INFORME TÉCNICO DE DESARROLLO DE LA API TAM

1. Introducción:

La API TAM fue desarrollada como una solución backend orientada a la gestión de información y procesos relacionados con usuarios, productos, pedidos, carritos de compra, soporte y autenticación. La arquitectura implementada sigue el modelo de API REST, permitiendo la comunicación entre aplicaciones cliente y el servidor mediante peticiones HTTP.

El proyecto fue construido utilizando tecnologías modernas del ecosistema JavaScript y TypeScript, enfocándose en la modularidad, escalabilidad y facilidad de mantenimiento.

2. Objetivo de la API:

El objetivo principal de la API es centralizar la lógica de negocio y el acceso a la base de datos para diferentes módulos funcionales del sistema.

Entre sus principales funcionalidades se encuentran:

- Gestión de usuarios.
- Gestión de productos.
- Gestión de roles.
- Gestión de pedidos.
- Gestión de carrito de compras.
- Historial de comparaciones.
- Gestión de proveedores.
- Soporte técnico.
- Sistema de autenticación y seguridad.

La API permite que aplicaciones frontend o aplicaciones externas consuman los servicios mediante endpoints organizados.

3. Tecnologías Utilizadas:

3.1 Node.js:

Se utilizó Node.js como entorno de ejecución del lado del servidor.

Funciones principales:

- Ejecutar JavaScript fuera del navegador.
- Gestionar peticiones HTTP.
- Manejar procesos asincrónicos.
- Permitir alta concurrencia.

Ventajas:

- Alto rendimiento.
- Arquitectura no bloqueante.
- Excelente integración con APIs REST.
- Amplio ecosistema de librerías.

3.2 TypeScript:

El proyecto fue desarrollado utilizando TypeScript.

TypeScript añade tipado estático sobre JavaScript, permitiendo detectar errores durante el desarrollo.

Evidencia técnica:

- Archivo tsconfig.json
- Archivos fuente con extensión .ts

Beneficios:

- Mayor control de tipos.
- Código más mantenible.
- Mejor organización.
- Reduce errores en producción.

3.3 Express.js

Se utilizó Express.js como framework principal para la creación del servidor web y las rutas de la API.

Funciones principales:

- Gestión de rutas.
- Manejo de middleware.
- Recepción de solicitudes HTTP.
- Respuesta en formato JSON.

Ejemplo técnico:

```
this.router.use("/products", product);
this.router.use("/users", user);
this.router.use("/orders", order);
```

3.4 MySQL:

La base de datos utilizada fue MySQL.

Funciones:

- Almacenar información persistente.
- Gestionar relaciones entre entidades.
- Ejecutar consultas SQL.

Características implementadas:

- Consultas parametrizadas.
- Conexiones asincrónicas.
- Manejo de resultados mediante promesas.

Ejemplo:

```
const [results] = await connection.query(sql, params);
```

3.5 mysql2:

Se utilizó la librería mysql2 para conectar Node.js con MySQL.

Ventajas:

- Compatible con promesas.
- Mejor rendimiento.
- Manejo asincrónico.
- Consultas seguras.

Implementación:

```
async function query(sql:string, params:any) {
 const connection = await mysql.createConnection(config.db);
 const [results] = await connection.query(sql, params);
 return results;
}
```

3.6 JWT (JSON Web Token):

La autenticación fue implementada mediante JWT.

Funciones:

- Generar tokens de acceso.
- Validar sesiones.
- Permitir autenticación segura.

Proceso implementado:

1. El usuario envía email y contraseña.
2. La API valida las credenciales.
3. Se genera un token.
4. El cliente utiliza el token para acceder a rutas protegidas.

3.7 bcrypt:

La librería bcrypt fue utilizada para el cifrado y validación de contraseñas.

Funciones:

- Comparar contraseñas cifradas.
- Incrementar la seguridad de autenticación.
- Evitar almacenamiento de contraseñas en texto plano.

Ejemplo:

```
const validPassword = await bcrypt.compare(
 data.password,
 user.password
);
```

3.8 dotenv:

La configuración de variables de entorno fue manejada mediante dotenv.

Funciones:

- Cargar variables desde un archivo .env.
- Proteger configuraciones sensibles.
- Separar configuración del código.

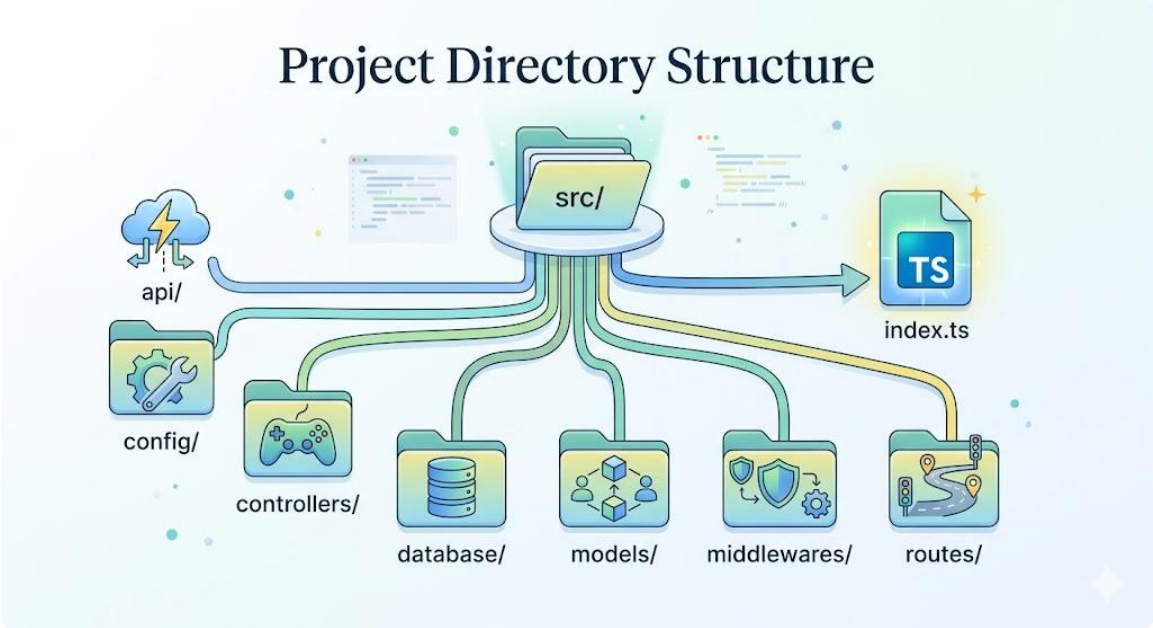
Ejemplo:

```
dotenv.config();
```

4. Arquitectura del Proyecto:

La API fue desarrollada siguiendo una arquitectura modular basada en capas.

Estructura general:



4.1 Capa de Rutas:

Las rutas se encargan de definir los endpoints de la API.

Funciones:

- Recibir solicitudes HTTP.
- Redireccionar al controlador correspondiente.

4.2 Capa de Controladores:

Los controladores contienen la lógica de negocio.

Funciones:

- Validar información.
- Procesar solicitudes.
- Conectarse con la base de datos.
- Retornar respuestas.

4.3 Capa de Modelos:

Los modelos definen las estructuras de datos utilizadas por la aplicación.

Ejemplo:

```
export interface Auth {
 email: string;
 password: string;
}
```

4.4 Capa de Base de Datos:

La capa de base de datos centraliza las conexiones y consultas SQL.

Ejemplo:

```
const connection = await mysql.createConnection(config.db);
```

5. Funcionamiento de la API:

El punto de entrada principal es:

src/index.ts

Proceso:

1. Se cargan variables de entorno.
2. Se crea la instancia de Express.
3. Se configuran rutas.
4. Se inicia el servidor HTTP.

Ejemplo técnico:

```
const api: API = new ExpressApi(BASE_URL);
const httpServer = new ServerHTTP(host, port, api.createServer());
```

La API utiliza el protocolo HTTP y maneja métodos GET, POST, PUT y DELETE.

6. Seguridad Implementada:

6.1 Autenticación:

Se implementó autenticación mediante JWT.

Beneficios:

- Seguridad.
- Manejo de sesiones.
- Control de acceso.

6.2 Protección de Contraseñas:

Las contraseñas son validadas mediante bcrypt.

Beneficios:

- Protección de credenciales.
- Prevención de ataques.
- Mayor seguridad de usuarios.

6.3 Consultas Parametrizadas:

La API utiliza parámetros en las consultas SQL.

Ejemplo:

SELECT * FROM usuarios WHERE email = ?

Beneficios:

- Prevención de SQL Injection.
- Mayor seguridad.

7. Módulos Funcionales Identificados:

- Auth
- Users
- Products
- Orders
- Cart
- CartDetail
- Support
- Supplier
- Category
- Subcategory
- Role
- Calification
- UserAddress
- UserPaymentMethod
- PhoneUser
- ActionControl

8. Compilación y Ejecución:

La aplicación utiliza TypeScript compilado hacia JavaScript.

Ejemplo:

"start": "npx tsc && node dist/src/index.js"

Proceso:

1. TypeScript compila el código.
2. Los archivos compilados se almacenan en dist/
3. Node.js ejecuta la aplicación compilada.

9. Herramientas Complementarias:

Postman fue utilizado para:

- Probar endpoints.
- Simular peticiones.
- Validar respuestas.
- Documentar la API.

10. Flujo General de Desarrollo:

Etapas implementadas:

1. Diseño de arquitectura.
2. Configuración del entorno.
3. Desarrollo de endpoints.
4. Integración con base de datos.
5. Implementación de autenticación.
6. Pruebas con Postman.

11. Ventajas Técnicas de la Solución:

-Escalabilidad: La estructura modular permite agregar nuevos módulos fácilmente.
-Mantenibilidad: El uso de TypeScript mejora la organización del código.
-Seguridad: Se implementaron JWT, bcrypt y consultas parametrizadas.
-Rendimiento: Node.js permite manejo eficiente de múltiples conexiones.
-Organización: Existe separación clara entre rutas, lógica, modelos y base de datos.

12. Conclusión:

La API TAM fue desarrollada utilizando una arquitectura moderna basada en Node.js, Express y TypeScript, integrando MySQL como sistema gestor de base de datos.

El proyecto implementa buenas prácticas de desarrollo backend mediante separación por capas, autenticación segura con JWT, protección de contraseñas con bcrypt y consultas parametrizadas para mejorar la seguridad.

La estructura modular permite que la API sea escalable, mantenible y preparada para integrarse con aplicaciones frontend, aplicaciones móviles o sistemas externos.