

PRÁCTICA 2

MÉTODOS HTTP: PUT, PATCH, DELETE, GET

Equipo:

Beltrán Saucedo Axel Alejandro

Cerón Samperio Lizeth Montserrat

Higuera Pineda Angel Abraham

Lorenzo Silva Abad Rey

4BV1

ESCUELA SUPERIOR DE
CÓMPUTO

**TECNOLOGÍAS PARA EL DESARROLLO DE APLICACIONES
WEB**

25/09/2025

1. Introducción

Objetivo de la práctica:

El objetivo de la práctica es implementar los métodos HTTP: PUT, PATCH, GET, DELETE en Python con FastAPI, también realizar pruebas de cada uno de los métodos, poniendo a prueba las respuestas que brindará. La aplicación simulará la gestión de paquetes, es decir, que se puede crear un item nuevo con peso y valor, también se le puede poner un ID para ejecutar las funciones de borrar y actualizar parcialmente por el ID.

Métodos HTTP:

- **GET:** Este método se utiliza para solicitar datos de un recurso específico. En la práctica, se implementa para obtener la lista completa de paquetes o un paquete específico por su ID.
- **POST:** Este método se utiliza para enviar datos a un servidor para crear un nuevo recurso. En la práctica, se implementa para crear un nuevo paquete con atributos como peso y valor.
- **PUT:** Este método se utiliza para actualizar completamente un recurso existente. En la práctica, se implementa para actualizar todos los atributos de un paquete específico identificado por su ID.
- **PATCH:** Este método se utiliza para actualizar parcialmente un recurso existente. En la práctica, se implementa para modificar solo algunos atributos de un paquete específico identificado por su ID.
- **DELETE:** Este método se utiliza para eliminar un recurso específico. En la práctica, se implementa para eliminar un paquete identificado por su ID. [\[1\]](#)

2. Fundamentos teoricos

FastAPI:

FastAPI es un framework web para construir APIs en Python, lanzado en diciembre de 2018 por Sebastián Ramírez, un desarrollador colombiano. Está diseñado para ser:

- **Rápido:** comparable en rendimiento a Node.js y Go gracias a su base en ASGI (Asynchronous Server Gateway Interface).
- **Moderno:** aprovecha las anotaciones de tipo de Python 3.6
- **Automático:** genera documentación interactiva con Swagger UI y ReDoc.
- **Seguro y robusto:** ideal para APIs que requieren validación estricta y autentic

Pero, ¿qué hace realmente FastAPI?

Nos permite crear endpoints o rutas para nuestra API, que pueden ser accedidas mediante los diferentes metodos HTTP, como GET, POST, PUT, DELETE, entre otros. Además nos permite:

- Validar automaticamente los datos de entrada y salida utilizando Pydantic.
- Servir modelos de machine learning en producción.
- Integrarse facilmente con base de datos, etc.
- Generar documentación interactiva sin esfuerzo.

Pero, ¿para qué lo usamos realmente?

Usamos FastAPI para crear aplicaciones que se comunican por internet, como por ejemplo:

- Aplicaciones web que mandan y reciben datos (como una app de clima o una tienda en línea).
- Sistemas que conectan con modelos de inteligencia artificial (como chatbots o análisis de imágenes).
- Servicios que otras apps usan para pedir información (como “dame los datos del usuario” o “guárdame esta foto”).

HTTP:

FastAPI utiliza HTTP para comunicarse entre la aplicación web y el cliente. Cada vez que alguien hace una petición (Por ejemplo: GET, POST, etc) a nuestra API, FastAPI recibe esa petición, la procesa y responde con los datos solicitados.

¿Qué es HTTP?

HTTP (Hypertext Transfer Protocol). Es un lenguaje utilizado para realizar una comunicación entre un cliente (como un navegador web o una app móvil) y un servidor (donde vive la aplicación web).

Funciona con peticiones y respuestas, algunas de ellas son:

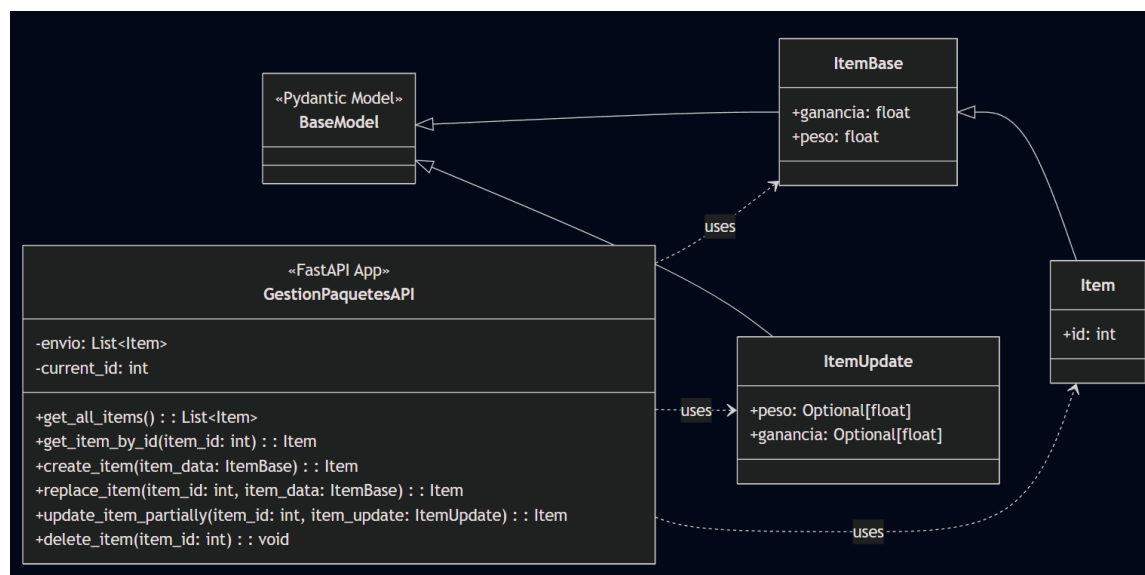
- **GET:** Solicita datos de un recurso específico.
- **POST:** Envía datos para crear un nuevo recurso.
- **PUT:** Actualiza completamente un recurso existente.

- **PATCH:** Actualiza parcialmente un recurso existente.
- **DELETE:** Elimina un recurso específico.

También se cuentan con las HTTP Exceptions, que son respuestas a errores que ocurren cuando algo sale mal en la comunicación entre el cliente y el servidor. Algunos ejemplos son:

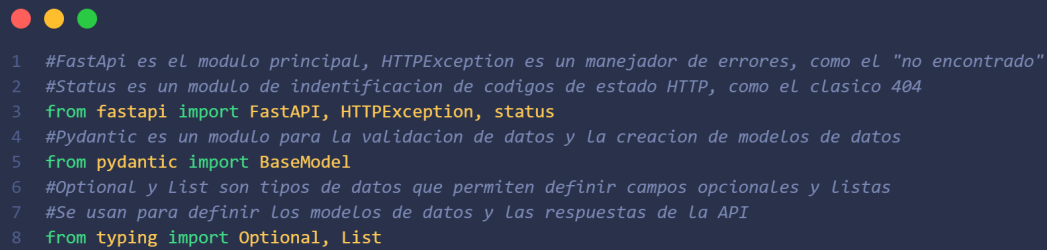
- **404:** No encontrado
- **403:** Prohibido
- **500:** Error interno del servidor
- **400:** Solicitud incorrecta

3. Diagrama UML



4. Implementación

En este apartado se tienen los módulos a utilizar para el funcionamiento de la práctica.



```

1 #FastApi es el modulo principal, HTTPException es un manejador de errores, como el "no encontrado"
2 #Status es un modulo de indentificacion de codigos de estado HTTP, como el clasico 404
3 from fastapi import FastAPI, HTTPException, status
4 #Pydantic es un modulo para la validacion de datos y la creacion de modelos de datos
5 from pydantic import BaseModel
6 #Optional y List son tipos de datos que permiten definir campos opcionales y listas
7 #Se usan para definir los modelos de datos y las respuestas de la API
8 from typing import Optional, List

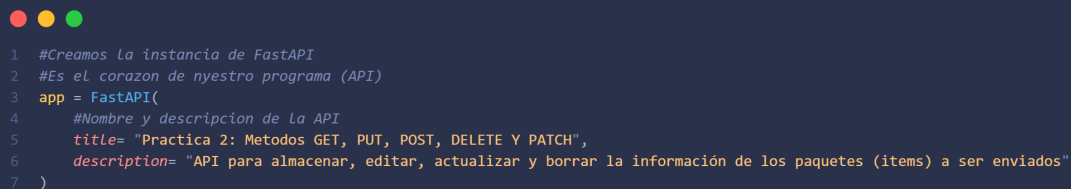
```

FastApi nos permite contruir APIS de manera rapida y sencilla, permitiendo definir rutas y manejar solicitudes HTTP de manera eficiente.

Las rutas que utilizaremos en esta practica son: GET, POST, PUT y DELETE.

Utilizamos Pydantic para validar entradas y para la creacion de un modelo de datos, que en este caso sera para tener la id de un objeto, y su respectivo contenido.

Optional y list son tipos de datos que nos permiten definir atributos que pueden ser opcionales y listas respectivamente.



```

1 #Creamos la instancia de FastAPI
2 #Es el corazon de nuestro programa (API)
3 app = FastAPI(
4     #Nombre y descripcion de la API
5     title= "Practica 2: Metodos GET, PUT, POST, DELETE Y PATCH",
6     description= "API para almacenar, editar, actualizar y borrar la información de los paquetes (items) a ser enviados"
7 )

```

Iniciamos con el corazon de nuestro programa, que es la instancia de FastAPI, en donde definimos el nombre y la descripcion de la API.

```

1  #ItemBase es la estructura base de un item, con los campos ganancia y peso
2  #Digamos que es lo que tiene cada item
3  class ItemBase(BaseModel):
4      ganancia: float
5      peso: float
6  #Item contiene el indentificador unico (id) y hereda los campos de ItemBase
7  class Item(ItemBase):
8      id: int
9  #ItemUpdate es una clase especial utilizada para cuando queremos actualizar un item parcialmente
10 #Los campos son opcionales, ya que podemos querer actualizar solo uno de ellos
11 #El valor por defecto es None
12 class ItemUpdate(BaseModel):
13     peso: Optional[float] = None
14     ganancia: Optional[float] = None

```


Gracias a Pydantic, podemos crear una estructura de datos para nuestros objetos a utilizar. Se tiene un id de tipo entero, este servira para identificar cada objeto. Cda objeto tiene un contenido de datos, en este caso cuenta con ganancia y peso, ambos de tipo flotante. Además se tiene una estructura para actualizar los datos parcialmente, los campos son opcionales, esto es debido a que no es obligatorio actualizar ambos.

```

1  #Creamos una lista llamada envio para almacenar los items
2  envio: List[Item] = []
3  #Variable para tener el control de id que se le asigna a cada item
4  current_id = 0
5

```

Utilizamos variables globales para almacenar los objetos y llevar un control del id. El id se inicializa en 0, y se incrementa cada vez que se crea un nuevo objeto. Los objetos se almacenan en una lista, que inicialmente esta vacia.




```

1  #Get: Permite obtener todos los items cuando se accede a la ruta /items/
2  @app.get("/items/", response_model=list[Item], tags=["Items"])
3  def get_all_items():
4      return envio
5

```

El primer metodo sera GET '/items' el cual permitira obtener todos los items creados que estan en la lista 'envio'.



```

1  #Get: Permite obtener un item por su id
2  @app.get("/items/{item_id}", response_model=Item, tags=["Items"])
3  def get_item_by_id(item_id: int):
4      #Recorremos la lista de items para encontrar el que tiene el id que buscamos
5      item = next((item for item in envio if item.id == item_id), None)
6      #Si no se encuentra, marcamos un error 404 (no encontrado)
7      if item is None:
8          raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Item no encontrado")
9      #Si se encuentra da el item
10     return item

```

Como segundo metodo tenemos otro GET, pero la diferencia es que este permite obtener un item por su ID. Recorrera la lista hasta encontrar el item que coincida con el ID escrito y si lo encuentra lo mostrara. Si no lo encuentra, arrojara el error 404 [No_encontrado].

```

1  #Post: Permite crear un item nuevo
2  #Se devuelve el codigo 201 (creado) si todo sale bien
3  @app.post("/items/", response_model=Item, status_code=status.HTTP_201_CREATED, tags=["Items"])
4  def create_item(item_data: ItemBase):
5      #Accede a la variable global, para poder modificarla e indicar el numero de ids creados
6      global current_id
7      #Se incrementa el id actual en 1
8      current_id += 1
9      #Se crea un nuevo item con el id actual y los datos proporcionados
10     new_item = Item(id=current_id, **item_data.model_dump())
11     #Se agrega el nuevo item a la lista de envio
12     envio.append(new_item)
13     #Da el nuevo item
14     return new_item

```

El tercer metodo es POST('/items') y permite crear nuevos items, accediendo a la variable global para poderla modificar. El ID se ira incrementando en 1 y se creara un nuevo item con el nuevo ID y los datos ingresados. Si no hay problema, el nuevo item se agregara a la lista de envio y arrojará el codigo 201 [creado].

```

1  #Put: Reemplaza un item existente por completo por su id
2  @app.put("/items/{item_id}", response_model=Item, tags=["Items"])
3  def replace_item(item_id: int, item_data: ItemBase):
4      #Recorremos la lista de items para encontrar el que tiene el id que buscamos para reemplazarlo
5      for i, item in enumerate(envio):
6          #Si encontramos el id buscado
7          if item.id == item_id:
8              #Actualiza el item con los nuevos datos
9              #Crear un nuevo objeto con el mismo ID y reemplaza el antiguo
10             updated_item = Item(id=item_id, **item_data.model_dump())
11             envio[i] = updated_item
12             #Da el item actualizado
13             return updated_item
14     #Si no se encuentra, marcamos un error 404 (no encontrado)
15     raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Item no encontrado")

```

Como cuarto metodo esta PUT('/items/') el cual reemplazara un item ya existente por completo. Recorrera la lista hasta encontrar el ID del item buscado y cuando lo encuentre actualizara el item con los nuevos datos. Creara un nuevo objeto con el mismo ID y reemplazara el previo. Si no encuentra el item, volvera a arrojar el error 404.


```

1  #Patch: Actualiza parcialmente un item por su id
2  @app.patch("/items/{item_id}", response_model=Item, tags=["Items"])
3  def update_item_partially(item_id: int, item_update: ItemUpdate):
4      #Busca el item en la lista
5      stored_item = next((item for item in envio if item.id == item_id), None)
6      #Si no se encuentra, marcamos un error 404 (no encontrado)
7      if not stored_item:
8          raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Item no encontrado")
9      #Actualiza solo los campos proporcionados en item_update
10     update_data = item_update.model_dump(exclude_unset=True)
11     #Recorremos los campos a actualizar y los asignamos al item almacenado
12     for key, value in update_data.items():
13         setattr(stored_item, key, value)
14     #Devuelve el item actualizado
15     return stored_item

```

El quinto metodo es PATCH('/items') el cual solo actualizara parcialmente un item por su ID. Funcionara igual que PUT, con la diferencia de que este solo actualizara los campos proporcionados en [item_update].

```

1  #Delete: Borra un item por su id
2  @app.delete("/items/{item_id}", status_code=status.HTTP_204_NO_CONTENT, tags=["Items"])
3  def delete_item(item_id: int):
4      #Recoore la lista para buscar el item a borrar
5      item_to_delete = next((item for item in envio if item.id == item_id), None)
6      #Si no se encuentra, marcamos un error 404 (no encontrado)
7      if not item_to_delete:
8          raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Item no encontrado")
9      #Si se encuentra, lo elimina de la lista
10     envio.remove(item_to_delete)
11     return

```

Como ultimo metodo tenemos DELETE('/items') el cual tendra la tarea de borrar los items. Recorrera la lista hasta encontrar el ID del item que queremos borrar, y si lo encuentra lo eliminara de la lista. Y si no lo encuentra, arrojará error 404.

Referencias

- [1] Métodos de petición HTTP. (s/f). *MDN Web Docs*. Recuperado el 23 de septiembre de 2025, de <https://developer.mozilla.org/es/docs/Web/HTTP/Reference/Methods>
- [2] Ramírez Montaña, S. (s.f.). "Historia, diseño y futuro de FastAPI". *FastAPI*. Recuperado el 23 de septiembre de 2025, de <https://fastapi.tiangolo.com/es/history-design-future/>
- [3] Wikipedia. (2025, julio 11). "FastAPI". *Wikipedia, la enciclopedia libre*. Recuperado el 23 de septiembre de 2025, de <https://es.wikipedia.org/wiki/FastAPI>
- [4] Lubanovic, B. (2019). *Introducing Python: Modern Computing in Simple Packages* (2ª ed.). O'Reilly Media. ISBN: 9781492051367
- [5] Linode Guides & Tutorials. (2021, agosto 6). "Document a FastAPI App with OpenAPI". *Linode*. Recuperado el 23 de septiembre de 2025, de <https://www.linode.com/docs/guides/document-a-fastapi-app-with-openapi/>