Jul. 2022

Unidad 7: Servicios REST

Introducción

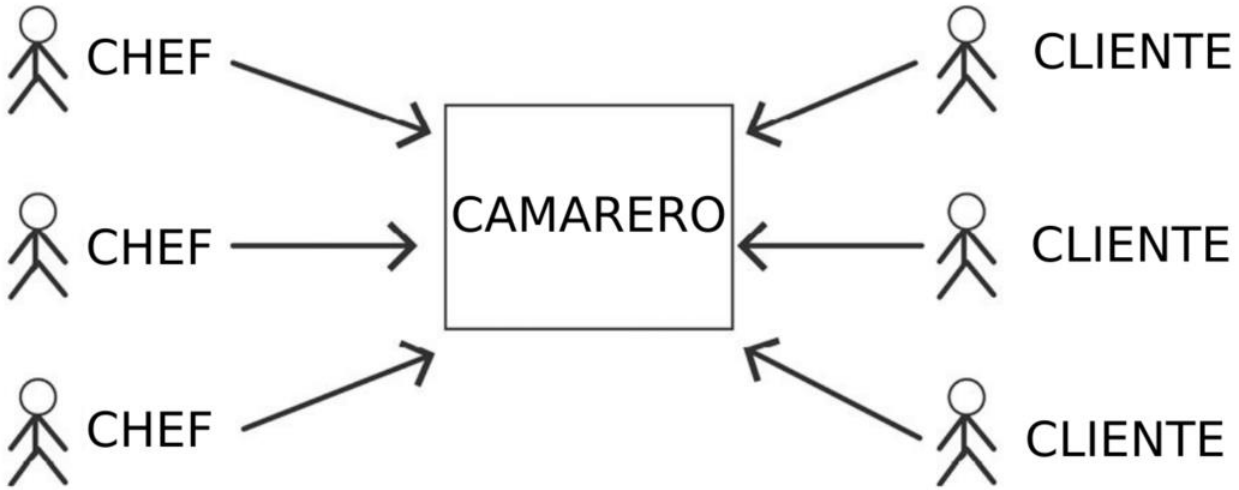
- Estamos en la era de Internet, un mundo donde todo está conectado.
- Los datos fluyen sin problemas de un lugar a otro.
- Desde en un sitio web podemos obtener toda la información del mundo con unos pocos clics.
- Detrás de todo ese intercambio de datos/información esta la API.
- En esta unidad, aprenderemos todo el proceso de desarrollo de aplicaciones web
  - Desde conceptos: formato JSON, protocolo HTTP, códigos de estado HTTP, servicio web, una API y REST.
  - Hasta codificar y comunicar los datos entre el frontend y el backend.
- El trabajo de desarrollo será verificado y probado usando Postman

Objetivos de aprendizaje

- Comprender conceptos de la API RESTful
- Entender el significado de los diferentes métodos y estados HTTP
- Construir una API RESTful y ejecutar CRUD
- Utilizar mensajes JSON para comunicarse con las APIs
- Probar los puntos finales de la API utilizando Postman

La interfaz de programación de aplicaciones (API)

- Una interfaz para que el sitio web se comunique con la lógica del backend.
- Encapsula la lógica para que la gente de fuera no pueda verla



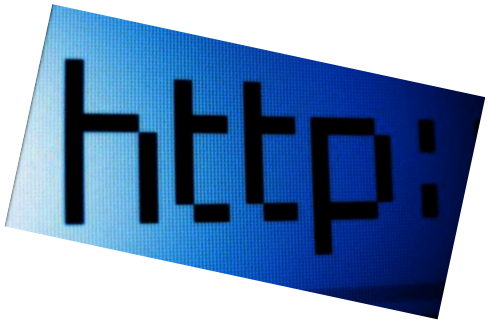
Interfaz de Transferencia de Estados Representacional *(API RESTful)*

- Se definió en la tesis de Roy Fielding en el año 2000.
  - disertación considerada como la biblia en el ámbito de la web
- No es un estándar Ni un protocolo
  - REST es un estilo arquitectónico de software
- Útil para construir aplicaciones escalables
  - que sirven enormes cantidades de tráfico cada segundo
- API RESTful es una API que se ajusta a las restricciones/principios REST

5 Restricciones/Principios REST

1. Cliente-servidor:
   - Existe una interfaz entre el cliente y el servidor a través de la cual se comunican.
   - Puede ser sustituido cualquiera de los dos lados siempre que la interfaz siga siendo la misma.
   - Las solicitudes siempre provienen del lado del cliente.
2. Sin estado:
   - No existe el concepto de estado para una solicitud.
   - Cada solicitud se considera independiente y completa.
   - Para mantener el estado de la conexión, no hay dependencia ni de petición anterior ni de una sesión
3. Almacenable en caché:
   - Las cosas se pueden almacenar en caché para mejorar el rendimiento.
4. Sistema por capas:
   - Puede haber múltiples capas en el sistema, con el objetivo de ocultar la lógica/los recursos reales.
   - Las capas realizan diferentes funciones (almacenamiento en caché, la codificación, etc).
5. Interfaz uniforme:
   - Desacoplar la lógica del cliente y del servidor.

Protocolo de Transferencia de Hipertexto



- HTTP es una implementación REST
- Es el protocolo estándar utilizado en la web mundial (prefijo http).
  - Lo utilizamos todos los días para navegar por diferentes sitios web.
- En la interacción del *frontend* con la API del *backend* se necesita definir el método HTTP.
- Los métodos HTTP son como los verbos de la API REST.

  ~ realizar diferentes acciones sobre los datos

métodos HTTP	Función
GET	Leer datos
POST	Crear datos
PUT	Sustituir datos
PATCH	Actualizar datos modificando atributos
DELETE	Eliminar datos

Métodos HTTP y CRUD

- Create Read Update Delete (CRUD)
- CRUD modela el ciclo de vida de la gestión de registros de la base de datos
- Acciones relacionadas con los métodos HTTP

métodos HTTP	Acción	Descripción
GET	Leer/Recuperar	Recuperar un recurso
POST	Crear	Crear un recurso
PUT	Actualizar	Actualizar un recurso
DELETE	Eliminar	Eliminar un recurso

Códigos de estado HTTP

- Cada respuesta HTTP del servidor contiene un código de estado.
  - Ayuda a entender el estado de su solicitud (éxito o fracaso)
- Códigos de estado HTTP más utilizados

Código	Significado	la solicitud
200	OK	GET, PUT o PATCH GET, PUT ha sido exitosa
201	Creado	POST ha sido exitosa
204	Sin contenido	DELETE ha sido exitosa
400	Petición mala	hay algo incorrecto en la solicitud
401	No autorizado	le faltan detalles de autenticación
403	Prohibido	el recurso solicitado está prohibido
404	No encontrado	el recurso solicitado no existe

Framework Web

- Un framework web es un conjunto de herramientas
  (librerías, funciones, clases, etc.)
  que podemos utilizar para codificar un sitio web
  centrandonos únicamente en la lógica de la aplicación
- Principales Frameworks para el desarrollo web:

El framework web **Flask**

- Se considera un micro-marco web que sólo proporciona los paquetes absolutamente necesarios para construir aplicaciones web con funcionalidades como:
  - interactuar con las peticiones de los clientes
  - enrutar las URL a los recursos
  - renderizar las páginas web
  - interactuar con las bases de datos
- Flask es:
  - minimalista
  - fácil de aprender

flask.palletsprojects.com

Instalar Flask

1. Instalar Flask (Flask-2.1.3) – Jul,2022
2. Verificar

```
pip install flask
>>> import flask
>>> from flask import Flask
>>> flask
```

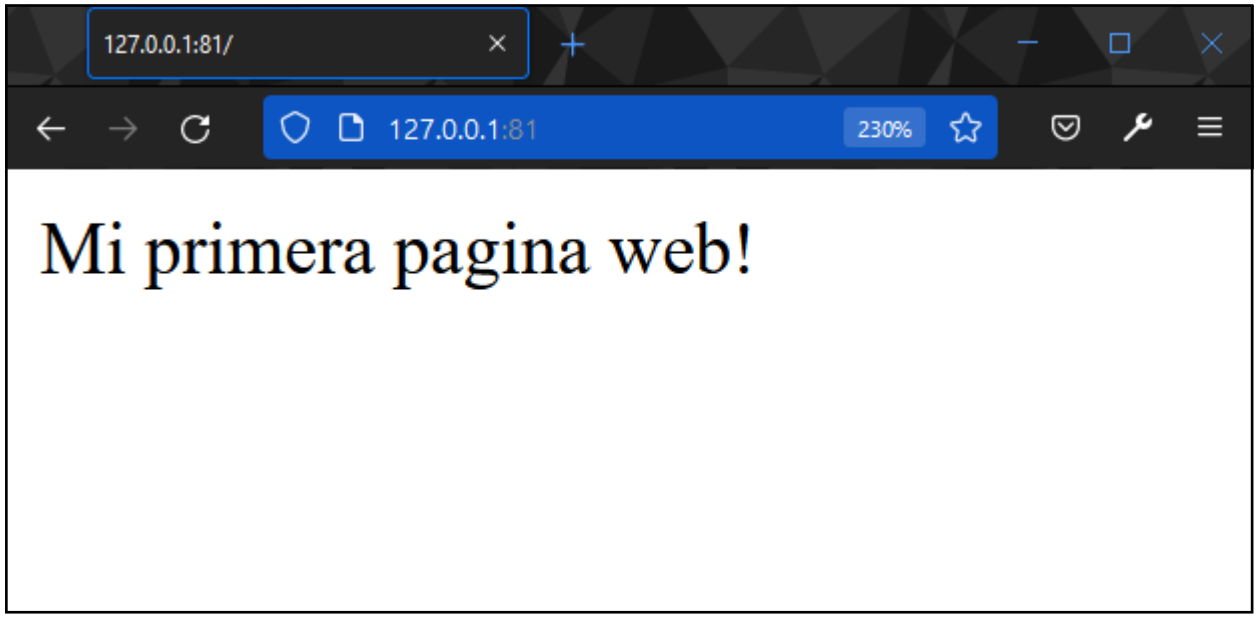
Hola mundo en Flask

1. Creamos una App instancia de Flask
2. Creamos una ruta que llame a una función
3. Creamos la función
4. Ejecutamos la app en el servidor

```
>>> app=Flask(__name__)
>>> @app.route('/')
... def index():
...     return "Mi primera pagina web!"
...
>>> app.run(host='0.0.0.0', port=81)
```

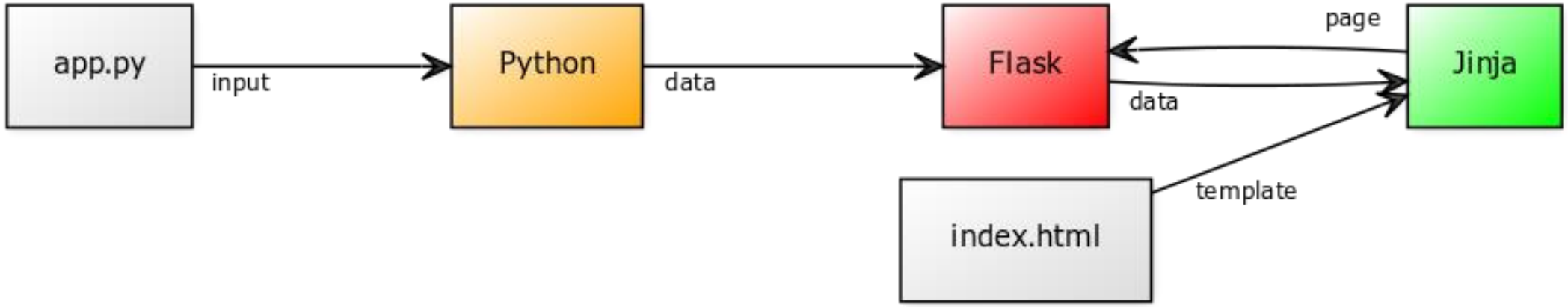
Hola mundo en Flask (cont.)

- En el navegador web ir a la url http://localhost:81/



Incluir elementos en una plantilla

- Jinja2 permite incluir en la plantilla:
  - variables, sentencias *if* y bucles



Variables en la plantilla.html de flask

- Utilizan marcadores de posición que están entre estos símbolos

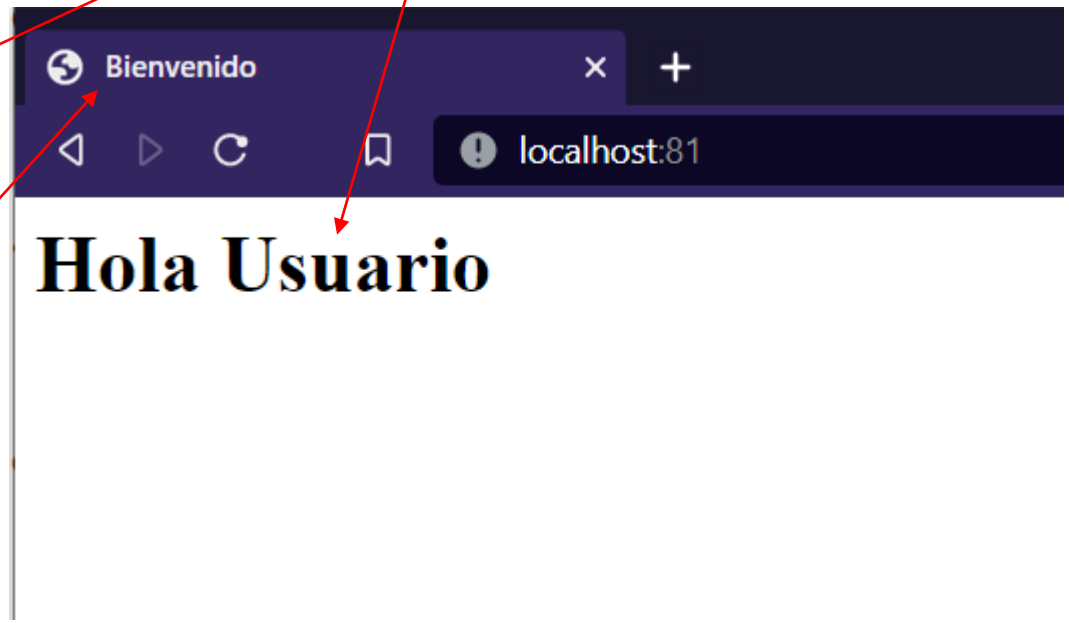
  {{ variable }}

app.py

```
from flask import
render_template

…

nombre = “Usuario”

return render_template(
          'index.html’,
          titulo='Bienvenido’,
          nombre_alumno=nombre)
```

templates/index.html

```
<h1>Hola {{ nombre_alumno }}</h1>
```



Sentencia *if* en la plantilla

Se pueden utilizar los mismos operadores (==, >, <, >=, <=)

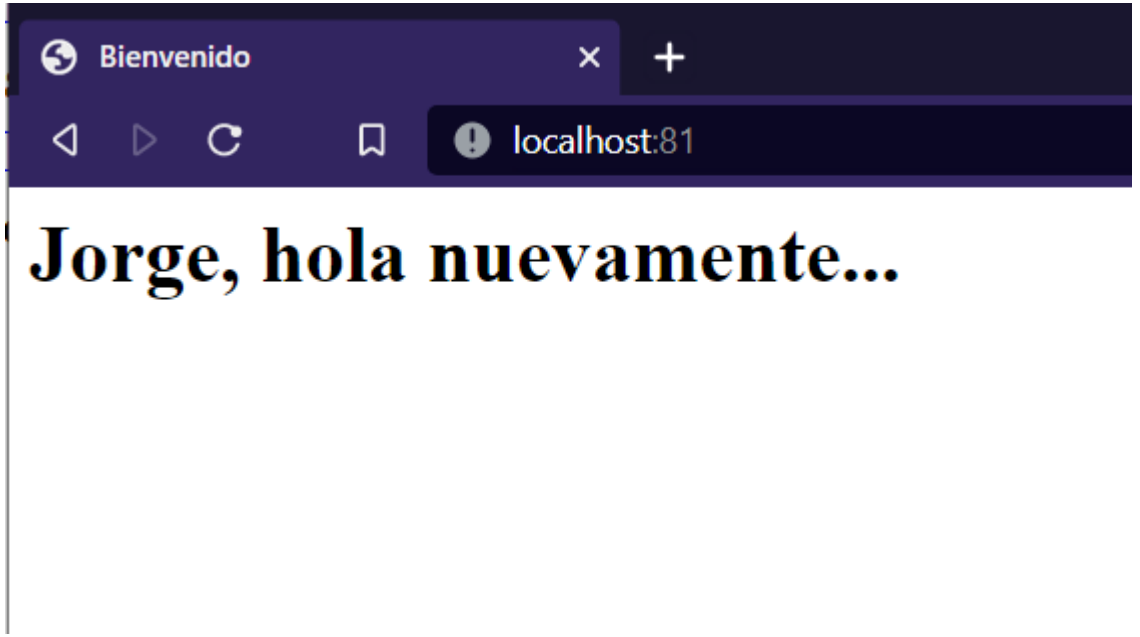
y cláusulas como *else* o *elif,* utilizando {% operador %}

app.py

```
nombre = “Jorge”
return render_template(
        'index.html’,
        titulo='Bienvenido’,
        nombre_alumno=nombre)
```

templates/index.html

```
{% if nombre_alumno == "Jorge": %}
  <h1>Jorge, hola nuevamente... </h1>
{% else %}
  <h1>Hola {{nombre_alumno}} </h1>
{% endif %}
```



Bucle *for* en la plantilla

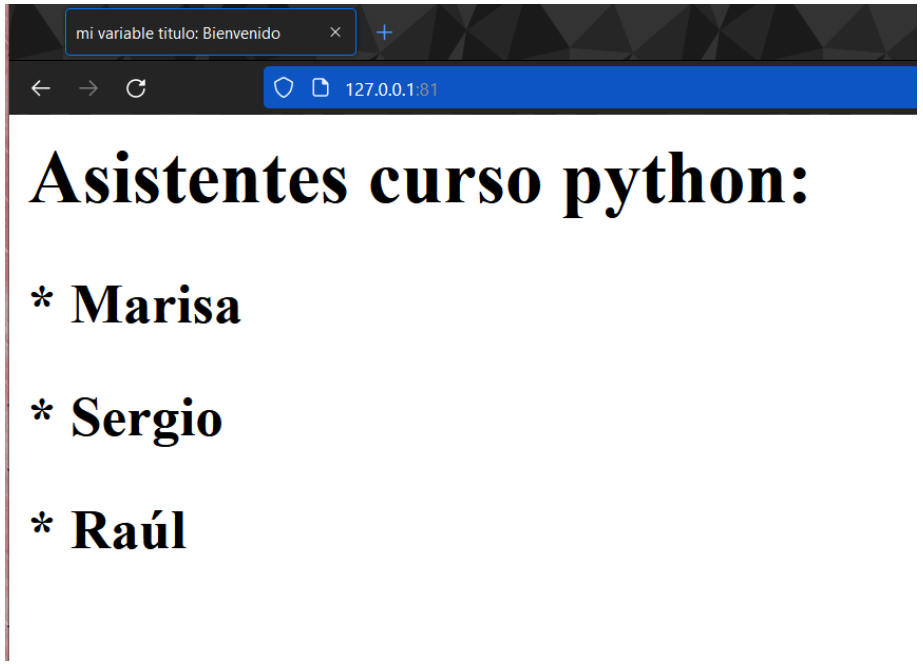
Para grandes cantidades de datos utilizando {% operador %}

app.py

```
alumnos = [“Marisa",“Sergio",“Raúl"]
return render_template(
          'index.html’,
          titulo='Bienvenido’,
          elementos=alumnos)
```

templates/index.html

```
{% for elemento in elementos: %}
  <h1>{{ elemento }}</h1>
{% endfor %}
```



Enrutamiento

- Técnica presente en las aplicaciones web modernas
- Ayuda al usuario a recordar las URLs
  - /reserva.php -> /reserva/
  - /transaccion.asp?id=1234/ -> /transaccion/1234/
- Para vincular una URL a una función se usa el decorador route(),

```
@app.route()
```

- La ruta `/hola` vinculada a la función

```
hola_mundo()
```

La salida de la función `hola_mundo()` se muestra en el navegador.

```
@app.route('/hola')
def hola_mundo():
    return "hola mundo"
```

Enrutamiento con peticiones HTTP POST

- Crea una plantilla llamada **login.html**

```
<body>
  <form action = "http://localhost:5000/login" method = "post">
     <p>Nombre de usuario:</p>
     <p><input type = "text"    name = "nombre" /></p>
     <p><input type = "submit" value = "submit" /></p>
  </form>
</body>
```

Enrutamiento con peticiones HTTP POST

```
from flask import redirect, url_for,request
```

- **app.py**

```
@app.route('/login’, methods = ['POST', 'GET'])
def login():
   if request.method == 'POST':
      usuario = request.form['nombre']
      return redirect(url_for('dashboard',nombre = usuario))
   else:
      usuario = request.args.get('nombre')
      return render_template('login.html’
```

```
@app.route('/dashboard/<nombre>')
def dashboard(nombre):
   return 'Bienvenido %s' % nombre
```

Construyendo una aplicación de gestión de recetas

Vamos a construir una plataforma para compartir recetas
y la API es la interfaz que exponemos al público.

1. Definir las funciones que vamos a proporcionar y las URLs
   - ID, Nombre, Descripción
2. Construir una API que liste todas las recetas almacenadas en el sistema
   - todas las recetas: http://localhost:5000/recetas
   - la receta con ID=20: http://localhost:5000/recetas/20

Construyendo una aplicación de gestión de recetas

1. Importa los paquetes que necesitamos para este servicio web

```
from flask import Flask, jsonify, request
from http import HTTPStatus
```

2. Crea una instancia de la clase Flask

```
app = Flask(__name__)
```

3. Definir una lista de recetas que contenga un diccionario con dos recetas de ejemplo

```
recetas = [ { 'id': 1,
              'nombre': 'ensalada rusa’,
              'descripcion': 'encantadora receta de…’ }, …]
```

Construyendo una aplicación de gestión de recetas

4. Utiliza el decorador de ruta para indicar a Flask

que la ruta `/recetas` se dirigirá a la función obtener_recetas,

y el argumento methods = ['GET’]

```
@app.route('/recetas/', methods=['GET’])
def obtener_recetas():
```

5. Utiliza la función jsonify para convertir y devolver la lista de recetas a formato JSON

```
return jsonify({'datos': recetas})
```

Construyendo una aplicación de gestión de recetas

6. Recuperar una receta específica,

   utiliza la ruta `/recetas/<int:receta_id>`

   para activar la función get_receta(receta_id).

```
@app.route('/recetas/<int:receta_id>', methods=['GET'])
```

7. Dentro de la función: Si encontramos la receta la devolvemos caso contrario enviamos un mensaje de receta no encontrada

```
for receta in recetas:
        if receta['id'] == receta_id:
            return jsonify(receta)
    return jsonify({'message': 'receta no encontrada'})
```

Construyendo una aplicación de gestión de recetas

8. Crear en la función crear_receta, utiliza la ruta /recetas para la función create_receta y el argumento "methods = [POST]" para especificar que el decorador de la ruta responderá a peticiones POST

```
@app.route('/recetas', methods=['POST’])
```

9. Utiliza el método request.get_json para obtener el nombre y la descripción de la solicitud POST del cliente.

Estos dos valores, junto con un id autoincrementado que generamos,

se almacenarán en la receta (objeto diccionario) y luego se añadirán a nuestra lista de recetas.

Construyendo una aplicación de gestión de recetas

```
def crear_receta():
    data = request.get_json()
    nombre = data.get('nombre')
    descripcion = data.get('descripcion')
    receta = {
        'id': len(recetas) + 1,
        'nombre': nombre,
        'descripcion': descripcion
    }
    recetas.append(receta)
```

Construyendo una aplicación de gestión de recetas

10. Devuelve la receta que se acaba de crear en formato JSON, junto con un estado HTTP 201 (CREADO).

```
        return jsonify(receta), HTTPStatus.CREATED
```

11. Actualización de las recetas, utilice la misma línea de código del iterador next

```
@app.route('/recetas/<int:receta_id>', methods=['PUT'])
def actualizar_receta(receta_id):
    receta = next((receta for receta in recetas if receta['id'] == receta_id), None)
```

Construyendo una aplicación de gestión de recetas

12. Si no podemos encontrar la receta devolveremos un mensaje de receta no encontrada en formato JSON, junto con un estado HTTP NOT_FOUND

```
if not receta:
        return jsonify({'message': 'receta no encontrada'}), HTTPStatus.NOT_FOUND
```

13. Si encontramos la receta, entonces ejecutamos la función recipe.update, y ponemos el nuevo nombre y descripción que obtenemos de la petición del cliente:

```
data = request.get_json()
    receta.update(
        {
            'nombre': data.get('nombre'),
            'descripcion': data.get('descripcion')
        }
    )
```

Construyendo una aplicación de gestión de recetas

14. Convertir la receta actualizada a formato JSON utilizando la función jsonify y la devolvemos junto con un estado HTTP 200 (OK) por defecto.

```
return jsonify(receta)
```

15. El código que sirve para arrancar el servidor Flask:

```
if __name__ == '__main__':
    app.run()
```

16. Ejecutar para iniciar la aplicación.

El servidor Flask se iniciará y nuestra aplicación estará lista para ser probada

Postman

- Herramienta útil para las pruebas de la API.
- Tiene una interfaz gráfica de usuario amigable
- Se puede enviar peticiones HTTP con diferentes métodos(GET, POST, PUT y DELETE) y podemos comprobar la respuesta del servidor.
- Permite guardar nuestros casos de prueba y agruparlos en diferentes colecciones.

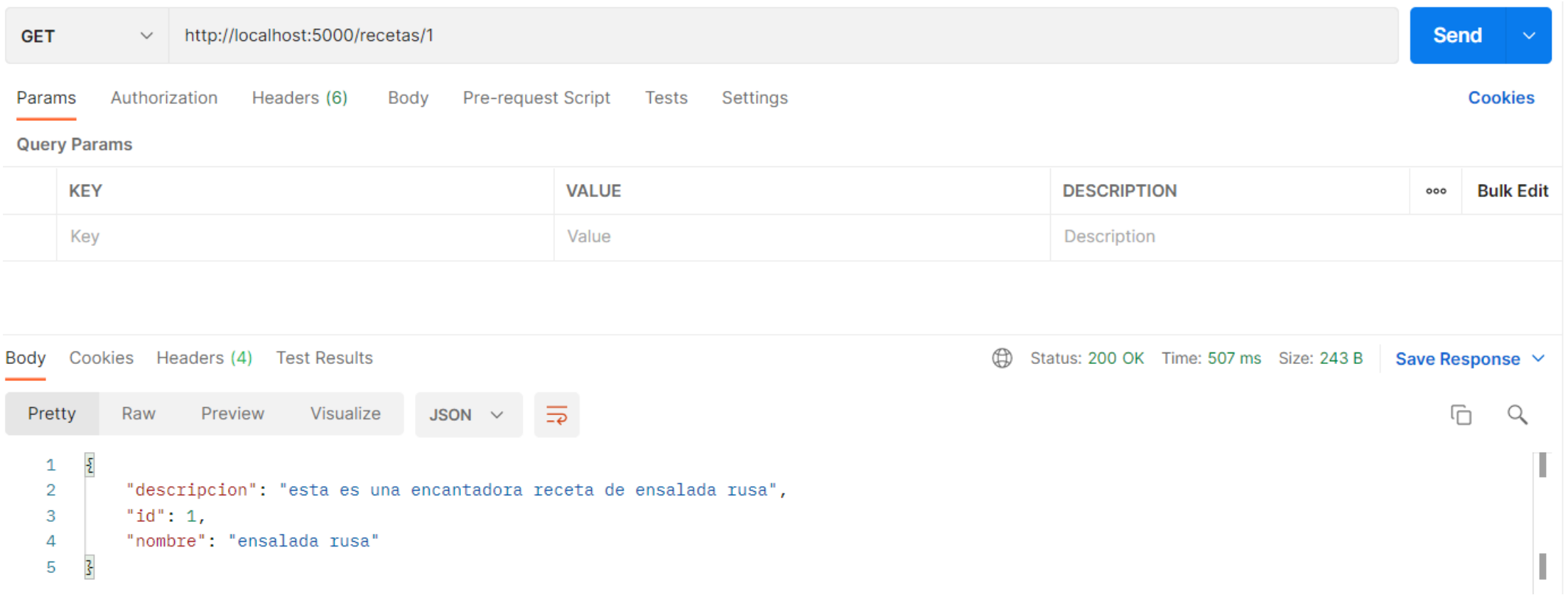
Listar todas las recetas

Body ▾ 200 OK 522 ms 360 B Save Response

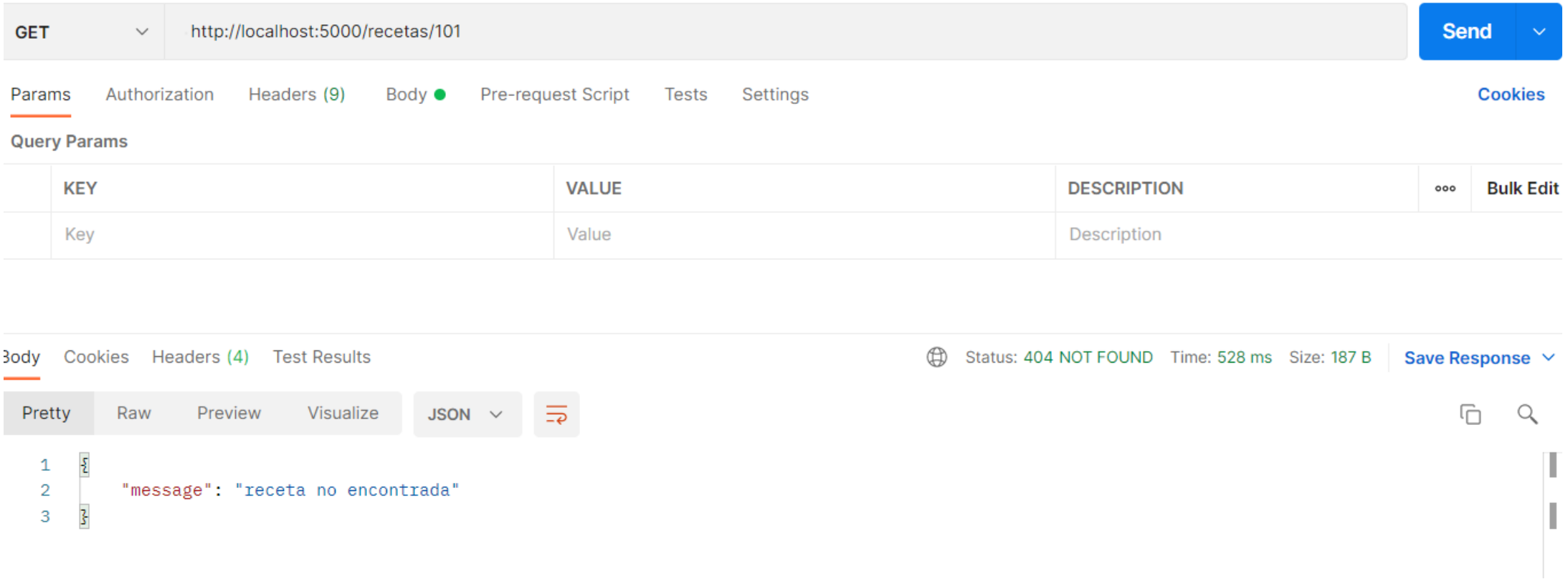
Pretty Raw Preview Visualize JSON ▾


```
{
    "data": [
        {
            "descripcion": "esta es una encantadora receta de ensalada rusa",
            "id": 1,
            "nombre": "ensalada rusa"
        },
        {
            "descripcion": "esta es una encantadora receta de arroz con tomate",
            "id": 2,
            "nombre": "rissoto con pasta"
        }
    ]
}
```

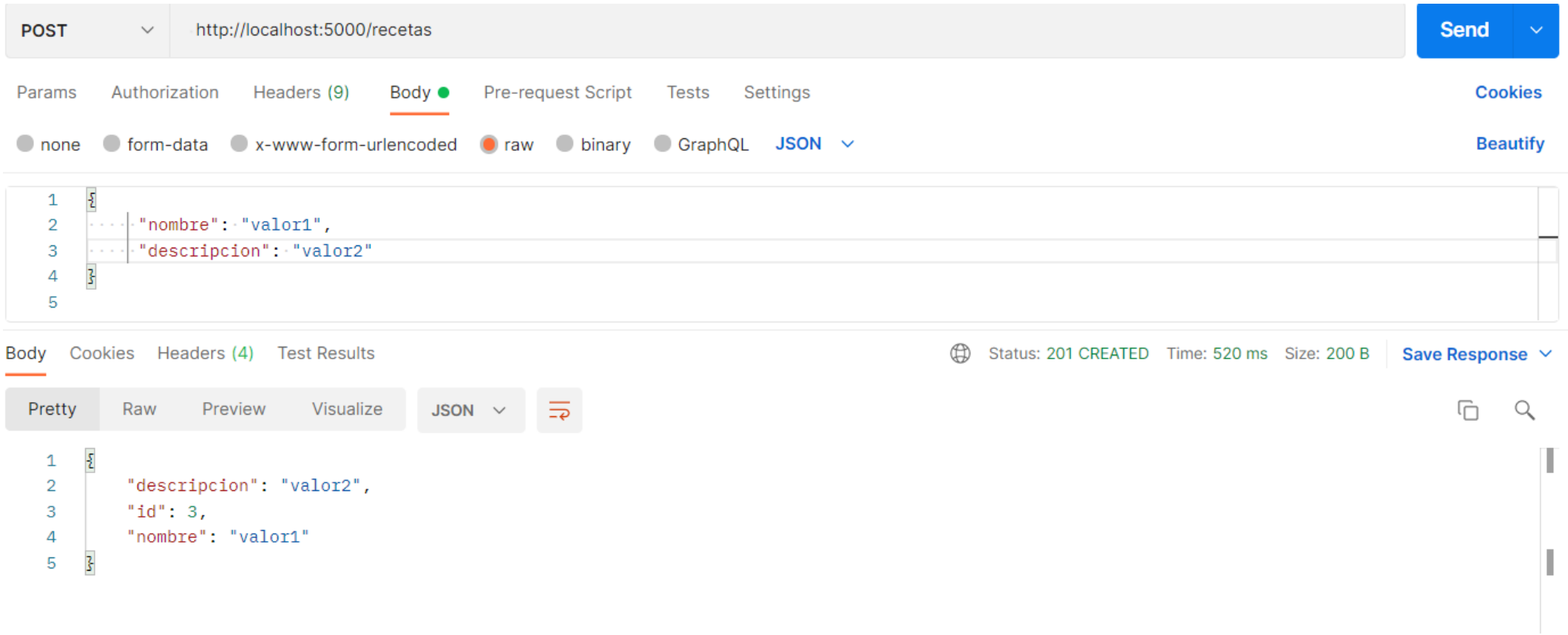
Listar una receta en especifico



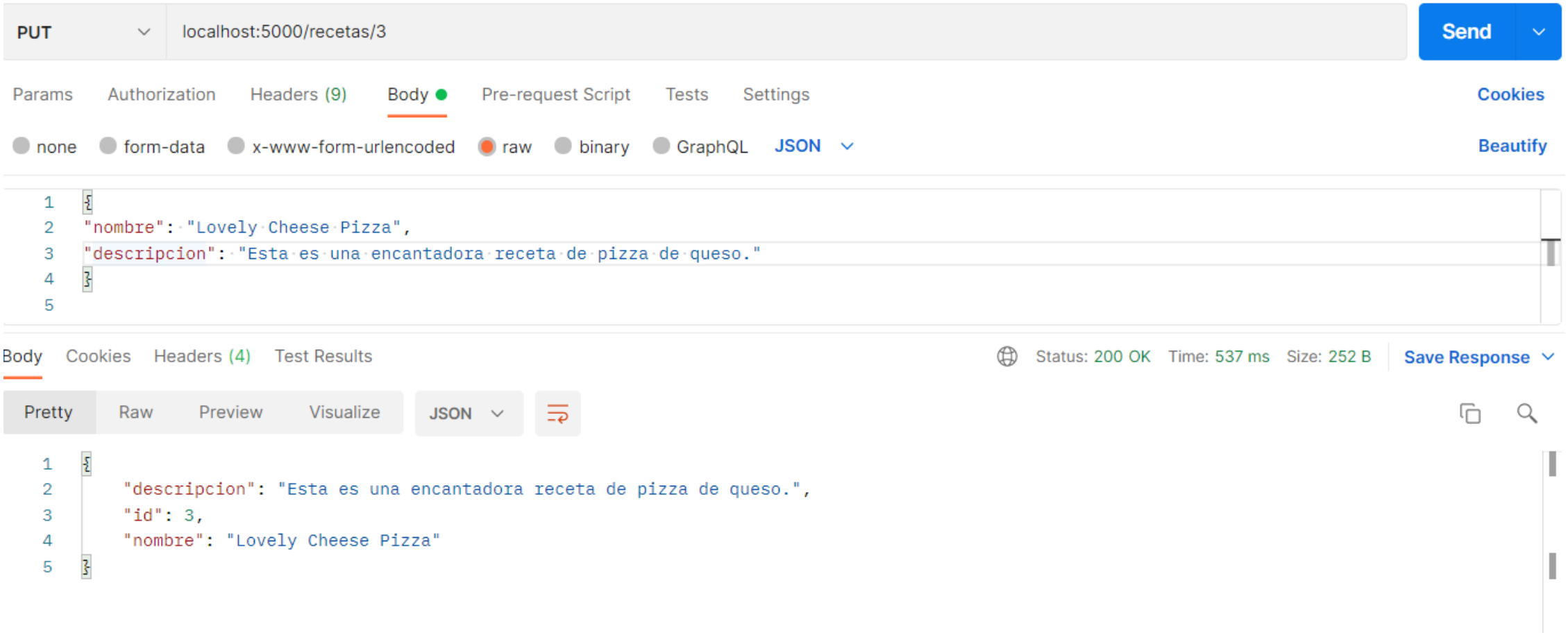
404 - Receta no encontrada



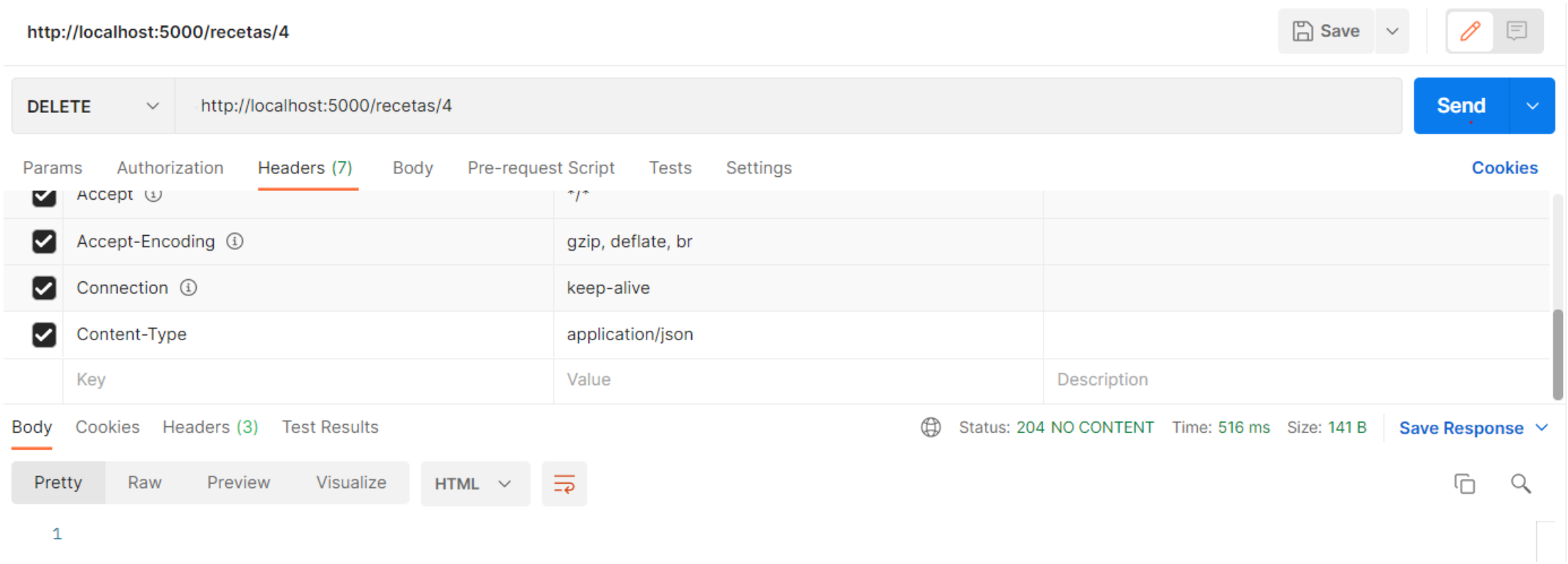
Insertar una nueva receta



Actualización de una receta



Eliminar una receta



Resumen

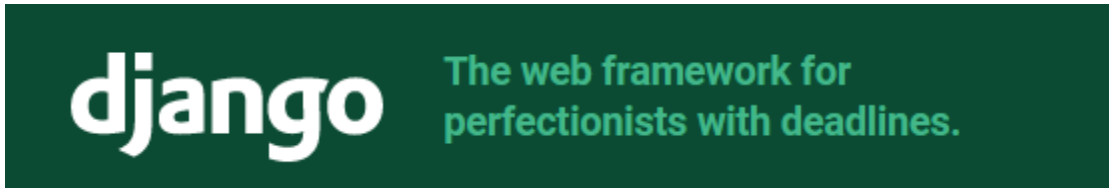
En este capítulo:

- Hemos repasado los conceptos relevantes al desarrollo web con Python como los códigos de estado HTTP, JSON, el enrutamiento, etc
- Hemos construido una API RESTful básica usando Flask.
- Hicimos operaciones CRUD (Create, Read, Update, Delete) con varios ejemplos
- Con una aplicación de recetas hemos comprendido los conceptos y fundamentos de las APIs.
- Terminamos probando los servicios web que hemos construido.

djangoproject.com

El framework web Django

- Es ampliamente utilizado y cada día más gente lo esta usando.
- Django v1.0 en **2007**
  - su longevidad ha demostrado que es fiable y consistente
- Django sigue en desarrollo activo
  - con correcciones de errores y parches de seguridad (mensuales)
- Es un framework “con las pilas incluidas"
  - no hay que buscar e instalar otras librerías para poner en marcha una aplicación
  - Django incorpora soporte para
    - consulta de la base de datos
    - mapeo de URLs
    - renderizado de plantillas



- Django facilita la creación de aplicaciones web de forma más rápida y con menos código

- Utilizado por

Pantalla de bienvenida por defecto de Django

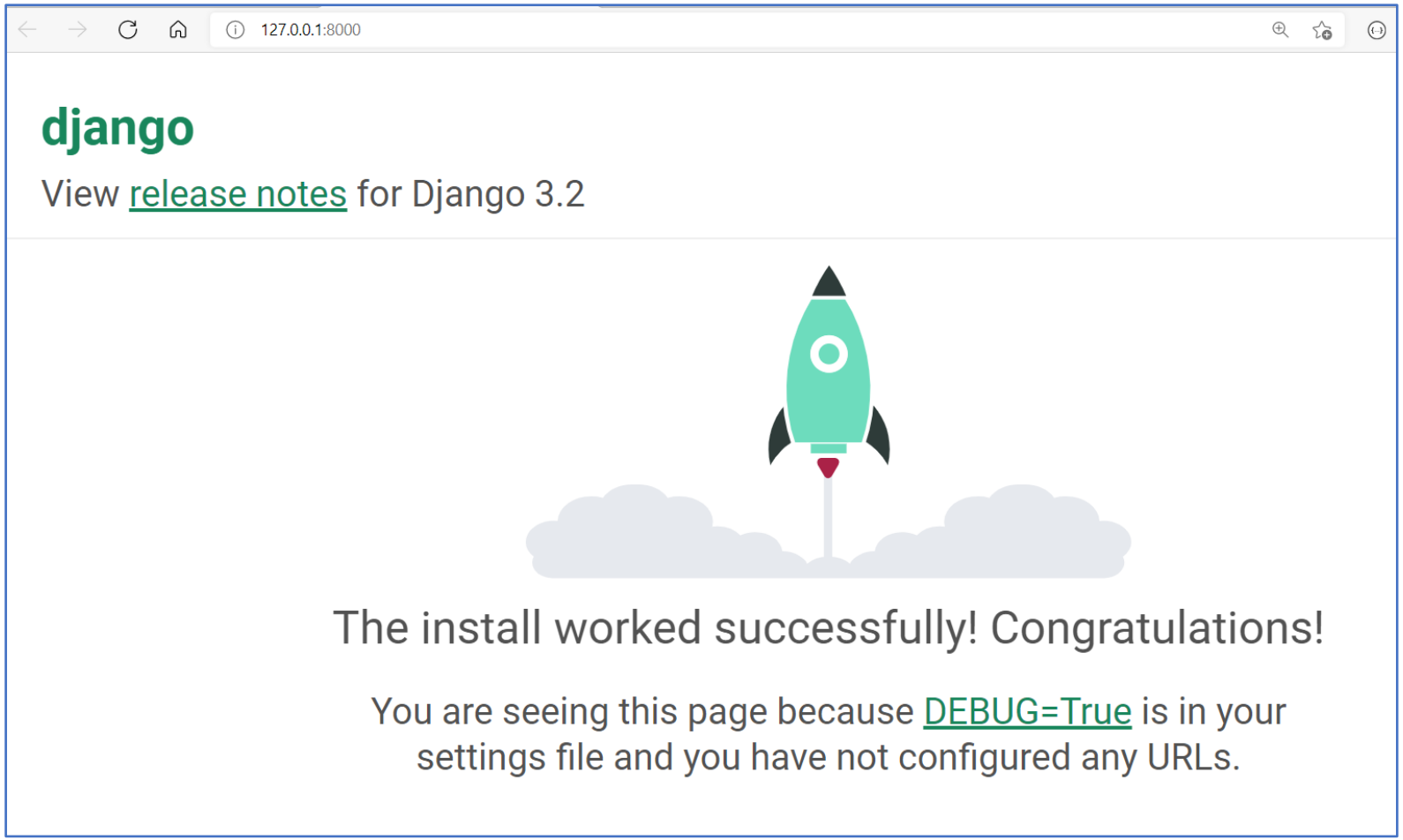
Dentro del espacio virtual ejecutar:

```
1. pip install django              #Django-3.2.9
2. django-admin startproject hola_mundo_django
3. cd hola_mundo_django
4. python manage.py migrate
5. python manage.py runserver
```

6. Ir a http://127.0.0.1:8000/

Pantalla de bienvenida por defecto de Django

http://127.0.0.1:8000/



Construcción de un proyecto Django

- Un proyecto Django es un directorio que contiene todos los datos de tu proyecto:
  - código
  - configuraciones
  - plantillas
  - activos
- El proyecto se crea y se organiza ejecutando `django-admin.py` con el argumento `startproject y` el nombre del mismo
  - `django-admin.py startproject miproyecto`
- Para iniciar el servidor Django dev
  - `python3 manage.py runserver`

Crear un proyecto y una aplicación, e iniciar el servidor de desarrollo

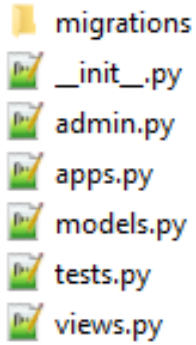
- Construiremos un sitio web de reseñas de libros llamado Criticalibros
- Permitirá añadir campos para editores, colaboradores, libros y criticas.
- Un editor publicará uno o más libros,
- Cada libro tendrá uno o más colaboradores (autor, editor, coautor, etc.).
- Los usuarios pueden añadir criticas de libros

1. Ejecute

```
django-admin startproject criticalibros
cd criticalibros
```

2. Crear la app de criticas para el proyecto

```
python manage.py startapp criticas
```



Aplicaciones Django

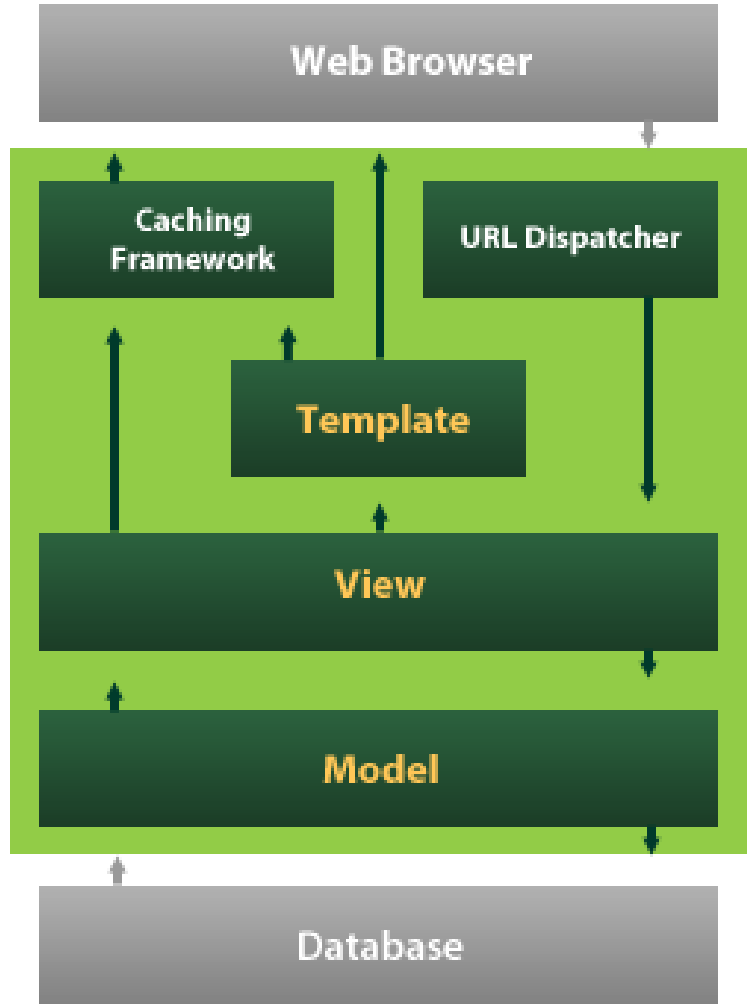
- El comando **startapp** crea
  - un directorio dentro del proyecto con el nombre de la app
  - varios archivos para la aplicación
- **__init.py__:** archivo vacío que indica que este directorio es un módulo de Python
- **admin.py**: Define cómo se exponen los modelos de la aplicación en el sitio de administración de Django
- **apps.py:** Contiene configuraciones para los metadatos de la app

Aplicaciones Django *(cont.)*

- **models.py**: Donde se define los modelos para la aplicación.
- **migraciones:** En este directorio se almacenan los archivos generados por Django cuando se ejecuta el comando *manage.py makemigrations* y no se aplican a la BBDD hasta que se ejecuta *manage.py migrate*.
- **tests.py:** Escribir tests para comprobar que tu código se comporta correctamente
- **views.py:** El código que responde a las peticiones HTTP ira aquí.

Modelo Vista Plantilla (MVT)

- Django sigue el paradigma donde una vista consultará un modelo y luego lo renderizará con una plantilla
- Con MVT, la plantilla puede estar en un lenguaje diferente los modelos y las vistas están escritos en Python y la plantilla en HTML



Modelos

- Definen los datos de la aplicación y proporcionan una capa de abstracción para el acceso a la BBDD SQL a través de un Mapeador Relacional de Objetos (ORM)
  - Definir tu capa de base de datos en Python y Django se encargará de generar las consultas SQL
  - Normalmente, cuando se consulta una BBDD los resultados se devuelven como objetos primitivos (listas de cadenas, enteros, flotantes o bytes)
  - Cuando se utiliza el ORM los resultados se convierten automáticamente en instancias de las clases modelo definidinas
  - Automáticamente protegido de inyección SQL

Vistas

- Define la mayor parte de la lógica de la aplicación
- Es una función que recibirá la petición en forma de objeto Python (HttpRequest) y debe decidir cómo responder a la petición y qué debe devolver.
- Debe devolver un objeto HttpResponse que encapsula toda la información que se proporciona al cliente: contenido, estado HTTP y otras cabeceras
- Un patrón de diseño común
  - Consultar una base de datos a través del ORM de Django utilizando un ID que se pasa a la vista y la plantilla renderiza los datos del modelo (base de datos) de HttpResponse.

Plantillas

- Son archivos HTML que contienen marcadores de posición especiales que son reemplazados por variables que la aplicación proporciona.
- Ejemplo:
  - La aplicación que representa una lista de elementos en un diseño de galería o en un diseño de tabla.
  - La vista obtiene los mismos modelos para cualquiera de los dos diseños pero es capaz de renderizar un archivo HTML para presentar los datos de manera diferente.

MVT en la práctica

- Tenemos:
- un modelo de Libro que almacena información sobre diferentes libros, y
- un modelo de Critica que almacena información sobre diferentes criticas de los libros.

En el primer ejemplo,

- queremos ser capaces de editar la información sobre un libro o una critica.
- Tomemos el primer escenario,
- editar los detalles de un libro.
- Tendríamos una vista para obtener los datos del libro de la base de datos
- y proporcionar el modelo del libro.

Escribir una vista y asignarle una URL

1. En criticalibros/criticas/views.py

Elimina el texto y en su lugar inserta:

```
from django.http import HttpResponse
def index(request):
    return HttpResponse("Hola, mundo")
```

Escribir una vista y asignarle una URL

2. En criticaslibros/criticaslibros/urls.py.

- Añada una URL para reemplazar el índice por defecto que proporciona Django

```
import criticas.views
```

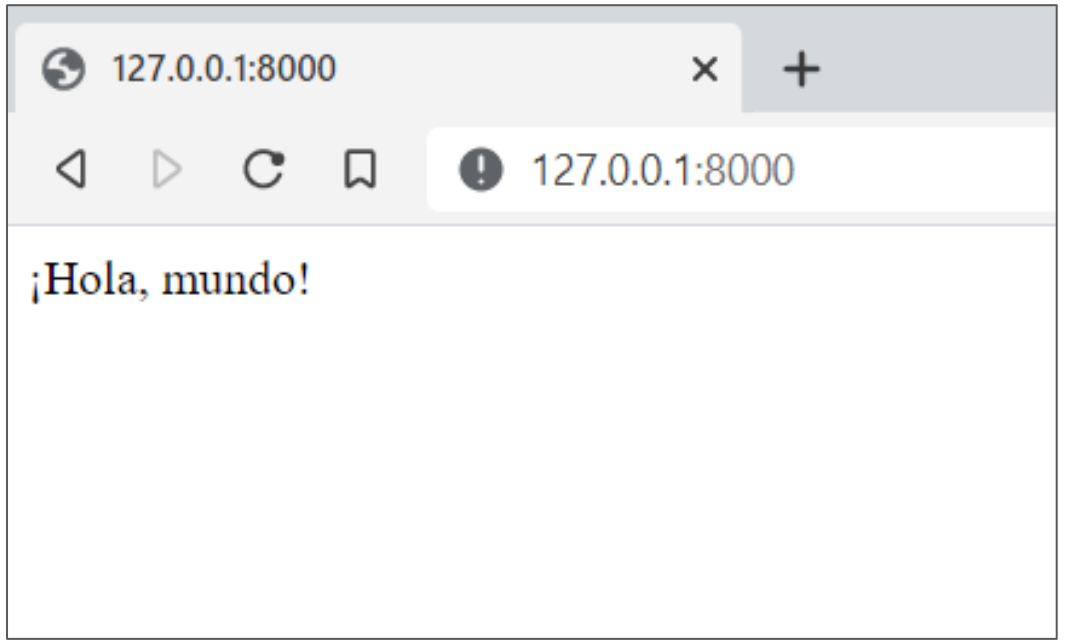
- Añada una llamada a la función path a la lista de urlpatterns

añadiendo una cadena vacía y una referencia a la función index

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', criticas.views.index)
]
```

Escribir una vista y asignarle una URL

3. Vuelva a actualizar el navegador web



Leer valores de la URL en el atributo GET

1. Abre criticalibros/criticas/views.py

- Añade después de la definición de la función index una variable llamada nombre que lea un nombre desde los parámetros GET

```
nombre = request.GET.get(“nombre") or "mundo"
```

- Cambia el valor de retorno para que se utilice el nombre como parte del contenido que se devuelve:

```
return HttpResponse(f"Hola, {nombre}!")
```

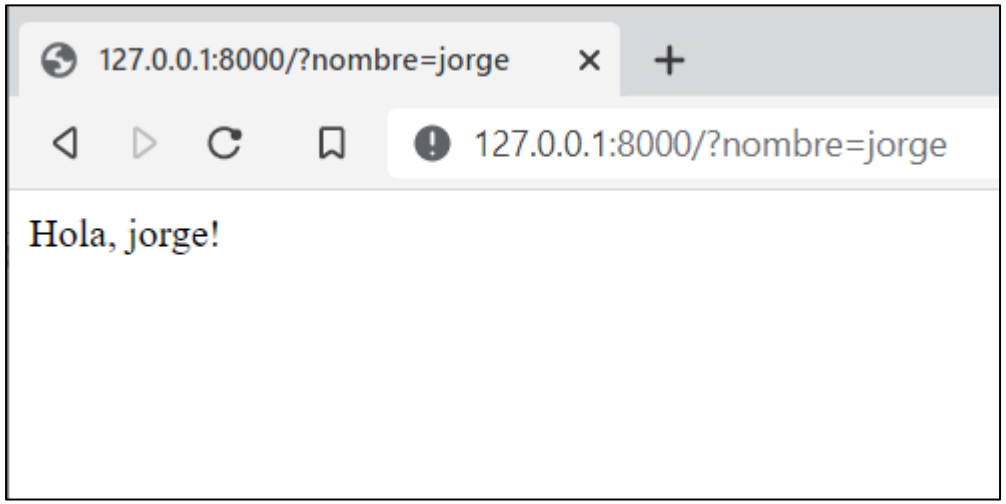
Leer valores de la URL en el atributo GET

- Vuelva a actualizar el navegador web
- Deberías notar que la página sigue diciendo “Hola, mundo”

  Se debe a que no hemos proporcionado un parámetro de nombre
- Añadir tu nombre en la URL

por ejemplo:

http://127.0.0.1:8000?name=jorge



Explorando la configuración de Django

- Este archivo contiene muchas configuraciones que pueden ser utilizadas para personalizar Django.
- Cada ajuste en este archivo es sólo una variable global del archivo.
- **SECRET_KEY = '...'** valor generado automáticamente que no debe ser compartido con nadie.
- **DEBUG = True** Para mostrar automáticamente excepciones al navegador para permitir depurar cualquier problema

Explorando la configuración de Django *(cont.)*

- **INSTALLED_APPS = [...]** Deben estar en esta lista las apps para que Django pueda encontrar automáticamente las plantillas, los archivos estáticos, las migraciones y otras configuraciones.
- **ROOT_URLCONF = 'criticalibros.urls'** Indica el módulo de Python que Django cargará primero para encontrar las URLs.
- **TEMPLATES = [...]**
- **'APP_DIRS': True** Para que busque en un directorio de plantillas dentro de cada INSTALLED_APP cuando cargue una plantilla para renderizar

Crear un directorio de plantillas y una plantilla base

Necesitamos añadir la aplicación “criticas” a INSTALLED_APPS

para que Django pueda encontrar las plantillas.

- Abre criticalibros/criticalibros/settings.py

  añade “criticas” al final de INSTALLED_APPS

```
INSTALLED_APPS = ['django.contrib.admin’,\
                  …
                  ‘criticas']
```

Renderizando una plantilla con render()

- Render es una función de acceso directo que devuelve una instancia de HttpResponse
- render toma al menos dos argumentos:
  1. es siempre la solicitud que se pasó a la vista,
  2. es el nombre/ruta relativa de la plantilla que se está renderizando.

- Necesitamos actualizar la vista índice para que renderice la plantilla en lugar de devolver el texto ¡Hola (nombre)!

Renderizando una plantilla con render()

1. Crear un nuevo directorio llamado “plantillas” dentro de “criticas”
2. Crear un archivo HTML nombre base.html en el directorio “plantillas” con el siguiente contenido:

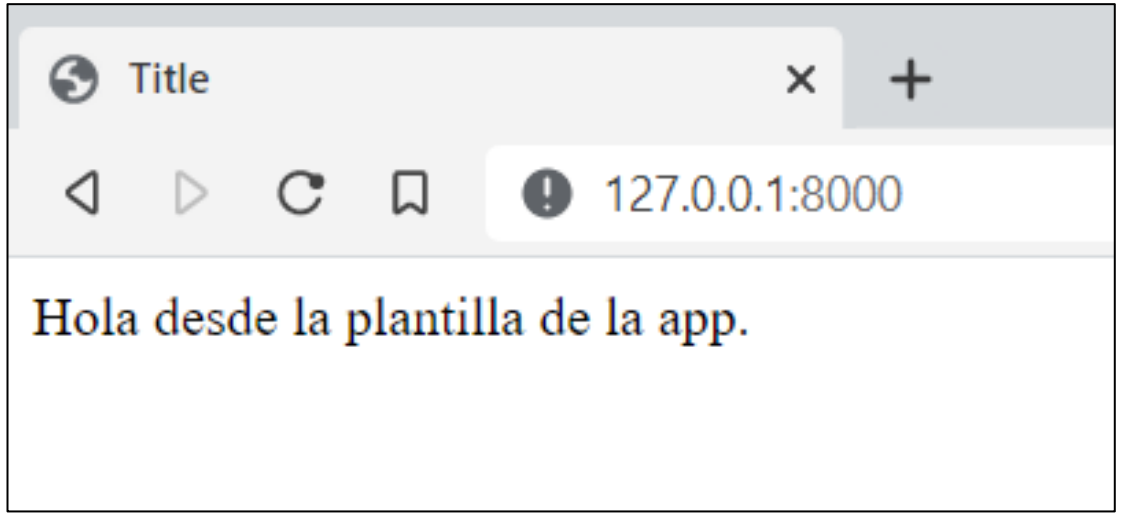
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title> Titulo </title>
</head>
<body>
    Hola desde la plantilla de la app.
</body>
</html>
```

Renderizando una plantilla con render()

- Abra “views.py” en el directorio criticas y actualize con este contenido

```
from django.shortcuts import render
def index(request):
    return render(request, "base.html")
```

- Luego, abre tu navegador web
- y refresca http://127.0.0.1:8000.



Renderización de variables en las plantillas

- La mayoría de las veces como parte del proceso de renderizado de plantillas estas contienen variables que se interpolan
- Las variables que una plantilla puede utilizar se pasan desde la vista
  - utilizando un contexto
  - un diccionario (o un objeto similar)
- Dentro de una plantilla las variables se denotan con llaves dobles {{ }}
- Dentro de la plantilla utilizamos una variable {{ nombre_del_libro }}

  y la vista proporciona a la plantilla una variable nombre_libro

  con el título del modelo de libro que ha cargado.

Renderización de variables en las plantillas

1. Abre base.html y actualiza el elemento <body> para renderizar la variable nombre

```
<body>
    Hola, {{ nombre }}
</body>
```

2. Abra views.py y añada una variable llamada nombre con el valor “marte“ dentro de la función index:

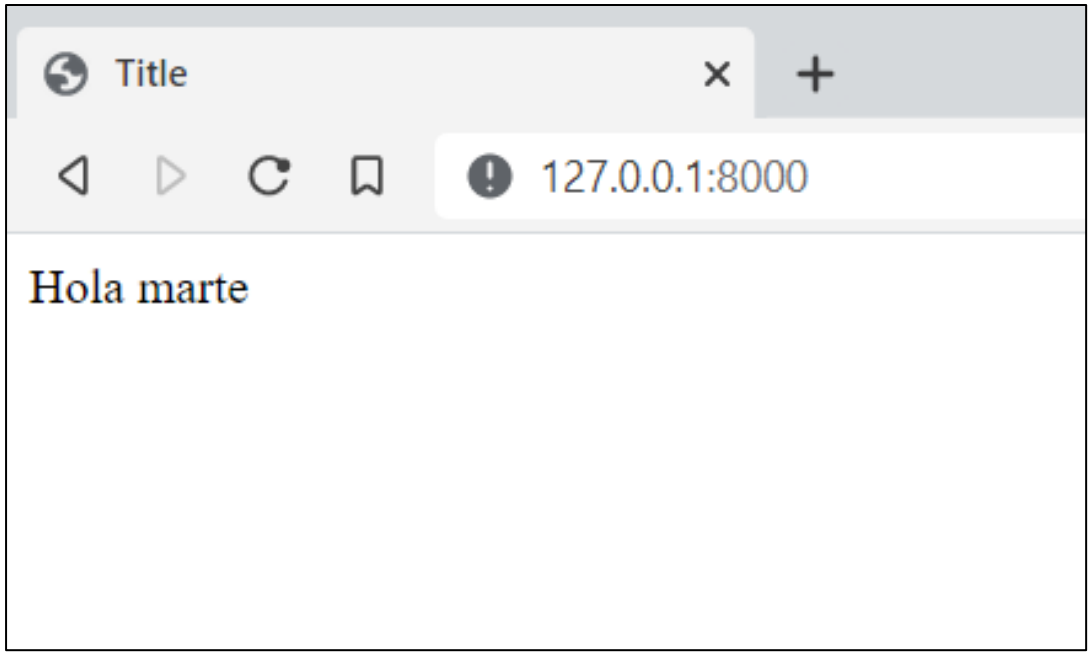
```
def index(request):
    nombre = “marte"
    return render(request, "base.html")
```

Renderización de variables en las plantillas

3. Añade el diccionario de contexto como tercer argumento a la función de renderizado.

```
return render(request, "base.html", {"nombre":nombre})
```

4. Refresca tu navegador de nuevo



Bases de datos

- Una base de datos es una colección estructurada de datos que ayuda a gestionar la información fácilmente.

- Una capa de software llamada Sistema de Gestión de Bases de Datos (SGBD) se utiliza para almacenar, mantener y realizar operaciones con los datos.

- Las bases de datos son de dos tipos,
  - relacionales
  - no relacionales.

Bases de datos (BBDD) *(cont.)*

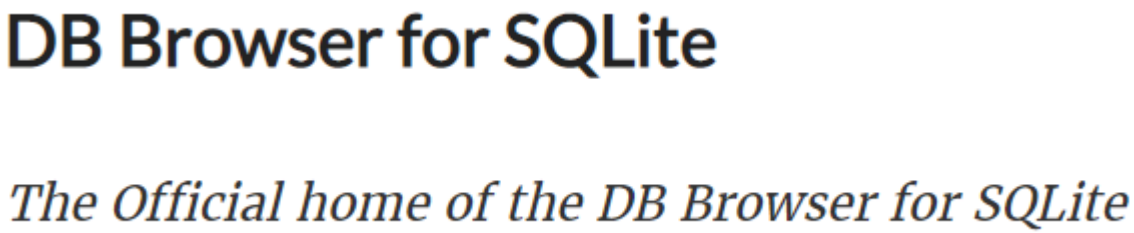
- Django soporta bases de datos relacionales como
  - SQLite, PostgreSQL, Oracle Database y MySQL.
- La capa de abstracción de la bbdd de Django
  - asegura que el mismo código fuente de Python/Django puede ser utilizado a través de cualquiera de las bases de datos relacionales anteriores con muy poca modificación de la configuración del proyecto.
- La configuración por defecto de la base de datos es de SQLite3.
- La configuración de la bbdd esta presente en el directorio del proyecto, en el archivo settings.py.

Operaciones con bases de datos usando SQL

- SQL utiliza un conjunto de comandos para realizar una variedad de operaciones en la base de datos, como
  - creación de una entrada,
  - lectura de valores,
  - actualización de una entrada y
  - eliminación de una entrada.
- Estas operaciones Crear, Leer, Actualizar y Eliminar se denominan colectivamente operaciones CRUD
- Utilizaremos SQLite como base de datos

SQLite

- Es una base de datos relacional ligera
- Forma parte de las bibliotecas estándar de Python
- Django utiliza SQLite como su configuración de base de datos por defecto
  - para utilizar otras bases de datos se puede realizar cambios de configuración
- Visor: sqlitebrowser.org



Mapeo relacional de objetos en Django(ORM)

- Las aplicaciones web interactúan constantemente con las bases de datos (utilizando SQL)
- Django proporcionan un nivel de abstracción con el que podemos trabajar con las bases de datos.
- La parte de Django que nos ayuda a hacer esto se llama ORM
- El ORM de Django convierte el código Python orientado a objetos
- en construcciones reales de bases de datos,
- como tablas de bases de datos con definiciones de tipos de datos,
- y facilita todas las operaciones de bases de datos
- a través de código Python simple.

- No tenemos que lidiar con comandos SQL
- Ayuda a un desarrollo más rápido de la aplicación
- Facilita el mantenimiento del código fuente de la aplicación

Configuración de la base de datos

- En la configuración de la base de datos

Como estamos usando SQLite usaremos la configuración de la base de datos que ya existe, no hay necesidad de hacer ninguna modificación.

Creación de modelos y migraciones Django

En criticas/models.py vamos a añadir los modelos Libro y Colaborador

```
class Libro(models.Model):
    titulo = models.CharField\
            (max_length=70, \
             help_text="El titulo del libro.")
    fecha_publicacion = models.DateField\
        (verbose_name="La fecha en que el libro fue publicado.")
    isbn = models.CharField\
           (max_length=20, \
            verbose_name="El numero ISBN del libro.")
    def __str__(self):
        return self.titulo
```

Creación de modelos y migraciones Django

Ejecutar el comando para crear los scripts de migración

```
python manage.py makemigrations criticas
Migrations for 'criticas’:
  criticas\migrations\0001_initial.py
    - Create model Colaborador
    - Create model Editor
    - Create model Libro
```

Los scripts de migración se crearán en una carpeta llamada migraciones en la carpeta de la aplicación

Creación de modelos y migraciones Django

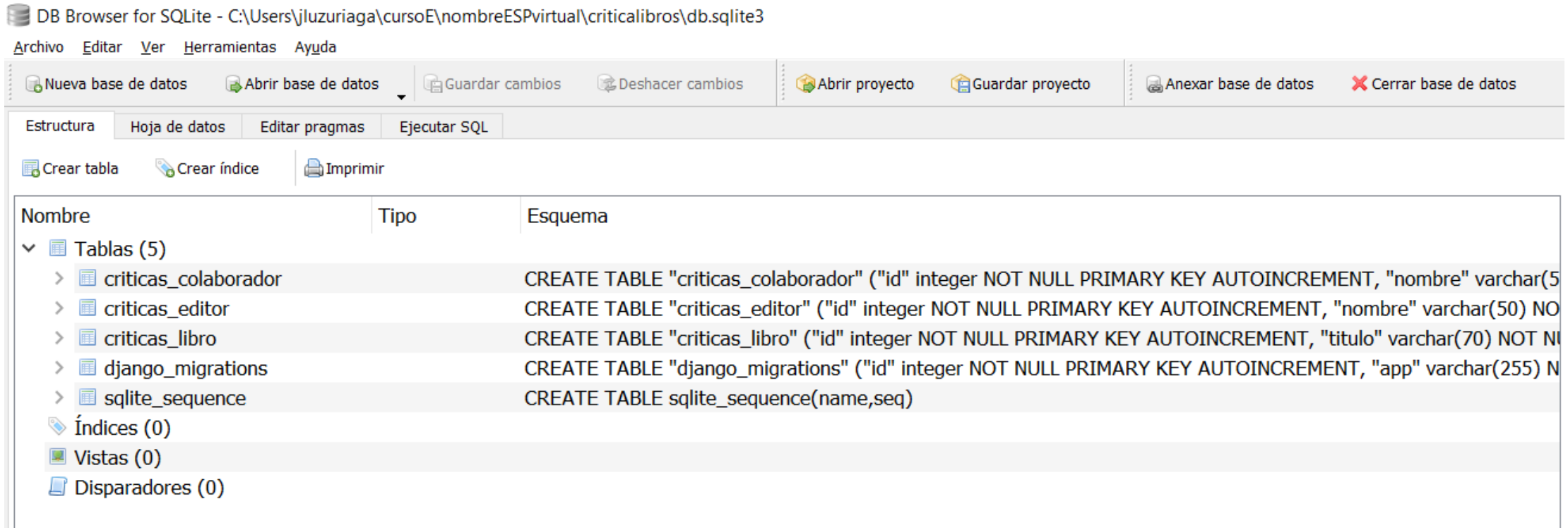
Migra todos los modelos a la base de datos

```
python manage.py migrate criticas
Operations to perform:
  Apply all migrations: criticas
Running migrations:
  Applying criticas.0001_initial... OK
```

Hemos creado con éxito las tablas de la BBDD definidas en la aplicación de “criticas”

Creación de modelos y migraciones Django

- Para explorar las tablas que se acaban de crear después de la migración se puede utilizar "*DB Browser for SQLite*"



Operaciones CRUD de la BBDD de Django

- Entramos en el shell de línea de comandos de Django:

```
python manage.py shell
```

- Operaciones de escritura
  - create()
  - set()
- Operaciones de lectura
  - get()
  - all()

Tabla Editor

```
>>> from criticas.models import Editor
>>> editorial = Editor(nombre='Planeta', website='https://www.planetadelibros.com', email='info@planetadelibros.com')
>>> editorial.save
<bound method Model.save of <Editor: Planeta>>
>>> editorial.email
'info@planetadelibros.com'
>>> editorial.email = 'customersupport@planetadelibros.com'
>>> editorial.save
<bound method Model.save of <Editor: Planeta>>
>>> editorial.email
'customersupport@planetadelibros.com'
```

Tabla Colaborador

```
>>> from criticas.models import Colaborador
>>> autor = Colaborador.objects.create(nombre="Jorge", apellido="Ariaga", email="jorge@arriaga.com")
>>> autor
<Colaborador: Jorge>
>>> autor.nombre
'Jorge'
>>> autor.apellido
'Ariaga'
>>> autor.email
'jorge@arriaga.com'
```

Tabla Libro

```
>>> from criticas.models import Libro
>>> libro1 = Libro.objects.create(titulo="La Bestia",fecha_publicacion="2021-08-15", isbn="12345679")
>>> libro1
<Libro: La Bestia>
>>> libro1.titulo
'La Bestia'
>>> libro1.isbn
'12345679'
>>> libro1.fecha_publicacion
'2021-08-15'
```

Recuperar información

```
>>> libro_r = Libro.objects.get(titulo="El Bola")
>>> libro_r
<Libro: El Bola>
>>> libro_r.isbn
'12345679'
>>> Libro.objects.all()
<QuerySet [<Libro: La Bestia>, <Libro: El Quijote>, <Libro: El Bola>, <Libro: El pais>, <Libro: El cosmos>]>
>>> libros=Libro.objects.all()
>>> libros[1]
<Libro: El Quijote>
>>> Libro.objects.filter(isbn='12345679')
<QuerySet [<Libro: La Bestia>, <Libro: El Quijote>, <Libro: El Bola>, <Libro: El pais>, <Libro: El cosmos>]>
```

Resumen 2da Parte

La 2da parte de esta unidad fue una introducción rápida a Django.

- Vimos cómo Django utiliza el paradigma MVT, cómo analiza una URL, genera una solicitud HTTP, y la envía a una vista para obtener una respuesta HTTP.
- Construimos vistas de ejemplo para ilustrar cómo obtener datos de una solicitud y utilizarlas al renderizar plantillas.

Resumen 2da Parte

Además hemos aprendido

- Conceptos básicos de bases de datos y su importancia en el desarrollo de aplicaciones.
- Django proporciona una capa de abstracción llamada ORM para interactuar sin problemas con las bases de datos relacionales utilizando código de Python
- Crear modelos para la aplicación desarrollada y adquirimos todas las habilidades que necesitamos para interactuar con los datos almacenados dentro de la base de datos de la aplicación.