Universidad de Las Américas
Facultad de Ingenierías y Ciencias Agropecuarias
Ingeniería De Software
Progreso 1

Nombres: Sammy Porras, Daniel Vizcarra

Fecha: 14/11/2025

Diseño e Implementación de una API REST

Objetivo de la actividad:

El objetivo de este taller es diseñar, implementar y documentar una API REST utilizando un stack tecnológico flexible, cumpliendo los principios RESTful, la definición formal mediante OpenAPI 3.0, y las prácticas de integración descritas en el escenario del taller.}

1. Requisitos técnicos

Para el desarrollo del taller se seleccionó el stack **Python + FastAPI**, cumpliendo con la flexibilidad tecnológica permitida por el enunciado. Este stack cumple todas las condiciones solicitadas:

- Implementación de los endpoints requeridos (GET, POST y GET por ID).
- Generación automática de contrato OpenAPI 3.0.
- Ejecución reproducible mediante comandos simples (`uvicorn main:app --reload`).
- Documentación Swagger incorporada por defecto.
- Endpoint de salud `/health`.
- Colección de pruebas Postman.
- README técnico con instrucciones.

2. Paso a Paso para Desplegar la Solución

En esta sección se detalla el procedimiento seguido para diseñar, implementar y poner en funcionamiento la API REST utilizando **Python + FastAPI**, cumpliendo con las condiciones mínimas establecidas para el uso de un stack tecnológico alternativo, tales como la implementación de los endpoints requeridos, la provisión de un contrato OpenAPI 3.0 válido, la ejecución reproducible del servicio, el uso de una colección de pruebas en Postman y la exposición de un endpoint de salud /health.

2.1. Selección del stack tecnológico y preparación del entorno

Como alternativa a la línea base propuesta en Java + Apache Camel, se eligió el stack **Python + FastAPI**, debido a las siguientes razones:

- FastAPI ofrece soporte nativo para **OpenAPI 3.0** y genera automáticamente la documentación Swagger.
- Permite implementar APIs REST de forma rápida, manteniendo buenas prácticas de diseño.
- La ejecución del servicio es reproducible mediante un solo comando (uvicorn), cumpliendo con las condiciones del taller.

Para preparar el entorno de trabajo se realizaron los siguientes pasos:

1. Creación de una carpeta de proyecto, por ejemplo: api-envios-fastapi/.
2. Creación y activación de un entorno virtual de Python para aislar las dependencias del proyecto:
3. python -m venv venv
4. # Windows
5. venv\Scripts\activate
6. # Linux/MacOS
7. source venv/bin/activate
8. Instalación de las dependencias necesarias:
9. pip install fastapi uvicorn

Con esto, el entorno quedó listo para iniciar la implementación de la API.

Evidencias sugeridas:

- Captura de la consola mostrando la creación y activación del entorno virtual.
- Captura de la instalación de fastapi y uvicorn.

2**.2. Estructura del proyecto**

Dentro de la carpeta del proyecto se creó el archivo principal main.py, donde se concentra la lógica de la API. La estructura mínima inicial fue la siguiente:

```
api-envios-fastapi/
├── venv/
├── main.py
└── README.md
```

Esta organización permite mantener un proyecto sencillo pero ordenado, sobre el cual se pueden añadir más módulos o paquetes en caso de que la API crezca.

Evidencia sugerida:

- Captura del explorador de archivos o de VS Code mostrando la estructura del proyecto.

2**.3. Definición del modelo de dominio**

Para representar los datos de un envío, se definió un modelo utilizando **Pydantic**, el cual permite validar de forma automática la estructura de los datos de entrada y salida:

- **Modelo Envio**: contiene atributos como id, destinatario, direccion y estado.

Este modelo se utilizó tanto en las respuestas de los endpoints como en el cuerpo del POST /envios, garantizando consistencia en la representación de los datos.

Adicionalmente, se implementó una estructura de datos en memoria (envios_db) que simula una base de datos, permitiendo realizar pruebas sin necesidad de un motor de base de datos real.

Evidencia sugerida:

- Captura del código del modelo Envio y de la lista envios_db en main.py.

2.4. Implementación de los endpoints REST

Se implementaron los tres endpoints solicitados en el enunciado del taller, cumpliendo con las operaciones básicas de consulta y creación de envíos:

1. **GET /envios**
   - Retorna la lista completa de envíos almacenados en envios_db.
   - Se declaró con tipo de respuesta List[Envio] para garantizar una salida tipada.
2. **GET /envios/{id}**
   - Recibe un identificador de envío como parámetro de ruta.
   - Recorre la colección envios_db para encontrar el envío correspondiente.
   - Si lo encuentra, devuelve sus datos; en caso contrario, retorna un mensaje de error simple (en una mejora futura podría devolverse un código de estado 404).
3. **POST /envios**
   - Recibe un objeto Envio en el cuerpo de la petición.
   - Añade el envío a la colección envios_db.
   - Retorna un mensaje de confirmación y un código de estado 201 Created.

Cada endpoint fue diseñado respetando los principios REST:

- Uso de métodos HTTP semánticos (GET, POST).
- Uso de rutas orientadas a recursos (/envios, /envios/{id}).
- Manejo de representaciones en formato JSON.

Evidencias sugeridas:

- Capturas de main.py donde se observe la declaración de cada endpoint.
- Capturas de pruebas exitosas en Swagger o Postman para cada operación.

2.5. Endpoint de salud /health

En cumplimiento de los requisitos del stack libre, se añadió un endpoint de salud:

- **GET /health**
  - Devuelve un objeto JSON sencillo, por ejemplo: {"status": "ok"}.
  - Permite verificar rápidamente que la aplicación está desplegada y respondiendo, sin necesidad de invocar los endpoints funcionales.

Este endpoint es especialmente útil en escenarios de monitoreo o despliegue en contenedores, donde herramientas externas necesitan verificar el estado del servicio.

Evidencia sugerida:

- Captura de la respuesta del endpoint /health en Swagger o Postman.

2.6. Documentación automática con Swagger/OpenAPI

Una de las ventajas principales de FastAPI es que genera automáticamente la documentación OpenAPI 3.0, cumpliendo así el requisito de incluir un contrato válido (openapi.json).

Al ejecutar la aplicación, se exponen las siguientes URLs:

- **Documentación interactiva (Swagger UI):** http://127.0.0.1:8000/docs
- **Contrato OpenAPI 3.0 en formato JSON:** http://127.0.0.1:8000/openapi.json

Desde Swagger UI fue posible:

- Visualizar todos los endpoints definidos.
- Consultar las estructuras de entrada y salida.
- Probar los métodos de forma interactiva (GET y POST) sin necesidad de herramientas externas.

El archivo openapi.json se descargó y se incluyó dentro del repositorio como parte del contrato formal de la API.

Evidencias sugeridas:

- Captura de la página /docs mostrando la lista de endpoints.
- Capturas de pruebas realizadas desde Swagger.
- Captura del JSON de OpenAPI visualizado en el navegador.

2.7. Ejecución reproducible de la API

Para garantizar que cualquier integrante del equipo, docente o evaluador pueda ejecutar la API de forma reproducible, se definió el siguiente comando estándar:

uvicorn main:app --reload

Este comando:

- Levanta el servidor en http://127.0.0.1:8000.
- Permite recargar automáticamente la aplicación ante cambios en el código (--reload).

En el README técnico se documentaron los pasos necesarios:

1. Clonar el repositorio.
2. Crear y activar el entorno virtual.
3. Instalar las dependencias.
4. Ejecutar el comando de uvicorn.

Evidencias sugeridas:

- Captura de la consola con el servidor corriendo y el log de las peticiones realizadas.

(Si más adelante decides usar Docker, aquí también se podría documentar el Dockerfile y el comando docker build / docker run como alternativa de ejecución reproducible.)

2.8. Pruebas con Postman y generación de colección

Además de las pruebas realizadas desde Swagger, se construyó una colección de Postman con los siguientes endpoints:

- GET /envios
- GET /envios/{id}
- POST /envios
- GET /health

Cada petición fue configurada con:

- URL base: http://localhost:8000.
- Método HTTP correspondiente (GET o POST).
- En el caso de POST /envios, se utilizó un cuerpo JSON con los campos del modelo Envio.

Después de verificar que todas las peticiones devolvían las respuestas esperadas, la colección se exportó en formato .json, cumpliendo el requisito de entregar una colección de pruebas con evidencias de cada endpoint.

Evidencias sugeridas:

- Capturas de Postman mostrando las respuestas correctas.
- Captura del proceso de exportación de la colección.

2**.9. Organización de evidencias y documentación**

Finalmente, se organizaron todos los artefactos del proyecto conforme a los entregables establecidos en el taller:

- Código fuente de la API (main.py).
- Contrato OpenAPI (openapi.json).
- Colección de Postman exportada en formato JSON.
- README técnico con pasos de instalación, ejecución y pruebas.
- Capturas de pantalla evidenciando el funcionamiento de cada endpoint.
- Informe escrito con la descripción detallada del proceso, conclusiones y recomendaciones.

Esta organización facilita la revisión por parte del tutor, asegura la trazabilidad del desarrollo y demuestra el cumplimiento de los criterios de evaluación definidos.

3. Link a repositorio GIT

Se utilizó el siguiente repositorio de git

4. **Preguntas**

1. ¿Qué ventajas ofrece una API REST bien diseñada frente a los enfoques tradicionales de integración?

Una API REST bien diseñada presenta múltiples ventajas frente a métodos tradicionales como la transferencia de archivos o la integración punto a punto. En primer lugar, REST permite interoperabilidad entre sistemas heterogéneos utilizando estándares ampliamente adoptados como HTTP y JSON. Además, favorece el bajo acoplamiento, de manera que los sistemas pueden evolucionar de forma independiente sin afectar a los consumidores del servicio.

Otra ventaja relevante es la escalabilidad, ya que REST se adapta fácilmente a arquitecturas distribuidas y entornos en la nube. La claridad en la definición de recursos y rutas facilita el mantenimiento, la documentación y la incorporación de nuevos desarrolladores. Finalmente, herramientas como OpenAPI permiten documentar la API de forma estandarizada, generando contratos claros y posibilitando la automatización de pruebas y clientes. Esto incrementa la confiabilidad y reduce errores de integración.

2. ¿Qué patrón de integración se aplicó en esta solución?

El patrón principal aplicado en esta solución es el Patrón de Servicio (Service API), el cual consiste en exponer funcionalidades como servicios accesibles mediante interfaces bien definidas. Este patrón favorece la integración modular entre sistemas, permitiendo que los consumidores accedan a recursos como “envíos” a través de endpoints estandarizados siguiendo los principios RESTful.

Adicionalmente, se hace referencia al patrón de Request-Reply, ya que cada operación ejecutada por el cliente recibe una respuesta inmediata del servidor en formato JSON. Este tipo de interacción permite consultas directas, operaciones CRUD y validaciones inmediatas, lo cual resulta apropiado para escenarios modernos de integración basados en APIs.

3. ¿Cuál es la importancia del contrato OpenAPI dentro del desarrollo de la API?

El contrato OpenAPI desempeña un rol central en el diseño y la integración de la API. Su importancia radica en que es la fuente de verdad que describe de forma estandarizada los endpoints, parámetros, cuerpos de petición, tipos de respuesta, códigos de estado y modelos de datos. Esto garantiza que todos los desarrolladores y sistemas externos consuman la API de manera consistente.

Además, el contrato facilita la documentación automática, la generación de clientes en distintos lenguajes, la validación de solicitudes, la detección anticipada de errores y la

automatización de pruebas. Desde una perspectiva de integración empresarial, disponer de un contrato claro reduce los riesgos, evita malentendidos y acelera el proceso de adopción de la API por parte de otras aplicaciones.

4. ¿Por qué es necesario exponer un endpoint de salud (/health)?

El endpoint `/health` permite verificar rápidamente si el servicio está activo y funcionando correctamente sin necesidad de ejecutar las operaciones completas del sistema. Este endpoint es especialmente importante en entornos de producción, despliegues automatizados, contenedores Docker o sistemas de monitoreo, donde herramientas externas requieren verificar el estado del servicio de forma periódica.

La presencia de un endpoint de health check asegura:

- Monitoreo continuo del estado del servicio,
- Detección anticipada de fallas,
- Facilidad para sistemas de orquestación (como Kubernetes),
- Mayor confiabilidad y disponibilidad del API.

Su simplicidad lo convierte en un componente esencial dentro de cualquier arquitectura moderna basada en servicios.

5. ¿Cuál es la relación entre las pruebas en Postman y la calidad de la integración?

Las pruebas realizadas en Postman permiten validar que cada endpoint responde de acuerdo con lo esperado, siguiendo los parámetros definidos en el contrato OpenAPI. Estas pruebas aseguran que la API es funcional, consistente y confiable, lo cual es indispensable para una integración exitosa.

Postman también facilita la creación de colecciones que pueden ser reutilizadas por otros equipos, garantizando la repetibilidad de las pruebas, reduciendo errores humanos y brindando una forma estandarizada de validar la API antes de ser consumida en entornos reales. Al exportar la colección, se proporciona una evidencia clara de que la API cumple con los requisitos funcionales establecidos.

6. ¿Qué mejoras podrían implementarse sobre la solución actual?

Aunque la implementación es funcional y cumple con los requisitos del taller, se identifican varias oportunidades de mejora:

- Incorporar una base de datos real para persistir los envíos.
- Implementar validaciones más estrictas en el modelo `Envio`.
- Integrar un sistema de logs estructurados para facilitar auditoría.
- Añadir pruebas automatizadas con pytest o herramientas similares.

- Manejar errores de forma más robusta utilizando excepciones personalizadas.
- Implementar paginación en el endpoint `GET /envios`.

Estas mejoras permitirían escalar la API hacia entornos empresariales reales y elevar el nivel de robustez de la solución.

5. Conclusiones y Recomendaciones

Conclusiones

- La implementación de la API REST con FastAPI permitió comprender de manera práctica los principios fundamentales de una arquitectura orientada a recursos. La estructura clara de los endpoints, combinada con la validación automática del modelo Envio, facilitó la creación de un servicio consistente y alineado con los estándares REST. Además, la generación automática del contrato OpenAPI evidenció la importancia de mantener una documentación precisa que facilite la interoperabilidad entre sistemas.

- El uso del stack Python + FastAPI demostró ser una alternativa eficiente frente a enfoques tradicionales de integración. La disponibilidad inmediata de Swagger y la simplicidad para ejecutar el servicio permitieron acelerar el proceso de desarrollo, sin sacrificar calidad técnica. Las pruebas realizadas mediante Postman confirmaron la correcta funcionalidad de la API y reforzaron la importancia de validar cada operación para asegurar un comportamiento confiable y reproducible.

Recomendaciones

- Se recomienda evolucionar la API incorporando persistencia mediante una base de datos relacional o NoSQL, con el fin de almacenar los envíos de manera permanente. Asimismo, sería conveniente incluir un sistema robusto de manejo de errores y respuestas estándar, lo cual mejoraría la experiencia de los consumidores del servicio y alinearía la API con prácticas profesionales de desarrollo backend.

- Como mejora adicional, se sugiere integrar herramientas de monitoreo y registros estructurados que permitan evaluar el rendimiento y detectar fallos en tiempo real. También sería valioso complementar el proyecto con pruebas automatizadas para garantizar la estabilidad del servicio ante futuros cambios de alcance o incrementos en la complejidad funcional.