Creating A Beautiful Web API In Python

Create a complete web API with Python, Flask, and MongoDB using sustainable coding practices.

[](https://towardsdatascience.com/@jrbarhydt?source=post_page-----6415a40789af----------------------)

[Johnathon Barhydt](https://towardsdatascience.com/@jrbarhydt?source=post_page-----6415a40789af----------------------)

[Follow](https://medium.com/m/signin?operation=register&redirect=https%3A%2F%2Ftowardsdatascience.com%2Fcreating-a-beautiful-web-api-in-python-6415a40789af&source=-3f4f45200621-------------------------follow_byline-)

[Apr 27](https://towardsdatascience.com/creating-a-beautiful-web-api-in-python-6415a40789af?source=post_page-----6415a40789af----------------------) · 17 min read



Immagine che contiene screenshot, erba, tavolo, monitor

Descrizione generata automaticamente



Immagine che contiene disegnando

Descrizione generata automaticamente



Immagine che contiene disegnando, animale

Descrizione generata automaticamente

Flask and MongoDB

Creating a website or API with a database backend is a great programming project, but it can be tough to get started since there are so many concepts to work with. This guide is meant to help you build your first (or 5th) API using Python Flask and MongoDB, from start to finish.

Before we start: This guide is aimed towards beginners / intermediate level. I expect a bit of python knowledge and object-oriented programming fundamentals. As for myself, my background is in physics, math, and machine-learning — but web stuff had always seemed daunting. It’s easy to find tutorials for each piece individually, but this tutorial is more about putting all the pieces together to make sustainable code. Hopefully, this guide will help take the edge off.

*Once you have understood the steps and how each piece fits together, I*highly *recommend learning about each component in more detail. Another piece of advice: try to re-build the whole thing from scratch on your own afterwards.*

Here is a breakdown of all the technologies and tools we’ll look at (don’t worry about the size of this list, most of these are not too threatening!):

**What Is A Web API?**

* Intro to Web APIs with examples

**MongoDB:**

* Intro to MongoDB
* Installation and starting a server
* Adding some data and running a test query

**Flask:**

* Intro to Flask
* Flask-MongoEngine {*for handling MongoDB in an object-oriented way*}
* Flask-Bcrypt {*for encrypting passwords*}
* Flask-RESTful {*for building REST APIs*}
* Flask-JWT-Extended {*for authentication and authorization*}

**Postman:**

* Intro to Postman
* Sampling data and using GET, PUT, POST, DELETE requests

**Hosting Your API:**

* Running and testing on your local computer
* Hosting the API publicly on Heroku using gunicorn
* (Optional guide for hosting on Google Cloud Platform could be added in the future)

Ok, let’s get started!

Grab the GitHub template code:

git clone -b template <https://github.com/jrbarhydt/FlaskWebAPI.git>

Or build it yourself:

│  
│ app.py  
│ Procfile  
│ requirements.txt  
│ runtime.txt  
│  
├───api  
│ authentication.py  
│ meal.py  
│ routes.py  
│ user.py  
│ \_\_init\_\_.py  
│  
├───models  
│ meals.py  
│ users.py  
│ \_\_init\_\_.py  
│  
├───resources  
│ meal\_data.csv  
│ user\_data.csv  
│  
└───tools  
 load\_data.py  
 \_\_init\_\_.py

**What Is A Web API?**

A Web API (Application Programming Interface) allows you to serve data over the web, typically in JSON or XML format. Generally, this is done by exposing endpoints to make requests.

Immagine che contiene sedendo, computer, monitor, nero

Descrizione generata automaticamente

Immagine che contiene testo, sedendo

Descrizione generata automaticamente

An example is the PokéAPI ([*https://pokeapi.co/*](https://pokeapi.co/)), which gives you access to a Pokemon database. Try the link below to get all the data you could ever want to know about a Squirtle!

[*https://pokeapi.co/api/v2/pokemon/squirtle*](https://pokeapi.co/api/v2/pokemon/squirtle)

You may notice that there is a giant wall of text returned! This is why I like to use the [**JSON Viewer**](https://chrome.google.com/webstore/detail/json-viewer/gbmdgpbipfallnflgajpaliibnhdgobh?hl=en-US) extension for Chrome to see the data in a more readable format like shown in the image.

Let’s break down the Squirtle link to learn a bit about endpoints (or ‘routes’ as we will call them in Flask)





Entry Point (LEFT BOX) and Endpoint (RIGHT BOX)

By convention, an API will have one entry point. This is like the root folder in your File Explorer. There can be any number of endpoints (which I will start calling ‘routes’ from now on.)

Data can be delivered in different forms depending on the given route. Try navigating to the /pokemon route and you will get the data for ALL Pokemon, instead of just for Squirtle.

For our API, we will create a restaurant database where a user can signup, login, save their details, and keep a list of their favorite meals. Some users will be allowed to do administrative tasks like deleting users or accessing any user’s meal list.

**MongoDB { () }**

***Intro to MongoDB:***

Choosing the right type of database for your project is incredibly important. A traditional SQL database stores information in tables. This is in contrast to a noSQL database.

MongoDB is a noSQL database which stores data in JSON format. As opposed to tables, JSON forms a tree data structure. The individual records are known as ‘documents.’

(I would recommend staring at the Pokemon JSON data above until you are convinced that it is a tree instead of a table.)

***Installation and starting a server:***

You can get the free version of MongoDB at [*https://www.mongodb.com/download-center/community*](https://www.mongodb.com/download-center/community)



Immagine che contiene screenshot

Descrizione generata automaticamente

Default Installation Options

The defaults are fine for now. Compass is optional, we won’t be using it in this tutorial. Wherever you install MongoDB and your database, make sure to remember the location.

**Launch Server:** *C:\Program Files\MongoDB\Server\4.2\bin\mongod.exe*

Leave your server running in the background, you can now access your database from the mongo shell or as we will see later, also from python. This is an unsecured server, and later we will add authorization.

**Launch Mongo Shell:***C:\Program Files\MongoDB\Server\4.2\bin\mongo.exe*

*I would recommend looking at the MongoDB documentation as you play around*[*https://docs.mongodb.com/manual/reference/mongo-shell/*](https://docs.mongodb.com/manual/reference/mongo-shell/)

Let’s insert a few records (documents) into our database. (You can copy and then use right-click to paste these into the mongo command shell.)

use test\_dbdb.meals.insert({"name": "Mystery Pie", "description": "Not even the baker knows what's inside..."})db.meals.insert({"name": "Apple Pie", "description": "A delicious, home-made apple pie. Best served on a window sill in the summer."})

The ‘use’ will select a db (or create one), and you may notice that the records we inserted are in JSON format. Also, mongo will automatically create the collection if it doesn’t exist. Above, our collection is called ’meals’.

Verify that everything worked by pasting each of the following into mongo.

show databasesshow collectionsdb.meals.find()db.meals.find({"name": "Apple Pie"})

Using db.<collection>.find() will return all of the data, and we can filter by passing JSON as a parameter into the find() method.





Every item you add will have a unique ObjectId. This will become handy later for referencing items from other collections, such as users.

*Play around with making databases, collections, and documents. You now have the****very*** ***basics****of running a database.*

For security, set up admin credentials to your database. (Maybe don’t make your password as “password”)

use admin  
db.createUser(  
 {  
 user: "admin",  
 pwd: "password",  
 roles: [ { role: "userAdminAnyDatabase", db: "admin" }, "readWriteAnyDatabase" ]  
 }  
)  
db.adminCommand( { shutdown: 1 } )

The last command will shut down our server. To launch the server with authentication enabled, run this from the terminal instead.

mongod --auth

With this server running, you won’t be able to do anything without authenticating first. You can test this by relaunching mongo.exe and trying to access some data. Enter the following into the mongo shell to sign in and gain access.

use admin  
db.auth("admin", "password")

We now have our secured database server, so it’s time to start building the API.

**Flask \\\_**

Flask is a light-weight web framework. With Flask, you get to pick-and-choose what components and extension needed for your site. I like this aspect since the modular nature allows you to build everything up, without too much clutter.

The main Flask extensions we will use are Flask MongoEngine and Flask RESTful. The first will let us build Classes as templates for our data and the latter is designed for building an API, and makes the process simpler.

Additionally, we will use JWT-Extended and BCrypt, but I will cover those later.

Now, it’s coding time…

Create a clean virtual environment and get Flask.

*If you aren’t familiar with virtual environments, it won’t hurt my feelings but you should definitely check out*[*https://docs.python.org/3/tutorial/venv.html*](https://docs.python.org/3/tutorial/venv.html)

pip install Flask

Now, before digging in any further, I want to make some remarks about my coding methodology here. Here are some things I included:

* I have documented each part. This could be very helpful if I have to come back to this code in many months to add new features, fix something, or even to revisit some problem I had to solve.
* I have a default configuration, which will be useful for testing. In a production environment, you can easily pass the non-test info into the function. (If you do not call the function with a config, the default will be used)
* I used Python’s type hints to keep track of parameter and return value types.
* I have the app created and returned in a function. This will also be good for testing and scalability. For instance, you can run get\_flask\_app twice with different configurations.
* I added the \_\_main\_\_ entry point, so if I execute this script as-is, it will launch in debug mode. I will cover launching with gunicorn later.

Paste the following code.

* app.py

You can run app.py now, and you will be hosting a Flask server. It won’t do anything yet, though.



Immagine che contiene screenshot, monitor, schermo, computer

Descrizione generata automaticamente

Running Flask Server on Debug Mode: Using PyCharm Community Edition

**Flask-MongoEngine:**

Get Flask-MongoEngine, as well as BCrypt for password encryption.

pip install flask-mongoengine  
pip install flask-bcrypt

*Check out the documentation for more details and implementation examples.*[*http://docs.mongoengine.org/projects/flask-mongoengine/en/latest/*](http://docs.mongoengine.org/projects/flask-mongoengine/en/latest/)

*Flask-MongoEngine also comes with MongoEngine and PyMongo. You may find these libraries interesting on their own. Check out their docs as well!*[*http://docs.mongoengine.org/*](http://docs.mongoengine.org/)[*https://pymongo.readthedocs.io/en/stable/*](https://pymongo.readthedocs.io/en/stable/) *We won’t be using PyMongo, but if you get very comfortable using the mongo shell, you may like this package. It uses python commands that are almost identical to what you would enter into a mongo shell!*

We will create a class model for users and another for meals. These will live in the models folder. Compare the Meal Class below to the entry we put into mongo through the shell. As you can see, MongoEngine is a lot more object-oriented.

* models/meals.py (without documentation)

from mongoengine import Document, StringField, FloatField  
  
  
class Meals(Document):  
 name = StringField(required=True)  
 description = StringField()  
 price = FloatField()  
 image\_url = StringField()

Before doing users, I will add some documentation. I wanted to show the difference between the raw code, and well-documented version. In the example section, there is code included which can be run directly, or by using doctest. I like adding doctest style comments since they both act as a test and to give a usage example.

:: Execute a doctest in the terminal  
python -m doctest api/meals.py

*All code within the documentation that starts with****>>>****will be run by doctest.*

* models/meals.py (with documentation)

The Users Class is a bit more complicated. There is a Field for determining access level, one for validating phone numbers, and the Users Class overrides the Document.save() method, so we can generate a password hash before saving to our database. This way, if our database was ever compromised, the attacker would not get actual passwords.

*Here are the docs about Flask-BCrypt.*[*https://flask-bcrypt.readthedocs.io/en/latest/*](https://flask-bcrypt.readthedocs.io/en/latest/) *Learn more about slow-hashing and SALT, here!*[*https://crackstation.net/hashing-security.htm*](https://crackstation.net/hashing-security.htm)

* models/users.py

Congratulations, now the MongoEngine models are finished! Next, we will build the interface for the API.

**Flask-RESTful**

pip install flask-restful

The Flask-RESTful library will require a setup much like MongoEngine, where we will make a Class for each API interaction. These Classes are called Resources. In Flask, connecting a Resource to an endpoint is called a **route**. This is what we are referring to when we say routes.

Our API will have a few routes. One for authentication (signing up and logging in), another for users (to GET or PUT or DELETE user data), and another for meals (to GET, PUT, or POST).

We will eventually require the use of web tokens to access all of the data, but first we will add the basics necessary to make your first API call.

*Learn more about JSON Web Tokens here*[*https://jwt.io/*](https://jwt.io/)

Now, to get our first API call:

* api/meal.py

# flask packages  
from flask import jsonify  
from flask\_restful import Resource  
  
# mongo-engine models  
from models.meals import Meals  
  
  
class MealsApi(Resource):  
 def get(self):  
 output = Meals.objects()  
 return jsonify({'result': output})

When we make a GET request to the correct route, the get() method of this class will be called. This method returns all of the objects in Meals (which we currently have 2) and turns them into a JSON response.

So, now we need to specify the correct route to access this. Add the following to routes.

* api/routes.py

from api.meal import MealsApi  
  
  
def create\_routes(api):  
 api.add\_resource(MealsApi, '/meal/')

Here, we added the MealsApi Resource that was just created, and gave it the ‘/meal/’ endpoint. So now we have a complete route, and just like with the PokeAPI we can start serving data! After setting up app.py some more.

Update the import section to get the necessary modules in app.py

# flask packages  
from flask import Flask, app  
from flask\_restful import Api  
from flask\_mongoengine import MongoEngine# local packages  
from api.routes import create\_routes

Add the following lines to the end of get\_flask\_app() in app.py in order to setup the RESTful API andMongoEngine.

# init api and routes  
api = Api(app=flask\_app)  
create\_routes(api=api)# init mongoengine  
db = MongoEngine(app=flask\_app)return flask\_app

Try it out. Run app.py and you will see the default URL displayed. You can navigate to <http://127.0.0.1:5000/meal/> and get a printout of all the meals in the MongoDB database. (You will need to still have mongod.exe running for this to work)



Immagine che contiene computer

Descrizione generata automaticamente

Great. Now we will continue with the rest of the Resources and routes by doing the same steps. I will add documentation as well.

Authentication allows users to log in before accessing the data. We have two routes included, one for singing up and another for logging in. The SignUpApi Class is the same as a POST request to the User model, except it doesn’t require prior authentication (since the user hasn’t signed up yet).

A couple of notes about the coding of these classes:

* the post() method is static since the class has no special context, such as requiring JSON Web Tokens.
* The get, post, put, and delete return type is Response, this is a class that Flask inherits from Werkzeug, but it isn’t necessary to understand the underlying mechanics. It just handles HTTP communication stuff. Flask takes care of this when you pass a dictionary object into flask.jsonify. Feel free to see the documentation for more info.
* If the asterisks in Users(\*\*data) is new to you, don’t worry it is just a concise way of passing arguments into a Class or method using a dictionary.

*Take a look at*[*Dictionary Unpacking*](https://python-reference.readthedocs.io/en/latest/docs/operators/dict_unpack.html)*to learn more.*

* The LoginApi checks the given password to see if it matches and then creates web tokens if successful. These tokens allow a user to continue using the API without the need to login for each request. A refresh token is generated, but not used in this tutorial. This would allow a user to continue their session after the token expires.

*Learn more about keeping your data secure with tokens here*[*https://auth0.com/docs/tokens/concepts/access-tokens*](https://auth0.com/docs/tokens/concepts/access-tokens)

Here is the completed authentication code.

* api/authentication.py

The error messages should be fairly self-explanatory.

* api/errors.py

There are two things to mention about the two meal api Resources. Firstly, you’ll notice that some of the methods have arguments now. This is handled by routes.py which will snag arguments from your endpoint. More on that in the routes section below. Secondly, you’ll notice the @jwt\_required decorator before some of the methods. This single line allows you to enforce the use of web tokens to gain access to the data.

*If you haven’t used decorators in the past, I’d recommend learning a bit more. They can be super handy for quickly adding functionality to a method.*[*https://realpython.com/primer-on-python-decorators/*](https://realpython.com/primer-on-python-decorators/)

And below is the code.

* api/meal.py

One of the additions to routes.py is super important. You will notice an argument is passed inside the endpoint itself. Arguments are denoted by placing the name inside angled brackets. Flask RESTful will recognize these brackets and pass the value into the get(), post(), put(), or delete() methods.

* api/routes.py

The code in user.py is very similar to what was used in meal.py above.

* api/user.py

Finally, don't forget to update app.py. Aside from adding the new imports and initializing JWT, we also have to add a jwt secret key to the default config. Obviously, you would not want your production code to use this key, but just like the default username and password, these are only designated for testing. In a production setting, you would save the configuration file elsewhere, and pass it into the get\_flask\_app() method. The final version of app.py is below.

* app.py

Don’t forget to add flask-jwt-extended

pip install flask-jwt-extended

That’s it. Now the API is complete. Well, sorta. We don’t have a good way to test every aspect of it (like creating a new user, etc.) Although this can be done directly in python using the requests library, I’ll use Postman instead. I find it to be a great way to test APIs.

**Postman ( / )**

Rather than explain all of the fun you can have with Postman, I’ll step you through signing up, logging in, and adding data. This will get you started, but there’s plenty of great features inside.

*Download Postman at*[*postman.com/downloads*](http://postman.com/downloads)



Immagine che contiene nero, screenshot, monitor, sedendo

Descrizione generata automaticamente

You can see the result of a get request above. It is the same as what we got earlier by entering ‘http://127.0.0.1:5000/meal’ into a web browser while both mongod.exe and app.py were running.

* If you try this again, it will fail! Can you guess why? Scroll down for the answer, but bonus points if you can figure it out yourself.



Immagine che contiene monitor, nero, screenshot, schermo

Descrizione generata automaticamente

Here’s a hint…

We don’t have a web token!

To get one, we need to log in. In order to login we need to sign up.

Ok, so let’s sign up. Set a **POST**request and set the url to <http://127.0.0.1:5000/authentication/signup/> and add a **Body**, with type **raw JSON**. Enter the following into the Body and click Send.

{  
 "email": "[test@not-a-fake-email.com](mailto:test@not-a-fake-email.com)",  
 "password": "hunter2",  
 "access": {  
 "admin": "True"  
 },  
 "name": "testy buddy",  
 "phone": "555-555-5432"  
}

The response includes the unique user\_id. This can also be helpful for the route which requires it.



Immagine che contiene nero, screenshot, monitor, sedendo

Descrizione generata automaticamente

Now, we will log in to get our web token. Fun fact, your unique user\_id will also be encoded inside the web token, so it isn’t absolutely necessary to remember it. You can see the method get\_jwt\_identity() for this in meal.py in the MealApi class.

*If you are feeling adventurous, you could even change the ‘/user/<user\_id>’ endpoint in our routes. Since we can use get\_jwt\_identity() to grab this info from the web token, we could have a single endpoint for getting a single user. Maybe try ‘user/get\_user/’ and ditch the user\_id parameter in MealApi.*

To log in, we will POST an email and password over the [login route](http://127.0.0.1:5000/authentication/login/).

{  
 "email": "[test@not-a-fake-email.com](mailto:test@not-a-fake-email.com)",  
 "password": "hunter2"  
}

You can see that the API tokens are successfully returned.

Immagine che contiene interni, monitor, sedendo, computer

Descrizione generata automaticamente

Immagine che contiene screenshot, nero, monitor, sedendo

Descrizione generata automaticamente

We can copy the access token and paste it into an Authorization header.

Try again to access a **GET** request from the **[meal endpoint](http://127.0.0.1:5000/meal/" \t "_blank)**, but this time put the **access\_token** into the **Token** Field of the **Authorization** tab, by selecting **Bearer Token** as the **TYPE**. Look at you, following along to a bunch of web jargon!

Immagine che contiene monitor, interni, sedendo, schermo

Descrizione generata automaticamente

Immagine che contiene screenshot, monitor, nero, sedendo

Descrizione generata automaticamente

Accessing MongoDB data over internal Web API

Now you can add data of your own to fill up the database. Or you can use the resources/meal\_data.tsv and resources/user\_data.tsv which contain a bunch of entries. I’d recommend writing your own scripts for doing this, but I will include tools/load\_data.load\_all() if you just want to input some test data.

Here is the data:

* resources/meal\_data.csv

[Right-Click This Link and Save](https://gist.githubusercontent.com/jrbarhydt/c9bf0b2a2a1b1905fe0c9b6a669fecc4/raw/4658802dc2f1fea91d625646fc737a6d9b7b5e70/meal_data.csv)

* resources/user\_data.csv

[Right-Click This Link and Save](https://gist.githubusercontent.com/jrbarhydt/c9bf0b2a2a1b1905fe0c9b6a669fecc4/raw/51d7a3d814ddf0fbbc6942d2033f3503788e2e39/user_data.csv)

And here are the tools I made for loading the data into MongoDB:

* tools/get\_random\_document.py
* tools/load\_data.py
* tools/mongo\_loader.py

**Hosting Your API**

So far everything has run locally. For anyone to access your API over the internet, you will have to host a web server and database server somewhere publicly accessible. There are many ways to do this, but I will cover deploying with Heroku.

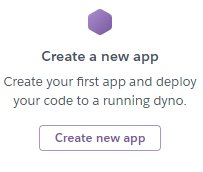
**Heroku [K]**

An easy way to do host and API is with Heroku. Heroku is pretty simple, as it allows you to have a git repository specifically for your web server.

First, set up a free account on <https://www.heroku.com/> and once you are signed in, select ‘Create a new app’ from the main menu. You will create a unique app name that will be hosted at *<your-app-name>.herokuapp.com*

Immagine che contiene screenshot

Descrizione generata automaticamente



First, to get Heroku to know how to run your app, it will look for three files in your main project directory.

* Procfile: This is a shell script command that Heroku will run first. In our case, it launches gunicorn in the same way we launched it locally before.

web: gunicorn 'app:get\_flask\_app()'

* requirements.txt: This contains all of the python packages we used. You can get this list by running *pip freeze.* I will paste my list below, yours may differ slightly.

aniso8601==8.0.0  
bcrypt==3.1.7  
cffi==1.14.0  
click==7.1.1  
Flask==1.1.2  
Flask-Bcrypt==0.7.1  
Flask-JWT-Extended==3.24.1  
flask-mongoengine==0.9.5  
Flask-RESTful==0.3.8  
Flask-WTF==0.14.3  
gunicorn==20.0.4  
itsdangerous==1.1.0  
Jinja2==2.11.2  
MarkupSafe==1.1.1  
mongoengine==0.19.1  
pycparser==2.20  
PyJWT==1.7.1  
pymongo==3.10.1  
pytz==2019.3  
six==1.14.0  
Werkzeug==1.0.1  
WTForms==2.2.1

* runtime.txt: This tells Heroku what version of python we are using

python-3.7.6

Finally, here is a list of tabs on the Heroku web page that you’ll configure to get your API up and running.

* **Resources**: In Add-ons, search for *mLab MongoDB* and add it to your app. You can choose the free version. You will have to enter your payment details for this, but as long as you stick with the free version you will be safe from charges.
* **Deploy**: Connect to your GitHub account, and choose the GitHub project that includes all your API project files. You can have Heroku auto-deploy from the master branch.

*If you don’t want to use a GitHub account, you can also choose to deploy with the Heroku CLI. If you choose this method, there are instructions on the Heroku site for installing the app and setting this up.*

* **Settings**: Under Config Vars, click Reveal Config Vars to get a list of important variables. You will want two variables in here: MONGODB\_URI and JWT\_SECRET\_KEY. The secret key is private and up to you to determine, as it will allow encryption/decryption of the web tokens.

Ok, that should be everything. You can look in the Activity tab to see if the build was successful, or to read the build logs, etc.

Now, using Postman to try the same as before, except this time using your new, public API URL and endpoint.



Immagine che contiene nero, monitor, screenshot, metallo

Descrizione generata automaticamente

Signing up with the Public Web API with Postman

Immagine che contiene interni, monitor, sedendo, computer

Descrizione generata automaticamente

Immagine che contiene screenshot, nero, monitor, sedendo

Descrizione generata automaticamente

Logging into the Public Web API with Postman

For another challenge, see if you can load your data into the public site. You can use the load\_data.py, but you’ll have to use the MONGODB\_URI that Heroku gave you and the new JWT\_SECRET\_KEY. Here’s a sample script below.

from tools.load\_data import load\_all  
  
MONGODB\_URI = 'YOUR-URI-HERE'  
JWT\_SECRET\_KEY = 'YOUR-KEY-HERE'  
new\_config = {'MONGODB\_SETTINGS': {'host': MONGODB\_URI, 'retryWrites': False},   
 'JWT\_SECRET\_KEY': JWT\_SECRET\_KEY}  
  
load\_all(new\_config)