Evaluación Sumativa – Unidad 1

1. Datos de Identificación del Estudiante y la Práctica

Nombre del estudiante(s)	Stalin Joel Tapia Pinta
Asignatura	Desarrollo de Plataformas
Ciclo	Quinto
Unidad	1
Título de la Práctica	Aplicaciones Web Tecnologías de Lado del Servidor
Nombre del Docente	Edison Coronel
Fecha	Jueves 6 de noviembre

2. Informe técnico

1. Descripción de la Arquitectura

El backend del proyecto **GastanGO** es una API RESTful monolítica modular desarrollada con **Node.js** y el framework **Express**. La arquitectura está diseñada para ser escalable y mantenible, separando las responsabilidades en las siguientes carpetas principales:

- /src/config: Centraliza la configuración de la base de datos (PostgreSQL) y la documentación de la API (Swagger).
- /src/models: Define los esquemas de datos y sus relaciones utilizando el ORM **Sequelize**. Incluye los modelos User, Transaction y Budget, con un archivo index.js que gestiona las asociaciones.
- /src/routes: Contiene los archivos que definen los endpoints de la API (ej. /auth, /transactions). También aplica los middlewares de validación.
- /src/controllers: Alberga la lógica de negocio. Recibe las peticiones de las rutas, interactúa con los modelos y envía las respuestas (ej. auth.controller.js, transaction.controller.js).
- /src/middlewares: Contiene funciones intermedias, destacando auth.middleware.js, que protege las rutas mediante la verificación de tokens JWT.
- /src/services: Implementa lógica desacoplada, como el notification.service.js, que simula un servicio de notificaciones y actúa como un módulo independiente.

2. Dependencias Clave del Proyecto

El funcionamiento del backend se basa en las siguientes dependencias NPM, definidas en package.json:

- **Framework y Servidor:**
  - express: Framework web para la creación de la API.
- **Base de Datos y ORM:**
  - sequelize: ORM (Object-Relational Mapper) para interactuar con la base de datos.
  - pg: Driver de Node.js para PostgreSQL.
- **Seguridad y Autenticación:**
  - bcryptjs: Librería para el hashing (cifrado) de contraseñas.
  - jsonwebtoken: Para la generación y verificación de JSON Web Tokens (JWT).
  - helmet: Añade cabeceras HTTP de seguridad para proteger la aplicación.
  - cors: Habilita el Cross-Origin Resource Sharing.
- **Utilidades y Validación:**
  - dotenv: Carga variables de entorno desde un archivo .env.
  - express-validator: Middleware para la validación y sanitización de los datos de entrada.
- **Documentación y Desarrollo:**
  - swagger-jsdoc y swagger-ui-express: Generan y sirven la documentación interactiva de la API.
  - nodemon: Reinicia automáticamente el servidor durante el desarrollo.

3. Documentación de la API (Swagger)

El proyecto integra **Swagger** para generar documentación automática de la API a partir de comentarios JSDoc en los archivos de rutas.

La documentación interactiva está disponible en la siguiente ruta mientras el servidor está en ejecución: **URL:** http://localhost:3000/api-docs

4. Informe de Seguridad (Principios OWASP)

Se han implementado medidas de seguridad clave para mitigar riesgos comunes del Top 10 de OWASP:

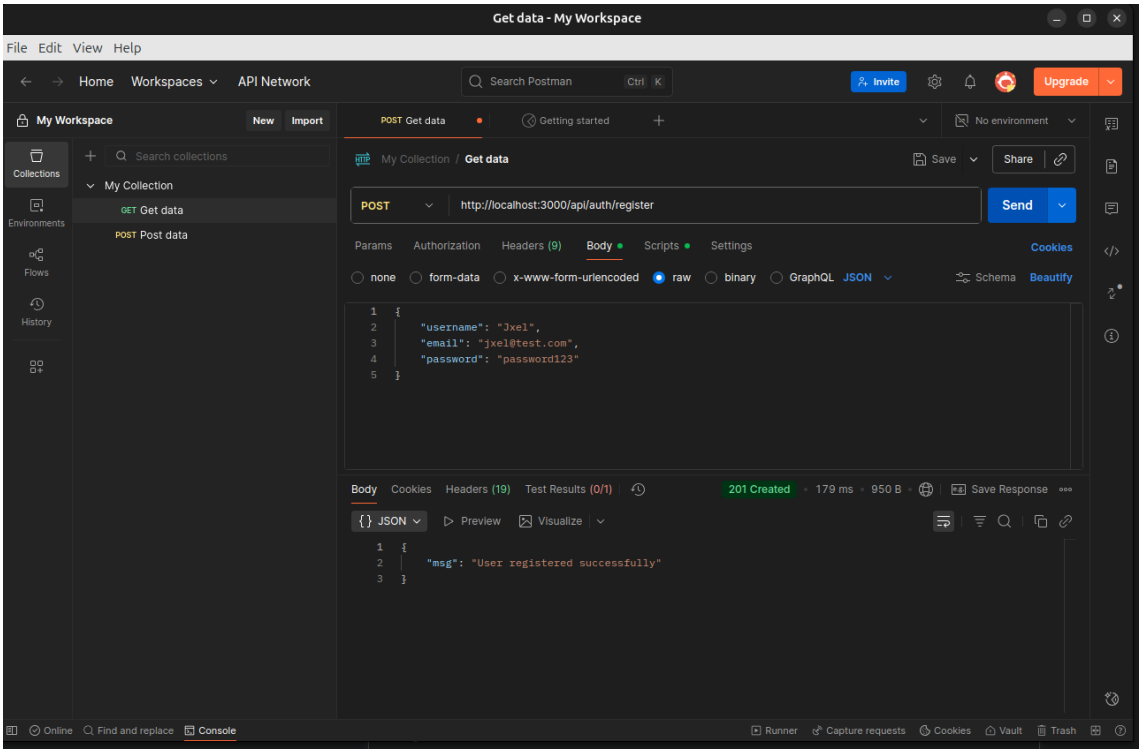
1. **Cifrado de Contraseñas (OWASP A02: Fallos Criptográficos):**
   - **Implementación:** Las contraseñas de los usuarios nunca se almacenan en texto plano. Se utiliza bcryptjs para hashear las contraseñas antes de guardarlas en la base de datos.
   - **Evidencia:** El modelo User.model.js usa un "hook" beforeCreate de Sequelize para hashear automáticamente la contraseña con un salt de 10 rondas. La verificación en el login se realiza con bcrypt.compare.
2. **Autenticación y Autorización JWT (OWASP A01: Control de Acceso Roto):**
   - **Implementación:** El acceso a los endpoints sensibles está protegido.
   - **Evidencia:**
     - **Autenticación:** Al hacer login, auth.controller.js genera un **JSON Web Token (JWT)** firmado con un secreto (JWT_SECRET) y un tiempo de expiración.
     - **Autorización:** Las rutas como /api/transactions y /api/users/me utilizan el middleware verifyToken. Este middleware comprueba la validez del Bearer Token en la cabecera Authorization, denegando el acceso si el token es inválido o no se proporciona.

3. **Validación de Entradas (OWASP A03: Inyección):**
    - **Implementación:** Toda la información proveniente del cliente es validada y sanitizada.
    - **Evidencia:** Se utiliza express-validator en las definiciones de rutas (ej. auth.routes.js, transaction.routes.js). Se aplican reglas como .isEmail(), .isLength({ min: 6 }), .isFloat({ gt: 0 }) y .isIn(['expense', 'income']), previniendo datos malformados y mitigando riesgos de inyección.

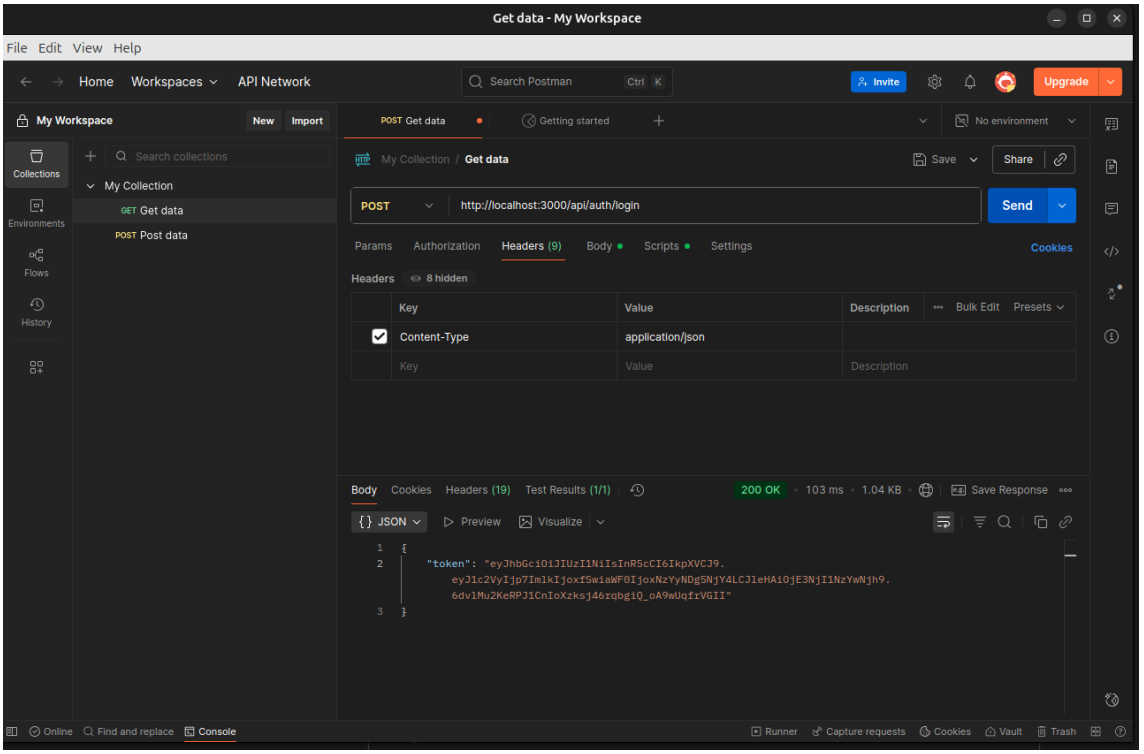
5. Evidencia de Funcionamiento (Pruebas en Postman)

A continuación, se presenta la evidencia del funcionamiento de los endpoints de la API utilizando Postman.

(5.1) Endpoint de Registro de Usuario: POST /api/auth/register Se envía un username, email y password. La API responde con un 201 Created y un mensaje de éxito.

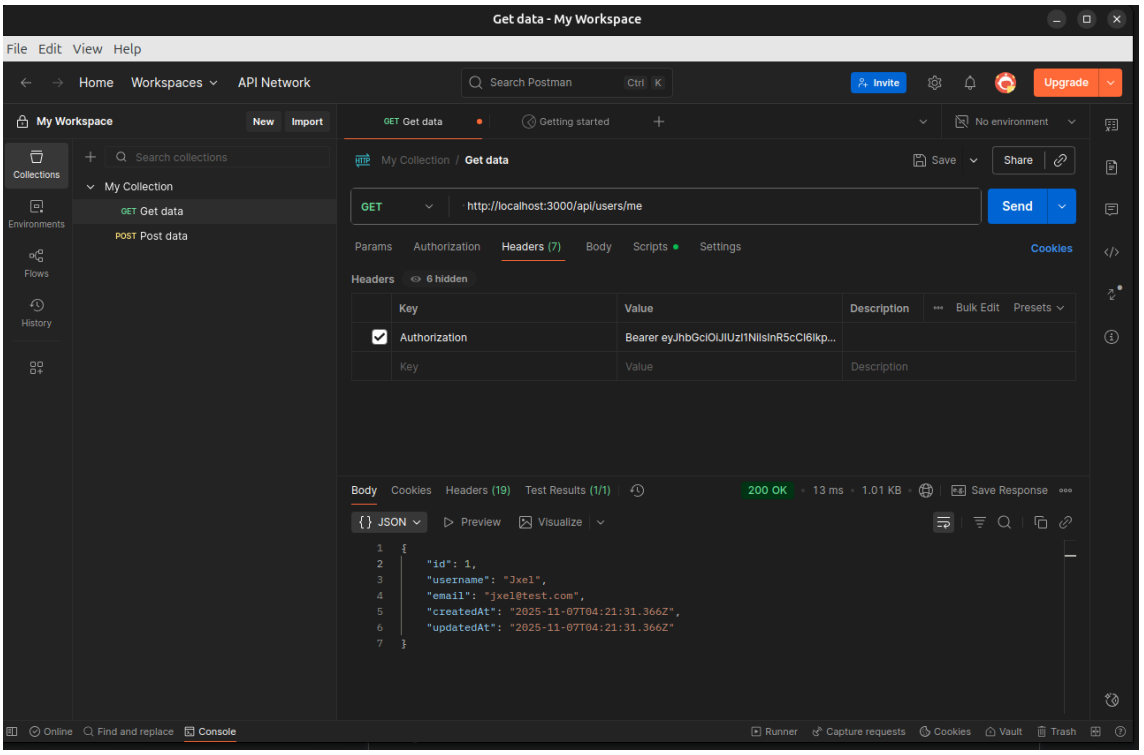


(5.2) Endpoint de Inicio de Sesión: POST /api/auth/login Se envía el email y password del usuario registrado. La API responde con un 200 OK y el token JWT.

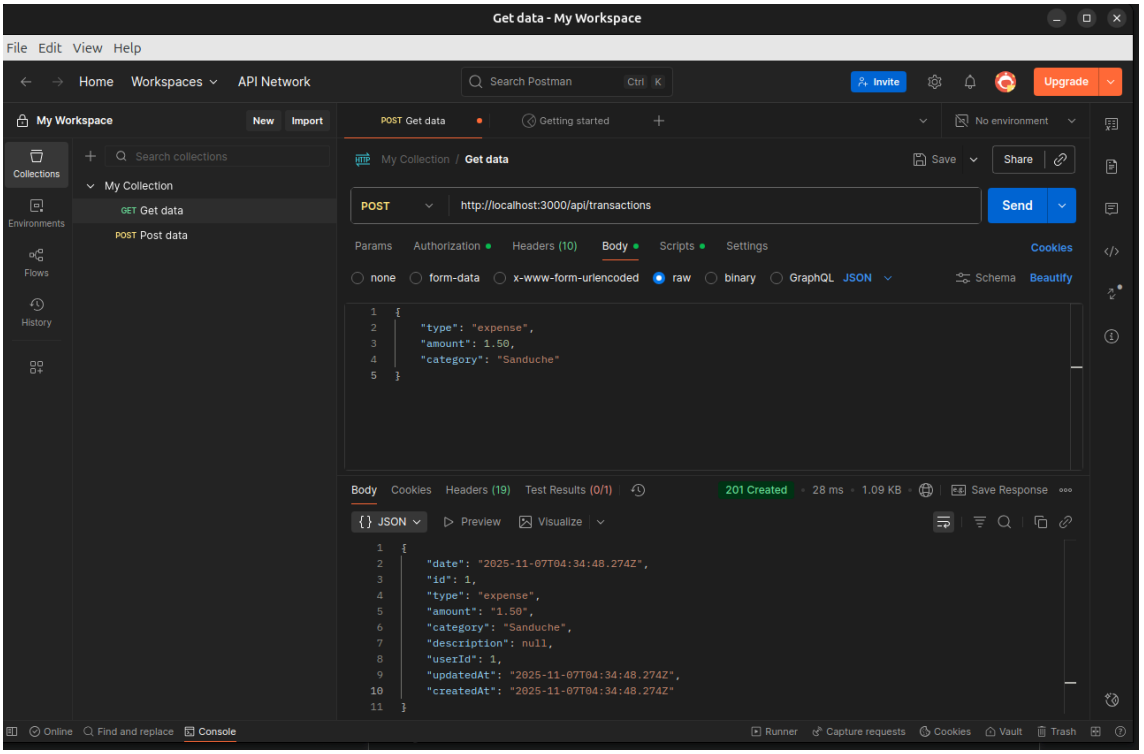


curl -X POST http://localhost:3000/api

(5.3) Endpoint Protegido: GET /api/users/me Se realiza una petición GET utilizando el token JWT del paso anterior en la pestaña Authorization (Tipo: Bearer Token). La API responde con los datos del usuario autenticado.



(5.4) Endpoint Protegido: POST /api/transactions Se crea una nueva transacción (gasto) enviando el type, amount y category en el Body, junto con el token JWT en la autorización. La API responde con 201 Created y el objeto de la transacción creada.



(5.5) Endpoint Protegido: GET /api/transactions Se realiza una petición GET con el token JWT para listar todas las transacciones del usuario. La API responde con 200 OK y un array que incluye la transacción creada en el paso anterior.