorías**: Array de ObjectId ref 'Categoría', para clasificación flexible (un producto puede tener múltiples).
    - **precio**: Number requerido, mínimo 0, para validación económica.
    - **moneda**: String enum ["PESO_ARG", "DOLAR_USA", "REAL"], requerido, para soporte multi-moneda regional (alineado con pedidos).
    - **stock**: Number default 0, mínimo 0, para control de inventario.
    - **totalVendido**: Number default 0, mínimo 0, para métricas de popularidad (actualizado en creación de pedidos).
    - **Fotos**: Array de strings (URLs), para multimedia flexible.
    - **Activo**: Booleano default true, para soft-delete o pausar ventas.
- Justificación: El esquema soporta los requerimientos de búsqueda y visualización: campos como título, descripción y categorías habilitan filtros parciales/exactos; precio y moneda permiten rangos y ordenamientos; stock y totalVendido validan pedidos y ordenan por "más vendido". Arrays para categorías y fotos ofrecen flexibilidad sin normalización excesiva. En OpenAPI, ProductoCrear mapea inputs a este modelo, y ProductoRespuesta

expone outputs enriquecidos con timestamps y totalVendido. Soft-delete via activo preserva historial de ventas.

4.4 Modelo de Pedido

El modelo **Pedido** captura el ciclo de vida completo, con subesquemas para ítems e historial.

- Subesquemas:
    - **ItemPedidoSchema**: Incluye **productoId** (ObjectId ref 'Producto', requerido), **cantidad** (Number requerido) y **precioUnitario** (Number requerido).

      Justificación: Desglosa pedidos en ítems atómicos para calcular totales y validar stock individualmente, con precio fijo al momento de compra (evita inflación).

    - **HistorialEstadoSchema**: Incluye **fecha** (Date default Date.now), **estado** (String enum ["PENDIENTE", "CONFIRMADO", "EN_PREPARACION", "ENVIADO", "ENTREGADO", "CANCELADO"], requerido), **quien** (ObjectId ref 'Usuario', requerido) y **motivo** (String opcional).

      Justificación: Proporciona auditoría inmutable de transiciones, rastreando quién (comprador/vendedor) y por qué (e.g., cancelación por stock bajo), esencial para disputas o analíticas.

- Estructura Principal:
    - **compradorId**: String ref 'Usuario', requerido (nota: usa String en schema, pero idealmente ObjectId para consistencia).
    - **items**: Array de ItemPedidoSchema, para múltiples productos.
    - **moneda**: String enum ["PESO_ARG", "DOLAR_USA", "REAL"], requerido, consistente con productos.
    - **direccionEntrega**: Object requerido (embebido, con campos como calle, ciudad; extensible para lat/lon).
    - **estado**: String enum (mismo que historial), default "PENDIENTE".
    - **fechaCreacion**: Date default Date.now.
    - **historialEstados**: Array de HistorialEstadoSchema, inicializado con el estado inicial.
- Justificación: Este esquema embebido reduce complejidad de relaciones, permitiendo queries atómicas (e.g., obtener pedido completo con items populated). Enums estandarizan estados para lógica de negocio (e.g., solo cancelar si PENDIENTE). El total se calcula dinámicamente en servicios (suma de ítems), no almacenado para evitar inconsistencias.

4.5 Modelo de Notificación

El modelo **Notificacion** maneja alertas push/in-app para eventos clave.

- Estructura Principal:
    - **usuarioDestinoId**: ObjectId ref 'Usuario', requerido, con índice para queries rápidas.
    - **mensaje**: String requerido y trimmeado (nota: en OpenAPI incluye **título** y **tipo** enum; aquí se simplifica a mensaje único, pero puede expandirse).
    - **leída**: Booleano requerido, default false, con índice para filtrado eficiente.
    - **fechaCreacion**: Date default Date.now.
    - **fechaLectura**: Date default null, para tracking de interacción.

5. Conclusión

Este diseño de API REST cumple integralmente con los requerimientos de la iteración, priorizando usabilidad, eficiencia y mantenibilidad. La persistencia asegura durabilidad de datos, mientras que paginación/filtros optimizan performance. La implementación de tests unitarios en servicios validará la lógica crítica. La especificación OpenAPI facilita el desarrollo frontend y la colaboración, posicionando la plataforma para expansiones futuras como pagos o reseñas. Si se requieren ajustes, se pueden iterar basados en feedback.