Descripción Detallada del Servidor TinyFaaS HTTP (v2.0)

Este documento explica la arquitectura, el flujo operativo y la estructura interna del código del servidor **TinyFaaS HTTP** (server_tinyfaas_persistent_http_v2.py), que utiliza el *framework* **Flask** para manejar peticiones web.

---

1. 🏗️ Diagrama Conceptual y Flujo Operativo

El servidor TinyFaaS HTTP opera como un **servicio web persistente** basado en Flask. En lugar de suscribirse a un *broker*, el servidor **expone rutas HTTP** que los clientes invocan directamente. La **Autenticación Básica HTTP** protege todas las rutas, siendo el primer punto de control en cada petición.

El aislamiento y la ejecución de funciones son **síncronos** en el contexto de la petición HTTP, lo que significa que la respuesta al cliente solo se envía una vez que la ejecución de la función ha finalizado (con éxito o error).

Diagrama de Bloques (Flujo de Funcionamiento)

Bloque	Descripción
Clientes (Usuario/Admin)	Envían peticiones HTTP (POST, GET, DELETE) con el *header* de autenticación a *endpoints* específicos.
Servidor Flask	Componente principal que escucha en un puerto TCP (ej. 8080) y gestiona el enrutamiento de peticiones a los *endpoints* definidos.
Capa de Autenticación	El decorador **@requires_auth** verifica las credenciales HTTP Basic Auth (admin:1234) en **todas** las peticiones antes de permitir el acceso a la lógica de negocio.

Manejador de Administración	Procesa rutas bajo /admin/* (Upload, List, Delete, Status). Las operaciones suelen ser síncronas y modifican el estado de las funciones.
Capa de Ejecución (FaaS)	Maneja la ruta /function/*. Carga dinámicamente el código (importlib.util), ejecuta la función main y captura el tiempo y el resultado/error. **Es síncrono** con la petición HTTP.
Capa de Persistencia	Utiliza los directorios functions y data para almacenar el código de las funciones (con venv) y los archivos de estado (functions.json, logs.json).
Respuestas HTTP	El servidor devuelve objetos **JSON** con el resultado o un código de estado apropiado (200 OK, 404 Not Found, 401 Unauthorized).

---

2. 🐍 Estructura del Código y Componentes Principales

El archivo server_tinyfaas_persistent_http_v2.py está dividido en bloques que reflejan la arquitectura web y la lógica FaaS.

A. 🔒 CONFIGURACIÓN DE SEGURIDAD (HTTP Basic Auth)

Este bloque define el mecanismo de protección para el servidor, central en la versión HTTP.

Componente	Descripción Detallada
USERNAME, **PASSWORD**	Constantes globales que almacenan las credenciales de administración.

check_auth(username, password)	Función que implementa la lógica de verificación de credenciales contra las constantes globales.
authenticate()	Genera y retorna una respuesta HTTP 401 Unauthorized que incluye el *header* WWW-Authenticate: Basic realm="Login Required", indicando al cliente cómo debe autenticarse.
requires_auth(f)	**Decorador de seguridad.** Se aplica a todas las funciones de vista (endpoints). Su lógica verifica si existe el *header* Authorization y si las credenciales son válidas antes de ejecutar la función de ruta original.

B. ⚡ CONFIGURACIÓN Y PERSISTENCIA

Similar a la versión MQTT, gestiona el estado del sistema y la creación de entornos aislados.

Función	Descripción Detallada
FUNCTIONS_DIR, DATA_DIR, etc.	Definición de rutas y directorios donde se guardan los archivos de estado (.json) y el código/entorno virtual de las funciones.

save_state() / load_state()	Funciones que guardan o cargan los diccionarios globales **functions** y **logs** a/desde archivos JSON en el disco, asegurando la persistencia.
create_venv(func_name, requirements)	Crea el **entorno virtual** dedicado para la función usando **subprocess.check_call** para ejecutar python -m venv y la posterior instalación de dependencias con pip (si existe requirements.txt).

C. 📂 ENDPOINTS DE ADMINISTRACIÓN

Funciones de vista de Flask protegidas por @requires_auth que gestionan el ciclo de vida de las funciones.

Endpoint (Función de Vista)	Método	Función Principal y Módulos Utilizados
/admin/upload (upload_function)	POST	Gestiona el formulario multipart/form-data. Guarda el código (func.py), guarda requirements.txt (si existe) y llama a create_venv para aislar el entorno.
/admin/functions (list_functions)	GET	Retorna una lista simple de los nombres de las funciones cargadas, obtenida del diccionario functions.

/admin/functions/<func_name> (delete_function)	DELETE	Elimina la función de los diccionarios globales, llama a **shutil.rmtree** para borrar físicamente el directorio de la función (código y venv), y guarda el estado.
/admin/logs/<func_name> (get_logs)	GET	Retorna el historial completo de ejecución de la función solicitada, consultando el diccionario global logs.
/admin/status (get_server_status)	GET	**Monitorización.** Utiliza la librería **psutil** para obtener métricas de consumo de CPU (en Milicpu) y memoria (en MB) del proceso actual y del sistema operativo.

D. ⚡ EJECUCIÓN DE FUNCIONES

El *endpoint* principal para la ejecución de código por parte de los usuarios.

Endpoint (Función de Vista)	Método	Función Principal y Módulos Utilizados

/function/<func_name> (execute_function)	POST	1. **Carga Dinámica:** Utiliza **importlib.util** para cargar el archivo func.py del disco como un módulo. 2. **Medición:** Marca los tiempos de inicio y fin con time.time(). 3. **Ejecución:** Invoca **module.main(*args)** pasando los argumentos del cuerpo JSON. 4. **Registro:** Captura el resultado (o la excepción), genera un objeto de log con **uuid.uuid4()**, actualiza el diccionario global logs y persiste el estado.

E. 🚀 MAIN

El bloque de inicio del servidor.

1. **load_state()**: Se ejecuta para cargar el estado del sistema desde los archivos JSON antes de que el servidor comience a aceptar conexiones.
2. **app.run(...)**: Inicia el servidor web Flask, haciéndolo accesible en la red (host="0.0.0.0", port=8080).