

Curso de FastAPI

Diego Saavedra Miguel Amaya

Jul 26, 2024

Table of contents

1	Bienvenido	5
1.1	¿De qué trata este curso?	5
1.2	¿Para quién es este curso?	5
1.3	¿Cómo contribuir?	5
I	Unidad 1: Introducción a Python	7
2	Configuración y Sintaxis básica	8
2.1	Sintaxis básica	9
3	Variables y Control de flujo	11
4	Funciones y Parámetros	14
4.1	Parámetros con valores por defecto	14
4.2	Parámetros con nombre	15
II	Unidad 2: Estructura de Datos en Python	16
5	Listas y Tuplas	17
5.1	Listas	17
5.2	Tuplas	18
6	Diccionarios y Conjuntos	19
6.1	Diccionarios	19
6.2	Conjuntos	20
III	Unidad 3: Programación Orientada a Objetos en Python	21
7	Conceptos Básicos	22
7.1	Clases y Objetos	22
8	Herencia, Polimorfismo y Encapsulación	24
8.1	Herencia	24
8.2	Polimorfismo	25
8.3	Encapsulamiento	26

IV	Unidad 4: Herramientas de Desarrollo	28
9	Git y Github	29
9.1	Git	29
9.1.1	Instalación de Git	29
9.1.2	Comandos básicos de Git	30
9.2	Github	30
9.2.1	Crear una cuenta en Github	31
9.2.2	Crear un repositorio en Github	31
9.2.3	Clonar un repositorio en Github	31
9.2.4	Subir cambios a un repositorio en Github	32
9.2.5	Descargar cambios de un repositorio en Github	32
V	Unidad 5: Introducción a FastAPI	33
10	Configuración y Estructura	34
10.1	Instalación de FastAPI	34
10.2	Estructura de un proyecto FastAPI	35
10.3	Ejecución de un proyecto FastAPI	37
10.4	Documentación de una API FastAPI	37
11	Pydantic en FastAPI	40
11.1	¿Qué es Pydantic?	40
12	Otro Ejemplo	43
13	Modelos en FastAPI	48
VI	Unidad 6: Desarrollo Avanzado en FastAPI	49
14	Rutas y Validaciones en FastAPI	50
15	APIs RESTful con FastAPI	51
16	Pruebas Unitarias en FastAPI	52
17	Optimización y Rendimiento	53
18	Creación de una API de gestión de tareas utilizando FastAPI	54
VII	Unidad 7: Consumo API con FastAPI	55
19	Configuración y Uso de FastAPI para consumir APIs	56
20	Integración de APIs de terceros con FastAPI	57

VIII Unidad 8: Desarrollo Avanzado	58
21 Manejo de Estado en FastAPI, Context API	59
22 Navegación en FastAPI con FastRouter	60
23 Aplicación en FastAPI que consume APIs de terceros	61
IX Unidad 9: Prácticas Avanzadas de Programación	62
24 Patrones de Diseño en FastAPI	63
25 Arquitectura de Software y Diseño Modular	64
X Proyecto Final	65
XI Laboratorios	69
29 Desarrollo del Backend para un E-Commerce con Django Rest Framework	70
29.1 1. Configuración Inicial del Proyecto	70
29.1.1 1.1. Crear un Proyecto Django	70
29.1.2 1.2 Crear una Aplicación Django	70
29.1.3 1.3 Instalar Django Rest Framework	70
29.1.4 1.4 Configurar el Proyecto	70
29.2 2. Definir el Modelo de Datos	71
29.2.1 2.1 Crear Modelos en products/models.py	71
29.2.2 2.2 Crear y Aplicar Migraciones	71
29.3 3. Crear Serializers	71
29.3.1 3.1 Definir Serializers en products/serializers.py	71
29.4 4. Crear Vistas y Rutas	72
29.4.1 4.1 Definir Vistas en products/views.py	72
29.5 4.2 Configurar Rutas en products/urls.py	73
29.6 4.3 Incluir las URLs en ecommerce_project/urls.py	73
29.7 5. Probar la API	73
29.7.1 5.1 Ejecutar el Servidor de Desarrollo	73
29.8 5.2 Probar los Endpoints	74
30 Extra	75
30.1 1. Instalar Django Rest Swagger	75
30.2 2. Configurar Django Rest Swagger	75
30.3 3. Configurar las URLs	76
30.4 4. Probar la Documentación	76

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

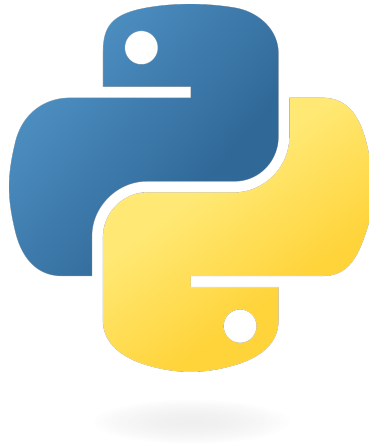


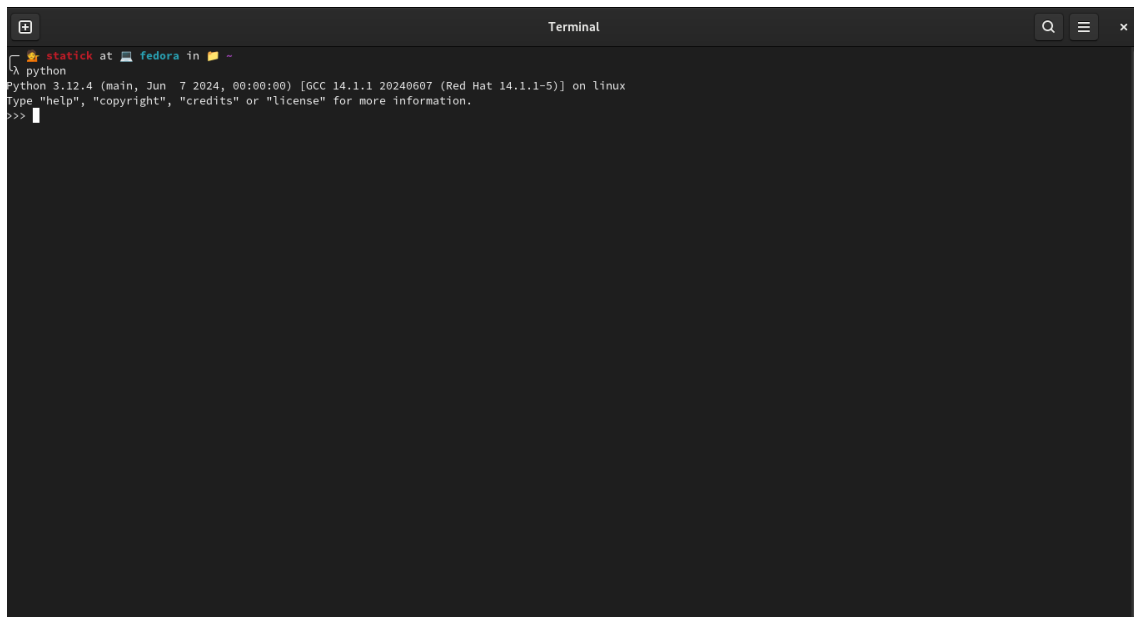
Figure 2.1: Python

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 a user named "static" at a machine named "fedora". The user has entered the command "python". The terminal displays the Python 3.12.4 startup banner, which includes the version, release date (Jun 7 2024), GCC version (14.1.1 20240607), and the fact that it's on Linux. It also provides instructions to type "help", "copyright", "credits", or "license" for more information. The prompt is now ">>>" with a cursor.

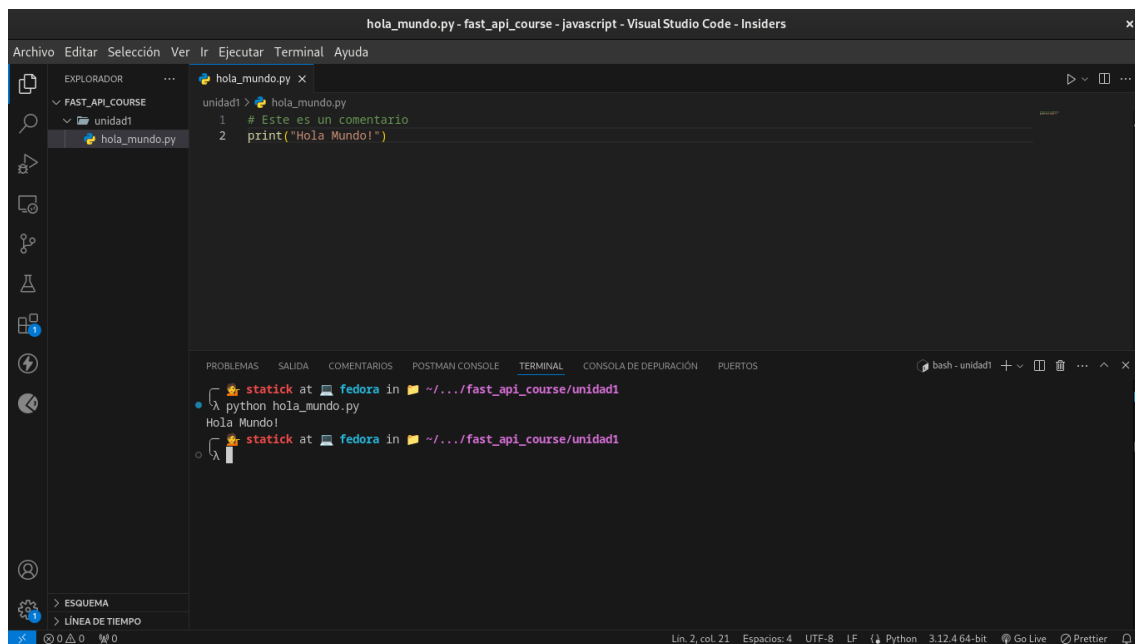
Para salir del interprete de python puedes escribir `exit()` o **Ctrl + D**.

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

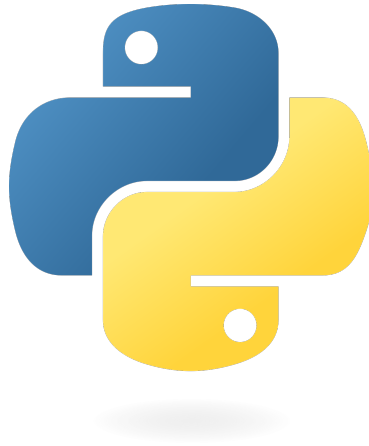


Figure 3.1: Python

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

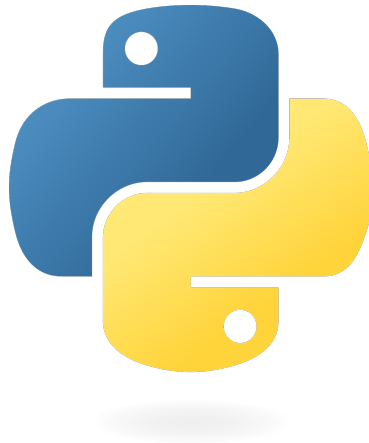


Figure 4.1: Python

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

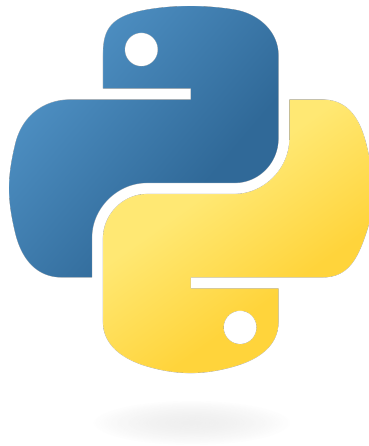


Figure 5.1: Python

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 Dicionarios y Conjuntos

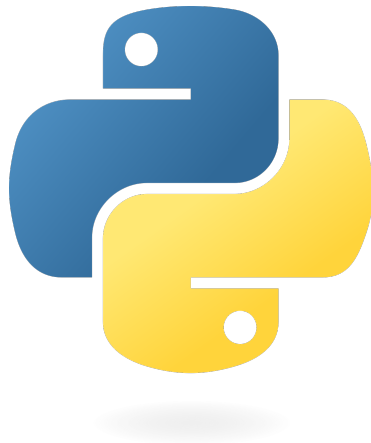


Figure 6.1: Python

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

6.1 Dicionarios

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

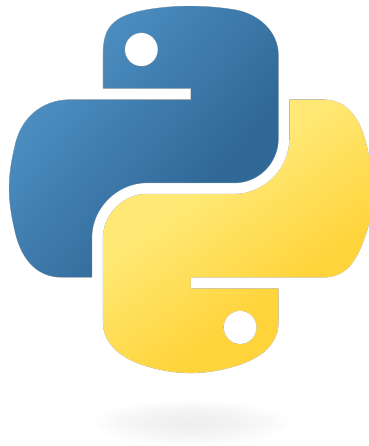


Figure 7.1: Python

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

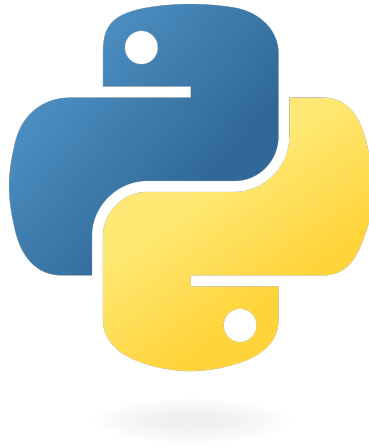


Figure 8.1: Python

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

Después 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:

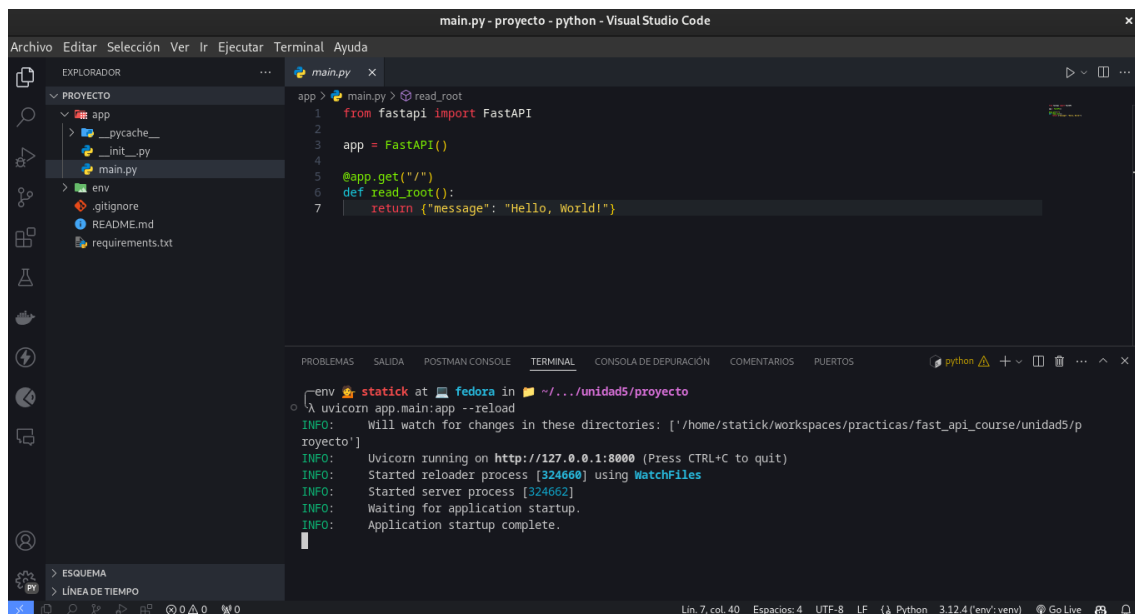
```
proyecto/  
  
  app/  
    __init__.py  
    main.py  
  
  .gitignore  
  README.md  
  requirements.txt
```

- **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:



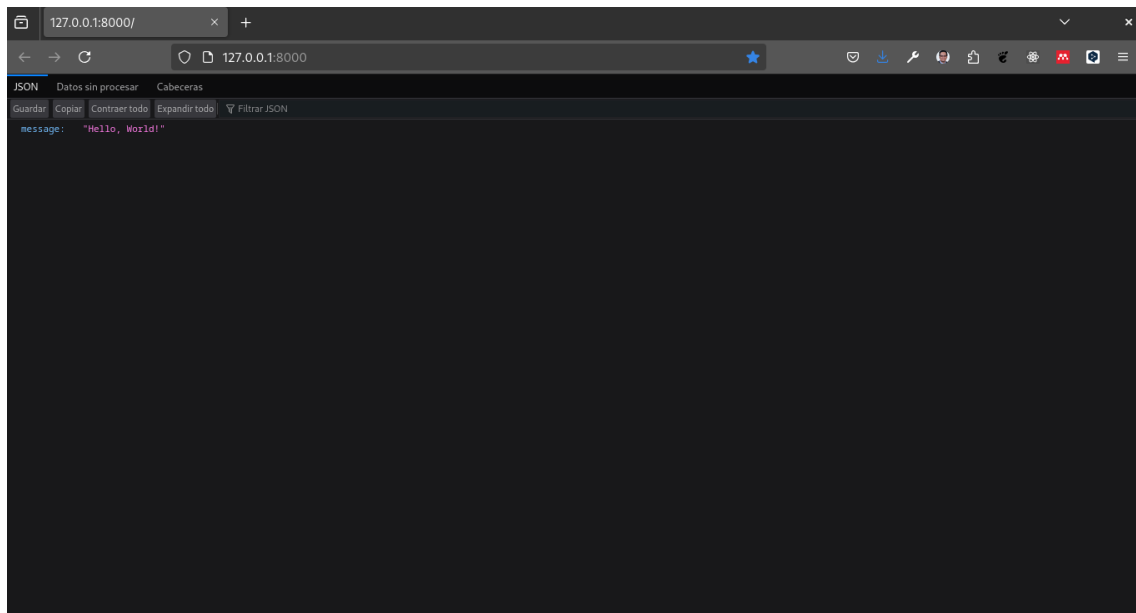
```
from fastapi import FastAPI

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.

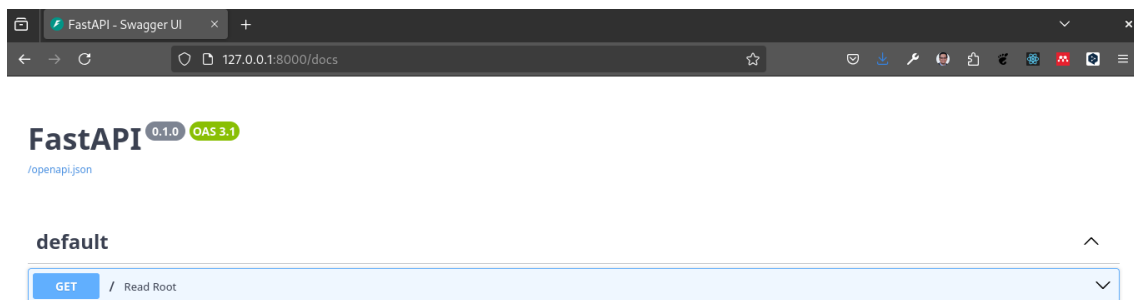
Al ejecutar el comando anterior se inicia el servidor web y se muestra la URL de la API en la consola. Para acceder a la API se utiliza la URL que se muestra en la consola.

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**.

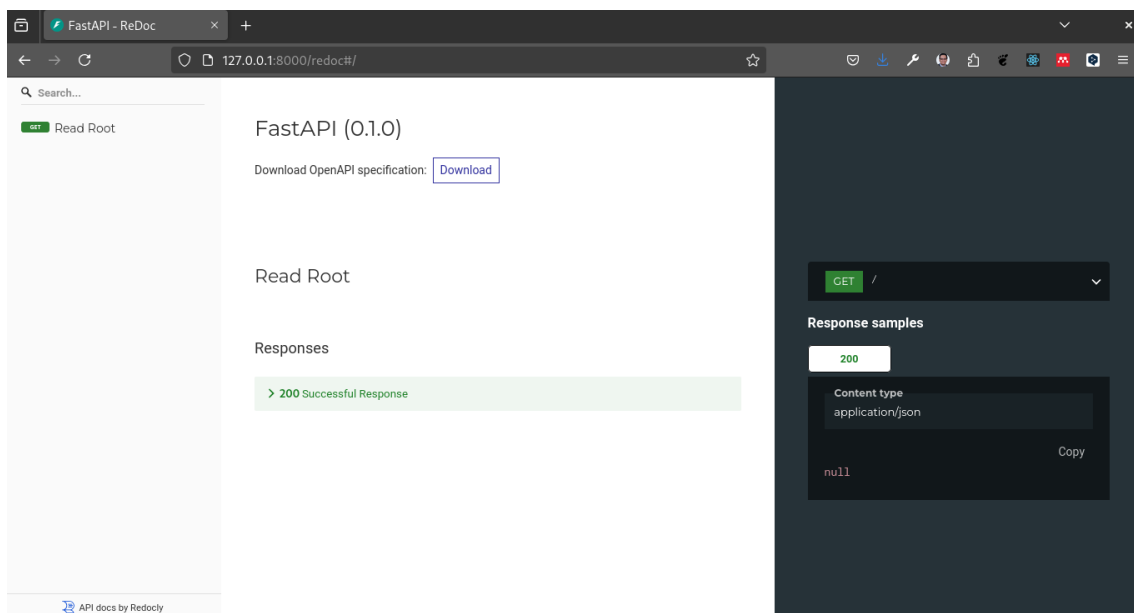
En la interfaz de usuario se muestra una lista de las rutas de la API y las operaciones que se realizan en cada ruta. Para probar una operación se hace clic en la operación y se ingresan los parámetros de la operación.

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

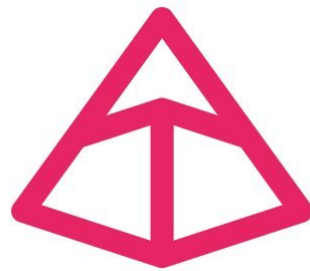


Figure 11.1: Pydantic

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
```

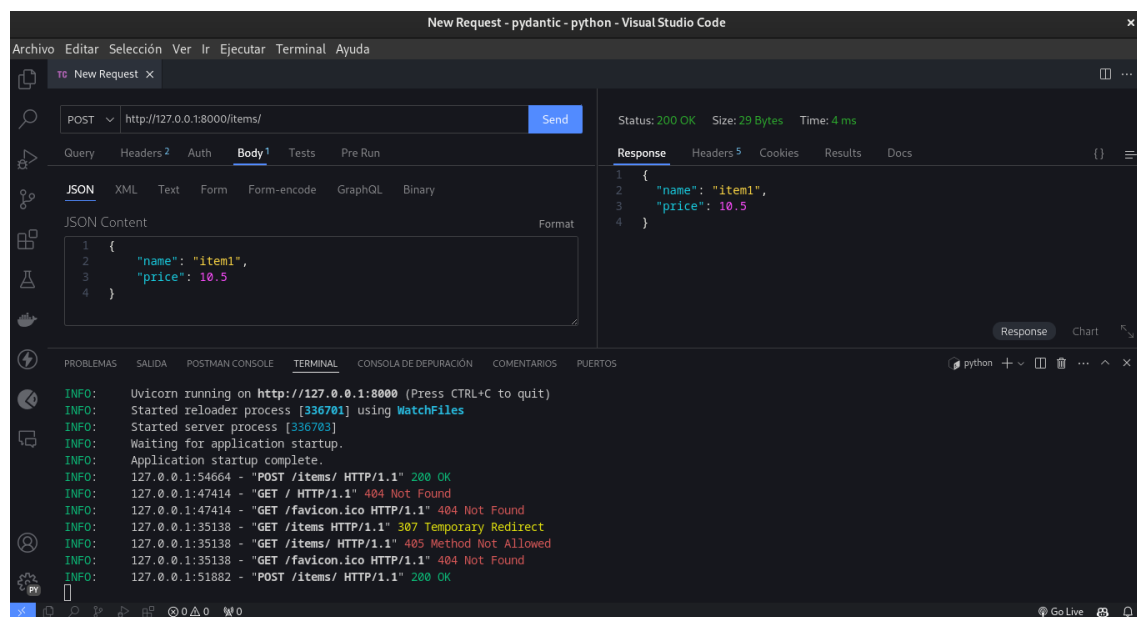
Para probar la operación `create_item()` se puede utilizar una herramienta como **Tunder Client** o **Postman**. A continuación se muestra un ejemplo de cómo probar la operación `create_item()` con **Tunder Client**:

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**:

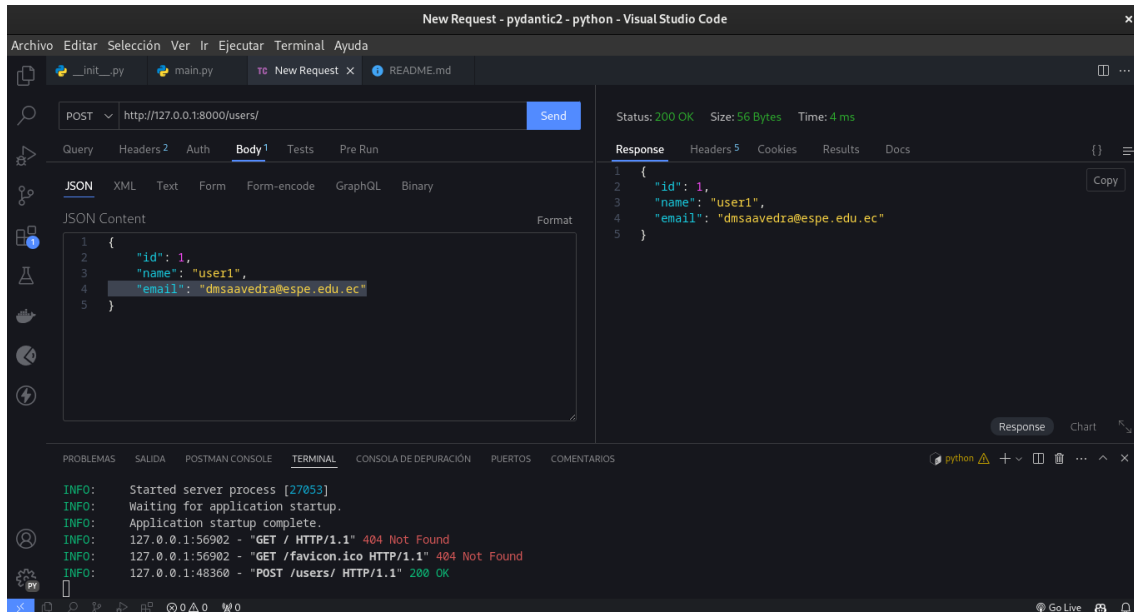
```

POST /users/
Content-Type: application/json

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

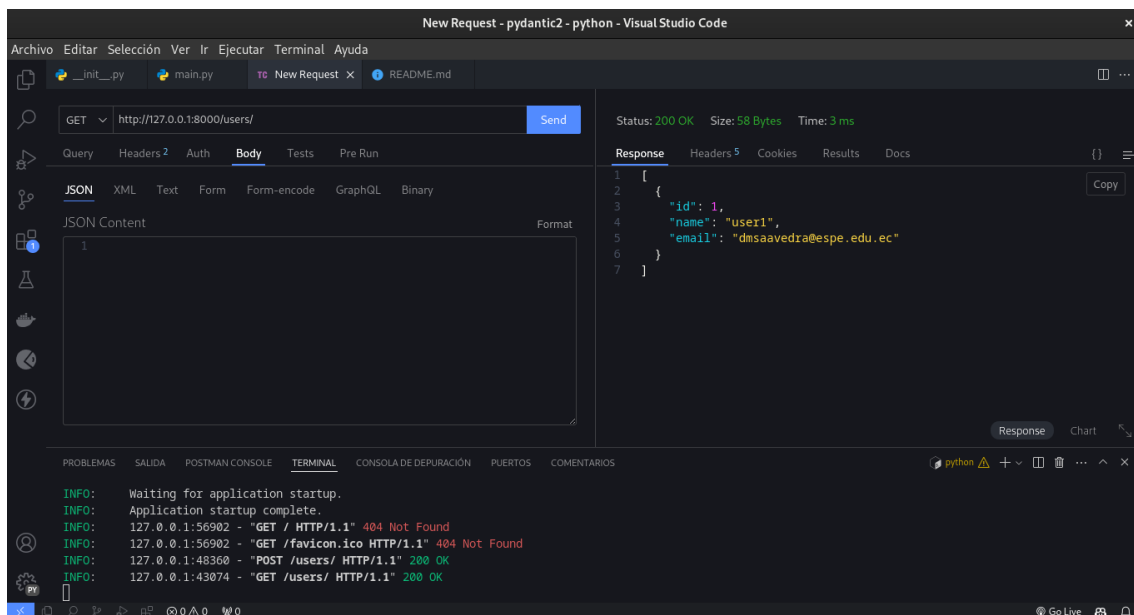
```

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



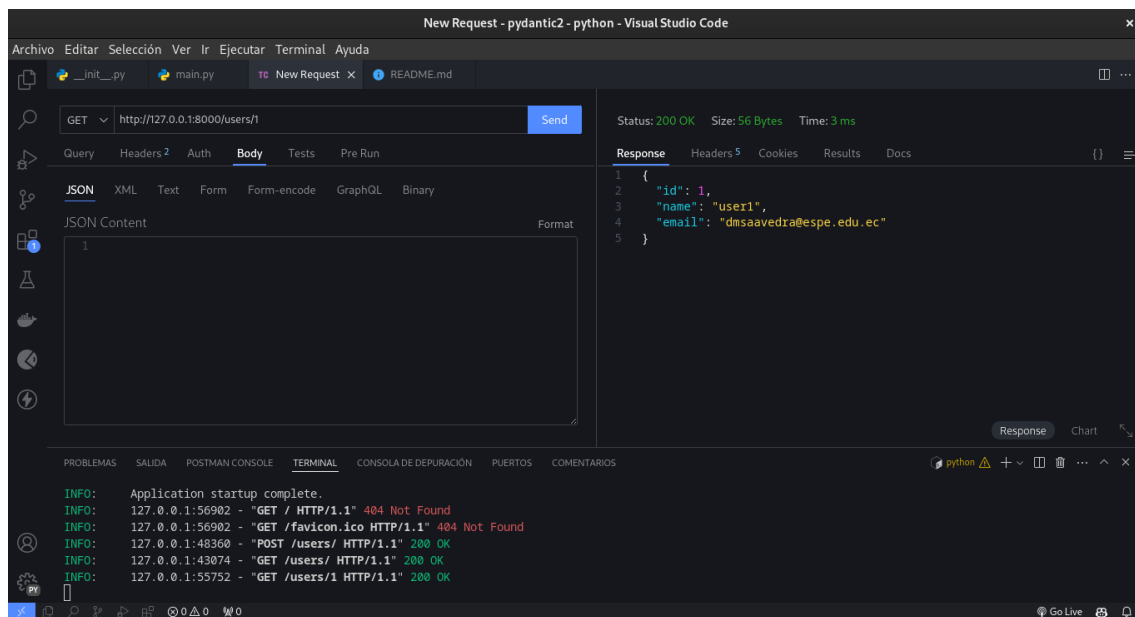
Para obtener todos los usuarios se realiza una solicitud **GET** a la ruta `/users/`:

```
GET /users/
```



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

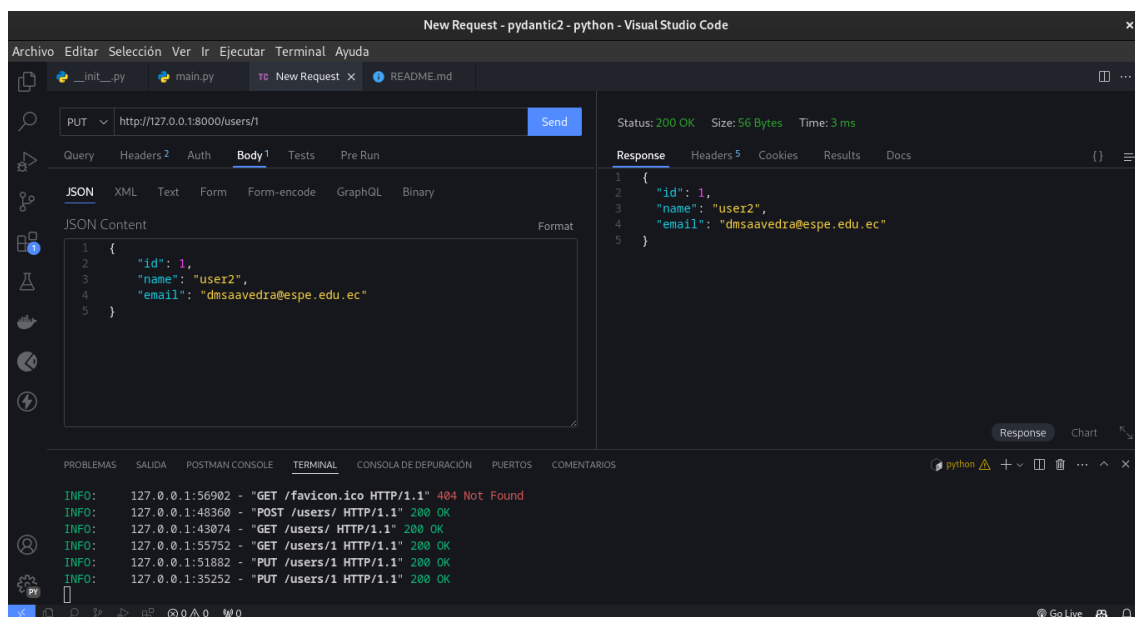
```
GET /users/1
```



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

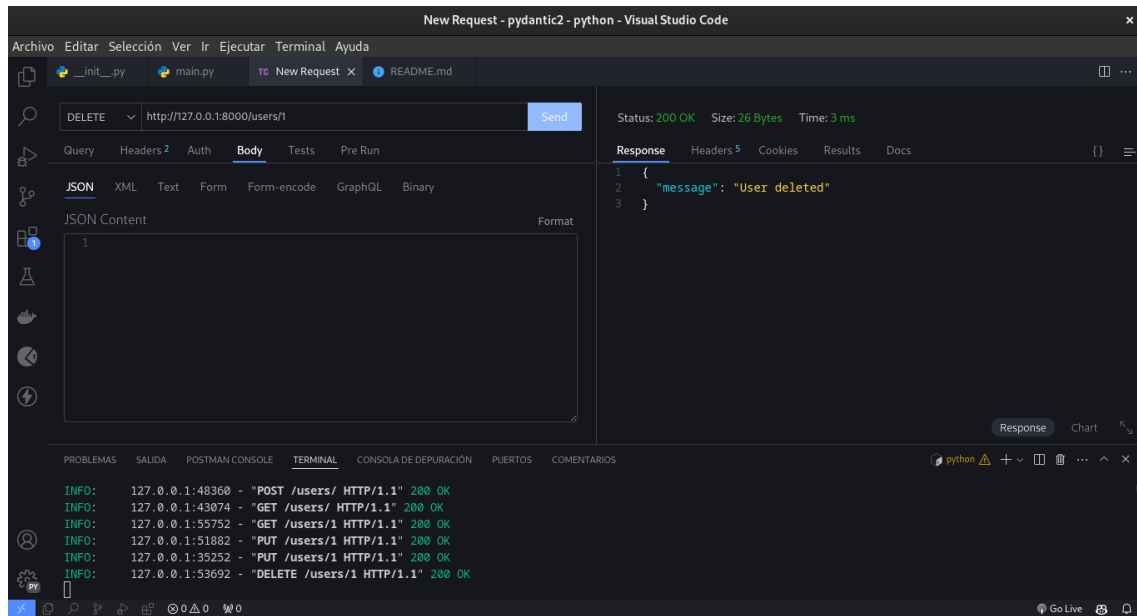
```
PUT /users/1
Content-Type: application/json

{
  "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

Part VI

Unidad 6: Desarrollo Avanzado en FastAPI

14 Rutas y Validaciones en FastAPI

15 APIs RESTful con FastAPI

16 Pruebas Unitarias en FastAPI

17 Optimización y Rendimiento

18 Creación de una API de gestión de tareas utilizando FastAPI

Part VII

Unidad 7: Consumo API con FastAPI

19 Configuración y Uso de FastAPI para consumir APIs

20 Integración de APIs de terceros con FastAPI

Part VIII

Unidad 8: Desarrollo Avanzado

21 Manejo de Estado en FastAPI, Context API

22 Navegación en FastAPI con FastRouter

23 Aplicación en FastAPI que consume APIs de terceros

Part IX

Unidad 9: Prácticas Avanzadas de Programación

24 Patrones de Diseño en FastAPI

25 Arquitectura de Software y Diseño Modular

Part X

Proyecto Final

26

27

28

Part XI

Laboratorios

29 Desarrollo del Backend para un E-Commerce con Django Rest Framework

29.1 1. Configuración Inicial del Proyecto

29.1.1 1.1. Crear un Proyecto Django

Abre tu terminal y ejecuta los siguientes comandos para crear un nuevo proyecto Django:

```
python -m venv env
source env/bin/activate
pip install django==4.2
django-admin startproject ecommerce_project .
cd ecommerce_project
```

29.1.2 1.2 Crear una Aplicación Django

Dentro del directorio del proyecto, crea una aplicación para manejar el e-commerce:

```
python manage.py startapp products
```

29.1.3 1.3 Instalar Django Rest Framework

Instala DRF usando pip:

```
pip install djangorestframework
```

29.1.4 1.4 Configurar el Proyecto

Añade 'rest_framework' y tu nueva aplicación 'products' a la lista `INSTALLED_APPS` en `ecommerce_project/settings.py`:

```
INSTALLED_APPS = [
    # ... otras apps
    'rest_framework',
    'products',
]
```

29.2 2. Definir el Modelo de Datos

29.2.1 2.1 Crear Modelos en products/models.py

Define los modelos para el e-commerce, como Product, Category, y Order:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    category = models.ForeignKey(Category, related_name='products', on_delete=models.CASCADE)
    stock = models.PositiveIntegerField()

    def __str__(self):
        return self.name

class Order(models.Model):
    product = models.ForeignKey(Product, related_name='orders', on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField()
    total_price = models.DecimalField(max_digits=10, decimal_places=2)
    order_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Order {self.id} - {self.product.name}"
```

29.2.2 2.2 Crear y Aplicar Migraciones

Genera y aplica las migraciones para los modelos:

```
python manage.py makemigrations
python manage.py migrate
```

29.3 3. Crear Serializers

29.3.1 3.1 Definir Serializers en products/serializers.py

Los serializers se encargan de transformar los modelos en formatos JSON y viceversa:

```

from rest_framework import serializers
from .models import Category, Product, Order

class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = '__all__'

class ProductSerializer(serializers.ModelSerializer):
    category = CategorySerializer()

    class Meta:
        model = Product
        fields = '__all__'

class OrderSerializer(serializers.ModelSerializer):
    product = ProductSerializer()

    class Meta:
        model = Order
        fields = '__all__'

```

29.4 4. Crear Vistas y Rutas

29.4.1 4.1 Definir Vistas en products/views.py

Utiliza las vistas basadas en clases de DRF para crear y manejar las operaciones CRUD:

```

from rest_framework import generics
from .models import Category, Product, Order
from .serializers import CategorySerializer, ProductSerializer, OrderSerializer

class CategoryListCreate(generics.ListCreateAPIView):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

class CategoryDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

class ProductListCreate(generics.ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

class ProductDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Product.objects.all()

```



```

        serializer_class = ProductSerializer

class OrderListCreate(generics.ListCreateAPIView):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer

class OrderDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer

```

29.5 4.2 Configurar Rutas en products/urls.py

Define las rutas para acceder a las vistas:

```

from django.urls import path
from . import views

urlpatterns = [
    path('categories/', views.CategoryListCreate.as_view(), name='category-list-create'),
    path('categories/<int:pk>/', views.CategoryDetail.as_view(), name='category-detail'),
    path('products/', views.ProductListCreate.as_view(), name='product-list-create'),
    path('products/<int:pk>/', views.ProductDetail.as_view(), name='product-detail'),
    path('orders/', views.OrderListCreate.as_view(), name='order-list-create'),
    path('orders/<int:pk>/', views.OrderDetail.as_view(), name='order-detail'),
]

```

29.6 4.3 Incluir las URLs en ecommerce_project/urls.py

Añade las URLs de la aplicación al archivo principal de URLs del proyecto:

```

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('products.urls')),
]

```

29.7 5. Probar la API

29.7.1 5.1 Ejecutar el Servidor de Desarrollo

Inicia el servidor de desarrollo de Django:

```
python manage.py runserver
```

29.8 5.2 Probar los Endpoints

Utiliza herramientas como Postman o cURL para probar los endpoints:

- Listar categorías: GET /api/categories/
- Crear categoría: POST /api/categories/
- Obtener categoría específica: GET /api/categories/{id}/
- Actualizar categoría: PUT /api/categories/{id}/
- Eliminar categoría: DELETE /api/categories/{id}/

Y lo mismo para productos y pedidos.

- Listar productos: GET /api/products/
- Crear producto: POST /api/products/
- Obtener producto específico: GET /api/products/{id}/
- Actualizar producto: PUT /api/products/{id}/
- Eliminar producto: DELETE /api/products/{id}/

30 Extra

Agreguemos Swagger a nuestro proyecto para tener una documentación de nuestra API.

30.1 1. Instalar Django Rest Swagger

Instala Django Rest Swagger usando pip:

```
pip install drf-yasg
```

30.2 2. Configurar Django Rest Swagger

Añade 'rest_framework_swagger' a la lista INSTALLED_APPS en ecommerce_project/settings.py:

```
from django.contrib import admin
from django.urls import path, include
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="E-commerce API",
        default_version='v1',
        description="API documentation for the E-commerce project",
        terms_of_service="https://www.google.com/policies/terms/",
        contact=openapi.Contact(email="contact@ecommerce.local"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('products.urls')),
    path('docs/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-u'),
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
]
```

30.3 3. Configurar las URLs

Añade las URLs de Swagger al archivo principal de URLs del proyecto:

```
'''
from rest_framework_swagger.views import get_swagger_view

schema_view = get_swagger_view(title='E-Commerce API')

urlpatterns = [
    '''
    path('docs/', schema_view),
]
```

Tip

Para evitar un error común es necesario instalar **setuptools** con el siguiente comando:

```
pip install setuptools
```

Finalmente es necesario agregar el siguiente código al final del archivo settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'drf_yasg',
    'rest_framework',
    'products',
]

'''
REST_FRAMEWORK = {
    'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema'
}

CORS_ALLOWED_ORIGINS = [
    "http://localhost:8000",
    "http://127.0.0.1:8000",
    # Añade otros orígenes permitidos aquí
]
```

30.4 4. Probar la Documentación