Documentación Técnica: QuetzAI

Autores: Marthon Leobardo Yañez Martínez, Eduardo Muñoz González

Tecnologías: Flutter, FastAPI, MySQL, Google Gemini AI, Docker.

1. Visión General del Proyecto

QuetzAI es un sistema interactivo para museos que utiliza etiquetas NFC e Inteligencia Artificial Generativa. Su objetivo es enriquecer la experiencia del visitante ofreciendo información contextualizada según su edad y generando trivias educativas en tiempo real.

Arquitectura del Sistema

El sistema sigue un patrón **Cliente-Servidor**:

1. **Cliente (Móvil):** App en Flutter que lee tags NFC y presenta la información.
2. **Servidor (Backend):** API REST en Python (FastAPI) que orquesta la lógica.
3. **Base de Datos:** MySQL para persistencia de visitantes, objetos y métricas.
4. **Motor de IA:** Google Gemini (1.5 Flash) para generación de contenido (RAG - Retrieval Augmented Generation).

---

2. Estructura del Repositorio y Archivos Clave

A continuación, se detallan los directorios y archivos más críticos para el desarrollo.

`server/` (Backend & Lógica de Negocio)

Es el núcleo del procesamiento. Aquí reside la API y la conexión con la IA.

- **`Backend.py`**: **(Punto de entrada)**
  - Inicializa la app `FastAPI`.
  - Define los endpoints (`/registrar_visitante`, `/scan_id`, `/trivia`).
  - Configura CORS para permitir peticiones desde el móvil.
- **`sqlConnector.py`**: Capa de acceso a datos (DAO). Contiene todas las funciones que ejecutan sentencias SQL directas (INSERT, SELECT, UPDATE) hacia MySQL.
- **`GeminiAPIResumen.py`**: Contiene la lógica para construir el *prompt* de resumen. Recibe el texto crudo de la BD y solicita a Gemini una versión adaptada a la edad del usuario.
- **`GeminiAPITrivia.py`**: Genera preguntas de opción múltiple ("Multiple Choice") basadas en el resumen previo. **Nota:** Actualmente solicita el output en formato JSON crudo.
- **`Dockerfile`**: Define el entorno de ejecución (Python 3.13-slim), instalación de dependencias y comando de arranque (`uvicorn`).

`lib/` (Frontend - Flutter)

Código fuente de la aplicación móvil.

- **main.dart**: Punto de entrada de la app Flutter.
- **nfc_screen.dart**: **(Archivo Crítico)** Maneja la sesión de lectura NFC.
    - Detecta tags NDEF.
    - Limpia el payload (elimina prefijos de idioma como \x02en).
    - Llama al backend cuando se detecta un tag válido.
- **services/api_service.dart**: Capa de red. Encapsula las peticiones HTTP (POST) hacia el backend. Maneja la serialización de JSON.

---

3. Guía del Desarrollador: Backend (API)

Configuración del Entorno

El backend requiere un archivo .env en la carpeta server/ con las siguientes variables:

```
Fragmento de código

DB_HOST=host.docker.internal
DB_PORT=3306
DB_NAME=museo
DB_USER=root
DB_PASSWORD=tu_password
```

Endpoints Principales

Método	Ruta	Descripción	Input (JSON)	Output (JSON)
POST	/registrar_visitante	Crea un usuario temporal.	{ "nombre": "Leo", "edad": 22 }	{ "id_visitante": 48 }
POST	/scan_id	Procesa un NFC leído. Busca el objeto, genera resumen con IA y guarda historial.	{ "scan_data": "HASH...", "id_visitante": 48 }	{ "resumen": "Texto generado..." }

POST	/trivia	Genera preguntas basadas en lo visto.	{ "id_visitante": 48, "noResumen": 1 }	{ "preguntas": [...] }

Lógica de IA (Prompts)

- **Resumen (GeminiAPIResumen.py):**
  - *Input:* Texto del objeto + Edad.
  - *Prompt actual:* Pide resumir en máximo 100 palabras, adaptado a la edad, y **solicita la respuesta en inglés** (según el código actual Summarize the following...).
- **Trivia (GeminiAPITrivia.py):**
  - *Input:* Resumen generado.
  - *Prompt actual:* Pide generar un JSON con pregunta, opciones, respuesta. **Nota:** El prompt actual fuerza el idioma inglés ("Dame una pregunta... en ingles").

---

4. Esquema de Base de Datos (MySQL)

Basado en Dump20250829.sql, las tablas clave son:

- **visitante**: Almacena Nombre, Edad y Puntuacion_trivia.
- **objeto_historico**: Catálogo del museo. Vincula un hash (del NFC) con Nombre_objeto e Informacion (texto largo).
- **resumen**: Guarda los textos generados por Gemini para no regenerarlos innecesariamente (aunque la lógica actual parece generar uno nuevo por visita/usuario).
- **trivia**: Almacena las preguntas generadas y la respuesta del usuario.
- **objetos_visitante**: Tabla pivote (Muchos a Muchos) para saber qué objetos ha visitado un usuario (Historial de escaneo).

---

5. Instrucciones de Despliegue (Deploy)

Dado que ya tienes el contenedor configurado para Docker Hub, el flujo de despliegue sugerido para un servidor (VPS tipo AWS/DigitalOcean/Azure) es:

Paso 1: Preparar la Base de Datos

El contenedor de la aplicación **no** incluye la base de datos. Debes tener una instancia de MySQL corriendo.

1. Instala MySQL en el servidor o usa un servicio gestionado (AWS RDS).
2. Ejecuta el script Dump20250829.sql para crear el esquema.

Paso 2: Publicar la Imagen (En tu máquina local)

Si hiciste cambios en el código Python:

```Bash
cd server
docker build -t leoyamm/quetzai:latest .
docker push leoyamm/quetzai:latest
```

Paso 3: Desplegar en el Servidor

En tu servidor Linux, crea un archivo `docker-compose.yml`:

```YAML
version: '3.8'
services:
  backend:
    image: leoyamm/quetzai:latest
    ports:
      - "8000:8000"
    environment:
      - DB_HOST=172.17.0.1  # IP del host desde el contenedor (en Linux) o la IP de tu MySQL
      - DB_PORT=3306
      - DB_NAME=museo
      - DB_USER=tu_usuario_prod
      - DB_PASSWORD=tu_password_prod
      - GEMINI_API_KEY=tu_api_key_prod
    restart: always
```

Ejecuta: `docker-compose up -d`

> **Nota importante de Red:** En `docker-compose.yml` local usas `host.docker.internal`. Esto suele funcionar en Docker Desktop (Windows/Mac). En servidores Linux puros, debes usar la IP de la red del docker (`172.17.0.1` usualmente) o la IP pública del servidor si la BD no está en el mismo contenedor.

---

6. Trabajo Futuro (Roadmap)

Esta sección describe las mejoras planificadas para evolucionar QuetzAI hacia un producto más robusto.

A. Selección de Idioma (Internacionalización)

Estado Actual: Los prompts en GeminiAPIResumen.py y GeminiAPITrivia.py tienen instrucciones explícitas (hardcoded) para responder en inglés ("Summarize...", "in english").

Solución Propuesta:

1. Agregar un campo idioma como selección en el modelo para la interfaz Visitante
2. Modificar los f-strings en Python para inyectar el idioma deseado.
    - *Código:* contents=f"Resume la siguiente información... en el idioma {idioma_usuario}..."

Los archivos a modificar: login_screen.dart, backend.py, GeminiAPI.py

B. Text-to-Speech (TTS)

Objetivo: Que la app lea el resumen generado para accesibilidad.

Solución Propuesta:

1. **Opción Cloud:** Usar la API Google Cloud TTS. El backend recibiría el texto, generaría un MP3 y devolvería una URL o Base64 al frontend.
2. **Opción Local (PREFERENTE):** Usar el paquete flutter_tts. Esto es más económico y rápido. El móvil recibe el texto JSON y lo lee usando el motor nativo del celular.

Los archivos a modificar: nfc_screen.dart

C. Mejoras en el Despliegue (Backend)

Para pasar a producción real:

1. **Servidor WSGI:** Actualmente se usa uvicorn directo. Para producción, se recomienda poner **Gunicorn** con Uvicorn workers para manejar concurrencia.
2. **HTTPS:** El backend corre en HTTP. Se debe configurar **Nginx** como Proxy Inverso y usar **Certbot** para certificados SSL gratuitos (Let's Encrypt).
3. **Cache:** Implementar Redis para guardar resúmenes comunes y no gastar tokens de Gemini repetidamente en la misma información.
