

Curso de FastAPI

Diego Saavedra Miguel Amaya

Aug 30, 2024

Table of contents

1	Bienvenido	6
1.1	¿De qué trata este curso?	6
1.2	¿Para quién es este curso?	6
1.3	¿Cómo contribuir?	6
I	Unidad 1: Introducción a Python	8
2	Configuración y Sintaxis básica	9
2.1	Sintaxis básica	10
3	Variables y Control de flujo	12
4	Funciones y Parámetros	15
4.1	Parámetros con valores por defecto	15
4.2	Parámetros con nombre	16
II	Unidad 2: Estructura de Datos en Python	17
5	Listas y Tuplas	18
5.1	Listas	18
5.2	Tuplas	19
6	Diccionarios y Conjuntos	20
6.1	Diccionarios	20
6.2	Conjuntos	21
III	Unidad 3: Programación Orientada a Objetos en Python	22
7	Conceptos Básicos	23
7.1	Clases y Objetos	23
8	Herencia, Polimorfismo y Encapsulación	25
8.1	Herencia	25
8.2	Polimorfismo	26
8.3	Encapsulamiento	27

IV	Unidad 4: Herramientas de Desarrollo	29
9	Git y Github	30
9.1	Git	30
9.1.1	Instalación de Git	30
9.1.2	Comandos básicos de Git	31
9.2	Github	31
9.2.1	Crear una cuenta en Github	32
9.2.2	Crear un repositorio en Github	32
9.2.3	Clonar un repositorio en Github	32
9.2.4	Subir cambios a un repositorio en Github	33
9.2.5	Descargar cambios de un repositorio en Github	33
V	Unidad 5: Introducción a FastAPI	34
10	Configuración y Estructura	35
10.1	Instalación de FastAPI	35
10.2	Estructura de un proyecto FastAPI	36
10.3	Ejecución de un proyecto FastAPI	38
10.4	Documentación de una API FastAPI	38
11	Pydantic en FastAPI	41
11.1	¿Qué es Pydantic?	41
12	Otro Ejemplo	44
13	Modelos en FastAPI	49
13.1	Definición de modelos en FastAPI	49
13.2	Uso de modelos en FastAPI	50
14	Rutas y Validaciones en FastAPI	52
14.1	Definición de rutas en FastAPI	52
14.2	Creación de las Rutas	53
14.3	Uso de las Rutas en FastAPI	54
14.4	Probar las Rutas en FastAPI	54
14.5	Validaciones en FastAPI	60
15	APIs RESTful con FastAPI	62
15.1	Creación de una API RESTful	62
15.2	Creación de las Rutas	63
15.3	Uso de las Rutas en FastAPI	64
16	Probar las Rutas en FastAPI	66
16.1	Crear una Adopción	66
16.2	Obtener las Adopciones	67
16.3	Obtener una Adopción	67
16.4	Crear un Animal	68
16.5	Obtener los Animales	68
16.6	Obtener un Animal	69

16.7 Crear una Persona	69
16.8 Obtener las Personas	70
16.9 Obtener una Persona	71
17 Pruebas Unitarias en FastAPI	72
17.1 Pruebas Unitarias	72
17.2 Pruebas Unitarias en FastAPI	72
18 Optimización y Rendimiento	76
18.1 Caché	76
19 Creación de una API de gestión de tareas utilizando FastAPI	82
19.1 Creación de una API de gestión de tareas	82
19.2 Creación de las Rutas	83
19.3 Ejecución de la API	84
VI Unidad 6: Consumo API con FastAPI	87
20 Configuración y Uso de FastAPI para consumir APIs	88
20.1 Creación de un entorno virtual	88
20.2 Consumo de la API	89
20.3 Ejecución de la API	90
20.4 Pruebas de la API	91
21 Integración de APIs de terceros con FastAPI	94
VII Unidad 7: Desarrollo Avanzado	95
22 Manejo de Estado en FastAPI, Context API	96
23 Navegación en FastAPI con FastRouter	97
24 Aplicación en FastAPI que consume APIs de terceros	98
VIII Unidad 8: Prácticas Avanzadas de Programación	99
25 Patrones de Diseño en FastAPI	100
26 Arquitectura de Software y Diseño Modular	101
IX Proyecto Final	102

X	Laboratorios	106
30	Taller Práctico: Implementando una CNN para Reconocimiento de Imágenes (20 minutos)	107
30.1	Objetivo	107
30.2	Requisitos Previos	107
30.3	Materiales	107
30.4	Pasos del Taller	107
31	Cargar el Conjunto de Datos CIFAR-10	109
32	Reto	114
33	Laboratorio de CNN con FastAPI	115
33.1	1. Cargar el modelo creado en el laboratorio anterior	115
33.2	2. Ejecutar la API	116
33.3	3. Probar la API	116

1 Bienvenido

¡Bienvenido al Curso Completo de FastAPI!

En este curso, exploraremos todo, desde los fundamentos hasta las aplicaciones prácticas.

1.1 ¿De qué trata este curso?

Este curso completo me llevará desde los fundamentos básicos de la programación hasta la construcción de aplicaciones prácticas utilizando los frameworks Django y la biblioteca de React.

A través de una combinación de teoría y ejercicios prácticos, me sumergiré en los conceptos esenciales del desarrollo web y avanzaré hacia la creación de proyectos del mundo real.

Desde la configuración del entorno de desarrollo hasta la construcción de una aplicación web de pila completa, este curso me proporcionará una comprensión sólida y experiencia práctica con FastAPI.

1.2 ¿Para quién es este curso?

Este curso está diseñado para principiantes y aquellos con poca o ninguna experiencia en programación.

Ya sea que sea un estudiante curioso, un profesional que busca cambiar de carrera o simplemente alguien que quiere aprender desarrollo web, este curso es para usted. Desde adolescentes hasta adultos, todos son bienvenidos a participar y explorar el emocionante mundo del desarrollo web con FastAPI.

1.3 ¿Cómo contribuir?

Valoramos su contribución a este curso. Si encuentra algún error, desea sugerir mejoras o agregar contenido adicional, me encantaría saber de usted.

Puede contribuir a través del repositorio en línea, donde puede compartir sus comentarios y sugerencias.

Juntos, podemos mejorar continuamente este recurso educativo para beneficiar a la comunidad de estudiantes y entusiastas de la programación.

Este ebook ha sido creado con el objetivo de proporcionar acceso gratuito y universal al conocimiento.

Estará disponible en línea para cualquier persona, sin importar su ubicación o circunstancias, para acceder y aprender a su propio ritmo.

Puede descargarlo en formato PDF, Epub o verlo en línea en cualquier momento y lugar.

Esperamos que disfrute este emocionante viaje de aprendizaje y descubrimiento en el mundo del desarrollo web con FastAPI!

Part I

Unidad 1: Introducción a Python

2 Configuración y Sintaxis básica

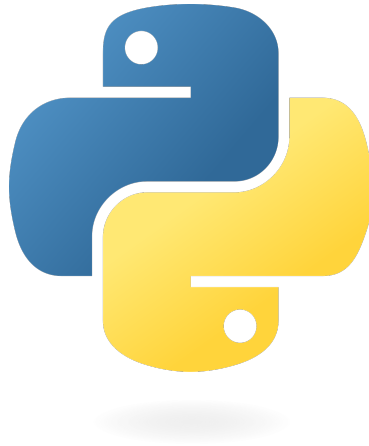


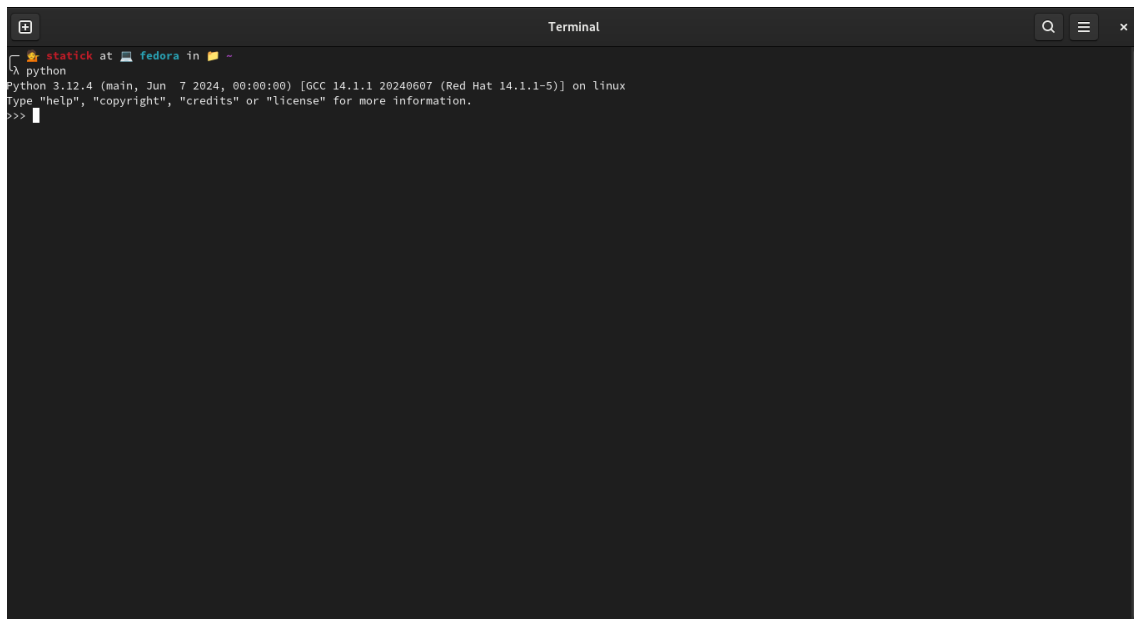
Figure 2.1: Python

Para empezar a trabajar con python es necesario tener instalado el interprete de python en tu computadora. Para ello puedes descargarlo desde la página oficial de python <https://www.python.org/>.

Una vez instalado, puedes abrir una terminal y escribir **python** para abrir el interprete de python. Si ves un mensaje similar a este:

```
Python 3.12.4 (main, Jun 7 2024, 00:00:00) [GCC 14.1.1 20240607 (Red Hat 14.1.1-5)] on 1
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Significa que python está instalado correctamente en tu computadora.

A terminal window titled "Terminal" with a dark background. The prompt shows the user is static at a fedora machine. The user has entered the command 'python', which has started Python 3.12.4. The startup screen displays the version, date, time, GCC version, and OS (Linux). It also shows the copyright and license information. The prompt is now '>>>' with a cursor.

Para salir del interprete de python puedes escribir `exit()` o **Ctrl + D**.

2.1 Sintaxis básica

Python es un lenguaje de programación interpretado, lo que significa que el código se ejecuta línea por línea. A continuación se muestra un ejemplo de un programa simple en python:

```
# Este es un comentario  
print("Hola Mundo!")
```

Para ejecutar este programa, puedes guardar el código en un archivo con extensión **.py** y ejecutarlo desde la terminal con el comando **python nombre_del_archivo.py**.

The screenshot shows the Visual Studio Code editor with a file named `hola_mundo.py` open. The file contains the following code:

```
1 # Este es un comentario
2 print("Hola Mundo!")
```

The terminal at the bottom shows the command `python hola_mundo.py` being executed, resulting in the output `Hola Mundo!`.

En este caso, el programa imprimirá en la terminal el mensaje **Hola Mundo!**.

En este primer capítulo de la unidad, aprendimos la configuración básica de python y la sintaxis básica para escribir programas en python.

3 Variables y Control de flujo

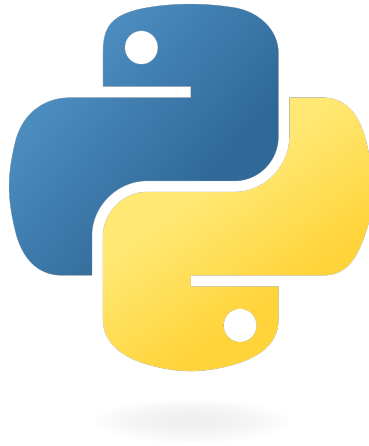


Figure 3.1: Python

En python las variables se pueden declarar sin necesidad de especificar el tipo de dato, por lo que se puede asignar cualquier tipo de dato a una variable, sin embargo en FastAPI es necesario especificar el tipo de dato de las variables.

```
# Declaración de variables  
a = 5  
b = 3.14  
c = "Hola Mundo"
```

Para imprimir el valor de una variable se utiliza la función **print()**.

```
print(a)  
print(b)  
print(c)
```

Para especificar el tipo de dato de una variable se utiliza la siguiente sintaxis:

```
# Declaración de variables con tipo de dato  
a: int = 5  
b: float = 3.14  
c: str = "Hola Mundo"
```

Para realizar operaciones aritméticas se utilizan los siguientes operadores:

- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- Módulo: %
- Exponente: **
- División entera: //

```
# Operaciones aritméticas
suma = a + b
resta = a - b
multiplicacion = a * b
division = a / b
modulo = a % b
exponente = a ** b
division_entera = a // b

print(suma)
print(resta)
print(multiplicacion)
print(division)
print(modulo)
print(exponente)
print(division_entera)
```

Para realizar comparaciones se utilizan los siguientes operadores:

- Igual que: ==
- Diferente de: !=
- Mayor que: >
- Menor que: <
- Mayor o igual que: >=
- Menor o igual que: <=

```
# Comparaciones
igual = a == b
diferente =
```

Para realizar operaciones lógicas se utilizan los siguientes operadores:

- AND: and
- OR: or
- NOT: not

```
# Operaciones lógicas
and = True and False
or = True or False
not = not True
```

Para realizar estructuras de control de flujo se utilizan las siguientes estructuras:

- if
- elif
- else

```
# Estructuras de control de flujo
if a > b:
    print("a es mayor que b")
elif a < b:
    print("a es menor que b")
else:
    print("a es igual a b")
```

Tambien existen otras estructuras de control de flujo como:

- for
- while

```
# Estructuras de control de flujo
for i in range(5):
    print(i)

i = 0
while i < 5:
    print(i)
    i += 1
```

En FastAPI se pueden declarar variables en las rutas y se pueden especificar el tipo de dato de las variables.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

En el ejemplo anterior se declara una ruta con una variable llamada **item_id** de tipo entero, cuando analicemos FastAPI en sus primeros capítulos se explicará con más detalle.

En este capítulo de la unidad, aprendimos a declarar variables, realizar operaciones aritméticas, comparaciones, operaciones lógicas y estructuras de control de flujo en python.

4 Funciones y Parámetros

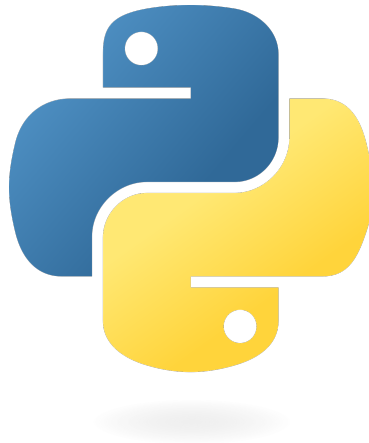


Figure 4.1: Python

En Python las funciones se definen con la palabra clave **def** seguida del nombre de la función y los parámetros entre paréntesis. A continuación se muestra un ejemplo de una función que suma dos números:

```
def saludo():  
    return "Hola Mundo!"  
  
mensaje = saludo()  
print(mensaje)
```

En el ejemplo anterior, la función **saludo()** retorna el mensaje **Hola Mundo!**.

4.1 Parámetros con valores por defecto

En Python es posible asignar valores por defecto a los parámetros de una función. A continuación se muestra un ejemplo de una función que suma dos números con un valor por defecto para el segundo parámetro:

```
def suma(a, b=0):  
    return a + b  
  
resultado = suma(5)  
print(resultado)
```

En el ejemplo anterior, la función **suma()** recibe dos parámetros, el primer parámetro es obligatorio y el segundo parámetro tiene un valor por defecto de **0**.

4.2 Parámetros con nombre

En Python es posible pasar los parámetros de una función por nombre. A continuación se muestra un ejemplo de una función que multiplica dos números con los parámetros pasados por nombre:

```
def multiplicacion(a, b):  
    return a * b  
  
resultado = multiplicacion(b=5, a=3)  
print(resultado)
```

En el ejemplo anterior, los parámetros de la función **multiplicacion()** se pasan por nombre, por lo que el orden de los parámetros no importa.

Tip

Cuando aprendamos acerca de la POO (Programación Orientada a Objetos) veremos que las **funciones** que se definen dentro de una clase se llaman **métodos**.

En FastAPI es posible definir funciones que se ejecutan cuando se realiza una petición HTTP a una ruta específica. Cuando analicemos FastAPI veremos cómo definir funciones que se ejecutan cuando se realiza una petición HTTP a una ruta específica.

En este tercer capítulo de la unidad, aprendimos a definir funciones en Python y a utilizar parámetros con valores por defecto y parámetros con nombre.

Part II

Unidad 2: Estructura de Datos en Python

5 Listas y Tuplas

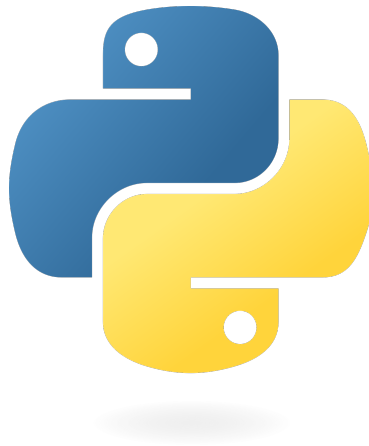


Figure 5.1: Python

En este capítulo de la segunda unidad vamos a aprender acerca de las listas y las tuplas en Python.

En Python, las listas y las tuplas son estructuras de datos que permiten almacenar múltiples elementos en una sola variable.

5.1 Listas

Las listas en Python se definen utilizando corchetes [] y los elementos de la lista se separan por comas ,. A continuación se muestra un ejemplo de una lista con números enteros:

```
# Declaración de una lista
numeros = [1, 2, 3, 4, 5]
```

Para acceder a un elemento de la lista se utiliza el índice del elemento. Los índices en Python empiezan en 0. A continuación se muestra un ejemplo de cómo acceder al primer elemento de la lista:

```
# Acceso a un elemento de la lista
primer_elemento = numeros[0]
print(primer_elemento)
```

Para agregar un elemento a la lista se utiliza el método **append()**. A continuación se muestra un ejemplo de cómo agregar un elemento a la lista:

```
# Agregar un elemento a la lista
numeros.append(6)
print(numeros)
```

Para eliminar un elemento de la lista se utiliza el método **remove()**. A continuación se muestra un ejemplo de cómo eliminar un elemento de la lista:

```
# Eliminar un elemento de la lista
numeros.remove(3)
print(numeros)
```

5.2 Tuplas

Las tuplas en Python se definen utilizando paréntesis () y los elementos de la tupla se separan por comas ,. A continuación se muestra un ejemplo de una tupla con números enteros:

```
# Declaración de una tupla
numeros = (1, 2, 3, 4, 5)
```

Para acceder a un elemento de la tupla se utiliza el índice del elemento. Los índices en Python empiezan en 0. A continuación se muestra un ejemplo de cómo acceder al primer elemento de la tupla:

```
# Acceso a un elemento de la tupla
primer_elemento = numeros[0]
print(primer_elemento)
```

Las tuplas son inmutables, lo que significa que una vez que se crea una tupla no se pueden modificar los elementos de la tupla. A continuación se muestra un ejemplo de cómo intentar modificar un elemento de la tupla:

```
# Intentar modificar un elemento de la tupla
numeros[0] = 10
```

En FastAPI es posible utilizar listas y tuplas para definir los parámetros de una función. Cuando analicemos FastAPI veremos cómo utilizar listas y tuplas para definir los parámetros de una función.

En este capítulo de la unidad, aprendimos acerca de las listas y las tuplas en Python.

6 Dictionarios y Conjuntos

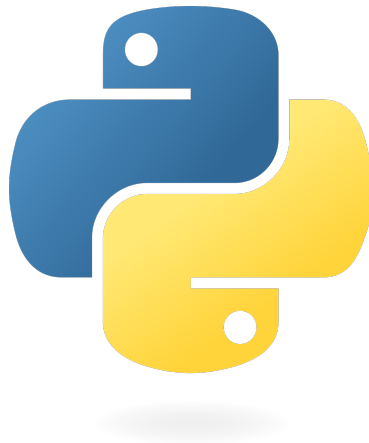


Figure 6.1: Python

En este capítulo vamos a aprender acerca de los diccionarios y los conjuntos en Python.

6.1 Dictionarios

Los diccionarios en Python son estructuras de datos que permiten almacenar pares clave-valor en una sola variable. Los diccionarios se definen utilizando llaves `{ }` y los pares clave-valor se separan por comas `,`. A continuación se muestra un ejemplo de un diccionario con nombres de personas y sus edades:

```
# Declaración de un diccionario

personas = {
    "Juan": 25,
    "Maria": 30,
    "Pedro": 35
}
```

Para acceder a un valor del diccionario se utiliza la clave del valor. A continuación se muestra un ejemplo de cómo acceder a la edad de la persona **Juan**:

```
# Acceso a un valor del diccionario

edad_juan = personas["Juan"]
print(edad_juan)
```

Para agregar un par clave-valor al diccionario se utiliza la siguiente sintaxis:

```
# Agregar un par clave-valor al diccionario

personas["Ana"] = 40
print(personas)
```

Para eliminar un par clave-valor del diccionario se utiliza la siguiente sintaxis:

```
# Eliminar un par clave-valor del diccionario

del personas["Pedro"]
print(personas)
```

6.2 Conjuntos

Los conjuntos en Python son estructuras de datos que permiten almacenar elementos únicos en una sola variable. Los conjuntos se definen utilizando llaves `{ }` y los elementos del conjunto se separan por comas `,`. A continuación se muestra un ejemplo de un conjunto con números enteros:

```
# Declaración de un conjunto

numeros = {1, 2, 3, 4, 5}
```

Para agregar un elemento al conjunto se utiliza el método **add()**. A continuación se muestra un ejemplo de cómo agregar un elemento al conjunto:

```
# Agregar un elemento al conjunto

numeros.add(6)
print(numeros)
```

Para eliminar un elemento del conjunto se utiliza el método **remove()**. A continuación se muestra un ejemplo de cómo eliminar un elemento del conjunto:

```
# Eliminar un elemento del conjunto

numeros.remove(3)
print(numeros)
```

En FastAPI es posible utilizar diccionarios y conjuntos para almacenar información y realizar operaciones con ellos. Cuando analicemos FastAPI veremos cómo utilizar diccionarios y conjuntos para almacenar información y realizar operaciones con ellos.

En este capítulo aprendimos acerca de los diccionarios y los conjuntos en Python.

Part III

Unidad 3: Programación Orientada a Objetos en Python

7 Conceptos Básicos

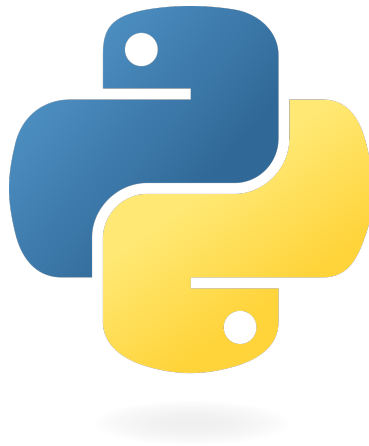


Figure 7.1: Python

En este capítulo vamos a aprender acerca de los conceptos básicos de la Programación Orientada a Objetos (POO) en Python.

7.1 Clases y Objetos

En la Programación Orientada a Objetos (POO) los objetos son instancias de clases. Las clases son plantillas que definen las propiedades y los métodos de los objetos. A continuación se muestra un ejemplo de una clase en Python:

```
# Declaración de una clase

class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        return f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años."
```

En el ejemplo anterior se declara una clase llamada **Persona** con dos propiedades **nombre** y **edad** y un método **saludar()** que retorna un mensaje con el nombre y la edad de la persona.

Para crear un objeto de la clase **Persona** se utiliza la siguiente sintaxis:

```
# Creación de un objeto de la clase Persona
```

```
persona = Persona("Juan", 25)  
mensaje = persona.saludar()  
print(mensaje)
```

En el ejemplo anterior se crea un objeto de la clase **Persona** con el nombre **Juan** y la edad **25** y se llama al método **saludar()** del objeto **persona**.

En FastAPI es posible definir clases que se utilizan para definir los modelos de los datos que se envían y reciben en las peticiones HTTP. Cuando analicemos FastAPI veremos cómo definir clases que se utilizan para definir los modelos de los datos que se envían y reciben en las peticiones HTTP.

En este capítulo de la unidad, aprendimos acerca de las clases y los objetos en la Programación Orientada a Objetos (POO) en Python.

8 Herencia, Polimorfismo y Encapsulación

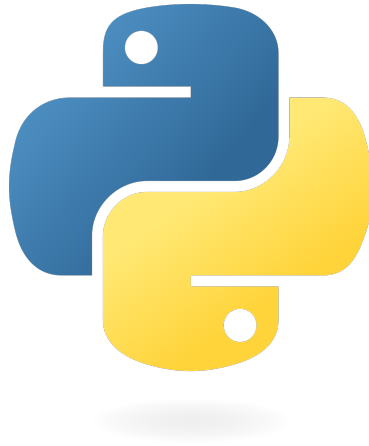


Figure 8.1: Python

En este capítulo vamos a aprender acerca de la herencia y el polimorfismo en la Programación Orientada a Objetos (POO) en Python.

8.1 Herencia

La herencia en la Programación Orientada a Objetos (POO) es un mecanismo que permite crear una nueva clase a partir de una clase existente. La nueva clase hereda las propiedades y los métodos de la clase existente. A continuación se muestra un ejemplo de una clase **Vehículo** con las propiedades **marca** y **modelo** y un método **mostrar()** que retorna un mensaje con la marca y el modelo del vehículo:

```
# Declaración de una clase Vehículo

class Vehículo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def mostrar(self):
        return f"Vehículo: {self.marca} {self.modelo}"
```

En el ejemplo anterior se declara una clase **Vehículo** con las propiedades **marca** y **modelo** y un método **mostrar()** que retorna un mensaje con la marca y el modelo del vehículo.

Para crear una nueva clase **Auto** que hereda de la clase **Vehiculo** se utiliza la siguiente sintaxis:

```
# Declaración de una clase Auto que hereda de la clase Vehiculo

class Auto(Vehiculo):
    def __init__(self, marca, modelo, color):
        super().__init__(marca, modelo)
        self.color = color

    def mostrar(self):
        return f"Auto: {self.marca} {self.modelo} de color {self.color}"
```

En el ejemplo anterior se declara una clase **Auto** que hereda de la clase **Vehiculo** con la propiedad **color** y un método **mostrar()** que retorna un mensaje con la marca, el modelo y el color del auto.

Para crear un objeto de la clase **Auto** se utiliza la siguiente sintaxis:

```
# Creación de un objeto de la clase Auto

auto = Auto("Toyota", "Corolla", "Rojo")
mensaje = auto.mostrar()
print(mensaje)
```

En el ejemplo anterior se crea un objeto de la clase **Auto** con la marca **Toyota**, el modelo **Corolla** y el color **Rojo** y se llama al método **mostrar()** del objeto **auto**.

8.2 Polimorfismo

El polimorfismo en la Programación Orientada a Objetos (POO) es un mecanismo que permite que un objeto se com

```
# Declaración de una clase Vehiculo

class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def mostrar(self):
        return f"Vehículo: {self.marca} {self.modelo}"
```

En el ejemplo anterior se declara una clase **Vehiculo** con las propiedades **marca** y **modelo** y un método **mostrar()** que retorna un mensaje con la marca y el modelo del vehículo.

Para crear una nueva clase **Auto** que hereda de la clase **Vehiculo** se utiliza la siguiente sintaxis:

```
# Declaración de una clase Auto que hereda de la clase Vehiculo

class Auto(Vehiculo):
    def __init__(self, marca, modelo, color):
        super().__init__(marca, modelo)
        self.color = color

    def mostrar(self):
        return f"Auto: {self.marca} {self.modelo} de color {self.color}"
```

En el ejemplo anterior se declara una clase **Auto** que hereda de la clase **Vehiculo** con la propiedad **color** y un método **mostrar()** que retorna un mensaje con la marca, el modelo y el color del auto.

Para crear un objeto de la clase **Auto** se utiliza la siguiente sintaxis:

```
# Creación de un objeto de la clase Auto

auto = Auto("Toyota", "Corolla", "Rojo")
mensaje = auto.mostrar()
print(mensaje)
```

En el ejemplo anterior se crea un objeto de la clase **Auto** con la marca **Toyota**, el modelo **Corolla** y el color **Rojo** y se llama al método **mostrar()** del objeto **auto**.

8.3 Encapsulamiento

El encapsulamiento en la Programación Orientada a Objetos (POO) es un mecanismo que permite ocultar los detalles de implementación de una clase y exponer solo la interfaz de la clase. En Python, el encapsulamiento se logra utilizando los siguientes modificadores de acceso:

- **Público:** Los miembros de la clase son públicos y se pueden acceder desde cualquier parte del programa.
- **Protegido:** Los miembros de la clase son protegidos y se pueden acceder desde la clase y las clases derivadas.
- **Privado:** Los miembros de la clase son privados y solo se pueden acceder desde la clase.

A continuación se muestra un ejemplo de una clase **Persona** con los modificadores de acceso **público**, **protegido** y **privado**:

```
# Declaración de una clase Persona

class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self._edad = edad
        self.__dni = "12345678"

    def mostrar(self):
        return f"Persona: {self.nombre} {self._edad} {self.__dni}"

# Creación de un objeto de la clase Persona

persona = Persona("Juan", 25)
mensaje = persona.mostrar()
print(mensaje)
```

En el ejemplo anterior se declara una clase **Persona** con los modificadores de acceso **público**, **protegido** y **privado** y se crea un objeto de la clase **Persona** con el nombre **Juan**, la edad **25** y el DNI **12345678**.

En FastAPI es posible definir clases que se utilizan para definir los modelos de los datos que se envían y reciben en las peticiones HTTP. Cuando analicemos FastAPI veremos cómo definir clases que se utilizan para definir los modelos de los datos que se envían y reciben en las peticiones HTTP.

En este capítulo de la unidad, aprendimos acerca de la herencia, polimorfismo y encapsulación en la Programación Orientada a Objetos (POO) en Python.

Part IV

Unidad 4: Herramientas de Desarrollo

9 Git y Github



En este capítulo un poco diferente vamos a aprender acerca de **Git** y **Github**. Sin embargo lo aplicaremos con el lenguaje de programación **Python**.

9.1 Git



Figure 9.1: Git

Git es un sistema de control de versiones distribuido que permite llevar un registro de los cambios en los archivos de un proyecto. **Git** es ampliamente utilizado en el desarrollo de software para colaborar en proyectos con otros desarrolladores.

9.1.1 Instalación de Git

Para instalar **Git** en tu computadora, sigue los siguientes pasos:

1. Descarga el instalador de **Git** desde el siguiente enlace: <https://git-scm.com/>.
2. Ejecuta el instalador de **Git** y sigue las instrucciones del instalador.
3. Verifica que **Git** se ha instalado correctamente ejecutando el siguiente comando en la terminal:

```
git --version
```

Si **Git** se ha instalado correctamente, verás un mensaje similar a este:

```
git version 2.45.2
```

9.1.2 Comandos básicos de Git

A continuación se muestran algunos comandos básicos de **Git** que te serán útiles para trabajar con **Git**:

- **git init**: Inicializa un repositorio de **Git** en el directorio actual.

```
git init
```

- **git add**: Agrega los archivos al área de preparación.

```
git add archivo.py
```

- **git commit**: Guarda los cambios en el repositorio.

```
git commit -m "Mensaje del commit"
```

- **git status**: Muestra el estado de los archivos en el repositorio.

```
git status
```

- **git log**: Muestra el historial de los commits en el repositorio.

```
git log
```

9.2 Github



Figure 9.2: Github

Github es una plataforma en línea que permite alojar proyectos de **Git** de forma gratuita. **Github** es ampliamente utilizado en el desarrollo de software para colaborar en proyectos con otros desarrolladores.

9.2.1 Crear una cuenta en Github

Para crear una cuenta en **Github**, sigue los siguientes pasos:

1. Ingresa a la página de **Github**: <https://github.com>.
2. Haz clic en el botón **Sign up**.
3. Completa el formulario de registro con tu nombre de usuario, dirección de correo electrónico y contraseña.
4. Haz clic en el botón **Create account**.
5. Verifica tu dirección de correo electrónico.

9.2.2 Crear un repositorio en Github

Para crear un repositorio en **Github**, sigue los siguientes pasos:

1. Inicia sesión en tu cuenta de **Github**.
2. Haz clic en el botón **New**.
3. Completa el formulario con el nombre del repositorio, la descripción y la visibilidad del repositorio.
4. Haz clic en el botón **Create repository**.
5. Copia la URL del repositorio.

9.2.3 Clonar un repositorio en Github

Para clonar un repositorio en **Github**, sigue los siguientes pasos:

1. Copia la URL del repositorio.
2. Abre la terminal y ejecuta el siguiente comando:

```
git clone URL_del_repositorio
```

3. Ingresa tus credenciales de **Github**.
4. El repositorio se clonará en tu computadora.

9.2.4 Subir cambios a un repositorio en Github

Para subir cambios a un repositorio en **Github**, sigue los siguientes pasos:

1. Agrega los archivos al área de preparación.

```
git add archivo.py
```

2. Guarda los cambios en el repositorio.

```
git commit -m "Mensaje del commit"
```

3. Sube los cambios al repositorio en **Github**.

```
git push
```

4. Ingresa tus credenciales de **Github**.
5. Los cambios se subirán al repositorio en **Github**.

9.2.5 Descargar cambios de un repositorio en Github

Para descargar cambios de un repositorio en **Github**, sigue los siguientes pasos:

1. Descarga los cambios del repositorio en **Github**.

```
git pull
```

2. Ingresa tus credenciales de **Github**.
3. Los cambios se descargarán del repositorio en **Github**.

En FastAPI es posible utilizar **Git** y **Github** para colaborar en proyectos con otros desarrolladores. Cuando analicemos FastAPI veremos cómo utilizar **Git** y **Github** para colaborar en proyectos con otros desarrolladores.

En este capítulo de la unidad, aprendimos acerca de **Git** y **Github**.

Part V

Unidad 5: Introducción a FastAPI

10 Configuración y Estructura



Figure 10.1: FastAPI

Despues de conocer algunos conceptos fundamentales de Python, es momento de conocer FastAPI, un framework web moderno y rápido para crear APIs con Python 3.6+ basado en estándares abiertos y estándares de tipo Python (PEP 484).

10.1 Instalación de FastAPI

Antes de realizar la instalación de FastAPI es muy recomendable que en cualquier proyecto de python se cree un entorno virtual, para ello se puede utilizar la herramienta **virtualenv** que permite crear entornos virtuales de python. O de forma nativa con el módulo **venv** que viene incluido en la instalación de python.

Para crear un entorno virtual con venv se utiliza el siguiente comando:

```
python -m venv nombre_entorno
```

Para activar el entorno virtual se utiliza el siguiente comando:

```
source nombre_entorno/bin/activate
```

En el caso de sistemas operativos Windows se utiliza el siguiente comando:

```
nombre_entorno\Scripts\activate
```

Para desactivar el entorno virtual se utiliza el siguiente comando:

```
deactivate
```

Para instalar FastAPI se utiliza el siguiente comando:

```
pip install fastapi
```

Para instalar el servidor web **uvicorn** se utiliza el siguiente comando:

```
pip install uvicorn
```

10.2 Estructura de un proyecto FastAPI

A continuación se muestra la estructura de un proyecto FastAPI:

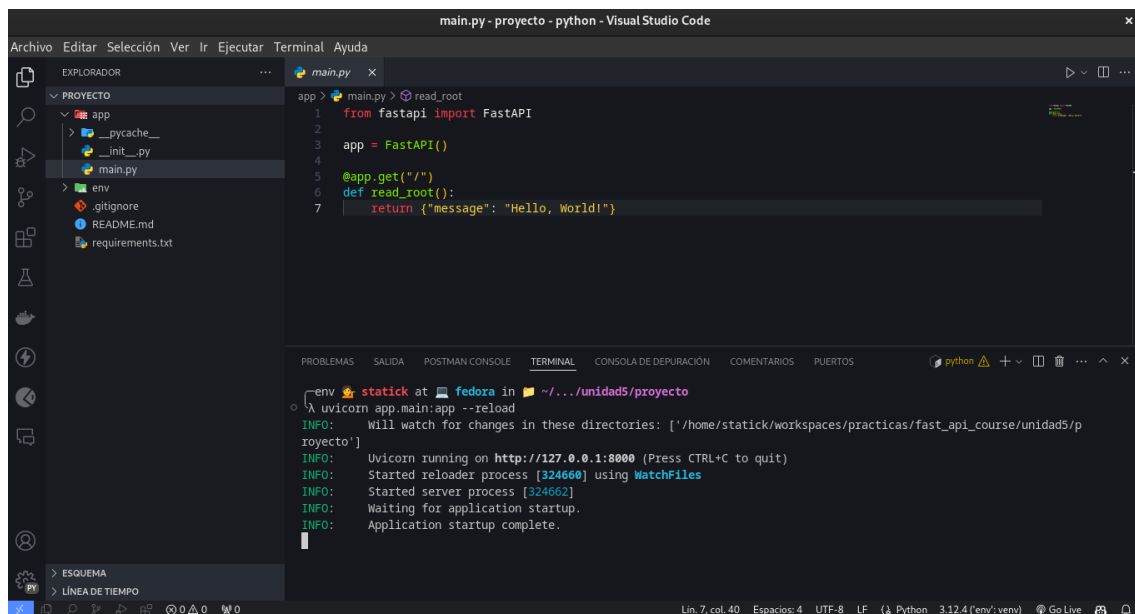
```
proyecto/  
  
  app/  
    __init__.py  
    main.py  
  
  .gitignore  
  README.md  
  requirements.txt
```

- **app/**: Directorio que contiene el código fuente de la aplicación.
 - **__init__.py**: Archivo que indica que el directorio es un paquete de Python.
 - **main.py**: Archivo principal de la aplicación que contiene la lógica de la API.
- **.gitignore**: Archivo que contiene los archivos y directorios que se deben ignorar en el control de versiones.
- **README.md**: Archivo que contiene la documentación del proyecto.
- **requirements.txt**: Archivo que contiene las dependencias del proyecto.

En el directorio **app/** se encuentra el archivo **main.py** que contiene la lógica de la API. En el archivo **main.py** se definen las rutas de la API y las operaciones que se realizan en cada ruta.

En el archivo **main.py** se importan las clases **FastAPI** y **Request** de la librería **fastapi** y se crea una instancia de la clase **FastAPI** que representa la aplicación. A continuación se definen las rutas de la API utilizando la instancia de la clase **FastAPI** y se definen las operaciones que se realizan en cada ruta.

En el archivo **main.py** se define una ruta de la API utilizando el decorador (**app.get?**)() y se define una operación que retorna un mensaje de bienvenida. A continuación se muestra un ejemplo de un archivo **main.py** con una ruta de la API y una operación que retorna un mensaje de bienvenida:



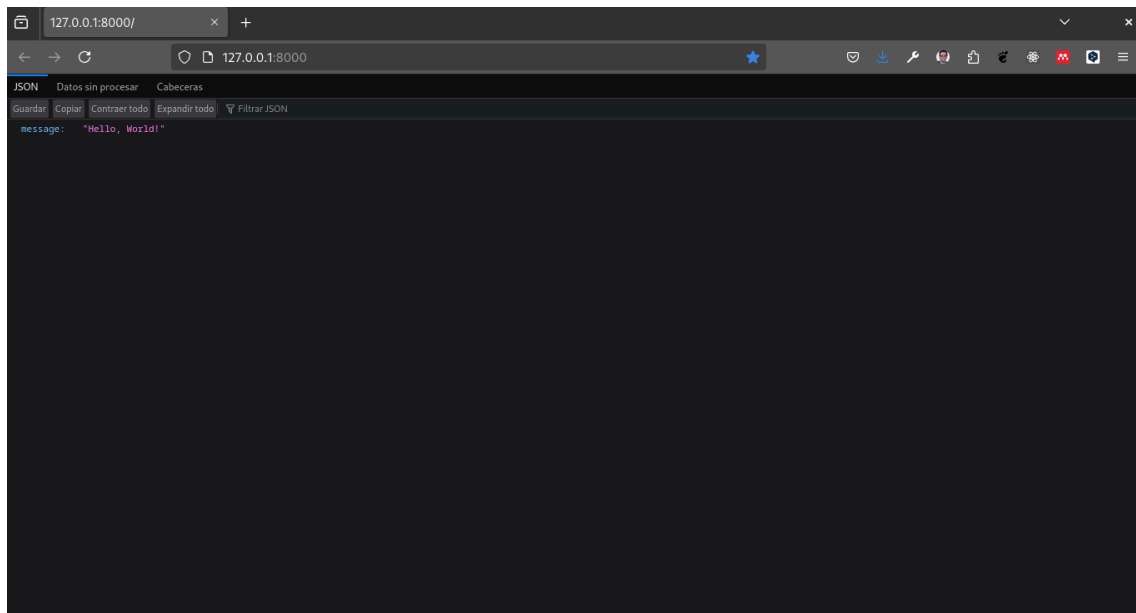
```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello, World!"}
```

En el ejemplo anterior se importa la clase **FastAPI** de la librería **fastapi** y se crea una instancia de la clase **FastAPI** llamada **app**. A continuación se define una ruta de la API utilizando el decorador (**app.get?**)() y se define una operación llamada **read_root()** que retorna un mensaje de bienvenida.

10.3 Ejecución de un proyecto FastAPI



Para ejecutar un proyecto FastAPI se utiliza el siguiente comando:

```
uvicorn app.main:app --reload
```

En el comando anterior se utiliza el comando **uvicorn** para ejecutar el servidor web y se especifica el archivo **main.py** que contiene la lógica de la API y la instancia de la clase **FastAPI** llamada **app**. El parámetro **--reload** indica que el servidor web se reinicia automáticamente cuando se realizan cambios en el código fuente.

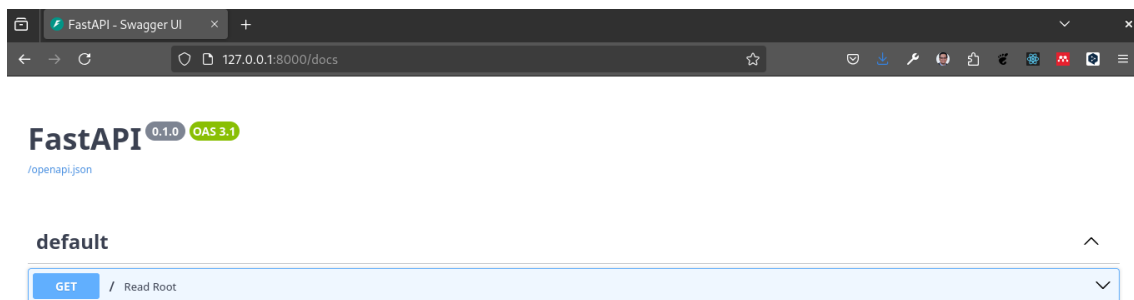
Al ejecutar el comando anterior se inicia el servidor web y se muestra la URL de la API en la consola. Para acceder a la API se utiliza la URL que se muestra en la consola.

10.4 Documentación de una API FastAPI

FastAPI proporciona una interfaz de usuario interactiva que permite visualizar y probar la API. Para acceder a la interfaz de usuario se utiliza la URL de la API seguida de **/docs**. Por ejemplo, si la URL de la API es **http://127.0.0.1:8000**, la URL de la interfaz de usuario es **http://127.0.0.1:8000/docs**.

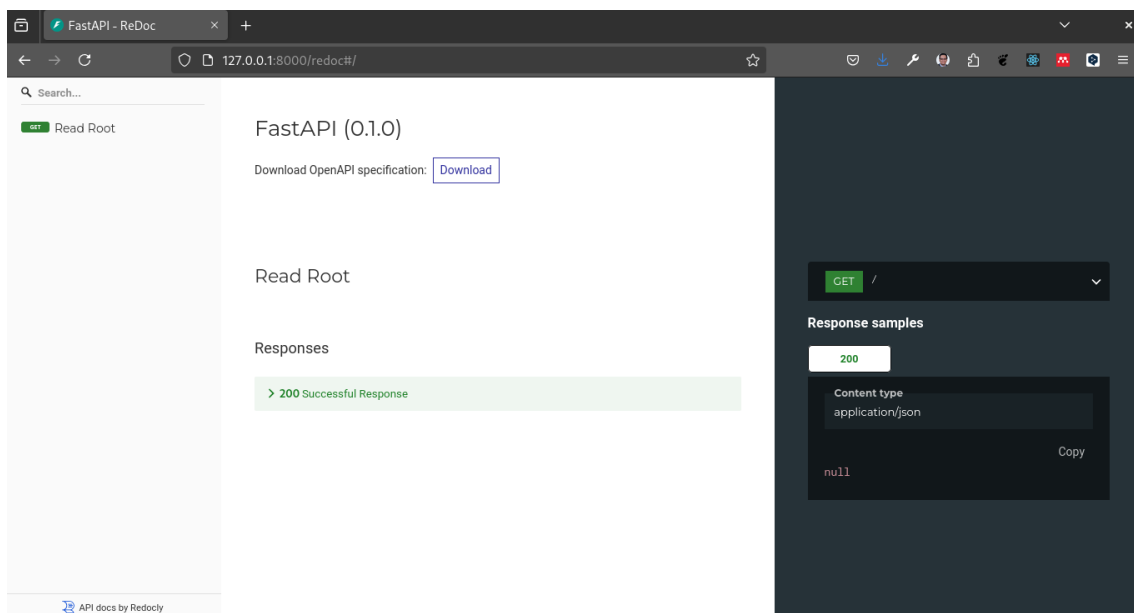
En la interfaz de usuario se muestra una lista de las rutas de la API y las operaciones que se realizan en cada ruta. Para probar una operación se hace clic en la operación y se ingresan los parámetros de la operación.

A continuación se muestra un ejemplo de la interfaz de usuario de FastAPI:



En la interfaz de usuario se muestra una lista de las rutas de la API y las operaciones que se realizan en cada ruta.

FastAPI cuenta con una segunda opción de documentación llamada **/redoc** que es una interfaz de usuario alternativa que permite visualizar y probar la API. Para acceder a la interfaz de usuario **redoc** se utiliza la URL de la API seguida de **/redoc**. Por ejemplo, si la URL de la API es **http://127.0.0.1:8000**, la URL de la interfaz de usuario **redoc** es **http://127.0.0.1:8000/redoc**.



En la interfaz de usuario **redoc** se muestra una lista de las rutas de la API y las operaciones que se realizan en cada ruta.

En este capítulo se ha mostrado la instalación de FastAPI, la estructura de un proyecto FastAPI, la ejecución de un proyecto FastAPI y la documentación de una API FastAPI.

En los siguientes capítulos se mostrará cómo definir rutas y operaciones en FastAPI, cómo validar datos en FastAPI y cómo trabajar con bases de datos en FastAPI.

11 Pydantic en FastAPI

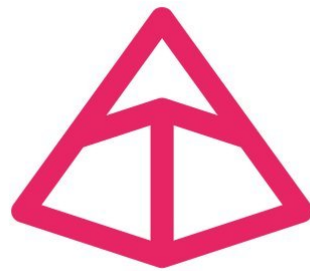


Figure 11.1: Pydantic

11.1 ¿Qué es Pydantic?

Pydantic es una librería de Python que permite definir esquemas de datos y validarlos. Pydantic se utiliza en FastAPI para definir los modelos de datos que se utilizan en la API y validar los datos que se reciben en las solicitudes.

En FastAPI se utiliza Pydantic para definir los modelos de datos que se utilizan en la API. Pydantic es una librería que permite definir esquemas de datos y validarlos.

La estructura del proyecto de esta unidad es la siguiente:

```
proyecto/  
  
  app/  
    __init__.py  
    main.py  
  
  .gitignore  
  README.md  
  requirements.txt
```

A continuación se muestra un ejemplo de cómo definir un modelo de datos con Pydantic:

Es necesario instalar Pydantic para poder utilizarlo en FastAPI. Para instalar Pydantic se utiliza el siguiente comando:

```
pip install pydantic
```

Sin olvidar nuestro framework FastAPI:

```
pip install fastapi uvicorn
```

Ahora se puede definir un modelo de datos con Pydantic. A continuación se muestra un ejemplo de cómo definir un modelo de datos con Pydantic:

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = None
```

En el ejemplo anterior se define un modelo de datos **Item** que contiene tres campos: **name**, **price** e **is_offer**. El campo **name** es de tipo **str**, el campo **price** es de tipo **float** y el campo **is_offer** es de tipo **bool** con un valor por defecto de **None**.

Para utilizar el modelo de datos **Item** en una operación de la API se importa la clase **Item** y se utiliza como tipo de parámetro en la operación. A continuación se muestra un ejemplo de cómo utilizar el modelo de datos **Item** en una operación de la API:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = None

@app.post("/items/")
async def create_item(item: Item):
    return {"name": item.name, "price": item.price}
```

En el ejemplo anterior se importa la clase **Item** y se define una operación **create_item()** que recibe un parámetro **item** de tipo **Item**. En la operación se retorna un diccionario con los campos **name** y **price** del objeto **item**.

Es necesario probar nuestro código, para ello se ejecuta el servidor web con el siguiente comando:

```
uvicorn app.main:app --reload
```

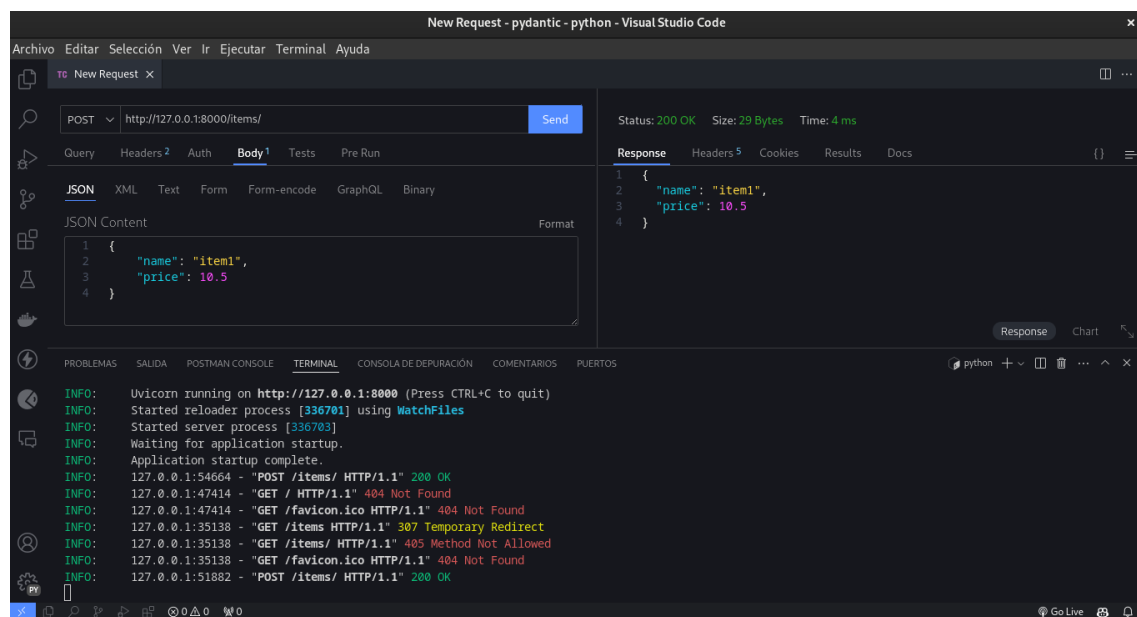
Para probar la operación `create_item()` se puede utilizar una herramienta como **Tunder Client** o **Postman**. A continuación se muestra un ejemplo de cómo probar la operación `create_item()` con **Tunder Client**:

Para realizar una solicitud **POST** a la ruta `/items/` con un objeto **item** se utiliza la siguiente solicitud:

```
POST /items/
Content-Type: application/json

{
  "name": "item1",
  "price": 10.5
}
```

En la solicitud anterior se envía un objeto **item** con los campos **name** y **price**. La operación `create_item()` recibe el objeto **item** y retorna un diccionario con los campos **name** y **price** del objeto **item**.



En este ejemplo se ha utilizado Pydantic para definir un modelo de datos **Item** y validar los datos que se reciben en la solicitud. Pydantic permite definir esquemas de datos y validarlos, lo que facilita la creación de APIs con FastAPI.

Para comprobar que todo funciona correctamente, se puede probar la operación `create_item()` con **Tunder Client** o **Postman** y verificar que se retorna un diccionario con los campos **name** y **price** del objeto **item**.

12 Otro Ejemplo

En este nuevo ejemplo se va a definir un modelo de datos **User** con Pydantic y se va a utilizar un CRUD para la generación de la API. A continuación se muestra el código del ejemplo:

La estructura del proyecto de esta unidad es la siguiente:

```
proyecto/  
  
  app/  
    __init__.py  
    main.py  
  
  .gitignore  
  README.md  
  requirements.txt
```

El código de la API (main.py) es el siguiente:

```
from fastapi import FastAPI  
from pydantic import BaseModel  
  
app = FastAPI()  
  
class User(BaseModel):  
    id: int  
    name: str  
    email: str  
  
users = []  
  
@app.post("/users/")  
async def create_user(user: User):  
    users.append(user)  
    return user  
  
@app.get("/users/")  
async def read_users():  
    return users  
  
@app.get("/users/{user_id}")
```

```

async def read_user(user_id: int):
    for user in users:
        if user.id == user_id:
            return user
    return {"message": "User not found"}

@app.put("/users/{user_id}")
async def update_user(user_id: int, user: User):
    for i, u in enumerate(users):
        if u.id == user_id:
            users[i] = user
            return user
    return {"message": "User not found"}

@app.delete("/users/{user_id}")
async def delete_user(user_id: int):
    for i, user in enumerate(users):
        if user.id == user_id:
            del users[i]
            return {"message": "User deleted"}
    return {"message": "User not found"}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="localhost", port=8000)

```

En el ejemplo anterior se define un modelo de datos **User** con tres campos: **id**, **name** y **email**. Se define una lista **users** para almacenar los usuarios y se definen las operaciones **create_user()**, **read_users()**, **read_user()**, **update_user()** y **delete_user()** para realizar las operaciones CRUD sobre los usuarios.

Para probar el ejemplo se ejecuta el servidor web con el siguiente comando:

```
uvicorn app.main:app --reload
```

Para probar las operaciones CRUD sobre los usuarios se puede utilizar una herramienta como **Tunder Client** o **Postman**. A continuación se muestra un ejemplo de cómo probar las operaciones CRUD sobre los usuarios con **Tunder Client**:

Para crear un usuario se realiza una solicitud **POST** a la ruta **/users/** con un objeto **user**:

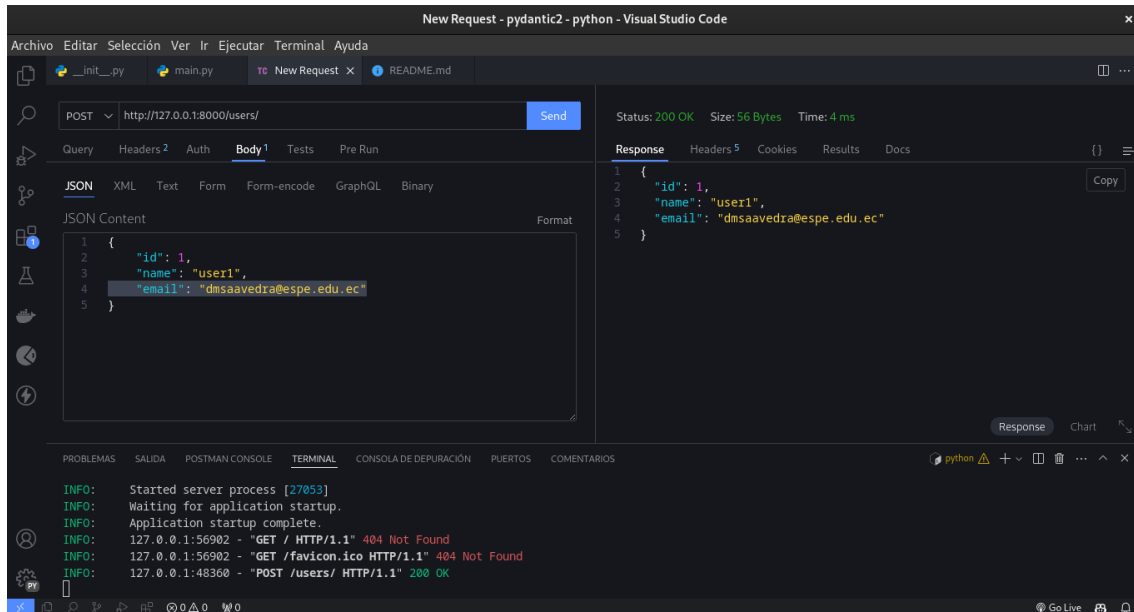
```

POST /users/
Content-Type: application/json

{
    "id": 1,
    "name": "user1",

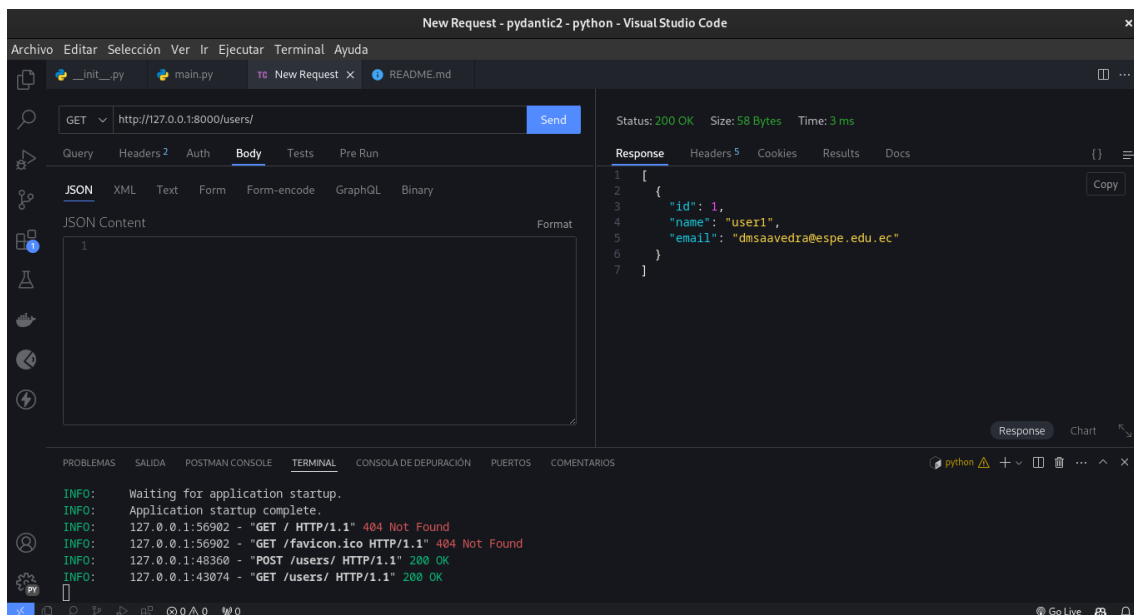
```

```
"email": "dmsaavedra@espe.edu.ec"
}
```



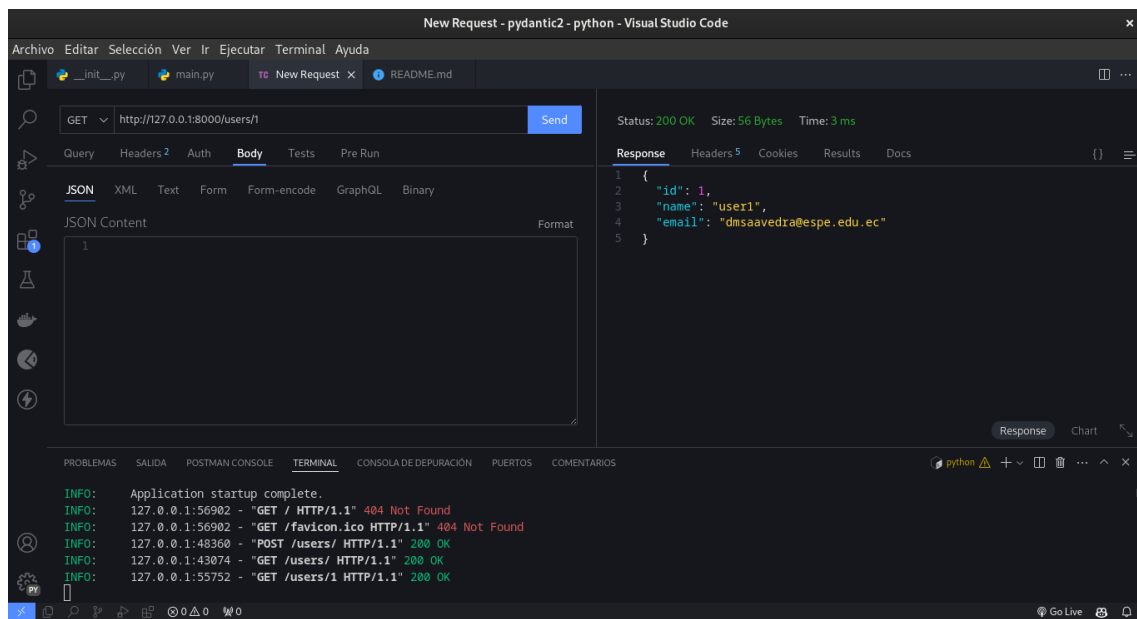
Para obtener todos los usuarios se realiza una solicitud **GET** a la ruta `/users/`:

```
GET /users/
```



Para obtener un usuario se realiza una solicitud **GET** a la ruta `/users/{user_id}` con el **id** del usuario:

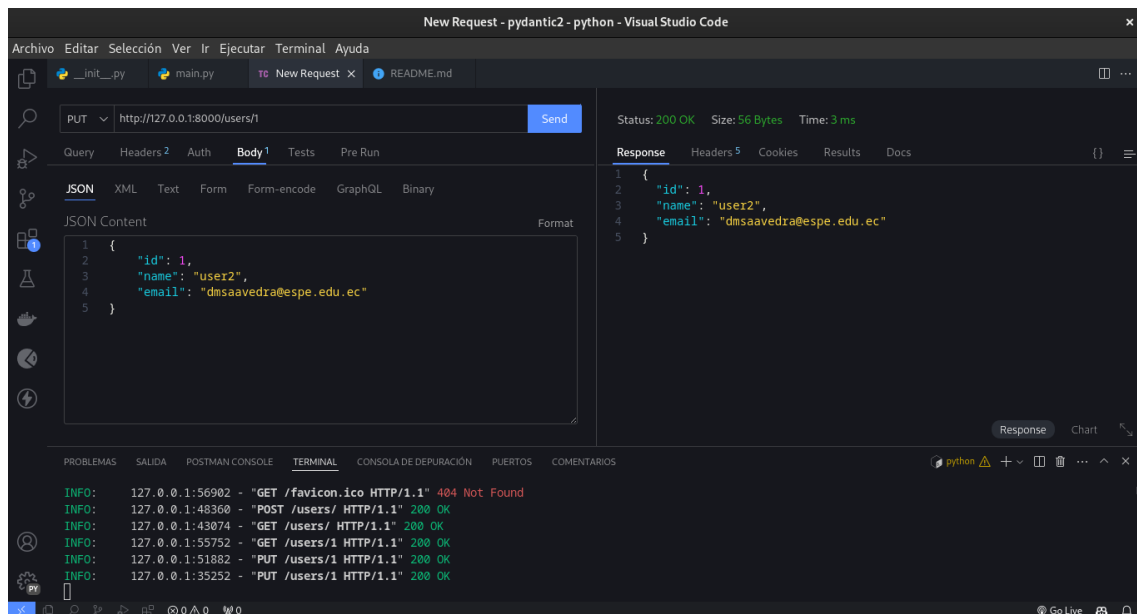
```
GET /users/1
```



Para actualizar un usuario se realiza una solicitud **PUT** a la ruta `/users/{user_id}` con el **id** del usuario y un objeto **user**:

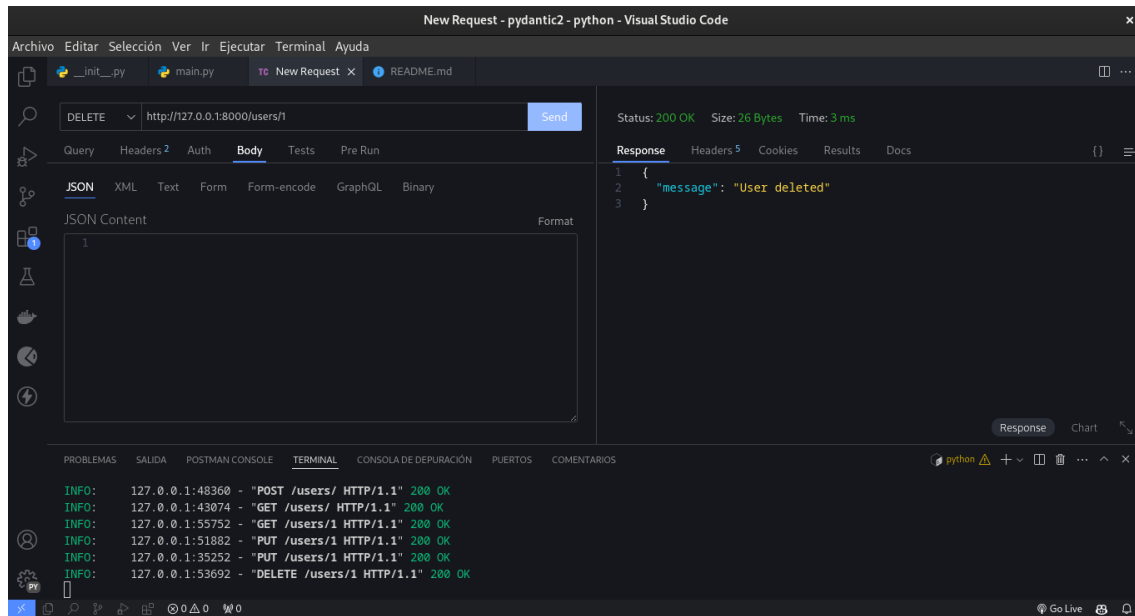
```
PUT /users/1
Content-Type: application/json

{
  "id": 1,
  "name": "user2",
  "email": "dmsaavedra@espe.edu.ec"
}
```



Para eliminar un usuario se realiza una solicitud **DELETE** a la ruta `/users/{user_id}` con el **id** del usuario:

DELETE `/users/1`



En este ejemplo se ha utilizado Pydantic para definir un modelo de datos **User** y validar los datos que se reciben en las solicitudes. Pydantic permite definir esquemas de datos y validarlos, lo que facilita la creación de APIs con FastAPI.

En este capítulo se ha mostrado cómo utilizar Pydantic en FastAPI para definir modelos de datos y validar los datos que se reciben en las solicitudes. Pydantic es una librería de Python que permite definir esquemas de datos y validarlos, lo que facilita la creación de APIs con FastAPI.

13 Modelos en FastAPI



Figure 13.1: FastAPI

En el capítulo anterior aprendimos acerca de Pydantic y cómo se puede utilizar para validar y serializar datos en Python. En este capítulo veremos cómo utilizar Pydantic para definir modelos en FastAPI.

13.1 Definición de modelos en FastAPI

En FastAPI se pueden definir modelos utilizando Pydantic. Un modelo en FastAPI es una clase que hereda de la clase **BaseModel** de Pydantic y define los campos que componen el modelo. A continuación se muestra un ejemplo de un modelo en FastAPI, para este ejemplo vamos a crear una tienda de vehículos:

El proyecto tendrá la siguiente estructura:

```
proyecto/  
  
  app/  
    __init__.py  
    main.py  
    models.py  
  
  .gitignore  
  README.md  
  requirements.txt
```

En el directorio **app/** se encuentra el archivo **models.py** que contiene la definición del modelo **Vehicle**. A continuación se muestra el contenido del archivo **models.py**:

```
from pydantic import BaseModel  
  
class Vehicle(BaseModel):  
    make: str
```

```
model: str
year: int
price: float
```

En el ejemplo anterior se define un modelo llamado **Vehicle** que hereda de la clase **BaseModel** de Pydantic. El modelo **Vehicle** tiene los siguientes campos:

- **make:** Campo de tipo **str** que representa la marca del vehículo.
- **model:** Campo de tipo **str** que representa el modelo del vehículo.
- **year:** Campo de tipo **int** que representa el año del vehículo.
- **price:** Campo de tipo **float** que representa el precio del vehículo.

13.2 Uso de modelos en FastAPI

Una vez que se ha definido un modelo en FastAPI, se puede utilizar el modelo en las rutas de la API para validar y serializar datos. A continuación se muestra un ejemplo de cómo utilizar el modelo **Vehicle** en una ruta de la API:

```
from fastapi import FastAPI
from .models import Vehicle

app = FastAPI()

@app.post("/vehicles/")
def create_vehicle(vehicle: Vehicle):
    return vehicle

@app.get("/vehicles/{vehicle_id}")
def read_vehicle(vehicle_id: int):
    return {"vehicle_id": vehicle_id}

@app.put("/vehicles/{vehicle_id}")
def update_vehicle(vehicle_id: int, vehicle: Vehicle):
    return {"vehicle_id": vehicle_id, "vehicle": vehicle}

@app.delete("/vehicles/{vehicle_id}")
def delete_vehicle(vehicle_id: int):
    return {"vehicle_id": vehicle_id}
```

En el ejemplo anterior se importa la clase **Vehicle** del archivo **models.py** y se define una ruta de la API que recibe un objeto de tipo **Vehicle** en el cuerpo de la petición. La ruta de la API utiliza el decorador `@app.post()` para indicar que se trata de una petición de tipo POST. La operación `create_vehicle()` recibe un objeto de tipo **Vehicle** y retorna el mismo objeto.

En el ejemplo anterior también se definen otras rutas de la API que utilizan el modelo **Vehicle** para validar y serializar datos. En la ruta de la API que recibe un parámetro de

tipo **int** se utiliza el modelo **Vehicle** para serializar el objeto de tipo **Vehicle** y retornar el objeto serializado.

En este capítulo aprendimos acerca de los modelos en FastAPI y cómo se pueden utilizar para validar y serializar datos en una API. En el próximo capítulo veremos cómo utilizar rutas y validaciones en FastAPI.

14 Rutas y Validaciones en FastAPI

En el capítulo anterior aprendimos acerca de la creación de modelos en FastAPI utilizando Pydantic. En este capítulo veremos cómo definir rutas y validaciones en FastAPI.

14.1 Definición de rutas en FastAPI

En FastAPI se pueden definir rutas utilizando decoradores. **Un decorador es una función que toma otra función y extiende su funcionalidad sin modificarla.** En FastAPI, los decoradores se utilizan para **definir rutas en la API**. A continuación se muestra un ejemplo de un Sistema de Inventario para aprender cómo definir una ruta en FastAPI:

El proyecto tendrá la siguiente estructura:

```
proyecto/

  app/
    __init__.py
    main.py
  |  |  models.py
  |  |  routes.py

  .gitignore
  README.md
  requirements.txt
```

En el directorio **app/** se encuentra el archivo **routes.py** que contiene la definición de las rutas de la API. Vamos a crear los modelos, las rutas y las validaciones para un Sistema de Inventario. A continuación se muestra el contenido del archivo **models**:

```
from pydantic import BaseModel

class Item(BaseModel):
    id: int
    name: str
    description: str
    price: float
    tax: float
```

En el ejemplo anterior se define un modelo llamado **Item** que hereda de la clase **BaseModel** de Pydantic. El modelo **Item** tiene los siguientes campos:

- **id**: Campo de tipo **int** que representa el identificador del artículo.
- **name**: Campo de tipo **str** que representa el nombre del artículo.
- **description**: Campo de tipo **str** que representa la descripción del artículo.
- **price**: Campo de tipo **float** que representa el precio del artículo.
- **tax**: Campo de tipo **float** que representa el impuesto del artículo.

14.2 Creación de las Rutas

En este proyecto también crearemos un CRUD para el modelo **Item**. A continuación se muestra el contenido del archivo **routes.py** que contiene la definición de las rutas de la API:

```
from fastapi import APIRouter, HTTPException
from .models import Item

router = APIRouter()

inventory = []

@router.post("/items/")
def create_item(item: Item):
    inventory.append(item)
    return item

@router.get("/items/")
def read_items():
    return inventory

@router.get("/items/{item_id}")
def read_item(item_id: int):
    for item in inventory:
        if item.id == item_id:
            return item
    raise HTTPException(status_code=404, detail="Item not found")

@router.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
    for i in range(len(inventory)):
        if inventory[i].id == item_id:
            inventory[i] = item
            return item
    raise HTTPException(status_code=404, detail="Item not found")

@router.delete("/items/{item_id}")
def delete_item(item_id: int):
    for i in range(len(inventory)):
        if inventory[i].id == item_id:
```

```
        item = inventory.pop(i)
        return item
    raise HTTPException(status_code=404, detail="Item not found")
```

En el ejemplo anterior se definen las siguientes rutas:

- **POST** `/items/`: Ruta para crear un nuevo artículo en el inventario.
- **GET** `/items/`: Ruta para obtener todos los artículos del inventario.
- **GET** `/items/{item_id}`: Ruta para obtener un artículo específico del inventario.
- **PUT** `/items/{item_id}`: Ruta para actualizar un artículo específico del inventario.
- **DELETE** `/items/{item_id}`: Ruta para eliminar un artículo específico del inventario.

14.3 Uso de las Rutas en FastAPI

Para utilizar las rutas definidas en FastAPI, se deben importar las rutas en el archivo principal de la aplicación. A continuación se muestra un ejemplo de cómo importar las rutas en el archivo `main.py`:

```
from fastapi import FastAPI
from .routes import router

app = FastAPI()

app.include_router(router)
```

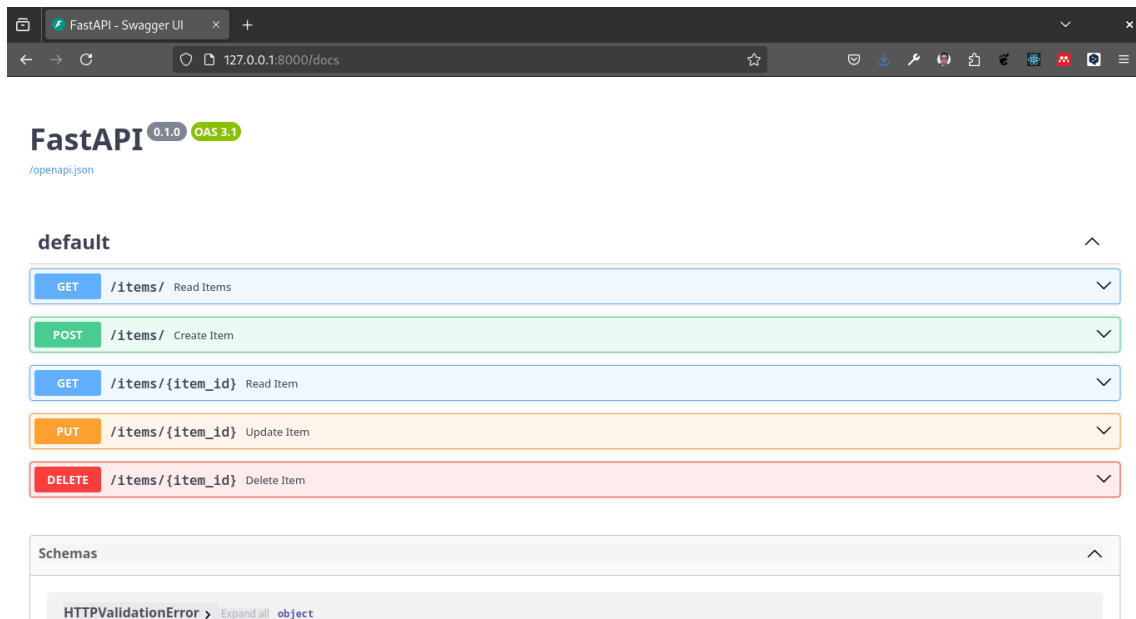
En el ejemplo anterior se importa el objeto **router** que contiene las rutas definidas en el archivo `routes.py`. Luego, se utiliza el método `include_router()` para incluir las rutas en la aplicación FastAPI.

14.4 Probar las Rutas en FastAPI

Para ejecutar la aplicación, se debe utilizar el comando **uvicorn** en la terminal.

```
uvicorn app.main:app --reload
```

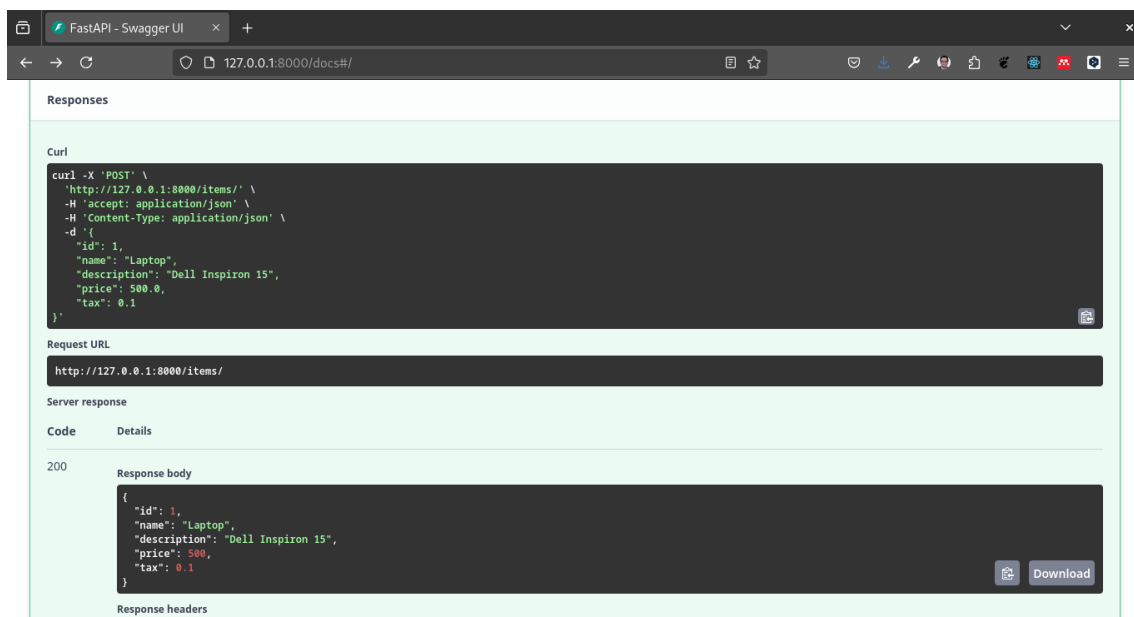
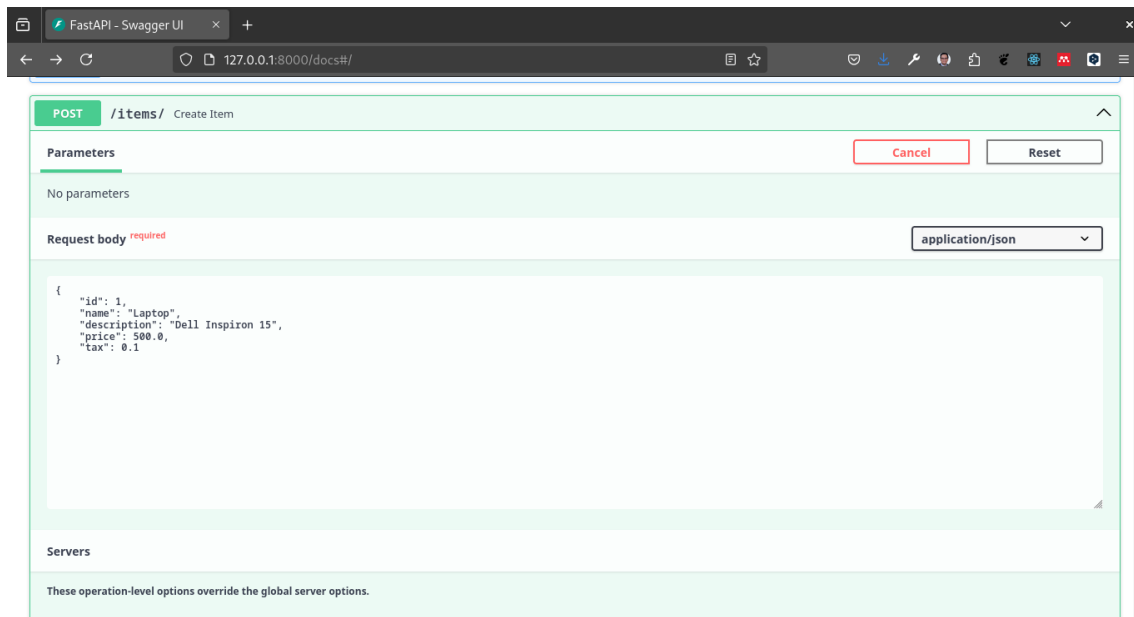
Para probar las rutas definidas en FastAPI, se puede utilizar la interfaz de usuario de Swagger que se genera automáticamente al ejecutar la aplicación. Para acceder a la interfaz de usuario de Swagger, se debe abrir un navegador web y visitar la URL <http://localhost:8000/docs>.



En la interfaz de usuario de Swagger se pueden probar las rutas de la API enviando peticiones HTTP y visualizando las respuestas. Por ejemplo, se puede probar la ruta **POST** `/items/` para crear un nuevo artículo en el inventario enviando un objeto JSON con los datos del artículo.

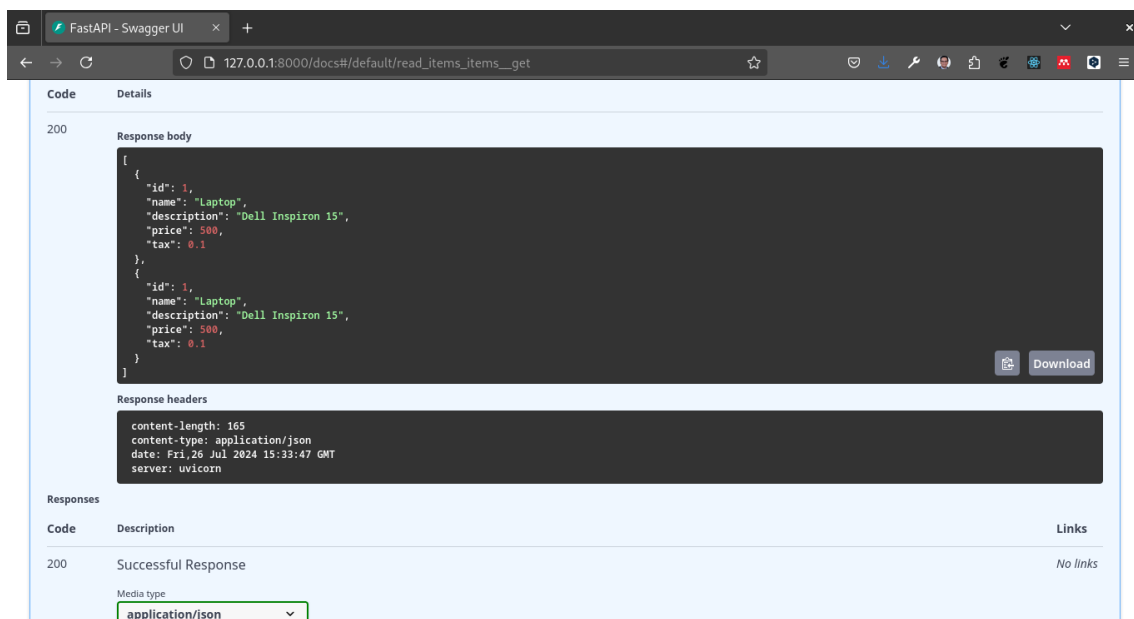
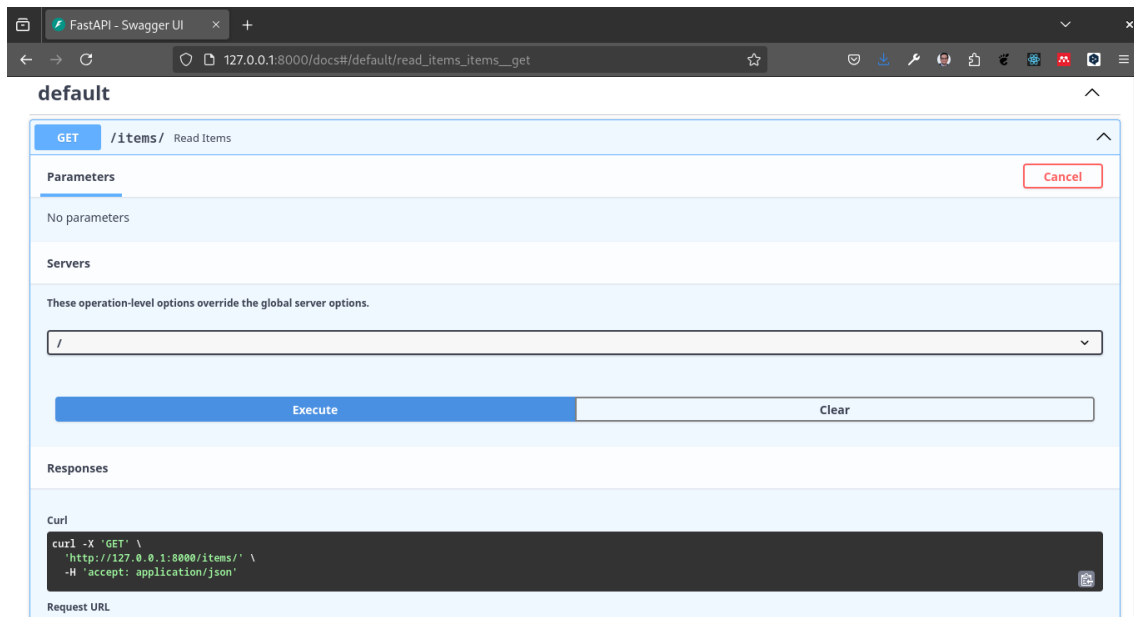
Ejemplo:

```
{
  "id": 1,
  "name": "Laptop",
  "description": "Dell Inspiron 15",
  "price": 500.0,
  "tax": 0.1
}
```



En el ejemplo anterior se envía una petición POST a la ruta **/items/** con un objeto JSON que representa un artículo. La API responde con el mismo objeto JSON que se envió en la petición. También podemos probar las demás rutas de la API.

Para probar la ruta **GET /items/** se puede enviar una petición GET a la ruta **/items/** para obtener todos los artículos del inventario.



Para probar la ruta **GET /items/{item_id}** se puede enviar una petición GET a la ruta **/items/{item_id}** para obtener un artículo específico del inventario.

The top screenshot shows the Swagger UI for the endpoint `GET /items/{item_id}` (Read Item). The parameter `item_id` is required and is an integer (path). The value `1` is entered in the input field. The `Execute` button is visible.

The bottom screenshot shows the response for the `GET /items/1` endpoint. The response is a JSON object representing a laptop item:

```
{
  "id": 1,
  "name": "Laptop",
  "description": "Dell Inspiron 15",
  "price": 500,
  "tax": 0.1
}
```

The response headers are also displayed:

```
content-length: 81
content-type: application/json
date: Fri, 26 Jul 2024 15:34:41 GMT
server: uvicorn
```

Para probar la ruta **PUT** `/items/{item_id}` se puede enviar una petición PUT a la ruta `/items/{item_id}` con un objeto JSON que representa un artículo para actualizar un artículo específico del inventario.

Actualizaremos body por el siguiente JSON

```
{
  "id": 1,
  "name": "Laptop",
  "description": "Lenovo Thinkpad",
  "price": 350,
  "tax": 0.1
}
```

FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/update_item_items__item_id__put

PUT /items/{item_id} Update Item

Parameters

Cancel Reset

Name	Description
item_id * required integer (path)	<input type="text" value="1"/>

Request body required

application/json

```
{
  "id": 1,
  "name": "Laptop",
  "description": "Lenovo Thinkpad",
  "price": 350,
  "tax": 0.1
}
```

FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/update_item_items__item_id__put

Responses

Curl

```
curl -X 'PUT' \
  'http://127.0.0.1:8000/items/1' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 1,
    "name": "Laptop",
    "description": "Lenovo Thinkpad",
    "price": 350,
    "tax": 0.1
  }'
```

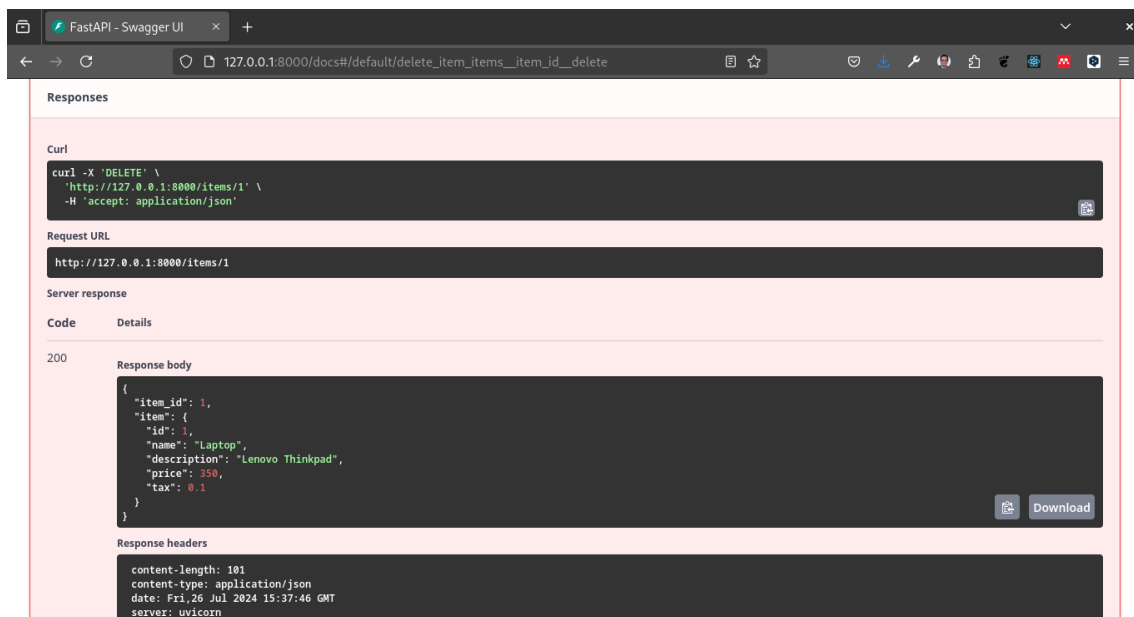
Request URL

http://127.0.0.1:8000/items/1

Server response

Code	Details
200	<p>Response body</p> <pre>{ "item_id": 1, "item": { "id": 1, "name": "Laptop", "description": "Lenovo Thinkpad", "price": 350, "tax": 0.1 } }</pre> <p>Download</p>

Para probar la ruta **DELETE** /items/{item_id} se puede enviar una petición DELETE a la ruta /items/{item_id} para eliminar un artículo específico del inventario.



De esta forma se pueden probar las rutas definidas en FastAPI utilizando la interfaz de usuario de Swagger.

14.5 Validaciones en FastAPI

En FastAPI se pueden realizar validaciones de datos utilizando los modelos definidos con Pydantic. Los modelos de Pydantic permiten definir tipos de datos, valores por defecto, validaciones y documentación para los campos de un modelo. En el ejemplo anterior se definió un modelo **Item** con los campos **id**, **name**, **description**, **price** y **tax**. A continuación se muestran algunas validaciones que se pueden realizar con Pydantic:

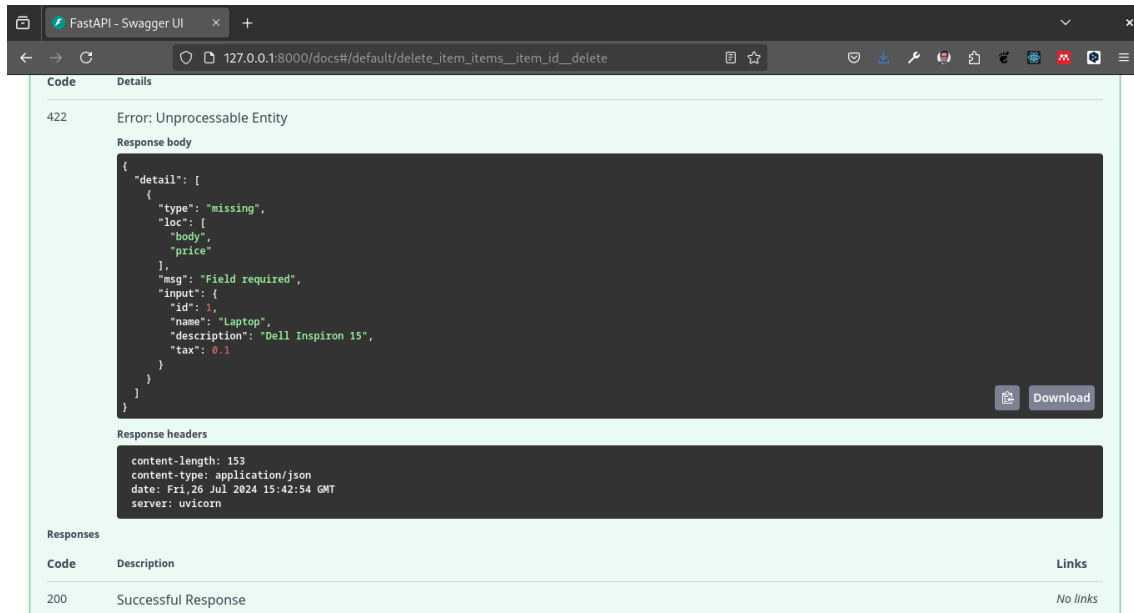
- **Tipos de datos:** Se pueden definir los tipos de datos de los campos de un modelo. Por ejemplo, el campo **id** es de tipo **int** y el campo **name** es de tipo **str**.
- **Valores por defecto:** Se pueden definir valores por defecto para los campos de un modelo. Por ejemplo, el campo **tax** tiene un valor por defecto de **0.0**.
- **Validaciones:** Se pueden definir validaciones para los campos de un modelo. Por ejemplo, se puede definir una validación para el campo **price** que requiera que el valor sea mayor que cero.
- **Documentación:** Se puede añadir documentación a los campos de un modelo utilizando la anotación **Field** de Pydantic. Por ejemplo, se puede añadir una descripción al campo **name** utilizando la anotación **Field**.

En el ejemplo anterior se utilizó el modelo **Item** para validar los datos de los artículos en el inventario. Al enviar una petición a la ruta **POST /items/** con un objeto JSON que no cumpla con las validaciones del modelo **Item**, la API responde con un error indicando que los datos no son válidos.

Por ejemplo si enviáramos un petición post con el siguiente JSON

```
{
  "id": 1,
  "name": "Laptop",
  "description": "Dell Inspiron 15",
  "tax": 0.1
}
```

La API responderá con un error indicando que el campo **price** es requerido.



De esta forma se pueden realizar validaciones de datos en FastAPI utilizando los modelos definidos con Pydantic.

En este capítulo aprendimos acerca de cómo definir rutas en FastAPI utilizando decoradores y cómo realizar validaciones de datos utilizando los modelos definidos con Pydantic. En el próximo capítulo veremos cómo realizar API RESTful en FastAPI.

15 APIs RESTful con FastAPI

En el capítulo anterior aprendimos acerca de la creación de Rutas y Validaciones en FastAPI. En este capítulo veremos cómo crear una API RESTful utilizando FastAPI.

15.1 Creación de una API RESTful

Una API RESTful es una API que sigue los principios de REST (Representational State Transfer). REST es un estilo de arquitectura de software que define un conjunto de restricciones para el diseño de servicios web. Las API RESTful son fáciles de entender, escalables y flexibles.

En FastAPI se pueden crear APIs RESTful utilizando rutas y modelos. A continuación se muestra un ejemplo de API RESTful para una Fundación de Adopción Animal, definiremos las adopciones, los animales y las personas que adoptan a los animales:

El proyecto tendrá la siguiente estructura:

```
proyecto/

  app/
    __init__.py
    main.py
  |  |  models.py
  |  |  routes.py

  .gitignore
  README.md
  requirements.txt
```

En el directorio **app/** se encuentra el archivo **routes.py** que contiene la definición de las rutas de la API. Vamos a crear los modelos, las rutas y las validaciones para una Fundación de Adopción Animal. A continuación se muestra el contenido del archivo **models**:

```
from pydantic import BaseModel

class Adoption(BaseModel):
    id: int
    animal_id: int
    person_id: int
    date: str
    status: str
```

```

class Animal(BaseModel):
    id: int
    name: str
    species: str
    breed: str

class Person(BaseModel):
    id: int
    name: str
    email: str
    phone: str

```

En el ejemplo anterior se definen los modelos **Adoption**, **Animal** y **Person** que heredan de la clase **BaseModel** de Pydantic. Cada modelo tiene sus propios campos que representan los datos de las adopciones, los animales y las personas.

15.2 Creación de las Rutas

A continuación se muestra el contenido del archivo **routes.py** que contiene la definición de las rutas de la API:

```

from fastapi import APIRouter, HTTPException
from .models import Adoption, Animal, Person

router = APIRouter()

adoptions = []
animals = []
persons = []

@router.post("/adoptions/")
def create_adoption(adoption: Adoption):
    adoptions.append(adoption)
    return adoption

@router.get("/adoptions/")
def read_adoptions():
    return adoptions

@router.get("/adoptions/{adoption_id}")
def read_adoption(adoption_id: int):
    for adoption in adoptions:
        if adoption.id == adoption_id:
            return adoption

@router.post("/animals/")

```

```

def create_animal(animal: Animal):
    animals.append(animal)
    return animal

@router.get("/animals/")
def read_animals():
    return animals

@router.get("/animals/{animal_id}")
def read_animal(animal_id: int):
    for animal in animals:
        if animal.id == animal_id:
            return animal

@router.post("/persons/")
def create_person(person: Person):
    persons.append(person)
    return person

@router.get("/persons/")
def read_persons():
    return persons

@router.get("/persons/{person_id}")
def read_person(person_id: int):
    for person in persons:
        if person.id == person_id:
            return person

```

En el ejemplo anterior se definen las rutas para las adopciones, los animales y las personas. Cada ruta tiene un método HTTP asociado (POST, GET) y una función que se ejecuta cuando se accede a la ruta. Las rutas de la API permiten crear, leer, actualizar y eliminar datos de las adopciones, los animales y las personas.

15.3 Uso de las Rutas en FastAPI

Para utilizar las rutas definidas en FastAPI, se deben importar las rutas en el archivo principal de la aplicación. A continuación se muestra un ejemplo de cómo importar las rutas en el archivo **main.py**:

```

from fastapi import FastAPI
from .routes import router

app = FastAPI()

app.include_router(router)

```


En el ejemplo anterior se importa el objeto **router** que contiene las rutas definidas en el archivo **routes.py**. Luego, se utiliza el método **include__router()** para incluir las rutas en la aplicación FastAPI.

16 Probar las Rutas en FastAPI

Podemos correr el servidor de pruebas de FastAPI con el siguiente comando:

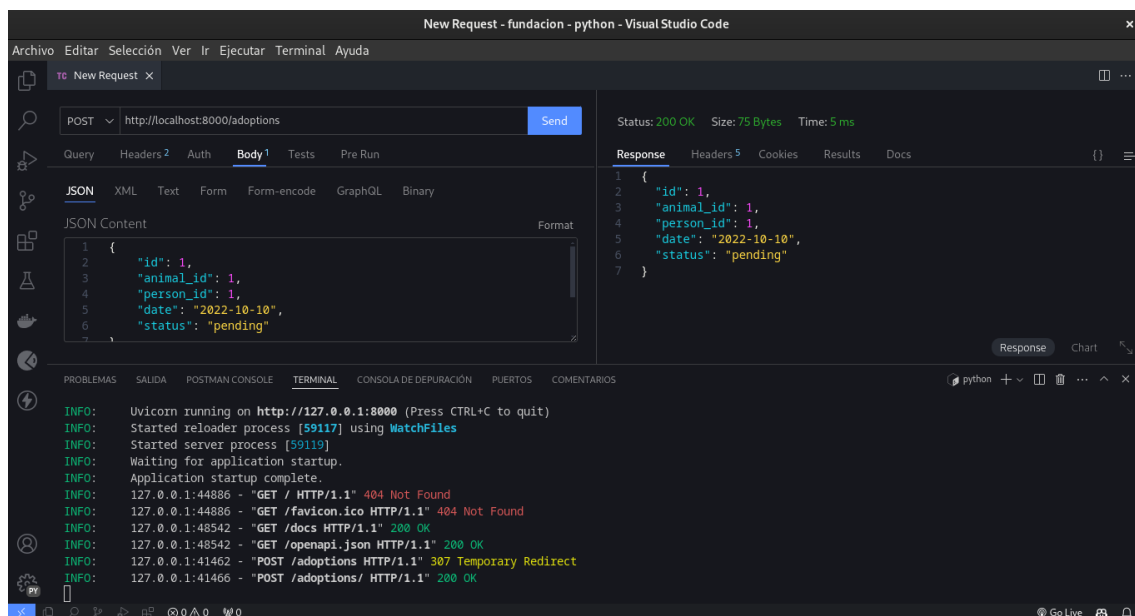
```
uvicorn app.main:app --reload
```

A continuación vamos a observar cómo funcionan las rutas de la API a través de Thunder Client en Visual Studio Code.

16.1 Crear una Adopción

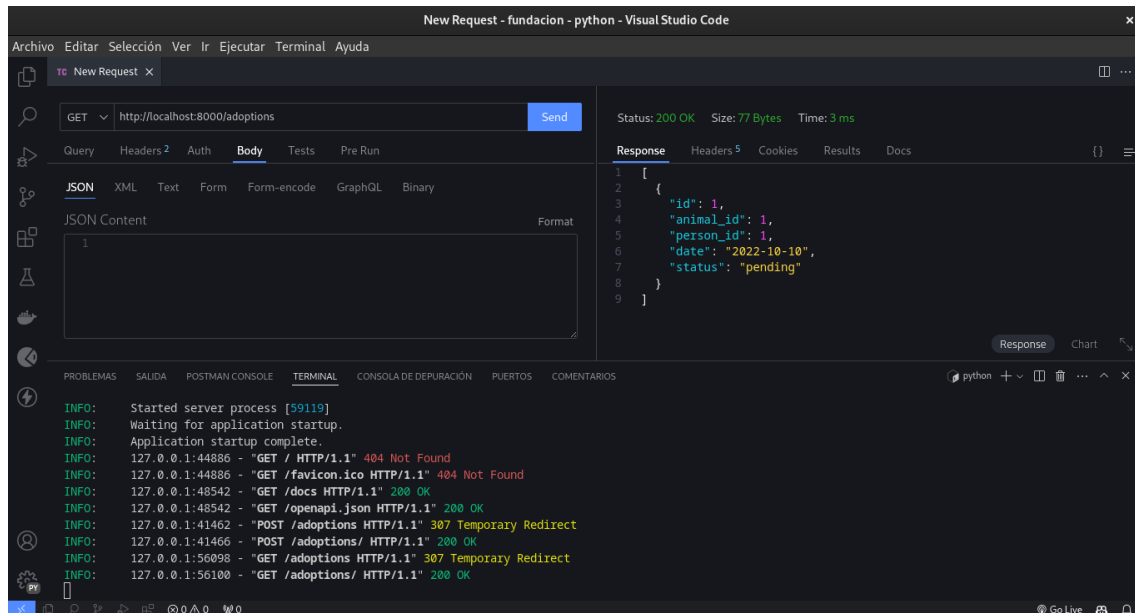
Para crear una adopción, se debe enviar una petición POST a la ruta `/adoptions/` con los datos de la adopción en el cuerpo de la petición. A continuación se muestra un ejemplo de cómo crear una adopción:

```
{
  "id": 1,
  "animal_id": 1,
  "person_id": 1,
  "date": "2022-10-10",
  "status": "pending"
}
```



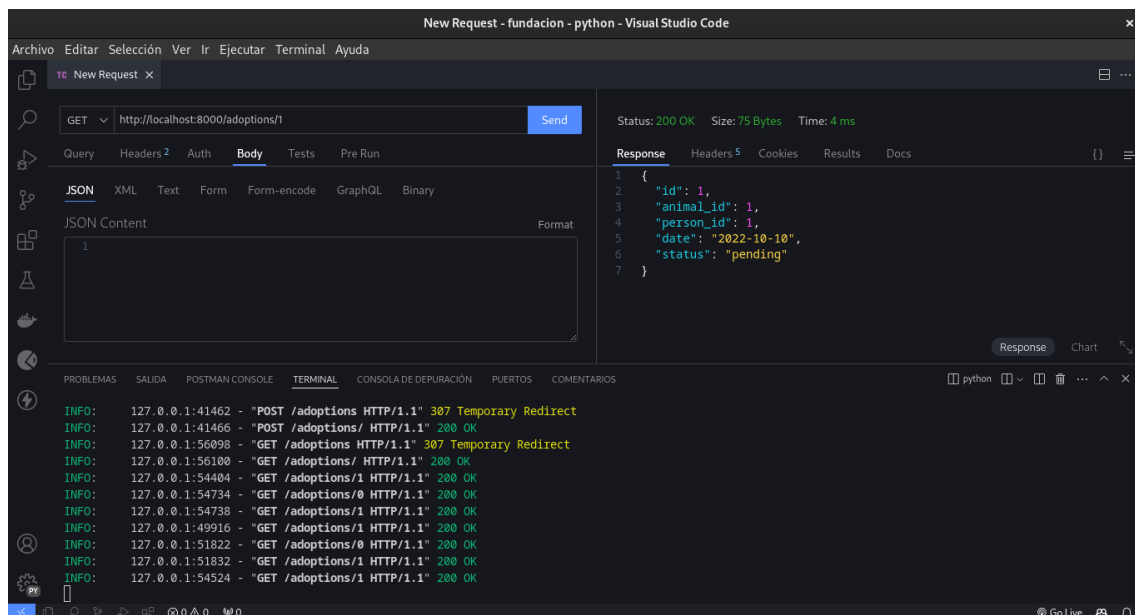
16.2 Obtener las Adopciones

Para obtener todas las adopciones, se debe enviar una petición GET a la ruta `/adoptions/`. A continuación se muestra un ejemplo de cómo obtener todas las adopciones:



16.3 Obtener una Adopción

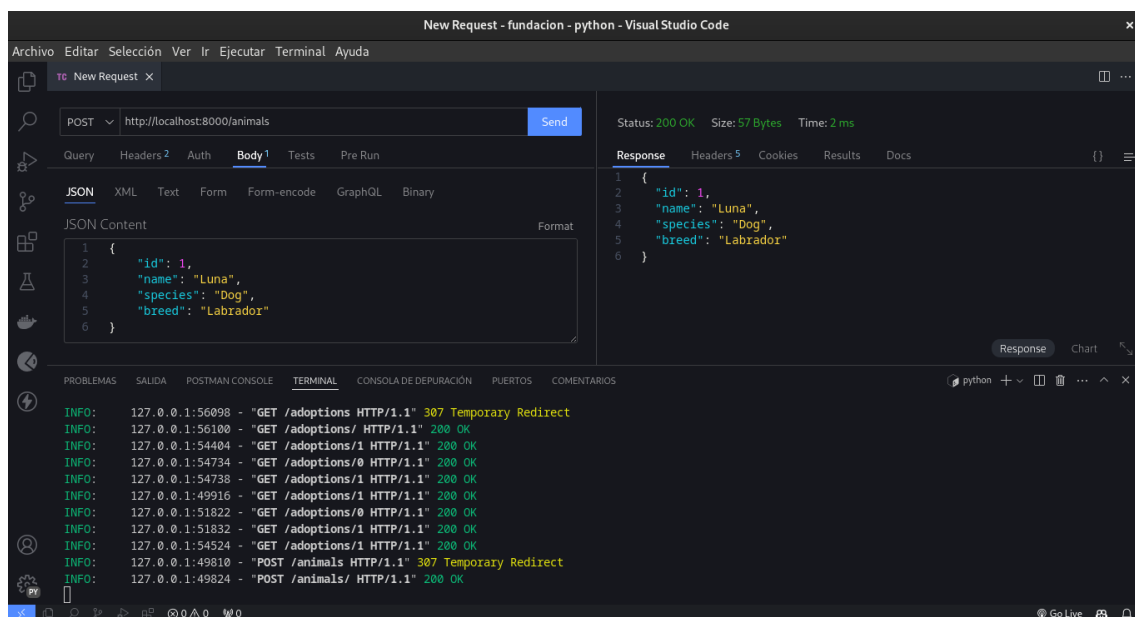
Para obtener una adopción específica, se debe enviar una petición GET a la ruta `/adoptions/{adoption_id}` con el ID de la adopción en la URL. A continuación se muestra un ejemplo de cómo obtener una adopción específica:



16.4 Crear un Animal

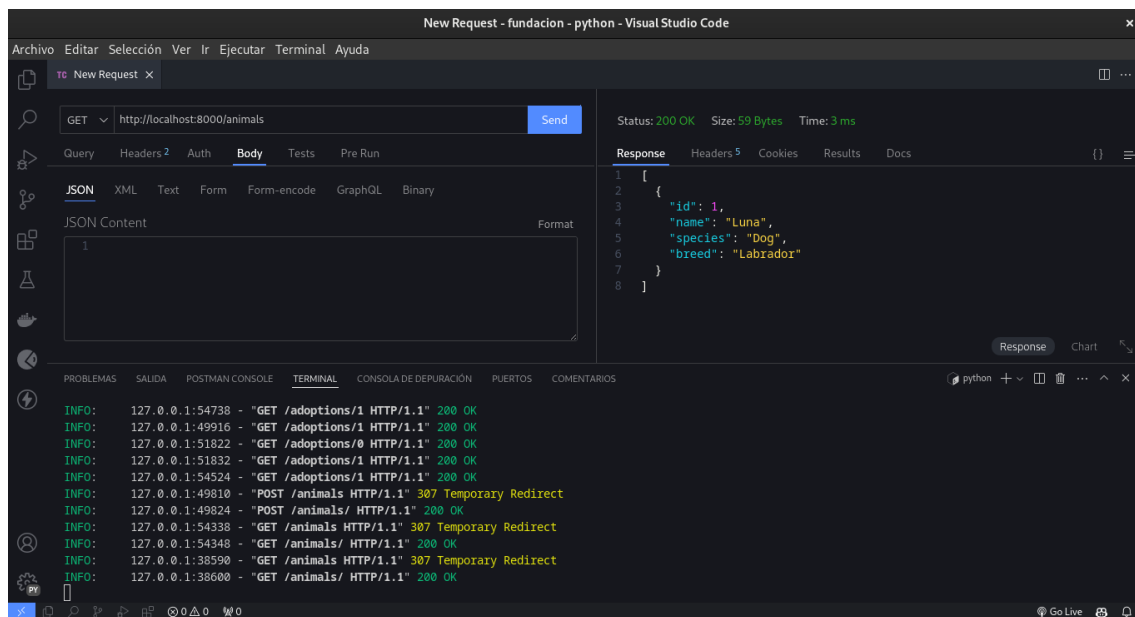
Para crear un animal, se debe enviar una petición POST a la ruta `/animals/` con los datos del animal en el cuerpo de la petición. A continuación se muestra un ejemplo de cómo crear un animal:

```
{  
  "id": 1,  
  "name": "Luna",  
  "species": "Dog",  
  "breed": "Labrador"  
}
```



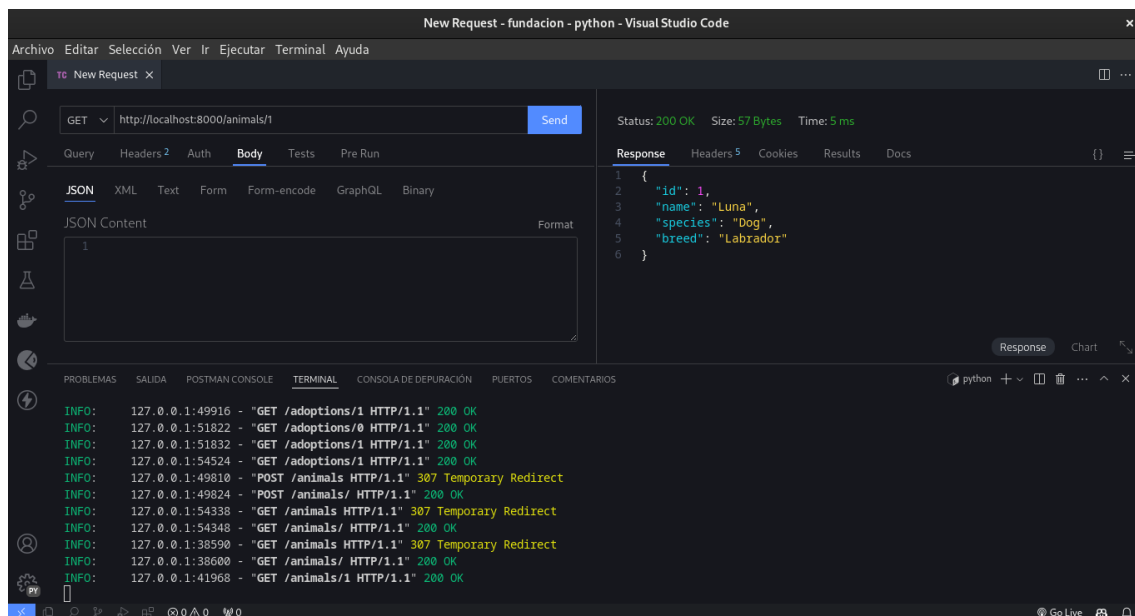
16.5 Obtener los Animales

Para obtener todos los animales, se debe enviar una petición GET a la ruta `/animals/`. A continuación se muestra un ejemplo de cómo obtener todos los animales:



16.6 Obtener un Animal

Para obtener un animal específico, se debe enviar una petición GET a la ruta `/animals/{animal_id}` con el ID del animal en la URL. A continuación se muestra un ejemplo de cómo obtener un animal específico:

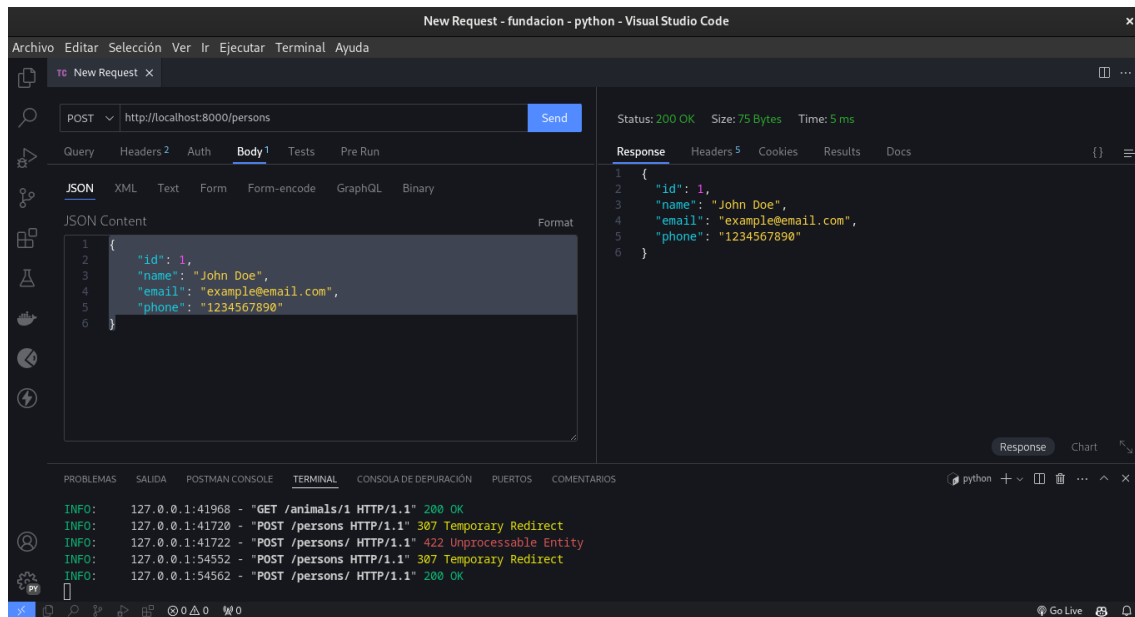


16.7 Crear una Persona

Para crear una persona, se debe enviar una petición POST a la ruta `/persons/` con los datos de la persona en el cuerpo de la petición. A continuación se muestra un ejemplo de

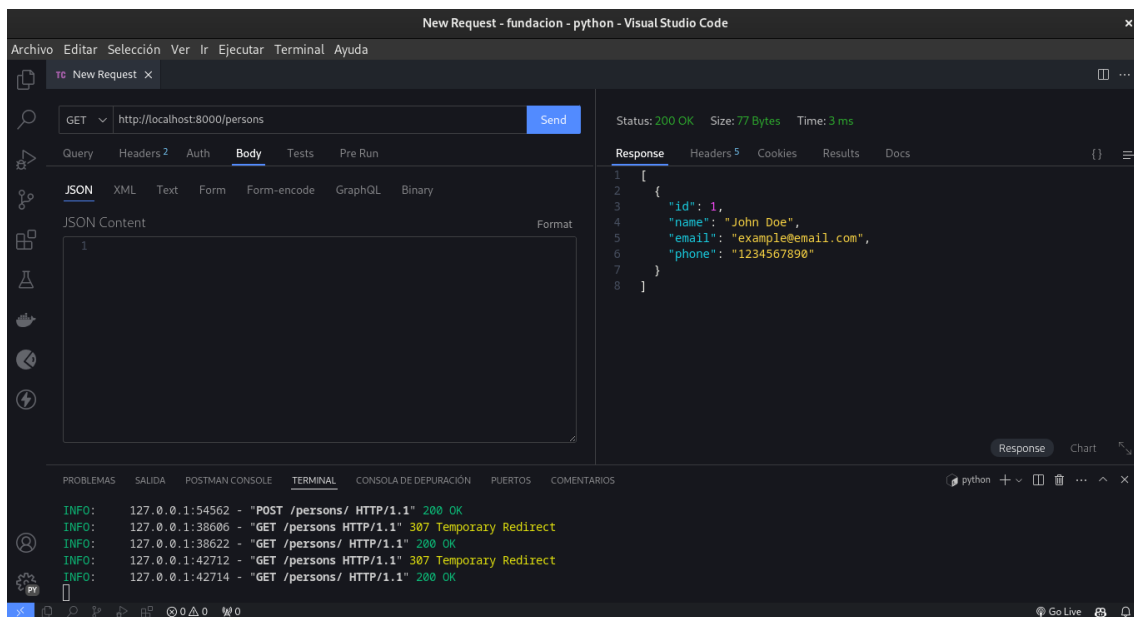
cómo crear una persona:

```
{  
  "id": 1,  
  "name": "John Doe",  
  "email": "example@email.com",  
  "phone": "1234567890"  
}
```



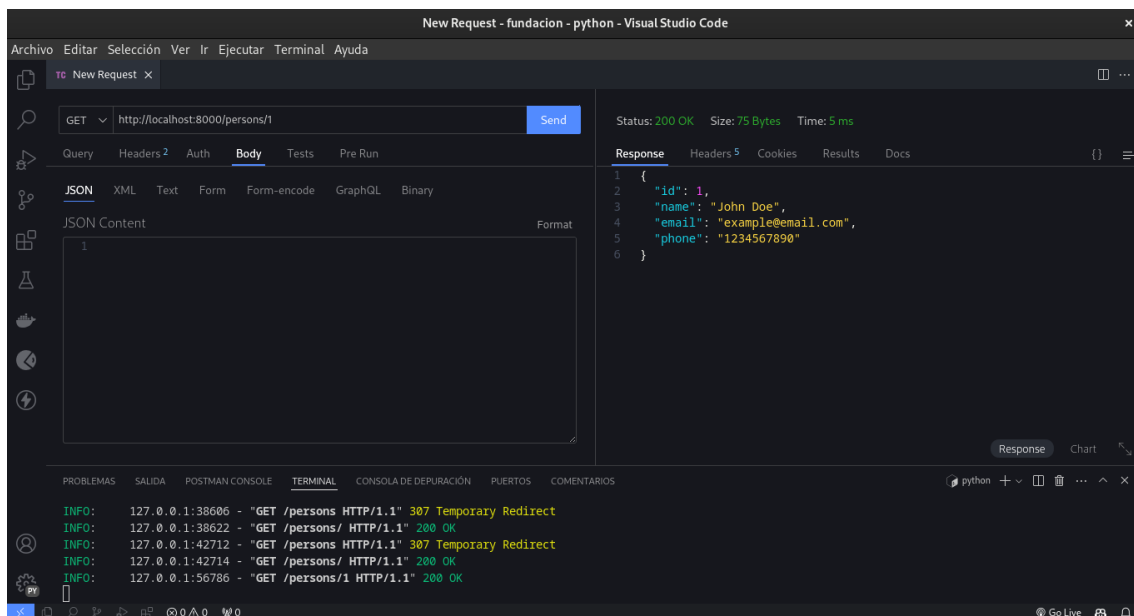
16.8 Obtener las Personas

Para obtener todas las personas, se debe enviar una petición GET a la ruta `/persons/`. A continuación se muestra un ejemplo de cómo obtener todas las personas:



16.9 Obtener una Persona

Para obtener una persona específica, se debe enviar una petición GET a la ruta `/persons/{person_id}` con el ID de la persona en la URL. A continuación se muestra un ejemplo de cómo obtener una persona específica:



En el ejemplo anterior se envía una petición GET a la ruta `/persons/{person_id}` para obtener una persona específica. La API responde con un objeto JSON que representa la persona con el ID especificado en la URL.

Como podemos observar, hemos creado una API RESTful utilizando FastAPI. En el siguiente capítulo veremos cómo realizar pruebas unitarias en FastAPI.

17 Pruebas Unitarias en FastAPI

En el capítulo anterior se creó una API RESTful con FastAPI. En este capítulo se mostrará cómo realizar pruebas unitarias a las rutas de la API.

17.1 Pruebas Unitarias

Las pruebas unitarias son una técnica de programación que consiste en verificar que una unidad de código (como una función o un método) funcione correctamente. En el caso de una API RESTful, las pruebas unitarias se utilizan para verificar que las rutas de la API devuelvan la respuesta esperada.

17.2 Pruebas Unitarias en FastAPI

FastAPI proporciona una forma sencilla de realizar pruebas unitarias a las rutas de la API. Para ello, se utiliza la biblioteca **pytest** y el módulo **test__app.py** de FastAPI.

```
pip install pytest httpx
```

A continuación se muestra un ejemplo de cómo realizar pruebas unitarias a las rutas de la API creada en el capítulo anterior:

```
import pytest
from fastapi.testclient import TestClient
from app.main import app
from app.models import Adoption, Animal, Person

client = TestClient(app)

def test_create_adoption():
    response = client.post("/adoptions/", json={
        "id": 1,
        "animal_id": 1,
        "person_id": 1,
        "date": "2023-07-30",
        "status": "pending"
    })
    assert response.status_code == 200
    assert response.json() == {
```



```

        "id": 1,
        "animal_id": 1,
        "person_id": 1,
        "date": "2023-07-30",
        "status": "pending"
    }

def test_read_adoptions():
    response = client.get("/adoptions/")
    assert response.status_code == 200
    assert isinstance(response.json(), list)

def test_read_adoption():
    test_create_adoption()
    response = client.get("/adoptions/1")
    assert response.status_code == 200
    assert response.json()["id"] == 1

def test_create_animal():
    response = client.post("/animals/", json={
        "id": 1,
        "name": "Buddy",
        "species": "Dog",
        "breed": "Golden Retriever"
    })
    assert response.status_code == 200
    assert response.json() == {
        "id": 1,
        "name": "Buddy",
        "species": "Dog",
        "breed": "Golden Retriever"
    }

def test_read_animals():
    response = client.get("/animals/")
    assert response.status_code == 200
    assert isinstance(response.json(), list)

def test_read_animal():
    test_create_animal() # Ensure there's at least one animal
    response = client.get("/animals/1")
    assert response.status_code == 200
    assert response.json()["id"] == 1

def test_create_person():
    response = client.post("/persons/", json={
        "id": 1,
        "name": "John Doe",

```

```

        "email": "john.doe@example.com",
        "phone": "1234567890"
    })
    assert response.status_code == 200
    assert response.json() == {
        "id": 1,
        "name": "John Doe",
        "email": "john.doe@example.com",
        "phone": "1234567890"
    }

def test_read_persons():
    response = client.get("/persons/")
    assert response.status_code == 200
    assert isinstance(response.json(), list)

def test_read_person():
    test_create_person() # Ensure there's at least one person
    response = client.get("/persons/1")
    assert response.status_code == 200
    assert response.json()["id"] == 1

```

En el ejemplo anterior se importa la clase **TestClient** de FastAPI y se crea una instancia de **TestClient** con la aplicación FastAPI. Luego se definen dos pruebas unitarias: una para verificar que la ruta **/adoptions/** devuelva una lista vacía de adopciones y otra para verificar que la ruta **/adoptions/** cree una nueva adopción con los datos proporcionados.

Para ejecutar las pruebas unitarias, se utiliza el comando **pytest** en la terminal:

```
pytest
```

Al ejecutar las pruebas unitarias, se mostrará el resultado de las pruebas en la terminal. Si todas las pruebas pasan, se mostrará un mensaje indicando que todas las pruebas han pasado correctamente.

The screenshot shows the Visual Studio Code interface with a file explorer on the left, a code editor in the center, and a terminal at the bottom. The file explorer shows a project named 'FUNDACION' with files like 'test_app.py', 'main.py', 'models.py', and 'routes.py'. The code editor displays the content of 'test_app.py', which includes imports for 'pytest', 'fastapi.testclient', and 'app', and a test function 'test_create_adoption' that uses 'TestClient' to post data to '/adoption/' and asserts the response status and JSON content. The terminal at the bottom shows the output of running the tests, indicating that all 9 tests passed successfully in 0.50s.

```
test_app.py - fundacion - python - Visual Studio Code
Archivo  Editar  Selección  Ver  Ir  Ejecutar  Terminal  Ayuda

EXPLORADOR
FUNDACION
├── .pytest_cache
├── app
│   ├── __pypcache__
│   ├── __init__.py
│   ├── main.py
│   ├── models.py
│   ├── routes.py
│   └── test_app.py
├── env
├── .gitignore
└── README.md

test_app.py
1 import pytest
2 from fastapi.testclient import TestClient
3 from app.main import app
4 from app.models import Adoption, Animal, Person
5
6 client = TestClient(app)
7
8 def test_create_adoption():
9     response = client.post("/adoption/", json={
10         "id": 1,
11         "animal_id": 1,
12         "person_id": 1,
13         "date": "2023-07-30",
14         "status": "pending"
15     })
16     assert response.status_code == 200
17     assert response.json() == {
18         "id": 1,
19         "animal_id": 1,
20         "person_id": 1,
21         "date": "2023-07-30"
22     }

TERMINAL
plugins: anyio-4.4.0
collected 9 items

app/test_app.py ..... [100%]

===== 9 passed in 0.50s =====
env static at fedora in ~/.../unidad5/fundacion
```

En resumen, en este capítulo se mostró cómo realizar pruebas unitarias a las rutas de una API RESTful creada con FastAPI. Las pruebas unitarias son una técnica de programación que permite verificar que una unidad de código funcione correctamente y que la API devuelva la respuesta esperada.

En el siguiente capítulo se mostrará cómo mejorar la optimización y el rendimiento.

18 Optimización y Rendimiento

En el capítulo anterior se creó una API RESTful con FastAPI y se realizaron pruebas unitarias a las rutas de la API. En este capítulo se abordarán temas relacionados con la optimización y el rendimiento de una API, como la implementación de caché, la compresión de respuestas y la configuración de la API para producción.

18.1 Caché

La caché es una técnica que se utiliza para almacenar temporalmente los resultados de operaciones costosas en memoria o en disco, de modo que puedan ser reutilizados en futuras solicitudes. La caché mejora el rendimiento de una API al reducir el tiempo de respuesta de las solicitudes.

FastAPI proporciona soporte para la caché a través de la biblioteca **cachetools**. A continuación se muestra un ejemplo de cómo implementar la caché, lo primero que haremos será instalar la librería **cachetools**:

```
pip install cachetools
```

Ahora modificamos el archivo **routes.py** para implementar la caché en la ruta de lectura de animales:

```
from fastapi import APIRouter, HTTPException
from .models import Adoption, Animal, Person
from cachetools import TTLCache, cached

router = APIRouter()

adoptions = []
animals = []
persons = []

# Define TTL caches for each resource
adoptions_cache = TTLCache(maxsize=100, ttl=300)
animals_cache = TTLCache(maxsize=100, ttl=300)
persons_cache = TTLCache(maxsize=100, ttl=300)

@router.post("/adoptions/")
def create_adoption(adoption: Adoption):
    adoptions.append(adoption)
```

```

        adoptions_cache.clear() # Clear cache when new data is added
        return adoption

@router.get("/adoptions/")
@cached(adoptions_cache)
def read_adoptions():
    return adoptions

@router.get("/adoptions/{adoption_id}")
def read_adoption(adoption_id: int):
    for adoption in adoptions:
        if adoption.id == adoption_id:
            return adoption

@router.post("/animals/")
def create_animal(animal: Animal):
    animals.append(animal)
    animals_cache.clear() # Clear cache when new data is added
    return animal

@router.get("/animals/")
@cached(animals_cache)
def read_animals():
    return animals

@router.get("/animals/{animal_id}")
def read_animal(animal_id: int):
    for animal in animals:
        if animal.id == animal_id:
            return animal

@router.post("/persons/")
def create_person(person: Person):
    persons.append(person)
    persons_cache.clear() # Clear cache when new data is added
    return person

@router.get("/persons/")
@cached(persons_cache)
def read_persons():
    return persons

@router.get("/persons/{person_id}")
def read_person(person_id: int):
    for person in persons:
        if person.id == person_id:
            return person

```

En el ejemplo anterior, se crearon tres cachés TTL (Time-To-Live) para almacenar los resultados de las operaciones de lectura de adopciones, animales y personas. La caché se activa mediante el decorador `@cached`, que toma como argumento la caché a utilizar. La caché se limpia cuando se añaden nuevos datos a la lista correspondiente.

Ahora modificamos el archivo `test_app.py` para probar la caché en la ruta de lectura de animales:

```
import pytest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_create_adoption():
    response = client.post("/adoptions/", json={
        "id": 1,
        "animal_id": 1,
        "person_id": 1,
        "date": "2023-07-30",
        "status": "pending"
    })
    assert response.status_code == 200
    assert response.json() == {
        "id": 1,
        "animal_id": 1,
        "person_id": 1,
        "date": "2023-07-30",
        "status": "pending"
    }

def test_read_adoptions():
    response = client.get("/adoptions/")
    assert response.status_code == 200
    assert isinstance(response.json(), list)

def test_read_adoption():
    response = client.post("/adoptions/", json={
        "id": 1,
        "animal_id": 1,
        "person_id": 1,
        "date": "2023-07-30",
        "status": "pending"
    })
    response = client.get("/adoptions/1")
    assert response.status_code == 200
    assert response.json()["id"] == 1

def test_create_animal():
```

```

response = client.post("/animals/", json={
    "id": 1,
    "name": "Buddy",
    "species": "Dog",
    "breed": "Golden Retriever"
})
assert response.status_code == 200
assert response.json() == {
    "id": 1,
    "name": "Buddy",
    "species": "Dog",
    "breed": "Golden Retriever"
}

def test_read_animals():
    response = client.get("/animals/")
    assert response.status_code == 200
    assert isinstance(response.json(), list)

def test_read_animal():
    response = client.post("/animals/", json={
        "id": 1,
        "name": "Buddy",
        "species": "Dog",
        "breed": "Golden Retriever"
    })
    response = client.get("/animals/1")
    assert response.status_code == 200
    assert response.json()["id"] == 1

def test_create_person():
    response = client.post("/persons/", json={
        "id": 1,
        "name": "John Doe",
        "email": "john.doe@example.com",
        "phone": "1234567890"
    })
    assert response.status_code == 200
    assert response.json() == {
        "id": 1,
        "name": "John Doe",
        "email": "john.doe@example.com",
        "phone": "1234567890"
    }

def test_read_persons():
    response = client.get("/persons/")
    assert response.status_code == 200

```

```

    assert isinstance(response.json(), list)

def test_read_person():
    response = client.post("/persons/", json={
        "id": 1,
        "name": "John Doe",
        "email": "john.doe@example.com",
        "phone": "1234567890"
    })
    response = client.get("/persons/1")
    assert response.status_code == 200
    assert response.json()["id"] == 1

```

En el ejemplo anterior, se añadieron pruebas unitarias para probar la caché en la ruta de lectura de animales. Se crearon pruebas para las operaciones de creación y lectura de animales, y se verificó que los resultados de las operaciones de lectura se almacenan en caché.

Finalmente ejecutaremos las pruebas unitarias:

pytest

The screenshot shows the Visual Studio Code interface with a file named `test_app.py` open. The file contains two test functions: `test_read_persons()` and `test_read_person()`. The `test_read_person()` function is highlighted, showing a POST request to `/persons/` with a JSON body containing an animal record, followed by a GET request to `/persons/1` and assertions for the status code and the returned JSON.

Below the editor, the `TERMINAL` panel shows the output of running `pytest` in the `app` directory. The output indicates that all tests passed successfully.

```

app/test_app.py ..... [100%]

===== 9 passed in 0.48s =====

```

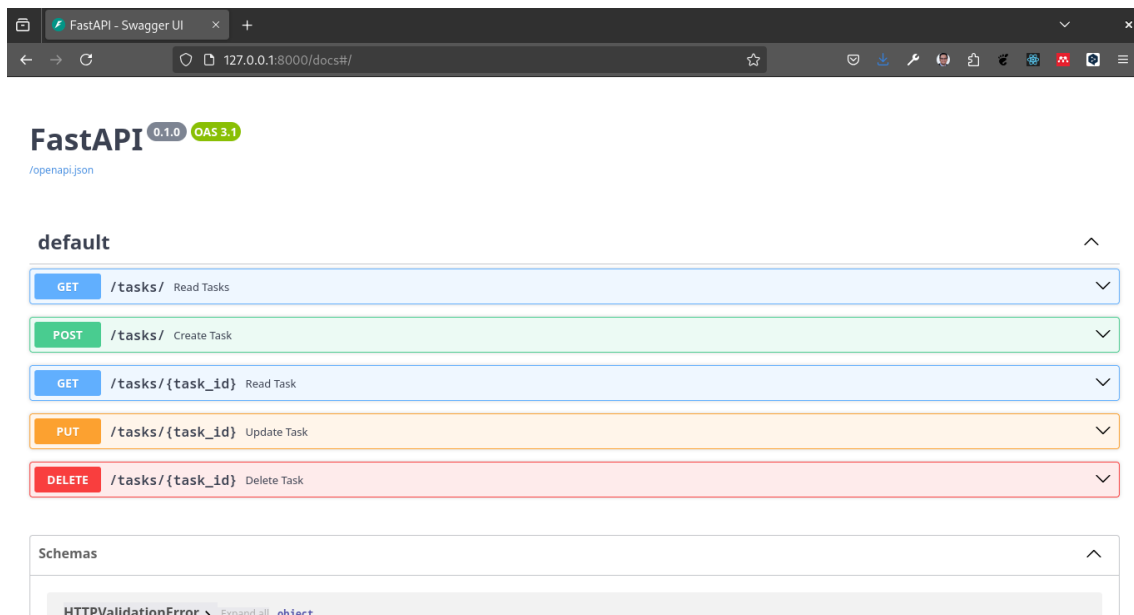
The status bar at the bottom shows the file is at line 102, column 1, using UTF-8 encoding and LF line endings, with Python 3.12.4 (venv) as the interpreter.

En el ejemplo anterior se ejecutaron las pruebas unitarias para verificar que la caché funciona correctamente en la ruta de lectura de animales. Las pruebas pasaron con éxito,

lo que indica que la caché está implementada correctamente y mejora el rendimiento de la API.

En este capítulo se abordaron temas relacionados con la optimización y el rendimiento de una API, como la implementación de caché. En el siguiente capítulo se verá cómo crear una api de gestión de tareas con FastAPI.

19 Creación de una API de gestión de tareas utilizando FastAPI



En el capítulo anterior se vio cómo optimizar y mejorar el rendimiento de una API utilizando caché. En este capítulo se verá cómo crear una API de gestión de tareas utilizando FastAPI.

19.1 Creación de una API de gestión de tareas

Empezamos por la creación de un entorno virtual y la instalación de FastAPI, Uvicorn y pydantic:

```
pip install fastapi uvicorn pydantic
```

El proyecto tendrá la siguiente estructura:

```
proyecto/  
  
  app/  
    __init__.py  
    main.py  
    models.py
```

```
routes.py

.gitignore
README.md
requirements.txt
```

En el directorio **app/** se encuentra el archivo **routes.py** que contiene la definición de las rutas de la API. Vamos a crear los modelos, las rutas y las validaciones para una API de gestión de tareas. A continuación se muestra el contenido del archivo **models**:

```
from pydantic import BaseModel

class Task(BaseModel):
    id: int
    title: str
    description: str
    status: str
```

En el ejemplo anterior se define el modelo **Task** que hereda de la clase **BaseModel** de Pydantic. El modelo tiene los campos **id**, **title**, **description** y **status** que representan los datos de las tareas.

19.2 Creación de las Rutas

A continuación se muestra el contenido del archivo **routes.py** que contiene la definición de las rutas de la API:

```
from fastapi import APIRouter, HTTPException
from .models import Task

router = APIRouter()

tasks = []

@router.post("/tasks/")
def create_task(task: Task):
    tasks.append(task)
    return task

@router.get("/tasks/")
def read_tasks():
    return tasks

@router.get("/tasks/{task_id}")
def read_task(task_id: int):
    for task in tasks:
```

```

        if task.id == task_id:
            return task

@router.put("/tasks/{task_id}")
def update_task(task_id: int, task: Task):
    for t in tasks:
        if t.id == task_id:
            t.title = task.title
            t.description = task.description
            t.status = task.status
            return t

@router.delete("/tasks/{task_id}")
def delete_task(task_id: int):
    for i, task in enumerate(tasks):
        if task.id == task_id:
            del tasks[i]
            return task

```

En el ejemplo anterior se definen las rutas para crear, leer, actualizar y eliminar tareas. La ruta **POST** `/tasks/` permite crear una nueva tarea, la ruta **GET** `/tasks/` permite leer todas las tareas, la ruta **GET** `/tasks/{task_id}` permite leer una tarea específica, la ruta **PUT** `/tasks/{task_id}` permite actualizar una tarea y la ruta **DELETE** `/tasks/{task_id}` permite eliminar una tarea.

Finalmente, en el archivo **main.py** se importan las rutas y se crea la aplicación FastAPI:

```

from fastapi import FastAPI
from .routes import router

app = FastAPI()

app.include_router(router)

```

19.3 Ejecución de la API

Para ejecutar la API, se utiliza el siguiente comando:

```
uvicorn app.main:app --reload
```

La API estará disponible en la dirección <http://127.0.0.1:8000>. Se pueden probar las rutas utilizando un cliente Thunder Client en Visual Studio Code o Postman. También visitando en el navegador <http://127.0.0.1:8000/docs> se puede acceder a la documentación interactiva de la API.

FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/create_task_tasks_post

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/tasks/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 0,
    "title": "This is a example of task",
    "description": "This is a example of description task",
    "status": "realized"
  }'
```

Request URL

http://127.0.0.1:8000/tasks/

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 0, "title": "This is a example of task", "description": "This is a example of description task", "status": "realized" }</pre> <p>Response headers</p> <pre>content-length: 118 content-type: application/json date: Tue, 30 Jul 2024 20:27:42 GMT server: uvicorn</pre>

Responses

FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/read_task_tasks_task_id_get

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/tasks/0' \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8000/tasks/0

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 0, "title": "This is a example of task", "description": "This is a example of description task", "status": "realized" }</pre> <p>Response headers</p> <pre>content-length: 118 content-type: application/json date: Tue, 30 Jul 2024 20:28:37 GMT server: uvicorn</pre>

Responses

Code	Description	Links
------	-------------	-------

FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/update_task_tasks__task_id__put

Responses

Curl

```
curl -X 'PUT' \
  'http://127.0.0.1:8000/tasks/0' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 0,
    "title": "This is a example of task update",
    "description": "This is a example of description task",
    "status": "realized"
  }'
```

Request URL

http://127.0.0.1:8000/tasks/0

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 0, "title": "This is a example of task update", "description": "This is a example of description task", "status": "realized" }</pre> <p>Response headers</p>

FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/delete_task_tasks__task_id__delete

Responses

Curl

```
curl -X 'DELETE' \
  'http://127.0.0.1:8000/tasks/0' \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8000/tasks/0

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 0, "title": "This is a example of task update", "description": "This is a example of description task", "status": "realized" }</pre> <p>Response headers</p> <pre>content-length: 125 content-type: application/json date: Tue 30 Jul 2024 20:30:12 GMT server: uvicorn</pre>

Responses

Code	Description
------	-------------

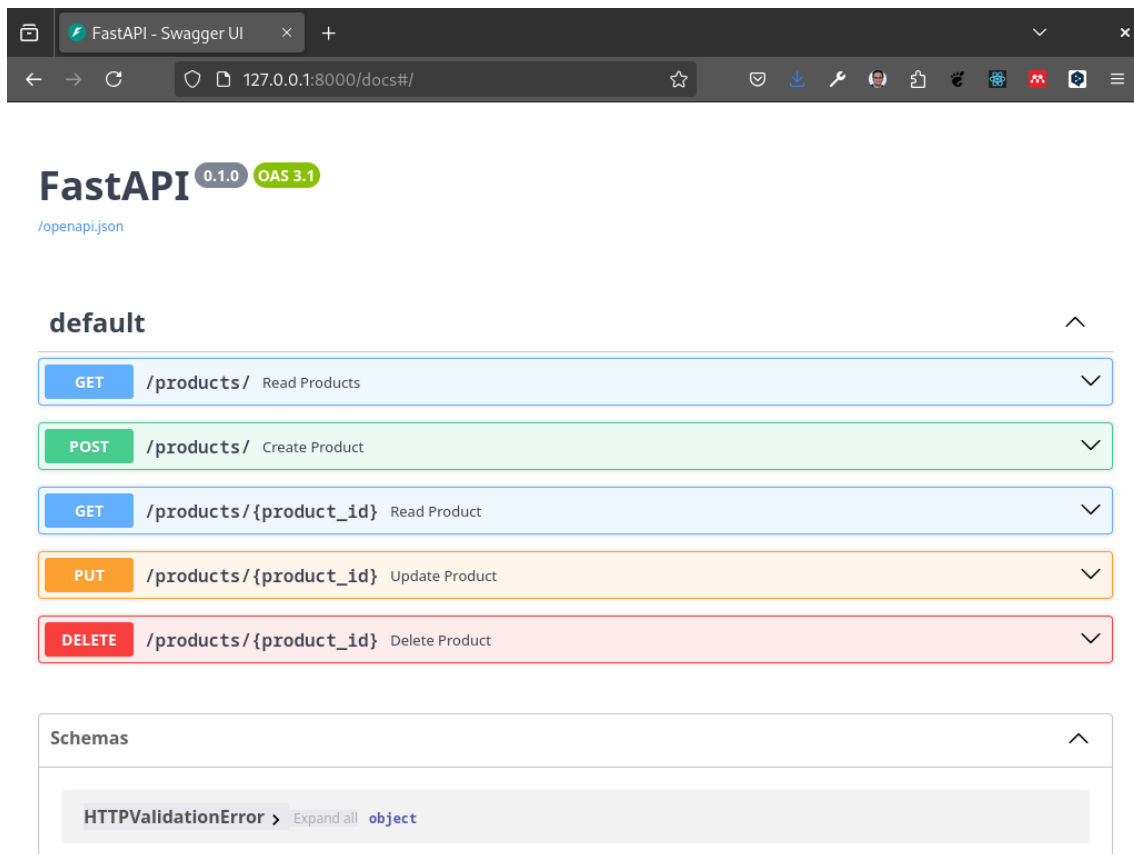
Links

En este capítulo se vio cómo crear una API de gestión de tareas utilizando FastAPI. En el próximo capítulo se verá cómo realizar pruebas unitarias a las rutas de la API.

Part VI

Unidad 6: Consumo API con FastAPI

20 Configuración y Uso de FastAPI para consumir APIs



FastAPI se caracteriza por su facilidad de uso y su capacidad para crear APIs RESTful de forma rápida y sencilla. En este apartado se muestra cómo configurar y utilizar FastAPI para consumir APIs para este ejemplo consumiremos la API de <https://fakestoreapi.com/>.

20.1 Creación de un entorno virtual

Para empezar, se debe crear un entorno virtual e instalar FastAPI y Uvicorn. Para ello, se ejecutan los siguientes comandos:

```
python3 -m venv env
source env/bin/activate
pip install fastapi uvicorn httpx pydantic
```


Nuestro proyecto tendrá la siguiente estructura:

```
proyecto/  
  
  app/  
    __init__.py  
    main.py  
    models.py  
    routes.py  
  
  .gitignore  
  README.md  
  requirements.txt
```

En el directorio **app/** se encuentra el archivo **routes.py** que contiene la definición de las rutas de la API. Vamos a crear los modelos, las rutas y las validaciones para consumir la API de <fakestoreapi.com>. A continuación se muestra el contenido del archivo **models**:

```
from pydantic import BaseModel  
  
class Product(BaseModel):  
    id: int  
    title: str  
    price: float  
    description: str  
    category: str  
    image: str
```

En el ejemplo anterior se define el modelo **Product** que hereda de la clase **BaseModel** de Pydantic. El modelo tiene los campos **id**, **title**, **price**, **description**, **category** e **image** que representan los datos de los productos.

20.2 Consumo de la API

A continuación se muestra el contenido del archivo **routes.py** que contiene la definición de las rutas de la API:

```
from fastapi import APIRouter, HTTPException  
from httpx import AsyncClient  
from .models import Product  
  
router = APIRouter()  
  
@router.get("/products/")  
async def read_products():
```

```

    async with AsyncClient() as client:
        response = await client.get("https://fakestoreapi.com/products")
        products = response.json()
        return products

@router.get("/products/{product_id}")
async def read_product(product_id: int):
    async with AsyncClient() as client:
        response = await client.get(f"https://fakestoreapi.com/products/{product_id}")
        product = response.json()
        return product

@router.post("/products/")
async def create_product(product: Product):
    async with AsyncClient() as client:
        response = await client.post("https://fakestoreapi.com/products", json=product.dict())
        product = response.json()
        return product

@router.put("/products/{product_id}")
async def update_product(product_id: int, product: Product):
    async with AsyncClient() as client:
        response = await client.put(f"https://fakestoreapi.com/products/{product_id}", json=product.dict())
        product = response.json()
        return product

@router.delete("/products/{product_id}")
async def delete_product(product_id: int):
    async with AsyncClient() as client:
        response = await client.delete(f"https://fakestoreapi.com/products/{product_id}")
        product = response.json()
        return product

```

En el ejemplo anterior se definen las rutas para consumir la API de <fakestoreapi.com>. Las rutas **read_products** y **read_product** permiten obtener la lista de productos y un producto en particular, respectivamente. Las rutas **create_product**, **update_product** y **delete_product** permiten crear, actualizar y eliminar un producto, respectivamente.

20.3 Ejecución de la API

Para ejecutar la API, se debe crear un archivo **main.py** en el directorio **app/** con el siguiente contenido:

```

from fastapi import FastAPI
from .routes import router

```

```
app = FastAPI()

app.include_router(router)
```

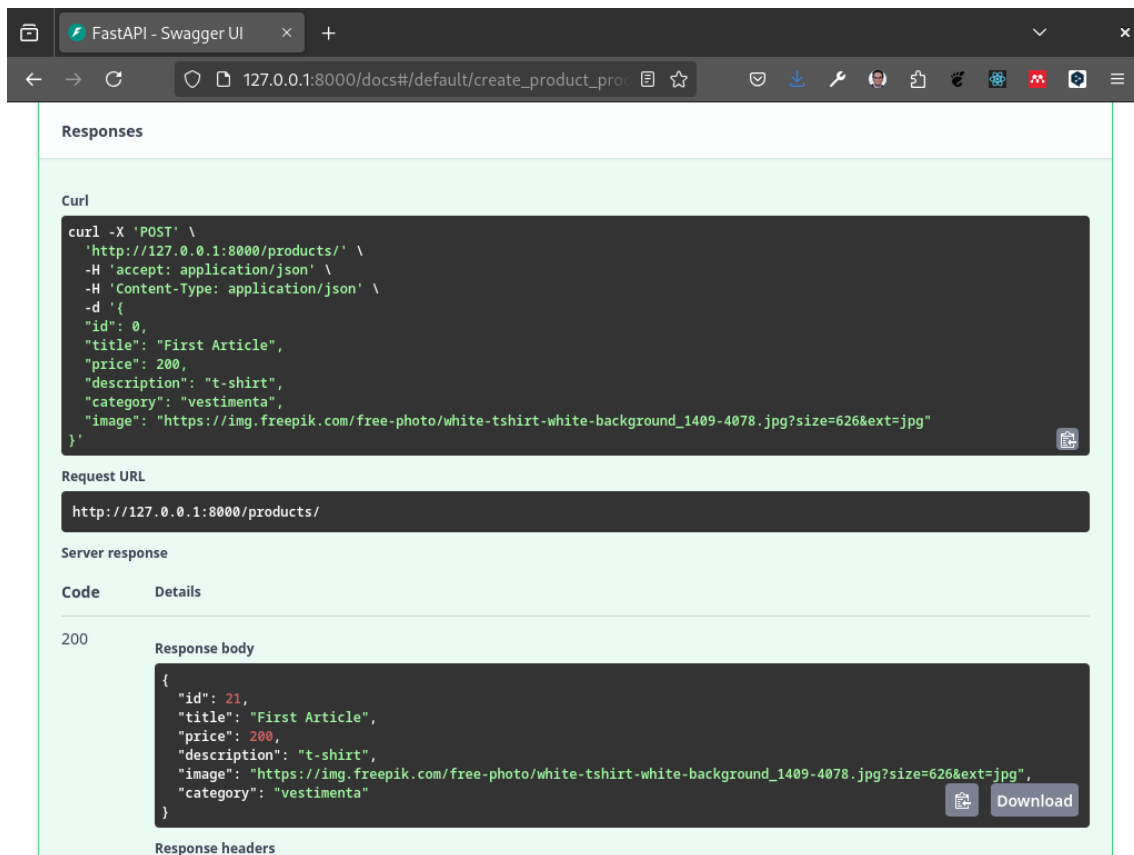
Para ejecutar la API, se utiliza el siguiente comando:

```
uvicorn app.main:app --reload
```

Una vez que la API esté en ejecución, se pueden realizar peticiones a las rutas definidas en el archivo **routes.py**. Por ejemplo, para obtener la lista de productos se puede acceder a la URL <http://127.0.0.1:8000/products/>.

20.4 Pruebas de la API

Para probar la API se puede utilizar un cliente HTTP como Thunder Client en Visual Studio Code o Postman. También se puede acceder a la documentación interactiva de la API en <http://127.0.0.1:8000/docs> para probar las rutas y ver los resultados de las peticiones.



FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/read_products_produ

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/products/' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/products/
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "category": "women's clothing", "image": "https://fakestoreapi.com/img/51eg55uWmdL._AC_UX679_.jpg", "rating": { "rate": 4.5, "count": 146 }, { "id": 20, "title": "DANVOUY Womens T Shirt Casual Cotton Short", "price": 12.99, "description": "95%Cotton,5%Spandex, Features: Casual, Short Sleeve, Letter Print,V-Neck,Fashion Tees, The fabric is soft and has some stretch., Occasion: Casual/Office/Beach/School/Home/Street. Season: Spring,Summer,Autumn,Winter.", "category": "women's clothing", "image": "https://fakestoreapi.com/img/61pHAEJ4NML._AC_UX679_.jpg",</pre>

FastAPI - Swagger UI

127.0.0.1:8000/docs#/default/read_product_produ

Responses

Curl

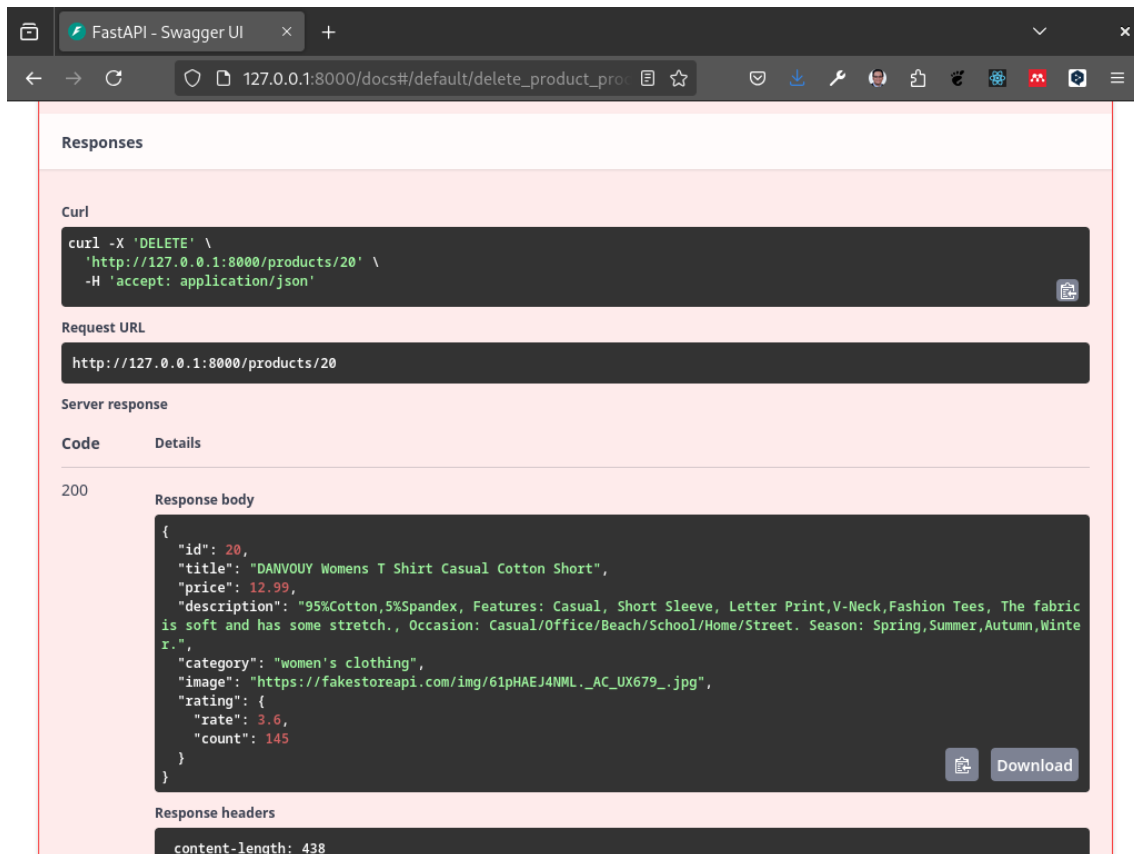
```
curl -X 'GET' \
  'http://127.0.0.1:8000/products/20' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/products/20
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 20, "title": "DANVOUY Womens T Shirt Casual Cotton Short", "price": 12.99, "description": "95%Cotton,5%Spandex, Features: Casual, Short Sleeve, Letter Print,V-Neck,Fashion Tees, The fabric is soft and has some stretch., Occasion: Casual/Office/Beach/School/Home/Street. Season: Spring,Summer,Autumn,Winter.", "category": "women's clothing", "image": "https://fakestoreapi.com/img/61pHAEJ4NML._AC_UX679_.jpg", "rating": { "rate": 3.6, "count": 145 } }</pre> <p>Response headers</p> <pre>content-length: 438</pre>



En este apartado se mostró cómo configurar y utilizar FastAPI para consumir APIs. FastAPI facilita la creación de APIs RESTful y permite consumir APIs de forma sencilla y eficiente. En el siguiente apartado se mostrará cómo realizar pruebas unitarias a las rutas de la API creada en este apartado.

En este apartado se mostró cómo configurar y utilizar FastAPI para consumir APIs. FastAPI facilita la creación de APIs RESTful y permite consumir APIs de forma sencilla y eficiente. En el siguiente apartado se mostrará cómo realizar pruebas unitarias a las rutas de la API creada en este apartado.

21 Integración de APIs de terceros con FastAPI

Part VII

Unidad 7: Desarrollo Avanzado

22 Manejo de Estado en FastAPI, Context API

23 Navegación en FastAPI con FastRouter

24 Aplicación en FastAPI que consume APIs de terceros

Part VIII

Unidad 8: Prácticas Avanzadas de Programación

25 Patrones de Diseño en FastAPI

26 Arquitectura de Software y Diseño Modular

Part IX

Proyecto Final

27

28

29

Part X

Laboratorios

30 Taller Práctico: Implementando una CNN para Reconocimiento de Imágenes (20 minutos)

30.1 Objetivo

En este taller, los estudiantes aprenderán a implementar una Red Neuronal Convolutiva (CNN) básica para realizar tareas de reconocimiento de imágenes utilizando la librería TensorFlow y Keras. Trabajarán con un conjunto de datos preexistente y observarán cómo se entrenan las redes neuronales para clasificar imágenes. Además, se explorarán técnicas de preprocesamiento de datos y se analizará el rendimiento del modelo mediante la evaluación de la precisión en los datos de prueba. Al finalizar el taller, los estudiantes estarán familiarizados con los conceptos básicos de las CNN y podrán aplicarlos en proyectos de visión por computadora.

30.2 Requisitos Previos

- Conocimientos básicos de Python.
- Familiaridad con conceptos de redes neuronales.
- TensorFlow y Keras instalados (si no están instalados, usar el siguiente comando):

```
pip install tensorflow matplotlib
```

30.3 Materiales

- Computadora con acceso a Internet.
- IDE o editor de código (como Jupyter Notebook, VSCode, o Google Colab).
- Conjunto de datos CIFAR-10 (disponible en <https://www.cs.toronto.edu/~kriz/cifar.html>).
- TensorFlow y Keras instalados (si no están instalados, usar el siguiente comando):

30.4 Pasos del Taller

0. Crear el Entorno de Desarrollo

- Crear el archivo creado con jupyter notebook se llame **cnn__taller.ipynb**.

- Si usamos Google Colab, creamos un nuevo cuaderno le damos el nombre **cnn_taller.ipynb**.

1. Configuración del Entorno

Importar las Librerías Necesarias.

Abre tu entorno de desarrollo y ejecuta el siguiente código para importar las librerías:

```
# Importamos las librerías necesarias
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Como podemos observar en el código anterior importamos las librerías necesarias para trabajar con TensorFlow y Keras, así como Matplotlib para visualizar las imágenes.

2. Cargar y Preprocesar el Conjunto de Datos

31 Cargar el Conjunto de Datos CIFAR-10

CIFAR-10 es un conjunto de datos popular que contiene 60,000 imágenes de 10 clases diferentes (como aviones, automóviles, pájaros, gatos, etc.). Carga y divide el conjunto en datos de entrenamiento y prueba:

```
# Cargar el conjunto de datos CIFAR-10
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
train_images, test_images = train_images / 255.0, test_images / 255.0
```

El código anterior carga el conjunto de datos CIFAR-10 y normaliza los valores de píxeles de las imágenes para que estén en el rango [0, 1].

Visualizar Algunas Imágenes de Entrenamiento Para familiarizarse con el conjunto de datos, visualiza algunas imágenes de entrenamiento:

```
# Nombres de las clases en español
class_names = ['avión', 'automóvil', 'pájaro', 'gato', 'ciervo',
               'perro', 'rana', 'caballo', 'barco', 'camión']

# Mostrar las primeras 16 imágenes del conjunto de entrenamiento
plt.figure(figsize=(10,10))
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

El código anterior muestra las primeras 16 imágenes del conjunto de entrenamiento con sus respectivas etiquetas. Cada imagen está asociada con una de las 10 clases en CIFAR-10.

3. Construir la CNN

Definir la Arquitectura de la CNN Implementa una red neuronal convolucional básica con capas convolucionales, de agrupamiento (pooling), y densamente conectadas (fully connected):

```
# Definir la arquitectura mejorada de la CNN
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax') # Asumiendo que hay 10 clases
])
```

El código anterior es muy importante ya que define la arquitectura de la CNN. La red consta de varias capas convolucionales y de agrupamiento, seguidas de capas densamente conectadas. La última capa utiliza una función de activación softmax para clasificar las imágenes en una de las 10 clases posibles.

Los conceptos que necesitamos recordar son:

- **Capa Convolucional (Conv2D):** Aplica un conjunto de filtros a la imagen de entrada para extraer características.
- **Capa de Agrupamiento (MaxPooling2D):** Reduce la dimensionalidad de las características extraídas para mejorar la eficiencia computacional.
- **Capa Densa (Dense):** Conecta todas las neuronas de la capa anterior con las de la capa actual.
- **Función de Activación (ReLU, Softmax):** Introduce no linealidades en la red para aprender patrones complejos y realizar clasificaciones.

Ahora nos vamos a compilar el Modelo Define la función de pérdida, el optimizador, y las métricas que se usarán para entrenar el modelo:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Resumen del modelo
model.summary()
```

El código anterior compila el modelo con el optimizador Adam, la función de pérdida de entropía cruzada categórica escasa (sparse categorical crossentropy), y la métrica de precisión. También muestra un resumen de la arquitectura de la CNN.

4. Entrenar y Evaluar la CNN

Entrenar el Modelo Entrena la CNN utilizando los datos de entrenamiento:

```
# Definir callbacks
callbacks = [
    tf.keras.callbacks.EarlyStopping(patience=3, monitor='val_loss', restore_best_weights=True),
    tf.keras.callbacks.ModelCheckpoint('best_model.keras', save_best_only=True, monitor='val_loss'),
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2, min_lr=0.0001)
]

# Entrenar el modelo
history = model.fit(
    train_images, train_labels,
    epochs=30, # Aumentar el número de epochs para un mejor entrenamiento
    validation_data=(test_images, test_labels),
    batch_size=64, # Ajustar el tamaño del lote para optimizar el uso de la GPU
    callbacks=callbacks
)
```

El código anterior entrena la CNN durante 30 épocas y utiliza un conjunto de validación para monitorear el rendimiento del modelo. También incluye callbacks para detener el entrenamiento temprano, guardar el mejor modelo, y ajustar la tasa de aprendizaje.

Evaluar el Modelo Evalúa el rendimiento del modelo en los datos de prueba:

```
# Evaluar el modelo
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'\nPrecisión en los datos de prueba: {test_acc:.2%}')
```

El código anterior evalúa la precisión del modelo en los datos de prueba y muestra el resultado en porcentaje.

5. Análisis de Resultados

Visualizar el Progreso del Entrenamiento Muestra cómo la precisión y la pérdida cambian durante el entrenamiento:

```
# Graficar la precisión durante el entrenamiento
plt.plot(history.history['accuracy'], label='Precisión en el entrenamiento')
plt.plot(history.history['val_accuracy'], label='Precisión en la validación')
plt.xlabel('Época')
plt.ylabel('Precisión')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()
```

El código anterior muestra cómo la precisión en el entrenamiento y la validación cambian a lo largo de las épocas. Idealmente, queremos que la precisión en el conjunto de validación aumente y se mantenga estable.

Hacer Predicciones Utiliza el modelo para predecir las etiquetas de las imágenes de prueba:

```

# Hacer predicciones
predictions = model.predict(test_images)

def plot_image(i, predictions_array, true_label, img):
    predictions_array = np.array(predictions_array)
    true_label = np.array(true_label)
    img = np.array(img)

    # Asegúrate de que las entradas sean escalares
    predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]
    true_label = true_label.item()
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    predicted_label = predicted_label.item()

    color = 'blue' if predicted_label == true_label else 'red'

    plt.xlabel(f"{class_names[int(predicted_label)]} {100*np.max(predictions_array):2.0f}%")

def plot_value_array(i, predictions_array, true_label):
    predictions_array = np.array(predictions_array)
    true_label = np.array(true_label)

    predictions_array, true_label = predictions_array[i], true_label[i]
    true_label = true_label.item()

    plt.grid(False)
    plt.xticks(range(len(class_names)))
    plt.yticks([])
    thisplot = plt.bar(range(len(class_names)), predictions_array, color="#777777")
    plt.ylim([0, 1])

    predicted_label = np.argmax(predictions_array)
    predicted_label = predicted_label.item()

    thisplot[int(predicted_label)].set_color('red')
    thisplot[int(true_label)].set_color('blue')

```

El código anterior lo analizaremos paso a paso:

- **plot_image**: Muestra una imagen con su predicción y etiqueta verdadera.
- **plot_value_array**: Muestra un gráfico de barras con las probabilidades de predicción para cada clase.

El mismo nos permite visualizar las predicciones del modelo en las imágenes de prueba.

Ahora para visualizar una imagen con su predicción:

```
# Mostrar predicciones para la primera imagen de prueba
i = 1
plt.figure(figsize=(12, 6)) # Ajusta el tamaño de la figura si es necesario
plt.subplot(1, 2, 1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1, 2, 2)
plot_value_array(i, predictions, test_labels)
plt.show()
```

Podemos cambiar el valor de `i` para visualizar diferentes imágenes y sus predicciones.

Por ejemplo si queremos visualizar la imagen 10:

```
# Mostrar predicciones para la décima imagen de prueba
i = 10
plt.figure(figsize=(12, 6)) # Ajusta el tamaño de la figura si es necesario
plt.subplot(1, 2, 1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1, 2, 2)
plot_value_array(i, predictions, test_labels)
plt.show()
```

32 Reto

- Modifica la arquitectura de la CNN para mejorar la precisión en los datos de prueba.
- Experimenta con diferentes hiperparámetros (como el número de filtros, el tamaño del kernel, la tasa de aprendizaje, etc.) para optimizar el rendimiento del modelo.
- Prueba con diferentes técnicas de regularización (como la disminución de la tasa de aprendizaje, la regularización L2, la eliminación de neuronas, etc.) para evitar el sobreajuste.
- Implementa una red neuronal preentrenada (como VGG16, ResNet, etc.) y compara su rendimiento con la CNN básica.

Conclusión

En este taller, implementaste una CNN básica para el reconocimiento de imágenes y la evaluaste en el conjunto de datos CIFAR-10. Este es un primer paso hacia el uso de redes neuronales para aplicaciones más complejas en visión por computador.

33 Laboratorio de CNN con FastAPI

En este laboratorio, vamos a utilizar el modelo de clasificación de imágenes de CIFAR-10 que creamos en el laboratorio anterior, y vamos a crear una API REST con FastAPI para poder hacer predicciones con el modelo.

33.1 1. Cargar el modelo creado en el laboratorio anterior

Crear el archivo **main.py** con el siguiente contenido:

```
from fastapi import FastAPI, File, UploadFile
import tensorflow as tf
import numpy as np
from PIL import Image
import io

app = FastAPI()

# Cargar el modelo
model = tf.keras.models.load_model('cifar10_model.keras')

@app.post("/predict/")
async def predict(file: UploadFile = File(...)):
    try:
        image = Image.open(io.BytesIO(await file.read()))
        image = image.convert("RGB") # Asegúrate de que la imagen esté en formato RGB
        image = image.resize((32, 32)) # Ajustar tamaño de imagen
        image = np.array(image) / 255.0 # Normalizar
        image = np.expand_dims(image, axis=0) # Añadir dimensión del batch

        predictions = model.predict(image)
        class_idx = np.argmax(predictions, axis=1)[0]
        return {"class_idx": int(class_idx), "confidence": float(np.max(predictions))}
    except Exception as e:
        return {"error": str(e)}

@app.get("/")
def read_root():
    return {"message": "Welcome to pyDay Piura 2024"}
```

33.2 2. Ejecutar la API

Para ejecutar la API, ejecuta el siguiente comando:

```
uvicorn main:app --reload
```

33.3 3. Probar la API

Para probar la API vamos a utilizar la herramienta **curl**. Ejecuta el siguiente comando en una terminal:

```
curl -X POST "http://127.0.0.1:8000/predict/" -H "accept: application/json" -H "Content-T
```

Por ejemplo:

```
curl -X POST "http://127.0.0.1:8000/predict/" -F "file=@/home/statick/workspaces/charlas/"
```

En el caso de que la imagen sea un **avión**, deberías obtener una respuesta similar a la siguiente:

```
{"class_idx": 0, "confidence": 0.9999998807907104}
```

Tip

Recuerda la lista de clases de CIFAR-10:

0. Avión
1. Automóvil
2. Pájaro
3. Gato
4. Ciervo
5. Perro
6. Rana
7. Caballo
8. Barco
9. Camión