**Using Flask to create Web Services**

**Why use Flask?**

* Build fast performing APIs out of the box quickly
* Uses the same python code you are familiar with
* Takes only few hours to learn
* Can be easily integrated into your current python project or a new project itself

**What is Flask:**

Barebones essential kit used to build a web app in python.

Barebones- provides key essential features, which includes:

* Receiving Http Requests
* Routing Http requests to Controller
* Dispatching Controller
* Returning Http Response to client

E.g:

@app.route('/books')

def hello\_world():

return 'Hello World!'

clients request to above endpoint and once this request is made then controller Hello\_world will run and we will get back Http Response in this case it’s Hello World!

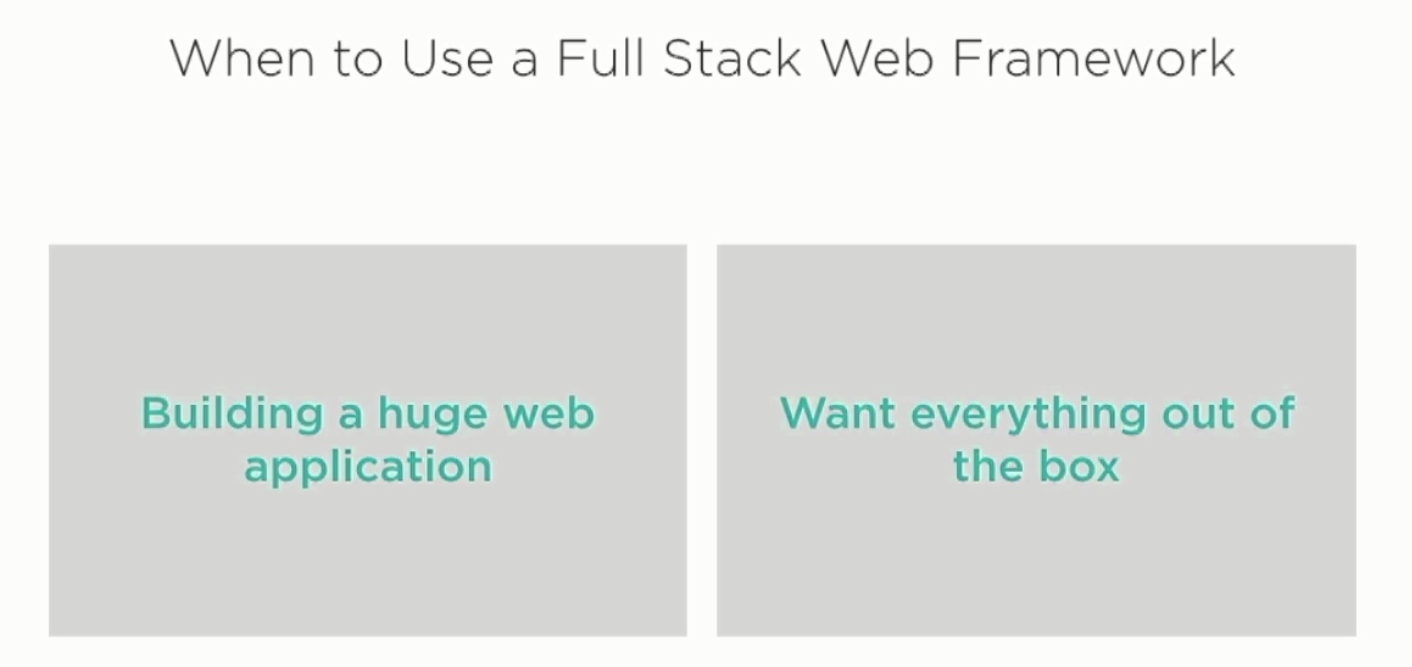
**Want more features in Flask:**

Flask out of the box does not provide this but Flask has plug-n-play model. You can add extensions as needed from Flask extension library. Pick whatever extensions you want or you could implement your own custom extensions.

**What about Full Stack Web Frameworks?**

Full stack web frameworks include Django and Rails. You would get a kitchen and a sink, which means they provide everything a developer needs such as:

Authentication, Authorization, DB abstraction, validating inputs, sanitizing inputs, templating engines, wipe down to how your html is generated.



**Limitations of a Full Stack Web Framework:**

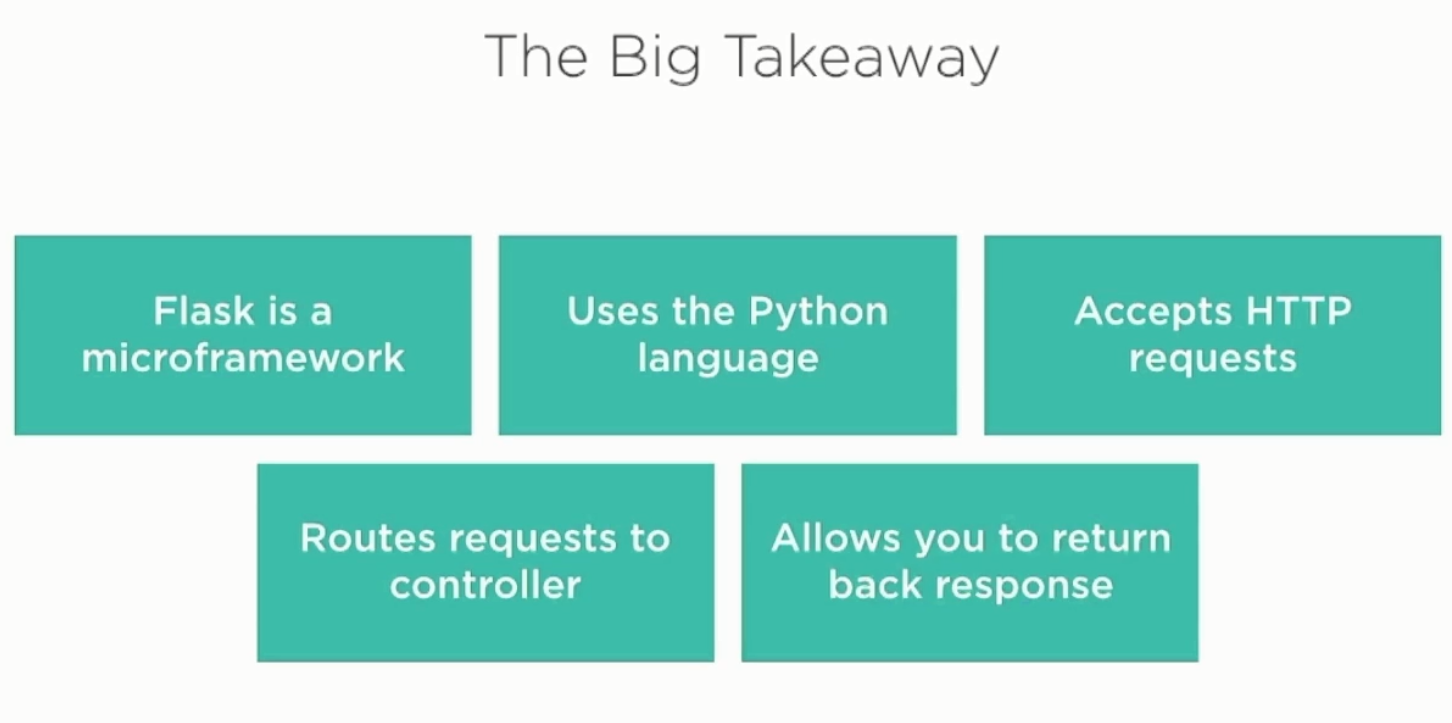
Can’t plug in different extensions freely because of features provided as inbuilt

Lack of flexibility of doing things outside of convention because these frameworks have a normalized way of doing things

**What do you get with using Flask:**

* Avoid bloating your web app
* Less unnecessary and complicated code

**Takeaway:**



* You would get excellent performance with Flask.
* Add functionality to Flask on a a-la-carte basis
* Add only what you need(avoid getting kitchen and sink)

**Why use Flask:**

If you want to build an api or application micro services, then Flask is an excellent choice.

If you want to use Pythion as well, Flask is one of the best options.

**What Flask is best for:**

* Build out APIs
* Very fast performing APIs
* Release fast performing APIs fast as it would take only hours to develop the APIs
* Write very little code as it is high level
* Prototyping code before starting development if your dev team uses a different stack and you need them to show how to structure your data then you can easily create the prototypes using flask
* Use Flask as a sandbox dev environment
* Test solutions using different modules and libraries
* You can also use Flask to develop web applications
* Using flask you would avoid “magic” black box code which happens as you can see how everything is happening and why
* Takes only couple of hours to learn
* Time investment is small
* Powerful enough to build enterprise level micro services and APIs
* Big takeaway: Excellent for building web services and micro services

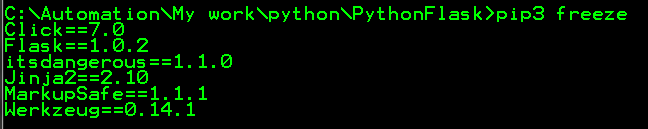
**Installing Flask:**

Flask recommends using python 3.3 or higher.

Install pip3 if you are using python3 or higher. You can select pip to be installed while installing Python in 3.7 so that you don’t need to explicitly install it.

Run the command to install Flask: **pip3 install flask**

To check if it is installed run the command: **pip3 freeze**



**Build your first Flask Application:**

Create a file with the below code and execute the file\_name using the command python filename.py

from flask import Flask

app = Flask(\_\_name\_\_)

print(app)

@app.route('/') # by default this would be a get request if second parameter is not specified

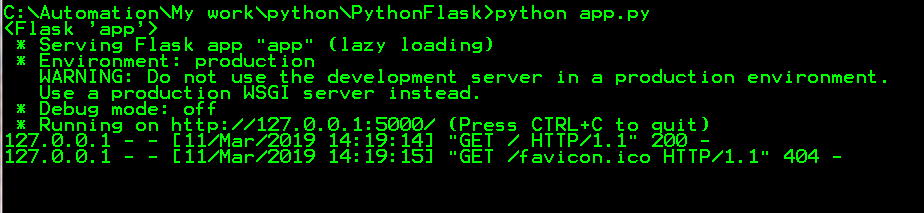
def hello\_world():

return 'Hello World!'

app.run(port=5000)

Note: @app is a decorator here. **Decorator – binds a function to a url**

When you run this program you will see the below msg:



Open a browser and navigate to: <http://127.0.0.1:5000/> which would return the output as shown below:



**Creating GETTER endpoints:**

**Create an endpoint to Get list of all books from a store**

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

books= [

{

'name': 'Green Eggs and Ham',

'price': 7.99,

'isbn': 978115161320

},

{

'name': 'The Cat in the Hat',

'price': 5.99,

'isbn': 9235632321353

},

{ 'name': 'Lord of the Rings',

'price': 10.99,

'isbn': 84651321333132

}

]

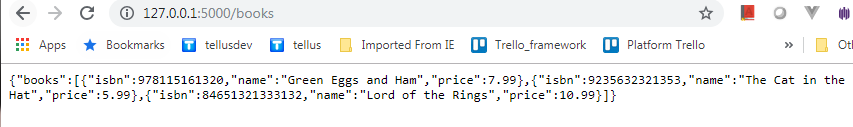
#GET /books #this would fetch all the books

@app.route('/books')

def get\_books():

return jsonify({'books':books}) #convert list into json object

**Response:**



**Create an api to fetch a specific book by using ‘/books/<ISBN>’**

#GET /books/9235632321353 #this would fetch a specific book from the books object

@app.route('/books/<int:isbn>')

def get\_book\_by\_isbn(isbn):

return\_value = {}

for book in books:

if book['isbn'] == isbn:

return\_value = {

'name': book['name'],

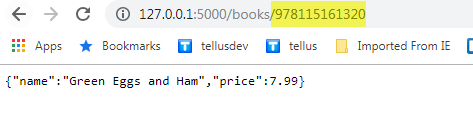
'price': book['price']

}

return jsonify(return\_value) #convert dict into json object

isbn is passed in as a request parameter to endpoint /books/156415615135 for ex, the router and function binding has to be updated to accept the request parameter as shown above

**Response:**



**Creating POST endpoints:**

We need to get the data from the request body to post the data to our endpoint. We need our client to send the data in the below format:

{

'name': 'A',

'price': 6.99,

'isbn': 161351033251

}

Here is how we use the controller:

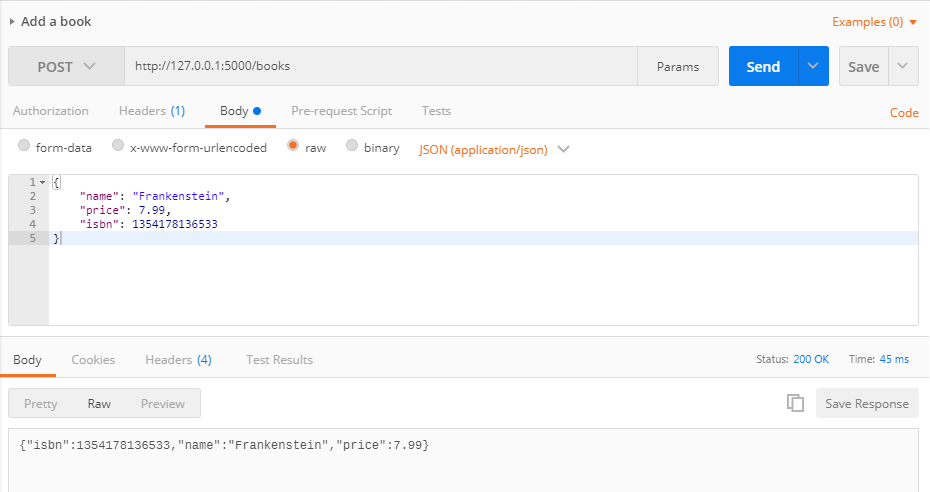
@app.route('/books', methods=["POST"])

Function params can be empty as shown below:



Let’s print the request json sent by the user using requests module present in Flask by using the method request.get\_json()

Send a postman POST request as shown below:

Note: make sure you set the MIME type in “Headers” section in postman to “application/json”. Also make sure you select JSON(application/json) in “Body” section. If you don’t specify it then request.get\_json() would return **null.** See<http://flask.pocoo.org/docs/1.0/api/#flask.Request.get_json> for more details.

So far this is how our code looks:

#POST /books

@app.route('/books', methods=["POST"])

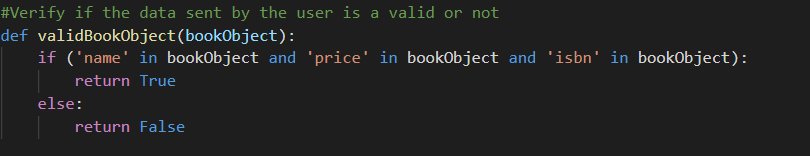
def add\_book():

return jsonify(request.get\_json())

**Sanitizing the data passed in the request by the user:**

Now that we can receive the response body client sends us, we could just take the data and insert into books collection, but this could cause potential problem as clients can send any data and we would end up parsing it. To avoid this and we can sanitize(filter) the data so that only valid data would be added to books collection.

Here’s how we would sanitize the data



How do we test the above function?

Take a new file and write the above function and add the below test data to it:

valid\_object = {

"name": "Harry Potter",

"price": 7.99,

"isbn": 418461516547862

}

missing\_name = {

"price": 7.99,

"isbn": 418461516547862

}

missing\_price = {

"name": "Harry Potter",

"isbn": 418461516547862

}

missing\_isbn = {

"name": "Harry Potter",

"price": 7.99

}

empty\_book = {

}

You can either call the function with different test data in the file or call it from the repl as shown below:

Approach #1:

print(validBookObject(valid\_object))

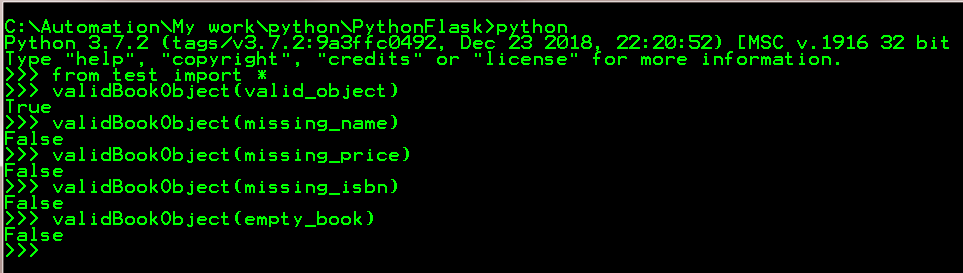
print(validBookObject(missing\_name))

print(validBookObject(missing\_price))

print(validBookObject(missing\_isbn))

print(validBookObject(empty\_book))

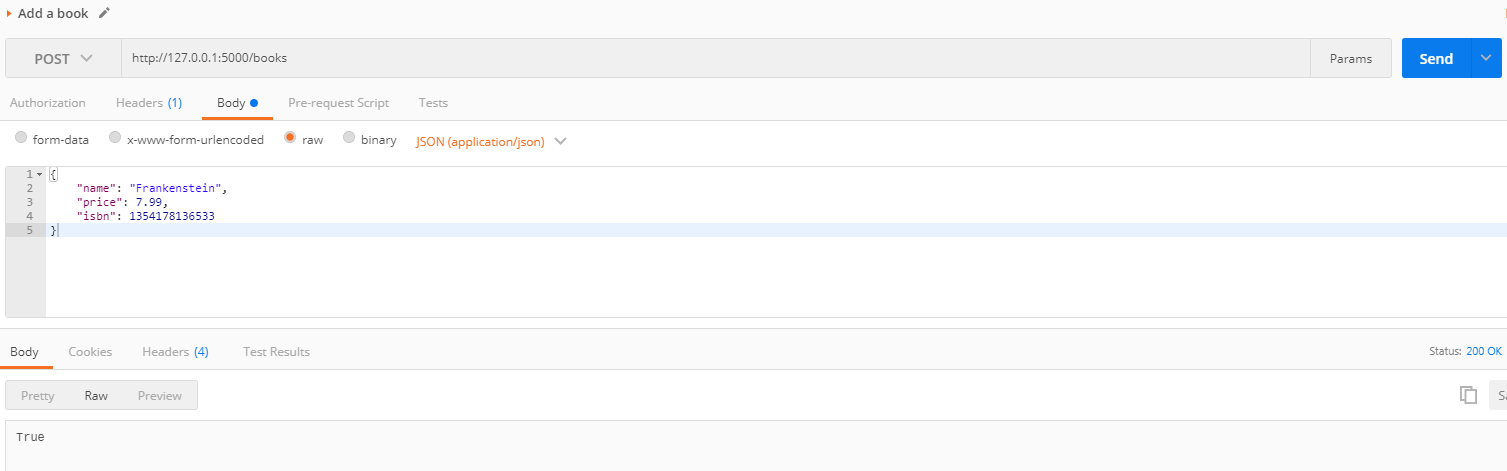
Approach #2:



Now let’s add the new book object to books collection

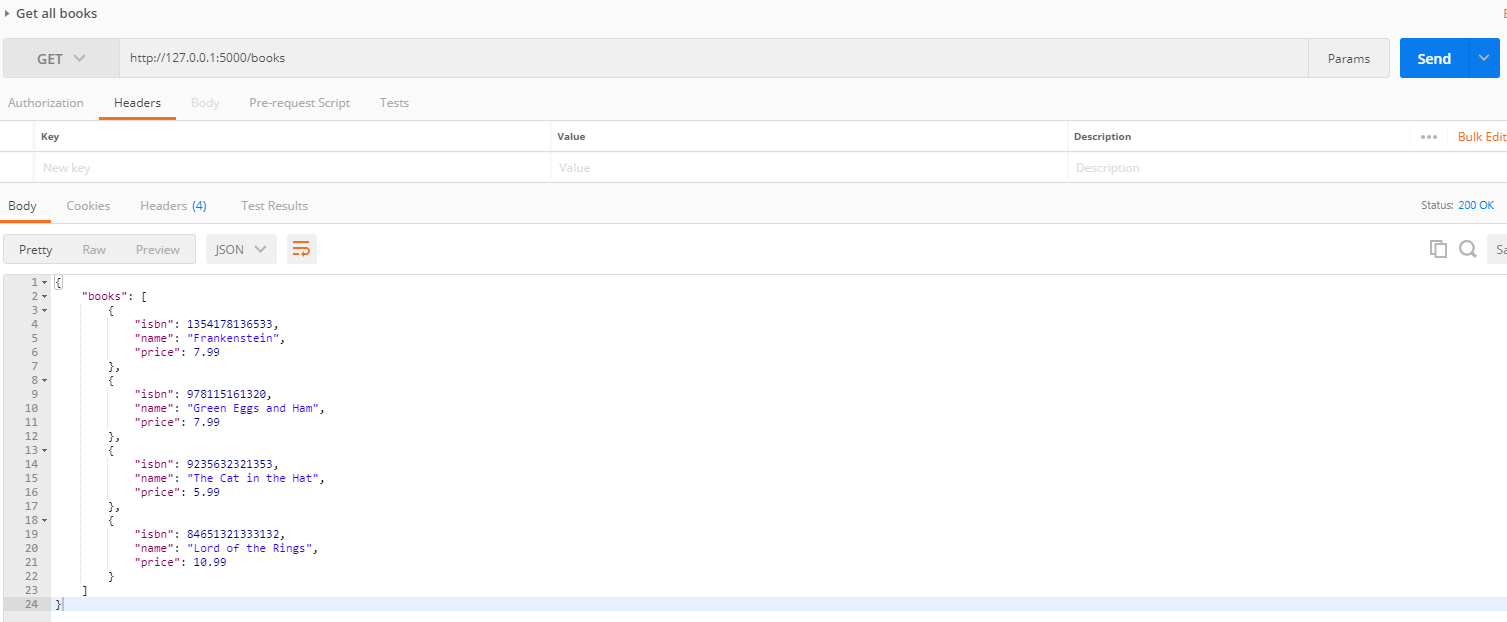
We are returning the string “True” on success and “False” on invalid data passed because in Flask whenever we return a string from a route, it would automatically set the response type to text/html and set the status code to **200**

**Output:**





If we send a get request now, the output will have the above new book data as well added at the beginning as shown below:



**Mistake to be aware of when using POST using Flask:**

If we send Garbage data along with the valid data the data would still be inserted in the books collection. For ex:

{

"name": "Vasu Nakka",

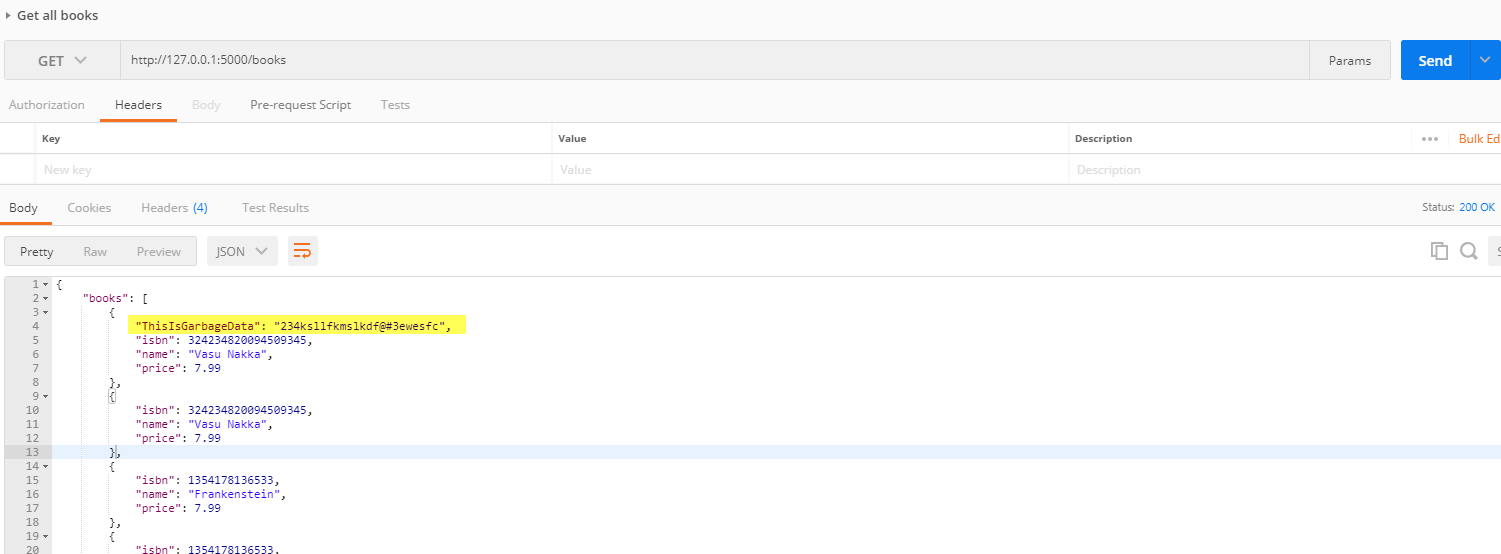
"price": 7.99,

"isbn": 324234820094509345,

"ThisIsGarbageData": "234ksllfkmslkdf@#3ewesfc"

}

In above case, ThisIsGarbageData key is invalid and not needed to pass to Add Book api request but our code inserts the new book data into books collection. See



We cannot disallow inserting the data into book collection, instead we can filter out what data we need to insert into book collection as shown below:

new\_book = {

"name": request\_data['name'],

"price": request\_data['price'],

"isbn": request\_data['isbn']

}

With addition of the above code the add\_book function would look like this:

@app.route('/books', methods=["POST"])

def add\_book():

request\_data = request.get\_json()

if(validBookObject(request\_data)):

new\_book = {

"name": request\_data['name'],

"price": request\_data['price'],

"isbn": request\_data['isbn']

}

books.insert(0, new\_book) #insert the new book at the first value

return "True"

else:

return "False"

**Sending status codes and response bodies for POST requests:**

In the above code for add\_book(), we are returning Strings(“True”/”False”) back to the client. By returning any string in Flask, we are implicitly setting the **Status Code** and **Content Type** as **200** and **text/html** back to the client. We should use a more specific Status code to indicate the client on what happened for their request.

In above example we can return a status code of 201 which represents the request has been fulfilled and one or more records has been created.

To set the Content Type and status code, we need to use Response class as shown in below format:

Response(response, status, **headers**, mimetype, content\_type, direct\_passthrough)

Ex: response = Response(“”, 201, mimetype=”application/json”)

Note: We need to use Response class in Camel case only for Response while importing whereas requests is imported in small case only.

Our code looks like:

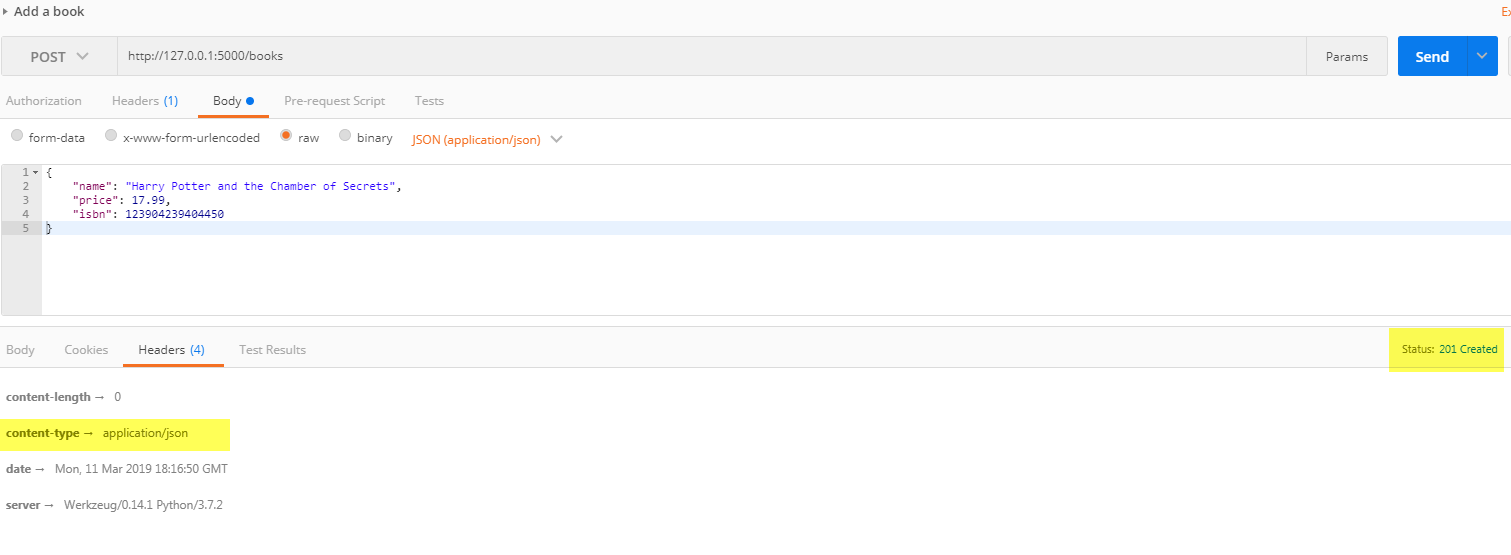
from flask import Flask, jsonify, request, Response

books.insert(0, new\_book) #insert the new book at the first value

response = Response("", "201", mimetype="application/json")

return response

This is how the response looks like after we make a change and restart the application:

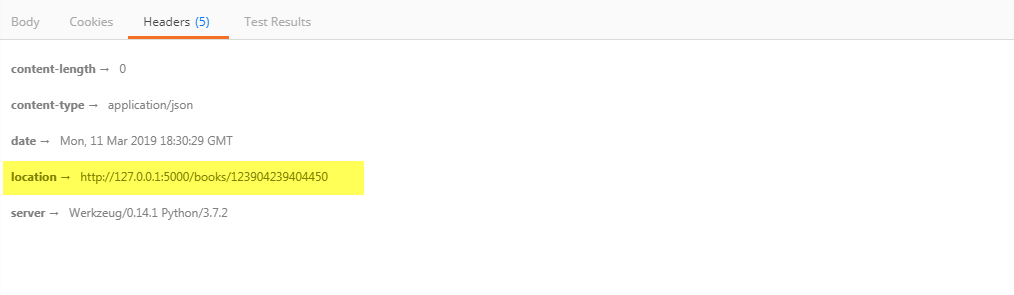


**Setting Location Headers for POST requests in Flask:**

In the last example, the client have received an empty string in the response body. The reason why is, client will receive access to a resource created not from the actual response body but from location header.

In Flask the headers work as a key,value pairs. Once we have the response then we can set the headers as shown below:

response.headers['Location'] = 'books/'+ str(new\_book['isbn'])



**Handling invalid POST requests:**

We are returning “False” if there is invalid book object sent the user. Instead of sending “False” we can send appropriate status codes along with the response.

We have added the below code in else block

invalidBookObjectErrorMessage = {

"error": "Invaid book object passed in the request",

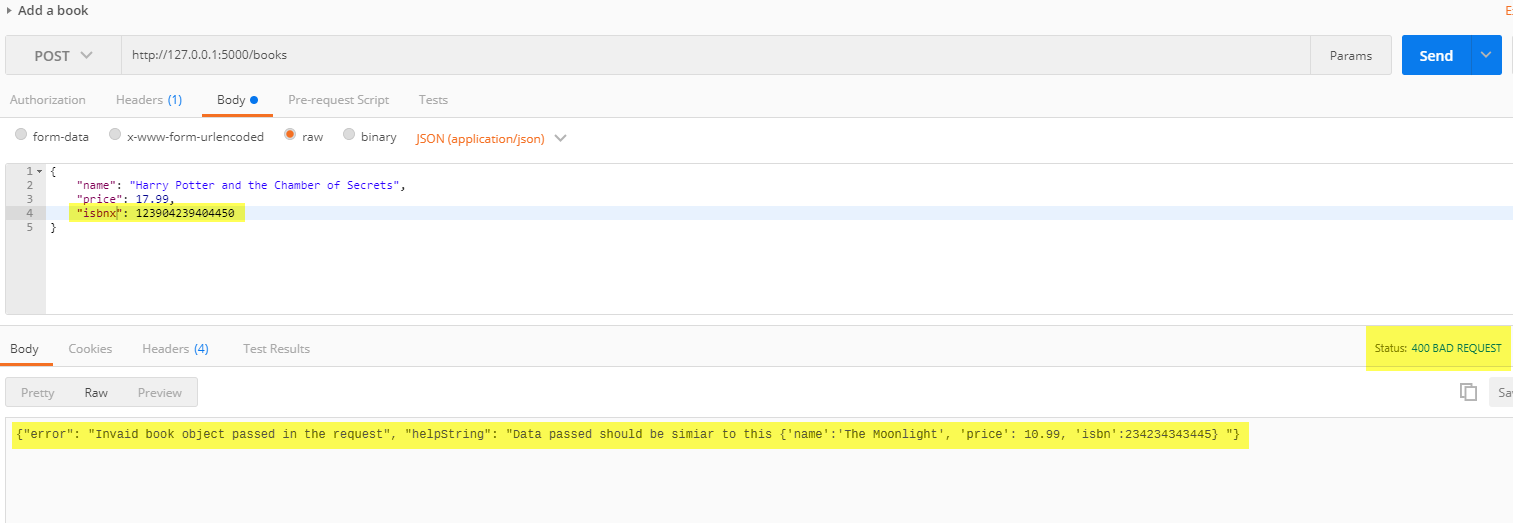
"helpString": "Data passed should be simiar to this {'name':'The Moonlight', 'price': 10.99, 'isbn':234234343445} "

}

response = Response(json.dumps(invalidBookObjectErrorMessage), status=400, mimetype="application/json")

return response

Output:



**Updating Data:**

Let’s update the data in the books collection by sending a PUT request which woud replace existing book object in books collection. Here’s how we can define the route:

@app.route('/books/<int:isbn>', methods=['PUT'])

Note that we need client to send all the data to update even though they want only some specific fields to update. If a client wants to update only partial data then they can use PATCH instead of PUT.

The put request to replace a book is implemented as shown below:

@app.route('/books/<int:isbn>', methods=['PUT'])

def replace\_book(isbn):

update\_request = request.get\_json()

new\_book = {

'name': update\_request['name'],

'price': update\_request['price'],

'isbn': isbn

}

i = 0

for book in books:

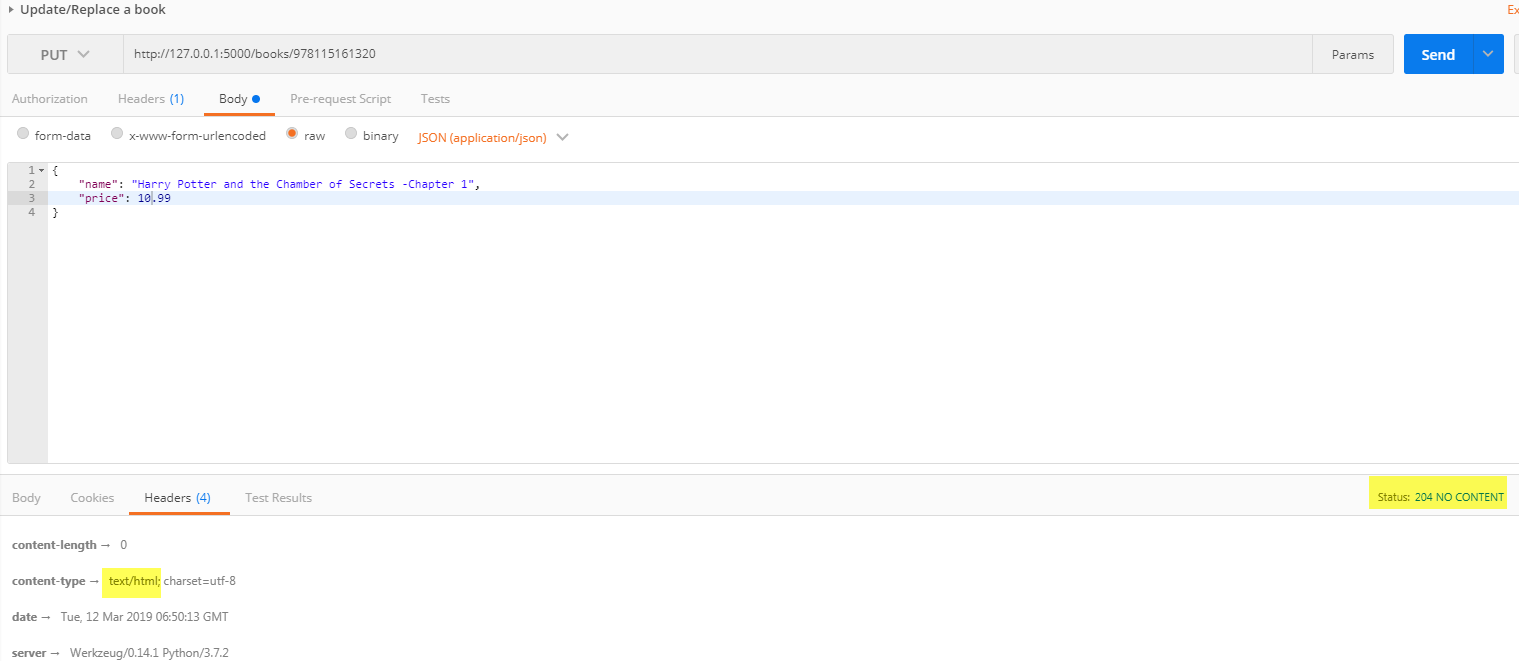
if(book['isbn'] == isbn):

books[i] = new\_book

i+=1

return Response("", status=204)

Output:



Let’s sanitize the input data sent by the client by adding the function below:

#sanitize the data sent by the user for the put request

def validPutRequestData(request\_data):

if ('name' in request\_data and 'price' in request\_data):

return True

else:

return False

After adding the code the final definition of the PUT request implementation looks like:

@app.route('/books/<int:isbn>', methods=['PUT'])

def replace\_book(isbn):

update\_request = request.get\_json()

#sanitize the input we are receiving as part of the request

if(not validPutRequestData(update\_request)):

invalidBookObjectErrorMessage = {

"error": "valid book object must be passed in the request",

"helpMessage": "data passed should be in this format {'name':'Harry Potter','price':11.99}"

}

response = Response(invalidBookObjectErrorMessage, status=400, mimetype="application/json")

return response

new\_book = {

'name': update\_request['name'],

'price': update\_request['price'],

'isbn': isbn

}

i = 0

for book in books:

if(book['isbn'] == isbn):

books[i] = new\_book

i+=1

response = Response("", status=204)

return response

Creating PATCH requests:

Now let’s create a PATCH request so that user can only send the partial data(which they want to) and we will update that only instead of replacing the book object.

Here is the code that would implements the functionality of a patch request:

#sanitize the data sent by the user for the put request

def validPatchRequestData(request\_data):

if ('name' in request\_data or 'price' in request\_data):

return True

else:

return False

#patch request for allowing a user to send only the data that needs to be updated

@app.route('/books/<int:isbn>', methods=['PATCH'])

def update\_book(isbn):

request\_data = request.get\_json()

updated\_book = {}

if 'name' in request\_data:

updated\_book['name'] = request\_data['name']

if 'price' in request\_data:

updated\_book['price'] = request\_data['price']

#santize the input we are receiving as part of the request

if(not validPatchRequestData(request\_data)):

invalidBookObjectErrorMessage = {

"error": "valid book object must be passed in the request",

"helpMessage": "data passed should have either {'name':'Harry Potter'} or {'price':11.99} or both"

}

response = Response(invalidBookObjectErrorMessage, status=400, mimetype="application/json")

return response

else:

for book in books:

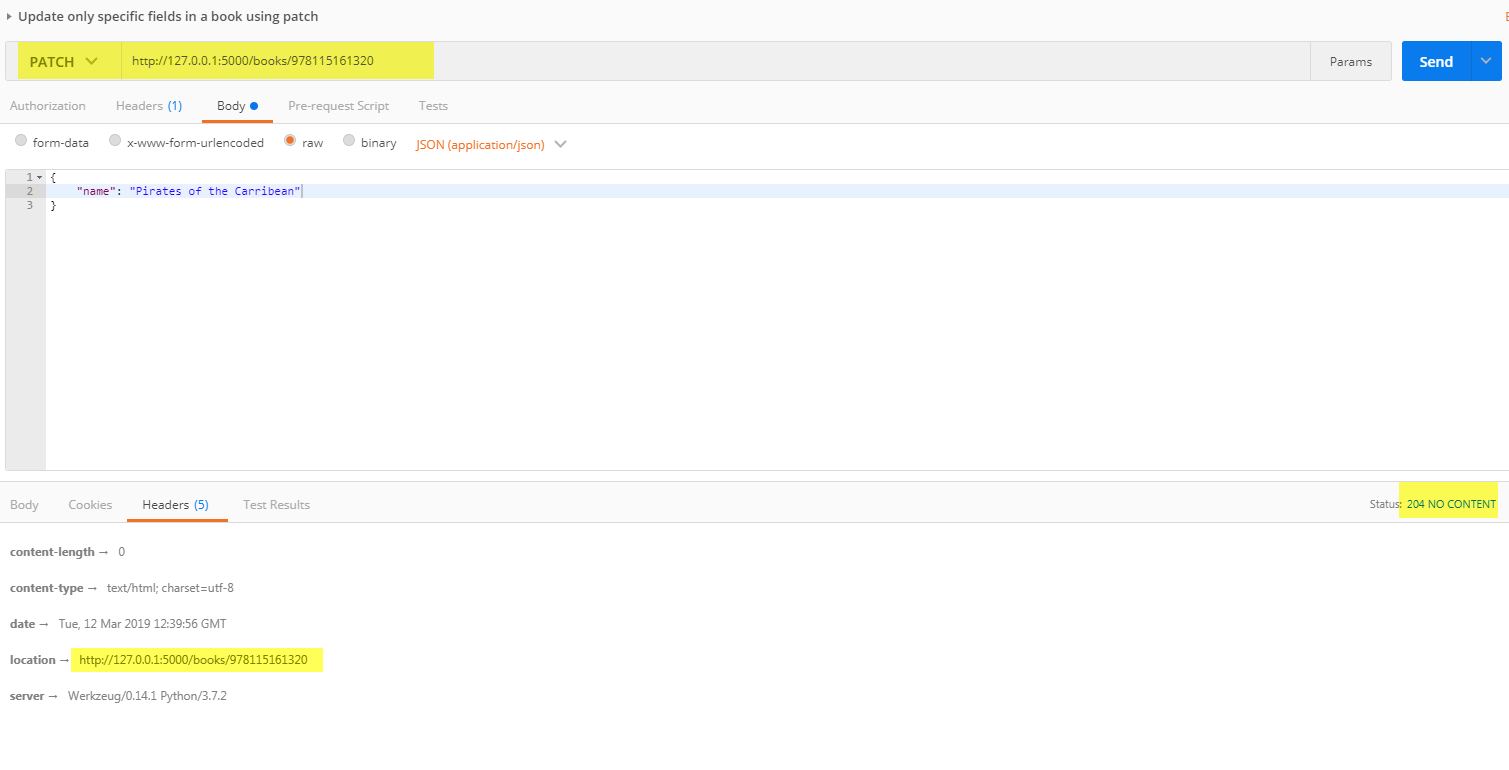
if(book['isbn'] == isbn):

book.update(updated\_book)

response = Response("", status=204)

response.headers['Location'] = '/books/' + str(isbn)

return response



Note: There is a bug in the above logic if we send a valid patch request object but the data is not present in the books collection we would still get 204 as a response.

Creating Delete request:

Let’s start adding a new route before we implement the delete method. The delete request deletes the entry from the books collection.

Here is the code that would delete a book from a store:

#DELETE /books/9235632321353 #this would delete the book entry from books collection by accepting the isbn

@app.route('/books/<int:isbn>', methods=['DELETE'])

def delete\_book(isbn):

request\_body = request.get\_json()

i = 0

for book in books:

if(book['isbn']== isbn):

books.pop(i)

response = Response("", status=204)

return response

i+=1

invalidBookObjectErrorMessage = {

"error": "Book with isbn provided is not found. Hence, cannot delete the book"

}

response = Response(json.dumps(invalidBookObjectErrorMessage), status=400, mimetype="application/json")

return response

If the request succeeds, we would get 204 and if the book with isbn is not found then we would get 400 with an error message returned to the client.

Storing the data in the DB:

Until now, we have hardcoded the data in the python app and have been performing operations on it. If we restart the app, then the changes made to it would be lost.

Let’s fetch the data from the DB so that the changes made to the data would be persistent even after restarting the app.

We will be using **sqlite** DB to fetch the data from, to do this let’s create a setttings.py file in our project folder. The settings.py file is used to store the settings of our application such as DB configuration.

**Configuring a Flask SQLAlchemy DB:** The following code in settings.py file would configure the SQLAlchemy database:

from flask import Flask

app = Flask(\_\_name\_\_)

#specify the path to the DB

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:////c/Automation/My work/python/PythonFlask/database.db'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

We can remove the books object in books.py file and add an import statement as shown below:

from settings import \*

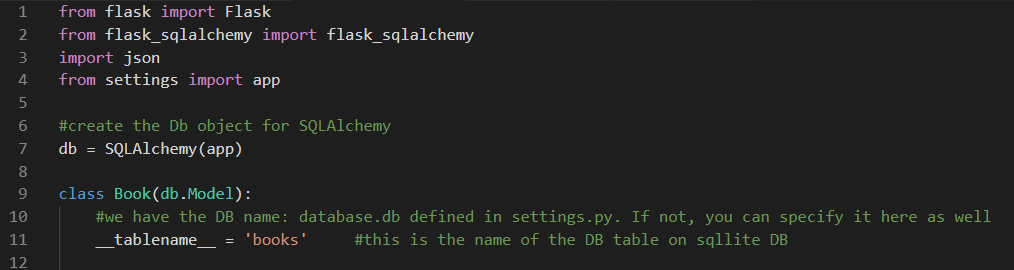
Now let’s define the tables and columns in the Flash Alchemy DB

Let’s install sqlalchemy module using pip so that we can use it to establish the DB connection.

E.g: pip/pip3 install flask\_sqlalchemy

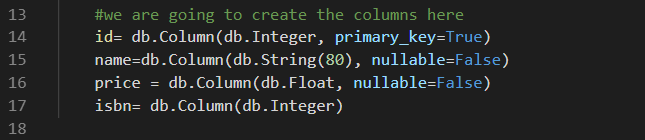
To check if the module is installed run “pip freeze” and which would list the installed modules.

Let’s create the DB connection and tablename below in a new file BookModel.py:



db.Model is the base class and Book is the child class where we would write the code to connect to Db and run our webservices.

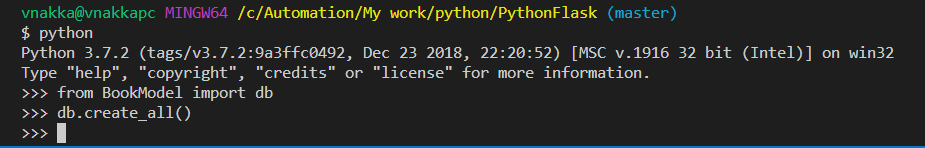
Now that we have configured the database, created the DB object and defined the table named “books” in sqllite DB, let’s add the columns now to this table:



SQLALchemy would map the python variables with the columns present in the books table in sqllite DB.

Perfect, now let’s go ahead and create the database now. We can create the DB from the db object we have defined in BookModel.py file. Here’s how we can create the db object and create the database file using python interpreter:

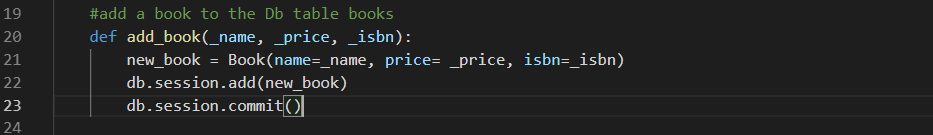
**Note: This needs to be executed only once as we create the Db only once.**



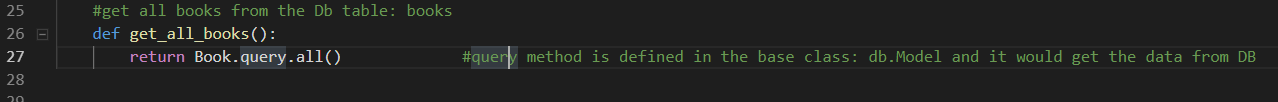
This would create the database filename at the path specified in app.config[‘SQLALCHEMY\_DATABASE\_URI’]

**Defining a Data Model with Flask SQL Alchemy:**

Let’s add a new book to the DB by creating a function as shown below:



We can also get all the data from the books table as shown below:

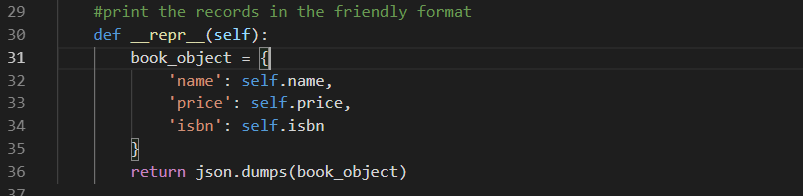


Here Book class is inherited from db.Model which has query method we can use it to fetch all rows from the table by calling Book.query.all()

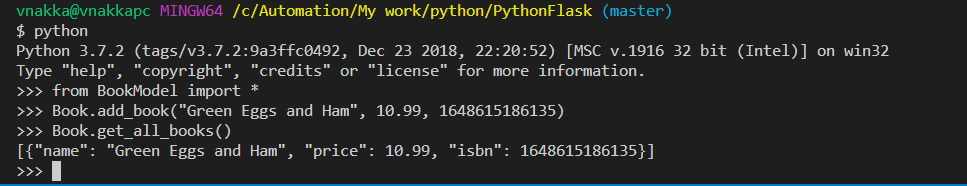
If we execute get\_all\_books() it will print the output in the below format:

[<Book1>, <Book2>, <Book3>]

We can also print the output from get\_all\_books to be returned in a friendly format. Let’s do that with this:



To test it, open python interpreter and run the commands as shown below:



**Add a function to get a book by providing the isbn number:**

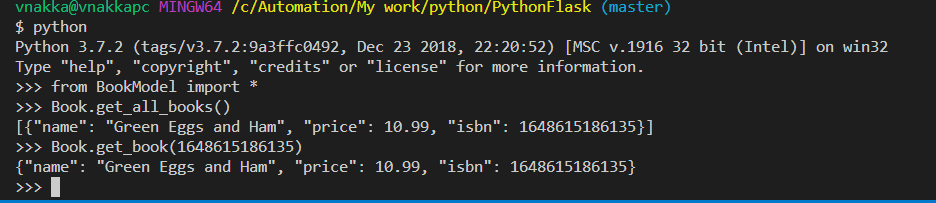
#get a book from the isbn number provided

def get\_book(\_isbn):

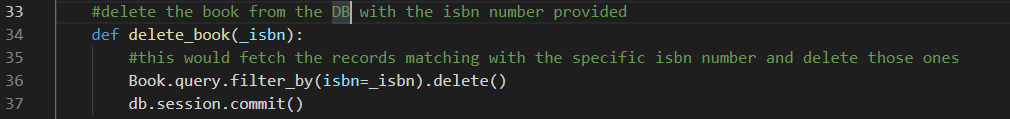
return Book.query.filter\_by(isbn=\_isbn).first()

filter\_by takes the parameter and it would match the column we defined and replaces the \_isbn value with the input provided by the user.

This would return the rows but if we want only first record we can chain with .first() so that it would return the first book object.

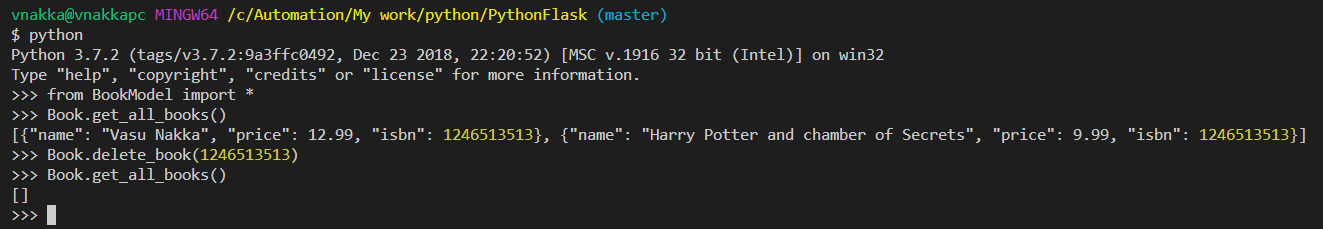


**Delete a book from the DB by providing isbn number as an input:**

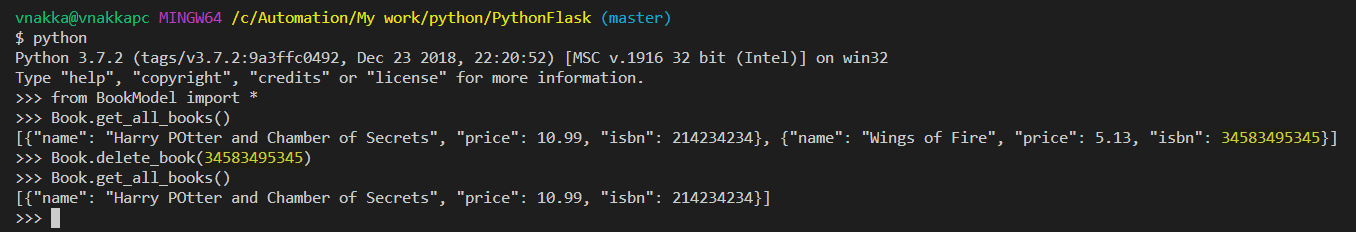


We would run the filter\_by query with the isbn number as an input and then execute delete() on them which would delete the matching records for the isbn number. See

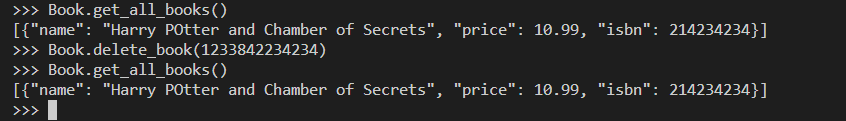
1. If we have multiple records with the same isbn number it will delete all records as shown below:



1. If we have a single entry with isbn number provided it will delete that record as shown:



1. If the isbn doesn’t match with the DB records, our code doesn’t throw any exception and the output would be as shown below:



**Update a book with the isbn number provided:**

To update the other fields such as price or name, with the isbn number provided we would first run the query.filter\_by() and update the respective fields as shown below:

#update the price for the book with a specific isbn number provided

def update\_book(\_isbn, \_price):

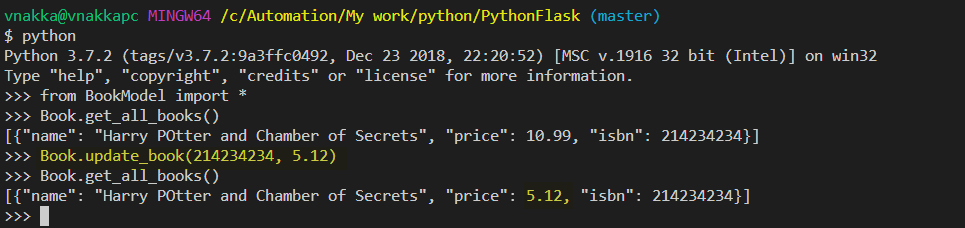
book\_to\_update = Book.query.filter\_by(isbn=\_isbn).first()

#the below command would update the price for the fetched records

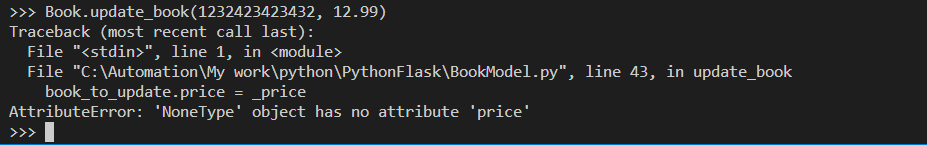
book\_to\_update.price = \_price

db.session.commit()

The above code will update the price for the entry with the isbn provided. Here’s how it would look if the valid isbn is provided



If there is no isbn corresponding to the input provided, it would throw an exception as shown below:



Here’s how the function to update the book name with the isbn provided as an input would look like:

#update the price for the book with a specific isbn number provided

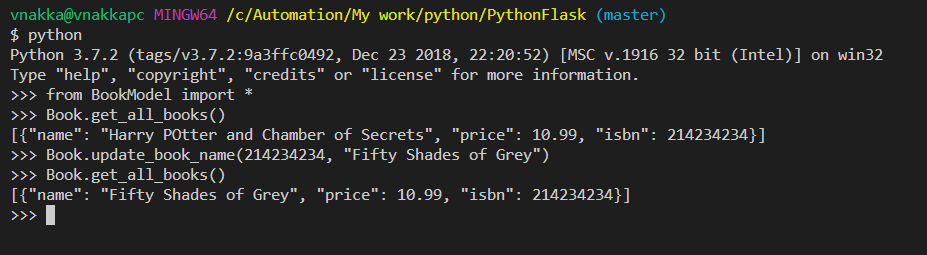
def update\_book\_name(\_isbn, \_name):

book\_to\_update = Book.query.filter\_by(isbn=\_isbn).first()

book\_to\_update.name = \_name

db.session.commit()

The below output would be displayed upon executing it



**Replace a book with new name and price with the isbn value:**

To replace a book with a new name and price, here is how we would do it:

#replace a book with name and price fields with the isbn provided

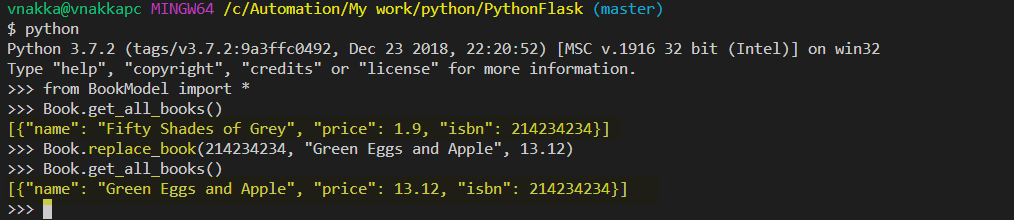
def replace\_book(\_isbn, \_name, \_price):

book\_to\_update = Book.query.filter\_by(isbn=\_isbn).first()

book\_to\_update.name = \_name

book\_to\_update.price = \_price

db.session.commit()

Running the function on python interpreter would display the output as shown below:

We need to find out a way for our application to get JSON representation of the data we are getting back from these queries. If we don’t do this, we would receive the SQLAlchemy objects that aren’t friendly to work with.

The \_\_repr\_\_ function would only display the data on the terminal to us in a friendly manner.

Let’s define a function that would convert the data to json format:

#converts the object into json format

def json(self):

return {'name': self.name, 'price':self.price, 'isbn':self.isbn}

Updating our function below to convert the SQLAlchemy objects into json format:

#get all books from the Db table: books

def get\_all\_books():

return [Book.json(book) for book in Book.query.all()] #query method is defined in the base class: db.Model and it would get the data from DB

**Updating BookModel to use Flask app:**

Let’s update the BookModel to use the Flask app so that we can run it as an api. The only change is in delete\_book() which returns True/False depending on the status of the operation. We need to make changes to books.py and it would look like:



We can run the app using the flask command but the following needs to be set before we run the flask run command as flask run would looks for app.py file to launch. If we have any other file like books.py in our case execute the below commands before you run “flask run” command

set FLASK\_APP=hello.py

set FLASK\_ENV=development

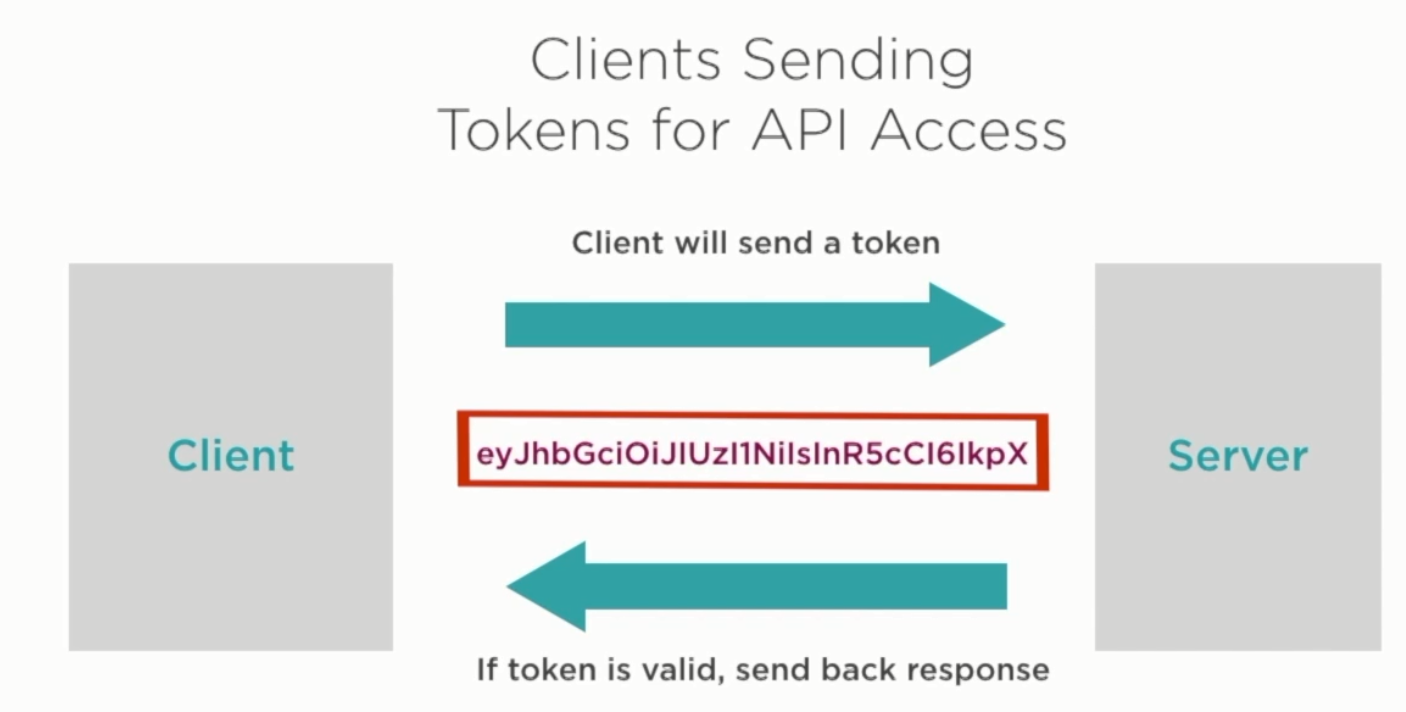
flask run

**Adding Authentication to our API:**

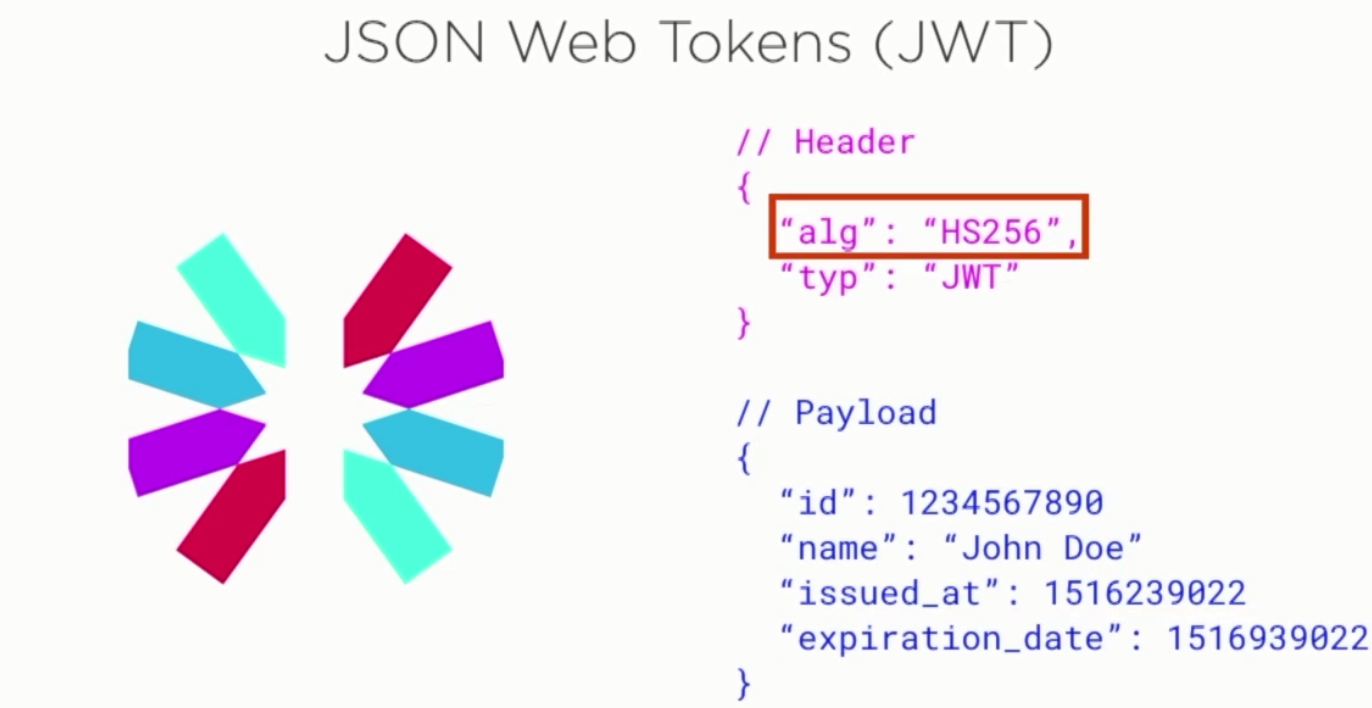
Let’s move ahead and add authentication to our API.

JWT is a way to handle authentication for our API without having our client to send us the username and password over for every api call.

We will do this by requiring our clients to send us the **JSON Web Token** in short called as **JWT** when requesting one of our APIs. We would grant access to the API only if the JWT is valid.



JWTs are JSON encoded objects that carry information about the client in between the requests. This information can be anything you want. JWTs typically include the following:



Let’s create a user table in our DB to store the user who have been granted access to our API and then use the SQL Alchemy ORM to apply things. Here’s how we would set up our DB for auth:



To use JWT, we need to install it using the pip command: **pip install PyJWT**

In our app.py(which was books.py), we need to define the secret key for our app by adding the line below:

app.config['SECRET\_KEY'] = 'vnakka'

**Creating JWT Tokens:**

Once this is done, we need to add a route to fetch the jwt token for everyone. In our app, we will allow everyone to send a request to fetch the jwt token.

#GET /login #this would fetch a jwt token with an expiration time of 100 seconds from now

@app.route('/login')

def get\_token():

expiration\_date = datetime.datetime.utcnow() + datetime.timedelta(seconds=100)

token = jwt.encode({'exp': expiration\_date}, app.config['SECRET\_KEY'], algorithm='HS256')

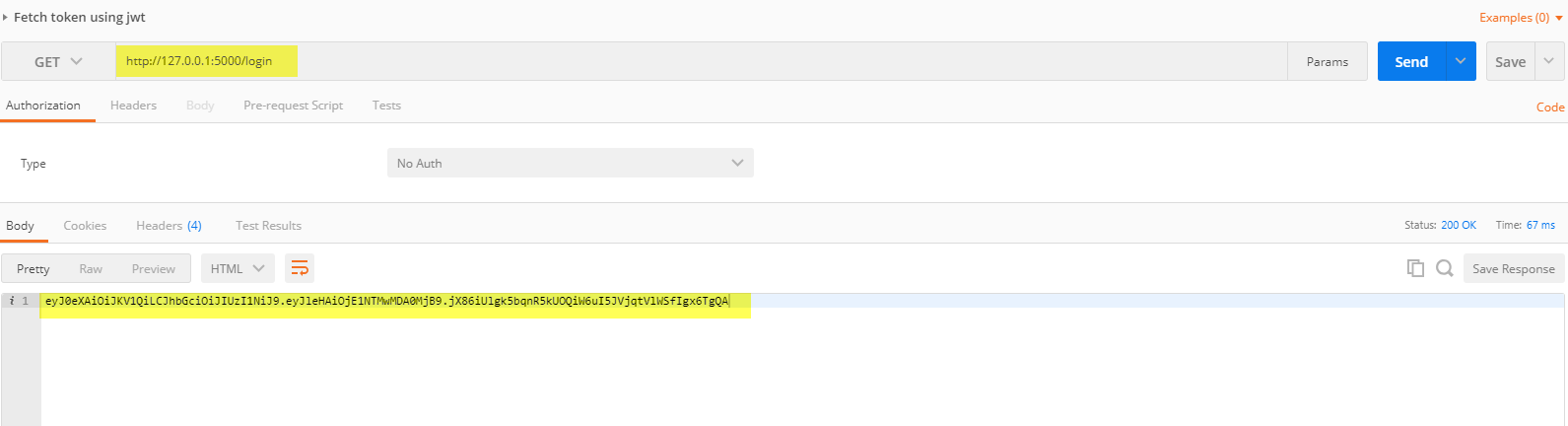
return token

We would use jwt.encode() which has the following syntax:

**encode(self, payload: dict, key: jwt.jwk.AbstractJWKBase = None, alg='HS256', optional\_headers: dict = None) -> str**

here we would use expiration\_date as a dict for payload, app-secret key and algorithm used to generate the token.

Here’s how we would test it:



As you can see the token is generated and it would expire in 100 seconds after it has been requested.

To decode the token back and get the payload data here’s how we would do it:

jwt.decode(jwt\_token, app.config['SECRET\_KEY'], algorithms=['HS256'])

**Adding Authentication to our GET route:**

Since we have defined a route to fetch the token we can use it in our route to get all the books endpoint by passing token as a query parameter and decode it. If it is successful then we would display the books data otherwise we would return an error message saying that “user has to provide a valid token to access the data”.

Here is the updated route for getting the /books using token value:

#this would fetch all the books

#GET /books?token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJkYXRhIjoicGF5bG9hZCJ9.SePiJiKnHWTZjk0SvBPqkX9G\_wtG9JqtNPKRrky8CDI

@app.route('/books')

def get\_books():

token = request.args.get('token')

try:

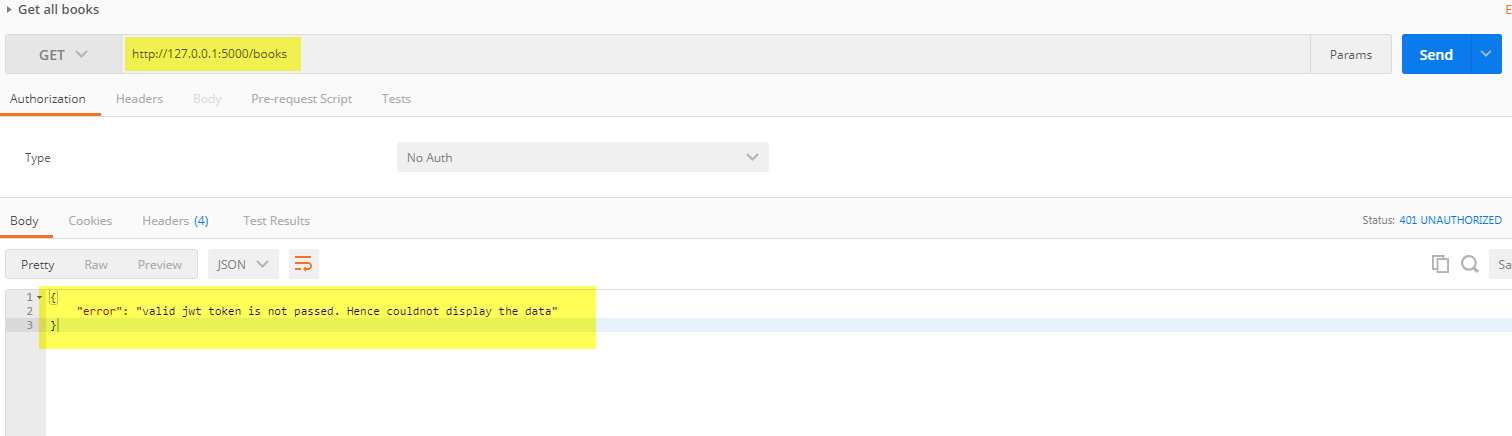
jwt.decode(token, app.config['SECRET\_KEY'])

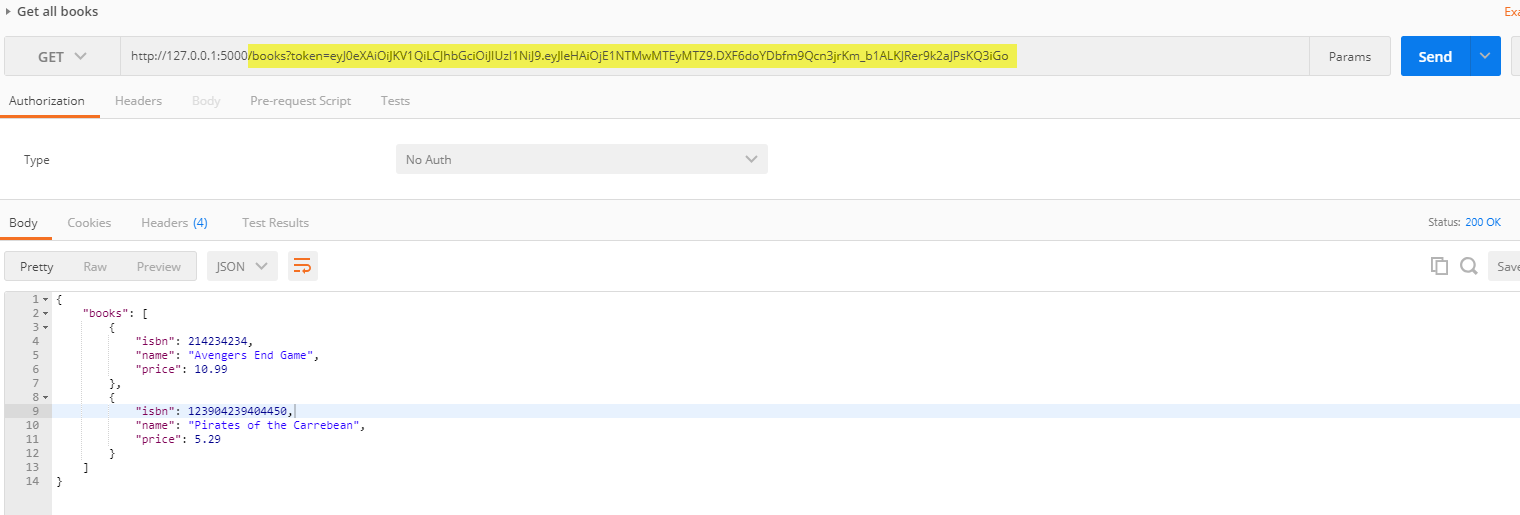
except:

return jsonify({'error': 'valid jwt token is not passed. Hence couldnot display the data'}), 401

return jsonify({'books': Book.get\_all\_books()}) #convert list into json object

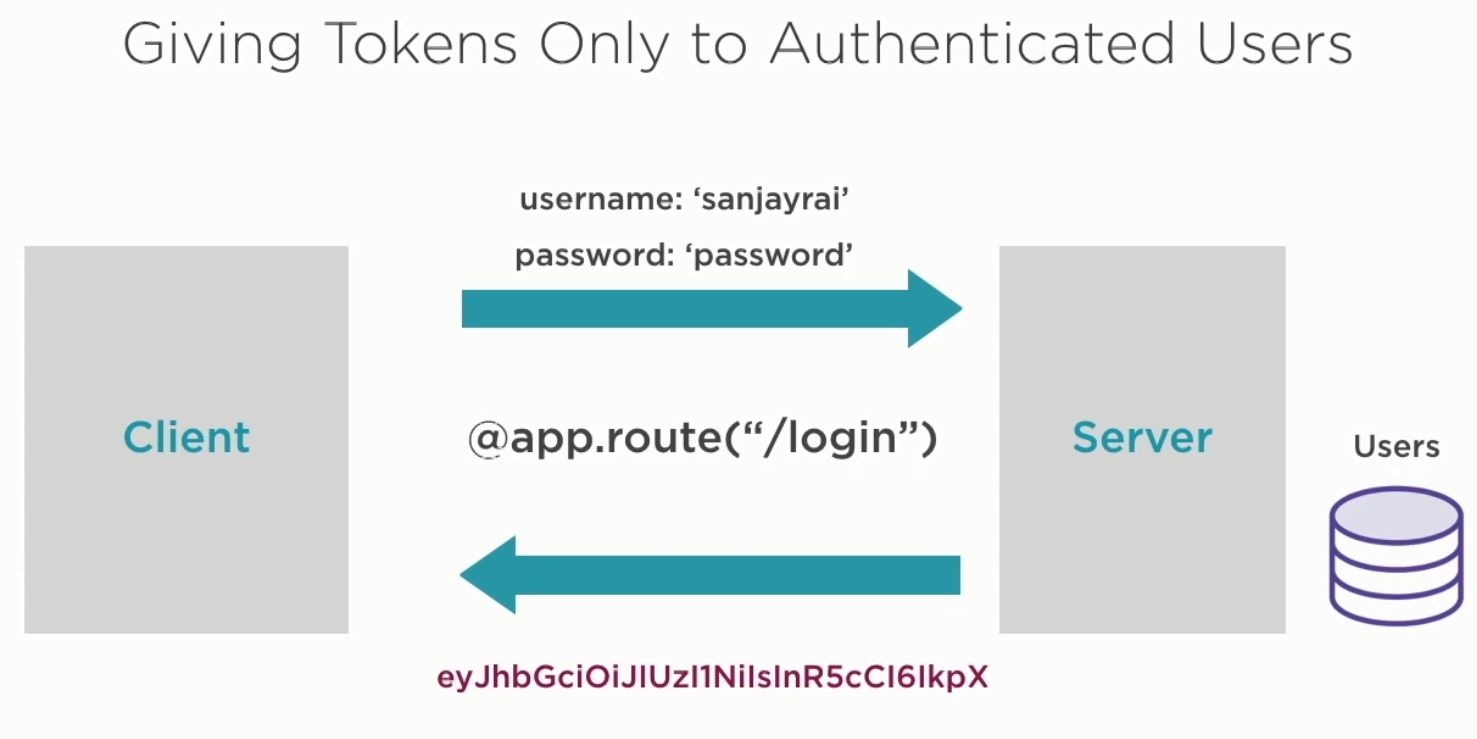
Let’s do a basic sanity test for this with/without passing the token:





**Creating a DataBase for Users who are allowed to access the API**

Until now, we allowed everyone to access the API by generating the token and using it to access the data. Let’s move on to provide restricted access to the API and allow only specific users to access the data as shown below:



To achieve this, we will create a new table named “Users” in our DB and add the users so that they would be able to access the api.

Clients would send their credentials to login route to get the token to use in the subsequent requests. We would add functionality in login route to verify if the user is valid or not and provide the token if the user is authorized. Otherwise, display an error as shown previously.

**Note: In our example, we will be using plain text passwords to store the data in our DB. On Production environments it’s recommend to use an authentication library because libraries provide features such as security and encoding passwords for you.**

Let’s get started by creating a User Model that would map the Python class to sql table. Once it is done, we will define the Users table which is similar to how we have done for BookModel module.

I have created a file named UserModel.py and added the below code to define the users table with the appropriate columns:

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from settings import app

db = SQLAlchemy(app)

class Users(db.Model):

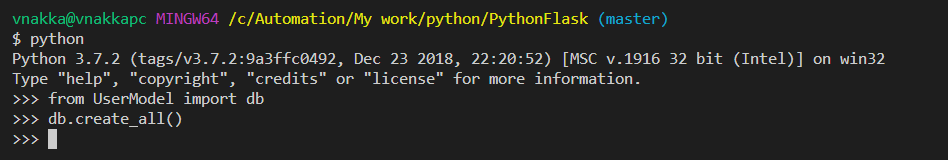
\_\_tablename\_\_ = 'users' #if we don't specify it then class name would be used as tablename implicitly

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

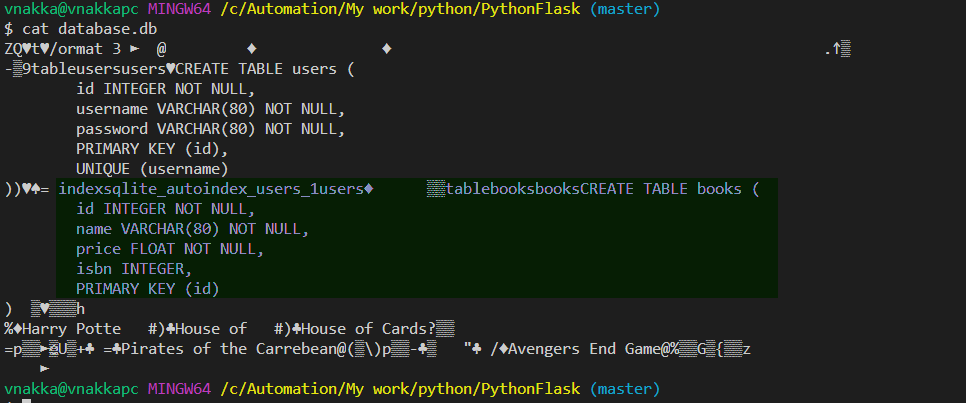
password = db.Column(db.String(80), nullable=False)

Here’s how we would execute this file to create the db table “users”:



How would we know if the db table is created or not:

Run the “cat database.db” command in git/shell prompt and it would display the output as shown:



**Creating Users that are allowed to have API access:**

Now that the Db table “users” is created, we will now add the users to this table so that they would have access to API to perform operations on books table.

Before we add the logic for adding users, let us define how we want to represent the output data. We would do it by using the below function:

def \_\_repr\_\_():

return str({

'username': self.username,

'password': self.password

})

This would print the output very similar to JSON.

Adding few functions that would verify if the username and password matches or not, getting the list of all users and adding a new user as shown below:

#this would verify if the username and password entered by user matches or not. Returns a boolean value

def username\_password\_match(\_username, \_password):

user = User.query.filter\_by(username=\_username).filter\_by(password=\_password).first()

if user is None:

return False

else:

return True

# get list of all users

def getAllUsers():

return User.query.all() #this would print the data in repr format as shown below

#create a new user

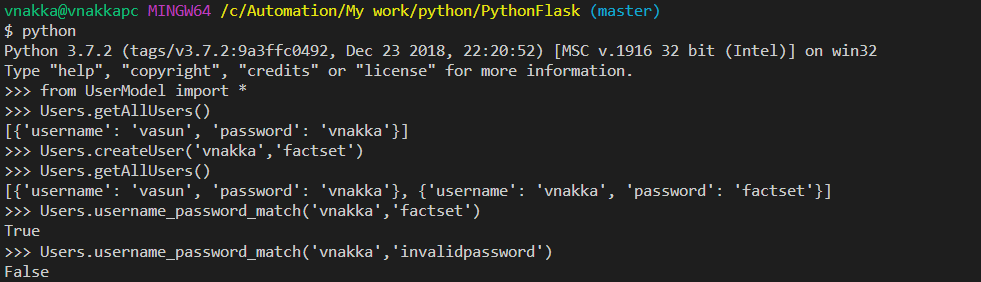
def createUser(\_username, \_password):

new\_user = User(username=\_username, password=\_password)

db.session.add(new\_user)

db.session.commit()

This is the output if we test it on python interpreter:



**Having Users Login to get a Token:**

Let’s go to app.py and in the route for login, let us make it as a post method as we want to accept the username and password from the client and call the Users.username\_password\_match() in UserModel.py file as shown below:

#POST /login #this would fetch a jwt token with an expiration time of 500 seconds from now

@app.route('/login', methods=['POST'])

def get\_token():

request\_data = request.get\_json()

username = request\_data['username']

password = request\_data['password']

user\_match = Users.username\_password\_match(username, password)

if user\_match:

expiration\_date = datetime.datetime.utcnow() + datetime.timedelta(seconds=500)

token = jwt.encode({'exp': expiration\_date}, app.config['SECRET\_KEY'], algorithm='HS256')

return token

else:

return Response(json.dumps({'error': 'username and passwords are not matching with the record in our DB. Please enter correct credentials'}), 401, mimetype="application/json")

**Creating decorators to add API authentication:**

By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it. You can learn more about decorators and it’s usage in the links mentioned in last page:

Let’s go back and take a look at the /books route which calls get\_books(). This is how it looks:

#this would fetch all the books

#GET /books?token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJkYXRhIjoicGF5bG9hZCJ9.SePiJiKnHWTZjk0SvBPqkX9G\_wtG9JqtNPKRrky8CDI

@app.route('/books')

def get\_books():

token = request.args.get('token')

try:

jwt.decode(token, app.config['SECRET\_KEY'])

except:

return jsonify({'error': 'valid jwt token is not passed. Hence couldnot display the data'}), 401

return jsonify({'books': Book.get\_all\_books()}) #convert list into json object

A lot of code related to fetching the token and verifying is used in this function and that is more compared to the business logic written. Let’s move that out so that this token related logic can be used by other routes as well. Let’s do that using decorators as shown below:

def token\_required(f):

@wraps(f) #this will preserve the original func name as we decorate different routes

def wrapper(\*args, \*\*kwargs):

token = request.args.get('token')

try:

jwt.decode(token, app.config['SECRET\_KEY'])

return f(\*args, \*\*kwargs)

except:

return jsonify({'errror':'valid jwt token is not passed. Hence couldnot display the data'}), 401

return wrapper

we need to use the below import statement so that we can use @wraps() and pass in the function to call so that it would preserve the original function to be called even we decorate different routes.

from functools import wraps

In order to use the decorator in our get\_books() we need to update the books route as shown below:

#this would fetch all the books

#GET /books?token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.

@app.route('/books')

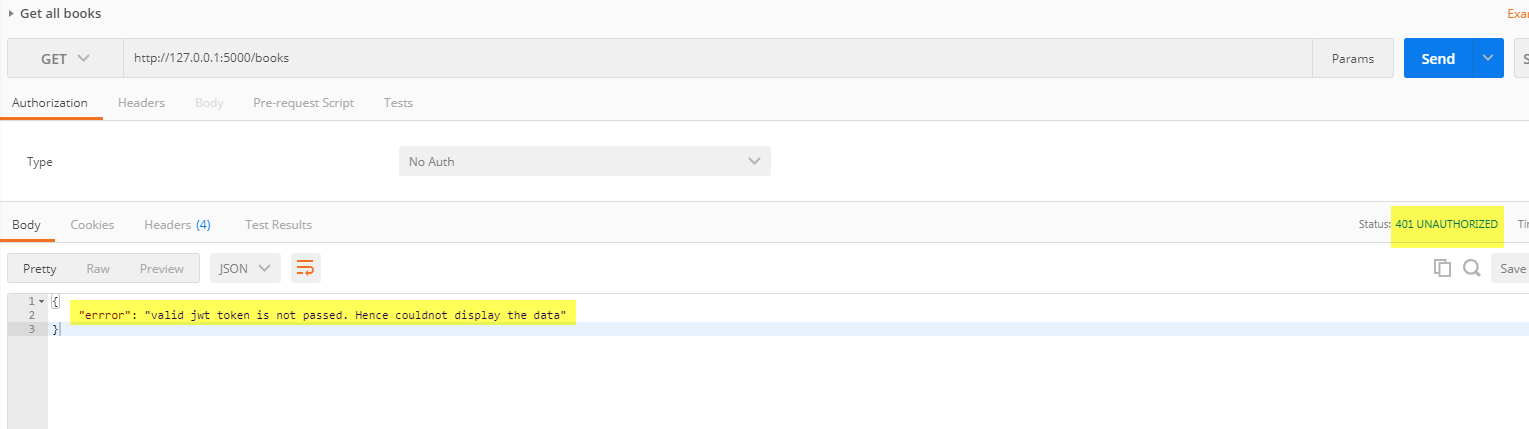
@token\_required

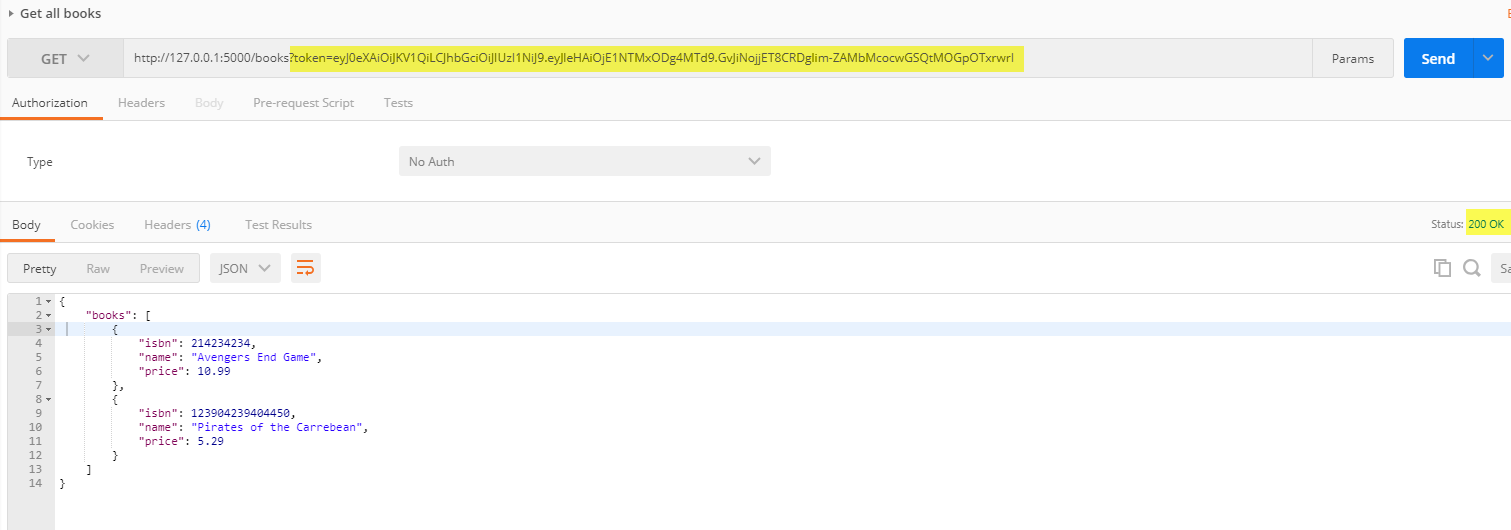
def get\_books():

return jsonify({'books': Book.get\_all\_books()}) #convert list into json object

When we call the /books route the decorator named token\_required() will be called and if it succeeds it would call the get\_books() route to display the data to the user.

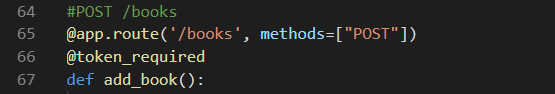
Let’s restart the app and test it to see if it works fine.

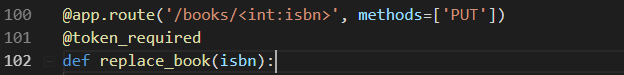


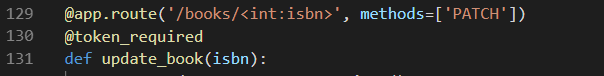


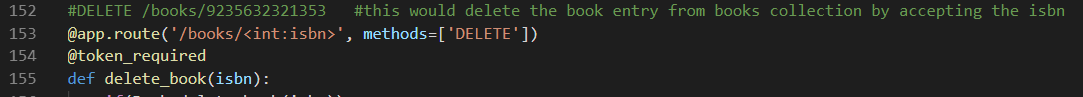
So far everything looks good, we have defined a decorator that will allow us to use an access token for accessing any part of the api in this case we have used it for getting all books.

Typically, for Rest APIs we allow read access to resources but restrict write access. We can correct this by allowing user to pass the token for POST, PUT, PATCH requests and no token required for GET requests. So let’s go ahead and update the routes for post, put and patch to use token\_required decorator and remove it for getters.











This concludes the course on building restful services using flask. Please refer to official documentation for more information.

I have placed the code used for this in the git hub repository here:

**Git hub link**

**References:**

<https://app.pluralsight.com/library/courses/python-flask-rest-api/table-of-contents>

<http://flask.pocoo.org/>

Decorators:

<https://realpython.com/primer-on-python-decorators/>

<https://www.programiz.com/python-programming/decorator>

cheat sheet: <https://static.realpython.com/decorators-cheatsheet.pdf?__s=jcgdmf7i4rmpaqkzxux4>

youtube: <https://www.youtube.com/watch?v=nYDKH9fvlBY>

usage examples: <https://github.com/realpython/materials/tree/master/primer-on-python-decorators>