ChatGPT

Módulo de Compras – Diseño de la Especificación OpenAPI

Contexto y Alcance del Módulo de Compras (Portal de Compras)

El **Portal de Compras** es el subsistema de la aplicación de comercio electrónico destinado a los usuarios finales. Según el enunciado del proyecto, este módulo debe permitir a los usuarios registrarse y autenticarse, realizar búsquedas de productos, efectuar compras de esos productos y hacer el seguimiento de sus pedidos hasta la entrega [1]. En otras palabras, el Portal de Compras comprende todas las funcionalidades de **front-end** de la tienda en línea: desde la gestión de cuentas de usuario y catálogo de productos, hasta el proceso de checkout (confirmación de la compra) y la consulta del estado de entrega. (Cabe aclarar que el **pago electrónico** está fuera del alcance en esta etapa, tal como indica el enunciado "El pago queda para otro momento" [1]).

Este módulo de compras no opera de forma aislada, sino que **se integra con los otros servicios** del sistema. En particular, cuando un usuario finaliza una compra, el Portal de Compras debe interactuar con el subsistema de **Stock** para **reservar el stock** de los productos comprados, y también con el servicio de **Transporte/Logística** para **generar una orden de envío** correspondiente [2]. Tras concretarse la compra, el servicio de transporte se encargará de la entrega física y de mantener informado al portal sobre la ubicación y estado del pedido, notificando al portal cuando el producto haya sido entregado [3]. Esto significa que el Portal de Compras deberá manejar actualizaciones de estado de los pedidos (por ejemplo, "enviado", "en camino", "entregado") según la información provista por el módulo de logística.

En resumen, el alcance funcional del módulo de compras incluye: registro/login de usuarios, exploración de productos (posiblemente con filtros), gestión de carrito de compra, proceso de checkout para crear pedidos, y visualización del estado de los pedidos (tracking). Este conjunto de capacidades coincide con el de muchas APIs de comercio electrónico existentes – por ejemplo, una especificación OpenAPI para una tienda en línea típica cubre justamente flujos como autenticación de usuarios, navegación de productos, actualización de carrito y realización de pedidos [4].

Endpoints propuestos para la API de Compras

Aunque los endpoints exactos aún **no han sido decididos por el equipo**, a continuación se propone un listado de recursos y operaciones **REST** que probablemente serán necesarios para cubrir las funcionalidades mencionadas. Estos endpoints siguen buenas prácticas de diseño RESTful (la API expone **entidades** como recursos y evita usar acciones como parte de la URL [5]) y están pensados para satisfacer los requisitos del Portal de Compras:

- **POST** `/auth/register` – Registro de un nuevo usuario. Permite crear una cuenta de cliente proporcionando los datos necesarios (p. ej. email, contraseña, nombre). Retorna un código **201 Created** si el registro fue exitoso, o errores de validación en caso contrario. *(Endpoint público, no requiere autenticación previa).*
- **POST** `/auth/login` – Autenticación de usuario existente. El cliente envía sus credenciales (email y contraseña) y, si son válidas, el sistema genera un **token JWT** de acceso (mecanismo

Bearer) que se devolverá en la respuesta [6]. Las siguientes solicitudes del usuario incluirán este token en la cabecera `Authorization` para acceder a endpoints protegidos. *(Endpoint público).*

- **GET** `/products` – Listado de productos disponibles en el catálogo. Permite **buscar y filtrar** productos; por ejemplo, filtrar por categoría, rango de precios, texto de búsqueda, etc. (estos filtros se enviarían como **query parameters**, e.g. `?category=Electrónica&min_price=100`...). Devuelve un arreglo de productos. La implementación detrás probablemente consultará al servicio de **Stock** para obtener la información actualizada de los productos publicados. *(Endpoint abierto, ya que el catálogo puede ser público, aunque podría requerir autenticación si así se define.)*
- **GET** `/products/{id}` – Detalle de un producto específico. Devuelve la información completa de un producto dado su identificador (por ejemplo, `id` podría ser un UUID). También obtendría la información desde el módulo de Stock. Respuestas posibles: **200 OK** con el producto si existe, o **404 Not Found** si no se encuentra el ID solicitado.
- **GET** `/cart` – Obtener el contenido del **carrito de compras** actual del usuario. Devuelve los items que el usuario añadió a su carrito (cada ítem contendrá identificador de producto, nombre, cantidad, precio, etc.). Requiere autenticación (el token indica qué usuario realiza la solicitud, y el backend determina su carrito correspondiente). Respuesta **200 OK** con la lista de ítems, o posiblemente **401 Unauthorized** si falta el token.
- **POST** `/cart/items` – Agregar un producto al carrito. El cuerpo de la petición incluirá al menos el `product_id` del producto y la `cantidad` deseada [7]. El backend agregará ese producto al carrito del usuario (creando el carrito si aún no existe). Responde **200 OK** (o **201 Created**) confirmando la adición, con el estado actualizado del carrito, o errores como **400 Bad Request** si la solicitud es inválida (por ejemplo, falta el ID de producto) y **401** si el usuario no está autenticado.
- **POST** `/checkout` – Finalizar la compra (iniciar el proceso de **checkout**). Este endpoint toma la información necesaria para crear un **pedido**: típicamente la confirmación de los productos del carrito y datos de envío del pedido. Por ejemplo, podría requerir un `address_id` (dirección de entrega seleccionada) y un `payment_method_id` (aunque en este caso el pago no se procesa realmente, podría ser un campo simulado) [8] [9]. Al invocar este endpoint, el portal de compras deberá coordinar varios pasos internamente: verificar stock disponible (vía servicio de Stock), calcular el costo de envío (vía servicio de Transporte, usando la dirección y otros datos [10]), **reservar el stock** en el sistema de Stock y **crear el pedido** tanto en su propia base como registrar un **pedido de entrega** en el sistema de logística [2]. Si todo resulta bien, devuelve **201 Created** con los detalles del pedido confirmado (incluyendo un ID de pedido y estado inicial, e.g. "pendiente" o "confirmado") [11]. En caso de errores (por ej. stock insuficiente durante la reserva) podría devolver **409 Conflict** u otro código apropiado, además de **400** si hay datos inválidos. *(Requiere autenticación.)*
- **GET** `/orders` – Lista los pedidos realizados por el usuario autenticado (su **historial de compras**). Retorna una lista de órdenes con información resumida (ID, fecha, total, estado actual) [12]. Permite al cliente ver sus compras pasadas y en curso. *(Requiere autenticación.)*
- **GET** `/orders/{orderId}` – Detalle de un pedido específico. Devuelve información completa de la orden con ID dado, incluyendo los ítems comprados, monto total, dirección de envío, método de entrega y el **estado** del pedido (ej. pendiente, enviado, en camino, entregado) [13] [14]. Este estado se actualizará conforme el módulo de Transporte informe novedades (por ejemplo, cambiando a "shipped/enviado" cuando se despacha). Respuestas: **200 OK** con el detalle si el pedido pertenece al usuario autenticado, **404** si no existe tal pedido (o no le pertenece), **401** si no autenticado.
- **GET** `/addresses` – (Opcional) Obtener las **direcciones de envío** guardadas del usuario. Es común permitir que el usuario tenga direcciones almacenadas (domicilios) para seleccionar en el

checkout. Este endpoint devolvería una lista de direcciones asociadas a la cuenta [15]. *(Requiere auth.)*

- **POST** `/addresses` – (Opcional) Agregar una nueva dirección de envío a la cuenta del usuario [16]. El cuerpo incluiría los datos de dirección (calle, ciudad, provincia, código postal, etc.). Responde **201 Created** si se agregó correctamente, devolviendo quizá un ID de dirección, o errores de validación (**400**) si algún campo obligatorio falta. *(Requiere auth.)*

(Nota: Los nombres de rutas y recursos pueden ajustarse según las convenciones que adopte el equipo; por ejemplo, podría usarse el prefijo `/api/v1/` delante de las rutas para versionar la API. Asimismo, se podrían utilizar nombres en inglés –como en muchos APIs REST– o en español, pero lo importante es la **coherencia** y claridad en el diseño. En lo posible, mantener los nombres en plural para colecciones –e.g. `/products` – y en singular cuando corresponda a un item específico –e.g. `/products/{id}` –).

Esta lista abarca los elementos esenciales identificados. De hecho, coincide con los componentes de otras APIs e-commerce completas: creación y autenticación de usuarios, manejo de direcciones de envío, catálogo de productos, carrito, pedidos, etc. [17]. *En nuestro caso se excluyen por ahora las partes de pago*, pero se podría ampliar en un futuro integrando endpoints de pago o integraciones con pasarelas. También cabe mencionar que las funcionalidades de **categorías** o **búsquedas avanzadas** (por ejemplo, productos por categoría, rango de precios, etc.) pueden implementarse vía parámetros en `GET /products` en vez de endpoints separados.

Importante: Todos estos endpoints deben diseñarse siguiendo principios REST. Esto implica, por ejemplo, evitar endpoints que representen acciones imperativas (p. ej., en lugar de tener un `/comprarAhora` se utiliza correctamente un **POST** sobre la colección de pedidos `/orders` para representar la acción de crear una compra) [5]. Asimismo, las operaciones deben ser **sin estado** en el sentido HTTP (stateless): cada request contiene la información necesaria (como el token de usuario y los datos del pedido) sin depender de un estado de sesión en el servidor [18]. Esto mejora la escalabilidad y simplicidad de la API.

Estructura y Construcción de la Especificación OpenAPI

A la hora de **plantear la especificación OpenAPI** para este módulo, se recomienda seguir un enfoque **API-first**: primero definir claramente el contrato de la API (los endpoints, datos y errores) y someterlo a revisión, antes de implementar la lógica. Los profesores han sugerido trabajar en una *feature branch* separada para cada módulo y luego generar un **Pull Request** hacia la rama main para revisión y aprobación. A continuación, se presenta una guía paso a paso (con buenas prácticas) para crear la especificación OpenAPI del módulo de Compras:

1. **Identificar endpoints y requerimientos:** Como primer paso, enumeren las operaciones que el módulo debe exponer (muchas de las cuales ya listamos arriba). Revisen el escenario y requisitos para no omitir nada. Por ejemplo, sabemos que debemos incluir registro/login, listado de productos, etc., según lo solicitado en el Portal de Compras [1]. También consideren los casos de error o excepciones (¿qué pasa si un producto no existe? ¿y si no hay stock? etc.) para contemplar respuestas adecuadas.

2. **Definir los** modelos de datos **(schemas) necesarios:** Identifiquen las **entidades** principales y sus atributos. En este módulo podrían ser: **Usuario**, **Producto**, **ItemCarrito**, **Pedido**, **Dirección**, etc. Por ejemplo, un objeto **Producto** debería incluir campos como `id` (UUID), `name` (nombre), `description`, `price` (precio), `stock` (cantidad disponible), `category` (categoría o tipo), `image_url` (foto), fechas de creación/actualización, etc. [19]. Un **Pedido**

incluirá un `id`, lista de ítems comprados, monto total, estado (p.ej. pendiente/confirmado/enviado/entregado) [13], fecha, etc. Es útil bosquejar estos modelos con sus tipos de datos y relaciones. Decidan también qué campos son requeridos y cuáles opcionales en cada caso. Estos esquemas se incorporarán luego en la sección `components/schemas` de la especificación OpenAPI para poder reutilizarlos en las definiciones de requests y responses.

3. **Crear la estructura base del documento OpenAPI:** Inicien un nuevo archivo (por convención, suele llamarse **openapi.yaml**). En la primera línea se especifica la versión de OpenAPI que van a usar, por ejemplo:

```
openapi: 3.0.3
```

(O la 3.1.0 si prefieren la versión más reciente). Luego añadan el objeto `info` con los metadatos de la API: un **título claro**, descripción concisa y versión del API [20]. Por ejemplo:

```
info:
  title: Portal de Compras API
  version: "1.0.0"
  description:
API del Portal de Compras - permite registro de usuarios, listado de
productos, creación de pedidos y seguimiento de entregas.
```

Asegúrense de que el título sea descriptivo del servicio (evitar nombres genéricos como "Servicio API" sin contexto) [21] [22]. La descripción puede mencionar brevemente las funciones del módulo y su lugar en el sistema (p.ej. "parte de la plataforma de compras online de UTN, módulo que expone funciones a usuarios finales"). Opcionalmente pueden incluir aquí contacto, términos de servicio o licencia si aplica.

4. **Especificar los servidores (servers):** En OpenAPI se puede listar uno o más **servers** que indiquen las URL base donde estará desplegada la API. Por ejemplo, un servidor de desarrollo/local y otro de producción. Ejemplo:

```
servers:
  - url: https://api.mi-tienda.com/v1
    description: Servidor de producción
  - url: http://localhost:8080/api/v1
    description: Servidor local de desarrollo
```

Esto ayuda a los lectores de la documentación a conocer las URLs base. En una fase inicial pueden dejar solo uno genérico. También decidan si versionarán la API en la URL (como `/v1` en el ejemplo) o mediante el `info.version`. Es común incluir la versión en el path para facilitar evoluciones sin romper compatibilidad.

5. **Definir la seguridad (autenticación):** Dado que varias operaciones requieren que el usuario esté autenticado mediante token, es importante definir un **esquema de seguridad** en la especificación. En la sección `components/securitySchemes` se puede agregar un esquema, por ejemplo:

```
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

Esto declara que usamos auth tipo HTTP Bearer JWT. Luego, a nivel global del documento, podemos agregar:

```
security:
  - BearerAuth: []
```

para indicar que por defecto **todas** las operaciones requieren ese esquema de auth [23]. Finalmente, en cada operación que *no* requiera autenticación (como *register* y *login*), se puede **anular** la seguridad declarando `security: []` vacío a nivel de operación [24]. De esta manera logramos que la mayoría de endpoints requieran JWT por defecto, excepto aquellos explícitamente públicos. Esto queda claro en la documentación y además las herramientas podrán inferir que deben enviar un header `Authorization: Bearer <token>` en los endpoints protegidos.

6. **Documentar los componentes reutilizables:** Utilicen la sección `components` para definir **schemas** (esquemas JSON) de las entidades identificadas en el paso 2, así como **parameters** reutilizables o **responses** comunes si lo ven necesario. Por ejemplo: pueden definir un esquema **Producto** detallado una única vez bajo `components/schemas/Producto` y luego referenciarlo en las respuestas de `GET /products` y similares [25] [26]. Asimismo, podrían definir un esquema de **Error** estándar (con campos como `code` y `message`) y usar *$ref* para todas las respuestas de error de la API [27] [28]. Esto evita duplicación y asegura consistencia. Otro componente útil podría ser un parámetro común, por ejemplo un parámetro de ruta `id` que se repite en múltiples endpoints – se puede declarar una vez (nombre, tipo, formato) y luego reutilizar [29] [25].
   Buenas prácticas: Asegúrense de indicar en cada esquema cuáles campos son `required` y los tipos correctos. Pueden usar formatos como `format: uuid` para IDs, `format: date-time` para fechas, etc., lo cual agrega precisión. También es importante anotar descripciones en los campos para que la documentación generada sea clara para los futuros consumidores. Por ejemplo:

```
components:
  schemas:
    Pedido:
      type: object
      required: [id, items, total_amount, status, created_at]
      properties:
        id:
          type: string
          format: uuid
          description: Identificador único del pedido.
        items:
```

```
          type: array
          items:
            $ref: '#/components/schemas/ItemCarrito'
        total_amount:
          type: number
          format: float
          description: Monto total de la orden (suma de los precios de
items).
        status:
          type: string
          description: Estado actual del pedido.
          enum: [pending, confirmed, shipped, delivered]
        created_at:
          type: string
          format: date-time
          description: Fecha y hora de creación del pedido.
```

(El ejemplo anterior ilustra cómo podría definirse el esquema de un Pedido, incluyendo campos requeridos, tipos y descripciones. Nótese el uso de enum para el estado, acotando los valores posibles.)
Definir bien los componentes facilita mucho la siguiente etapa de documentar las rutas, y garantiza que la estructura de datos esté unificada en toda la API. Además, mejora la legibilidad al separar la definición de los modelos de la definición de cada endpoint [30] [31] .

7. **Describir los** paths **(endpoints) con sus operaciones:** Esta es la sección central donde se listan todas las rutas (`/auth/register`, `/products`, `/orders/{orderId}`, etc.) y debajo de cada ruta las operaciones HTTP soportadas (get, post, put, delete, etc.). Para cada operación se debe proveer al menos: un **summary** breve, una **descripción** opcional más detallada, los **parámetros** de entrada, el **request body** (si aplica) y las **responses** con sus códigos y esquemas de datos devueltos [32] [25] . Al documentar cada endpoint, tengan en cuenta lo siguiente:

8. **Summary y description:** El **summary** debe ser una frase corta que resuma la acción (por ejemplo: “Create a new user account” para POST /auth/register) [33] , pero en español según corresponda en la documentación final (e.g. "Registrar un nuevo usuario"). La **description** puede agregar detalles de negocio o comportamiento, por ejemplo “Este endpoint registra a un usuario cliente en el sistema. Devuelve un token JWT si el registro es exitoso...”. Incluir estas descripciones ayuda a los desarrolladores a entender el propósito sin ambigüedades [34] [35] .
9. **Parámetros:** Listar todos los parámetros que la operación acepta. Esto incluye parámetros de **path** (por ejemplo `{orderId}`), query params (ej. `search`, `min_price` en GET /products) y headers relevantes. Cada parámetro debe documentar su nombre, ubicación (`in: path/query/header`), si es `required` y su esquema de tipo de dato (string, integer, etc.), así como una descripción de su significado [29] [25] . Por ejemplo:

```
/products/{id}:
  get:
    summary: Obtener detalle de producto
    parameters:
      - name: id
        in: path
        required: true
```

```
        schema:
          type: string
          format: uuid
        description: ID del producto a obtener.
      - name: currency
        in: query
        required: false
        schema:
          type: string
        description: Moneda para mostrar el precio (ej. USD, EUR). Por
defecto, pesos.
    responses:
      ...
```

Buena práctica: Para cada parámetro proporcionen toda la información posible: tipo de objeto (string, int), formato (si tiene uno específico, e.g. email, uuid), valores posibles (si solo admite ciertos valores, usar enum o describirlo), restricciones (longitud máxima, regex para formato como email) [36], y si es obligatorio u opcional. Mientras más completo, mejor entenderán otros cómo usar el endpoint. Esto incluye los parámetros del cuerpo también (ver siguiente punto).

10. **Cuerpo de la petición (requestBody):** Para endpoints *POST/PUT* donde se envía JSON en el cuerpo, definir el esquema del JSON esperado. Lo ideal es referenciar uno de los **schemas** definidos en components (por ejemplo, `schema: $ref: '#/components/schemas/CartItem'` en el POST /cart/items [37] [38]). Si es un objeto sencillo quizá se define inline, pero generalmente reutilizar esquemas es mejor. Indicar si el body es `required: true` (en la mayoría de POST lo será). También especificar el **media type** esperado, típicamente `application/json`. Se pueden añadir ejemplos de cuerpo válido para mayor claridad.

11. **Responses:** Enumerar las posibles respuestas que la operación puede dar, con su código HTTP y la estructura de datos devuelta. Siempre incluir al menos la respuesta de éxito principal (por ejemplo, **200 OK** o **201 Created** según el caso) y describirla brevemente. Asociar un esquema JSON para la respuesta exitosa: puede ser un objeto (p.ej. Order) o quizá un listado (array) de objetos. Por ejemplo, GET /products tendría respuesta `'200': schema: { type: array, items: { $ref: '#/components/schemas/Producto' } }` [39]. Además de las respuestas exitosas, es fundamental documentar las respuestas de error: por ejemplo **400 Bad Request** (cuando hay errores de validación o faltan datos), **401 Unauthorized** (si requiere auth y no se proporcionó token válido), **404 Not Found** (si el recurso solicitado no existe), **500 Internal Server Error** (error no esperado) [40], etc. Cada código de respuesta debe ir acompañado de una breve descripción del motivo. Si todas las respuestas de error comparten el mismo formato (por ej., `{ "code": 400, "message": "Detalle del error" }`), pueden hacer que esas respuestas referencien el schema de Error común definido en components [27] [28]. Esto evita repetir la estructura en cada endpoint.
    Buena práctica: Asegúrense de **capturar todos los códigos de éxito y error relevantes** que la operación pueda retornar [41] [42]. No dejen códigos implícitos sin documentar. Por ejemplo, si un DELETE puede retornar 204 en caso exitoso y 404 si el elemento no existe, ambos deben aparecer. También es recomendable incluir un mensaje o estructura en 4XX errores para que el cliente sepa qué falló. Definir un **único esquema de error estandarizado** para todas las respuestas de error es muy aconsejable (así todos los errores tienen, por ejemplo, `{ "error": "Mensaje" }` o similar) [42]. Asimismo, es muy útil proporcionar **ejemplos** de respuestas en la especificación. OpenAPI permite incluir ejemplos tanto en requestBodies como en responses. Pueden mostrar un ejemplo de respuesta 200 con datos ficticios (por ej., un objeto Pedido completo) y ejemplo de error 400. Incluir estos ejemplos mejora la documentación

generada y ayuda a entender el formato esperado [43] [44]. De hecho, "no existe una buena especificación de API sin ejemplos" es un consejo repetido en la comunidad [43], ya que facilita la comprensión y las pruebas en entornos simulados.

12. **Organización opcional por tags:** Para facilitar la lectura, pueden emplear **etiquetas (tags)** en las operaciones, agrupando endpoints relacionados bajo una categoría común [45]. Por ejemplo, tag "Auth" para `/auth/*`, tag "Productos" para `/products` y `/cart`, tag "Pedidos" para `/checkout` y `/orders`. En OpenAPI se pueden definir estas etiquetas globalmente (con nombre y descripción) y luego asignarlas en cada operación. Si bien esto no afecta la funcionalidad, sí mejora la estructura de la documentación y la navegación de los lectores por los distintos grupos de endpoints.

13. **Revisar consistencia y buenas prácticas:** Al terminar de describir todos los endpoints, hagan una pasada de revisión general. Verifiquen que los nombres de parámetros sean consistentes (por ejemplo, si usan `{id}` en un path, no llamarlo `{productId}` en otro similar sin necesidad). Comprueben que todos los esquemas usan un **case** uniforme (camelCase o snake_case para nombres de campos, elegir uno y usarlo en toda la API) [46]. Asegúrense de que no haya contradicciones (ej: un campo requerido en schema pero opcional en otro lugar). También revisen que cada operación tenga sus códigos de respuesta adecuados y que para escenarios similares se usen los mismos códigos (coherencia en el manejo de errores). Una API bien diseñada debería devolver el mismo código ante situaciones comparables; por ejemplo, cualquier petición con autenticación inválida debería devolver 401 siempre, en lugar de a veces 403 u otros códigos, para mantener la uniformidad. Si encuentran repetición excesiva de alguna definición, consideren factorizarla en components (p.ej. definieron el mismo esquema de paginación en varios endpoints, mejor ponerlo como componente reusable).

14. **Validar la especificación con herramientas:** Es muy recomendable utilizar herramientas automáticas para chequear que el archivo OpenAPI **no contenga errores de sintaxis o estructura**. Una opción fácil es copiar el YAML en el **Swagger Editor** en línea (editor.swagger.io) para ver si aparecen errores y visualizar cómo quedaría la documentación [47]. El Swagger Editor resaltará en rojo cualquier incumplimiento de la especificación. Otras herramientas útiles incluyen **linters** como *Spectral* que analizan el YAML según reglas de buenas prácticas, o incluso plugins de VS Code especializados en OpenAPI [48]. Estas herramientas no solo validan sintaxis, sino que a veces sugieren mejoras (por ejemplo, detectar que falta una descripción en una response). Dado que ~70% de los proyectos experimentan problemas por documentación de API incorrecta [49], tomarse el tiempo de validar puede prevenir muchos inconvenientes antes de integrar la API. También pueden usar servicios como **YAML Lint** o conversores JSON<->YAML para asegurarse de que la indentación y formato estén correctos [50]. En esta etapa, corrijan cualquier error encontrado hasta que la especificación pase las validaciones satisfactoriamente.

15. **Publicar la especificación en una feature branch y solicitar revisión (Pull Request):** Con el OpenAPI completado y validado, deberán llevar el archivo al repositorio de código. Siguiendo la instrucción dada, se crea una **rama de característica (feature branch)** específica para el módulo de compras (por ejemplo, `feature/openapi-compras`). Suban el archivo `openapi.yaml` (y cualquier otro recurso relacionado, si aplica) a esa rama con un commit descriptivo. Luego, abran un **Pull Request** desde esa rama hacia `main` para que los profesores y compañeros puedan revisar fácilmente el contenido de la especificación. En la descripción del PR pueden resumir qué se incluye (por ej. "Especificación OpenAPI inicial del módulo Portal de Compras, con endpoints X, Y, Z..."). Este proceso de revisión por pares es muy valioso: ayuda a detectar omisiones o detalles inconsistentes en el diseño antes de codificar. De hecho, involucrar al equipo en la revisión de la especificación fomenta la calidad y exactitud del API [51]. Estudios

indican que hasta un 65% de los desarrolladores consideran que una documentación clara de APIs mejora la colaboración y reduce el tiempo de incorporación de nuevos miembros [51]. Por ello, aprovechen el PR para obtener feedback: por ejemplo, quizás otro módulo necesite un ajuste en un endpoint común, o el profesor sugiera devolver algún campo adicional. Tras iterar y ajustar según comentarios (commits adicionales en la rama feature), finalmente se aprobará el PR y la especificación OpenAPI del módulo de compras quedará integrada en la rama principal del proyecto.

16. **Generación de stubs o implementación:** (Ya fuera del alcance de la pregunta, pero a modo de cierre del proceso API-first). Con la especificación final aprobada, el equipo de desarrollo del módulo de compras puede usarla como contrato para implementar el servicio. Pueden incluso utilizar generadores de código a partir del OpenAPI para crear **stubs** de controladores o modelos, si lo ven conveniente, o arrancar un proyecto base. Herramientas como Swagger Codegen / OpenAPI Generator o las integraciones de Postman pueden acelerar este arranque, creando esqueleto de servidor en diversos lenguajes a partir del YAML [52]. En cualquier caso, la spec finalmente servirá como documentación de referencia para consumidores externos o para otros equipos (por ejemplo, el módulo de Frontend web o móvil podría consumir directamente esta API). Mantenerla actualizada con cada cambio es crucial.

En conclusión, al elaborar la especificación OpenAPI del módulo de compras conviene ser **lo más completo y meticuloso posible**, abarcando todos los recursos y casos de uso identificados, pero también siguiendo las pautas estándar que hacen a una buena API (consistencia, claridad, documentación suficiente). Resumiendo las mejores prácticas clave:

- **Título, descripciones y ejemplos claros:** Facilitan la comprensión de la API [21] [43].
- **Endpoints RESTful bien nombrados:** usar sustantivos para recursos, evitar verbos en la URL [5].
- **Definir todos los parámetros y respuestas exhaustivamente:** con tipos, formatos, códigos esperados [36] [40].
- **Seguridad declarada centralmente:** para no repetir en cada endpoint y no olvidar proteger los necesarios [23].
- **Reusable components:** esquemas de datos y mensajes de error comunes para uniformidad [42].
- **Validación y revisión colaborativa:** usar herramientas automáticas y revisión humana (PR) para mejorar la calidad antes de la implementación [47] [51].

Con todo esto en cuenta, el módulo de compras contará con una especificación OpenAPI sólida que servirá de guía para el desarrollo y garantizará que el servicio expuesto cumpla con lo requerido, integrándose correctamente con el resto del sistema. ¡Manos a la obra con la feature branch y mucha suerte con la implementación!

Fuentes Consultadas: Se han incorporado en el texto referencias a la especificación del proyecto [1] [2], ejemplos de API e-commerce similares [17] [4], así como guías y buenas prácticas de OpenAPI [5] [43], para respaldar las recomendaciones realizadas. Cada cita marcada en el formato 【n†Lx-Ly】 corresponde a materiales que amplían o verifican el punto discutido. Todas ellas fueron consideradas al elaborar esta investigación.

---

[1] [2] [3] [10] DS-2025 - TPI-v1.pdf
file://file-PUkPzhSJLn2YfUQNgnoy6K

4 7 8 9 11 12 13 14 15 16 19 20 33 37 38 39 Sample OpenAPI Spec | Beeceptor
https://beeceptor.com/docs/concepts/openapi-sample-spec/

5 18 36 Buenas prácticas en el diseño de APIs
https://www.enmilocalfunciona.io/buenas-practicas-en-el-diseno-de-apis/

6 17 GitHub - jahazieljbh/e-commerce-api: Rest-API-URL
https://github.com/jahazieljbh/e-commerce-api

21 22 34 35 41 42 10 Essentials When Creating an Open API Specification - Tyk
https://tyk.io/blog/10-essentials-when-creating-an-open-api-specification/

23 24 25 26 27 28 29 30 31 32 43 44 45 Especificar un servicio API con open API 3.1
https://www.redsauce.net/es/articulo?post=describir-API-con-openAPI-3.1

40 46 47 48 49 50 51 Understanding OpenAPI 3.0 Structure with YAML Guide | MoldStud
https://moldstud.com/articles/p-exploring-the-structure-of-openapi-30-with-yaml-a-comprehensive-guide

52 The Reimagined API-First Workflow, Part 1: for Developers | Postman Blog
https://blog.postman.com/the-reimagined-api-first-workflow-for-developers/