

Programming and Communications III: REST + Request + FastAPI




Jordi Ricard Onrubia Palacios

Departament d'Informàtica i Enginyeria Industrial Universitat de Lleida



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Programming and Communications III: REST + Request + FastAPI

{ REST }

- REST
 - ❑ Communication
 - ❑ Example
 - ❑ Operations
 - ❑ Codes
 - ❑ Constraints
- ❑ FastAPI
 - ❑ Overview
 - ❑ Installation
 - ❑ Operations
- ❑ Request
 - ❑ Python
 - ❑ Postman
 - ❑ Swagger UI

{ REST }



REST

A RESTful web application exposes information about itself in the form of information about its resources. Furthermore, enables the client to take actions on those resources in the form of CRUD actions, Create, Read, Update and Delete resources.

To develop a RESTful API we need to follow the REST constraints. These constraints make our APIs easier to use and easy to discover.



{ REST }



REST

REST stands for REpresentational State Transfer.

It means when a RESTful API is called, the server will transfer to the client a representation of the state of the requested resource.

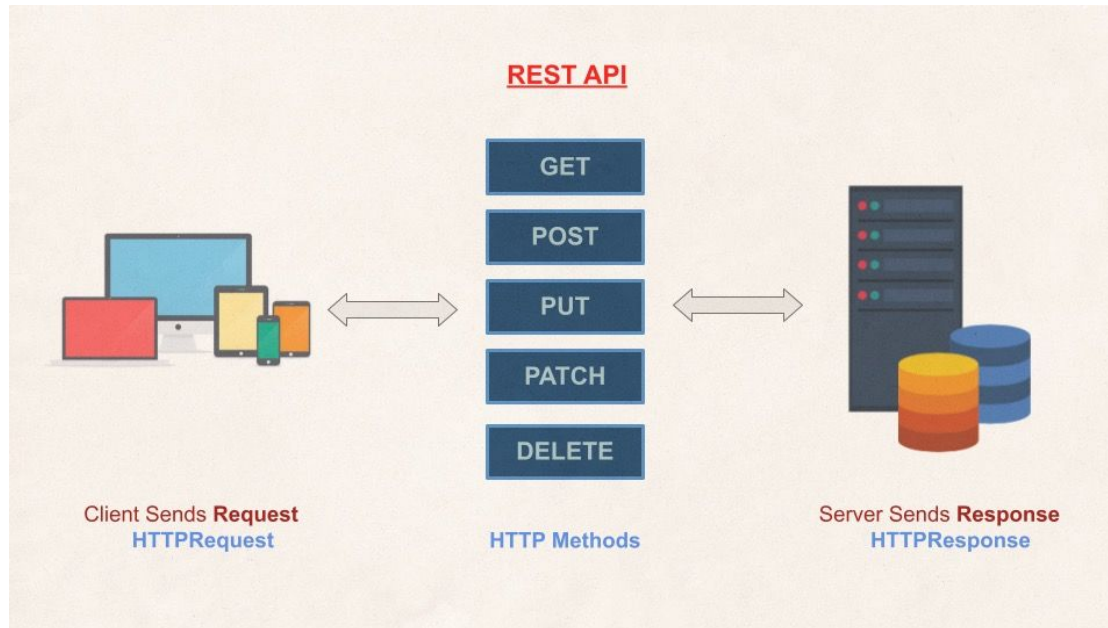
For example, when a developer calls Instagram API to fetch a specific user (the resource), the API will return the state of that user, including their name, the number of posts that user posted on Instagram so far, how many followers they have, and more.

Usually, the representation of the state is done in JSON format, although, we can use other representations such as XML.



{ REST }

REST: Communication



{ REST }



REST - Communication

How does the client sends a request to a server?

- It uses a URL (Uniform Resource Locator), also known as the endpoint to reach the server.
- Within the request it will send the desired operation GET, POST, PUT, PATCH and DELETE (these are the most common operations).
- The URL will also include the resource we want to access.



{ REST }



REST: Example

Example: If we want to access to Mr.Jagger twitter profile we have the following URL

<https://twitter.com/MisterJagger>

- As we are accessing to recover the profile, we are sending a GET request inside the HTTP request.
- The resource we want to recover is MisterJagger_, in this case this resource is the id of the user

We could translate this into, from twitter, show me the profile from MisterJagger_





REST: Operations

The most common operations are the following:

- GET - This method is used to retrieve a data from database / server.
- POST - This method is used to create a new record.
- PUT - This method is used to modify / replace the record. It replaces the entire record.
- PATCH - This method is used to modify / update the record. It replaces parts of the record.
- DELETE - This method is used to delete the record.





REST: Codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

- Informational responses (100–199)
- Successful responses (200–299)
- Redirection messages (300–399)
- Client error responses (400–499)
- Server error responses (500–599)

You can find more details on this page for specific codes:
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>



{ REST }



REST: Constraints

- Uniform interface
- Client — server separation
- Stateless
- Layered system
- Cacheable
- Code-on-demand



{ REST }



REST: Constraints - Uniform interface

It has 4 parts:

1. The request to the server has to include a resource identifier.
2. The response the server returns include enough information so the client can modify the resource.
3. Each request to the API contains all the information the server needs to perform the request, and each response the server returns contain all the information the client needs in order to understand the response.



{ REST }



REST: Operations - Uniform interface

4. Hypermedia as the engine of application state:

This means that the server can inform the client , in a response, of the ways to change the state of the web application.

If the client asked for a specific user, the server can provide not only the state of that user but also information about how to change the state of the user, for example how to update the user's name or how to delete the user.



{ REST }

REST: Constraints - Client — server separation

The client and the server act independently.

The interaction between the client and the server is only in the form of requests, initiated by the client only, and responses, delivered by the server as a consequence of the clients requests.



{ REST }



REST: Constraints - Stateless

Stateless means the server does not remember anything about the user who uses the API.

Each individual request contains all the information the server needs to perform the request and return a response, regardless of other requests made by the same API user.

{ REST }



REST: Constraints - Layered system

Between the client who requests a representation of a resource's state, and the server who sends the response back, there might be a number of servers in the middle.

Security layer, a caching layer, a load-balancing layer, or other functionality.

Those layers should not affect the request or the response. The client is agnostic as to how many layers, there are between the client server.





REST: Constraints - Cacheable

This means that the data the server sends contain information about whether or not the data is cacheable.

Cacheable data might contain a version number, this makes possible to know whether the data is new or not, furthermore, this data has to contain an expiration date.






REST: Constraints - Code-on-demand

This constraint is optional — an API can be RESTful even without providing code on demand.

The client can request code from the server, and then the response from the server will contain some code, usually in the form of a script, when the response is in HTML format. The client then can execute that code.





Programming and Communications III: REST + Request + FastAPI



- ❑ REST
 - ❑ Communication
 - ❑ Example
 - ❑ Operations
 - ❑ Codes
 - ❑ Constraints
- FastAPI
 - ❑ Overview
 - ❑ Installation
 - ❑ Operations
- ❑ Request
 - ❑ Python
 - ❑ Postman
 - ❑ Swagger UI

A horizontal bar with a teal segment on the left and an orange segment on the right.

Overview

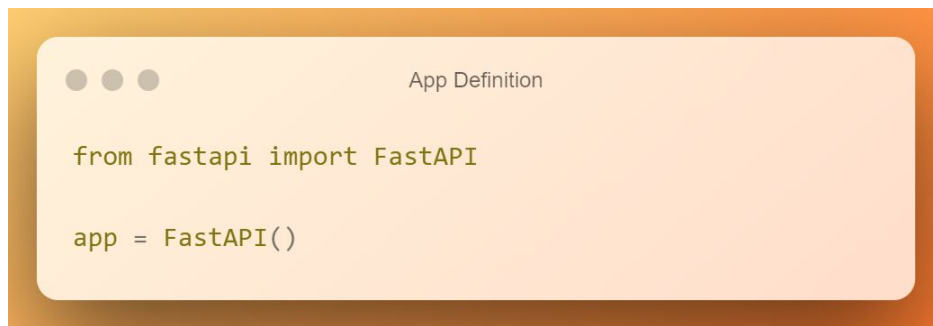
- FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.
- Software designed to support the development of web applications including web services, web resources, and web APIs.
- Provide a standard way to build and deploy web applications
- Aim to automate the overhead associated with common activities performed in web development providing libraries for database access, templating frameworks, and session management, and promoting code reuse.

A horizontal bar with a teal segment on the left and an orange segment on the right.

Installation

- Create a virtual environment (Not mandatory, but recommended)
 - `python3 venv -m <path/to/install/env>`
 - `source env/bin/activate` (linux)
 - `env/scripts/activate` (windows)
- Install FastAPI
 - `pip install fastapi`
- Install Uvicorn
 - `pip install "uvicorn[standard]"`

Operations: App Definition



```
from fastapi import FastAPI

app = FastAPI()
```

- FastAPI is a Python class that provides all the functionality for your API.
- Here the app variable will be an "instance" of the class FastAPI. This will be the main point of interaction to create all your API. This app is the same one referred by uvicorn in the execution command.

Operations: Get All

- To access all users we define a get using the app as follows
- `@app.get("/users")`
- This means that if we access to `http://<url>/users` with a get trigger the function below returning all users
- Here users would be the resource and get the verb

```
Get All

@app.get("/users")
def get_all():
    return [User(user_id=1, id=1, title='user1_id_1', body='descr_1_1'),
            User(user_id=1, id=2, title='user1_id_2', body='descr_1_2'),
            User(user_id=1, id=3, title='user1_id_3', body='descr_1_3'),
            User(user_id=2, id=1, title='user2_id_1', body='descr_2_1'),
            User(user_id=2, id=2, title='user2_id_2', body='descr_2_2'),
            User(user_id=3, id=3, title='user3_id_3', body='descr_3_3'),
            User(user_id=3, id=1, title='user2_id_1', body='descr_2_1'),
            User(user_id=3, id=2, title='user2_id_2', body='descr_2_2'),
            User(user_id=3, id=3, title='user3_id_3', body='descr_3_3')]
```

Operations: Get Filter

- To access an specific user we would call users endpoint but, this time defining a parameter {user_id}
- This user id will be received as a parameter in the function below.

```
Get Filter

@app.get("/users/{user_id}")
async def get_filter(user_id:int):
    users = [User(user_id=1, id=1, title='user1_id_1', body='descr_1_1'),
             User(user_id=1, id=2, title='user1_id_2', body='descr_1_2'),
             User(user_id=1, id=3, title='user1_id_3', body='descr_1_3'),
             User(user_id=2, id=1, title='user2_id_1', body='descr_2_1'),
             User(user_id=2, id=2, title='user2_id_2', body='descr_2_2'),
             User(user_id=3, id=3, title='user3_id_3', body='descr_3_3'),
             User(user_id=3, id=1, title='user2_id_1', body='descr_2_1'),
             User(user_id=3, id=2, title='user2_id_2', body='descr_2_2'),
             User(user_id=3, id=3, title='user3_id_3', body='descr_3_3')]
    return [user for user in users if user.user_id == user_id]
```

Operations: Post



```
@app.post("/users/")
async def post_user(user: User):
    return user
```

- To create a user we can call the endpoint as we did before, but this time we define a post instead of a get, this way we are saying that we want to create and not recover a user.
- The user in this case is a class, when we send a JSON with the user parameters, FastAPI will map it to an instance and it will be received as parameter by the function.

Operations: Put

```
Put

@app.put("/users/")
async def update_user(user: User):
    return user
```

- To update a user we can call the endpoint as we did before, but defining a put.
- As we are talking about users we can do the same and use a class so the request gets mapped and we can work easily.

Operations: Patch

```
● ● ● Patch

@app.patch("/users/")
async def patch_user(user: User):
    return user
```

- To patch a user we can call the endpoint as we did before, but defining a patch.
- As we are talking about users we can do the same and use a class so the request gets mapped and we can work easily.

Operations: Delete

```
Delete

@app.delete("/users/{user_id}")
async def delete_suer(user_id: int):
    return {'user_id': user_id}
```

- To delete a user we can call the endpoint as we did before, this time defining a delete.
- Usually the deletes can be done by specifying a whole user, or just the id, the second one is more efficient, specifically if we want to remove a lot of elements.

A horizontal bar with a teal segment on the left and an orange segment on the right.

Operations: Note

When we are defining endpoints we can define the same endpoint multiple times

- Get /users
- Post /users
- Put /users

The name is the same, but the verb changes, therefore FastAPI can differentiate them easily.

The name of each endpoint must be the most specific possible so it can be intuitive and ease the usage.

Execution

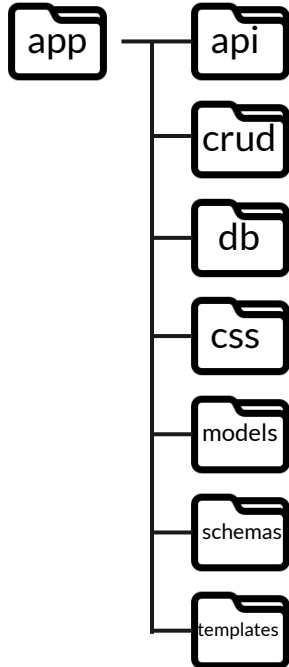
uvicorn <file.py containing the app>:app --reload



Starting FastAPI

```
(temp_env) PS C:\Users\Jordi\Desktop> uvicorn FastAPI_Example:app --reload
INFO:      Will watch for changes in these directories: ['C:\\Users\\Jordi\\Desktop']
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [23540] using StatReload
INFO:      Started server process [5788]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Project Organization



- API: Stores all route related code
- CRUD: Stores all CRUD DB related operations
- DB: Stores all DB connection operations
- CSS: Stores all style related code
- Models: Stores all DB related models
- Schemas: Stores all models related with the app
- Templates: Stores all html related code

Programming and Communications III:

REST + Request + FastAPI



Requests
http for humans



- ❑ REST
 - ❑ Communication
 - ❑ Example
 - ❑ Operations
 - ❑ Codes
 - ❑ Constraints
- ❑ FastAPI
 - ❑ Overview
 - ❑ Installation
 - ❑ Operations
- Request
 - ❑ Python
 - ❑ Postman
 - ❑ Swagger UI



Requests
http for humans

Requests: Python

Requests is a python library that allows us to send http requests easily

- `requests.get("url")`
- `requests.post("url",data)`
- `requests.put("url",data)`
- `requests.patch("url",data)`
- `requests.delete("url")`



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Requests
http for humans

Requests: Python - Get

```
get_all()

def get_all():
    resp = requests.get("http://jsonplaceholder.typicode.com/posts/")
    if resp.status_code != 200:
        # This means something went wrong.
        raise Exception('GET /tasks/{}'.format(resp.status_code))
    for element in resp.json():
        print(element)
```



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Requests
http for humans

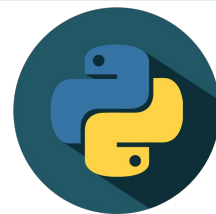
Requests: Python - Get

```
get_one()

def get_one():
    resp = requests.get("http://jsonplaceholder.typicode.com/posts/1")
    if resp.status_code != 200:
        # This means something went wrong.
        raise Exception('GET /tasks/{}'.format(resp.status_code))
    print(resp.json())
```



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Requests
http for humans

Requests: Python - Get



filter()

```
def filter():  
    resp = requests.get("http://jsonplaceholder.typicode.com/posts?userId=1")  
    if resp.status_code != 200:  
        # This means something went wrong.  
        raise Exception('GET /tasks/{}'.format(resp.status_code))  
    for element in resp.json():  
        print(element)
```



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



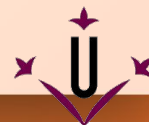
Requests
http for humans

Requests: Python - Post

```
post()

def post():
    data = {
        'id': 101,
        'title': 'foo',
        'body': 'bar',
        'userId': 1
    }

    resp = requests.post("http://jsonplaceholder.typicode.com/posts/", data=data)
    if resp.status_code != 201:
        raise Exception('POST /posts/{}'.format(resp.status_code))
    print('Created task. ID: {}'.format(resp.json()))
```



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Requests
http for humans

Requests: Python - Delete



delete()

```
def delete():  
    resp = requests.delete("http://jsonplaceholder.typicode.com/posts/1")  
    if resp.status_code != 200:  
        raise Exception('DELETE /posts/{}'.format(resp.status_code))  
    print('Deleted task. ID: {}'.format(resp.json()))
```



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Requests
http for humans

Requests: Postman

Postman is an API platform for building and using APIs.

It can be downloaded from: <https://www.postman.com/>

For this example we will need the json file uploaded at github:

[https://github.com/JordiROP/ProgComm-III/blob/main/REST Request FastAPI/Examples/Postman%20Request%20Examples.postman_collection.json](https://github.com/JordiROP/ProgComm-III/blob/main/REST%20Request%20FastAPI/Examples/Postman%20Request%20Examples.postman_collection.json)

This file is an export with all the methods previously seen

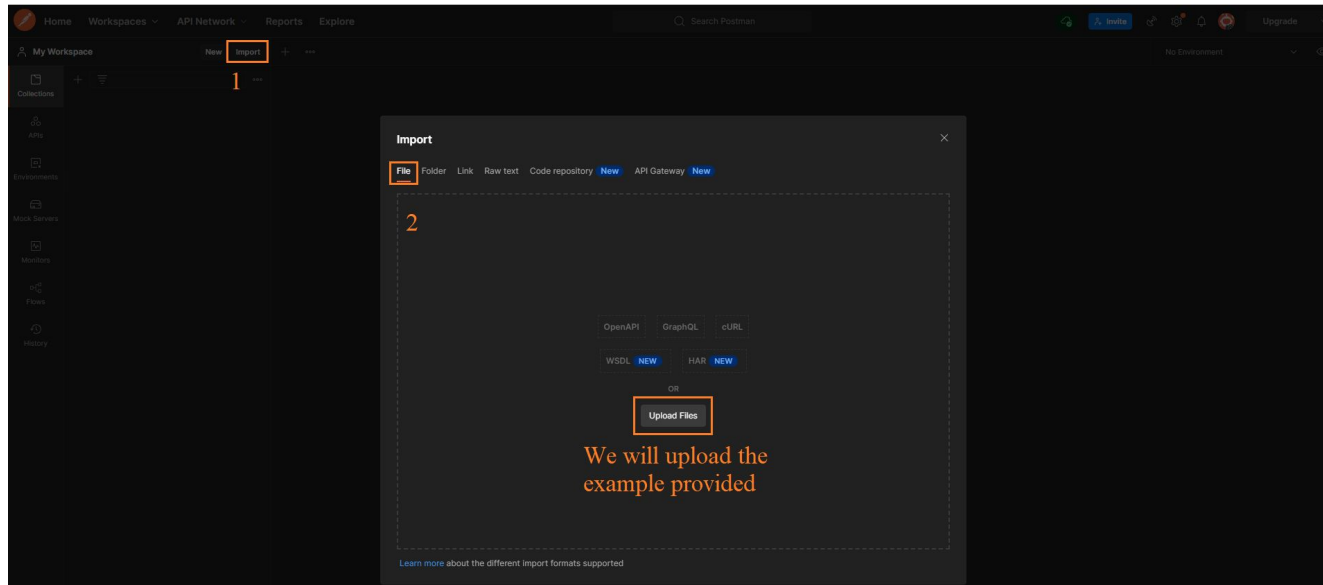


Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Requests: Postman



Requests
http for humans

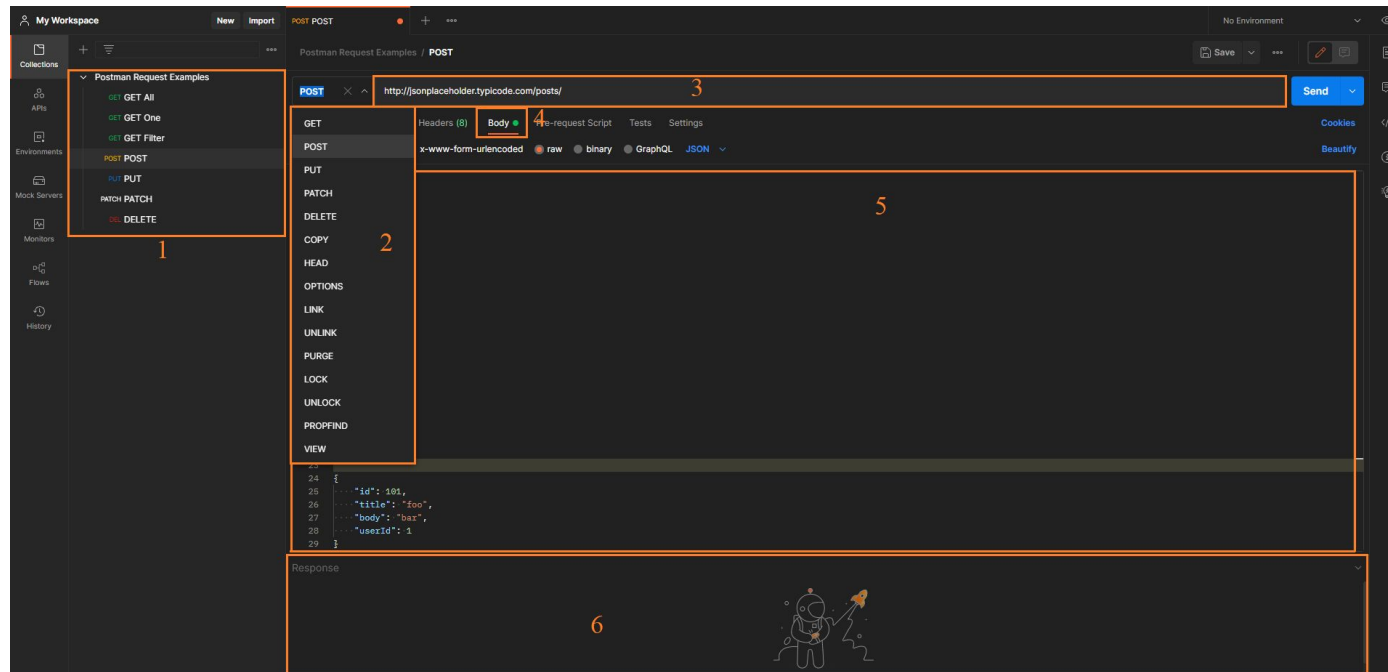


Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Requests
http for humans

Requests: Postman





Requests
http for humans

Requests: Postman

1. Requests inside our collection, we can add more clicking on the “+”
2. HTTP method of our request
3. URL for our request
4. Tab to add a body to our request
5. Body to enter the data we want to send with our request
6. Response of our request



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Requests
http for humans

Requests: Swagger

FastAPI provides automatic docs, using Swagger UI

Swagger UI allows anyone to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from your OpenAPI (formerly known as Swagger) Specification, with the visual documentation making it easy for back end implementation and client side consumption.



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial



Requests
http for humans

Requests: Swagger

By accessing to <http://127.0.0.1:8000/docs> once we have executed our FastAPI application we will see the Swagger UI related with our application.

In this interface we will find all the defined endpoints well documented, furthermore we will find all the classes that uses and their definition.



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

default



GET /users Get All



GET /users/{user_id} Get Filter



DELETE /users/{user_id} Delete Suer



PUT /users/ Update User



POST /users/ Post User



PATCH /users/ Patch User



Schemas



HTTPValidationError >

User >

ValidationError >



Requests

http for humans

GET

/users Get All

Parameters

No parameters

Cancel

Execute

Responses

Code

Description

Links

200

Successful Response

No links

Media type

application/json

Controls Accept header.

Example Value | Schema

"string"



Requests: Swagger



Requests
http for humans

For the first example we have the get that retrieves all users, to execute this endpoint we will have to press 1 so we can enable it, and 2 to execute it, the response provided by the endpoint will be found on the square with the 3 on top.



Requests

^mans

GET /users/{user_id} Get Filter

Parameters

Cancel

Name	Description
------	-------------

user_id * required integer (path)
--

Execute

Responses

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

200

Successful Response

No links

Media type

application/json

Controls Accept header.

Example Value | Schema

"string"



Requests
http for humans

Requests: Swagger

For the second example we have the same interface as before, but, this time we can see a parameter on it, this parameter is the one defined in the get with the filter to recover specific users.

Parameters will appear as we define them.

A similar interface would be shown in the delete endpoint.



Requests

http for humans

POST /users/ Post User

Parameters

Cancel

No parameters

Request body required

application/json

```
{
  "user_id": 0,
  "id": 0,
  "title": "string",
  "body": "string"
}
```

Execute

Responses

Code	Description
200	Successful Response

Links

No links



Requests
http for humans

Requests: Swagger

For the third and last example, we have the post endpoint, in this one we had defined as a parameter for the function a variable of type User, as this parameter is not defined in the url, FastAPI assumes that this parameter must be inside the body of the request.

Then, as we can see to introduce a JSON with the fields of our User, Swagger will gives us a box to send the JSON within the body of our request.

The same would be shown in case of the put and the patch.



More Info:

<https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f>

<https://www.numpyninja.com/post/rest-api-for-dummies-explained-using-mommies>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

<https://fastapi.tiangolo.com/>

<https://requests.readthedocs.io/en/latest/>

<https://www.postman.com/>

<https://github.com/swagger-api/swagger-ui>

https://github.com/JordiROP/ProgComm-III/tree/main/REST_Request_FastAPI

