

**Tecnologías de desarrollo en el servidor**

*Definición del proyecto Integrador*

**PRESENTA**

Lic. Ingeniería en Nanotecnología: Leonardo Daniel Hernández García

Lic. Ingeniería en Desarrollo de Software: Sergio Isaac Santana Jiménez

Lic. Ingeniería en Desarrollo de Software: José Eduardo Pérez Valenzuela

Profesor:

Francisco Javier Sevilla Medina

Tlaquepaque, Jalisco, 6 de febrero del 2025

### My-ecommerce Iteso

**Descripción**

Como estudiantes del ITESO, hemos experimentado el proceso de compra y venta a través del grupo de WhatsApp *"Ventas ITESO"*, donde los usuarios pueden ofertar y adquirir productos mediante mensajes. Sin embargo, este método presenta diversas limitaciones. Los compradores deben desplazarse constantemente por el chat para encontrar publicaciones anteriores sin garantía de que los productos sigan disponibles. Por su parte, los vendedores deben republicar con frecuencia para evitar que sus ofertas se pierdan entre tantas publicaciones. Una aplicación más estructurada optimizaría este proceso, brindando una experiencia más eficiente y accesible para todos.

**Integrantes:**

* Leonardo Daniel Hernández García
* Sergio Isaac Santana Jiménez

**Stack de tecnologías a utilizar:**

* La base de datos escogida es MongoDB, la cual es una base de datos no relacional.
* De framework se utilizaría bootstrap
* Tecnologías Node.js
* Express.js
* Mongoose
* Cors
* Stripr (API para gestionar pagos)
* Shopigy API (gestión de tiendas)

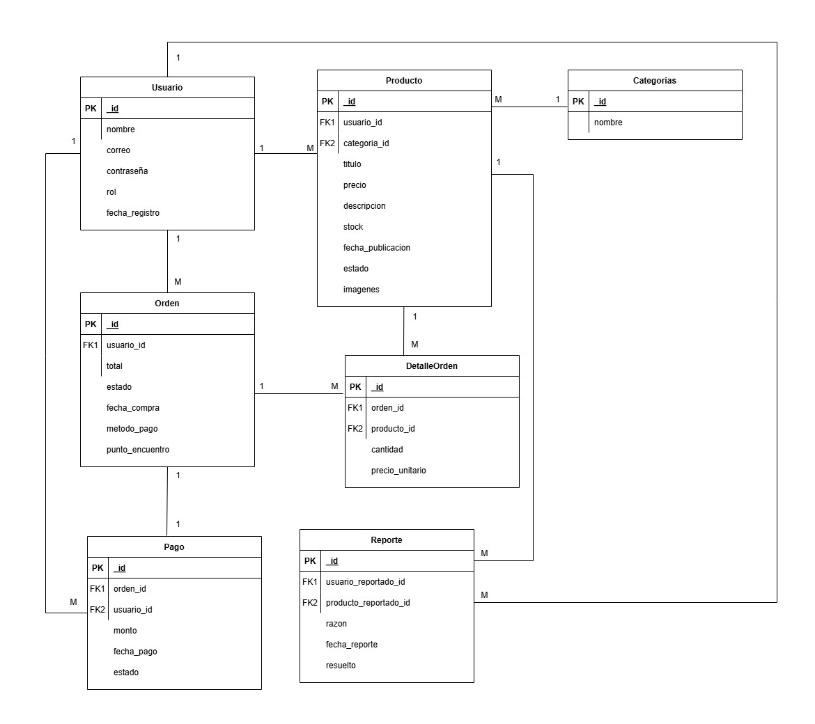
**Lista de roles que pueden registrarse y/o iniciar sesión en la aplicación:**

* Usuarios que cuenten con un correo institucional del ITESO.

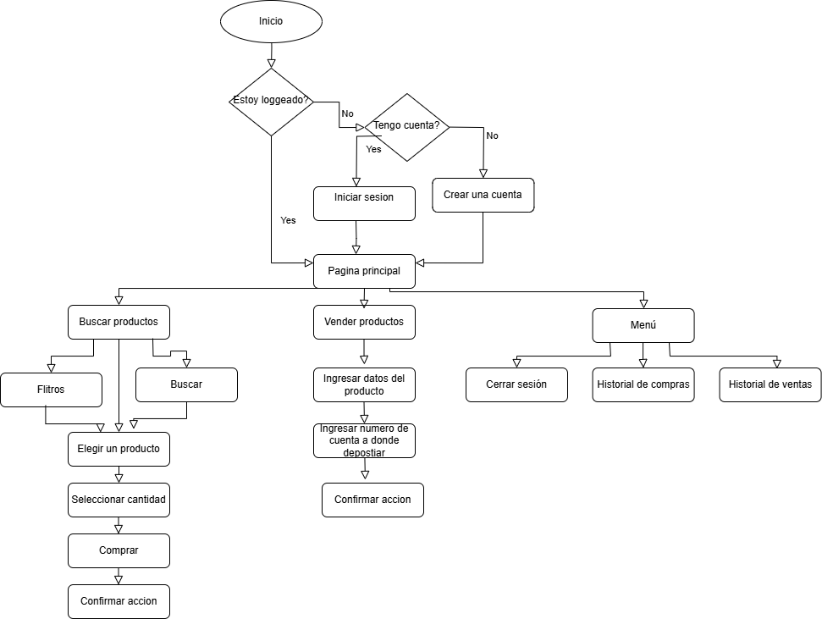
**Listado de usuarios y su rol/tarea:**

* Invitado (usuario que no ha iniciado sesión), solo puede ver contenido de la página, pero no puede comprar ni vender productos.
* Usuario (usuario que inicio sesión), puede agregar contenido a la pagina en forma de productos para su venta o puede comprar de otros usuarios.
* Administrador, puede dar de baja productos no apropiados y gestionar que se cumpla la política de la ecommerce, también puede de dar baja a los usuarios si rompen una norma.

**Diagrama Entidad-Relación:**



**Diagrama de flujo:**

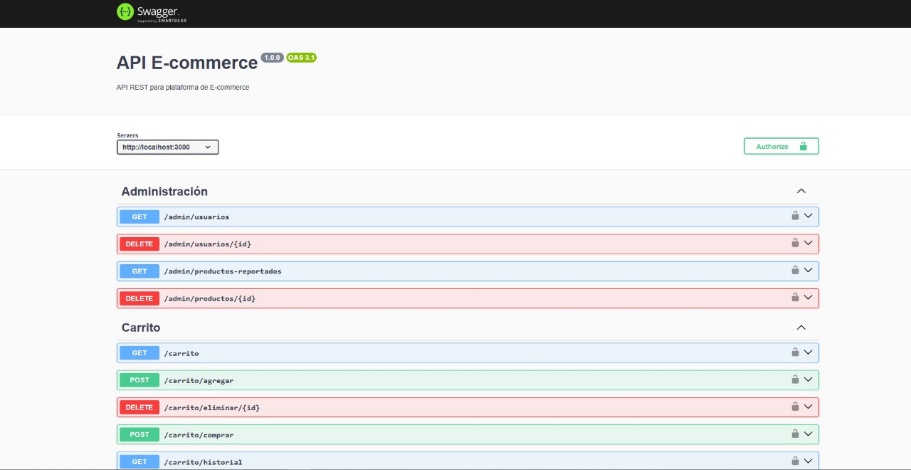


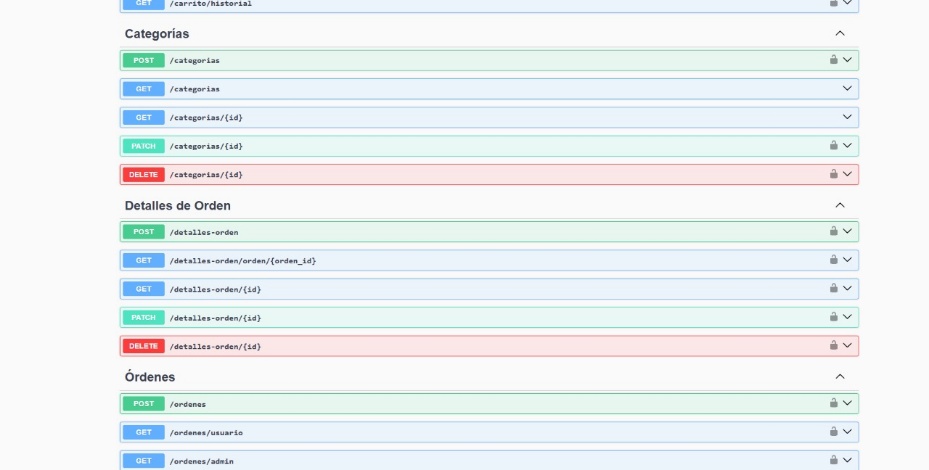
**Documentación:**

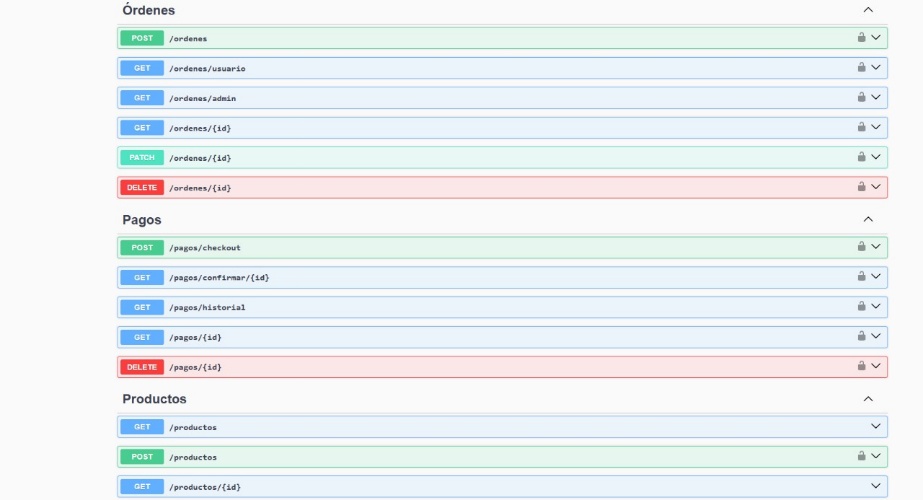
A continuación, se mostrará la configuración Swagger y su visualización, el endpoint creado para entrar a la documentación es localhost:3000/swagger :

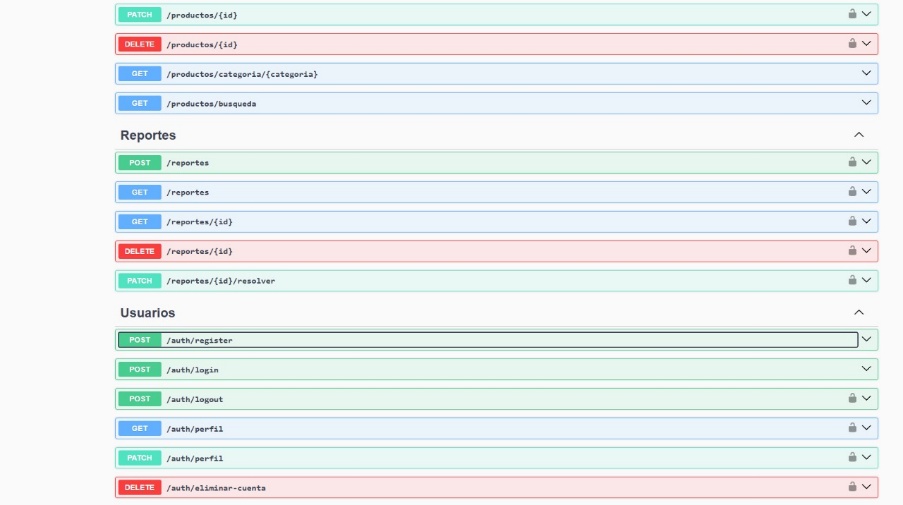


Imagen 1.- Configuración de Swagger.









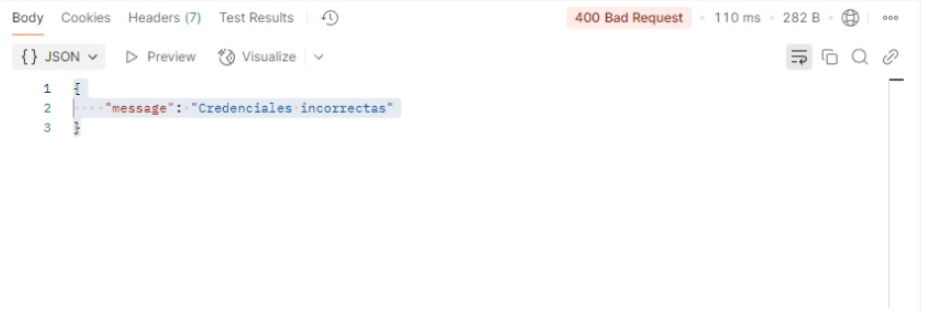
**Autenticación y Permisos**

Para los usuarios, se implementó un *middleware* que verifica la autenticación mediante un token. Este token se genera al iniciar sesión utilizando JWT (JSON Web Tokens). En el caso del administrador, se aplican dos *middlewares*: el primero valida que el usuario esté autenticado, y el segundo verifica su rol. Este último requiere que el usuario tenga un atributo admin para poder acceder a ciertos *endpoints* restringidos.

Los usuarios al tener token pueden tanto comprar productos como dar de alta para su venta, mientras que el administrador puede eliminar productos como usuarios.

Usuarios:

* En el caso de usuario una vez intenta iniciar sesión por medio de su correo, se valida en la base de datos, que este exista, posteriormente se valida la contraseña, si la contraseña es correcta y el usuario también entonces se genera un token por medio de jwt, este mismo token es necesario para el endpoint del perfil y los demás endpoint.
* En caso de no contar con el token:



Administrador:

* En el caso del administrador es muy parecido al del usuario, pero además del token, se valida que tenga un rol de administrador para poder entrar a los demás endpoints.
* Usuario sin rol de admin:



**Documentacion: Avance de Proyecto - Sockets y Archivos:**

Flujo de Comunicación en tiempo real con Socket.io

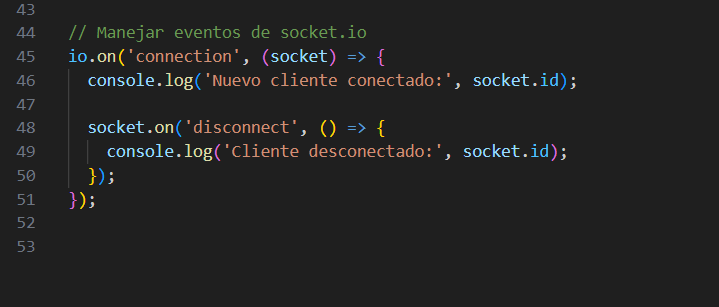
El proyecto tiene una implementación básica de Socket.io que está configurada pero solo parcialmente utilizada. La infraestructura está lista para notificar a los vendedores cuando sus productos son comprados, pero la funcionalidad específica está comentada en el código.

Configuración inicial del servidor Socket.io



El servidor HTTP se crea antes del servidor Socket.io porque Express por defecto usa su propio servidor, pero para Socket.io necesitamos uno personalizado. La configuración de CORS permite que cualquier cliente se conecte desde cualquier origen ("\*") con los métodos GET y POST.

Manejo de conexiones básico

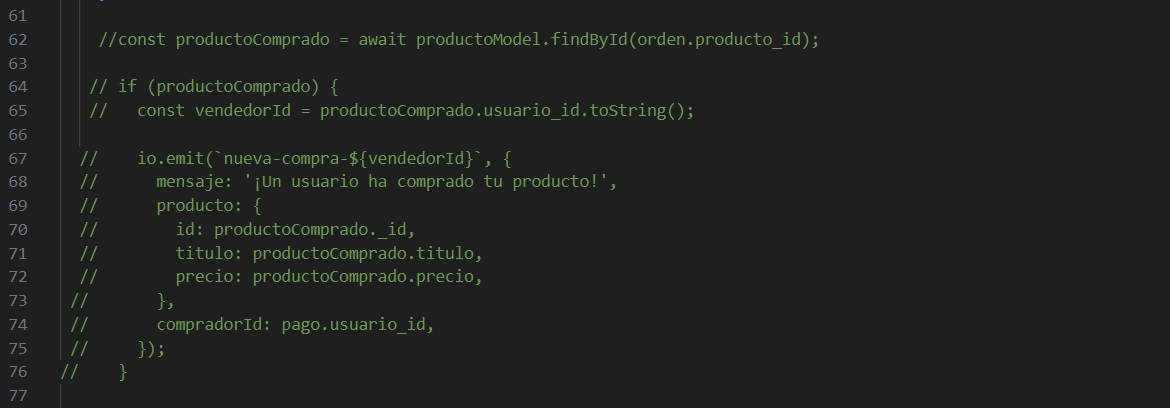


Este código simplemente registra cuando alguien se conecta o desconecta, mostrando el ID único del socket en la consola.

Notificación de compras (función comentada)

Función comentada de notificación:

Socket.io trabaja con canales, por eso usa nueva-compra-${vendedorId} para notificar solo al vendedor específico. El evento transporta un objeto con toda la información necesaria porque WebSocket no tiene el concepto de "request/response", solo mensajes asíncronos.



El flujo planeado es:

Cuando un comprador confirma un pago

Se busca el producto comprado

Se identifica al vendedor del producto

Se emite un evento específico al vendedor

Variables importantes:

io: La instancia de Socket.IO exportada desde index.ts

socket.id: Identificador único de cada cliente conectado

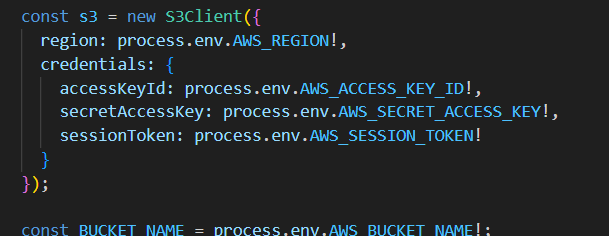
vendedorId: ID del usuario dueño del producto

Canal específico: nueva-compra-${vendedorId}

2. Flujo de Carga y Gestión de Archivos con AWS S3

Configuración del cliente S3:

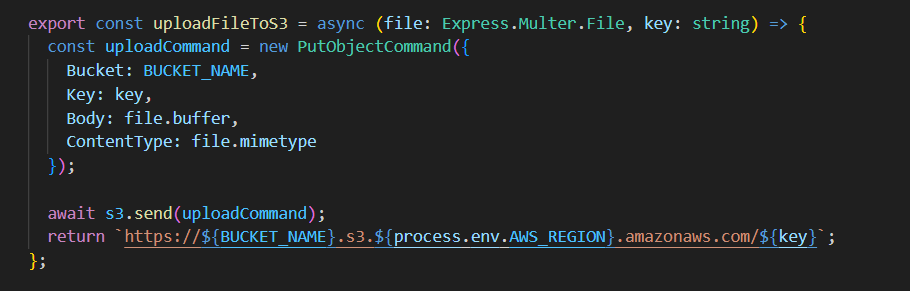
AWS SDK v3 requiere un cliente configurado previamente antes de hacer peticiones. Las credenciales se cargan desde variables de entorno porque hard-codearlas en el código sería un riesgo de seguridad. El cliente debe saber la región porque cada bucket S3 está asociado a una región específica de AWS.



Esta configuración requiere variables de entorno: AWS\_REGION, AWS\_ACCESS\_KEY\_ID, AWS\_SECRET\_ACCESS\_KEY, AWS\_SESSION\_TOKEN, y AWS\_BUCKET\_NAME.

Subida de archivos genérica  
  
uploadFileToS3:

S3 necesita tres elementos para subir un archivo: el destino (bucket), la ubicación dentro del bucket (key), y el contenido. El buffer es necesario porque S3 API trabaja con datos en memoria, no con archivos en disco. La URL se construye manualmente porque AWS no provee automáticamente la URL pública en la respuesta.

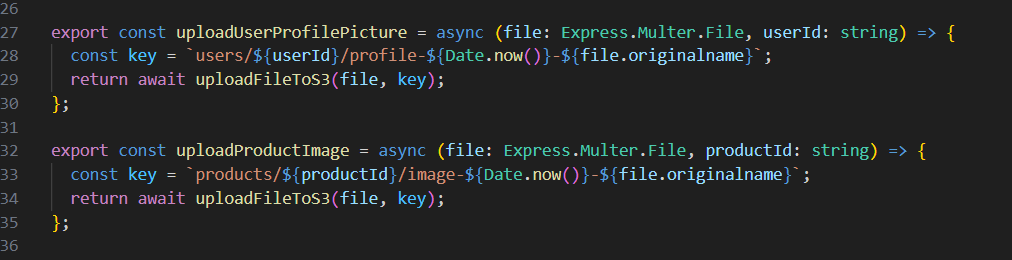


Subida de imágenes de perfil y de producto  
  
uploadUserProfilePicture:

La estructura de carpetas "users/{userId}/" mantiene organizados los archivos por usuario, facilitando gestión posterior. El timestamp en la clave previene colisiones cuando un usuario sube múltiples fotos seguidas. Se llama a uploadFileToS3 en lugar de implementar todo de nuevo porque respeta el principio DRY.

uploadProductImage:

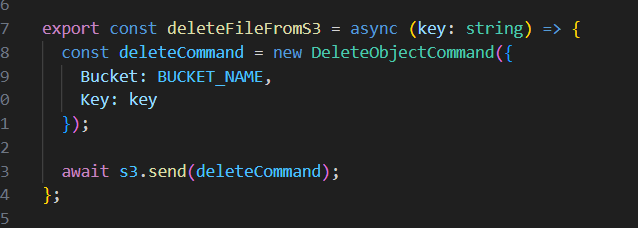
Similar al anterior pero con "products/" en lugar de "users/" porque la estructura de directorios en S3 debe reflejar el tipo de contenido almacenado, facilitando backups, permisos y limpieza posterior.



Eliminación de archivos

deleteFileFromS3:

S3 requiere la clave exacta porque no hay "búsqueda" de archivos en el API, solo eliminación directa por ruta. Es una operación irreversible, por eso los controladores verifican existencia antes de llamar esta función.



Flujo completo de gestión de archivos:

Cliente envía archivo multipart al endpoint

Multer procesa el archivo en memoria

Se verifica el usuario/producto

Se elimina archivo anterior si existe

Se sube nuevo archivo a S3

Se actualiza URL en la base de datos

Se retorna confirmación con URL pública

Variables importantes:

file.buffer: Contenido del archivo en memoria

key: Ruta/nombre único del archivo en S3

imageUrl/profilePictureUrl: URLs almacenadas en los modelos