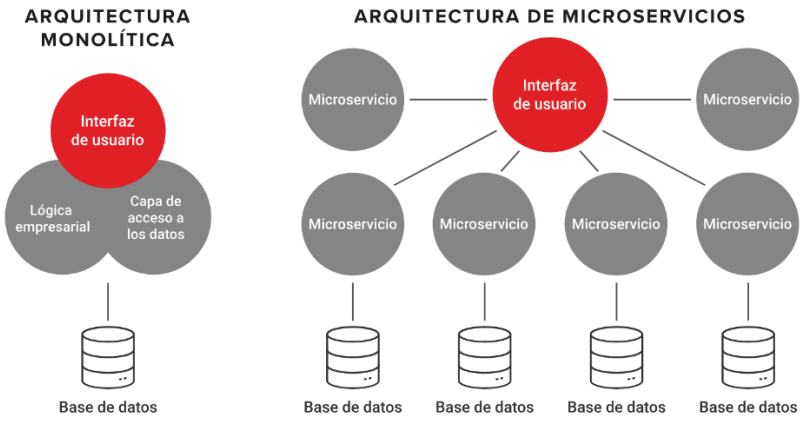
INSTITUTO POLITÉCNICO NACIONAL

Escuela Superior de Cómputo

Sistemas Distribuidos

Profesor: Chadwick Carreto Arrellano



7CM1

Práctica 7 “Microservicios”

Juan Fernando León Medellín

ANTECEDENTE

1. Microservicios

Los **microservicios** son una arquitectura de software que organiza una aplicación como un conjunto de pequeños servicios independientes, cada uno responsable de una función específica. Cada servicio tiene su propia lógica de negocio y base de datos, lo que facilita su mantenimiento, escalabilidad y despliegue. Esta arquitectura se diferencia de la arquitectura monolítica, donde todos los componentes están acoplados en una única unidad.

Características principales de los microservicios:

- **Descentralización:** Cada microservicio es autónomo y tiene su propia base de datos.
- **Escalabilidad:** Los microservicios pueden escalarse independientemente, lo que permite manejar cargas de trabajo específicas.
- **Despliegue independiente:** Los servicios pueden desplegarse y actualizarse sin afectar a toda la aplicación.
- **Desarrollo ágil:** Los microservicios facilitan la asignación de equipos a servicios independientes, permitiendo ciclos de desarrollo más rápidos.

Ventajas de los microservicios:

- Mayor **flexibilidad** y **agilidad** en el desarrollo.
- Escalabilidad **independiente** de los componentes.
- Mantenimiento simplificado.
- Posibilidad de utilizar diferentes **lenguajes** o **tecnologías** para cada microservicio.

2. APIs RESTful

Las **APIs RESTful** (Representational State Transfer) son un estilo arquitectónico para la creación de servicios web. REST se basa en el uso de **métodos HTTP** (como GET, POST, PUT, DELETE) para interactuar con los recursos de la web. En un servicio RESTful, los recursos son representaciones de datos o funcionalidades que pueden ser manipuladas a través de estas solicitudes HTTP.

Características clave de una API RESTful:

- **Stateless (sin estado):** Cada solicitud del cliente al servidor debe contener toda la información necesaria para procesar la solicitud, es decir, el servidor no mantiene el estado entre las peticiones.
- **Uso de métodos HTTP:** REST utiliza los métodos HTTP estándar para realizar operaciones en los recursos:
    - **GET:** Obtener datos.
    - **POST:** Crear un nuevo recurso.
    - **PUT:** Actualizar un recurso existente.
    - **DELETE:** Eliminar un recurso.
- **Recursos identificados por URIs:** Cada recurso tiene una URI (Uniform Resource Identifier) única que lo identifica.

3. Flask: Framework para Microservicios

Flask es un **framework** de microservicios para Python que permite desarrollar aplicaciones web ligeras y escalables. Es muy utilizado para crear APIs RESTful debido a su simplicidad y flexibilidad.

Flask no impone una estructura rígida, lo que lo hace ideal para desarrollos rápidos de aplicaciones y microservicios. Permite integrar fácilmente otras bibliotecas, como **SQLAlchemy** para bases de datos, o **Flask-RESTful** para facilitar la construcción de APIs REST.

Características de Flask:

- **Ligero y flexible:** No incluye funcionalidades innecesarias, lo que permite que el desarrollador defina las rutas y la estructura del proyecto según sus necesidades.
- **Desarrollo rápido:** Su sintaxis sencilla y la posibilidad de crear un servidor de desarrollo lo hacen ideal para la creación de prototipos y microservicios.
- **Extensible:** Se puede integrar con muchas bibliotecas y herramientas para ampliar sus capacidades.

4. Operaciones HTTP en el Microservicio

En el microservicio que desarrollamos, se implementaron las operaciones básicas de HTTP para interactuar con los recursos (en este caso, los productos):

- **GET:** Permite obtener los datos de los productos. Se utiliza para leer información sin modificarla.
- **POST:** Se usa para crear nuevos productos. Permite agregar nuevos recursos al sistema.
- **PUT:** Actualiza los datos de un producto existente. Permite modificar los recursos ya existentes en el servidor.
- **DELETE:** Elimina un producto del sistema. Permite borrar los recursos que ya no son necesarios.

Estas operaciones son fundamentales en las APIs RESTful y permiten realizar todas las funciones básicas necesarias para manejar recursos en un servidor.

5. Ventajas de Usar Microservicios con Flask

Al combinar los **microservicios** con **Flask** y una arquitectura **RESTful**, se pueden obtener varias ventajas:

- **Desarrollo rápido y flexible**: Flask permite un desarrollo ágil, y la arquitectura de microservicios facilita la distribución de tareas entre equipos pequeños y autónomos.
- **Escalabilidad independiente**: Los microservicios pueden escalarse según las necesidades específicas de cada componente, lo que mejora el rendimiento y la eficiencia de la aplicación.
- **Mantenimiento y pruebas más fáciles**: La modularidad de los microservicios facilita la localización y corrección de errores sin afectar a toda la aplicación.
- **Integración con otras tecnologías**: Cada microservicio puede estar basado en diferentes lenguajes de programación, bases de datos o incluso servicios externos, lo que brinda gran flexibilidad a la hora de elegir herramientas.

PLANTEAMIENTO DEL PROBLEMA

En el desarrollo de aplicaciones modernas, especialmente aquellas que requieren ser escalables y fáciles de mantener, la arquitectura de **microservicios** ha ganado gran relevancia. Esta arquitectura permite dividir una aplicación compleja en varios servicios independientes, cada uno de ellos encargado de una tarea específica. Esto facilita el desarrollo, el mantenimiento y la escalabilidad de las aplicaciones, pero también plantea ciertos retos, como la integración de múltiples servicios y la correcta gestión de las comunicaciones entre ellos.

En el contexto de la práctica realizada, se implementó un **microservicio básico** utilizando el framework **Flask**, con el objetivo de ofrecer una solución que permita gestionar productos mediante operaciones **CRUD** (Crear, Leer, Actualizar, Eliminar). Este microservicio tiene como base una API **RESTful**, que utiliza los métodos HTTP (GET, POST, PUT, DELETE) para interactuar con los recursos de la aplicación.

El principal problema a resolver es la **gestión eficiente de productos** dentro de un sistema distribuido o microservicios, donde se espera que cada solicitud sea gestionada de manera independiente, sin que un servicio dependa directamente de otro. Esto requiere una correcta implementación de las solicitudes HTTP, el manejo de datos y la actualización o eliminación de recursos en tiempo real.

A pesar de que se busca simplificar la interacción con el sistema, existen diversos desafíos:

1. **Interacción entre microservicios**: Aunque cada microservicio opera de manera independiente, se deben coordinar entre sí para que las aplicaciones funcionen correctamente. Es fundamental establecer una comunicación clara y eficiente entre los servicios, lo que requiere una adecuada configuración de las solicitudes y respuestas en la API.
2. **Persistencia y consistencia de los datos**: En una arquitectura de microservicios, los datos no se deben almacenar de manera centralizada, sino que cada servicio puede tener su propia base de datos. Esto puede generar problemas de **consistencia de los datos** y **sincronización entre servicios**, lo que dificulta el mantenimiento y la gestión.
3. **Escalabilidad**: Si bien los microservicios permiten una escalabilidad independiente, esto requiere un monitoreo constante y una correcta implementación de las rutas para evitar sobrecargar el sistema. La gestión de la carga de trabajo y la eficiencia en las respuestas también son problemas a considerar.

El objetivo principal de la práctica es **desarrollar un microservicio básico**, utilizando **Flask** y **RESTful API**, que permita gestionar productos a través de solicitudes HTTP, proporcionando una solución a la necesidad de escalabilidad y mantenimiento eficiente en aplicaciones distribuidas. A través de este ejercicio, se busca familiarizarse con la creación de servicios web, entender el funcionamiento de los microservicios, y los métodos **GET, POST, PUT y DELETE**, esenciales para la gestión de recursos en aplicaciones modernas.

MATERIALES Y MÉTODOS EMPLEADOS

Lenguaje de Programación:

- Python

Framework:

- Flask

Formato de Intercambio de Datos:

- JSON

Herramienta de Pruebas:

- Postman

PROPUESTA DE SOLUCIÓN

Para abordar el problema de gestión de productos en un sistema basado en microservicios, se propone el desarrollo de un **microservicio** utilizando el framework **Flask** para crear una API **RESTful** que permita realizar operaciones **CRUD** sobre los productos. Este enfoque permitirá que la aplicación sea escalable, independiente y fácilmente mantenible. A continuación, se detallan los elementos clave de la solución:

1. Implementación del Microservicio con Flask

El microservicio se desarrollará utilizando **Flask**, un framework ligero y flexible de Python que es ideal para crear aplicaciones web y APIs RESTful. Flask proporciona una estructura simple pero poderosa para manejar solicitudes HTTP y gestionar rutas, lo que lo convierte en una opción perfecta para proyectos de microservicios pequeños y medianos.

Rutas y operaciones RESTful: Se implementarán las operaciones básicas de una API RESTful utilizando los métodos HTTP estándar:

GET: Recuperar la lista de productos o un producto específico por su ID.

POST: Crear un nuevo producto en el sistema.

PUT: Actualizar un producto existente por su ID.

DELETE: Eliminar un producto existente por su ID.

2. Gestión de Productos en Memoria

Para simplificar el desarrollo, la solución usará una lista en memoria para almacenar los productos, lo que permitirá probar la funcionalidad básica sin la necesidad de configurar una base de datos. Cada producto tendrá los atributos más esenciales, como el **ID**, el **nombre** y el **precio**.

Si bien esta es una solución temporal, es posible ampliarla en el futuro para integrar una base de datos real, como **SQLite**, **MySQL** o **MongoDB**, para persistir los datos y garantizar que se mantengan incluso después de reiniciar el servidor.

3. Definición de la API

La API se diseñará para ser simple, permitiendo interactuar con los recursos de productos de manera eficiente. La definición de las rutas será la siguiente:

GET /productos: Devuelve la lista de todos los productos.

GET /productos/{id}: Devuelve un producto específico por su ID.

POST /productos: Crea un nuevo producto.

PUT /productos/{id}: Actualiza un producto existente por su ID.

DELETE /productos/{id}: Elimina un producto existente por su ID.

Este diseño básico cubre las operaciones más comunes que se necesitan para manejar recursos en un sistema distribuido.

4. Validación y Respuestas de la API

Se implementarán mecanismos de validación para asegurarse de que los datos enviados sean correctos antes de realizar operaciones de actualización o creación de productos. Si los datos proporcionados no son válidos (por ejemplo, si el precio es negativo o el nombre está vacío), el sistema devolverá un **código de error 400 (Bad Request)** y un mensaje de error adecuado.

Además, la API responderá con códigos de estado HTTP apropiados:

200 OK: Para una operación exitosa.

201 Created: Para indicar que un producto ha sido creado exitosamente.

400 Bad Request: Si los datos enviados no son válidos.

404 Not Found: Si se intenta acceder o modificar un producto que no existe.

500 Internal Server Error: En caso de fallos internos inesperados.

5. Escalabilidad y Mantenimiento

La solución propuesta es escalable, ya que el microservicio puede ampliarse para incluir más productos, integrar bases de datos y manejar diferentes operaciones a medida que la demanda crezca. Los microservicios pueden ser desplegados de manera independiente, y Flask permite la adición de extensiones y middleware para mejorar el rendimiento y la seguridad.

Además, la estructura modular del microservicio facilita su mantenimiento, ya que cada servicio se puede actualizar o modificar sin afectar a otros servicios del sistema.

6. Uso de Postman para Pruebas

Postman se utilizará para realizar pruebas de la API. Al emplear esta herramienta, se podrán probar las distintas operaciones de la API (GET, POST, PUT, DELETE) de manera rápida y eficiente. Postman también permite verificar las respuestas de la API, asegurándose de que el servicio esté funcionando como se espera.

7. Consideraciones Futuras

Si bien esta solución está basada en una implementación en memoria, la propuesta contempla la posibilidad de **migrar a una base de datos persistente** (como SQL o NoSQL) para manejar grandes volúmenes de datos de productos. Esto garantizaría que los datos no se pierdan cuando el servidor se reinicie.

DESARROLLO DE LA SOLUCIÓN

En el siguiente enlace se puede encontrar el código de la práctica:

https://github.com/JFernandoLe/SD_PRACTICA7.git

RESULTADOS

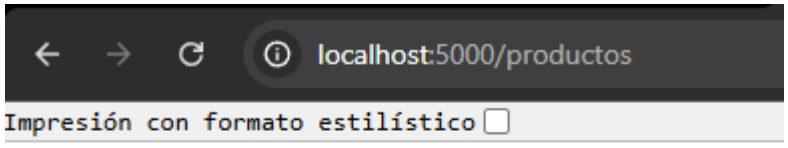
Instalamos FLASK

```
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica7\microservicio> pip install -r requirements.txt
Defaulting to user installation because normal site-packages is not writeable
Collecting Flask==2.2.5 (from -r requirements.txt (line 1))
  Downloading Flask-2.2.5-py3-none-any.whl.metadata (3.9 kB)
Requirement already satisfied: Werkzeug>=2.2.2 in c:\users\leonm\appdata\roaming\python\python313\site-packages (from Flask==2.2.5->-r requirements.txt (line 1)) (3.1.3)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\leonm\appdata\roaming\python\python313\site-packages (from Jinja2>=3.0->Flask==2.2.5->-r requirements.txt (line 1)) (3.0.2)
Downloading Flask-2.2.5-py3-none-any.whl (101 kB)
Installing collected packages: Flask
  Attempting uninstall: Flask
    Found existing installation: Flask 3.1.0
    Uninstalling Flask-3.1.0:
      Successfully uninstalled Flask-3.1.0
  WARNING: The script flask.exe is installed in 'C:\Users\leonm\AppData\Roaming\Python\Python313\Scripts' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed Flask-2.2.5
```

Ejecutamos el microservicio

```
PS C:\Users\leonm\OneDrive\Escritorio\documentos\6 semestre\Sistemas Distribuidos\Practica7\microservicio> & C:/Python313/python.exe "c:/Users/leonm/OneDrive/Escritorio/documentos/6 semestre/Sistemas Distribuidos/Practica7/microservicio/app.py"
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.100.130:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 107-980-968
127.0.0.1 - - [02/Apr/2025 11:01:09] "GET /productos%0A HTTP/1.1" 404 -
127.0.0.1 - - [02/Apr/2025 11:02:21] "GET /productos%0A%0A HTTP/1.1" 404 -
127.0.0.1 - - [02/Apr/2025 11:03:07] "GET /productos HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 11:03:07] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [02/Apr/2025 11:04:18] "GET /productos%0A HTTP/1.1" 404 -
127.0.0.1 - - [02/Apr/2025 11:04:52] "GET /productos HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 11:04:57] "GET /productos HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 11:06:51] "GET /productos HTTP/1.1" 200 -
127.0.0.1 - - [02/Apr/2025 11:07:09] "GET /productos HTTP/1.1" 200 -
```

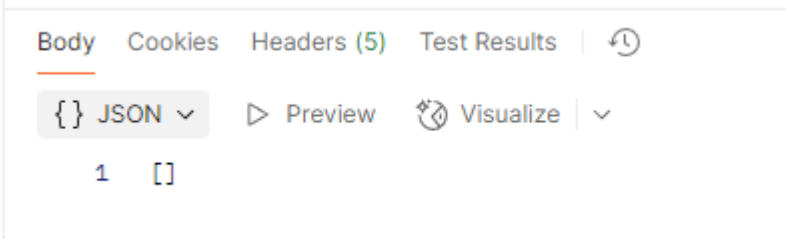
Probamos el microservicio en http://localhost:5000



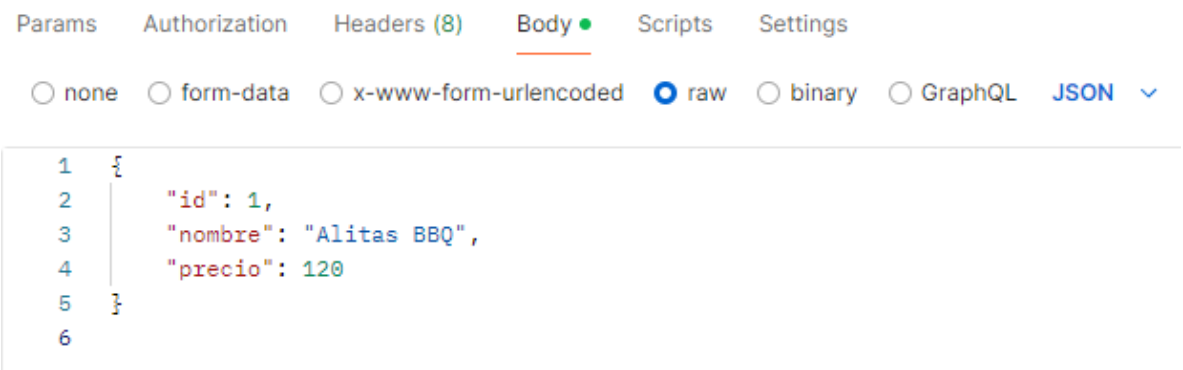
[]

Pruebas con Postman. Obtener todos los productos (GET):

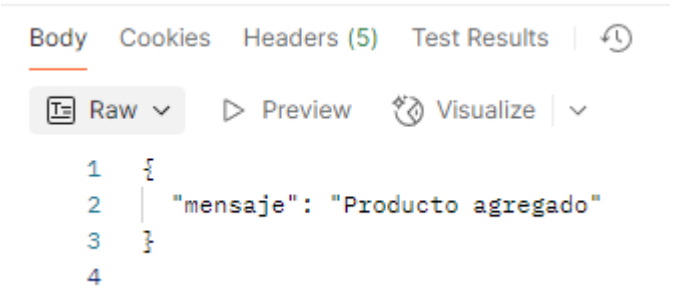
Respuesta:



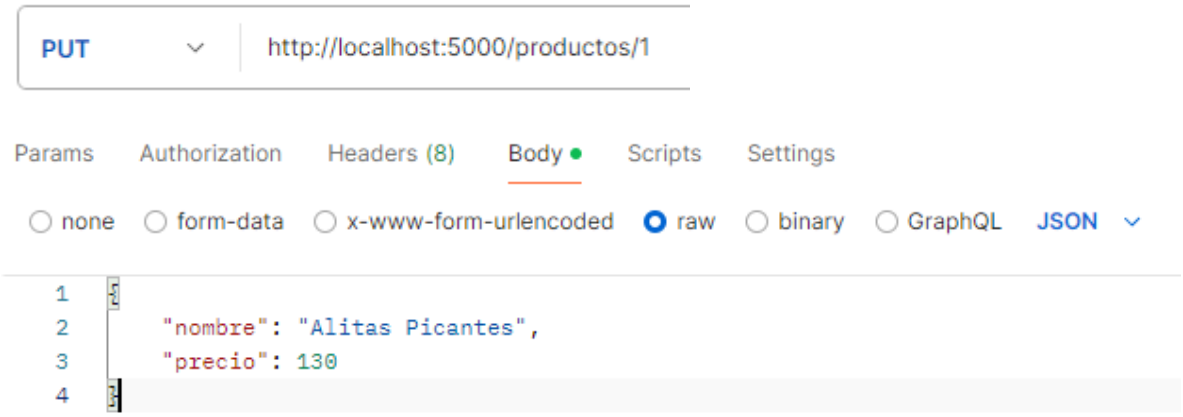
Agregar un producto (POST):



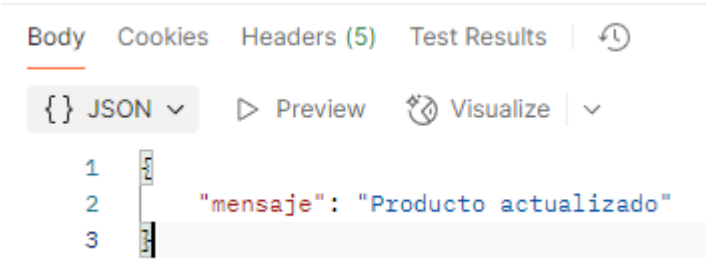
Respuesta:



Actualizar un producto (PUT):

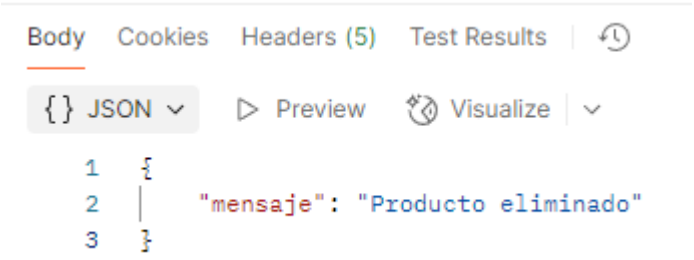


Respuesta:



Eliminar un producto (DELETE)

Respuesta:



CONCLUSIÓN

La práctica realizada ha permitido comprender y aplicar los conceptos fundamentales de la creación y prueba de servicios web utilizando el estilo arquitectónico **REST**. A lo largo del proceso, se desarrolló un servicio web básico con **Flask** en Python que permite gestionar datos de manera sencilla, realizando operaciones de **consulta** (GET), **creación** (POST) y **eliminación** (DELETE) de información, todo gestionado mediante formato **JSON**.

El uso de **Postman** ha sido clave para probar y verificar el correcto funcionamiento de la API, asegurando que las solicitudes y respuestas sean procesadas adecuadamente por el servidor. Las pruebas realizadas mediante **GET**, **POST** y **DELETE** demostraron que el servicio responde correctamente a las solicitudes y maneja adecuadamente los posibles errores.

Al mismo tiempo, la práctica ha permitido reflexionar sobre las diferencias entre los estilos arquitectónicos **REST** y **SOAP**, destacando la simplicidad y flexibilidad de REST para desarrollar aplicaciones ligeras y escalables. Además, la correcta implementación de los métodos HTTP y el manejo de datos en formato JSON resulta en un servicio eficiente y fácil de integrar con otros sistemas.

En resumen, esta práctica ha logrado el objetivo de ofrecer una solución funcional para la gestión de información mediante un servicio web RESTful, proporcionando una base sólida para el desarrollo y prueba de APIs en sistemas distribuidos. La experiencia adquirida en la creación, implementación y prueba de un servicio web es fundamental para afrontar proyectos más complejos en el futuro, donde se requiera integrar diferentes componentes o servicios.