Universidad Católica del Norte
Facultad de Ingeniería y Ciencias Geológicas
Departamento de Ingeniería de Sistemas y Computación

Introducción al Desarrollo Web/Móvil
Taller – E-Commerce
Requerimientos del Backend

Profesor: Jorge Rivera Mancilla.
Ayudantes: Fernando Chávez Briceño y Ernes Fuenzalida Tello.

1. Configuración y estructura del proyecto

El sistema debe contar con una estructura de código organizada y modular, que garantice la mantenibilidad y escalabilidad del desarrollo. Además, debe incluir configuraciones esenciales para su correcto funcionamiento y una documentación clara que facilite su instalación y uso.

1.1. Organización del código fuente

El código fuente debe seguir una arquitectura modular basada en separación de responsabilidades, permitiendo que cada capa del sistema tenga una función específica. Deben cumplirse los siguientes requisitos:

1. Estructura de directorios organizada: El código debe distribuirse en, al menos, los siguientes directorios específicos según su función:
    a. Controllers: Contiene los controladores que manejan las solicitudes HTTP.
    b. Services: Implementa la lógica de negocio de la aplicación.
    c. Repositories: Gestiona el acceso a la base de datos.
    d. Models: Define las entidades y estructuras de datos utilizadas en la aplicación.
    e. DTOs: Contiene los objetos de transferencia de datos que regulan la comunicación entre capas.
    f. Data: Gestiona la base de datos, incluyendo el contexto de Entity Framework Core, migraciones y Seeders.
    g. Interfaces: Contiene las interfaces para la inyección de dependencias, promoviendo el desacoplamiento y reutilización del código.
2. Separación de responsabilidades: Ninguna capa del sistema debe contener lógica ajena a su función.
3. Convenciones de nomenclatura: Los nombres de clases, métodos y archivos deben seguir las convenciones de C# y ASP.NET Core.

1.2. Configuración del archivo "appsettings.json"

El sistema debe incluir un archivo de configuración centralizado, "appsettings.json", que defina los parámetros esenciales de la aplicación. Deben configurarse las siguientes secciones dentro del archivo:

1. Conexión a la base de datos SQLite (revisar sección 2.1).

2. Parámetros para la autenticación con JSON Web Tokens (revisar sección 3.2).
3. Credenciales de acceso a Cloudinary (revisar sección 7.1).
4. Registro de logs con Serilog (revisar sección 6.3).
5. Configuración de CORS (revisar sección 4.3).

1.3. Documentación del proyecto en "README.md"

El sistema debe incluir un archivo de documentación README.md, ubicado en la raíz del repositorio, proporcionando instrucciones detalladas sobre la instalación y uso de la API. Deben incluirse los siguientes apartados en la documentación:

1. Descripción del proyecto: Breve explicación sobre la funcionalidad y propósito de la API REST.
2. Requisitos previos: Listado de herramientas necesarias para ejecutar el sistema.
3. Instrucciones de instalación: Pasos detallados para clonar el repositorio, restaurar las dependencias y configurar la base de datos.
4. Ejecutar la API: Comandos necesarios para iniciar la aplicación en un entorno local.

El archivo "README.md" debe mantenerse actualizado conforme se realicen cambios en el sistema y debe estar correctamente formateado para facilitar su lectura.

2. Base de datos y acceso a la información

El sistema debe utilizar SQLite como motor de base de datos, asegurando una gestión eficiente y estructurada de la información. La implementación debe realizarse mediante Entity Framework Core, garantizando la abstracción de consultas SQL y la integración con la arquitectura del sistema.

2.1. Configuración y conexión con SQLite

La API REST debe utilizar SQLite como base de datos relacional. Deben cumplirse los siguientes requisitos:

1. La cadena de conexión a SQLite debe definirse en el archivo "appsettings.json" en el campo "ConnectionStrings".
2. La configuración de la base de datos debe establecerse en el arranque de la aplicación a través de la inyección de dependencias.
3. Se debe garantizar que el sistema pueda generar y utilizar automáticamente la base de datos al iniciar la aplicación, sin necesidad de configuraciones manuales adicionales.
4. No deben utilizarse consultas SQL directas dentro del código; todas las operaciones de base de datos deben manejarse a través de Entity Framework Core.

2.2. Migraciones con Entity Framework Core

El sistema debe utilizar migraciones de Entity Framework Core para definir, modificar y mantener la estructura de la base de datos. Deben cumplirse los siguientes requisitos:

1. Las migraciones deben permitir la creación automática de la base de datos al iniciar la aplicación, si es que esta no existiese.
2. Todas las modificaciones en el esquema de la base de datos deben registrarse mediante migraciones.
3. Las migraciones deben ejecutarse utilizando los comandos proporcionados por .NET CLI, garantizando la replicabilidad del esquema de datos en distintos entornos.

2.3. Carga inicial de datos con Seeders

Para evitar la necesidad de ingresar datos manualmente en entornos de desarrollo y prueba, el sistema debe incluir Seeders que permitan poblar la base de datos con información predeterminada. Deben cumplirse los siguientes requisitos:

1. Los Seeders deben generar datos iniciales para todas las entidades clave del sistema.
2. La ejecución de Seeders debe realizarse automáticamente al iniciar la aplicación, sin requerir intervención manual.
3. El Seeder debe verificar si los datos ya existen antes de insertarlos, evitando duplicaciones en cada ejecución.

2.4. Implementación del patrón Repository

El sistema debe utilizar el patrón Repository para gestionar el acceso a la base de datos, desacoplando la lógica de negocio de la capa de persistencia. Deben cumplirse los siguientes requisitos:

1. Cada entidad gestionada por la base de datos debe contar con su correspondiente repositorio encargado de manejar las operaciones CRUD.
2. Los repositorios deben ser inyectados como dependencias, evitando la instanciación manual de objetos de acceso a datos.
3. No deben realizarse accesos directos a la base de datos desde otras capas; todas las consultas deben ser ejecutadas a través de los repositorios.
4. Los métodos en los repositorios deben ser genéricos y reutilizables, evitando la repetición de lógica en distintas partes del sistema.

2.5. Implementación del patrón Unit of Work

El sistema debe implementar el patrón Unit of Work para gestionar transacciones de base de datos de manera eficiente y garantizar la coherencia de la información. Deben cumplirse los siguientes requisitos:

1. La API REST debe contar con una clase centralizada que administre los repositorios y permita ejecutar múltiples operaciones en una misma transacción.
2. La confirmación de cambios en la base de datos debe realizarse exclusivamente a través de Unit of Work, evitando llamadas directas al guardado en la base de datos en diferentes partes del código.

3. Debe garantizarse que una transacción sólo se confirme si todas las operaciones involucradas han sido ejecutadas correctamente; en caso contrario, la transacción debe revertirse para evitar inconsistencias.
4. Unit of Work debe ser inyectado como dependencia en los servicios de la API.

3. Autenticación y control de acceso

El sistema debe implementar un mecanismo seguro de autenticación y control de acceso, asegurando que solo los usuarios autorizados puedan interactuar con funcionalidades protegidas de la API REST. La autenticación debe realizarse mediante JSON Web Tokens y la gestión de usuarios debe implementarse con Identity de .NET.

3.1. Gestión de usuarios con Identity

El sistema debe utilizar Identity de .NET para la administración de usuarios, roles y autenticación. Deben cumplirse los siguientes requisitos:

1. Los usuarios deben almacenarse en la base de datos utilizando las tablas y esquemas generados por Identity.
2. La API debe permitir registro de nuevos usuarios, asegurando que el correo electrónico sea único en la plataforma.
3. La API debe permitir inicio de sesión, validando las credenciales ingresadas y generando un JSON Web Token en caso de éxito.
4. Las contraseñas deben almacenarse utilizando un algoritmo de hashing seguro con salt, evitando el almacenamiento en texto plano.
5. Debe permitirse la actualización de datos personales de los usuarios autenticados.

3.2. Autenticación con JSON Web Tokens

El sistema debe utilizar JSON Web Tokens como mecanismo de autenticación para proteger las rutas de la API. Deben cumplirse los siguientes requisitos:

1. El token debe generarse cuando un usuario se autentica correctamente.
2. El token debe incluir información esencial del usuario, como su identificador y rol, sin exponer datos sensibles.
3. La firma del token debe realizarse con una clave segura, la cual debe almacenarse en el archivo "appsettings.json" (revisar sección 1.2).
4. La API debe validar la autenticidad del token en cada solicitud a un recurso protegido.
5. El token debe contar con un tiempo de expiración, evitando accesos indefinidos a los recursos de la API.

El sistema debe rechazar cualquier solicitud que no contenga un token válido, retornando un código de estado 401 Unauthorized (revisar sección 6.2).

3.3. Control de acceso por roles

El sistema debe implementar un modelo de control de acceso basado en roles, asegurando que cada usuario pueda acceder únicamente a los recursos permitidos según su perfil. Deben definirse al menos los siguientes roles dentro del sistema: Cliente autenticado y Administrador. Deben cumplirse los siguientes requisitos:

1. Cada usuario debe estar asociado a un único rol dentro del sistema.
2. El rol de un usuario debe determinar qué recursos puede acceder dentro de la API REST.
3. Los permisos asociados a cada rol deben definirse en la configuración de la API y no directamente en los controladores.

El sistema debe restringir el acceso a los recursos protegidos, retornando un código de estado 403 Forbidden cuando un usuario intenta acceder a una funcionalidad para la cual no tiene permisos (revisar sección 6.2).

3.4. Políticas de autorización y permisos por endpoint

El sistema debe definir políticas de autorización para establecer reglas de acceso a los endpoints protegidos. Deben cumplirse los siguientes requisitos:

1. Las políticas de autorización deben definir qué roles tienen acceso a cada recurso.
2. La API debe implementar restricciones adicionales según condiciones específicas, como la propiedad de un recurso.
3. Los controladores deben aplicar las políticas de autorización de manera centralizada, evitando validaciones dispersas en el código.
4. Se deben establecer restricciones claras para que los usuarios no puedan modificar o eliminar información de otros usuarios sin los permisos adecuados.

4. Seguridad y protección de la API

El sistema debe implementar mecanismos de seguridad que garanticen la protección de la información y la integridad de los datos. Se deben aplicar controles para evitar accesos no autorizados, ataques comunes y vulnerabilidades en la gestión de datos.

4.1. Configuración de seguridad y encriptación de datos

El sistema debe garantizar que los datos sensibles sean protegidos mediante técnicas de encriptación y almacenamiento seguro. Deben cumplirse los siguientes requisitos:

1. Cifrado de contraseñas (revisar sección 3.1).
2. Transmisión segura de datos: No deben enviarse credenciales o información sensible en las URL, solo en el cuerpo de las solicitudes.
3. Protección de información confidencial:

a. No deben exponerse claves de acceso, tokens o información privada en las respuestas de la API.
b. Las credenciales y valores sensibles deben gestionarse mediante variables de entorno y no deben estar incluidas en el repositorio del código fuente (revisar sección 4.1).

4.2. Prevención de ataques

El sistema debe contar con mecanismos de protección contra ataques comunes que puedan comprometer la seguridad de la API. Deben implementarse las siguientes medidas:

1. Para prevenir la inyección SQL, se debe usar siempre Entity Framework Core (revisar sección 2.1-2.2), evitando el uso de consultas en texto plano.
2. Protección contra Cross-Site Scripting (XSS): Todas las entradas de usuario deben ser sanitizadas y validadas para evitar la ejecución de código malicioso en las respuestas de la API.

4.3. Restricción de acceso con CORS

El sistema debe implementar Cross-Origin Resource Sharing (CORS) para controlar qué dominios pueden acceder a la API REST, evitando solicitudes no autorizadas desde orígenes desconocidos. Deben cumplirse los siguientes requisitos:

1. Configuración explícita de CORS en la API, definiendo los orígenes permitidos en la sección "CorsSettings" del archivo "appsettings.json" (revisar sección 1.2).
2. Restringir los métodos HTTP permitidos, evitando solicitudes innecesarias o potencialmente peligrosas desde clientes no autorizados.
3. Definir las cabeceras autorizadas, permitiendo solo aquellas que sean estrictamente necesarias para la comunicación con la API.

5. Estructura y manipulación de datos

5.1. Uso de Data Transfer Objects

El sistema debe implementar Data Transfer Objects (DTOs) para garantizar una comunicación estructurada entre la API REST y el cliente, evitando la exposición directa de las entidades de la base de datos. Deben cumplirse los siguientes requisitos:

1. Separación entre entidades y DTOs: Los DTOs deben ser utilizados en todas las interacciones con la API, evitando el uso directo de entidades de base de datos en los controladores.
2. Definición de DTOs específicos para cada operación, aplicando validaciones con Data Annotations (revisar sección 6.1).
3. Validaciones en los DTOs: Deben aplicarse restricciones en los DTOs de entrada mediante Data Annotations, asegurando que los datos ingresados sean correctos antes de ser procesados.

5.2. Definición de respuestas de la API

La API REST debe proporcionar respuestas estandarizadas para garantizar la consistencia en la comunicación con el cliente. Deben cumplirse los siguientes requisitos:

1. Formato de respuesta JSON: Todas las respuestas de la API deben seguir una estructura clara y homogénea en formato JSON.
2. Códigos de estado HTTP adecuados:
    a. 200 OK: Respuesta exitosa.
    b. 201 Created: Recurso creado correctamente.
    c. 400 Bad Request: Error en la solicitud debido a datos incorrectos.
    d. 401 Unauthorized: Acceso no autorizado por falta de autenticación.
    e. 403 Forbidden: Acceso prohibido por restricciones de permisos.
    f. 404 Not Found: Recurso no encontrado.
    g. 500 Internal Server Error: Error interno del servidor.
3. Mensajes de error descriptivos: En caso de error, la API debe proporcionar mensajes claros y estructurados que indiquen la causa del problema y las acciones necesarias para corregirlo.

El sistema debe garantizar que todas las respuestas sean coherentes y fáciles de interpretar por los clientes de la API.

5.3. Filtrado, búsqueda y ordenamiento de información

El sistema debe proporcionar mecanismos para la búsqueda y/o filtrado de datos, permitiendo a los clientes obtener información específica de manera eficiente. Deben cumplirse los siguientes requisitos:

1. Búsqueda por parámetros de consulta (Query Parameters): La API debe permitir buscar recursos utilizando parámetros en la URL.
2. Ordenación de resultados: La API debe permitir ordenar los resultados en orden ascendente o descendente según su criterio.

5.4. Implementación de paginación

Para mejorar la eficiencia y reducir la carga en el servidor, la API debe implementar paginación en las respuestas que devuelvan listados de algún recurso. Deben cumplirse los siguientes requisitos:

1. Definición del número de elementos por página: El sistema debe permitir que el cliente especifique la cantidad de elementos por página o utilizar un valor por defecto.
2. Parámetros de paginación:
    a. page: Número de página a obtener.
    b. limit: Cantidad de elementos por página.
3. Inclusión de metadatos en la respuesta:
    a. Número total de registros disponibles.

b. Número total de páginas.
c. Enlaces de navegación para acceder a páginas anteriores o siguientes.

6. Validaciones y manejo de errores

El sistema debe garantizar que los datos procesados por la API REST sean válidos, estructurados y seguros, evitando errores en la ejecución y asegurando respuestas claras ante situaciones inesperadas. Se deben implementar validaciones estrictas en la entrada de datos, estandarizar las respuestas de error y registrar los eventos relevantes para la depuración y monitoreo del sistema.

6.1. Validaciones con Data Annotations

El sistema debe aplicar validaciones en los DTOs para garantizar que los datos recibidos sean correctos antes de ser procesados. Las validaciones deben implementarse utilizando Data Annotations estándar y, cuando sea necesario, validaciones personalizadas para casos específicos del negocio. Deben cumplirse los siguientes requisitos:

1. Campos obligatorios: Los datos esenciales para la operación del sistema deben ser validados para evitar valores nulos o vacíos.
2. Formatos específicos: Se deben definir validaciones para datos estructurados como correos electrónicos, números de teléfono y fechas, asegurando su correcto formato y restricciones de valores.
3. Restricciones de valores: Se deben establecer límites en valores numéricos, fechas y cadenas de texto, según los requerimientos funcionales de la API.
4. Validaciones personalizadas: Para reglas de negocio específicas, deben implementarse validaciones a través de atributos personalizados, asegurando la coherencia de los datos.

El sistema debe rechazar cualquier solicitud que no cumpla con las validaciones establecidas.

6.2. Respuestas de error estandarizadas

La API debe proporcionar respuestas de error estructuradas, asegurando que los clientes puedan identificar fácilmente la causa de los problemas y tomar las acciones necesarias.Deben cumplirse los siguientes requisitos:

1. Formato estándar de respuesta de error:
    a. status: Código de estado HTTP correspondiente al error.
    b. error: Nombre del error.
    c. message: Descripción breve del problema.
    d. details: Lista opcional de detalles adicionales sobre el error.
2. Códigos de estado HTTP (revisar sección 5.2).
3. Manejo centralizado de excepciones:
    a. Todas las excepciones deben ser capturadas y procesadas por un middleware global de manejo de errores.

b. No deben devolverse detalles técnicos internos en las respuestas de error.

El sistema debe garantizar que los errores sean informativos sin comprometer la seguridad ni exponer información sensible.

6.3. Registro de eventos y logs con Serilog

El sistema debe implementar Serilog como mecanismo de registro de eventos y errores, asegurando la trazabilidad de las operaciones y facilitando la detección de fallos. Deben cumplirse los siguientes requisitos:

1. Configuración de Serilog en "appsettings.json":
    a. Los niveles de registro deben incluir Information, Warning y Error.
    b. Los logs deben almacenarse en un archivo de texto estructurado para su posterior análisis.
2. Registro de eventos clave:
    a. Autenticación y autorización: Intentos de inicio de sesión, accesos exitosos y fallidos.
    b. Operaciones como la creación, modificación y eliminación de recursos.
    c. Errores en la API: Excepciones capturadas con información sobre el origen del problema.
3. Formato estructurado de logs:
    a. Los registros deben incluir fecha y hora, tipo de evento y detalles del usuario o recurso afectado.

El sistema debe garantizar que los registros de eventos sean accesibles para auditoría y depuración sin comprometer la privacidad ni exponer información sensible.

7. Gestión de imágenes con Cloudinary

El sistema debe integrar Cloudinary como servicio de almacenamiento y gestión de imágenes, asegurando una manipulación eficiente de los archivos subidos por los usuarios. Se deben definir mecanismos para la configuración de credenciales, validación de archivos, recuperación de imágenes y eliminación de contenido en la plataforma.

7.1. Configuración y credenciales de Cloudinary

La API REST debe establecer una integración segura con Cloudinary mediante credenciales configuradas en el archivo "appsettings.json". Deben cumplirse los siguientes requisitos:

1. Almacenar las credenciales de Cloudinary en variables de entorno y referenciarlas en "appsettings.json" bajo el campo "CloudinarySettings".
2. Definir parámetros de conexión en la configuración de la API, incluyendo:
    a. CloudName: Nombre de la cuenta en Cloudinary.
    b. ApiKey: Clave de acceso para autenticación.
    c. ApiSecret: Secreto de acceso para operaciones seguras.

3. Inyectar la configuración en los servicios de la API, garantizando una gestión segura de las credenciales.

7.2. Subida y validación de imágenes

El sistema debe permitir la subida de imágenes a Cloudinary. Deben cumplirse los siguientes requisitos:

1. Validaciones en la subida de archivos:
    a. Solo se deben aceptar formatos de imagen compatibles con Cloudinary (jpg, png, jpeg, webp).
    b. Se debe establecer un límite de tamaño máximo para cada archivo, evitando el almacenamiento de imágenes innecesariamente grandes.
2. Organización del almacenamiento en Cloudinary:
    a. Las imágenes deben guardarse en carpetas organizadas por tipo de recurso (productos, usuarios, etc.).
    b. Cada imagen subida debe retornar una URL pública que será almacenada en la base de datos para su posterior recuperación.

El sistema debe garantizar que la subida de imágenes sea eficiente y segura, evitando la sobrecarga de almacenamiento con archivos no válidos.

7.3. Acceso y almacenamiento de imágenes

El sistema debe permitir recuperar y mostrar imágenes almacenadas en Cloudinary, asegurando que los usuarios puedan visualizar correctamente el contenido. Deben cumplirse los siguientes requisitos:

1. Almacenamiento de URLs en la base de datos: La API no debe almacenar archivos de imagen en la base de datos, solo las URLs proporcionadas por Cloudinary.
2. Recuperación de imágenes: Los endpoints de la API deben incluir las URLs de las imágenes en las respuestas al cliente.

7.4. Eliminación y actualización de imágenes

El sistema debe permitir la eliminación y actualización de imágenes en Cloudinary cuando un producto o recurso asociado sea modificado o eliminado. Deben cumplirse los siguientes requisitos:

1. Eliminación de imágenes en Cloudinary:
    a. Al eliminar un recurso con imagen asociada, la API debe eliminar automáticamente la imagen almacenada en Cloudinary.
    b. Si la imagen ya no existe en Cloudinary, el sistema debe manejar la respuesta sin generar errores críticos.

2. Actualización de imágenes:
    a. Cuando un usuario actualice la imagen de un producto, la API debe eliminar la imagen anterior en Cloudinary antes de almacenar la nueva.
    b. La URL de la nueva imagen debe ser actualizada en la base de datos sin afectar otras propiedades del recurso.

El sistema debe asegurar que las imágenes se mantengan sincronizadas con los datos de la API, evitando archivos huérfanos en Cloudinary.

8. Implementación de endpoints REST

El sistema debe proporcionar una API REST estructurada y bien definida, garantizando una comunicación eficiente entre el backend y los clientes. Se deben seguir los principios RESTful, definiendo rutas claras, métodos HTTP adecuados y operaciones coherentes para la gestión de recursos.

8.1. Definición de rutas y métodos HTTP

La API debe seguir una convención de rutas estandarizada, asegurando que los endpoints sean intuitivos y fáciles de consumir. Deben cumplirse los siguientes requisitos:

1. Estructura de rutas basada en recursos:
    a. Se debe utilizar una convención definida por el equipo para las rutas que representan colecciones de recursos (por ejemplo: /products o /product).
    b. Se deben usar identificadores en la URL para acceder a elementos específicos (ejemplo: /products/{id}).
2. Métodos HTTP según la operación:
    a. GET: Obtener información de uno o varios recursos.
    b. POST: Crear un nuevo recurso.
    c. PUT: Actualizar completamente un recurso existente.
    d. PATCH: Actualizar parcialmente un recurso.
    e. DELETE: Eliminar un recurso de la base de datos.
3. Uso de Query Parameters para opciones adicionales: La API debe permitir filtros, ordenación y paginación utilizando parámetros en la URL (ejemplo: /products?category=electronics&sort=price_desc).

El sistema debe garantizar que las rutas sean claras, predecibles y sigan una estructura RESTful estándar.

9. Pruebas mediante Postman

El sistema debe contar con un conjunto de pruebas estructuradas y automatizadas mediante Postman, asegurando la validación del correcto funcionamiento de la API REST en distintos escenarios. Las pruebas deben estar organizadas en dos categorías: pruebas individuales por recurso y flujos de prueba automatizados, garantizando la cobertura de validaciones unitarias y la correcta interacción entre múltiples endpoints.

9.1. Pruebas individuales por recurso

El sistema debe proporcionar una Postman Collection con pruebas unitarias para cada recurso de la API, organizadas de manera estructurada. Deben cumplirse los siguientes requisitos:

1. Estructura organizada: Las pruebas deben agruparse en carpetas según los recursos gestionados.
2. Cobertura CRUD: Se deben incluir solicitudes predefinidas para crear, consultar, actualizar y eliminar cada recurso, verificando que las peticiones sean correctas y cumplan con las validaciones esperadas.
3. Uso de variables de entorno: La colección debe emplear variables globales para manejar URLs, tokens de autenticación u otros valores dinámicos, evitando configuraciones manuales en cada solicitud.
4. Validación automática de respuestas: Se deben incluir scripts post-request para validar automáticamente los códigos de estado y la estructura de la respuesta, asegurando que los datos retornados cumplan con los formatos esperados.
5. Manejo de datos temporales: Se deben almacenar en variables globales los datos generados en una solicitud para su reutilización en pruebas posteriores dentro del mismo recurso.

9.2. Flujos de prueba automatizados

El sistema debe definir flujos de prueba automatizados para validar la interacción entre múltiples endpoints. Estos flujos deben ser diseñados en conjunto con el equipo de ayudantes de la asignatura, pero deben cumplir con los siguientes requisitos:

1. Pruebas de procesos completos: Se deben incluir flujos que simulen interacciones complejas del sistema, como:
    a. Autenticación y gestión de sesiones: Registro, inicio de sesión y validación de acceso con un JSON Web Token.
    b. Gestión de usuarios y roles: Creación de usuarios con distintos permisos y validación del acceso restringido.
    c. Proceso de compra: Agregar productos al carrito, generar un pedido, actualizar su estado y finalizar la compra.
    d. Manejo de imágenes con Cloudinary: Subida, obtención y eliminación de imágenes asociadas a un recurso.
2. Ejecución secuencial de pruebas: Cada solicitud debe depender de la anterior, utilizando scripts pre-request y/o post-request para almacenar datos generados en pruebas anteriores y reutilizarlos en pasos siguientes.
3. Validaciones automáticas en cada etapa: Se deben validar los códigos de respuesta, la estructura de los datos y las restricciones de acceso en cada paso del flujo.

10. Control de versiones y flujo de trabajo en Git

El sistema debe utilizar Git y GitHub para gestionar el control de versiones, asegurando un desarrollo estructurado y colaborativo. Se deben definir convenciones de trabajo, estructura de ramas, validaciones de código y reglas de commits para mantener la estabilidad del proyecto.

10.1. Organización del repositorio y ramas en GitHub

El código fuente debe estar alojado en un repositorio en GitHub, organizado de acuerdo con las mejores prácticas de control de versiones. Deben cumplirse los siguientes requisitos:

1. Estructura de ramas:
    a. main: Contiene la versión estable y lista para revisión formal.
    b. development: Rama de integración donde se prueban nuevas funcionalidades antes de ser fusionadas en main.
    c. features/{nombre}: Ramas temporales para desarrollar nuevas características. Deben crearse a partir de development y fusionarse en ella al completar la implementación.
2. Uso de Pull Requests (PR):
    a. Todo cambio en development y main debe realizarse mediante PRs revisados por otro desarrollador.
    b. Los PRs deben utilizar una estructura estandarizada de redacción.
3. Archivo .gitignore: Debe incluirse un archivo .gitignore adecuado para excluir archivos innecesarios, como dependencias, configuraciones locales y archivos generados automáticamente.

10.2. Configuración de Git Hooks para validaciones

El sistema debe implementar Git Hooks para automatizar la validación del código antes de cada commit y push, asegurando la calidad y consistencia del código. Deben cumplirse los siguientes requisitos:

1. Pre-commit hook: Validar el formato del código.
2. Pre-push hook: Verificar que el código se compile correctamente.

El sistema debe garantizar que las validaciones previas a los commits y pushes mejoren la calidad del código y eviten errores en la integración.

10.3. Convenciones en commits y control de cambios

El sistema debe utilizar una convención estandarizada para los mensajes de commit, asegurando claridad y trazabilidad en el historial del proyecto. Deben cumplirse los siguientes requisitos:

1. Estructura de mensajes de commit basada en Conventional Commits:
    a. feat: Se utiliza para nuevas funcionalidades.

   b. fix: Se emplea para correcciones de errores.
   c. docs: Se aplica para cambios en la documentación.
   d. refactor: Se usa para mejoras en el código sin modificaciones en la funcionalidad.
   e. test: Se destina a pruebas unitarias o de integración.
   f. chore: Se utiliza para cambios en la configuración del proyecto o mantenimiento sin impacto en la funcionalidad.
   g. style: Se aplica a modificaciones en la presentación del código sin afectar su comportamiento.
   h. perf: Se usa para mejoras en el rendimiento del sistema.
   i. build: Se utiliza para modificaciones en la configuración de compilación o dependencias.
   j. revert: Se destina a la reversión de commits anteriores.
   k. breaking change: Se usa para cambios incompatibles con versiones anteriores y debe ir acompañado de una advertencia.
2. Reglas para commits:
   a. Cada commit debe ser atómico, es decir, incluir solo un cambio específico.
   b. Los mensajes de commit deben describir con claridad los cambios realizados, evitando términos genéricos como "update" o "fix bug".
   c. En caso de breaking changes, se debe indicar explícitamente con el sufijo "!" en el tipo de commit (por ejemplo, feat!: eliminar endpoint obsoleto).