Python API Development

By Sanjeev Thiyagarajan

Introduction

Topics covered

* This course is all about building out your very own API in Python.
* However, keep in mind this course is so much more than just building out a simple API.
* First of all, we are going to build out a fully featured API that include
  + Authentication
  + CRUD Operation
  + Schema Validation
  + Documentation
* However, learning doesn’t stop there this course is going to extend well past just basic API development.
* We’ll also learn all of the tooling that surrounds building a complete and robust API.
* A large section of this course is dedicated to learning SQL.
* Author: I have noticed that a lot of API and Wed Development courses just quickly gloss over the SQL without diving into the nitty-gritty of how SQL databases works.
* For this course we are going to cover SQL extensively right from the basic. No previous experience needed.
* But by the time you complete this course you will be very proficient at
  + Generating database schemas
  + Core SQL concepts like
  + Primary Key, Foreign Key, Table constraints
  + And you are pretty much able to generate SQL quires to grab the exact data you are looking for.
* On the top of that I’ll show you how to integrate your SQL databases your API using two different methods.
  + Integration using raw SQL quires.
  + Integration using ORMs
  + So no matter which ultimately you prefer you have all the resources to start building out your own projects.
* We will also familiarize our self with Database Migration Tool like Alembic**.**
* Which allow us to make incremental changes to our database schema to track changes in git.
* We will also learn tools like Postman to construct http packets so we can develop our API during the whole development process.
* Author: When it comes to testing, I’ve added about two to three hours of content going over how to setup Automated Integration Test.
* So that when you make changes to you code you can run these automated tests to verify that your code changes haven’t broke any pre-existing functionality.
* After testing we are going to move on the deployment phase.
* We are going to actually deploy our application including two different deployment scenarios.
* The first one is probably the most common scenario which is deploying your app onto an Ubuntu machine that can be hosted on any cloud provider like AWS, GCP, AZURE or even Digital Ocean.
* We cover things like Nginx to act as reverse proxy.
* We’ll configure our own system D service.
* We’ll setup a firewall to block all non-http traffic.
* And we’ll even setup SSL tha­­­t our application can handle HTTPS traffic.
* The second one is how you can deploy an application onto Heroku because maybe you can’t afford to pay for cloud services or maybe you don’t have the ability to sign-up for an account or something like that so I do want to make sure that you still have a way of deploying your applications so you can show off to your friends and family what you created.
* I added Heroku section because they got a very nice and convenient free tier where we can deploy our entire application for free, we don’t need to sign-up with a credit card.
* So all the cool kinds are hard-core into Docker today I am going to show you to Dockerize your API in case that it is you preferred method of deployment.
* Then Finally We going to wrap things up by building out our very own CI/CD pipeline using GitHub Actions.
* This will allow us to pushout changes to GitHub resulting in our pipeline running which will pull our code run all over integration tests build all of the necessary images and if all the tests ultimately pass it will actually push our changes to production environment.
* So, that we can do all of this in an automated fashion with having to manually go in and run each step manually.

Tech Stack

* Let’s look at our Tech Stack since this is a Python API course, we will be using Python to build out API.
* There were a couple of frameworks in python that we could have used most notably Django and Flask but I decided to use neither one of them.
* I decided to use a newer framework called FastAPI and the reason why you choose to use this framework because it has an API kind of built-in mind. It wasn’t there to address like Model View Controller type scenario.
* It really is all about building out API and on top of that it is really fast in terms of performance and makes it really easy to spin up new APIs.
* One my favourite feature of this framework is the auto documentation functionality.
* When you built an API, you have to document how your API works and this is a very cumbersome task because any time you make any change in your API you have to remember to update your API with in the frontend could be making the wrong request.
* FastAPI automatically documents your API for you so you don’t have to do it yourself it’s a truly game changer.
* I choose PostgreSQL because it is my favourite. It doesn’t really matter what type of SQL database you use they’re all fundamentally the same with only minor differences.
* For ORM we use SQLAlchemy because that’s the most popular one for Python.

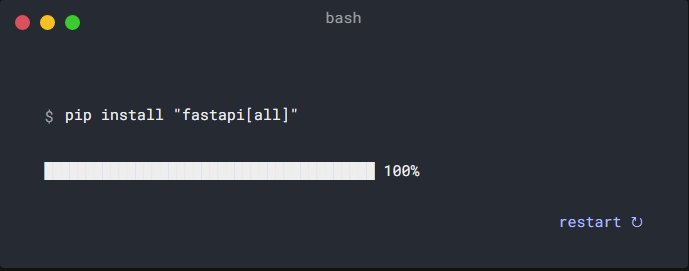
FASTAPI

Virtual Environment

* After Setting up Python and VS Code the next thing is to make virtual Environments.
* So let see what virtual environments are and what problems do they try to address.
* In short it helps by providing an isolated environment to work on projects that required a number of packages, libraries and framework as their dependencies and stay away from different version problems and other conflicts.

Install FastApi

* The first step is to install FastAPI.
* For the tutorial, you might want to install FastAPI with all the optional dependencies and features.
* That also includes **uvicorn** that you can use as the server that runs your code.

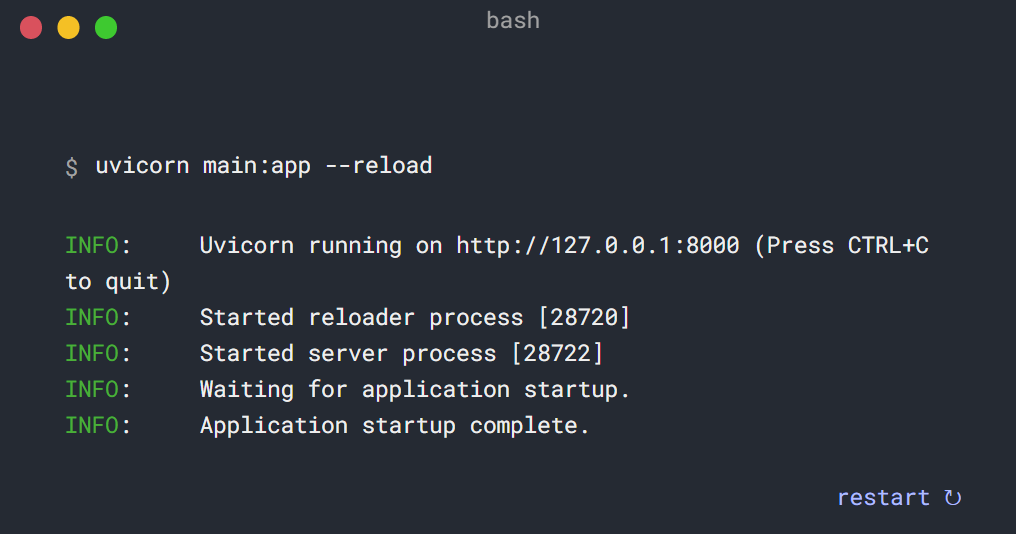


First Step

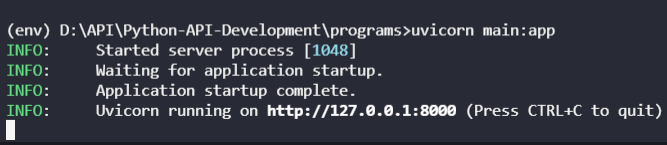
* Create our hello world API.



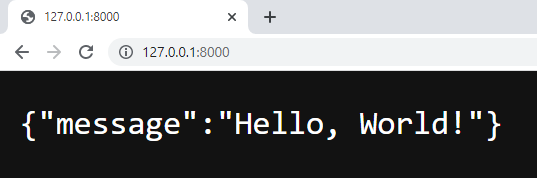
* So, let’s run our first API by starting **uvicorn**  web server.
* So how do we start our web server? Let’s go to the [documentation](https://fastapi.tiangolo.com/tutorial/). We are going to use **uvicorn** library.



* The command **uvicorn main:app** runs the web server refers to:
* **main:** the file **main.py** (the Python "module").
* **app:** FastApi instance the object created inside of main.py with the line **app = FastAPI().**
* **--reload:** make the server restart after code changes. This is an optional parameter only use for development mode. It will automatically restart the server when it found any changes in **main.py** file. We don’t need it in production mode.
* NOTE: We can choose **any\_name.py** for this file name so in this case we refer **any\_name** instead of **main** and this applies to **app** as well.
  + **uvicorn any\_file:any\_app**
* Now if you see server running in VScode/terminal so it is the sign of verification that everything is working fine.



* <http://127.0.0.1> is the location of server in this case this is local host :8000 is port number.
* Copy the **URL** and past in web browser and you will see.



First Step Code review

Path operation or Route



* According to documentation these three lines of code is what we called the path operation. In other languages and framework, we call it route. We will use these terms interchangeable.
* This path operation is made up of two components the first component is the function and the second component is the decorator.

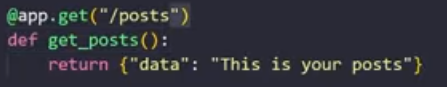
Function

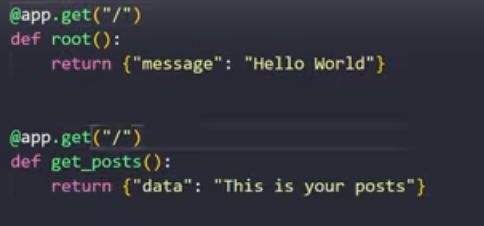
* The function **root()** in path operation is just a regular Python function that provide functionality for a particular task i.e., checking the password in a database and to make sure the credentials are proper and accurate.
* And after that just like any regular function, we can return something. So whatever we return here is going to be the data that sent back to the user. In our case the message hello world which is a dictionary object and FastApi is automatically converts it into a json object that’s why we can see it on the browser.
* Json is a main universal language of APIs.
* Keep in mind the names of function itself doesn’t really matter, it is an arbitrary name. If you change it to **get\_user()** or **login\_user()** then that’s going to be fine and bring no changes in the behaviour of code. I recommend you name your path operation functions as descriptive as possible.
* The keyword **async** is technically an optional keyword this only needed if you’re going to be performing some sort of asynchronous task that takes certain amount of time for example.
  + Making an API call.
  + Talking to the Databases.

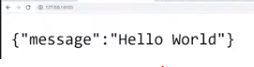
Decorator

* If you are not familiar with Decorators in Python it’s ok.
* Just understand that when you apply a decorator to a function it’s going to perform a little magic to this function.
* If we want to make our functions to act like an API, we have to use decorators.
* This magical decorator turns a function into an actual path operation so someone who want to use our API can hit this endpoint.
* In  **@** clarify a decorator then we reference our FastAPI instance in this case **app** is fastAPI instance and then we have a couple of different options.
* We are passing http method **get** that the user should use. But we can use a plenty of http methods [link](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods).
* So, **get** method means we have to send a get request to our API.
* So finally, we have route path **(“/”)** or **(Default).**
  + It is basically telling the root path of our API.
  + (i.e.: google.com and google.com/) get you on the same path.)
  + If we change it to **@app.get(“/login”)**
  + Our path must be **google.com/login**

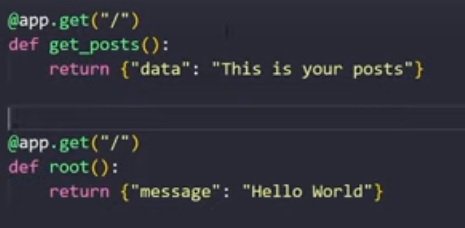
Order of Path operations

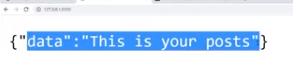


* So, now we have a solid understanding of how path operations work lets see if we can create a new path operation.
* Let say this path operation represents retrieving a bunch of social media post from our application.
* So, when it comes to retrieving data, we usually use a get operation. If you don’t know which one to use you can go to [this page](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods). It’s going to explain what each one used for.
* Any time you work in your browser your browser is always going to send a get method by default.
* Note: The way that fastapi works is that when anytime we send a request to our API sever its actually go down the list of all of the path operations and it’s going to find the first match and as soon it finds the first match it’s going to stop running you code.
* Blow is an example of how it works with different path operations with same URL so which one do you think is going to win?
* 
* Let’s take a look and as you can see the first one won.



* If we change the order, it change the result.





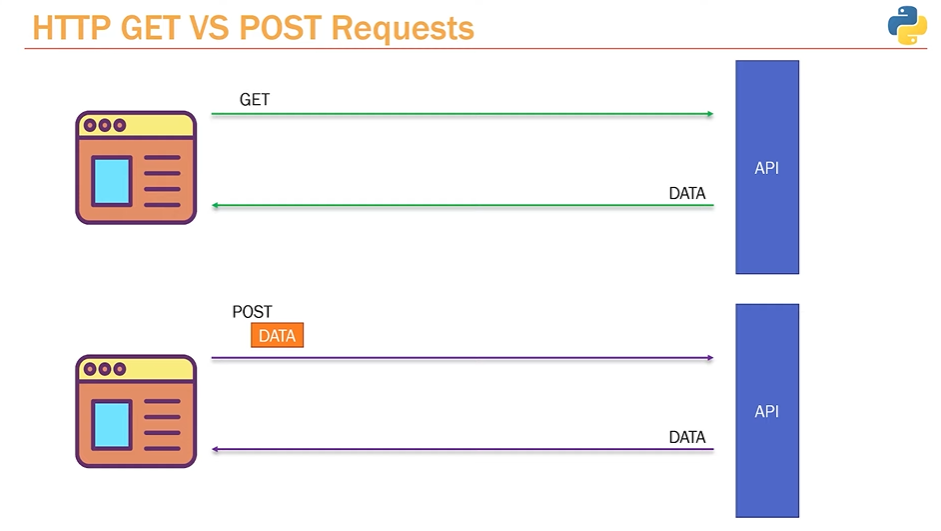
* So, the first path operation that matches is always going to be the one that runs simple is that. So, the order does matter and it can impact the way your API ultimately works.

Postman

* Till now we are using our browser to generate http request to test our API and that’s fine for now but once we get into more complex path operations and routes so the things that involves having to send other http methods like **put** or **patch** or etc., having to send data to our API is very complicated. Because there is no way to natively do that in browser without generating or building a complete full front-end application and to test your API you shouldn’t have to build an entire front-end application and that would be unmanageable and un-scalable.
* So, there are lots of different tools we can use to test our API and one of these tools is called postman.
* It’s a very simple tool which allow us to construct our own http requests.
* We can specify the individual filed of an http request like:
  + We can specify what is the http method
  + What’s the URL?
  + What are the headers that we are going to apply?
  + What’s the body?
  + What kind of data is going to carry?
  + Is it going to have any authorization headers?
* So, all these things we get to contract in a nice GUI so that we can test our API.
* You can download this app from [link](https://www.postman.com/downloads/).

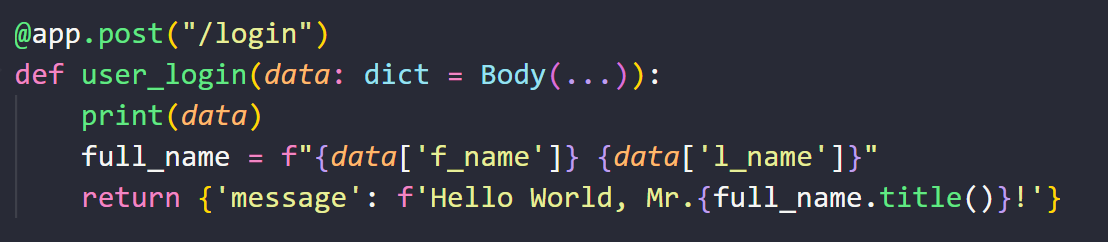
Http Get VS Post Requests

Till now we are working with **get** request and now we are going to learn **post** requests and what makes **post** request different from **get** request.



* At the left side we have a web browser and the right side we have our API server.
* So with the http get request what we do we send a get request to an API server and then API server send back some kind of data depending on the request.
* We use post request to **send data** to the server to **create** objects like a Social Media Post. Post request include all the data require to create a social media post on an API server and then API server will return a response depending on request.
* In real life if we are trying to create a post we will include all the data require to create a post like:
  + Title of post
  + Content of post
  + User of post
  + And server send back a response after doing some work with a message post successfully created.
* A get request is basically saying hey API server give me some data.
* A post request is saying hey API server here is some data do whatever you need to do with it.

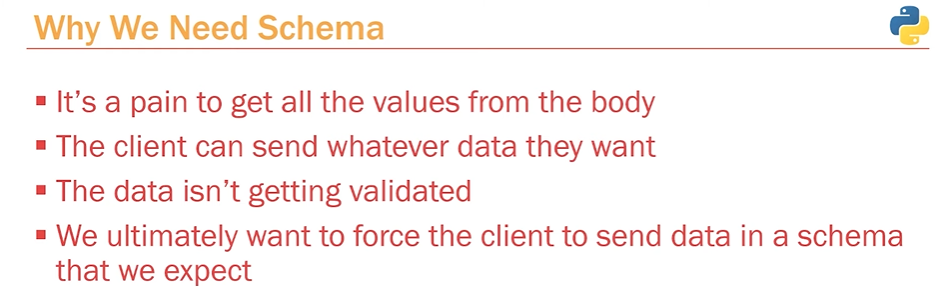
Post Requests





* Sending data to our API server using postman.
* Receiving data by providing parameters in our path function.
* Where **data** is the name of variable followed by a colon ‘:’.
* After colon we provide the **type** of data in our case **dict**.
* And then **Body(…)** is what catches the data coming out from client side.

Why we need Schema

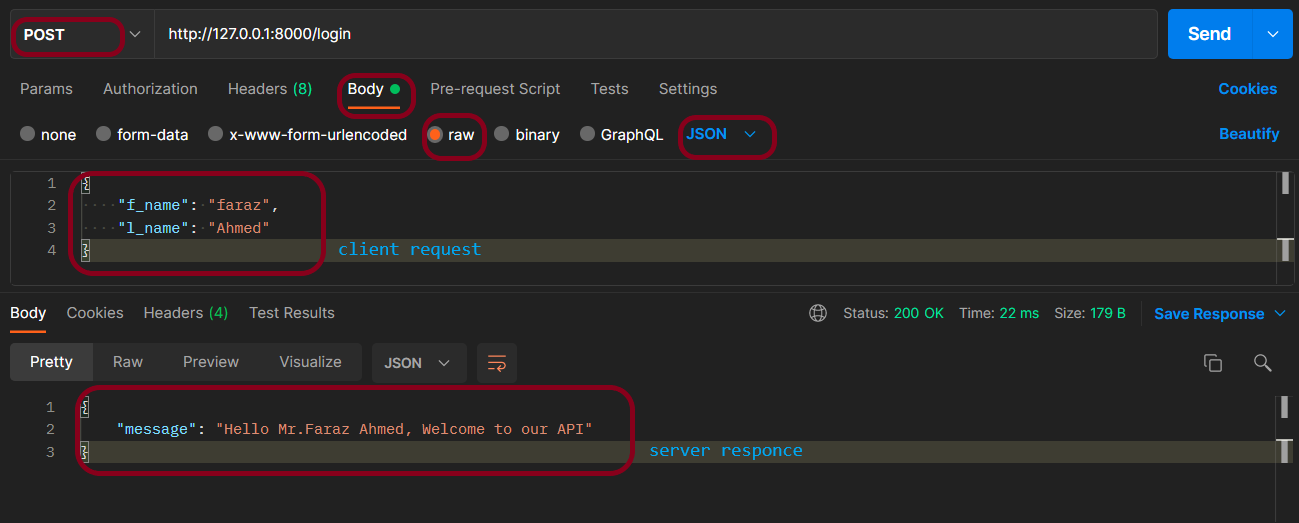


* Schema is like a contract between client and the server.
* A contract in which backend explicitly defines the rules and the format of data to be expected by restricting user/client to avoid errors.
* This is the way of working with APIs by defining how the data should look like so that the front-end can send you exactly what you expected.
* So, let’s see how we can do this in **FastAPI**.
* We are going to use a library called Pydantic**.**
* [Pydantic](https://pydantic-docs.helpmanual.io/usage/types/):*Data parsing and validation using Python type hints*.
* Let’s see how to work with Pydantic. It’s really simple.
* Keep in mind technically Pydantic has nothing to do with FastAPI.
* It’s a complete separate library that you can use with any of your python application to define a schema or defining what data do we expect from client?

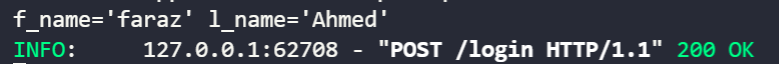
Pydantic BaseModel

 `

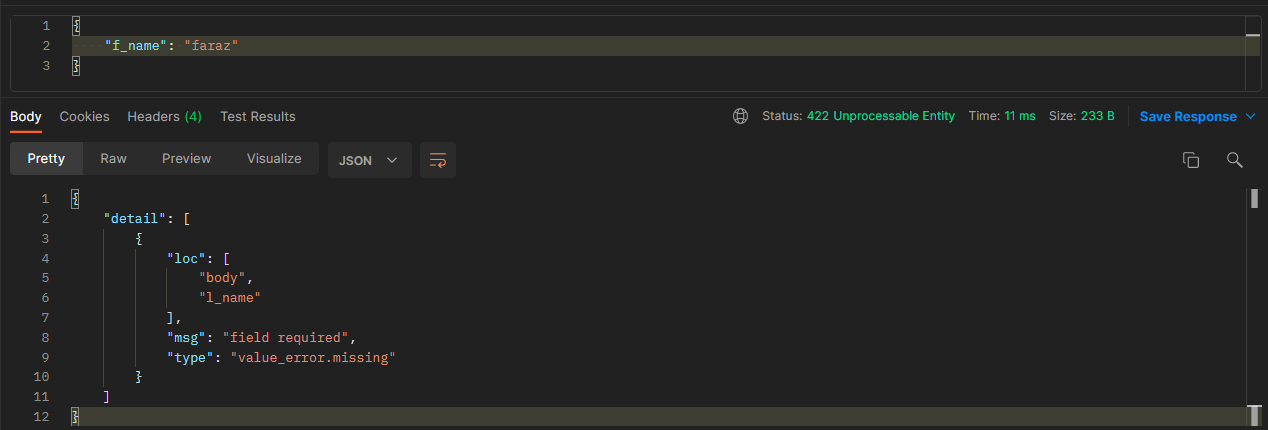
Defining schema in our BaseModel class



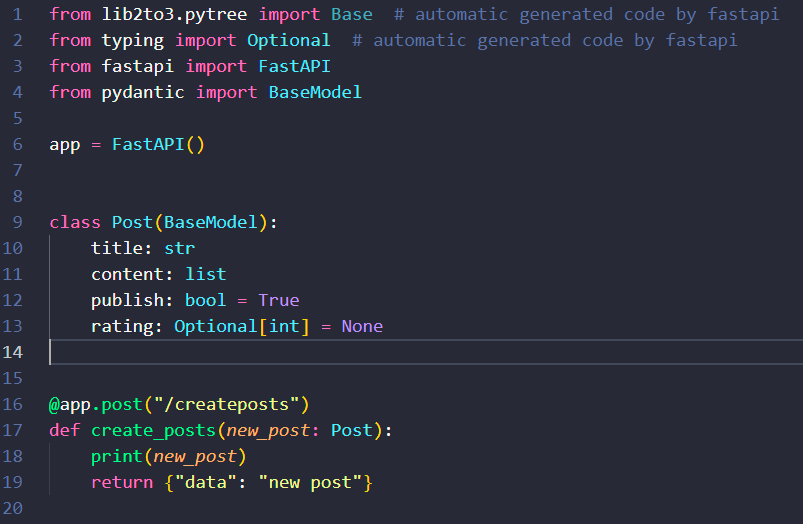
Consuming our API using postman



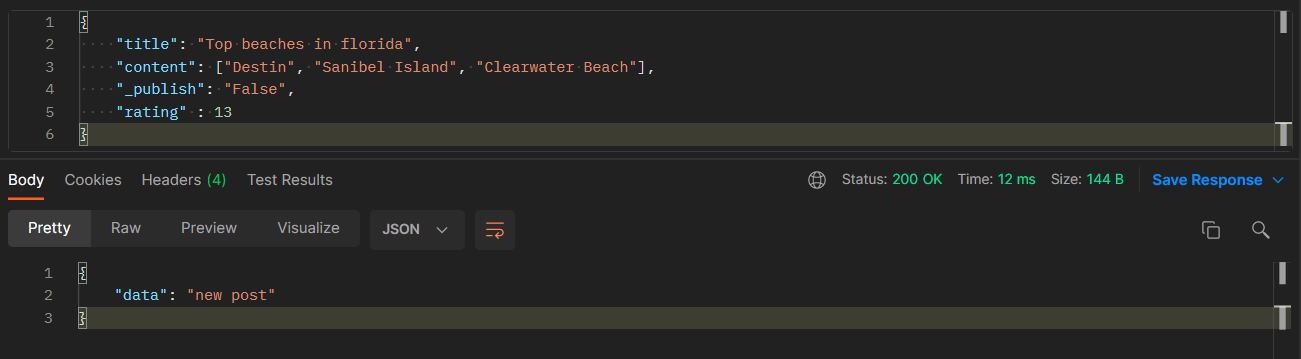
Server console

* **BaseModel** is a special Pydantic model we start by creating a class and inherit BaseModel following Pydantic [documentation](https://pydantic-docs.helpmanual.io/usage/types/).
* In our **User** class we are defining our schema so that client should follow otherwise it going to through an error.
* 
* BaseModel is doing all of the validation for us by defining the schema, like is the data is as per the rules we define in our schema.
* Now by passing BaseModel class it become very easy to extract data.
* Our code looks cleaner and more concise.

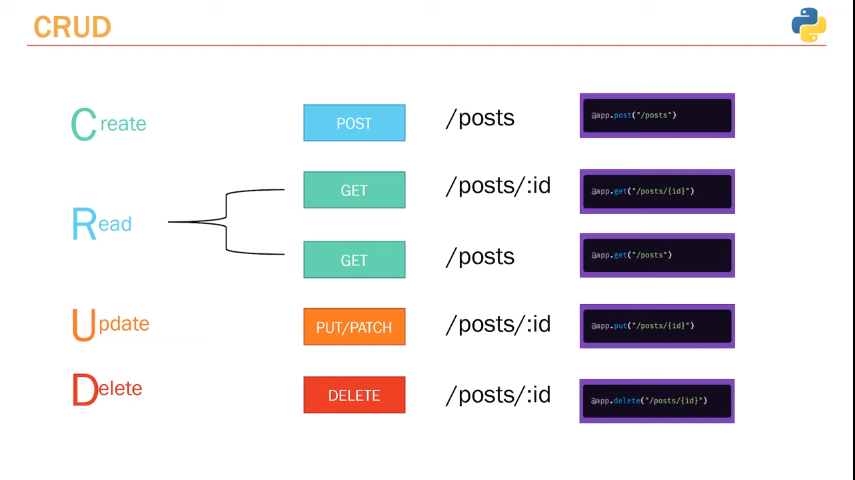
Another example Pydantic BaseModel

­

* This is a very simple example in which we start by defining our schema in **Post** class so we can get the data from client side as we define in **Post** class.
* In **Post** class we define
  + User must provide **title** and it must be string.
  + User must provide **content** and it must be a list.
  + And there are two optional values **publish** and **rating**
  + In the picture blow the key “\_publish” has a value False which we have to provide in double quotes even though it is a Boolean value we have to enclosed it in quotation marks but because we provided ‘\_’ before key it is a comment in json and that’s why the default values is set to true.
  + **Rating** is another optional field. if user doesn’t provide any integer value we set it to **none** instead of assigning any default value (eg: True/False).



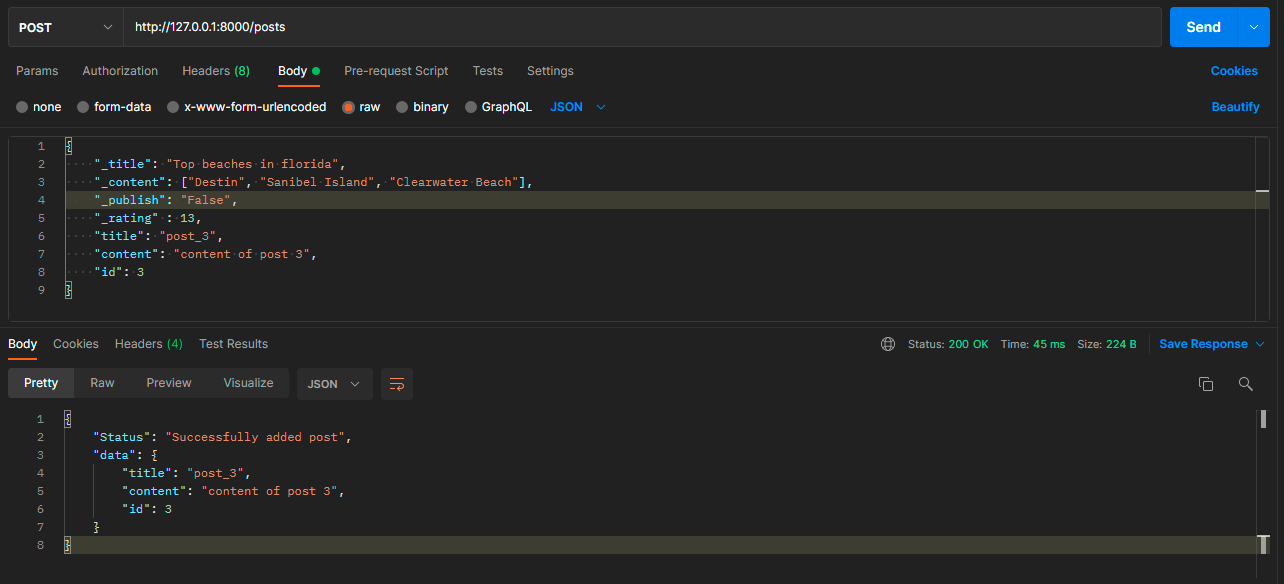
FASTAPI Crud operations



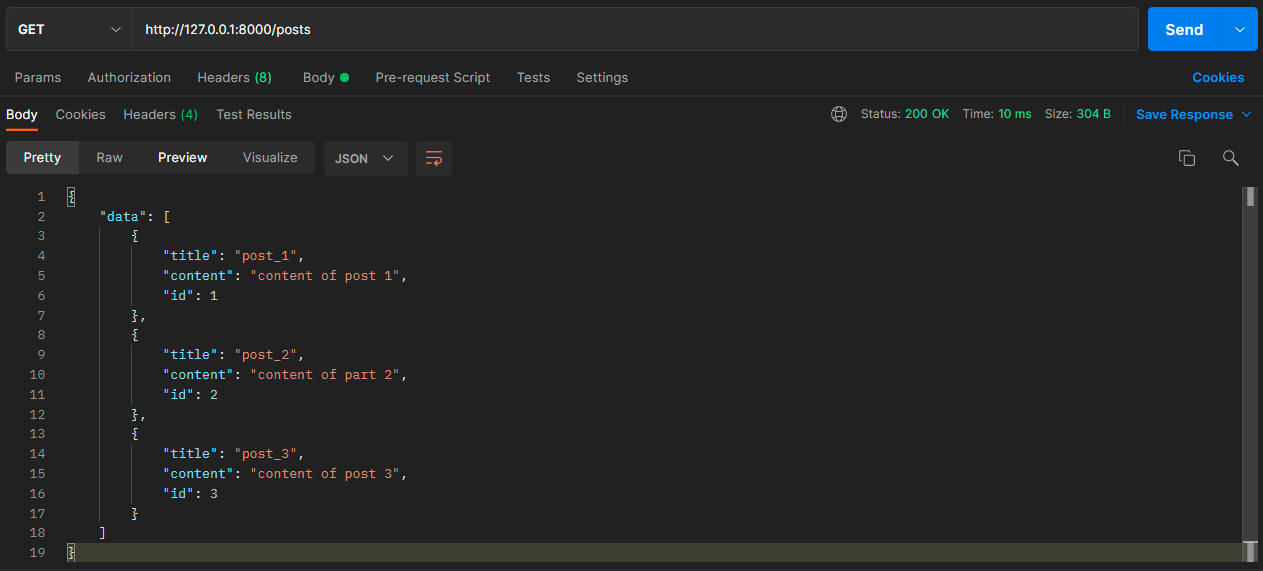
* Now we are going to talk about what a CRUD application is as well as what are standard conventions when it comes to creating an API for a CRUD based application.
* So, CRUD is an acronym that represent four main functions of an application.
  + Create, Read, Update and Delete.
* So, when it comes to naming the **URL** and the **paths** for each operation there is a standard convention and the first thing.
* I want to point out since we are working with posts it make sense to name all of the path with slash posts**(“/”posts)** and it is important that you use the **plural form** this is the standard convention for APIs. So, for example if we were working with users and we want to apply CURD operations on users it **wouldn’t** be slash user it would be slash users**(“/users”)**.
* Now when it comes to **creating** a post that’s always going to be a **post** request.
* For **reading / retrieving** pre-existing post there are two different path operations for that.
  + When it comes to retrieving information from a database or anything like this it’s always going to be a **get** request.
  + First one is **@app.get(“/posts”)** and this one is for retrieving either all of the post or multiple post depending on what filter that we use.
  + Secondly if you want to get information about one specific post we pass id of post which is a id field in database **@app.get(“/posts{id}”).**
* **Update** involve updating a pre-existing post and there are two http methods for that **Put / Patch.**
  + When applying an update operation on a post using **Put** method we have to send all of the fields that we want to change as well as fields that we don’t want to change that means all the same information about a post to API server to update a post.
  + On the other side Updating a post using **Patch** method we have to send only the specific field/fields that we want to change (e.g.: **title** of post).
  + Author: We will stick to put method but at the end of the day it doesn’t really matter its just a matter of user preference.
  + Just like to get a specific request we have to pass id of post in FastAPI decorator e.g.: **app.put(“/posts/{id}”) or app.patch(“/posts/{id}”)**
* To delete a post once again we have to send a delete request and for that the http verb is **delete** and we have to pass the specific post id as well.
* Eg: **app.delete(“/posts/{id}”)**
* It actually not that hard its fairly simple once you create one CRUD API creating another one is almost the matter of copying and pasting really.

Creating a Post and saving in List



Post Request 

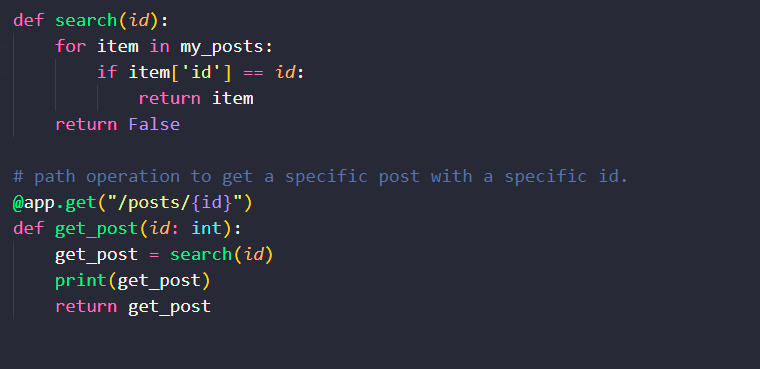
Get Request



Server Response

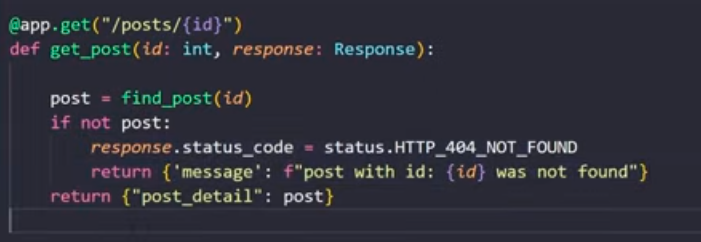


Getting a single post with path parameter



* We just include **/{id}** in path operation so we can get a specific post with help of unique id attached with a specific post and we call it as path parameter.
* Keep in mind the path operation with {id} can create some conflict.
  + Like if you create another path operation like **@app.get(“/posts/latest”)** right after the path operation with {id} included in it, so when client will hit this end point FastAPI will match latest path operation with id path operation and and will cause an error so in order to get rid of it paste this path operation above the one that include {id}.

Error handling



* Return customize status code as 404 and message following the error message hard coded stuff.
* I will need **from fastapi import Response, status**.
* The other way of doing this is by doing **from fastapi import HTTPException**

