Problemática 2: Exposición de API REST para deudas de clientes

1. Descripción general

En esta problemática se implementa una **API REST** que expone la información de deudas de clientes que fue cargada previamente en la base de datos durante la Problemática 1. El objetivo es permitir que sistemas externos puedan **consultar, crear, modificar y eliminar** deudas a través de un servicio web estándar, incorporando además un mecanismo de **autenticación básica** y la documentación automática de la API mediante **Swagger**.

De esta forma, la solución evoluciona desde un proceso de carga de datos por archivo plano (Problemática 1) hacia un servicio reutilizable y fácilmente integrable con otras aplicaciones (Problemática 2).

---

2. Arquitectura y tecnologías utilizadas

Para la construcción de la API se utilizó la siguiente arquitectura y stack tecnológico:

- **Backend:** Node.js + Express.
- **Base de datos:** MySQL, reutilizando la base deudas_clientes y la tabla deudas creada en la Problemática 1.
- **Conexión a BD:** librería mysql2/promise.
- **Configuración:** archivo .env gestionado con dotan para mantener credenciales y parámetros (host, usuario, password, puerto, usuario/clave de autenticación básica).
- **Documentación de la API:** swagger-ui-express + swagger-jsdoc.
- **Pruebas de la API:** Postman, con una colección que incluye las operaciones CRUD.

La aplicación se ejecuta con el comando:

node server.js en la **terminal de Visual Studio Code** en este caso.

levantando el servidor en http://localhost:3000 y http://localhost:3000/api-docs

---

3. Diseño de datos (reutilización de la Problemática 1)

En esta problemática se reutiliza el mismo modelo de datos definido anteriormente, basado en la tabla deudas de la base deudas_clientes. Las columnas principales son:

- id_deuda (VARCHAR): identificador único de la deuda.
- id_cliente (VARCHAR): identificador del cliente (RUT u otro código).
- nombre_cliente (VARCHAR): nombre del cliente.
- correo (VARCHAR): correo de contacto.
- monto_deuda (DECIMAL): monto de la deuda.

- fecha_vencimiento (DATE): fecha de vencimiento de la deuda.

En el diseño orientado a objetos esta tabla se representa con la clase **DeudaCliente**, lo que permite mantener coherencia entre la Problemática 1 y la Problemática 2.

---

4. Implementación de la API REST

La API fue desarrollada en el archivo server.js utilizando Express. Se definió un conjunto de endpoints bajo el prefijo /api/deudas que implementan el CRUD completo sobre la tabla deudas:

1. **Listar todas las deudas**
    - **Método:** GET
    - **Ruta:** /api/deudas
    - **Descripción:** retorna un arreglo con todas las deudas registradas en la base de datos.
    - **Respuesta exitosa:** 200 OK + JSON con la lista de deudas.
2. **Obtener una deuda por ID**
    - **Método:** GET
    - **Ruta:** /api/deudas/{id_deuda}
    - **Parámetros de ruta:** id_deuda (string)
    - **Descripción:** retorna la deuda asociada al identificador indicado.
    - **Respuestas:**
        - 200 OK + JSON con la deuda.
        - 404 Not Found si la deuda no existe.
3. **Crear una nueva deuda**
    - **Método:** POST
    - **Ruta:** /api/deudas
    - **Body (JSON):**
        - id_deuda, id_cliente, nombre_cliente, correo, monto_deuda, fecha_vencimiento.
    - **Descripción:** inserta un nuevo registro en la tabla deudas.
    - **Respuesta:** 201 Created + mensaje de confirmación.
4. **Actualizar una deuda existente**
    - **Método:** PUT

- **Ruta:** /api/deudas/{id_deuda}
- **Body (JSON):** mismos campos de la creación.
- **Descripción:** actualiza los datos de la deuda indicada.
- **Respuestas:**
  - 200 OK si la actualización fue exitosa.
  - 404 Not Found si no existe una deuda con ese id_deuda.

5. **Eliminar una deuda**
   - **Método:** DELETE
   - **Ruta:** /api/deudas/{id_deuda}
   - **Descripción:** elimina el registro correspondiente a la deuda indicada.
   - **Respuestas:**
     - 200 OK si la deuda fue eliminada.
     - 404 Not Found si el registro no existe.

Todas estas operaciones se implementan utilizando un **pool de conexiones MySQL** y consultas parametrizadas, garantizando un acceso eficiente y seguro a la base de datos.

---

5. Autenticación básica (Basic Auth)

Para controlar el acceso a la API se desarrolló un **middleware de autenticación básica** (AuthMiddleware). Este componente:

- Lee la cabecera HTTP Authorization: Basic ....
- Decodifica las credenciales en formato usuario:clave (Base64).
- Compara el usuario y la clave con los valores configurados en variables de entorno (BASIC_USER y BASIC_PASS).
- Si las credenciales son inválidas:
  - Retorna 401 Unauthorized si no se envió credencial.
  - Retorna 403 Forbidden si las credenciales no coinciden.
- Si las credenciales son correctas:
  - Permite el paso de la petición hacia las rutas bajo /api.

El middleware se aplica a todas las rutas de la API mediante:

app.use('/api', basicAuth);

De esta forma, cualquier cliente que desee consumir los endpoints debe autenticar primero, cumpliendo con el requisito de seguridad de la problemática.

---

6. Documentación de la API con Swagger

La API se documentó utilizando **Swagger OpenAPI 3.0**, definiendo:

- El esquema Deuda con sus atributos principales.
- La descripción y parámetros de cada endpoint (GET, POST, PUT, DELETE).
- Los códigos de respuesta (200, 201, 404, 401, 403, 500).
- El esquema de seguridad basicAuth para indicar que las rutas están protegidas con autenticación básica.

La documentación se genera automáticamente a partir de anotaciones en server.js y se expone en:

- **URL:** http://localhost:3000/api-docs

Desde esa interfaz es posible probar los métodos de la API, visualizando de forma amigable las entradas, salidas y ejemplos de JSON.

---

7. Colección de Postman para pruebas

Para validar el correcto funcionamiento de la API, se creó una **colección de Postman** llamada, por ejemplo, Problemática 2 - API Deudas, que contiene los siguientes requests:

1. GET /api/deudas – Lista todas las deudas.
2. GET /api/deudas/{id_deuda} – Consulta una deuda específica.
3. POST /api/deudas – Crea una nueva deuda (enviando un JSON en el body).
4. PUT /api/deudas/{id_deuda} – Actualiza una deuda existente.
5. DELETE /api/deudas/{id_deuda} – Elimina una deuda.

En todos los requests se configuró la pestaña **Authorization** con **Basic Auth**, utilizando las mismas credenciales definidas en el archivo .env.

Finalmente, la colección fue **exportada a un archivo .json**, cumpliendo el requisito del enunciado para entregar un conjunto de pruebas reproducibles de la API.

---

8. Diagrama de clases de la solución

Para modelar la estructura interna de la API se elaboró un **diagrama de clases** que reutiliza la entidad DeudaCliente ya utilizada en la Problemática 1, e incorpora nuevas clases de servicio y acceso a datos:

- DeudaCliente: representa la deuda de un cliente con los atributos idDeuda, idCliente, nombre, correo, monto y fechaVto.

- ConexionBD: encapsula los datos de conexión a MySQL (host, usuario, password, nombreBD) y provee métodos para conectar, ejecutar consultas y cerrar la conexión.
- DeudaRepository: utiliza ConexionBD para ejecutar operaciones CRUD sobre la tabla deudas (findAll, findById, save, update, delete).
- DeudaService: concentra la lógica de negocio y expone métodos de alto nivel (listarDeudas, obtenerDeuda, crearDeuda, actualizarDeuda, eliminarDeuda), delegando al repositorio el acceso físico a los datos.
- ApiDeudas: actúa como controlador REST, implementando los endpoints /api/deudas y utilizando DeudaService para resolver las solicitudes HTTP.
- AuthMiddleware: valida las credenciales de usuario mediante autenticación básica antes de que el controlador atienda las peticiones.

En conjunto, el diagrama de clases muestra una arquitectura por capas donde se separan claramente la presentación (controlador), la lógica de negocio (servicio) y el acceso a datos (repositorio + conexión a base de datos), reutilizando el mismo modelo de DeudaCliente definido en la Problemática 1.

---

9. Diagrama de secuencia de consulta de deuda por ID

El **diagrama de secuencia** describe el flujo dinámico cuando un cliente consulta una deuda específica por su identificador:

1. El **Cliente REST** (Postman) envía una petición GET /api/deudas/{idDeuda} agregando la cabecera Authorization: Basic ....
2. ApiDeudas invoca a AuthMiddleware para ejecutar validarCredenciales(usuario, clave).
3. Si las credenciales son válidas, AuthMiddleware responde a ApiDeudas; en caso contrario, la petición se corta devolviendo un error 401 o 403.
4. ApiDeudas delega el procesamiento en DeudaService mediante el método obtenerDeuda(idDeuda).
5. DeudaService llama a DeudaRepository usando findById(idDeuda).
6. DeudaRepository utiliza ConexionBD para ejecutar un SELECT * FROM deudas WHERE id_deuda = idDeuda sobre la base de datos MySQL.
7. La base de datos devuelve el registro encontrado a DeudaRepository.
8. DeudaRepository construye un objeto DeudaCliente y lo retorna a DeudaService.
9. DeudaService devuelve el objeto DeudaCliente a ApiDeudas.
10. Finalmente, ApiDeudas responde al **Cliente REST** con un HTTP 200 OK y el cuerpo JSON correspondiente a la deuda solicitada.

Este diagrama evidencia la interacción entre las distintas capas (controlador, servicio, repositorio y base de datos) y la aplicación del middleware de autenticación en cada petición a la API.

---

10. Relación con la Problemática 1 y conclusión

La **Problemática 1** se enfocó en la construcción de un proceso para **cargar un archivo de deudas de clientes a una base de datos**, utilizando las clases AppCargaDeudas, DeudaCliente y ConexionBD.

Sobre esa base, la **Problemática 2** toma la misma estructura de datos (DeudaCliente y la tabla deudas) y la expone a través de una **API REST segura**, documentada y fácilmente consumible por otras aplicaciones.

Como resultado, se obtiene una solución completa donde:

- Los datos se cargan y se almacenan de manera estructurada en MySQL (P1).
- La información se publica mediante un servicio web estándar con autenticación básica, documentación Swagger y pruebas en Postman (P2).

Esto demuestra un ciclo de desarrollo coherente: **ingestión de datos, persistencia y exposición de servicios**, cumpliendo con los requisitos técnicos de la prueba y buenas prácticas de diseño por capas.