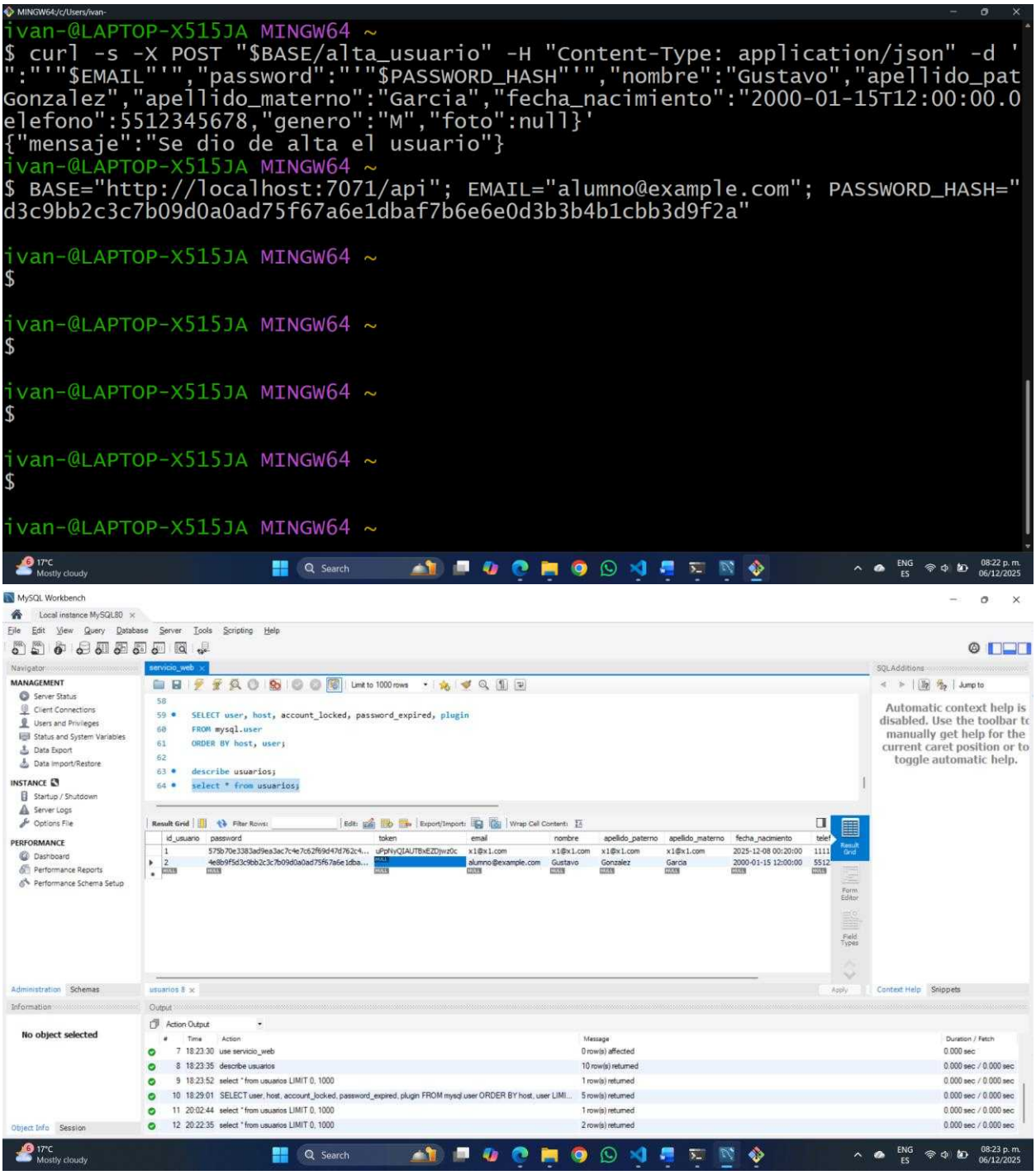


Figura 10. Código de alta_usuario y consulta de inserción exitosa en la base de datos.

6.3 consulta_usuario (perfil del usuario)

Se realizó la función consulta_usuario que, tras verificar el acceso con verifica_acceso(email, token), recuperó los datos del usuario y la foto (si existe), devolviendo un JSON con todos los campos requeridos por el front-end.

- Se accedió a la BD con MySqlConnection.
- Se utilizó LEFT JOIN con fotos_usuarios para un resultado opcional de imagen.
- Se cerró el MySqlDataReader antes de ejecutar otros comandos.

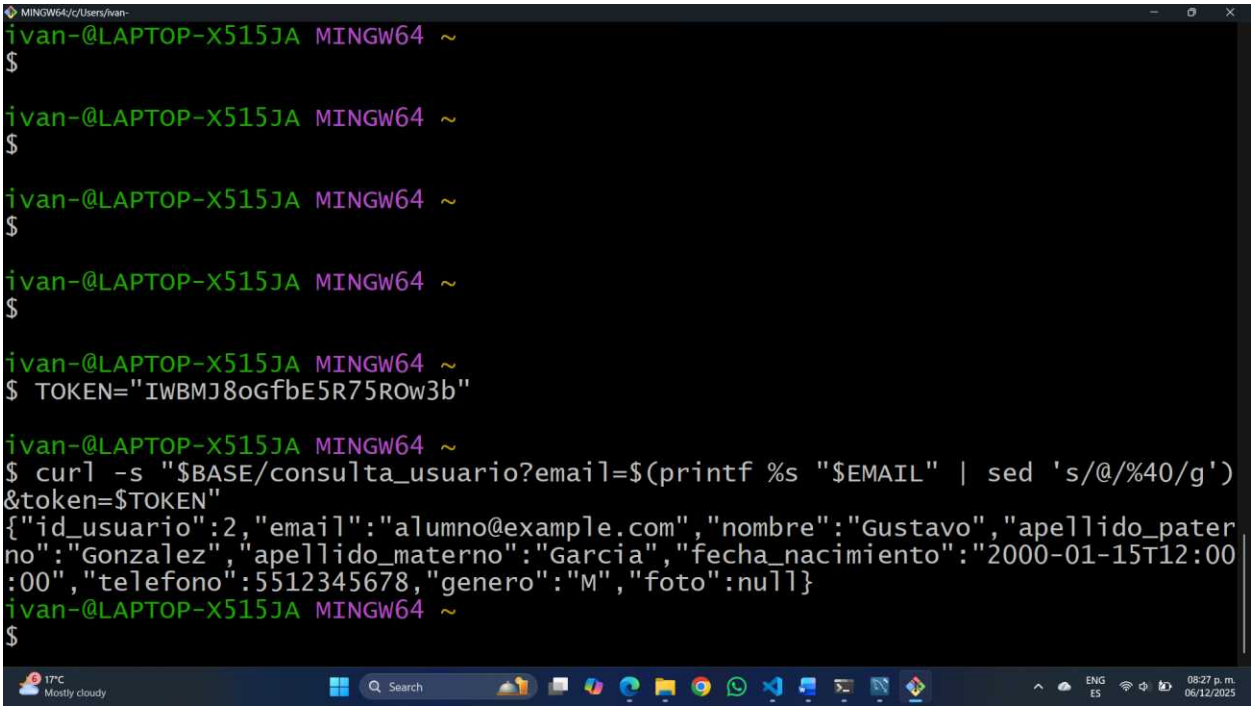


Figura 11. Respuesta JSON completa del perfil del usuario en pruebas locales.

6.4 modifica_usuario (actualización de perfil y foto)

Se realizó la función modifica_usuario para actualizar el perfil. Tras verificar el acceso, se ejecutó un UPDATE de los datos básicos; si el cliente envió un password no vacío, se actualizó la contraseña. Se reemplazó la foto mediante DELETE y INSERT en fotos_usuarios. Todo se ejecutó dentro de una transacción.

- Se accedió a email y token por query string y al body JSON con nuevos valores.
- Se validaron campos obligatorios y se procesó la imagen en base64 si se incluyó.
- Se manejó commit/rollback ante excepciones.

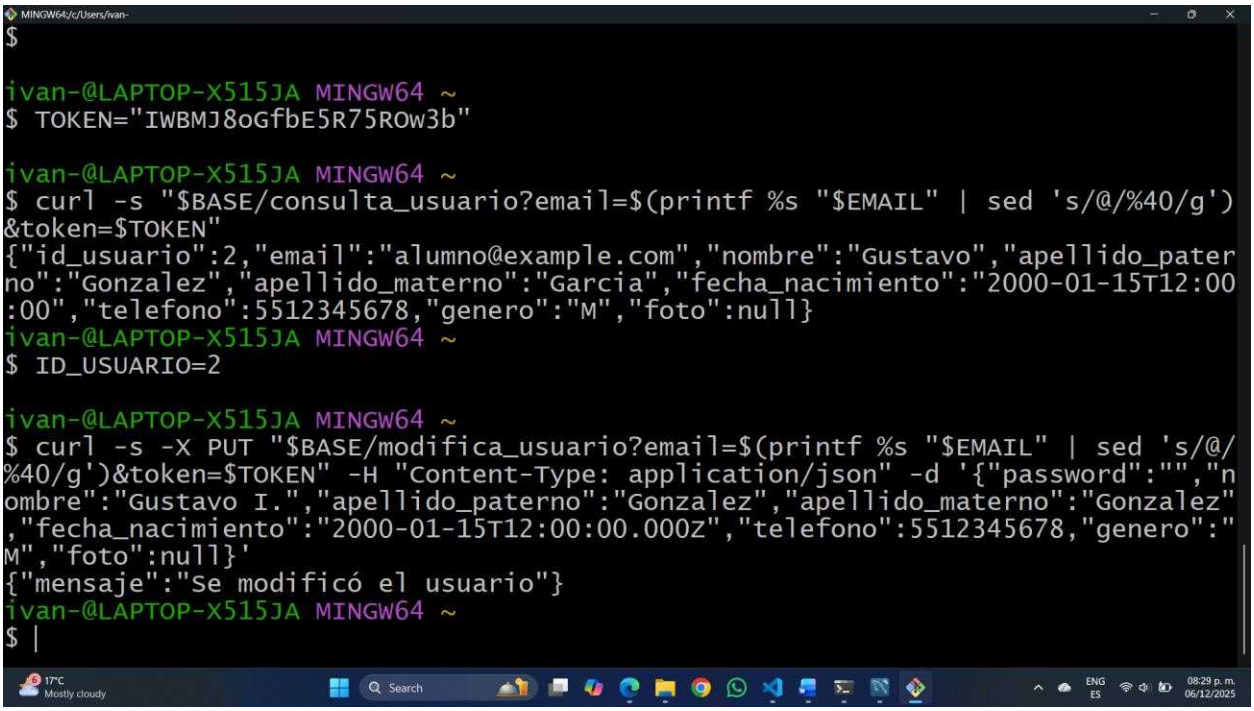


Figura 12. Evidencia de actualización de perfil y cambio de foto con transacción.

6.5 borra_usuario (eliminación del perfil)

Se realizó la función borra_usuario que eliminó primero las fotos del usuario en fotos_usuarios y posteriormente su registro en usuarios, tras verificar el acceso. Se utilizó transacción para asegurar que ambas operaciones se aplicaran de forma consistente.

- Se accedió con email y token del usuario autenticado.
- Se ejecutó DELETE en tabla de fotos y luego en usuarios.
- Se confirmaron respuestas con código 200 y mensaje de éxito.

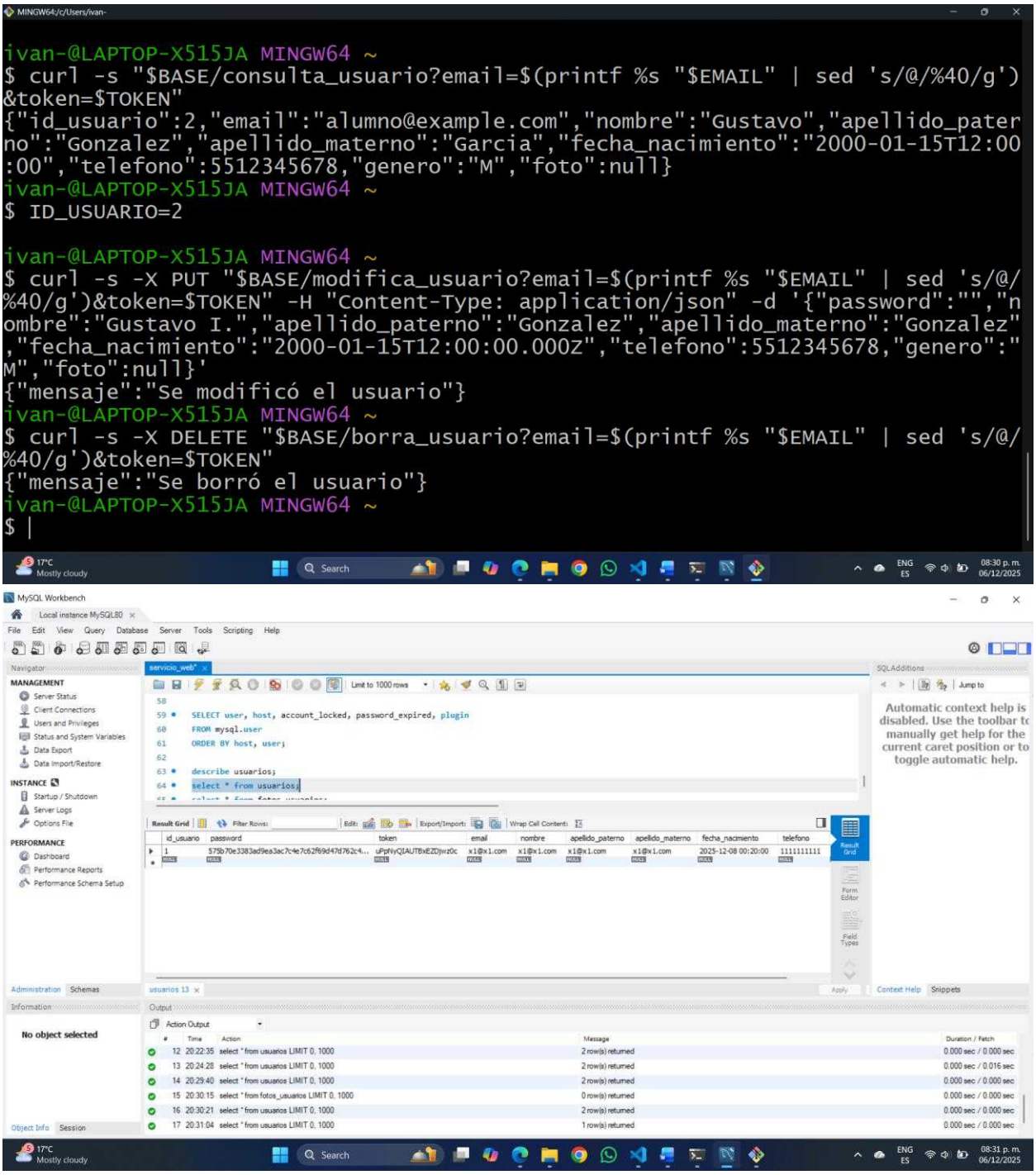


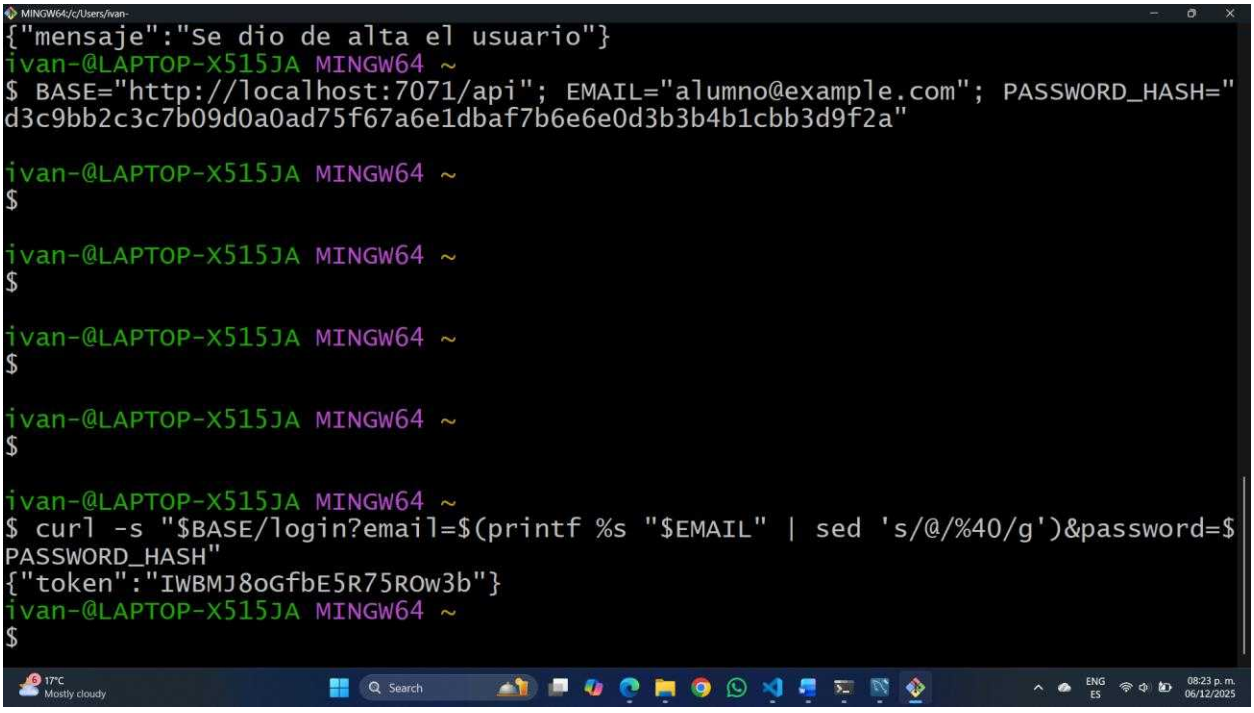
Figura 13. Ejecución de borra_usuario y validación del borrado en MySQL.

6.6 login y verifica_acceso (autenticación y seguridad)

Se realizó la función login que verificó credenciales (email y password SHA-256), generó un token alfanumérico y lo guardó en la tabla usuarios. Se implementaron dos

variantes de verifica_acceso (por email y por id_usuario) que se emplean en las demás funciones para autorizar operaciones.

- Se accedió a la BD y se actualizó el token mediante UPDATE.
- Se devolvió {"token": "..."} en éxito; se devolvió error con "Acceso denegado" en credenciales inválidas.
- Se instaló System.Security.Cryptography para generación de tokens.



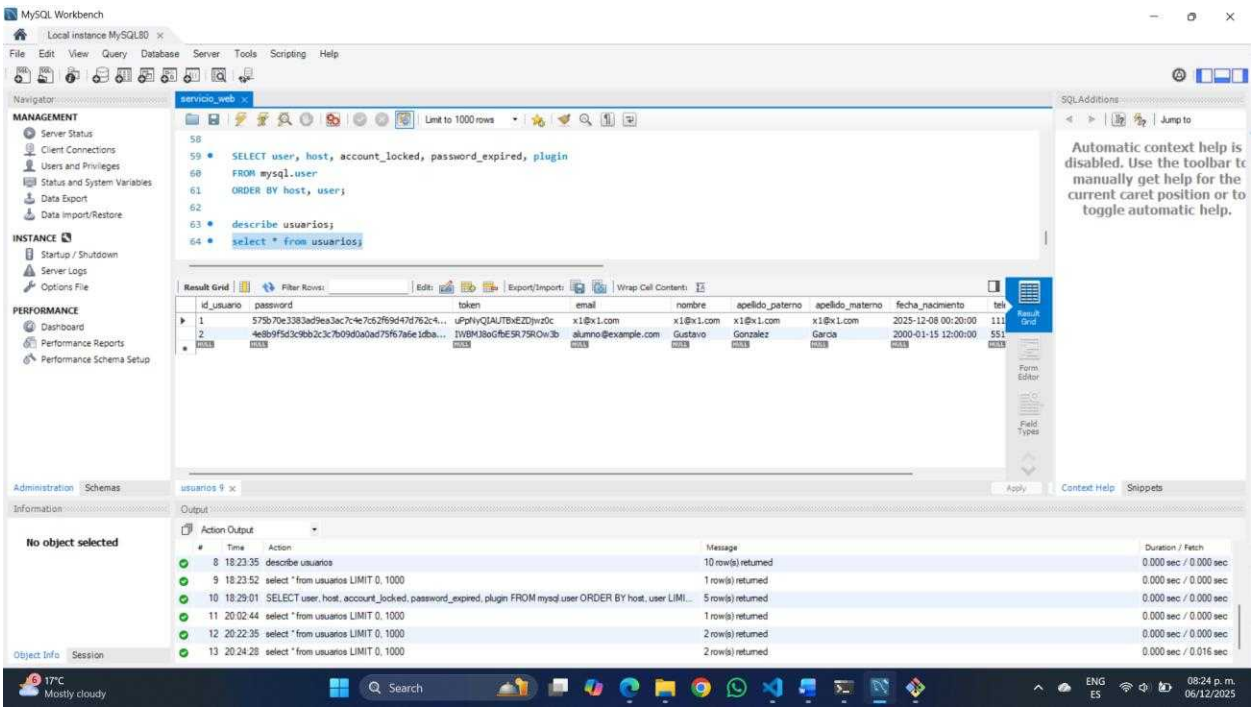


Figura 14. Prueba de login y actualización del token del usuario.

6.7 alta_articulo (captura de stock)

Se realizó la función alta_articulo para insertar artículos en la tabla stock con nombre, descripción, precio y cantidad; si se adjuntó foto, se insertó en fotos_articulos usando el id_articulo recién creado. Se verificó el acceso mediante verifica_acceso(id_usuario, token) y se utilizó transacción al incluir foto.

- Se accedió a parámetros enviados por el front-end (incluida foto en base64).
- Se manejó INSERT en stock y opcional INSERT en fotos_articulos.
- Se devolvió 200 en éxito y 400 en errores de validación o acceso.

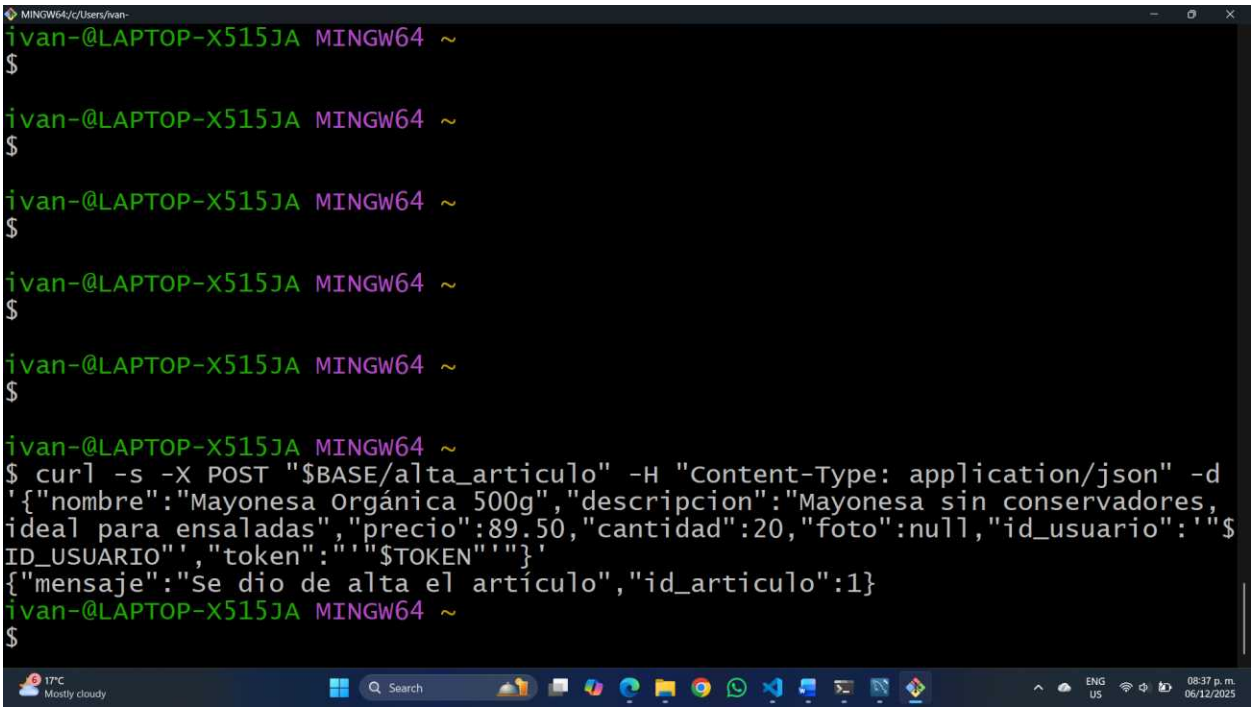


Figura 15. Inserción de artículo en stock y foto asociada con transacción.

6.8 consulta_articulos (búsqueda por palabra clave)

Se realizó la función consulta_articulos que, con acceso verificado, ejecutó una consulta SELECT con LIKE en nombre y descripcion para la palabra clave indicada. Se devolvió un arreglo de artículos con id_articulo, foto pequeña (si existe), nombre, descripción y precio.

- Se accedió con id_usuario y token para autorización.
- Se utilizó LEFT JOIN con fotos_articulos para obtener imagen opcional.
- Se devolvió un JSON arreglo listo para el renderizado en el front-end.

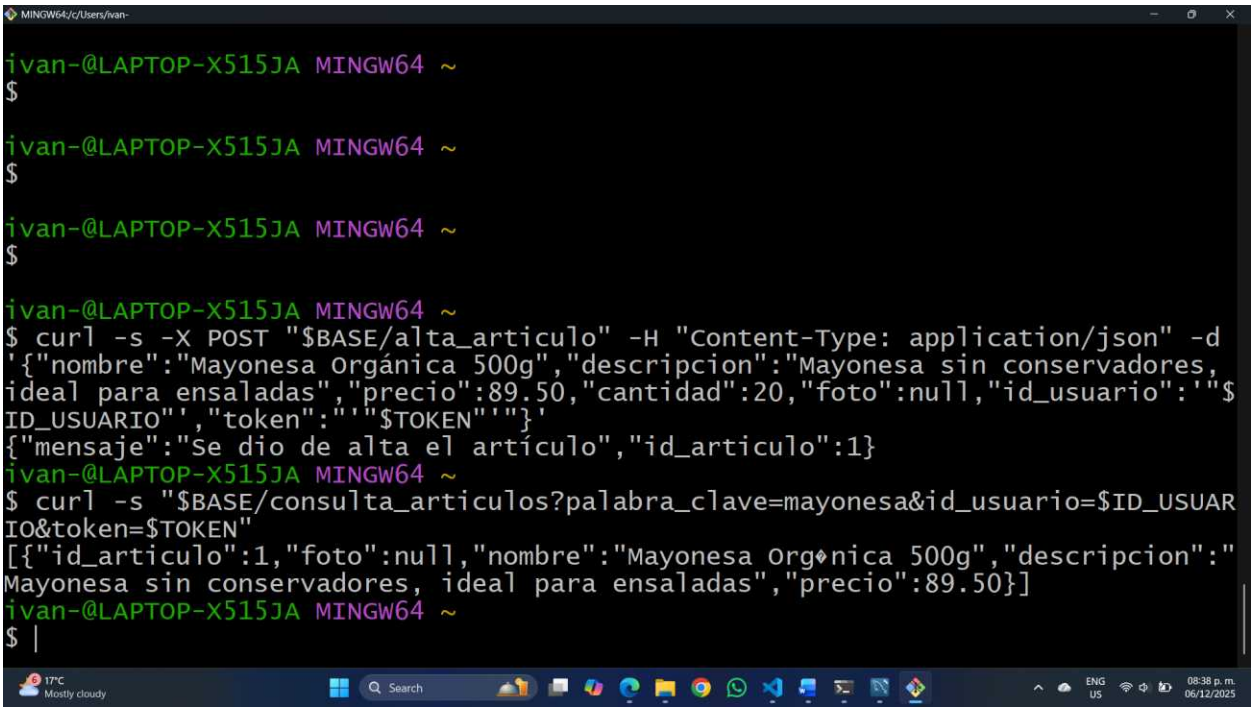


Figura 16. Resultado de consulta_articulos con diferentes palabras clave.

6.9 compra_articulo (compra con transacción)

Se realizó la función compra_articulo que valida si la cantidad solicitada está disponible en stock. Si no hay suficientes artículos, se devolvió 400 con el mensaje “No hay suficientes artículos en stock”. Si alcanza, se ejecutó una transacción que inserta o actualiza el registro en carrito_compra (respetando el índice único (id_usuario, id_articulo)), y actualiza stock restando la cantidad.

- Se accedió con id_usuario, id_articulo, cantidad, token.
- Se realizó SELECT de stock.cantidad, INSERT/UPDATE en carrito_compra, UPDATE en stock.
- Se aseguró commit/rollback ante errores.

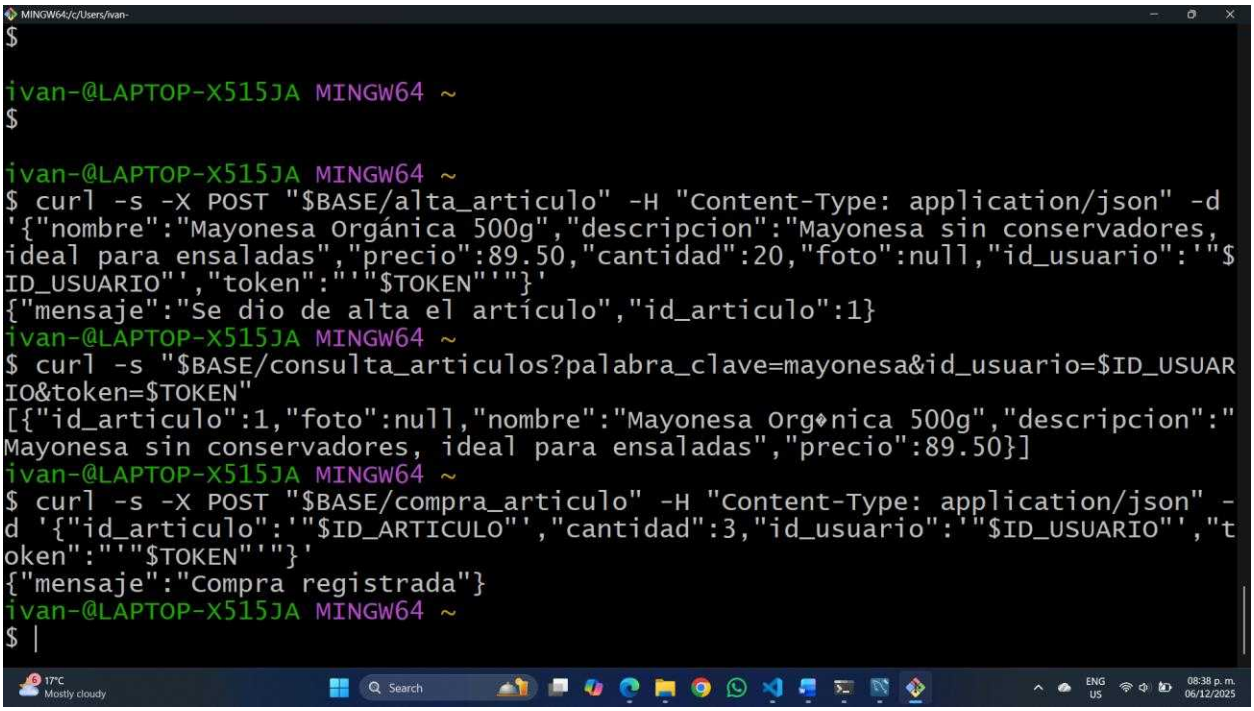


Figura 17. Prueba de compra exitosa y manejo del error por stock insuficiente.

6.10 elimina_articulo_carrito_compra (devolver al stock y borrar del carrito)

Se realizó la función elimina_articulo_carrito_compra para remover un artículo del carrito de un usuario. Se ejecutó una transacción que primero lee la cantidad en el carrito, actualiza stock sumando esa cantidad y posteriormente borra el registro del carrito.

- Se accedió con id_usuario, id_articulo, token.
- Se realizó SELECT de cantidad en carrito; UPDATE en stock (+cantidad); DELETE en carrito_compra.
- Se devolvió 200 al completar la transacción.

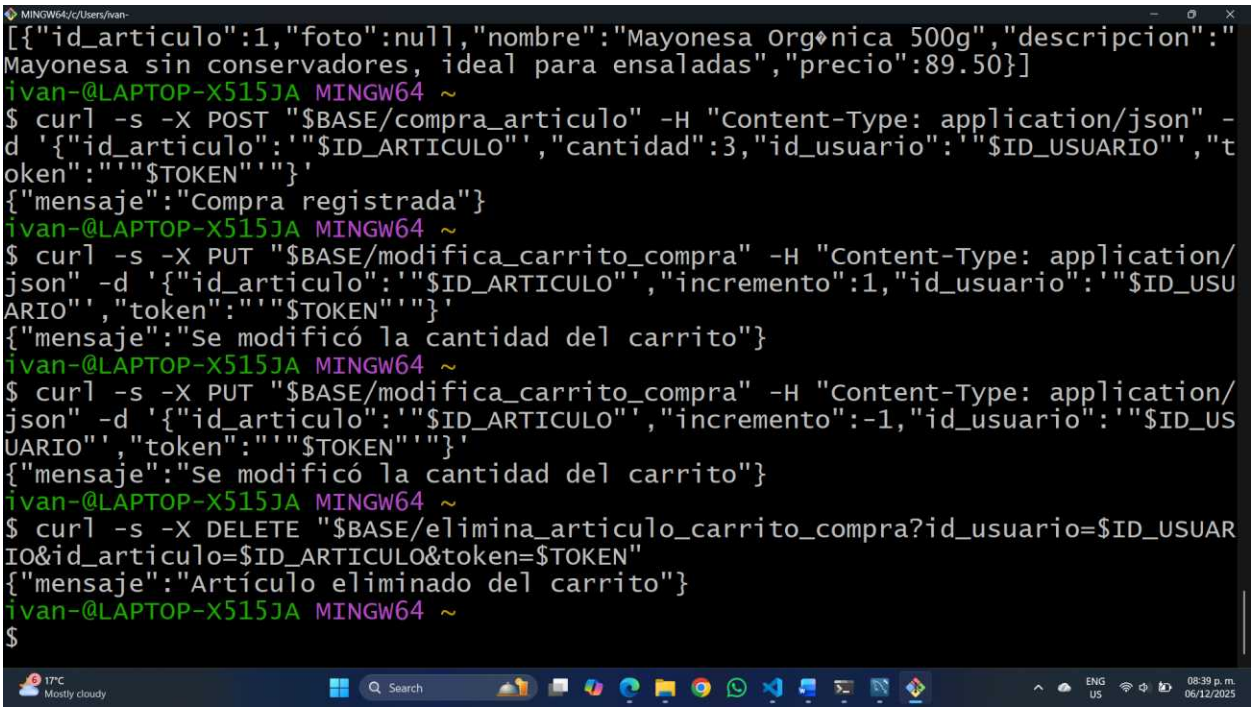


Figura 18. Evidencia de eliminación de artículo del carrito y restauración del stock.

6.11 elimina_carrito_compra (borrado total del carrito)

Se realizó la función elimina_carrito_compra para vaciar el carrito del usuario. Dentro de una transacción, se sumaron las cantidades de cada artículo de vuelta al stock y se borraron todos los registros del carrito del usuario.

- Se accedió con id_usuario y token.
- Se ejecutó SELECT de todos los artículos del carrito, UPDATE acumulado del stock y DELETE general del carrito para el usuario.
- Se garantizó integridad de datos con commit/rollback.

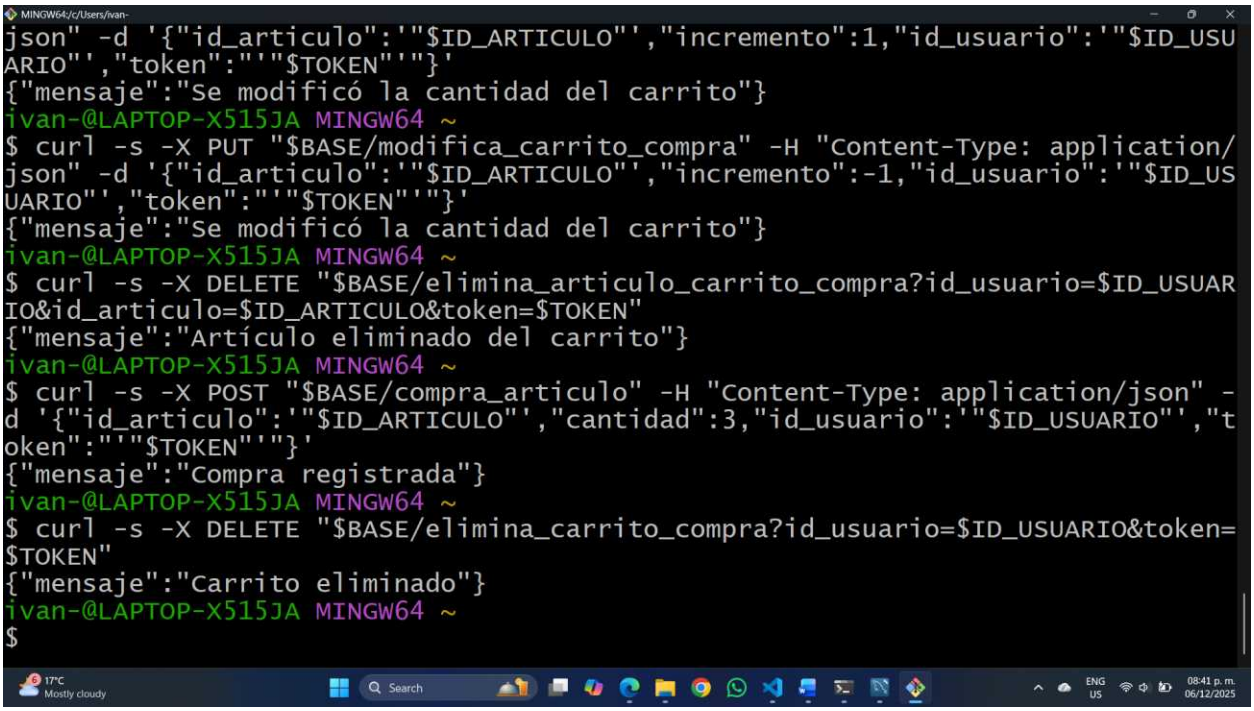


Figura 19. Eliminar carrito completo y verificación del stock recuperado.

6.12 modifica_carrito_compra (ajuste de cantidad +1/-1 con validaciones)

Se realizó la función modifica_carrito_compra que permite incrementar o decrementar la cantidad de compra. Si se incrementa y no hay stock suficiente, se devolvió 400 con “No hay suficientes artículos en stock”. Si se decrementa y ya no hay más artículos, se devolvió 400 con “No hay más artículos en el carrito”. Se actualizó el carrito y el stock en una transacción.

- Se accedió con id_articulo, incremento(+1/-1), id_usuario, token.
- Se validó existencia en carrito y disponibilidad en stock.
- Se ejecutaron UPDATE en carrito_compra y en stock coherentemente.

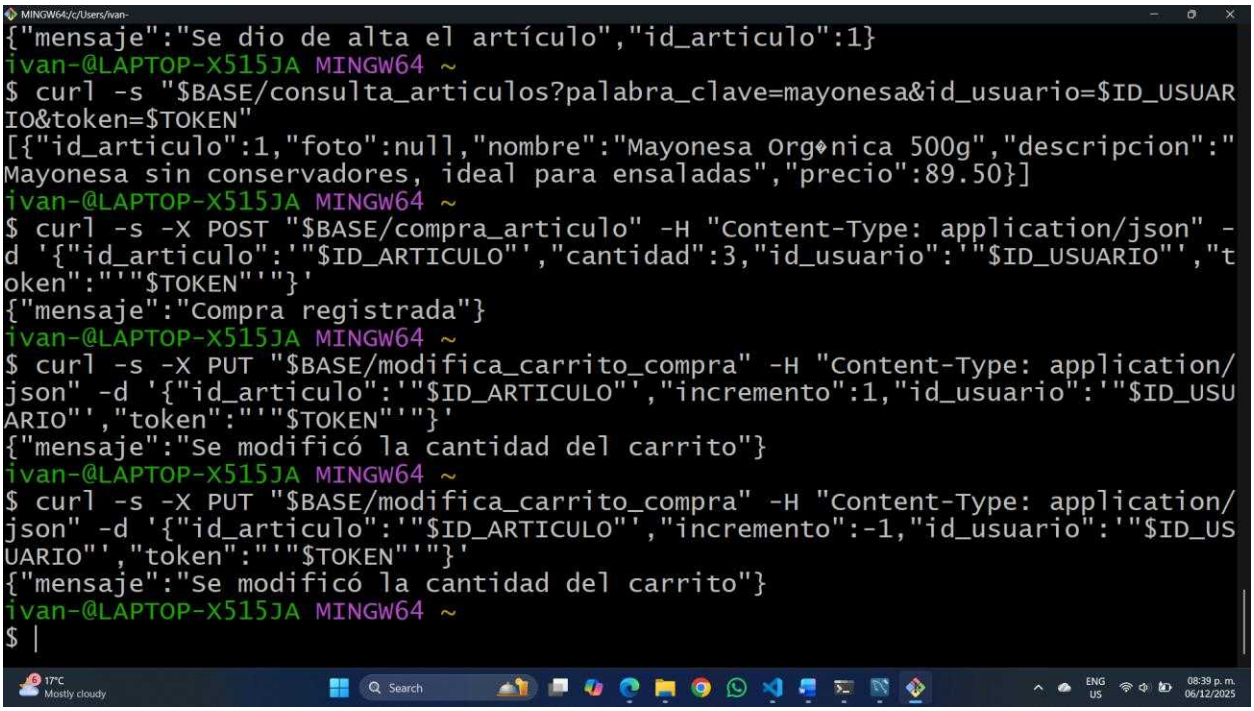


Figura 20. Ajustes de cantidad en carrito (+/-) y respuestas de validación.

7 Front-end: Aplicación web

Se realizó la extensión del front-end para soportar la captura de artículos, la compra y la administración del carrito, manteniendo compatibilidad móvil y sirviendo los archivos estáticos mediante la función Get. Se accedió a las funciones del back-end con WSClient.js usando métodos GET/POST/PUT/DELETE y se aseguró que el flujo completo opere con token e id_usuario obtenidos tras el login.

7.1 Estructura de archivos y carga del front-end

Se organizó el front-end en la carpeta front-end con los archivos prueba.html, WSClient.js y usuario_sin_foto.png. Se instaló la referencia del script en prueba.html por medio de la función Get:

<script src='/api/Get?nombre=/WSClient.js'></script>.

Se accedió al front-end con la URL local http://localhost:7071/api/Get?nombre=/prueba.html y, posteriormente, con la URL de la Function App tras el despliegue. Se validó la meta viewport para uso móvil y la disponibilidad de imágenes por defecto.

- Se realizó la verificación de carga de prueba.html en navegador de escritorio y móvil.
- Se accedió a WSClient.js vía Get y se comprobó que las rutas estén bajo /api.

Figura 22. Captura de pantalla de la estructura del front-end y carga de prueba.html utilizando la función Get.

7.2 Menú principal extendido

Se realizó la extensión del menú principal para incluir las opciones “Captura de artículo” y “Compra de artículos”. Se accedió a estas vistas tras el login, reutilizando las variables globales email, token y recuperando id_usuario desde el endpoint consulta_usuario después de iniciar sesión.

- Se realizó la integración de los nuevos botones en prueba.html.

- Se accedió correctamente a cada opción y se validó la visibilidad/ocultamiento de secciones.

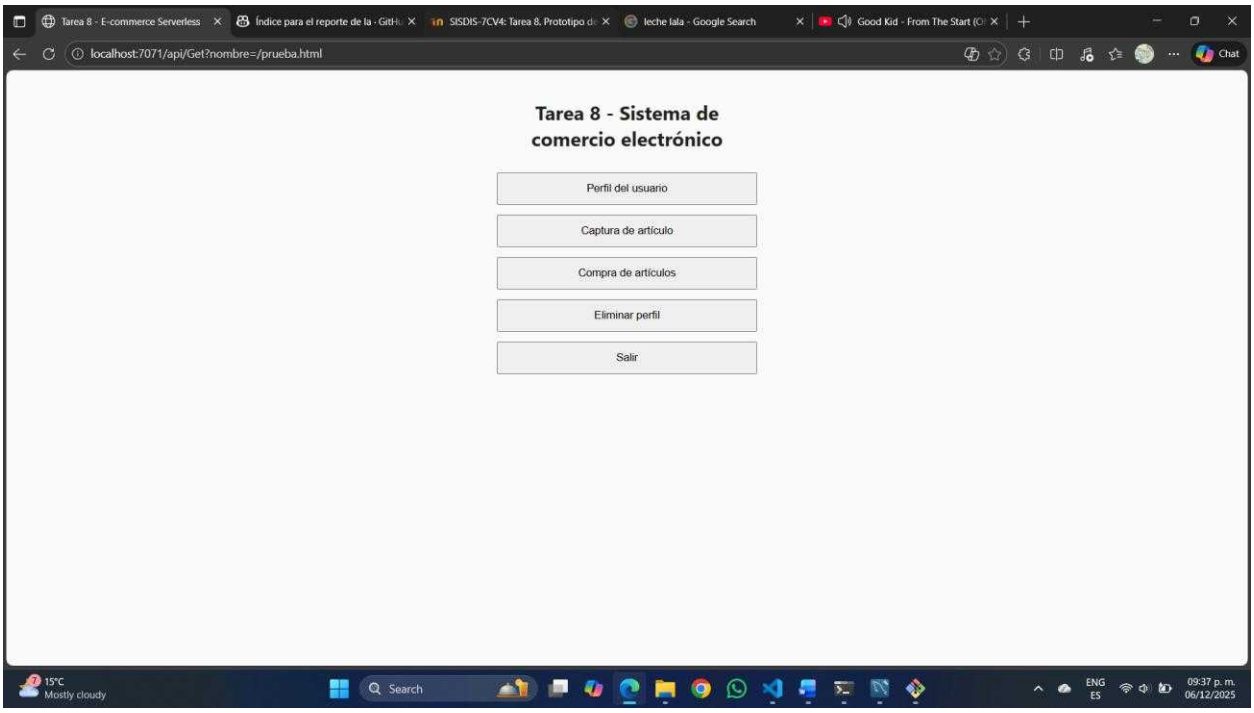


Figura 23. Captura del menú con las nuevas opciones visibles tras iniciar sesión.

7.3 Pantalla “Captura de artículo”

Se realizó la pantalla para alta de artículos que incluye campos: nombre, descripción, precio, cantidad y fotografía. Se reutilizó la función readSingleFile para convertir la imagen a base64 y se invocó el endpoint alta_articulo mediante cliente.post("alta_articulo", {...}) con id_usuario y token.

- Se realizó la validación mínima: campos obligatorios y formato de precio/cantidad.
- Se accedió al servidor y se obtuvo respuesta de éxito o error con mensajes JSON.

Tabla de campos utilizados:

Campo	Tipo	Descripción
nombre	texto	Nombre del artículo

descripcion	texto	Descripción del artículo
precio	número/decimal	Precio unitario
cantidad	número entero	Existencia inicial
foto	base64 (opcional)	Imagen del artículo

Tabla 9 Tabla de campos

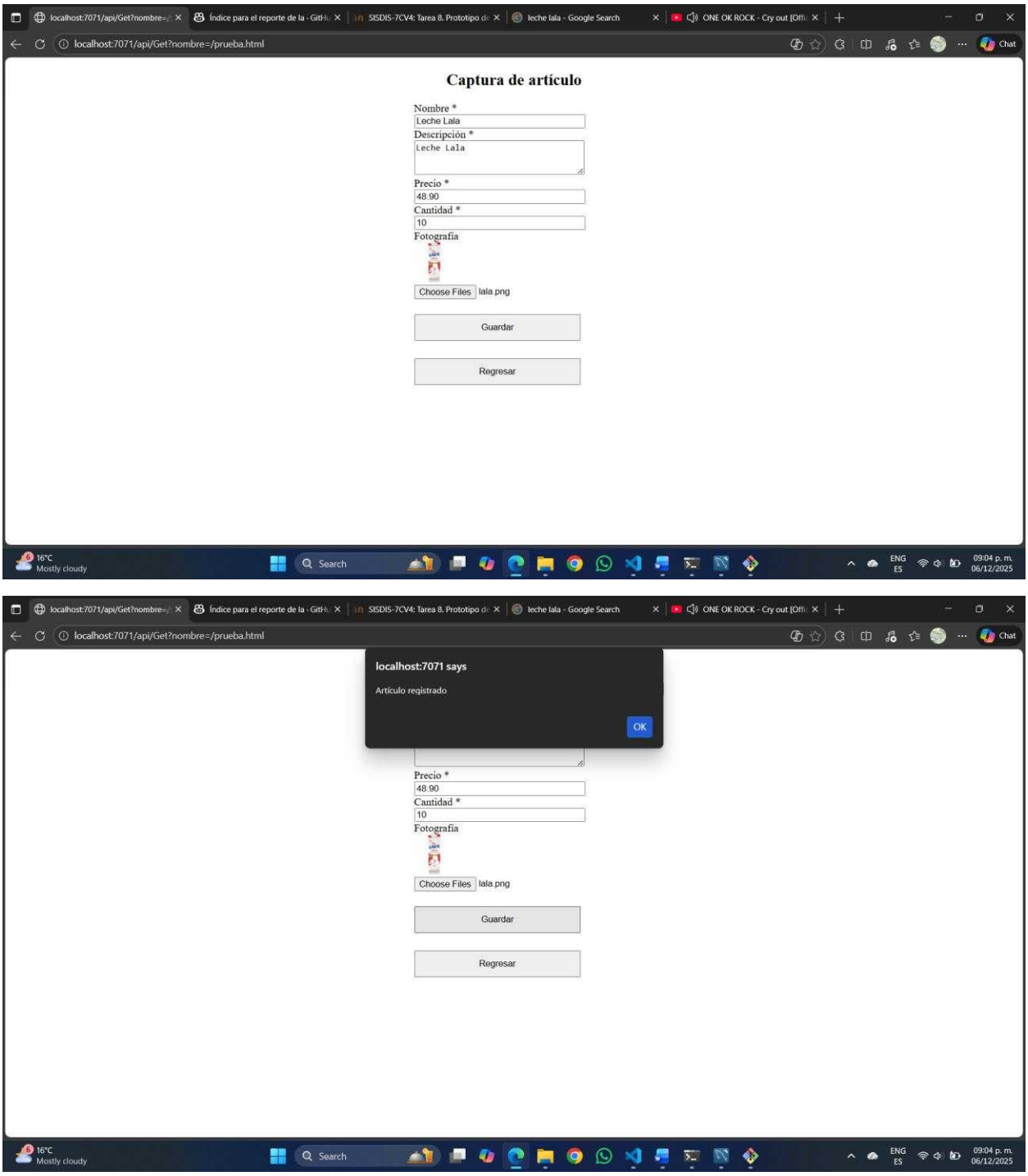


Figura 24. Pantalla de captura de artículo con imagen cargada y respuesta exitosa tras el envío.

7.4 Pantalla “Compra de artículos”

Se realizó la vista de compra en la que se accedió a consulta_articulos con una palabra_clave. Se mostraron resultados con mini-foto, nombre, descripción, precio, un campo de cantidad por cada artículo (default 1), y botones “+” y “-” para ajustar cantidad con el endpoint modifica_carrito_compra. Se instaló el botón “Compra” que invoca compra_articulo.

- Se accedió a la búsqueda con cliente.get("consulta_articulos", { palabra_clave, id_usuario, token }).
- Se realizó el control de cantidad por artículo, verificando las respuestas de validación del back-end.
- Se confirmó que al presionar “Compra” se registra la cantidad en el carrito y se descuenta del stock.

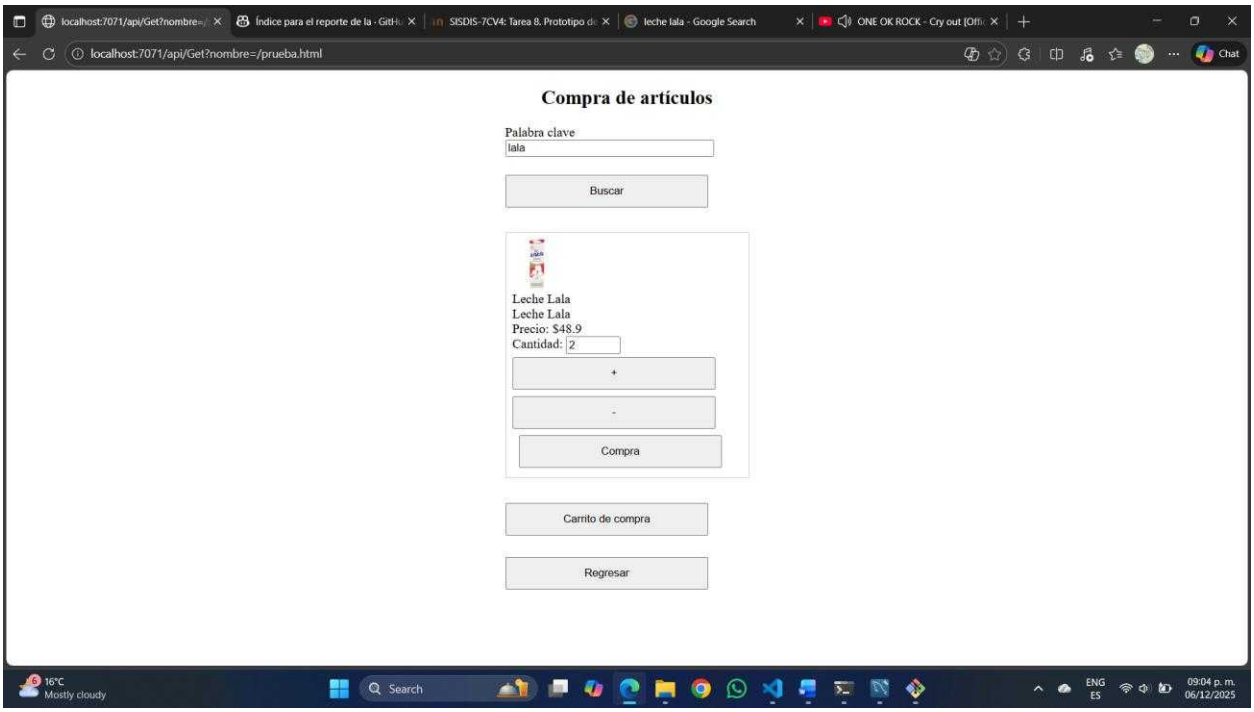


Figura 25. Listado de artículos mostrando imagen, precio, y controles de cantidad con el botón “Compra”.

7.5 Pantalla “Artículos en el carrito”

Se realizó la pantalla de carrito que muestra los artículos agregados: mini-foto, nombre, cantidad, precio y costo (cantidad x precio), además del total. Se instaló el botón “Eliminar artículo” que invoca elimina_articulo_carrito_compra y el botón “Eliminar carrito” que invoca elimina_carrito_compra. Se accedió al botón “Seguir comprando” para regresar a la vista de compra.

- Se realizó el cálculo de costos por artículo y total de la compra en el cliente.
- Se accedió a las operaciones de eliminación y se verificó la restauración de stock en las respuestas del servidor.

Tabla de acciones:

Acción	Endpoint	Método	Efecto
Ver carrito	(consulta local/refresh)	GET	Mostrar artículos, cantidades y total
Eliminar artículo	elimina_articulo_carrito_compra	DELETE	Sumar cantidad a stock y borrar del carrito
Eliminar carrito	elimina_carrito_compra	DELETE	Vaciar carrito y restaurar stock
Seguir comprando	Navegación	-	Regresar a búsqueda y lista de artículos

Tabla 10 Tabla de acciones

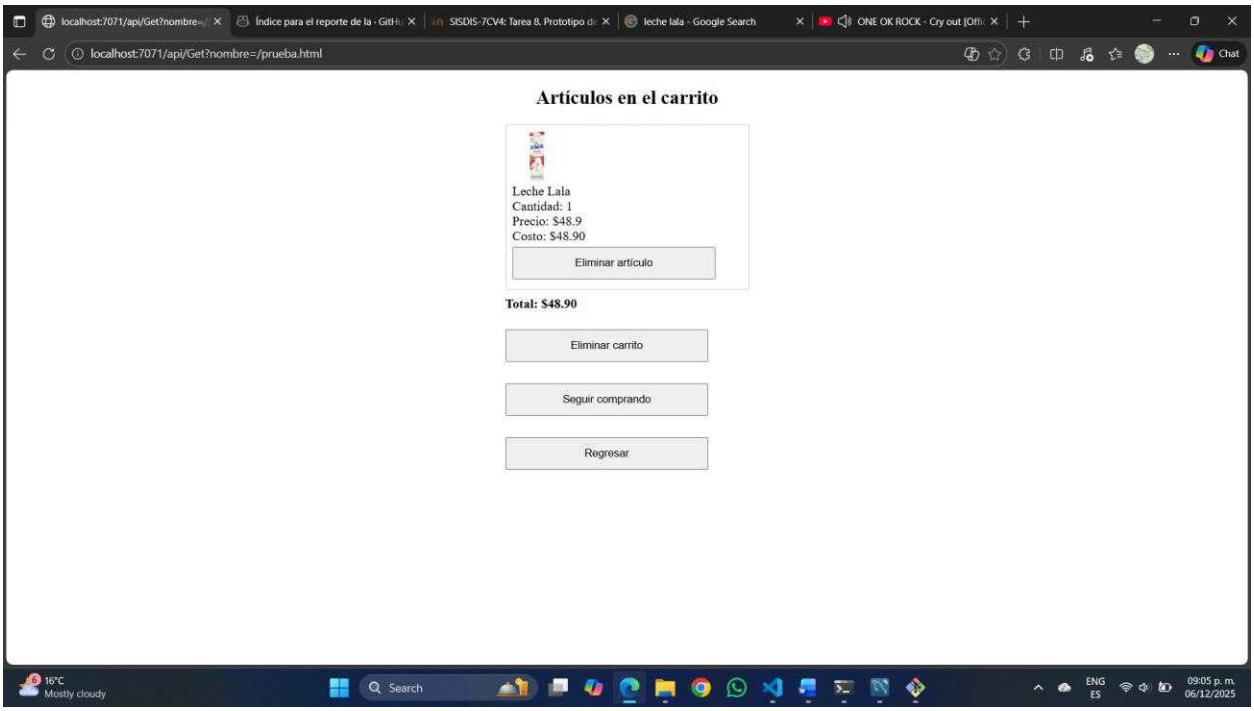


Figura 26. Vista del carrito mostrando costos por artículo, total, y botones de eliminación y navegación.

7.6 Integración con WSClient.js

Se realizó la integración de llamadas HTTP usando WSClient.js, aprovechando:

- get(operacion, args, callback) para consultas y login.
- post(operacion, body, callback) para altas, compras y registro.
- put(operacion, args, body, callback) para modificaciones.
- delete(operacion, args, callback) para eliminaciones.

Se accedió a los endpoints por medio de rutas /api/... y se validó la codificación de parámetros (uso de encodeURI y reemplazos de caracteres), además del manejo de Content-Type adecuado según método. Se realizaron pruebas de error y éxito, mostrando alertas con el contenido de los mensajes devueltos.

- Se realizó la verificación de callbacks y códigos HTTP.
- Se accedió a la consola del navegador para validar requests/responses.

8 Despliegue en Azure

Se realizó el despliegue del prototipo en Azure for Students, manteniendo la región Canada y la nomenclatura exigida por la tarea. Se instaló la base de datos en MySQL PaaS, se accedió a una cuenta de almacenamiento de Azure Files para hospedar el front-end y se publicó la Azure Function App configurando variables de entorno y el montaje de archivos. Se validó el acceso final mediante la URL pública del endpoint Get.

8.1 Creación de Azure Database for MySQL (t8mysql2022630278) e inicialización de esquema

Se realizó la creación de la instancia MySQL Flexible Server en Azure, con IP pública habilitada y reglas de firewall para acceso controlado desde la Function App y, temporalmente, desde la estación local. Se accedió al servidor con el usuario remoto (sin @localhost) y se ejecutó el esquema de la base de datos “servicio_web”, incluyendo las tablas iniciales y las tablas adicionales del e-commerce.

- Se instaló la instancia MySQL en la región Canada.
- Se accedió mediante MySQL Workbench/CLI con el usuario remoto y contraseña configurada.
- Se realizó la ejecución del script de creación de tablas: usuarios, fotos_usuarios, stock, fotos_articulos, carrito_compra e índice único en (id_usuario, id_articulo).

Tabla de verificación del esquema:

Recurso	Estado	Observaciones
Base de datos servicio_web	Creada	Acceso con usuario remoto
Tabla usuarios	OK	Índice único en email
Tabla fotos_usuarios	OK	FK hacia usuarios(id_usuario)
Tabla stock	OK	PK id_articulo AUTO_INCREMENT
Tabla fotos_articulos	OK	FK hacia stock(id_articulo)
Tabla carrito_compra	OK	UNIQUE (id_usuario, id_articulo)

Tabla 11 Tabla de verificación del esquema