



FastAPI



A decorative pattern of hexagons in various shades of blue and cyan. Some hexagons contain icons: a lightbulb, a thumbs up, a network of nodes, a smartphone, a magnifying glass, a gear, and a speech bubble. The number '1' is centered in a large cyan hexagon.

1

¿Que es una Rest API?

¿Qué entendemos por API en software?



REST

(Representational state transfer) es un estilo arquitectónico de software que define un conjunto de restricciones para crear servicios web





Restricciones

- ◆ **Arquitectura Cliente-servidor:** la lógica de interfaces y la lógica de la aplicación debería estar separada
- ◆ **Sin estado:** el servidor no debería almacenar datos de los pedidos del cliente es decir que cada pedido debe contener toda la información necesaria
- ◆ **Sistema construido por capas:** el cliente no debería conocer si está interactuando con el servidor directamente o con algún intermediario.





Componentes


- ◆ **Endpoint:** la url en la que está escuchando pedidos el servidor
- ◆ **Métodos:** GET, POST, PATCH/PUT, DELETE
- ◆ **Cabecera (Headers):** cabecera del mensaje
- ◆ **Cuerpo (Body):** contiene la información que queremos enviarle al server.





Métodos REST

Método	Descripción
GET	Obtener un recurso del servidor
POST	Crear un recurso nuevo en el servidor
PATCH/PUT	Modificar o editar un recurso existente
DELETE	Borrar un recurso del servidor





Un ejemplo:
contador de
chistes



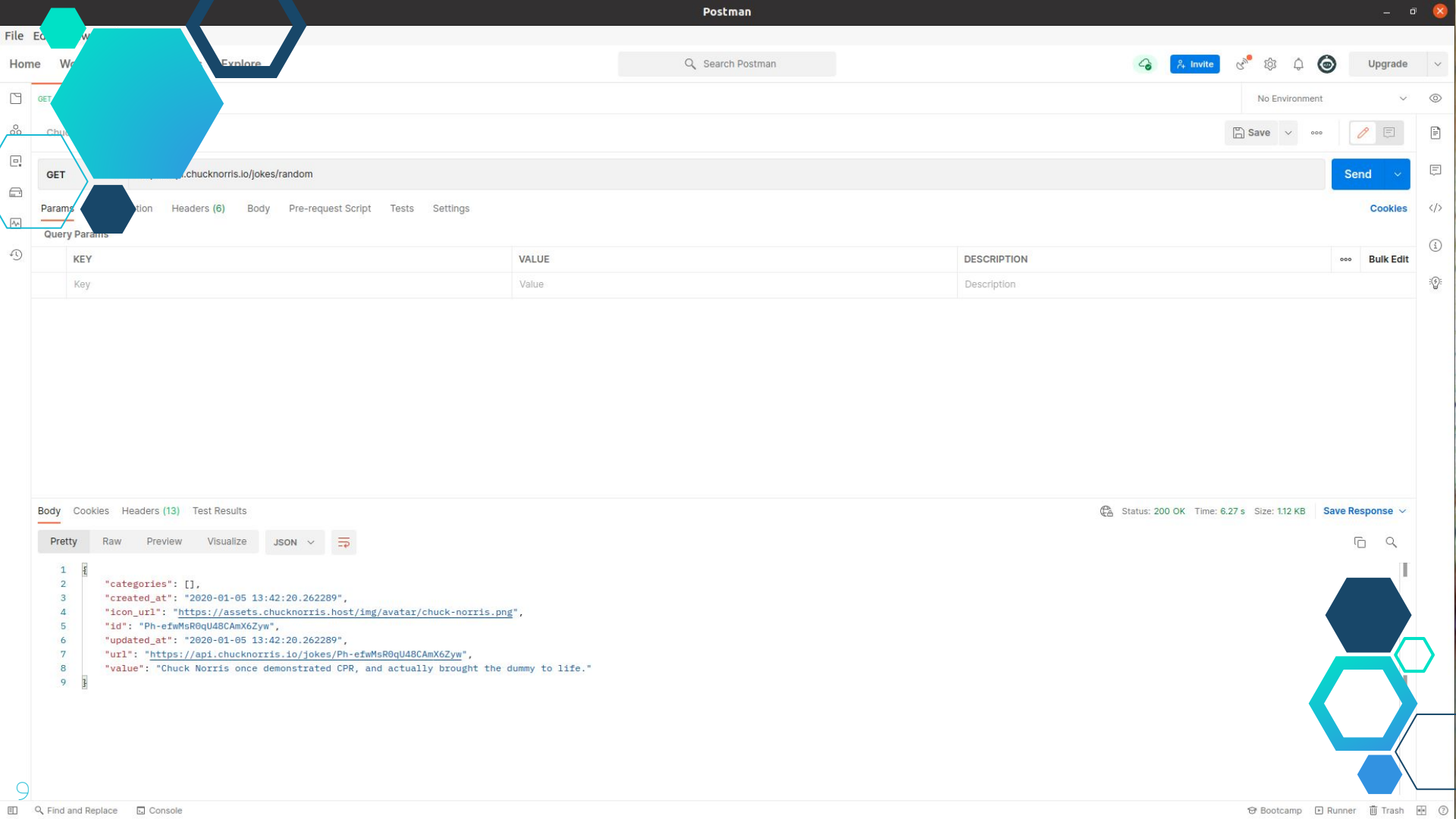
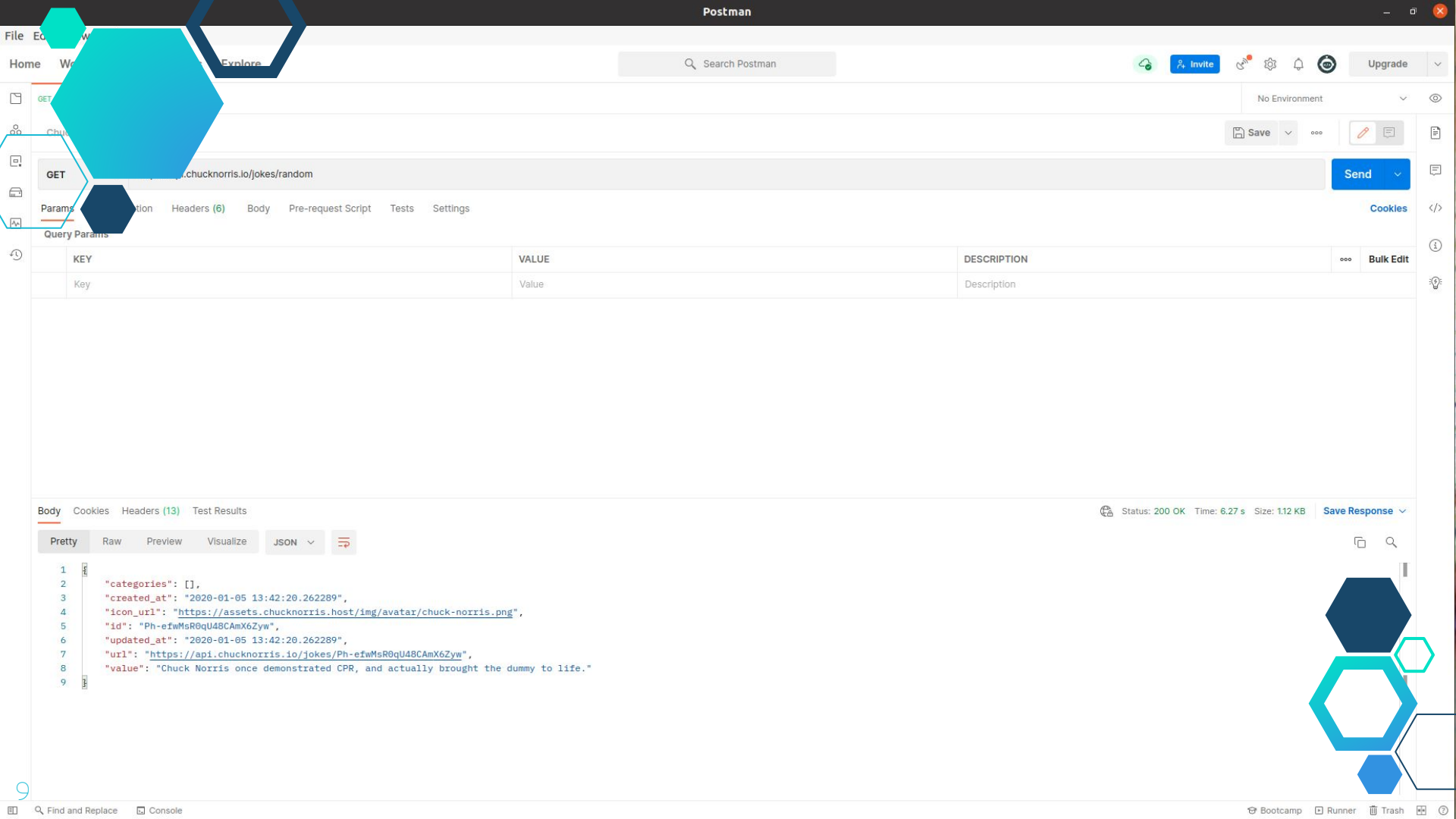
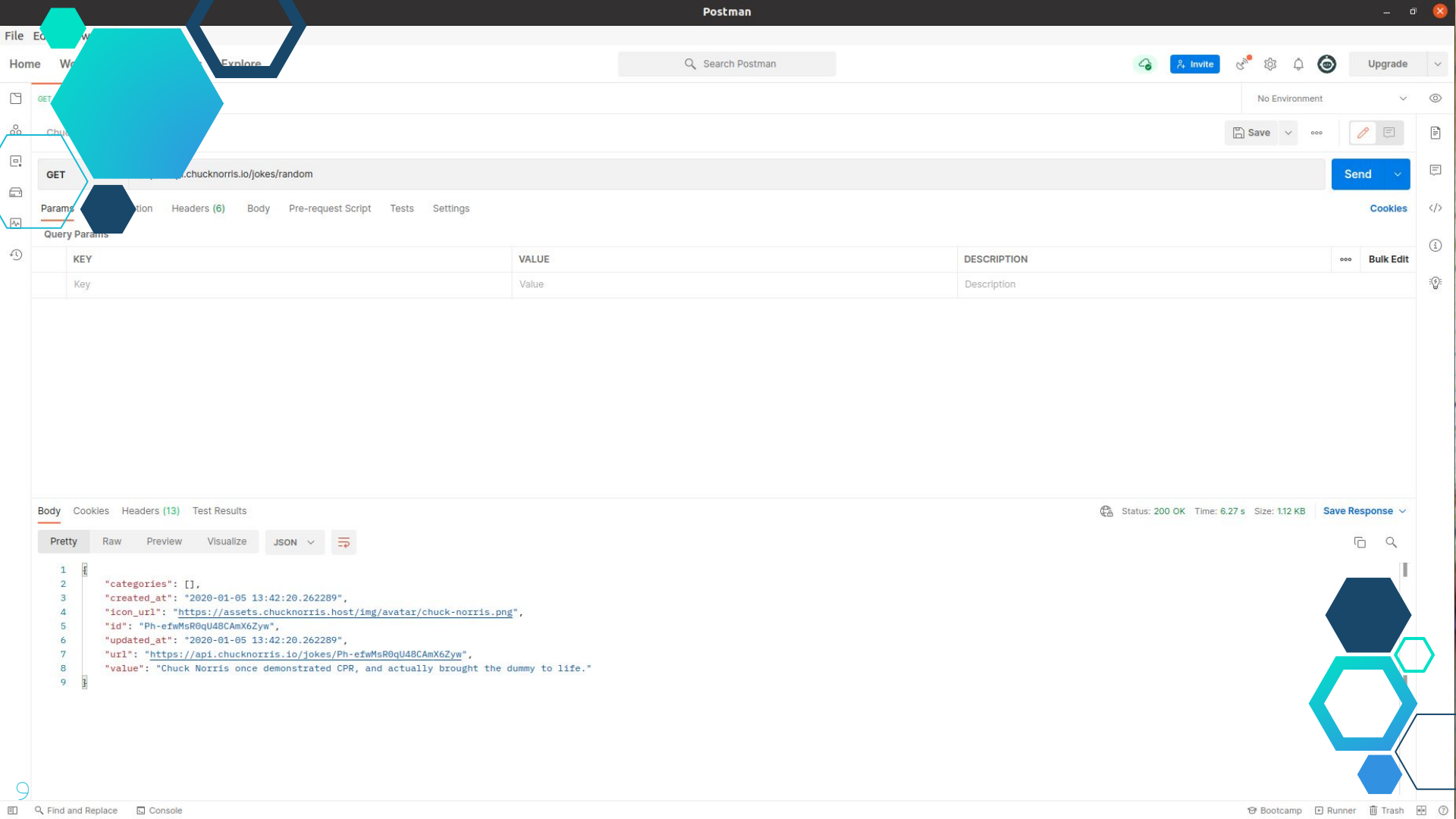


Contador de chistes

- ◆ Endpoint: <https://api.chucknorris.io/jokes/random>
- ◆ Método: GET
- ◆ Respuesta esperada:

```
{  
  "id": <string>,  
  "icon_url": <string>,  
  "url": <string>,  
  "value": <string>,  
  "updated_at": <string>,  
  "created_at": <string>,  
  "categories": <list<string>>  
}
```





A decorative pattern of hexagons in various shades of blue and cyan on the left side of the slide. Some hexagons contain icons: a lightbulb, a thumbs up, a network node, a smartphone, a magnifying glass, a gear, and a speech bubble.

2

FastAPI

Un framework creado para desarrollar web APIs de manera rápida y eficiente



FastAPI

- Es un framework de python para desarrollo de web apis
- Utiliza [Starlette](#) para resolver la capa web y [Pydantic](#) para la manipulación de datos
- Es eficiente ya que ejecuta los pedidos usando async/await.
- Soporta múltiples tecnologías actuales como: websockets, GraphQL, SQL and NO SQL databases, etc.
- Valida automáticamente los tipos de entrada y salida de los métodos.
- Generación automática de documentación
- Basada en estándares ([OpenAPI](#))
- Documentación oficial: <https://fastapi.tiangolo.com/>





Comenzando

Definamos nuestro primer programa

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

```
@app.get("/")  
async def root():  
    return {"message": "Hello World"}
```





Vamos a ejecutar

Para ejecutar corremos en la terminal el comando `uvicorn` como vemos en el ejemplo

```
user@admin:$ uvicorn main:app --reload
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [7773] using statreload
INFO:     Started server process [7775]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```





Probando

Para probar nuestros métodos vamos a usar la aplicación de línea de comando “curl”. Probemos el Hello World!

```
|user@admin:$ curl -X GET "http://localhost:8000/" -H "accept: application/json" -w "\n" {"message": "Hello World"}
```





Path parameters

Podemos pasar parámetros a nuestros métodos a través de la url. A estos parámetros los llamamos “PATH parameters”. Usando “type hinting” y pydantic FastAPI validará los tipos de estos parámetros. Pensemos en un ejemplo:

```
from fastapi import FastAPI, HTTPException
from users import USERS, get_user_by_id
```

```
app = FastAPI()
```

```
@app.get("/users/{user_id}")
```

```
async def users(user_id: int):
```

```
    return get_user_by_id(user_id=user_id)
```





Path parameters

El resultado será:

```
user@admin:$ curl -X GET "http://localhost:8000/users  
/1" -H "accept: application/json" -w "\n" -i  
HTTP/1.1 200 OK  
date: Sun, 06 Sep 2020 21:18:50 GMT  
server: uvicorn  
content-length: 99  
content-type: application/json  
  
{"id":1,"name":"Clark","surname":"Kent","age":37,"mai  
l":"superman@justiceleague.com","active":true}
```





Path parameters

Si intentamos obtener un user con un string en lugar de un int fastapi responderá con un error:

```
user@admin:$ curl -X GET "http://localhost:8000/users/clark" -H "accept: application/json" -w "\n" -i
HTTP/1.1 422 Unprocessable Entity
date: Sun, 06 Sep 2020 21:22:27 GMT
server: uvicorn
content-length: 104
content-type: application/json

{"detail":[{"loc":["path","user_id"],"msg":"value is not a valid integer","type":"type_error.integer"}]}
```





Path parameters

Podemos usar cualquier tipo definido en python como por ejemplo: int, str, float, etc. Inclusive podemos utilizar enumerados con valores definidos por nosotros mismos.

NOTA: Debemos tener cuidado en el orden que definimos los métodos. Otro método podría “pisar” un “Path parameter”





Path parameters

```
from enum import Enum
```

```
app = FastAPI()
```

```
class AvatarColor(str, Enum):
```

```
    red = "red"
```

```
    green = "green"
```

```
    blue = "blue"
```

```
@app.get("/users/{user_avatar_color}")
```

```
async def users(user_avatar_color: AvatarColor):
```

```
    return get_user_by_avatar(user_av=user_avatar_color)
```





Path parameters

¿Que ocurre si buscamos un usuario que no existe en el sistema?





Path parameters

```
user@admin:$ curl -X GET "http://localhost:8000/users/7" -H "accept: application/json" -w "\n" -i
HTTP/1.1 200 OK
date: Sun, 06 Sep 2020 22:30:59 GMT
server: uvicorn
content-length: 4
content-type: application/json

null
```

Este no es el resultado “esperado”. Esperaría que devuelva un 404 en caso de no encontrar el usuario y no un 200 con datos “null”





Error handling

Manejar errores es sencillo solo tenemos que seguir la semántica de excepciones.

Veamos el caso anterior pero controlando que si el usuario no existe devolvamos el correspondiente error





Error Handling

```
from fastapi import FastAPI, HTTPException, status
from users import USERS, get_user_by_id
```

```
app = FastAPI()
```

```
@app.get("/users/{user_id}")
async def users(user_id: int):
    res_user = None
    res_user = get_user_by_id(user_id=user_id)
    if not res_user:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="user not found"
        )
    return res_user
```





Error Handling

Ahora probemos con el mismo ejemplo anterior:

```
user@admin:$ curl -X GET "http://localhost:8000/users  
/7" -H "accept: application/json" -w "\n" -i  
HTTP/1.1 404 Not Found  
date: Sun, 06 Sep 2020 22:37:43 GMT  
server: uvicorn  
content-length: 27  
content-type: application/json
```

```
{"detail": "user not found"}
```

En detail podemos incluir cualquier tipo de objeto python serializable Dict, List, float, etc





Query Parameters

Query parameters son parámetros adicionales que podemos pasarle a una llama de la API. Por ejemplo si tenemos el endpoint users que retorna todos los usuarios podríamos querer limitar que solo se muestren los primeros N usuarios o incluso que se vean desde una posición determinada y hasta un índice dado.

```
@app.get("/users/")
async def get_user_list(
    user_from: Optional[int] = 0,
    user_to: Optional[int] = None
):
    return USERS[user_from:user_to]
```





Query Parameters

```
@app.get("/users/")
async def get_user_list(
    user_from: Optional[int] = 0,
    user_to: Optional[int] = None
):
    return USERS[user_from:user_to]
```

Aqui **user_from** y **user_to** son query parameters enteros que al tener valor por defecto, pueden no estar presentes en el request





Query Parameters

Al ejecutar obtenemos:

```
user@admin:$ curl -X GET "http://localhost:8000/users  
/?user_from=0&user_to=2" -H "accept: application/json"  
-w "\n" -i  
HTTP/1.1 200 OK  
date: Sun, 06 Sep 2020 23:29:37 GMT  
server: uvicorn  
content-length: 199  
content-type: application/json
```

```
[{"id":1,"name":"Clark","surname":"Kent","age":37,"ma  
il":"superman@justiceleague.com","active":true},{  
"id":2,"name":"Bruno","surname":"Diaz","age":34,"mail":"b  
atman@justiceleague.com","active":true}]
```





Query Parameters

Si los query parameters no tienen un valor por defecto, entonces FastAPI asume que son REQUERIDOS y si no están presentes en la URL generarán un error

```
@app.get("/users/")
async def get_user_list(
    user_from: int,
    user_to: int
):
    return USERS[user_from:user_to]
```





Query Parameters

```
user@admin:$ curl -X GET "http://localhost:8000/users  
/" -H "accept: application/json" -w "\n" -i  
HTTP/1.1 422 Unprocessable Entity  
date: Sun, 06 Sep 2020 23:38:17 GMT  
server: uvicorn  
content-length: 174  
content-type: application/json
```

```
{"detail":[{"loc":["query","user_from"],"msg":"field  
required","type":"value_error.missing"}, {"loc":["quer  
y","user_to"],"msg":"field required","type":"value_er  
ror.missing"}]}
```





Query Parameters

Si definimos que el tipo de query parameter es “bool” entonces FastAPI interpretará como bool cualquiera de las siguientes expresiones:

- ◇ True, False
- ◇ yes,no
- ◇ true,false
- ◇ 1,0
- ◇ on,off

Query Parameters se pueden combinar con Path Parameters sin problemas





Request Body

Ahora vamos a construir requests un poco más complejas que envían datos al servidor. Necesitamos un modelo para validar la estructura de los datos requeridos para un usuario.

FastAPI utiliza Pydantic para definir modelos. Al definir un modelo para una variable, FastAPI validará el contenido y se encargará de deserializar/serializar los objetos

Creemos un modelo para usuario





Request Body

Modelo para usuario

```
from pydantic import BaseModel, EmailStr

class UserIn(BaseModel):
    name: str
    surname: str
    age: Optional[int] = None
    mail: EmailStr
    active: bool
```





Request Body

```
class UserIn(BaseModel):  
    name: str  
    surname: str  
    age: Optional[int] = None  
    mail: EmailStr  
    active: bool
```

Notemos que mail no es simplemente un string, es un campo EmailStr de pydantic. Pydantic se encargará de validar automáticamente que mail además de ser un string, tiene el formato esperado.





Request Body

Y ahora que tenemos el modelo definido podemos agregar un nuevo método para crear un usuario nuevo

```
@app.post("/users/")
async def create_user(new_user: UserIn) -> int:
    new_id = len(USERS) + 1
    user_dict = new_user.dict()
    user_dict.update({"id": new_id})
    USERS.append(user_dict)
    return new_id
```





Request Body

Probemos ahora el método post

```
user@admin:$ curl -X POST "http://localhost:8000/users/" -H "accept: application/json" -w "\n" -i -d "{\n  \"name\": \"Carlos\", \"surname\": \"Gonzales\", \"age\": 23, \"mail\": \"cgonzales@gmial.com\", \"active\": true}\n\" HTTP/1.1 200 OK\n  date: Mon, 07 Sep 2020 00:44:21 GMT\n  server: unicorn\n  content-length: 1\n  content-type: application/json
```

5





Request Body

Si intentamos crear un usuario y el mail no es un mail válido se producirá un error

```
user@admin:$ curl -X POST "http://localhost:8000/users/" -H "accept: application/json" -w "\n" -i -d '{"name": "Carlos", "surname": "Gonzales", "age": 23, "mail": "cgonzales", "active": true}'  
HTTP/1.1 422 Unprocessable Entity  
date: Mon, 07 Sep 2020 00:45:21 GMT  
server: uvicorn  
content-length: 106  
content-type: application/json
```

```
{"detail": [{"loc": ["body", "mail"], "msg": "value is not a valid email address", "type": "value_error.email"}]}
```





Request Body

Si intentamos crear un usuario con algún campo requerido faltante también se generará un error

```
user@admin:$ curl -X POST "http://localhost:8000/users/" -H "accept: application/json" -w "\n" -i -d "{\n  \"name\": \"Carlos\", \"age\": 23, \"mail\": \"cgonzales@gmail.com\", \"active\": true}\n\" HTTP/1.1 422 Unprocessable Entity\n  date: Mon, 07 Sep 2020 00:45:50 GMT\n  server: uvicorn\n  content-length: 91\n  content-type: application/json
```

```
{\"detail\": [{\"loc\": [\"body\", \"surname\"], \"msg\": \"field required\", \"type\": \"value_error.missing\"}]}
```





Request Body

Actualmente estamos devolviendo un entero que indica el id del sujeto nuevo creado. Podemos devolver un modelo más complejo de salida y serializarlo usando un model diferente

```
class UserOut(BaseModel):  
    id: int  
    name: str  
    operation_result: str
```





Request Body

Y modificamos el método:

```
@app.post("/users/", response_model=UserOut)
async def create_user(new_user: UserIn) -> int:
    new_id = len USERS + 1
    user_dict = new_user.dict()
    user_dict.update({"id": new_id})
    USERS.append(user_dict)
    return UserOut(
        id=new_id,
        name=new_user.name,
        operation_result="Succesfully created!")
```





Request Body

Ahora al crear un nuevo user tenemos el siguiente resultado:

```
user@admin:$ curl -X POST "http://localhost:8000/users/" -H "accept: application/json" -w "\n" -i -d "{\"name\": \"Carlos\", \"surname\": \"Gonzales\", \"age\": 23, \"mail\": \"cgonzales@gmial.com\", \"active\": true}"
HTTP/1.1 200 OK
date: Mon, 07 Sep 2020 01:08:25 GMT
server: unicorn
content-length: 66
content-type: application/json

{"id":5,"name":"Carlos","operation_result":"Successfully created!"}
```





Request Body

Generalmente el status code de un “POST” exitoso suele ser 201 (created) y no 200.
¿Cómo cambiamos el código de éxito?





Request Body

```
@app.post(
    "/users/",
    response_model=UserOut,
    status_code=status.HTTP_201_CREATED
)

async def create_user(new_user: UserIn) -> int:
    new_id = len(USERS) + 1
    user_dict = new_user.dict()
    user_dict.update({"id": new_id})
    USERS.append(user_dict)
    return UserOut(
        id=new_id,
        name=new_user.name,
        operation_result="Successfully created!")
```





Request Body

El resultado ahora retorna 201:

```
user@admin:$ curl -X POST "http://localhost:8000/users/" -H "accept: application/json" -w "\n" -i -d "{\"name\": \"Carlos\", \"surname\": \"Gonzales\", \"age\": 23, \"mail\": \"cgonzales@gmial.com\", \"active\": true}"
HTTP/1.1 201 Created
date: Mon, 07 Sep 2020 01:18:21 GMT
server: uvicorn
content-length: 67
content-type: application/json

{"id":5,"name":"Carlos","operation_result":"Successfully created!"}
```





- Gracias a Starlette tenemos la clase `TestClient` que utiliza el módulo `requests`. Con dicha clase podemos escribir casos de prueba usando `pytest`.

Tené en cuenta que al momento de escribir tests reales quizás necesitemos manipular un poco los datos con los que trabaja FastAPI (mocks)





Testing

```
from fastapi.testclient import TestClient
from fastapi import status
```

```
from main import app
```

```
client = TestClient(app)
```

```
def test_get_single_user():
    response = client.get("/users/4")
    assert response.status_code == status.HTTP_200_OK
    assert response.json() == {
        'id': 4, 'name': 'Diana',
        'surname': 'Prince',
        'age': 28, 'mail':
        'wonderwoman@justiceleague.com', 'active': True
    }
```





Testing

```
user@admin:$ py.test -s test_users.py --disable-warnings
===== test session starts =====
platform linux -- Python 3.8.5, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /home/brujo/projects/facu/ingenieria/2020/FastAPI/examples/02_user
collected 1 item

test_users.py .

===== 1 passed, 6 warnings in 0.22s =====
```





Documentation

Una de las características más útiles de FastAPI es la generación automática de documentación de la API gracias a las definiciones de tipos y modelos realizados con Pydantic. Brinda 2 formatos de documentación:

- ◇ Swagger: <https://swagger.io/>
- ◇ ReDoc: <https://github.com/Redocly/redoc>

Para acceder a la documentación generada desde cualquier navegador:

- ◇ Swagger: `http://<ip>:<port>/docs`
- ◇ ReDoc: `http://<ip>:<port>/redoc`



🔍 Search...

GET Get User

GET Get User List

POST Create User

[Documentation Powered by ReDoc](#)

FastAPI (0.1.0)

Download OpenAPI specification: [Download](#)

Get User

PATH PARAMETERS

<div>user_id</div> <div>required</div>	integer (User Id)
--	-------------------

Responses

- > 200 Successful Response
- > 422 Validation Error

Get User List

QUERY PARAMETERS

user_from	Integer (User From) Default: 0
user_to	Integer (User To)

Responses

GET /users/{user_id}

Response samples

200

422

Content type

application/json

null

Copy Expand all Collapse all

GET /users/

Response samples

200

422

Content type

application/json

null

Copy Expand all Collapse all

FastAPI

0.1.0

OAS3

[/openapi.json](#)

default



GET

`/users/{user_id}` Get User

GET

`/users/` Get User List

POST

`/users/` Create User

Schemas



HTTPValidationError >

UserIn >

UserOut >

ValidationError >

POST**/users/** Create User**Parameters**[Try it out](#)

No parameters

Request body required

application/json

**Example Value** | [Schema](#)

```
{
  "name": "string",
  "surname": "string",
  "age": 0,
  "mail": "user@example.com",
  "active": true
}
```

Responses**Code****Description****Links**

201

Successful Response

No links

Media type

application/json

Controls Accept header.**Example Value** | [Schema](#)

```
{
  "id": 0,
  "name": "string",
  "operation_result": "string"
}
```

Execute Clear

Responses

Curl

```
curl -X POST "http://localhost:8000/users/" -H "accept: application/json" -H "Content-Type: application/json" -d '{"name":"string","surname":"string","age":0,"mail":"user@example.com","active":true}'
```

Request URL

http://localhost:8000/users/

Server response

Code

Details

201

Response body

```
{
  "id": 6,
  "name": "string",
  "operation_result": "Successfully created!"
}
```

Download

Response headers

```
content-length: 66
content-type: application/json
date: Mon07 Sep 2020 21:00:26 GMT
server: uvicorn
```

Responses

Code

Description

Links

201

Successful Response

No links

Media type

application/json

Controls Accept header.

Example Value

Schema

```
{
  "id": 0,
  "name": "string",
  "operation_result": "string"
}
```



Gracias!

Preguntas?





Ejercicios

- ◇ Agregar los métodos que faltan a user (PUT, DELETE)
- ◇ La clase usuario no tiene “username” y “password”.
Agregá estos campos cuidando que username sea único y que password no se guarde en texto plano. Ojo: quizás necesites un modelo distinto para la salida
- ◇ Investiga de qué manera se puede modularizar mejor el proyecto (routers)
- ◇ FastAPI permite manipular los headers del request.
Investiga cómo hacerlo y construí un ejemplo de lectura de headers.
- ◇ La documentación swagger que genera el proyecto se puede mejorar. Averiguá como. Ejemplo: agregar descripción de error 404 en GET





Links útiles

- ◇ **FastAPI tutorial** : <https://fastapi.tiangolo.com/tutorial/>
- ◇ **Starlette**: <https://www.starlette.io/>
- ◇ **Pydantic**: <https://pydantic-docs.helpmanual.io/>
- ◇ **Type Hints**: <https://realpython.com/lessons/type-hinting/>
- ◇ **Asyncio**: <https://realpython.com/async-io-python/>

