Informe Arquitectura programa sistema de EPS

Miguel Ángel Gutiérrez Jiménez
Juan Manuel Vásquez Nieto
Santiago Noriega

Universidad del Quindío

30 de octubre de 2025

Índice

1 Introducción y Arquitectura del Sistema

El proyecto analizado implementa un micro-sistema de información para una Entidad Promotora de Salud (EPS). A pesar de su aparente simplicidad, el sistema está construido sobre una robusta arquitectura de software multicapa que separa claramente las responsabilidades. Este enfoque es fundamental en la ingeniería de software moderna, ya que promueve la mantenibilidad, escalabilidad y la capacidad de prueba del código.

La arquitectura puede visualizarse como un modelo de 5 capas:

- **Capa de Simulación (Cliente):** Representada por `test_client.py` y `test_medical_client.py`. Estos archivos actúan como el usuario final, simulando flujos de trabajo completos.
- **Capa de Abstracción del Cliente (SDK):** Representada por `eps_client.py`. Este módulo actúa como un "control remoto" o SDK (Software Development Kit) que traduce funciones de Python simples en complejas peticiones de red (HTTP).
- **Capa de Servidor (API):** Representada por `eps_server.py`. Este es el "recepcionista" o el punto de entrada central. Escucha las peticiones de red, las valida y las dirige al módulo de lógica correcto.
- **Capa de Lógica de Negocio (El Cerebro):** Representada por `affiliates.py` y `clinical.py`. Aquí reside la inteligencia del sistema, las reglas de validación y la manipulación de datos.
- **Capa de Persistencia (Datos):** Representada por los archivos `.txt` y `.csv`. Actúan como la base de datos del sistema, almacenando el estado de la aplicación.

El flujo de una solicitud es, por lo tanto, secuencial: **Simulación** $\rightarrow$ **SDK** $\rightarrow$ **Servidor API** $\rightarrow$ **Lógica de Negocio** $\rightarrow$ **Persistencia**. En este informe, analizaremos cada uno de estos componentes en detalle.

2 Capa de Servidor: `eps_server.py`

Este archivo es el corazón de la arquitectura de comunicación. Su única responsabilidad es actuar como un servidor HTTP que recibe peticiones de la red y las traduce en llamadas a funciones de Python. Es importante notar que **este archivo no contiene lógica de negocio**; es un controlador de tráfico.

Utiliza las librerías `http.server` y `urllib.parse` de la biblioteca estándar de Python, lo cual lo hace ligero y sin dependencias externas.

Análisis de Código: `eps_server.py`

Importaciones y Configuración Inicial: El servidor importa los dos módulos de lógica de negocio. Esta es la conexión clave que le permite al servidor "delegar" el trabajo.

```
# Importa los módulos que contienen la lógica real
import affiliates         # funciones de afiliados + usuarios (roles/sesión)
import clinical           # citas + prescripciones (unificado)

# Configura el servidor para escuchar en todas las interfaces (0.0.0.0)
# en el puerto 8080.
HOST = "0.0.0.0"
PORT = 8080
```

Manejador de Peticiones (`EPSHandler`): La clase `EPSHandler` hereda de `BaseHTTPRequestHandler` y es donde se define cómo reaccionar a las peticiones. Sobrescribe los métodos `do_GET` (para peticiones de consulta) y `do_POST` (para peticiones de creación o modificación).

Manejo de Peticiones GET (`do_GET`): El método `do_GET` se activa para peticiones que solo solicitan datos. El código parsea la URL para identificar la ruta (`path`) y los parámetros de consulta (`qs`).

```
    def do_GET(self):
        # Parsea la URL para separar la ruta (ej: /list)
        # de los parámetros (ej: ?id=1010)
        url = urlparse(self.path)
        path = url.path
        q = url.query
        qs = parse_qs(q) if q else {} # 'qs' es un diccionario de parámetros

        # --- EJEMPLO DE RUTA GET: /search ---
        # Busca un afiliado por ID
        if path == "/search":
            try:
                # Obtiene el parámetro 'id' de la URL
                aff_id = qs.get("id", [""])[0]
                # Llama a la función de lógica en affiliates.py
                data = affiliates.searchById(aff_id)
                if data:
                    return _json_response(self, 200, data)
                else:
                    return _json_response(self, 404, {"error": "not found"})
            except Exception as e:
                # Manejo de errores
                return _json_response(self, 500, {"error": str(e)})

        # ... (otras rutas GET como /list, /stats, etc.) ...
```

Manejo de Peticiones POST (`do_POST`): El método `do_POST` maneja peticiones que envían datos para crear o actualizar recursos. El servidor primero debe leer y decodificar el "cuerpo" (`body`) de la petición.

```
    def do_POST(self):
        # Lee el cuerpo de la petición (los datos enviados por el cliente)
        b = _read_body(self)
        path = self.path

        # --- EJEMPLO DE RUTA POST: /user/register ---
        # Registra un nuevo usuario
        if path == "/user/register":
            try:
                # Llama a la función de lógica en affiliates.py
                # con los datos extraídos del cuerpo (b)
                msg = affiliates.registerUser(
                    str(b.get("name", "")).strip(),
                    str(b.get("password", "")).strip(),
                    str(b.get("role", "")).strip()
                )

                # Responde al cliente basado en el resultado de la lógica
                if msg == "ok":
                    return _json_response(self, 201, {"message": "registered"})
                if msg == "user exists":
                    return _json_response(self, 409, {"error": msg}) # 409:
    Conflicto
```

```
                return _json_response(self, 400, {"error": msg}) # 400: Petición incorrecta
            except Exception as e:
                return _json_response(self, 500, {"error": str(e)})

        # ... (otras rutas POST como /register, /user/session, etc.) ...
```

3 Capa de Lógica de Negocio (El Cerebro)

Esta capa es la más crítica desde la perspectiva del negocio, ya que implementa todas las reglas, validaciones y operaciones. Está dividida en dos módulos cohesivos.

3.1 Gestor de Identidad y Afiliados: `affiliates.py`

Este módulo tiene dos responsabilidades principales:

1. La gestión de **Afiliados** (CRUD, estadísticas) y **Encuestas**.
2. La gestión de **Usuarios del Sistema** (autenticación y roles), que es la base de la seguridad para el módulo clínico.

Análisis de Código: `affiliates.py`

Persistencia de Datos: El módulo define constantes para los nombres de archivo y usa funciones de apoyo (helpers) como `_read_affiliates` y `_load_users` para abstraer la lectura/escritura de los archivos CSV y JSONL.

```
AFF_FILE = "affiliates.csv"     # CSV para afiliados
SURV_FILE = "surveys.csv"       # CSV para encuestas
USERS_FILE = "users.txt"        # JSONL para usuarios

def _load_users():
    """Lee todos los usuarios desde el archivo JSONL."""
    _ensure_users() # Se asegura que el archivo exista
    lista = []
    with open(USERS_FILE, encoding="utf-8") as f:
        for linea in f:
            linea = linea.strip() # Quita saltos de línea
            if linea != "":
                # json.loads convierte el string de una línea a un diccionario
                usuario = json.loads(linea)
                lista.append(usuario)
    return lista

def _rewrite_users(lista):
    """Sobrescribe el archivo de usuarios con la nueva lista."""
    with open(USERS_FILE, "w", encoding="utf-8") as f:
        for u in lista:
            # json.dumps convierte el diccionario a un string
            linea = json.dumps(u)
            f.write(linea + "\n") # Escribe el string + salto de línea
```

Función Clave de Lógica (Gestión de Sesión): Esta función demuestra la lógica de negocio: recibe datos, lee el estado actual, aplica cambios y lo reescribe.

```
def openCloseSession(nombre, clave, bandera):
    """Abre o cierra sesión de un usuario."""
    # 1. Leer el estado actual de la "base de datos"
```

```
    usuarios = _load_users()

    # 2. Aplicar lógica de negocio y validación
    usuario = _find_user(usuarios, nombre)
    if usuario is None:
        return "invalid data" # Error: Usuario no existe
    if usuario["password"] != clave:
        return "wrong credentials" # Error: Contraseña incorrecta

    # 3. Modificar el estado en memoria
    usuario["session"] = bool(bandera) # Actualiza el estado de sesión

    # 4. Persistir (guardar) el nuevo estado en la "base de datos"
    _rewrite_users(usuarios)
    return "ok" # Operación exitosa
```

3.2 Gestor de Operaciones Clínicas: `clinical.py`

Este módulo gestiona la lógica de citas y prescripciones. Su característica más importante es su **dependencia de** `affiliates.py` para la autenticación y autorización.

Análisis de Código: `clinical.py`

Importación de Dependencia (Seguridad): La primera línea de código funcional es la más importante: importa `findUser` del otro módulo de lógica. Esto permite a `clinical.py` preguntar: "¿Quién es este usuario y qué permisos tiene?".

```
# Importamos la función findUser del otro archivo para validar usuarios.
from affiliates import findUser
```

Función Clave de Lógica (Agendar Cita): La función `scheduleAppointment` es un ejemplo perfecto de una "cadena de validación", donde cada paso debe ser exitoso antes de proceder al siguiente.

```
def scheduleAppointment(patient_name, patient_password, doctor_name, date_str,
    time_str):
    """Agenda una nueva cita médica aplicando todas las validaciones."""

    # 1. Validación del Doctor: ¿Existe? ¿Es un doctor?
    doctor = findUser(doctor_name)
    if not doctor or doctor.get("role") != "doctor":
        return "doctor not found"

    # 2. Validación del Paciente: ¿Existe? ¿Clave correcta? ¿Sesión iniciada?
    patient = findUser(patient_name)
    if not patient or patient.get("password") != patient_password or not patient.get
    ("session"):
        return "user not logged in"

    # 3. Validación de Reglas de Negocio: ¿Es un horario de atención válido?
    estado, _, _ = _valid_slot(date_str, time_str)
    if estado != "ok":
        return estado # Retorna "invalid data" o "out of range"

    # 4. Validación de Integridad (Colisión):
    citas = _load_jsonl(APPTS_FILE)
    for c in citas:
        if (c["doctor"] == doctor_name and c["date"] == date_str and
            c["time"] == time_str and c["status"] == "scheduled"):
```

```
            return "slot taken" # Error: Cita ya ocupada

    # 5. Ejecución: Si todo es válido, se crea y guarda la cita.
    nueva = {
        "id": _next_id(citas), "patient": patient_name, "doctor": doctor_name,
        "date": date_str, "time": time_str, "status": "scheduled"
    }
    citas.append(nueva)
    _rewrite_jsonl(APPTS_FILE, citas)
    return "ok"
```

4 Capa de Abstracción del Cliente: `eps_client.py`

Este archivo es un **SDK** (Software Development Kit) o ”control remoto”. Su propósito es ocultar la complejidad de la comunicación de red (HTTP, URLs, headers, codificación de datos) detrás de funciones de Python simples y limpias.

Cualquier programa que quiera ”hablar” con el servidor no debería construir peticiones HTTP manualmente; en su lugar, debería importar y usar este cliente.

Análisis de Código: `eps_client.py`

El archivo utiliza la librería `requests`, la cual es el estándar de facto en Python para realizar peticiones HTTP.

Función Cliente (Ejemplo POST): Cada función aquí mapea directamente a una ruta del servidor. Se encarga de empaquetar los datos y enviarlos.

```
import requests # Librería para hacer peticiones HTTP

# Define el tipo de contenido que se envía en peticiones POST
HEADERS = {"Content-Type": "application/x-www-form-urlencoded"}

def registerUser(url, name, password, role):
    """
    Esta función NO registra al usuario.
    Construye y ENVÍA una petición HTTP POST al servidor para
    que ÉL registre al usuario.
    """
    # 1. Define la URL completa del endpoint
    endpoint_url = url + "/user/register"

    # 2. Prepara el cuerpo (body) de la petición en el formato esperado
    # por el servidor (x-www-form-urlencoded)
    body = f"name={name}&password={password}&role={role}"

    # 3. Envía la petición al servidor
    response = requests.post(endpoint_url, data=body, headers=HEADERS)

    # 4. Devuelve la respuesta del servidor decodificada
    return response.content.decode("utf-8", errors="replace")
```

Función Cliente (Ejemplo GET): Las peticiones GET son más simples y a menudo pasan datos a través de la propia URL (query parameters).

```
def searchAffiliate(url, _id):
    """
```

```
    Envía una petición HTTP GET al servidor para buscar un afiliado.
    """
    # 1. Construye la URL completa, incluyendo el parámetro de consulta
    endpoint_url = url + f"/search?id={_id}"

    # 2. Envía la petición GET
    response = requests.get(endpoint_url)

    # 3. Devuelve la respuesta
    return response.content.decode("utf-8", errors="replace")
```

5 Capa de Simulación: `test_client.py` y `test_medical_client.py`

Estos archivos son los "usuarios finales" del sistema. Son scripts ejecutables que importan la librería cliente (`eps_client.py`) y la utilizan para simular un flujo de trabajo completo. Su propósito es actuar como un conjunto de pruebas de integración automáticas, verificando que todas las capas del sistema (Cliente → Servidor → Lógica → Datos) funcionan correctamente juntas.

Análisis de Código: `test_medical_client.py`

Este script simula un flujo clínico completo, desde el registro hasta la prescripción.

```
# Importamos nuestro "control remoto" y le ponemos el alias 'cli'
import eps_client as cli
import os

URL = "http://localhost:8080" # Dirección del servidor a probar

# --- Limpieza Opcional ---
# Borra los archivos de datos para asegurar una prueba limpia
# donde la primera cita siempre tendrá el ID "1".
for f in ("users.txt", "appointments.txt", "prescriptions.txt"):
    try:
        if os.path.exists(f):
            os.remove(f)
    except Exception as e:
        print(f"No se pudo borrar {f}:", e)

# --- INICIO DE LA SECUENCIA DE PRUEBA ---

print("== REGISTRO DE USUARIOS ==")
# 1. Crea los actores necesarios: un doctor y un paciente.
print(cli.registerUser(URL, "dr_lina", "abc", "doctor"))
print(cli.registerUser(URL, "juan", "222", "patient"))

print("\n== ABRIR SESIÓN ==")
# 2. Ambos actores inician sesión (prerrequisito para la cita).
print(cli.openSession(URL, "juan", "222"))
print(cli.openSession(URL, "dr_lina", "abc"))

print("\n== AGENDAR CITA ==")
# 3. El paciente (juan) agenda la cita con el doctor (dr_lina).
print(cli.scheduleAppointment(URL, patient="juan", password="222",
                              doctor="dr_lina", date_ddmmyyyy="19/09/2025",
                              time_hhmm="09:00"))

print("\n== LISTAR CITAS DEL PACIENTE ==")
# 4. El paciente verifica que la cita fue creada.
print(cli.listAppointments(URL, name="juan", password="222"))
```

```
38
39 print("\n== CREAR PRESCRIPCIÓN (usa appt_id=1 si es la primera cita) ==")
40 # 5. El doctor crea una prescripción asociada a la cita (ID "1").
41 print(cli.createPrescription(URL, doctor="dr_lina", password="abc",
42                              patient="juan", appt_id="1",
43                              text="Ibuprofeno 400 mg"))
44
45 print("\n== LISTAR PRESCRIPCIONES COMO PACIENTE ==")
46 # 6. El paciente verifica que puede ver su prescripción.
47 print(cli.listPrescriptions(URL, "patient", "juan", "222"))
48
49 # ... (y así sucesivamente)
```

6 Guía de Ejecución y Puesta en Marcha

Para que el sistema funcione, es fundamental seguir el orden correcto, ya que el servidor debe estar en ejecución antes de que cualquier cliente pueda conectarse a él.

Paso 1: Requisitos Previos

Asegúrese de tener Python instalado en su sistema. El único paquete externo requerido por este proyecto es `requests`, necesario para el cliente.

```
pip install requests
```

Todos los archivos del proyecto deben estar ubicados en la misma carpeta.

Paso 2: Iniciar el Servidor (Terminal 1)

El servidor es el programa que debe iniciarse primero y dejarse en ejecución.

1. Abra una terminal o símbolo del sistema.
2. Navegue hasta la carpeta donde se encuentran los archivos del proyecto. (Ej: `cd G:/DATOS/Escritorio/programacion_t`).
3. Ejecute el script del servidor con Python:

   ```
   python eps_server.py
   ```

4. Si tiene éxito, verá el mensaje: `EPS server running on http://0.0.0.0:8080`.
5. **No cierre esta terminal.** Debe permanecer activa para recibir peticiones.

Paso 3: Ejecutar los Clientes (Terminal 2)

Para interactuar con el servidor, debe abrir una **segunda terminal**.

1. Abra una nueva ventana de terminal.
2. Navegue hasta la misma carpeta del proyecto (Ej: `cd G:/DATOS/Escritorio/programacion_t`).
3. Ejecute los scripts de prueba, uno a la vez:

   ```
   # Prueba la lógica de afiliados y usuarios
   python test_client.py
   ```

```
# Cuando termine, pruebe la lógica clínica
python test_medical_client.py
```

Al ejecutar estos scripts, la Terminal 2 (Cliente) mostrará las respuestas del servidor (ej. `{"message": "registered"}`), mientras que la Terminal 1 (Servidor) mostrará un registro de las peticiones que ha recibido (ej. `POST /user/register ... 201 -`).

7 Conclusión

El proyecto implementa exitosamente un sistema de información de EPS mediante una arquitectura de microservicios basada en HTTP. La clara separación de responsabilidades entre el servidor (`eps_server.py`), la lógica de negocio (`affiliates.py`, `clinical.py`) y el cliente (`eps_client.py`) no solo demuestra un diseño de software robusto, sino que también facilita enormemente las pruebas, el mantenimiento y la futura expansión del sistema.

La lógica de negocio maneja de forma adecuada la persistencia de datos en archivos planos y aplica una cadena de validaciones rigurosas, destacando la función `scheduleAppointment` como un excelente ejemplo de implementación de reglas de negocio complejas, incluyendo la autenticación entre módulos para garantizar la seguridad e integridad de las operaciones clínicas.

8 Referencias Bibliográficas

Referencias

[1] Python Software Foundation. (2025). *The Python Standard Library (Módulo http.server)*. Python 3.12.x Documentation. Recuperado de `https://docs.python.org/3/library/http.server.html`

[2] Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks (5th ed.)*. Prentice Hall. (Capítulos sobre la capa de aplicación y la arquitectura cliente-servidor).