BASICS OF FastAPI

ARIHARASUDHAN

FastAPI

It is a superfast Python Web Framework.Some saliant features of it are Automatic Docs (Swagger UI & ReDoc),Support for Modern Python Syntax. It is based on JSON Schema. It is built over the starlette web framework of Python.

Virtual Environment

First, let's create and activate a virtual environment using the following commands.

```
$ python3 venv fast
$ source ./fast/bin/activate
(fast)$
```

Install

Using the pip command, install the fastAPI and uvicorn.

```
(fast)$ pip3 install fastapi
(fast)$ pip3 install "uvicorn[standard]"
```

First App

main.py > ...

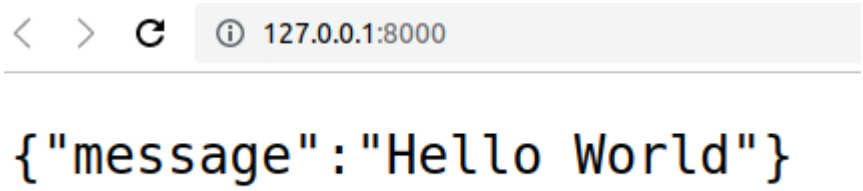
```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def myfunc():
    return {"message": "Hello World"}
```

Run It!

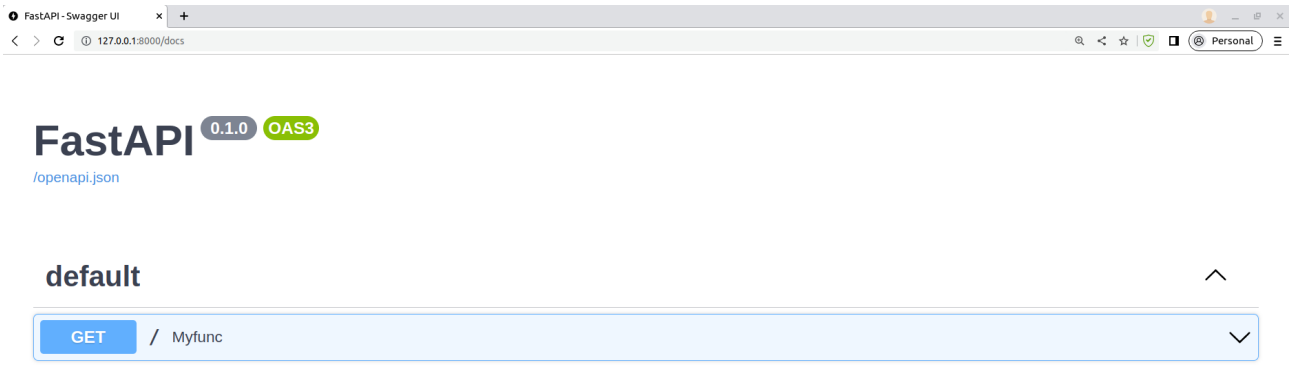
In the Terminal, type uvicorn and then the script name followed by a colon and app. If you specify --reload, we don't need to reload it manually.

```
(fast) sh-5.1$ uvicorn main:app --reload
INFO:     Will watch for changes in these directories: ['/home/ari-pt7127/fast']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [681] using WatchFiles
INFO:     Started server process [683]
INFO:     Waiting for application startup.
```

If we open the specified address, we can see, "Hello, World!"

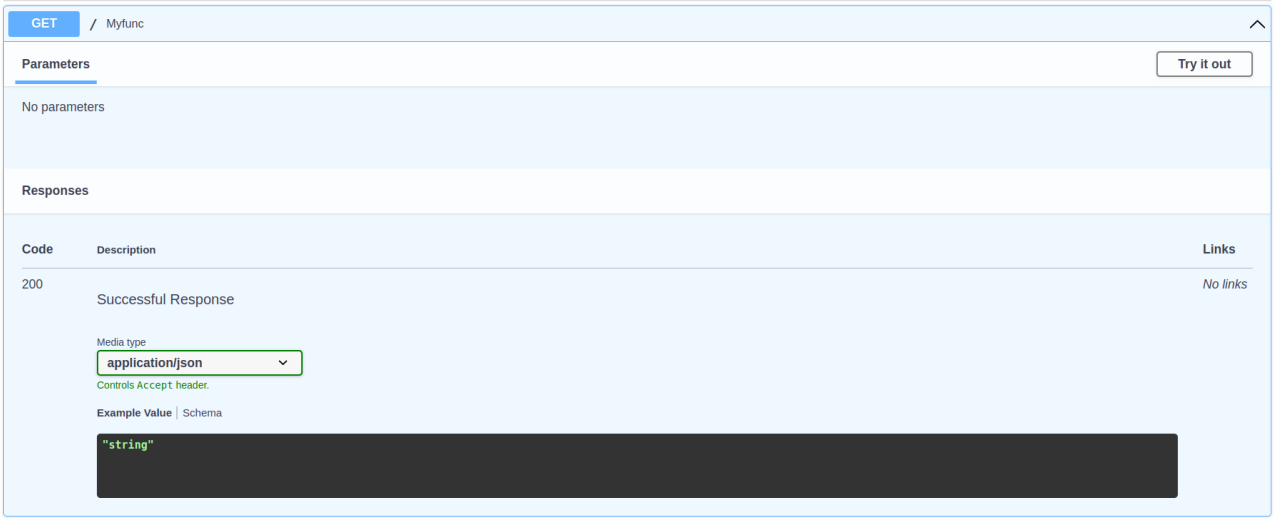


To open an interactive docs, we can go to, the address followed by docs.



Here, We can see the one and only GET request which will redirect us to the MyFunc.

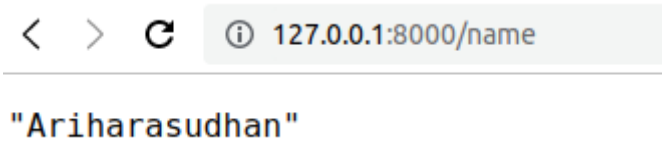
Here, we get a success message. It means , our GET works!



Yeah.. We can have multiple GET requests...

main.py > ...

```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def myfunc():
    return {"message": "Hello World"}

@app.get("/name")
def myfunc():
    return "Ariharasudhan"
```



"Ariharasudhan"

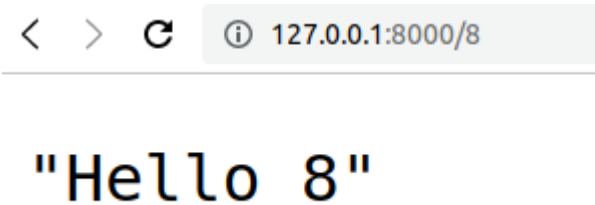
Some Cool Terminologies

Path : The routing path / Endpoint
HTTP Verb : Operation
Function : Path Operation Function
Decorator : Path Operation Decorator

Dynamic Routing using ID

```python
main.py > ...
from fastapi import FastAPI
app = FastAPI()
@app.get("/{id}")
def myfunc(id):
    return f'Hello {id}'
```

Use the curly braces to capture the ID value to perform dynamic routing.



Parameter Type

```python
main.py > ...
from fastapi import FastAPI
app = FastAPI()
@app.get("/{id}")
def myfunc(id:int):
    return f'Hello {type(id)
```

127.0.0.1:8000/8

"Hello <class 'int'>"

We can use the modern type specifying methods in Python. But, beware of.....

main.py > ...

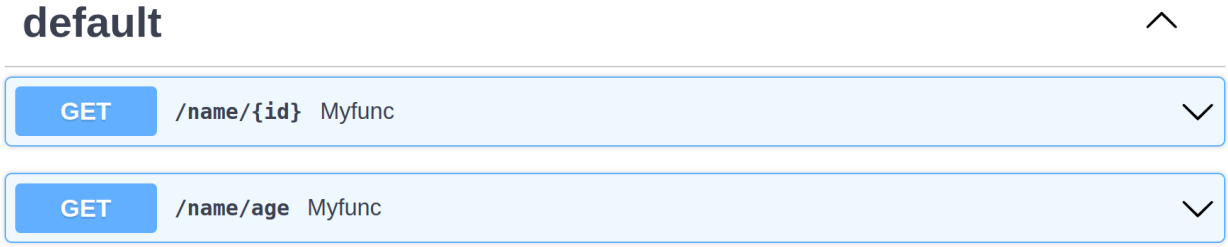
```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/name/{id}")
def myfunc(id:int):
    return f'Hello {type(id)}'

@app.get("/name/age")
def myfunc():
    return 'ARI 20'
```

127.0.0.1:8000/name/age

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

Here arises an anonymity since the path is matched wrongly. Here, We have to write the second myfunc() first. Beware of this sort of stuffs here.

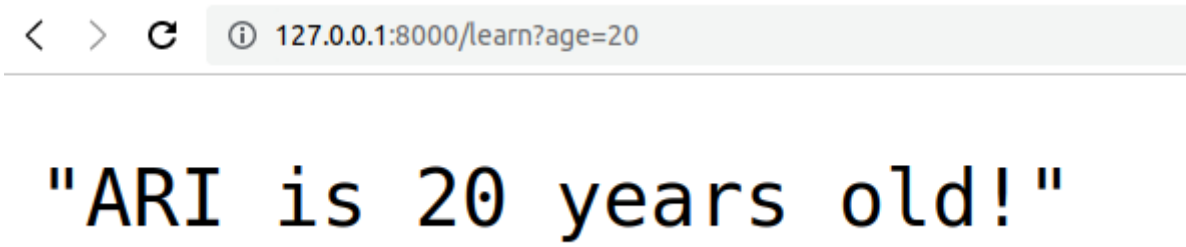
Now, since we have two routes, we can witness them there in the docs too.



Query Parameters

We can also use some query parameters in our program.

```
main.py > ...
from fastapi import FastAPI
app = FastAPI()

@app.get('/learn')
def myfunc(age:int):
    return f'ARI is {age} years old!'

```



We can specify any value to the age in the parameter and can get different outputs. Note ? Sign.

We can also have multiple query parameters. Let's do a conditional operation.

```python
main.py > ...
from fastapi import FastAPI
app = FastAPI()

@app.get('/learn')
def myfunc(age:int,show:bool):
    if show:
        return f'ARI is {age} years old!'
    else:
        return f'Can not show'
```

127.0.0.1:8000/learn?age=20&show=false

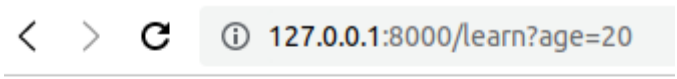
"Can not show"

It will show "Can not show" when we have the show parameter set to false. Use the & to give multiple parameters.

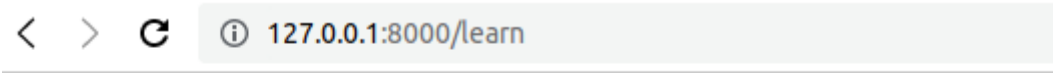
Optional Parameter

```python
main.py > ...
from fastapi import FastAPI
from typing import Optional
app = FastAPI()

@app.get('/learn')
def myfunc(age:Optional[int]=None):
    if age==None:
        return 'AGE IS NOT PROVIDED'
    return f'AGE IS {age}'
```

We can also use some query parameters in our program.



"AGE IS 20"



"AGE IS NOT PROVIDED"

Optional type is available in the typing module of Python.

Let's POST

Let's move to other operations.

```python
main.py > ...
from fastapi import FastAPI
from typing import Optional
app = FastAPI()

@app.post('/make')
def myfunc():
    return f'CREATED SOMETHING'

```

127.0.0.1:8000/make

{"detail":"Method Not Allowed"}

Yep! We can't expect what happened with GET. POST method is not allowed. But... Why worry ? We have docs.

POST /make Myfunc

Parameters

Cancel

No parameters

Execute

Code	Details
200	Response body "CREATED SOMETHING" Download

The response when we execute the POST request is “CREATED SOMETHING” But... If you want to send the request as request-body, we need to create model. That is exactly when we need Pydantic Module.

```python
main.py > ...
from fastapi import FastAPI
from typing import Optional
from pydantic import BaseModel

app = FastAPI()
class Body(BaseModel):
    name: str
    age: int

@app.post('/make')
def myfunc(body:Body):
    return body
```

Request body required

Example Value | Schema

```
{
  "name": "string",
  "age": 0
}
```

Now, our schema is ready!

Example Value | **Schema**

```
Body ⌄ {
    name*                    Name > [...]
    age*                     Age > [...]
}
```

Now, let's go a bit learny!

main.py > ...

```python
from fastapi import FastAPI
from typing import Optional
from pydantic import BaseModel

app = FastAPI()
class Body(BaseModel):
    name: str
    age: int

@app.post('/make')
def myfunc(body:Body):
    return f'Hello {body.name}! You are {body.age} years old!'

```

```
{
  "name": "ARI",
  "age": 20
}
```

Server response

Code	Details
200	Response body

```
"Hello ARI! You are 20 years old!"
```

We can keep our schemas in a separate file.

EXPLORER

FAST

- __pycache__
- bin
- include
- lib
- lib64
- __init__.py
- main.py
- pyvenv.cfg
- schemas.py

main.py | schemas.py

schemas.py > ...

```python
from pydantic import BaseModel

class Body(BaseModel):
    name: str
    age: int
```

EXPLORER

FAST

- __pycache__
- bin
- include
- lib
- lib64
- __init__.py
- main.py
- pyvenv.cfg
- schemas.py

main.py | schemas.py

main.py > myfunc

```python
from fastapi import FastAPI
import schemas
app = FastAPI()

@app.post('/make')
def myfunc(body:schemas.Body):
    return f'Hello {body.name}! You are {body.age} years old!'
```

ORM : Object Relational Mapper(Maps the object into a db table)

MongoDB and FastAPI

Let's connect with MongoDB. Have a .env file to keep the variables. Use it and configure it in our main script to connect. Import the router and include it.

.env

```
ATLAS_URI=mongodb+srv://ariharasudhan:
DB_NAME=animals
```

main.py

```
from fastapi import FastAPI
from dotenv import dotenv_values
from pymongo import MongoClient
from routes import router as animal_router
import uvicorn

config = dotenv_values(".env")
app = FastAPI()

@app.get('/')
def index():
    return 'WELCOME'

@app.on_event("startup")
def startup_db_client():
    app.mongodb_client = MongoClient(config["ATLAS_URI"])
    app.database = app.mongodb_client[config["DB_NAME"]]

@app.on_event("shutdown")
def shutdown_db_client():
    app.mongodb_client.close()

app.include_router(animal_router, prefix="/animals")

if __name__ == "__main__":
uvicorn.run("main:app", host="0.0.0.0", port=3000,
reload=True)
```

Follow the following structure.

```
.env
main.py
models.py
routes.py
```

.env

```
ATLAS_URI=mongodb+srv://<name>:<password>@connection
DB_NAME=animals
```

models.py

```
from typing import Optional
from pydantic import BaseModel

class Animal(BaseModel):
    name: str
    color: str

class AnimalUpdate(BaseModel):
    name: str
    color: str
```

Let’s define a POST Method in routes.py

```
from fastapi import APIRouter, Body, Request
from fastapi.encoders import jsonable_encoder
from models import Animal
router = APIRouter()

@router.post("/")
def create_animal(request: Request, animal: Animal):
    animal = jsonable_encoder(animal)
    new_animal =
request.app.database["animals"].insert_one(animal)
    return "DONE"
```

Go to the docs page. Click the POST button and try it out. Define the request body.

```
{
  "name": "TIGER",
  "color": "ORANGE"
}
```

We have used the jsonable_encoder in our code for JSON Support. Now, click EXECUTE.

Code	Details

200	Response body
	`"DONE"`
	Download
	Response headers
	`content-length: 6`
	`content-type: application/json`
	`date: Tue,21 Feb 2023 08:00:17 GMT`
	`server: uvicorn`

Responses

Code	Description	Links
200	Successful Response	*No links*

It is a SUCCESS Message! Now, let's check out our database.
YES! We are inserting animal and color.

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('63f47a1216af0dbd61c59d4a')
name: "TIGER"
color: "ORANGE"
```

Now, let's insert more values and do a GET Request.Use the Types updated!

routes.py > ...

```python
from fastapi import APIRouter, Body, Request
from fastapi.encoders import jsonable_encoder
from models import Animal
from typing import List
router = APIRouter()

@router.post("/",response_model=Animal)
def create_animal(request: Request, animal: Animal):
    animal = jsonable_encoder(animal)
    new_animal = request.app.database["animals"].insert_one(animal)
    return "DONE"

@router.get("/",response_model=List[Animal])
def get_animal(request: Request):
    animals = list(request.app.database["animals"].find(limit=3))
    return animals

```

When we give a GET request, we will get the following response.

200

Response body


```json
[
  {
    "name": "TIGER",
    "color": "ORANGE"
  },
  {
    "name": "PANTHER",
    "color": "BLACK"
  },
  {
    "name": "LION",
    "color": "YELLOW"
  }
]
```

If we want to get a selected item, we have to use any of the parameter. Suppose, if we want to access the item with the name Tiger, let' go...

```python
@router.get("/{name}", response_model=Animal)
def find_book(name: str, request: Request):
    if (animal := request.app.database["animals"].find_one({"name": name})) is not None:
        return animal
    raise HTTPException(status_code=404, detail="Not found")
```

DOCS :

GET /animals/{name} Find Book

Parameters

Name	Description
name * required string (path)	TIGER

RESPONSE :

Server response

Code	Details
200	Response body

```
{
  "name": "TIGER",
  "color": "ORANGE"
}
```

Let's do PUT Operation in order to update an existing element. If we want to change the PANTHER into PINK PANTHER.... First, add a PUT Handler in the routes.

```
@router.put("/{name}", response_model=AnimalUpdate)
def update_animal(name: str, request: Request, animal: AnimalUpdate):
    animal = {k: v for k, v in animal.dict().items() if v is not None}
    if len(animal) >= 1:
        update_result = request.app.database["animals"].update_one(
            {"name": name}, {"$set": animal}
        )
```

In the docs, the PUT is visible using which we can enter the name and modify the field.

Parameters

Name	Description
name * required string *(path)*	PANTHER

Request body required

```
{
  "name": "PINK-PANTHER",
  "color": "PINK"
}
```

Now, The Black is Pink.

200

Response body

```
[
  {
    "name": "TIGER",
    "color": "ORANGE"
  },
  {
    "name": "PINK-PANTHER",
    "color": "PINK"
  },
  {
    "name": "LION",
    "color": "YELLOW"
  }
]
```

One more thing left! How to DELETE ? SIMPLE AS ALL THE ABOVE.

```
router.delete("/{name}")
def delete_animal(name: str, request: Request):
    delete_result = request.app.database["animals"].delete_one({"name": name})
    if delete_result.deleted_count == 1:
        return 'FINE'
    raise HTTPException(status_code=404, detail="Not found")
```

Let's delete LION!

DELETE /animals/{name} Delete Animal

Parameters

Cancel

Name	Description
name * required string (path)	LION

Execute | Clear

We are done with CRUD Operations.

GET	/	Index
GET	/animals/	Get Animal
POST	/animals/	Create Animal
GET	/animals/{name}	Find Book
PUT	/animals/{name}	Update Animal
DELETE	/animals/{name}	Delete Animal

Templates

You can use any template engine you want with FastAPI. Let's go with JINJA.

First, let's install it.

```
(fast)$ pip3 install jinja2
```

Using Templates

To use the templates,

Import Jinja2Templates. Create a templates object that you can re-use later. Declare a Request parameter in the *path operation* that will return a template. Use the templates you created to render and return a TemplateResponse, passing the request as one of the key-value pairs in the Jinja2 "context".

We need the following libraries.

```
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
```

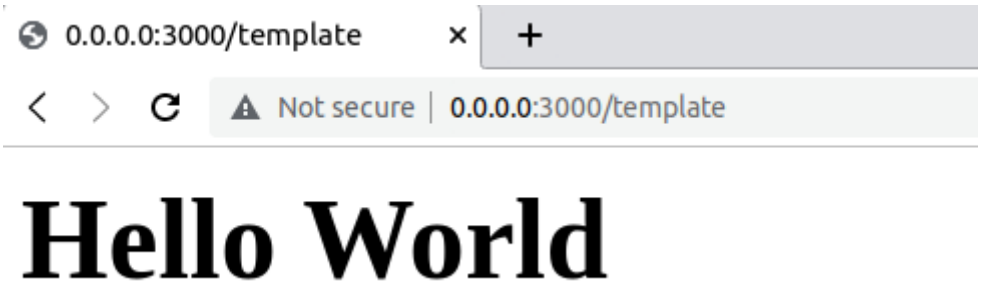
The most basic thing is we need a template and a static folder. We have to connect it like,

```
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")
```

Save the HTML Files inside the template folder.

Now, let's return the HTMLResponse.

```
@app.get("/template/{name}", response_class=HTMLResponse)
def read_item(request: Request,name: str):
    return templates.TemplateResponse("index.html", {"request":request,"name": name})
```



Template Manipulation

Now, let's change it a bit.

```
@app.get("/template/{name}", response_class=HTMLResponse)
def read_item(request: Request,name: str):
    return templates.TemplateResponse("index.html", {"name": name})
```

We are about to receive a name and display it in the template using JINJA Template Engine.

```
templates > index.html > ...
<html>
<head>
    <title>Item Details</title>
    <link href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet">
</head>
<body>
    <h1>Hello {{ name }}</h1>
</body>
</html>
```

To make it a bit stylish, let's add CSS.

Inside the static folder, save the css stylesheet.

static > # styles.css > ...

```
1   body {
2       background-color: red;
3       color: white;
4   }
5
```

Predefined Path Parameters

If we want to predefine the path parameters, we can use the Enum.

main2.py > ...

```python
from enum import Enum
from fastapi import FastAPI

class ModelName(str, Enum):
    tiger = "tiger"
    cheetah = "cheetah"
    lion = "lion"

app = FastAPI()

@app.get("/models/{model_name}")
def get_model(model_name: ModelName):
    if model_name is ModelName.tiger:
        return {"model_name": model_name, "message": "Tiger !"}
    if model_name.value == "cheetah":
        return {"model_name": model_name, "message": "Cheetah !"}

    return {"model_name": model_name, "message": "Leo !"}

```

If we go to the given address followed by models/tiger, we will see the appropriate output.

127.0.0.1:8000/models/tiger

{"model_name":"tiger","message":"Tiger !"}

127.0.0.1:8000/models/cheetah

{"model_name":"cheetah","message":"Cheetah !"}

Query Validations

There we have Query() to perform Query Validations.We are going to enforce that whenever q is provided, its length doesn't exceed 50 characters.

main2.py > ...

```python
import uvicorn
from fastapi import FastAPI, Query
app = FastAPI()

@app.get("/items/")
async def read_items(q: str = Query(default=None, max_length=50)):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)

```

We can add more validations like min_length and regex.

main2.py > ...

```python
import uvicorn
from fastapi import FastAPI, Query
app = FastAPI()

@app.get("/items/")
async def read_items(q: str = Query(default=None, min_length=3, max_length=50,
                                    regex="^fixed$")):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)

```

Default Query Values

Let's say that you want to declare the q query parameter to have a min_length of 3, and to have a default value of "fixedquery":

```python
main2.py > ...
import uvicorn
from fastapi import FastAPI, Query

app = FastAPI()
@app.get("/items/")
async def read_items(q: str = Query(default="fixedquery", min_length=3)):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)
```

Make Query Required

When we don't need to declare more validations or metadata, we can make the q query parameter required just by not declaring a default value, like:

```python
main2.py > ...
import uvicorn
from fastapi import FastAPI, Query
app = FastAPI()

@app.get("/items/")
async def read_items(q: str = Query(min_length=3)):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)
```

There's an alternative way to explicitly declare that a value is required. You can set the default parameter to the literal value ...

```python
main2.py > ...
import uvicorn
from fastapi import FastAPI, Query
app = FastAPI()

@app.get("/items/")
async def read_items(q: str = Query(default=..., min_length=3)):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)
```

If you feel uncomfortable using ..., you can also import and use Required from Pydantic:

```python
main2.py > ...
import uvicorn
from fastapi import FastAPI, Query
from pydantic import Required

app = FastAPI()

@app.get("/items/")
async def read_items(q: str = Query(default=Required, min_length=3)):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)
```

Query Parameter List

When you define a query parameter explicitly with Query you can also declare it to receive a list of values, or said in other way, to receive multiple values.

main2.py > read_items

```python
import uvicorn
from typing import Union
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
def read_items(q: Union[list[str], None] = Query(default=None)):
    query_items = {"q": q}
    return query_items

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)
```

Not secure | 0.0.0.0:3000/items/?q=1&q=2&q=22

{"q":["1","2","22"]}

Query Parameter Default List

We can also define a default list of values if none are provided:

```python
@app.get("/items/")
async def read_items(q: list[str] = Query(default=["apple", "orange"])):
    query_items = {"q": q}
    return query_items
```

Not secure | 0.0.0.0:3000/items/

{"q":["apple","orange"]}

Response Status Code

We can also declare the HTTP status code used for the response with the parameter status_code in any of the path operations.

```
@app.post("/items/", status_code=201)
async def create_item(name: str):
    return {"name": name}
```

We can specificallu state the status codes using status from fastapi.

```
main2.py > ...
import uvicorn
from fastapi import FastAPI, status

app = FastAPI()

@app.post("/items/", status_code=status.HTTP_201_CREATED)
async def create_item(name: str):
    return {"name": name}

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)
```

Uploading File

We can perform file uploading using the UploadFile() from fastapi.

main2.py > ...

```python
import uvicorn
from fastapi import FastAPI, UploadFile

app = FastAPI()
@app.post("/uploadfile/")
async def create_upload_file(file: UploadFile):
    return {"filename": file.filename}
if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)
```

POST /uploadfile/ Create Upload File

file * required

string($binary)

Choose file Screenshot f...5-40-09.png

Server response

Code	Details
200	Response body

```
{
  "filename": "Screenshot from 2023-02-22 15-40-09.png"
}
```

Exception Handling

We can use the HttpException from the fastapi in order to handle exceptions.

main2.py > ...

```python
import uvicorn
from fastapi import FastAPI, HTTPException
app = FastAPI()
items = {"wwe": "The Undertaker"}

@app.get("/items/{item_id}")
async def read_item(item_id: str):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item": items[item_id]}

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)

```

Server response

Code	Details
404 *Undocumented*	Error: Not Found **Response body** `{ "detail": "Item not found" }`

We can add custom Headers.

main2.py > read_item

```python
import uvicorn
from fastapi import FastAPI, HTTPException
app = FastAPI()
items = {"wwe": "The Undertaker"}

@app.get("/items/{item_id}")
async def read_item(item_id: str):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found",
        headers={"X-Error": "There goes my error"})
    return {"item": items[item_id]}

if __name__ == "__main__":
    uvicorn.run("main2:app", host="0.0.0.0", port=3000, reload=True)
```

Server response

Code	Details
404 *Undocumented*	Error: Not Found

Response body

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

Download

Response headers

```
content-length: 27
content-type: application/json
date: Wed,22 Feb 2023 10:21:59 GMT
server: uvicorn
x-error: There goes my error
```