Conversación con Gemini

Este enlace contiene información y más enlaces para usar fastMCP para la creación de servidores MCP, de model Context protocol:

https://developers.googleblog.com/en/gemini-cli-fastmcp-simplifying-mcp-server-development/

Me gustaría un estudio en profundidad, eminentemente práctico para usar fastMCP siguiendo esas indicaciones para configurar servidores MCP.

Este es el plan que he confeccionado. Dime si necesitas que haga algún cambio antes de que empiece con la investigación.

Guía Práctica fastMCP para Servidores
Investigar sitios web
(1) Analizar el artículo proporcionado para extraer la definición y el propósito del Protocolo de Contexto de Modelo (MCP) y la herramienta fastMCP. Sintetizar los problemas que fastMCP resuelve para los desarrolladores. (2) Identificar y detallar los requisitos previos (versiones de Python, dependencias, etc.) y el proceso de instalación de fastMCP, basándose en la información del blog y cualquier repositorio de código fuente enlazado. (3) Desglosar el proceso paso a paso para crear un servidor MCP básico utilizando fastMCP, tal como se describe en el artículo. Esto debe incluir la estructura de directorios, la definición de un modelo simple y los comandos para iniciar el servidor. (4) Investigar y explicar en detalle la sintaxis y la estructura necesarias para definir modelos personalizados dentro de fastMCP. Buscar ejemplos prácticos en el artículo o en la documentación enlazada sobre cómo configurar los parámetros y las capacidades de un modelo. (5) Detallar el procedimiento para definir e integrar herramientas (tools) que los modelos puedan utilizar. Explicar cómo se declaran estas herramientas en la configuración de fastMCP y cómo se establece la comunicación entre el modelo y la herramienta. (6) Describir los métodos para interactuar y probar el servidor MCP una vez configurado y en funcionamiento. Esto incluye buscar ejemplos de peticiones (por ejemplo, usando cURL o un cliente específico) para invocar los modelos y herramientas. (7) Sintetizar toda la información en una guía práctica consolidada. Identificar y explicar cualquier configuración avanzada, buenas prácticas de desarrollo o consejos para la solución de problemas que se mencionen en el artículo o en la documentación de soporte.
Más
Analizar resultados
Crear informe
Estará lista en unos minutos
Modificar planEmpezar la investigación

Empezar la investigación

He completado la investigación. No dudes en hacerme más preguntas o pedirme cambios.

Guía Práctica fastMCP para ServidoresCompletado22 sept, 13:41

Deep Research
Vídeo

Guía Práctica fastMCP para Servidores

ÍndiceExportar
Crear

Guía Práctica y Exhaustiva para el Desarrollo de Servidores MCP con fastMCP: De la Concepción a la Producción

Sección 1: Fundamentos del Protocolo de Contexto de Modelo (MCP) y el Ecosistema fastMCP

Antes de construir servidores, es imperativo comprender los cimientos sobre los que se edifican. Esta sección establece el marco conceptual del Protocolo de Contexto de Modelo (MCP) y su implementación de referencia en Python, `fastMCP`. El dominio de estos principios es crucial no solo para seguir instrucciones, sino para diseñar soluciones robustas, escalables y alineadas con la visión de un ecosistema de IA interconectado.

1.1. Desmitificando MCP: El "USB-C para la IA"

En el panorama actual de la inteligencia artificial, los Modelos de Lenguaje Grandes (LLMs) poseen capacidades de razonamiento sin precedentes, pero operan en un vacío. Carecen de acceso directo al mundo exterior: no pueden leer archivos locales, consultar bases de datos en tiempo real o interactuar con APIs de terceros de forma nativa. Históricamente, cada aplicación de LLM (como un asistente de chat o un entorno de desarrollo integrado) que necesitaba estas capacidades debía implementar conectores personalizados para cada herramienta, un proceso costoso, frágil y que no escala. Este es el problema fundamental que el Protocolo de Contexto de Modelo (MCP) está diseñado para resolver.

MCP es un protocolo abierto y estandarizado que define una forma común para que las aplicaciones de LLM (clientes) se comuniquen de manera segura y predecible con herramientas y fuentes de datos externas (servidores). La analogía más poderosa para describir MCP es la de un "puerto USB-C para la IA". Así como el USB-C unificó un ecosistema fragmentado de cables y puertos para datos, energía y vídeo, MCP busca proporcionar un único "puerto" universal. Esto permite que cualquier herramienta o servicio que implemente un servidor MCP pueda "enchufarse" a cualquier aplicación de LLM que actúe como un cliente MCP, fomentando la interoperabilidad y la reutilización.

Esta estandarización de la capa de conexión es una decisión estratégica con profundas implicaciones. Al mercantilizar la interoperabilidad, MCP desplaza el foco del valor. En lugar de que los desarrolladores inviertan tiempo en construir y mantener conectores personalizados, pueden concentrarse en crear herramientas más potentes e inteligentes. Esto fomenta un ecosistema vibrante donde la innovación ocurre en las herramientas mismas, no en la plomería que las conecta. La existencia de un registro de servidores MCP y SDKs oficiales en múltiples lenguajes de programación (Python, TypeScript, Java, C#, Rust, etc.), respaldados por un consorcio que incluye a actores clave como Anthropic y Microsoft, subraya su ambición de convertirse en un estándar industrial duradero.

Un servidor MCP estructura su funcionalidad en tres componentes principales:

- **Herramientas (Tools):** Representan acciones o funciones que un LLM puede solicitar al servidor que ejecute. Son análogas a los puntos finales `POST` o `PUT` en una API REST tradicional, ya que están diseñadas para producir un efecto secundario, como enviar un correo electrónico, escribir en una base de datos o realizar un cálculo complejo.
- **Recursos (Resources):** Exponen datos de solo lectura que un LLM puede consultar para enriquecer su contexto. Son similares a los puntos finales `GET` de una API, proporcionando información como perfiles de usuario, archivos de configuración o flujos de datos en tiempo real.
- **Prompts:** Son plantillas reutilizables y parametrizadas que definen patrones de interacción específicos. Ayudan a guiar al LLM para que realice tareas complejas de manera consistente y estructurada.

1.2. Presentación de fastMCP: El Framework Pythonic para la Producción

Si MCP es la especificación del protocolo, `fastMCP` es el motor de alto rendimiento que lo implementa en Python. `fastMCP` se ha consolidado como el framework estándar y de facto para construir aplicaciones MCP en este lenguaje. Es un proyecto de código abierto, activamente mantenido como

`fastMCP 2.0`, que extiende significativamente las capacidades del SDK base del protocolo.

La propuesta de valor central de `fastMCP` es la abstracción radical de la complejidad. Implementar el protocolo MCP desde cero requiere un manejo manual de los detalles de bajo nivel, la gestión de conexiones, la serialización de datos y el manejo de errores. `fastMCP` encapsula toda esta complejidad, permitiendo a los desarrolladores centrarse exclusivamente en la lógica de negocio de sus herramientas y recursos. Esta simplicidad se manifiesta en su API basada en decoradores, donde a menudo basta con añadir

`@mcp.tool` a una función de Python estándar para exponerla de forma segura a un LLM.

Sin embargo, `fastMCP` es mucho más que un simple envoltorio sintáctico. Es un ecosistema completo diseñado para llevar las aplicaciones MCP desde la idea hasta la producción. Sus características de nivel empresarial incluyen :

- **Autenticación Integrada:** Soporte nativo para proveedores de OAuth como Google, GitHub, Microsoft Azure, Auth0 y más, permitiendo la creación de servidores seguros con una configuración mínima.
- **Herramientas de Despliegue:** Un robusto CLI y patrones de configuración que facilitan el empaquetado y despliegue de servidores en infraestructuras locales, en la nube o en plataformas gestionadas como FastMCP Cloud.
- **Framework de Pruebas:** Un cliente programático y patrones de prueba diseñados para facilitar la escritura de pruebas unitarias y de integración rápidas y fiables.
- **Patrones de Arquitectura Avanzados:** Capacidades para la composición de servidores (estilo microservicios), la creación de proxies y la generación automática de servidores a partir de especificaciones OpenAPI o aplicaciones FastAPI existentes.

La afiliación del proyecto con Prefect, una empresa reconocida por sus herramientas de orquestación de flujos de trabajo de datos, no es una coincidencia. Esta conexión infunde en

`fastMCP` una filosofía de "preparado para producción" desde su concepción. Características como la fiabilidad, la observabilidad y la facilidad de despliegue no son ocurrencias tardías, sino principios de diseño fundamentales. Para un desarrollador, esto proporciona la confianza de que `fastMCP` no es una herramienta experimental, sino una base sólida sobre la cual construir aplicaciones críticas para el negocio.

1.3. Análisis Comparativo: ¿Por qué elegir fastMCP?

Para cuantificar las ventajas de `fastMCP`, es útil compararlo directamente con enfoques alternativos, como el uso del SDK de MCP de bajo nivel en Python o el SDK de TypeScript. Los datos revelan una diferencia drástica en la eficiencia del desarrollo y la preparación para la producción.

Tabla 1: Comparativa de Frameworks para Desarrollo MCP

Característica	FastMCP	Raw MCP SDK (Python)	TypeScript MCP
Complejidad de Configuración	Mínima (decoradores)	Alta (protocolo manual)	Media (definiciones de tipo)
Tiempo de Desarrollo	1-2 horas	8-12 horas	4-6 horas
Depuración Integrada	✅ MCP Inspector	❌ Pruebas manuales	✅ Herramientas básicas
Manejo de Errores	✅ Envoltura automática	❌ Implementación manual	✅ Seguridad de TypeScript
Curva de Aprendizaje	Baja	Alta	Media
Preparado para Producción	✅ Sí	⚠️ Requiere experiencia	✅ Sí

Exportar a Hojas de cálculo

El análisis de esta tabla es revelador. `fastMCP` puede acelerar el tiempo de desarrollo entre 5 y 8 veces en comparación con el uso del SDK nativo. Esto no es solo una mejora incremental; es una transformación del flujo de trabajo que reduce drásticamente el tiempo de comercialización y los costos de desarrollo. La abstracción de la configuración, el manejo automático de errores y las herramientas de depuración integradas como el MCP Inspector eliminan clases enteras de problemas comunes, permitiendo a los equipos centrarse en la innovación en lugar de en la infraestructura.

Sección 2: Configuración del Entorno y Creación del Primer Servidor

Esta sección es eminentemente práctica, diseñada para guiar al desarrollador desde un entorno vacío hasta un servidor MCP funcional en el menor tiempo posible. Este proceso práctico genera confianza y proporciona una comprensión tangible de la mecánica fundamental de `fastMCP`.

2.1. Requisitos Previos y Configuración del Entorno

Para comenzar a desarrollar con `fastMCP`, se necesita un entorno de Python moderno y un gestor de paquetes eficiente.

- **Requisitos:**
    - Python 3.10 o superior.

- `uv`, un instalador y resolutor de paquetes de Python extremadamente rápido.

Aunque `pip` puede utilizarse, `uv` es la herramienta recomendada por el ecosistema `fastMCP`. Su velocidad superior y su integración profunda con el CLI de `fastMCP` para comandos de instalación y ejecución lo convierten en la opción preferida para un desarrollo fluido y despliegues reproducibles.

A continuación se presenta una guía paso a paso para configurar un nuevo proyecto:

Crear el directorio del proyecto: Abra una terminal y ejecute los siguientes comandos para crear una carpeta para el proyecto y navegar dentro de ella.
Bash
mkdir mi-servidor-mcp
cd mi-servidor-mcp

1.
2. **Inicializar un proyecto `uv`:** Este comando crea un archivo `pyproject.toml`, que servirá como el manifiesto de su proyecto, definiendo sus metadatos y dependencias.

Bash
uv init

3.
4. **Crear y activar un entorno virtual:** Es una práctica recomendada aislar las dependencias de cada proyecto. `uv` simplifica enormemente este proceso.

Bash
uv venv
source.venv/bin/activate

5. En Windows, el comando de activación es `.venv\Scripts\activate`.
6. **Instalar `fastMCP`:** Con el entorno virtual activado, instale `fastMCP`. Se recomienda instalar el extra `[cli]`, que incluye herramientas esenciales como el MCP Inspector.

Bash
uv pip install "mcp[cli]"

7. Alternativamente, para proyectos que solo necesitan el framework sin las herramientas de línea de comandos, se puede usar `uv pip install "fastmcp"`.

Con estos pasos, el entorno de desarrollo está listo para crear el primer servidor.

2.2. Tutorial Práctico 1: Un Servidor Calculadora Básico

El primer servidor será un ejemplo canónico: una simple calculadora. Este ejercicio demuestra la simplicidad y elegancia del enfoque de `fastMCP`.

Cree un nuevo archivo llamado `servidor.py` en el directorio del proyecto y añada el siguiente código :

Python

```
# servidor.py
from fastmcp import FastMCP

# 1. Instanciar el servidor con un nombre descriptivo para su identificación.
#    Este nombre puede aparecer en los registros o en las interfaces de cliente.
mcp = FastMCP("ServidorCalculadora 🚀")

# 2. Definir una herramienta usando el decorador @mcp.tool.
#    fastMCP se encarga de todo el protocolo subyacente.
@mcp.tool
def add(a: int, b: int) -> int:
    """Suma dos números enteros y devuelve el resultado.

    Esta descripción (docstring) es crucial. El LLM la utilizará para
    entender qué hace esta herramienta y cuándo debe usarla.
    Las anotaciones de tipo (a: int, b: int) son igualmente importantes,
    ya que fastMCP las usa para generar un esquema y validar las entradas.
    """
    print(f"Ejecutando add({a}, {b})")
    return a + b

# 3. (Opcional pero recomendado para la ejecución directa)
#    Este bloque permite ejecutar el servidor directamente con `python servidor.py`.
if __name__ == "__main__":
    print("Iniciando servidor FastMCP en modo stdio por defecto...")
    mcp.run()
```

El análisis de este código revela la filosofía de `fastMCP`:

- **`mcp = FastMCP(...)`:** Esta línea crea la instancia central de la aplicación. Es el objeto que contendrá todas las herramientas, recursos y prompts.

- **`@mcp.tool`:** Este decorador es el corazón de `fastMCP`. Transforma una función Python normal en una herramienta MCP, manejando la generación de esquemas, la validación de entradas y la comunicación con el cliente.
- **Docstring y Anotaciones de Tipo:** `fastMCP` aprovecha las características modernas de Python para un desarrollo declarativo. El docstring se convierte en la descripción de la herramienta para el LLM, y las anotaciones de tipo se utilizan para crear un contrato de datos formal, asegurando que el LLM proporcione los argumentos correctos.
- **`if __name__ == "__main__":`:** Este es un patrón estándar de Python que permite que el archivo sea tanto un script ejecutable como un módulo importable. La llamada a `mcp.run()` inicia el servidor y comienza a escuchar las conexiones.

2.3. Ejecución del Servidor y Protocolos de Transporte

Existen dos métodos principales para ejecutar un servidor `fastMCP`, cada uno con sus propias ventajas y casos de uso. Esta dualidad refleja una filosofía de diseño que atiende tanto al prototipado rápido como a la configuración robusta para producción.

Método 1: Ejecución Directa de Python Este es el enfoque más simple e intuitivo, familiar para cualquier desarrollador de Python.
Bash
python servidor.py

- Por defecto, `mcp.run()` utiliza el protocolo de transporte `stdio`. En este modo, la comunicación entre el cliente (por ejemplo, un editor de código como Cursor o Claude Desktop) y el servidor se realiza a través de los flujos de entrada y salida estándar (
- `stdin` y `stdout`). El servidor se ejecuta como un subproceso del cliente y su ciclo de vida está ligado al de la sesión del cliente. Es ideal para herramientas locales y desarrollo rápido.

Método 2: El CLI de `fastmcp` Este método ofrece mayor flexibilidad y control, especialmente para configurar el transporte y otros parámetros sin modificar el código.
Bash
fastmcp run servidor.py:mcp

- El argumento `servidor.py:mcp` le dice al CLI que busque el objeto llamado `mcp` dentro del archivo `servidor.py`. Una ventaja clave de este enfoque es que no requiere el bloque `if __name__ == "__main__"`, ya que el CLI importa y ejecuta el objeto del servidor directamente. Esto promueve una separación más limpia entre la definición del servidor y su configuración de ejecución, un principio fundamental para el despliegue automatizado en entornos de producción.

Explorando los Protocolos de Transporte

El transporte es el "idioma" que el servidor utiliza para comunicarse con los clientes. `fastMCP` soporta varios protocolos, cada uno adaptado a diferentes escenarios.

Tabla 2: Protocolos de Transporte en fastMCP

Transporte	Mecanismo	Caso de Uso Principal	Ventajas	Desventajas
`stdio`	Entrada/Salida Estándar	Herramientas locales, desarrollo	Simple, sin red, seguro	Un solo cliente, ciclo de vida ligado
`http`	HTTP/1.1	Servidores remotos, APIs web	Múltiples clientes, accesible por red	Mayor sobrecarga, configuración de red
`sse`	Server-Sent Events (sobre HTTP)	Streaming de servidor a cliente	Eficiente para notificaciones	Menos eficiente para comunicación bidireccional
`In-Memory`	Llamadas a funciones directas	Pruebas unitarias y de integración	Extremadamente rápido, sin E/S	No para uso en producción

Exportar a Hojas de cálculo

Para ejecutar el servidor calculadora con el transporte `http`, permitiendo que sea accesible a través de la red, se usaría el CLI de `fastmcp`:

Bash
```
fastmcp run servidor.py --transport http --port 8080
```

Este comando inicia un servidor web que escucha en el puerto 8080. Ahora, cualquier cliente MCP que pueda alcanzar esta dirección de red puede conectarse e interactuar con la herramienta `add`. Esta flexibilidad es la base para el despliegue de servidores MCP como servicios remotos y escalables.

Sección 3: Anatomía de un Servidor fastMCP: Componentes Esenciales

Una vez establecido el flujo de trabajo básico, es hora de profundizar en los tres componentes fundamentales que dan vida a un servidor `fastMCP`: Herramientas, Recursos y Prompts. Estos

elementos no son una colección arbitraria de características; representan un modelo conceptual coherente para la interacción entre un agente de IA y el mundo exterior. Mapean directamente las tres capacidades que un agente necesita: actuar (Herramientas), percibir (Recursos) y ser guiado (Prompts). Comprender esta separación de preocupaciones es clave para diseñar servidores lógicos, eficientes y fáciles de mantener.

3.1. Herramientas (Tools): El Corazón Funcional del Servidor

Las herramientas son el componente más activo de un servidor MCP. Son funciones Python que el LLM puede invocar para realizar acciones concretas, ejecutar cálculos o producir efectos secundarios en sistemas externos. Son el mecanismo a través del cual un LLM trasciende su naturaleza de procesador de texto y se convierte en un agente capaz de operar en el mundo digital.

Sintaxis y Mejores Prácticas:

- **Decorador `@mcp.tool()`:** Como se vio en el ejemplo anterior, este decorador registra una función como una herramienta disponible. El nombre de la herramienta se infiere directamente del nombre de la función, por lo que es importante usar nombres descriptivos y claros (por ejemplo, `enviar_correo_electronico` en lugar de `func1`).
- **Docstrings Descriptivos:** El docstring de la función no es mera documentación para humanos; es la principal fuente de información que el LLM utiliza para su proceso de razonamiento. El LLM analiza esta descripción para decidir si la herramienta es relevante para la consulta del usuario, qué argumentos necesita y cómo interpretar el resultado. Un docstring bien redactado es la diferencia entre una herramienta que se usa correctamente y una que se ignora o se usa mal. Debe ser preciso, completo y no ambiguo.
- **Anotaciones de Tipo (`Type Hints`):** `fastMCP` utiliza las anotaciones de tipo de Python para generar automáticamente un esquema JSON que describe la firma de la herramienta. Este esquema se envía al cliente MCP durante la fase de descubrimiento. Cuando el LLM decide usar la herramienta, el cliente utiliza este esquema para estructurar los argumentos correctamente. Además, el servidor `fastMCP` usa el esquema para validar las solicitudes entrantes, proporcionando una capa robusta de seguridad de tipos.
- **Tipos de Retorno Soportados:** Las herramientas pueden devolver una variedad de tipos de datos, y `fastMCP` los maneja de forma inteligente :
    - `str`: Se envía como contenido de texto simple.
    - `dict`, `list`, modelos Pydantic: Se serializan a una cadena JSON y se envían como contenido de texto.
    - `bytes`: Se codifican en Base64 y se envían como un blob binario.

- `fastmcp.Image`: Una clase auxiliar para devolver datos de imagen de manera estandarizada.
- `None`: Indica que la herramienta se ejecutó con éxito pero no tiene un resultado que devolver.

A continuación, un ejemplo de una herramienta más compleja que interactúa con una API externa:

Python
```
import httpx
from fastmcp import FastMCP

mcp = FastMCP("ServidorDeUtilidadesWeb")

@mcp.tool
def obtener_clima(ciudad: str) -> str:
    """
    Obtiene el pronóstico del tiempo actual para una ciudad específica
    utilizando una API pública.
    """
    try:
        response = httpx.get(f"https://wttr.in/{ciudad}?format=j1")
        response.raise_for_status()  # Lanza una excepción para códigos de error HTTP
        data = response.json()
        condicion_actual = data['current_condition']
        return (f"El tiempo en {data['nearest_area'][0]['areaName'][0]['value']}: "
                f"{condicion_actual['value']}, "
                f"Temperatura: {condicion_actual['temp_C']}°C.")
    except httpx.HTTPStatusError as e:
        return f"Error al obtener el clima: No se pudo encontrar la ciudad '{ciudad}'."
    except Exception as e:
        return f"Error inesperado: {e}"
```

3.2. Recursos (Resources): Proporcionando Contexto al LLM

Mientras que las herramientas son para *hacer*, los recursos son para *saber*. Proporcionan al LLM acceso de solo lectura a fuentes de datos, permitiéndole obtener el contexto necesario para responder preguntas o tomar decisiones informadas. Son análogos a los puntos finales `GET` en una API REST.

Recursos Estáticos:

Un recurso estático tiene una URI (Identificador Uniforme de Recurso) fija y siempre devuelve el mismo contenido. Se definen con el decorador `@mcp.resource`.

Python
```
@mcp.resource("config://sistema/version")
def obtener_version_sistema() -> str:
    """Devuelve la versión actual del sistema."""
    return "1.2.3"
```

Un cliente podría leer este recurso para saber con qué versión del servidor está interactuando.

Recursos Dinámicos (Plantillas):

El verdadero poder de los recursos reside en las plantillas. Al incluir marcadores de posición con llaves `{}` en la URI, se pueden crear recursos dinámicos que aceptan parámetros. Esto permite al LLM solicitar subconjuntos específicos de datos.

Python
```
# Un "almacén de datos" simulado en memoria
DB_USUARIOS = {
    "101": {"nombre": "Alice", "rol": "admin"},
    "102": {"nombre": "Bob", "rol": "user"},
}

@mcp.resource("db://usuarios/{user_id}/perfil")
def obtener_perfil_usuario(user_id: str) -> dict:
    """
    Obtiene el perfil de un usuario específico a partir de su ID.
    Devuelve un diccionario con la información del usuario o un error si no se encuentra.
    """
    return DB_USUARIOS.get(user_id, {"error": f"Usuario con ID {user_id} no encontrado."})
```

En este ejemplo, un LLM podría razonar que para obtener información sobre el usuario "101", necesita solicitar la URI `db://usuarios/101/perfil`. `fastMCP` se encarga de analizar la URI, extraer el parámetro `user_id` y pasarlo a la función `obtener_perfil_usuario`.

3.3. Prompts: Patrones de Interacción Reutilizables

Los prompts son el componente de guía. Definen plantillas de mensajes reutilizables que pueden ser invocadas para iniciar flujos de trabajo específicos o para asegurar que el LLM responda de una manera particular y consistente. Son especialmente útiles para tareas complejas que requieren una estructura de conversación específica.

Los prompts se definen con el decorador `@mcp.prompt()`. La función decorada puede devolver una cadena de texto simple o, para interacciones más sofisticadas, una lista de objetos `Message` (como `UserMessage` y `AssistantMessage`) que simulan un historial de conversación.

Python
```
from fastmcp.prompts.base import UserMessage, AssistantMessage

@mcp.prompt()
def solicitar_revision_codigo(fragmento_codigo: str) -> str:
    """Genera una solicitud estándar para la revisión de un fragmento de código."""
    return (f"Por favor, revisa el siguiente fragmento de código en busca de "
            f"posibles errores, problemas de estilo y oportunidades de mejora:\n"
            f"```python\n{fragmento_codigo}\n```")

@mcp.prompt()
def iniciar_sesion_debug(mensaje_error: str) -> list[Message]:
    """Inicia una sesión de ayuda interactiva para depurar un error."""
    return [
        UserMessage(f"He encontrado un error: {mensaje_error}"),
        AssistantMessage("Entendido, puedo ayudar con eso. Para empezar, ¿puedes "
                         "proporcionar el traceback completo y describir qué "
                         "estabas intentando hacer cuando ocurrió el error?")
    ]
```

3.4. El Objeto `Context`: Capacidades Avanzadas en Tiempo de Ejecución

El objeto `Context` es una de las características más potentes y sofisticadas de `fastMCP`. Actúa como un canal de comunicación y un proveedor de servicios en tiempo de ejecución. Al añadir un parámetro anotado como `Context` a cualquier función decorada (`tool`, `resource` o `prompt`), `fastMCP` inyectará automáticamente una instancia de este objeto, desbloqueando un conjunto de capacidades avanzadas.

La inyección del objeto `Context` transforma las herramientas de simples funciones sin estado en componentes conscientes de su entorno, capaces de orquestar flujos de trabajo complejos. Esto desdibuja la línea entre el cliente y el servidor, permitiendo que los servidores `fastMCP` se conviertan en agentes activos y colaborativos.

Tabla 3: Referencia Rápida del Objeto `Context`

Método/Atributo	Descripción	Ejemplo de Uso

`ctx.info(msg)`, `ctx.error(msg)`	Envía mensajes de registro al cliente. Útil para depuración y para informar al usuario sobre el progreso.	`await ctx.info("Iniciando descarga del archivo...")`
`ctx.report_progress(current, total)`	Informa al cliente sobre el progreso de una tarea de larga duración. Puede ser visualizado en la UI del cliente.	`await ctx.report_progress(50, 100)`
`await ctx.read_resource(uri)`	Permite a una herramienta leer el contenido de otro recurso en el mismo servidor. Facilita la composición interna.	`data = await ctx.read_resource("db://usuarios/101/perfil")`
`ctx.request_id`, `ctx.client_id`	Proporciona metadatos sobre la solicitud actual y el cliente que la originó.	`log.info(f"Procesando req {ctx.request_id} de {ctx.client_id}")`

`await ctx.sample(prompt)`	**(Avanzado)** Solicita al LLM del cliente que genere una finalización (inferencia) y devuelve el resultado.	`resumen = await ctx.sample(f"Resume este texto: {datos_largos}")`

Exportar a Hojas de cálculo

La función `ctx.sample()` es particularmente revolucionaria. Permite un patrón de "razonamiento en el servidor". Una herramienta puede, por ejemplo, obtener un gran volumen de datos brutos de una base de datos, usar

`ctx.sample()` para pedirle al LLM del cliente que los resuma o extraiga las entidades clave, y luego usar ese resultado procesado para realizar una segunda acción, todo dentro de la misma ejecución de la herramienta. Esta capacidad convierte al servidor de una simple colección de puntos finales pasivos en un participante activo en el proceso de razonamiento del agente.

Sección 4: Construcción de un Servidor Avanzado: Lector de Documentos Inteligente

Para consolidar los conceptos de herramientas, recursos y el objeto de contexto, esta sección guiará la construcción de un proyecto práctico y realista: un servidor `DocumentReader`. Este servidor permitirá a un LLM interactuar con archivos del sistema de ficheros, demostrando cómo los componentes de `fastMCP` trabajan en conjunto para resolver un problema del mundo real. Este caso de uso está inspirado en ejemplos prácticos de procesamiento de documentos.

4.1. Definición del Caso de Uso

El objetivo es construir un servidor `DocumentReader` que dote a un agente de IA de las siguientes capacidades:

1. **Procesar un Documento:** Aceptar la ruta de un archivo local (por ejemplo, PDF o DOCX), extraer su contenido de texto y almacenarlo para futuras consultas.
2. **Reportar Progreso:** Informar al usuario en tiempo real sobre el estado del procesamiento del documento, especialmente si es un archivo grande.
3. **Listar Documentos Recientes:** Permitir al LLM consultar una lista de los documentos que han sido procesados recientemente.
4. **Recuperar Contenido:** Permitir al LLM leer el contenido de texto extraído de un documento previamente procesado, utilizando su nombre de archivo como identificador.

Este flujo de trabajo es un ejemplo canónico del patrón de Generación Aumentada por Recuperación (Retrieval-Augmented Generation - RAG). El servidor `fastMCP` proporcionará las primitivas para las fases de "ingesta" y "recuperación", permitiendo que el LLM implemente el patrón RAG de manera estructurada y estandarizada. La herramienta `process_document` gestiona la ingesta, mientras que los recursos `docs://...` gestionan la recuperación.

4.2. Estructura del Proyecto e Implementación

Primero, es necesario instalar las dependencias adicionales para el procesamiento de documentos. La biblioteca `markitdown` es una excelente opción que soporta múltiples formatos.

Bash
uv pip install "markitdown[all]"

A continuación, se presenta la implementación paso a paso del servidor en un archivo llamado `servidor_documentos.py`.

Paso 1: Instanciación del Servidor y Almacenamiento en Memoria

Se comienza por instanciar el servidor `fastMCP` y definir una estructura de datos simple para almacenar en memoria el contenido de los documentos procesados. En una aplicación de producción, esto sería reemplazado por una base de datos o un almacén de vectores.

Python
```
# servidor_documentos.py
import os
import logging
from fastmcp import FastMCP, Context
from markitdown import MarkItDown

# Configuración básica de logging
logging.basicConfig(level=logging.INFO, format='[%(levelname)s]: %(message)s')

# Instancia del servidor, especificando dependencias para despliegues futuros
mcp = FastMCP("DocumentReader", dependencies=["markitdown[all]"])

# Almacén de datos en memoria para simular una base de datos de documentos
PROCESSED_DOCUMENTS = {}

# Instancia del procesador de documentos
md = MarkItDown()
```

Paso 2: Creación de la Herramienta de Procesamiento de Documentos

Esta herramienta será el punto de entrada para la ingesta de nuevos documentos. Utilizará el objeto `Context` para reportar el progreso y realizará validaciones de seguridad básicas.

Python

```
MAX_FILE_SIZE = 10 * 1024 * 1024  # Límite de 10MB para seguridad

def validate_file(file_path: str) -> tuple[bool, str]:
    """Valida la existencia, tamaño y tipo de archivo."""
    if not os.path.exists(file_path):
        return False, "El archivo no existe."
    if os.path.getsize(file_path) > MAX_FILE_SIZE:
        return False, f"El archivo excede el límite de tamaño de {MAX_FILE_SIZE / 1024 / 1024}
MB."
    # Se podrían añadir más validaciones de extensión aquí
    return True, "Archivo válido."

@mcp.tool
async def process_document(file_path: str, ctx: Context) -> str:
    """
    Procesa un documento local (PDF, DOCX, etc.), extrae su texto y lo
    almacena para futuras consultas.
    """
    await ctx.info(f"Iniciando procesamiento de: {file_path}")

    is_valid, message = validate_file(file_path)
    if not is_valid:
        await ctx.error(message)
        return f"Error de validación: {message}"

    try:
        await ctx.report_progress(25, 100)
        await ctx.info("Extrayendo texto del documento...")

        # Usar markitdown para extraer el contenido como Markdown
        content = md.convert(file_path)

        await ctx.report_progress(75, 100)
        filename = os.path.basename(file_path)

        # Almacenar el contenido en nuestro "almacén de datos"
        PROCESSED_DOCUMENTS[filename] = content
```

```
        await ctx.info(f"Documento '{filename}' procesado y almacenado con éxito.")
        await ctx.report_progress(100, 100)

        return f"Éxito: El documento '{filename}' ha sido procesado."
    except Exception as e:
        error_msg = f"Ocurrió un error al procesar el archivo: {e}"
        await ctx.error(error_msg)
        logging.error(error_msg)
        return error_msg
```

Paso 3: Creación de Recursos para Acceder a los Datos

Ahora se implementan los recursos que permitirán al LLM "percibir" el estado del sistema de documentos.

Python

```
@mcp.resource("docs://recent")
def list_recent_documents() -> list[str]:
    """Devuelve una lista con los nombres de los documentos procesados recientemente."""
    return list(PROCESSED_DOCUMENTS.keys())

@mcp.resource("docs://file/{filename}")
def get_document_content(filename: str) -> str:
    """
    Devuelve el contenido de texto extraído de un documento previamente procesado.
    """
    if filename in PROCESSED_DOCUMENTS:
        return PROCESSED_DOCUMENTS[filename]
    else:
        return f"Error: No se encontró el documento con el nombre '{filename}'."
```

4.3. Código Completo y Análisis

A continuación se presenta el archivo `servidor_documentos.py` completo, incluyendo el bloque para su ejecución.

Python

```
# servidor_documentos.py
import os
import logging
from fastmcp import FastMCP, Context
from markitdown import MarkItDown
```

```
# Configuración básica de logging
logging.basicConfig(level=logging.INFO, format='[%(levelname)s]: %(message)s')

# Instancia del servidor, especificando dependencias para despliegues futuros
mcp = FastMCP("DocumentReader", dependencies=["markitdown[all]"])

# Almacén de datos en memoria para simular una base de datos de documentos
PROCESSED_DOCUMENTS = {}

# Instancia del procesador de documentos
md = MarkItDown()

MAX_FILE_SIZE = 10 * 1024 * 1024  # Límite de 10MB para seguridad

def validate_file(file_path: str) -> tuple[bool, str]:
    """Valida la existencia, tamaño y tipo de archivo."""
    if not os.path.exists(file_path):
        return False, "El archivo no existe."
    if os.path.getsize(file_path) > MAX_FILE_SIZE:
        return False, f"El archivo excede el límite de tamaño de {MAX_FILE_SIZE / 1024 / 1024}
MB."
    return True, "Archivo válido."

@mcp.tool
async def process_document(file_path: str, ctx: Context) -> str:
    """
    Procesa un documento local (PDF, DOCX, etc.), extrae su texto y lo
    almacena para futuras consultas.
    """
    await ctx.info(f"Iniciando procesamiento de: {file_path}")

    is_valid, message = validate_file(file_path)
    if not is_valid:
        await ctx.error(message)
        return f"Error de validación: {message}"

    try:
        await ctx.report_progress(25, 100)
        await ctx.info("Extrayendo texto del documento...")
        content = md.convert(file_path)
        await ctx.report_progress(75, 100)
        filename = os.path.basename(file_path)
        PROCESSED_DOCUMENTS[filename] = content
        await ctx.info(f"Documento '{filename}' procesado y almacenado con éxito.")
```

```
        await ctx.report_progress(100, 100)
        return f"Éxito: El documento '{filename}' ha sido procesado."
    except Exception as e:
        error_msg = f"Ocurrió un error al procesar el archivo: {e}"
        await ctx.error(error_msg)
        logging.error(error_msg)
        return error_msg

@mcp.resource("docs://recent")
def list_recent_documents() -> list[str]:
    """Devuelve una lista con los nombres de los documentos procesados recientemente."""
    return list(PROCESSED_DOCUMENTS.keys())

@mcp.resource("docs://file/{filename}")
def get_document_content(filename: str) -> str:
    """
    Devuelve el contenido de texto extraído de un documento previamente procesado.
    """
    return PROCESSED_DOCUMENTS.get(filename, f"Error: No se encontró el documento con el nombre '{filename}'.")

if __name__ == "__main__":
    print("Iniciando Servidor Lector de Documentos...")
    # Ejecutar con transporte HTTP para permitir la interacción desde un cliente separado
    mcp.run(transport="http", host="127.0.0.1", port=8080)
```

Análisis del Flujo de Trabajo Colaborativo:

Este servidor demuestra una sinergia perfecta entre herramientas y recursos. Un flujo de trabajo típico para un agente de IA sería:

1. El usuario pide al agente que analice un archivo local: "Resume el contenido de `/ruta/a/mi_informe.pdf`".
2. El agente, utilizando el LLM, examina las herramientas disponibles y determina que `process_document` es la más adecuada.
3. El agente invoca la herramienta: `call_tool("process_document", {"file_path": "/ruta/a/mi_informe.pdf"})`.
4. El servidor ejecuta la herramienta, extrae el texto y lo almacena en `PROCESSED_DOCUMENTS`.
5. Más tarde, el usuario pregunta: "¿Qué documentos hemos discutido?".
6. El agente necesita listar los documentos. Podría descubrir el recurso `docs://recent` y leerlo para obtener la lista `["mi_informe.pdf"]`.
7. Finalmente, el usuario pregunta: "¿Cuáles son los puntos clave de `mi_informe.pdf`?".

8. El agente utiliza el recurso dinámico `docs://file/{filename}`, solicitando la URI `docs://file/mi_informe.pdf` para recuperar el texto completo.
9. Con el texto recuperado en su contexto, el LLM puede generar el resumen solicitado.

Este ejemplo ilustra cómo `fastMCP` proporciona los bloques de construcción ideales para sistemas de IA complejos que necesitan interactuar con datos externos de manera estructurada.

Sección 5: Pruebas, Depuración e Interacción Programática

Un servidor robusto y fiable requiere un proceso de pruebas y depuración igualmente robusto. El ecosistema `fastMCP` ha sido diseñado con la "testeabilidad" como una característica de primer orden, proporcionando un conjunto de herramientas que aceleran el ciclo de desarrollo y aumentan la confianza en el código.

5.1. Depuración Visual con el MCP Inspector

El MCP Inspector es una herramienta de desarrollo visual basada en web que permite a los desarrolladores inspeccionar y probar interactivamente un servidor MCP en ejecución. Es la forma más rápida de verificar que las herramientas y recursos están definidos correctamente y se comportan como se espera.

Para lanzar el servidor `DocumentReader` con el Inspector, se utiliza el comando `fastmcp dev`:

Bash
fastmcp dev servidor_documentos.py

Este comando hace varias cosas :

- Inicia el servidor `fastMCP` en un proceso en segundo plano.
- Inicia un servidor proxy que se comunica con el servidor principal.
- Lanza una aplicación web (el Inspector) que se conecta al proxy.
- Abre automáticamente el Inspector en el navegador, generalmente en una dirección como `http://localhost:5173`.

Dentro de la interfaz del Inspector, el desarrollador puede:

- Ver una lista de todas las herramientas, recursos y prompts registrados en el servidor.
- Inspeccionar los esquemas de entrada y salida de cada componente.

- Ejecutar herramientas y leer recursos a través de un formulario interactivo, proporcionando argumentos y viendo los resultados en tiempo real.
- Observar los mensajes de protocolo intercambiados entre el cliente (el Inspector) y el servidor.

El Inspector es invaluable durante el desarrollo inicial para una retroalimentación rápida y una depuración visual.

5.2. Construcción de un Cliente Programático con `fastmcp.Client`

Para pruebas automatizadas, integraciones o para construir aplicaciones que consumen servidores MCP, `fastMCP` proporciona una potente clase `Client`. Este cliente puede interactuar con cualquier servidor MCP que cumpla con el protocolo, no solo los construidos con `fastMCP`.

A continuación, se muestra un script de Python (`cliente_prueba.py`) que se conecta al servidor `DocumentReader` (que debe estar ejecutándose con transporte HTTP) y realiza operaciones sobre él.

Python
```
# cliente_prueba.py
import asyncio
from fastmcp import Client

# Asegúrate de crear un archivo de prueba llamado 'test.txt' en el mismo directorio
# con algún contenido para que el script funcione.
with open("test.txt", "w") as f:
    f.write("Este es el contenido de un archivo de prueba.")

async def main():
    server_url = "http://127.0.0.1:8080/mcp/"
    print(f"Conectando al servidor en {server_url}...")

    try:
        # El cliente se usa como un gestor de contexto asíncrono.
        async with Client(server_url) as client:
            # 1. Verificar la conexión
            is_alive = await client.ping()
            print(f"Servidor vivo: {is_alive}")

            # 2. Listar herramientas disponibles
            tools = await client.list_tools()
            print(f"Herramientas disponibles: {[tool.name for tool in tools]}")
```

```
            # 3. Llamar a la herramienta 'process_document'
            #    Nota: El servidor necesita acceso a esta ruta de archivo.
            file_path_to_process = os.path.abspath("test.txt")
            print(f"\nProcesando documento: {file_path_to_process}")
            result = await client.call_tool(
                "process_document",
                {"file_path": file_path_to_process}
            )
            # El resultado es un objeto estructurado, accedemos al contenido
            print(f"Resultado del procesamiento: {result.content.text}")

            # 4. Leer el recurso 'docs://recent'
            print("\nConsultando documentos recientes...")
            resource_result = await client.read_resource("docs://recent")
            recent_docs = resource_result.content.text
            print(f"Documentos recientes: {recent_docs}")

            # 5. Leer el recurso dinámico 'docs://file/{filename}'
            filename = "test.txt"
            print(f"\nRecuperando contenido de '{filename}'...")
            content_result = await client.read_resource(f"docs://file/{filename}")
            content = content_result.content.text
            print(f"Contenido recuperado:\n---\n{content}\n---")

    except Exception as e:
        print(f"\nOcurrió un error al interactuar con el servidor: {e}")

if __name__ == "__main__":
    import os
    asyncio.run(main())
```

Este script demuestra el flujo completo de interacción programática: conectar, descubrir, actuar (llamar herramienta) y percibir (leer recursos).

5.3. Estrategias de Pruebas Unitarias y de Integración

Si bien la interacción con un servidor en ejecución es útil, es ineficiente y frágil para las pruebas automatizadas. Probar sistemas distribuidos a través de la red introduce latencia y posibles puntos de fallo (puertos ocupados, problemas de red). `fastMCP` ofrece una solución mucho más elegante y eficiente: las pruebas "en proceso" (*in-process*).

Este patrón de prueba es una consecuencia deliberada del diseño de `fastMCP` y representa una ventaja competitiva clave. En lugar de proporcionar una URL al constructor del cliente, se

puede pasar la instancia del objeto del servidor `mcp` directamente. El cliente se conectará al servidor en memoria, eliminando por completo la capa de red y los procesos separados.

Esto hace que las pruebas unitarias y de integración sean:

- **Extremadamente rápidas:** Las llamadas son directas a funciones en memoria, sin sobrecarga de red.
- **Fiables:** Se eliminan las fuentes de fragilidad relacionadas con la red y la gestión de procesos.
- **Fáciles de depurar:** Todo se ejecuta en un único proceso, lo que simplifica el uso de depuradores.

A continuación se muestra un ejemplo de cómo escribir pruebas para el `servidor_documentos.py` utilizando el popular framework `pytest`. Cree un archivo `test_servidor.py`.

Python

```
# test_servidor.py
import pytest
from fastmcp import Client, FastMCP

# Importar la instancia del servidor desde nuestro archivo
from servidor_documentos import mcp as document_server_mcp

# pytest.mark.asyncio es necesario para que pytest ejecute pruebas asíncronas
@pytest.mark.asyncio
async def test_list_recent_documents_empty():
    """Prueba que el recurso de documentos recientes esté vacío al inicio."""
    # Conectar el cliente directamente a la instancia del servidor en memoria
    async with Client(document_server_mcp) as client:
        result = await client.read_resource("docs://recent")
        # El resultado es una lista vacía en formato JSON, que se interpreta como "
        assert result.content.text == ""

@pytest.mark.asyncio
async def test_process_document_and_retrieve():
    """
    Prueba el ciclo completo: procesar un documento, verificar que aparece
    en la lista de recientes y luego recuperar su contenido.
    """
    # Crear un archivo de prueba temporal
    test_filename = "temp_test_doc.txt"
    test_content = "Hola, mundo de las pruebas."
    with open(test_filename, "w") as f:
```

```
    f.write(test_content)

# Usar una instancia limpia del servidor para cada prueba es una buena práctica,
# aunque para este ejemplo reutilizamos la global.
async with Client(document_server_mcp) as client:
    # Procesar el documento
    await client.call_tool("process_document", {"file_path": test_filename})

    # Verificar que aparece en la lista de recientes
    recent_res = await client.read_resource("docs://recent")
    assert f'["{test_filename}"]' in recent_res.content.text

    # Verificar que podemos recuperar su contenido
    content_res = await client.read_resource(f"docs://file/{test_filename}")
    assert content_res.content.text == test_content

# Limpiar el archivo de prueba
import os
os.remove(test_filename)
```

Para ejecutar estas pruebas, primero instale `pytest` y su plugin `asyncio`:

```
Bash
uv pip install pytest pytest-asyncio
```

Luego, ejecute `pytest` desde la terminal:

```
Bash
pytest
```

Este enfoque de pruebas en proceso fomenta el Desarrollo Guiado por Pruebas (TDD) y permite construir conjuntos de pruebas robustos que garantizan la calidad y facilitan la refactorización y el mantenimiento a largo plazo.

Sección 6: Despliegue y Consideraciones para Producción

Llevar un servidor MCP del entorno de desarrollo local a un entorno de producción accesible, seguro y escalable es el paso final y más crítico en el ciclo de vida de la aplicación. Esta sección detalla las estrategias de despliegue, con un enfoque práctico en Google Cloud Run, y aborda consideraciones esenciales de seguridad y rendimiento.

6.1. Estrategias de Despliegue

La elección de la estrategia de despliegue depende del caso de uso específico del servidor MCP.

1. **Despliegue Local (`stdio`):** Para herramientas que se integran con aplicaciones de escritorio (como Claude Desktop o Cursor), el transporte `stdio` es el más adecuado. La aplicación cliente gestiona el ciclo de vida del proceso del servidor. La "instalación" a menudo implica simplemente registrar la ruta al script de Python en la configuración del cliente.
2. **Auto-hospedado (`http`):** Para un control total sobre la infraestructura, el servidor se puede ejecutar con el transporte `http` en una máquina virtual (VM) o, más comúnmente, empaquetado en un contenedor Docker. Esto requiere gestionar la red, la seguridad y la escalabilidad manualmente.
3. **Nube Gestionada (PaaS - `http`):** Utilizar una Plataforma como Servicio (PaaS) es a menudo el enfoque más eficiente para servidores remotos. Plataformas como Google Cloud Run, Heroku o el servicio específico FastMCP Cloud abstraen la gestión de la infraestructura, permitiendo a los desarrolladores centrarse en el código. El despliegue se simplifica a subir el código o una imagen de contenedor.

6.2. Guía de Despliegue Detallada en Google Cloud Run

Google Cloud Run es una excelente opción para desplegar servidores `fastMCP` porque es una plataforma sin servidor (*serverless*) que escala automáticamente según la demanda, incluso a cero, lo que la hace muy rentable. El siguiente tutorial práctico se basa en la documentación oficial de Google para desplegar un servidor MCP remoto.

Paso 1: Contenerización con Docker

El primer paso es empaquetar la aplicación en una imagen de contenedor. Cree un archivo llamado `Dockerfile` en la raíz del proyecto `mi-servidor-mcp` (el que contiene `servidor_documentos.py`).

Dockerfile
Dockerfile

Usar una imagen base de Python delgada y oficial
FROM python:3.11-slim

Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app

```
# Instalar uv, el gestor de paquetes recomendado
RUN pip install uv

# Copiar los archivos de manifiesto del proyecto
COPY pyproject.toml./

# Instalar las dependencias del proyecto usando uv.
# Esto aprovecha el almacenamiento en caché de capas de Docker.
RUN uv pip install --system -r pyproject.toml

# Copiar el resto del código de la aplicación
COPY..

# Variable de entorno para que los logs de Python aparezcan inmediatamente
ENV PYTHONUNBUFFERED=1

# Cloud Run proporciona la variable de entorno PORT.
# El servidor debe escuchar en este puerto.
# Exponer el puerto para documentación.
EXPOSE $PORT

# El comando para ejecutar la aplicación.
# Usamos el CLI de fastmcp para iniciar el servidor con transporte http,
# escuchando en todas las interfaces (0.0.0.0) y en el puerto que
# nos proporciona Cloud Run.
CMD
```

Paso 2: Subir la Imagen a Google Artifact Registry

Artifact Registry es el servicio recomendado en Google Cloud para almacenar y gestionar imágenes de contenedor. (Asegúrese de tener `gcloud` CLI instalado y configurado).

Habilite las APIs necesarias:

Bash

```
gcloud services enable run.googleapis.com artifactregistry.googleapis.com
cloudbuild.googleapis.com
```

1.

Cree un repositorio en Artifact Registry:

Bash

```
gcloud artifacts repositories create remote-mcp-servers \
  --repository-format=docker \
  --location=us-central1 \
```

--description="Repositorio para servidores MCP remotos"

2.
3. **Construya y suba la imagen usando Cloud Build:** Este comando empaqueta el directorio actual, lo sube a Cloud Build, que construye la imagen de Docker y la empuja a su repositorio de Artifact Registry. Reemplace

`PROJECT_ID` con su ID de proyecto de Google Cloud.
Bash
gcloud builds submit --tag
us-central1-docker.pkg.dev/PROJECT_ID/remote-mcp-servers/mcp-document-reader:latest

4.

Paso 3: Despliegue en Cloud Run

Con la imagen en el registro, el despliegue es una sola línea de comando.

Bash
gcloud run deploy mcp-document-reader \
--image
us-central1-docker.pkg.dev/PROJECT_ID/remote-mcp-servers/mcp-document-reader:latest \
--region=us-central1 \
--no-allow-unauthenticated

La bandera `--no-allow-unauthenticated` es crucial. Asegura que el servicio sea privado y que solo las identidades autenticadas y autorizadas puedan invocarlo.

Paso 4: Autenticación y Acceso

Para que un cliente (ya sea un script, otra aplicación o un usuario) pueda acceder al servicio desplegado, debe autenticarse. Google Cloud utiliza IAM (Identity and Access Management) para controlar el acceso.

1. **Asignar el rol `run.invoker`:** La identidad que necesita acceder al servicio (por ejemplo, una cuenta de servicio para una aplicación o su propia cuenta de usuario para pruebas) debe tener el rol `roles/run.invoker` en el servicio de Cloud Run desplegado.

Bash
gcloud run services add-iam-policy-binding mcp-document-reader \
--member="user:your-email@example.com" \

--role="roles/run.invoker" \
--region=us-central1

2.

Autenticación del Cliente: El cliente debe obtener un token de identidad de Google y presentarlo en la cabecera `Authorization` de cada solicitud HTTP como un token `Bearer`. Si está ejecutando un script cliente desde una máquina donde ha iniciado sesión con `gcloud`, puede obtener un token mediante el siguiente comando:
Bash
gcloud auth print-identity-token

3. El script cliente debería incluir este token en sus solicitudes al endpoint de Cloud Run.

6.3. Seguridad y Autenticación en `fastMCP`

Más allá de la autenticación a nivel de infraestructura proporcionada por la nube, `fastMCP 2.0` incluye potentes capacidades de autenticación a nivel de aplicación, ideales para escenarios multiusuario o para proteger APIs con estándares como OAuth2.

El framework tiene soporte nativo para proveedores de identidad como Google, GitHub, Microsoft Azure, Auth0 y WorkOS. Esto permite, por ejemplo, construir un servidor MCP que solo pueda ser utilizado por usuarios que hayan iniciado sesión con su cuenta de GitHub y pertenezcan a una organización específica. La configuración de estos proveedores se realiza de forma declarativa al instanciar el servidor `fastMCP`, simplificando enormemente la implementación de flujos de autenticación complejos.

Al trabajar con APIs que requieren tokens, como la API de GitHub, es fundamental seguir las mejores prácticas de seguridad para el manejo de secretos :

- **Nunca codificar secretos:** Jamás incluya tokens de acceso, claves de API o contraseñas directamente en el código fuente.
- **Utilizar variables de entorno:** Almacene los secretos en variables de entorno. Las plataformas de despliegue como Cloud Run facilitan la inyección segura de variables de entorno en el contenedor.
- **Archivos `.env` para desarrollo local:** Para el desarrollo, utilice archivos `.env` (y asegúrese de que estén en su `.gitignore`) para cargar variables de entorno.
- **Principio de mínimo privilegio:** Al crear tokens (por ejemplo, un Token de Acceso Personal de GitHub), conceda solo los permisos (scopes) estrictamente necesarios para que la herramienta funcione.

6.4. Buenas Prácticas para Producción

- **Configuración Declarativa con `fastmcp.json`:** Para despliegues complejos o para estandarizar la configuración entre entornos, `fastMCP 2.0` introduce el archivo `fastmcp.json`. Este archivo permite definir de forma declarativa el punto de entrada del servidor, las dependencias, la configuración del transporte y otros metadatos. Esto hace que los servidores sean más portátiles y los despliegues más reproducibles, sentando las bases para un ecosistema donde los servidores MCP puedan ser compartidos y desplegados como imágenes de contenedor.
- **Gestión de Recursos:** En producción, un servidor debe ser resistente.
    - **Límites y Validación:** Implemente validaciones estrictas en las entradas, como límites de tamaño de archivo (como se hizo en el `DocumentReader`) para prevenir ataques de denegación de servicio.
    - **Procesamiento Asíncrono:** Utilice `async/await` para operaciones de E/S (llamadas a red, lectura de archivos) para que el servidor pueda manejar múltiples solicitudes concurrentes sin bloquearse.
    - **Rate Limiting y Health Checks:** Considere añadir un middleware o utilizar las características de su plataforma de despliegue para implementar limitación de velocidad (rate limiting) y puntos finales de comprobación de estado (health checks).

La existencia de guías de despliegue detalladas para plataformas de primer nivel como Google Cloud Run y un servicio gestionado como FastMCP Cloud es un fuerte indicador de la madurez del proyecto. Demuestra que `fastMCP` ha sido diseñado no solo para la experimentación, sino para un uso empresarial serio, reduciendo la fricción operativa y permitiendo a las organizaciones integrar sus soluciones en flujos de trabajo de CI/CD siguiendo las mejores prácticas de la industria.

Sección 7: Funcionalidades Avanzadas de fastMCP 2.0

Para los desarrolladores que han dominado los fundamentos, `fastMCP 2.0` ofrece un conjunto de características avanzadas que lo elevan de una simple biblioteca de servidor a un verdadero framework de integración de IA. Estas capacidades permiten construir arquitecturas de IA complejas, modulares y mantenibles, y, lo que es más importante, sirven como un puente para "IA-ficar" la infraestructura de TI existente de una empresa con un esfuerzo mínimo.

7.1. Composición de Servidores: Construyendo Microservicios MCP

Inspirado en el patrón de arquitectura de microservicios, `fastMCP` permite componer múltiples servidores más pequeños en una única aplicación cohesiva. Esto promueve la modularidad, la reutilización y la separación de preocupaciones. Cada servidor puede ser desarrollado y probado de forma independiente y luego "montado" en un servidor principal.

El método `mcp.mount("prefijo", otro_servidor)` expone dinámicamente todas las herramientas y recursos del `otro_servidor` bajo el `prefijo` especificado.

Python
```
from fastmcp import FastMCP

# Servidor 1: Utilidades de Clima
weather_server = FastMCP(name="Weather")
@weather_server.tool
def get_forecast(city: str) -> str:
    return f"Soleado en {city}"

# Servidor 2: Calculadora
calc_server = FastMCP(name="Calculator")
@calc_server.tool
def add(a: int, b: int) -> int:
    return a + b

# Servidor Principal que compone los otros dos
main_app = FastMCP(name="MainApp")

# Montar los subservidores con prefijos
main_app.mount("weather", weather_server)
main_app.mount("calc", calc_server)

# Ahora, main_app expone las herramientas 'weather_get_forecast' y 'calc_add'.
# El LLM las verá como herramientas calificadas, por ejemplo:
# call_tool("weather_get_forecast", {"city": "Madrid"})

if __name__ == "__main__":
    main_app.run()
```

Este patrón es extremadamente poderoso para construir sistemas complejos, permitiendo que diferentes equipos trabajen en diferentes "microservicios" MCP de forma independiente.

7.2. Creación de Proxies: Un Único Punto de Entrada

`fastMCP` puede actuar como un proxy inteligente frente a cualquier otro servidor MCP, sin importar si fue construido con `fastMCP`, si es local o remoto, o en qué lenguaje está escrito. Esto es análogo a un patrón de API Gateway.

Un proxy MCP puede ser útil para:

- **Unificar múltiples servidores:** Exponer varios servidores MCP de backend bajo una única URL de cara al cliente.
- **Añadir una capa de autenticación:** Colocar un proxy con autenticación empresarial frente a un servidor interno que no la tiene.
- **Transformar solicitudes/respuestas:** Modificar las solicitudes antes de que lleguen al backend o enriquecer las respuestas antes de que lleguen al cliente.
- **Hacer accesible un servidor local:** Exponer un servidor que se ejecuta con `stdio` (como el de Claude Desktop) a través de la red mediante un proxy `http`.

Python
```
from fastmcp import FastMCP, Client

# El cliente apunta a cualquier servidor MCP de backend.
# Podría ser una instancia local, un script, o una URL remota.
backend_client = Client("http://api.servicioclima.com/mcp/sse")

# Crear un servidor proxy a partir del cliente.
proxy_server = FastMCP.from_client(backend_client, name="ProxyDeClima")

# Ejecutar el proxy localmente, por ejemplo, para que Claude Desktop pueda usarlo.
if __name__ == "__main__":
    proxy_server.run() # Por defecto, se ejecuta con stdio
```

7.3. Generación Automática desde APIs Existentes

Quizás la característica más estratégica de `fastMCP 2.0` es su capacidad para generar automáticamente un servidor MCP completo a partir de una especificación OpenAPI (anteriormente Swagger) o una aplicación FastAPI existente.

Esta es una funcionalidad "puente" de un valor incalculable. La mayoría de las organizaciones tienen cientos, si no miles, de APIs REST internas y externas. Reimplementarlas como servidores MCP sería una tarea titánica. `fastMCP` elimina esta barrera, permitiendo que la vasta superficie de la API de una empresa se vuelva accesible para los LLMs de forma masiva y casi instantánea.

Python
```
from fastapi import FastAPI
from fastmcp import FastMCP

# 1. Una aplicación FastAPI existente y estándar.
fastapi_app = FastAPI(title="Mi API de Items")

@fastapi_app.get("/items/{item_id}")
```

```
def get_item(item_id: int):
    """Obtiene un item por su ID."""
    return {"id": item_id, "name": f"Item {item_id}"}

@fastapi_app.post("/items/")
def create_item(name: str, price: float):
    """Crea un nuevo item."""
    # Lógica para crear el item...
    return {"status": "creado", "name": name, "price": price}

# 2. Generar un servidor MCP a partir de la aplicación FastAPI.
#    fastMCP introspeccionará los endpoints, parámetros, tipos y docstrings
#    y los convertirá en herramientas y recursos MCP.
mcp_server = FastMCP.from_fastapi(fastapi_app)

# 3. Ejecutar el servidor MCP generado.
#    Ahora un LLM puede usar herramientas como 'get_items_item_id' y 'create_item_items'.
if __name__ == "__main__":
    mcp_server.run(transport="http", port=8081)
```

Esta capacidad posiciona a `fastMCP` no solo como una herramienta para construir nuevas aplicaciones de IA, sino como la capa de "IA-ficación" para toda la infraestructura de TI existente de una organización. Es el camino más corto para desbloquear el valor latente en las APIs existentes, permitiendo que los agentes de IA las orquesten para automatizar flujos de trabajo complejos.

Conclusión

El Protocolo de Contexto de Modelo (MCP) representa un cambio de paradigma fundamental en la forma en que los Modelos de Lenguaje Grandes interactúan con el mundo digital. Al proponer un estándar abierto y universal, similar al impacto de USB-C en el hardware, MCP está sentando las bases para un ecosistema de IA más interoperable, modular e innovador.

Dentro de este ecosistema, `fastMCP` emerge no solo como una implementación del protocolo en Python, sino como el framework de referencia para el desarrollo de nivel de producción. Su filosofía de diseño, centrada en la simplicidad, la eficiencia del desarrollador y la robustez, reduce drásticamente las barreras de entrada para crear herramientas de IA potentes. A través de su API declarativa basada en decoradores, `fastMCP` abstrae la complejidad del protocolo, permitiendo a los desarrolladores concentrarse en la lógica de negocio que aporta valor.

Este informe ha recorrido el ciclo de vida completo del desarrollo de un servidor `fastMCP`, desde la configuración inicial y la creación de componentes básicos hasta la construcción de aplicaciones avanzadas, estrategias de prueba exhaustivas y el despliegue en plataformas en

la nube de nivel empresarial como Google Cloud Run. Se ha demostrado que `fastMCP` es más que una biblioteca: es un ecosistema completo que incluye herramientas de depuración visual, un cliente programático robusto y patrones de prueba en memoria que promueven la fiabilidad y la velocidad.

Las características avanzadas de `fastMCP 2.0`, como la composición de servidores, el proxying y, de manera crucial, la generación automática a partir de APIs FastAPI y OpenAPI, lo posicionan como una herramienta estratégica para la transformación digital. `fastMCP` no solo permite crear nuevas herramientas desde cero, sino que actúa como un puente vital, permitiendo a las organizaciones exponer de forma segura y masiva su infraestructura de TI existente a la nueva generación de agentes de IA.

Para los desarrolladores y arquitectos de Python que buscan construir la próxima generación de aplicaciones impulsadas por IA, dominar `fastMCP` ya no es una opción, sino una necesidad. Proporciona el camino más rápido, seguro y escalable desde la concepción de una idea hasta un servidor MCP robusto y desplegado en producción, listo para potenciar a los agentes de IA con las herramientas y el contexto que necesitan para resolver problemas del mundo real.

Fuentes usadas en el informe

medium.com
FastMCP: The fastway to build MCP servers. | by CellCS | Medium
Se abre en una ventana nueva

apidog.com
A Beginner's Guide to Use FastMCP - Apidog
Se abre en una ventana nueva

github.com
Model Context Protocol - GitHub
Se abre en una ventana nueva

github.com

cyanheads/model-context-protocol-resources: Exploring the Model Context Protocol (MCP) through practical guides, clients, and servers I've built while learning about this new protocol. - GitHub
Se abre en una ventana nueva

gofastmcp.com
Welcome to FastMCP 2.0! - FastMCP
Se abre en una ventana nueva

github.com
jlowin/fastmcp: The fast, Pythonic way to build MCP servers and clients - GitHub
Se abre en una ventana nueva

github.com
microsoft/mcp-for-beginners: This open-source curriculum introduces the fundamentals of Model Context Protocol (MCP) through real-world, cross-language examples in .NET, Java, TypeScript, JavaScript, Rust and Python. Designed for developers, it focuses on practical techniques for building modular, scalable, - GitHub
Se abre en una ventana nueva

github.com
AI-App/JLowin.FastMCP: The fast, Pythonic way to build MCP servers and clients - GitHub
Se abre en una ventana nueva

datacamp.com
Building an MCP Server and Client with FastMCP 2.0 - DataCamp
Se abre en una ventana nueva

pypi.org
fastmcp 2.2.0 - PyPI
Se abre en una ventana nueva

pondhouse-data.com
Creating an MCP Server Using FastMCP: A Comprehensive Guide - Pondhouse Data
Se abre en una ventana nueva

gofastmcp.com
Quickstart - FastMCP
Se abre en una ventana nueva

deeplearningnerds.com
Getting Started with FastMCP Cloud: Set up a Workspace and connect your GitHub Account
Se abre en una ventana nueva

jlowin.dev
Stop Vibe-Testing Your MCP Server
Se abre en una ventana nueva

jlowin.dev
Introducing FastMCP 2.0 - Mostly Harmless
Se abre en una ventana nueva

firecrawl.dev
How to Build MCP Servers in Python: Complete FastMCP Tutorial for AI Developers
Se abre en una ventana nueva

cloud.google.com
Build and deploy a remote MCP server on Cloud Run
Se abre en una ventana nueva

pypi.org
fastmcp 2.2.7 - PyPI
Se abre en una ventana nueva

github.com
The official Python SDK for Model Context Protocol servers and clients - GitHub
Se abre en una ventana nueva

gofastmcp.com
Running Your Server - FastMCP
Se abre en una ventana nueva

cloud.google.com
Build and Deploy a Remote MCP Server to Google Cloud Run in Under 10 Minutes
Se abre en una ventana nueva

github.com
Releases · jlowin/fastmcp - GitHub
Se abre en una ventana nueva

github.com
GitHub's official MCP Server
Se abre en una ventana nueva

Fuentes consultadas, pero no usadas en el informe

medium.com
Developing with Agent Development Kit —featuring FastMCP & ngrok | by Glen Yu | Google Cloud - Community | Medium
Se abre en una ventana nueva

reddit.com
Tutorial: Build and Deploy an MCP Server to Google Cloud Run - Reddit
Se abre en una ventana nueva

youtube.com
FastMCP — the best way to build an MCP server with Python - YouTube
Se abre en una ventana nueva

github.com
punkpeye/awesome-mcp-servers: A collection of MCP servers. - GitHub
Se abre en una ventana nueva

github.com
model-context-protocol-server · GitHub Topics
Se abre en una ventana nueva

github.com
Issues · jlowin/fastmcp - GitHub
Se abre en una ventana nueva
Pensamientos