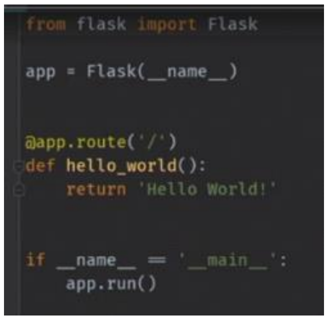
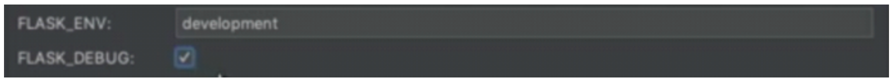
Simple API

Saturday, January 23, 2021 9:02 AM



```
from flask import Flask

app = Flask(__name__)


@app.route('/')
def hello_world():
    return 'Hello World!'


if __name__ == '__main__':
    app.run()
```

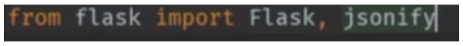
EndPoint - nothing but an URL



FLASK_ENV: development
FLASK_DEBUG: ✓

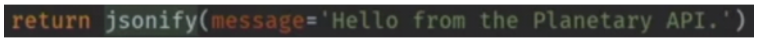
!Do in VS

- Flask debug allows the server to reload whenever there is a change in the script(Flask APP)



```
from flask import Flask, jsonify
```

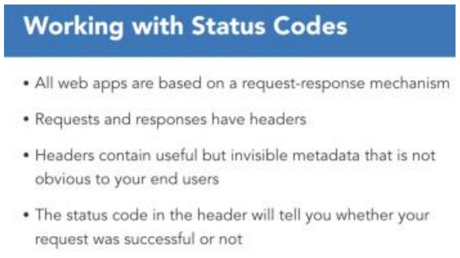
Jsonify - enables Api to return data in JSON format .



```
return jsonify(message='Hello from the Planetary API.')
```

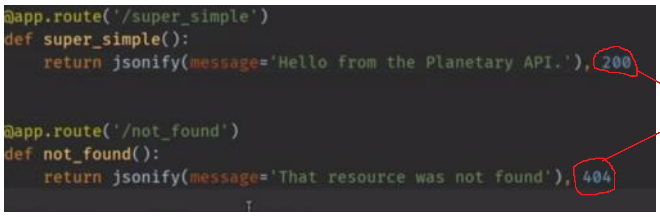
This is how we return JSON Formatted data from a api function.

HTML Status code:



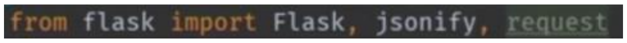
Working with Status Codes

- All web apps are based on a request-response mechanism
- Requests and responses have headers
- Headers contain useful but invisible metadata that is not obvious to your end users
- The status code in the header will tell you whether your request was successful or not



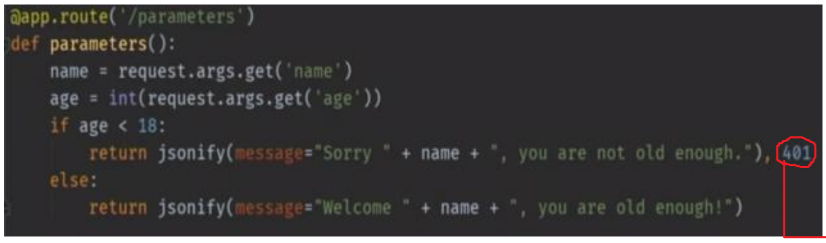
```
@app.route('/super_simple')
def super_simple():
    return jsonify(message='Hello from the Planetary API.'), 200


@app.route('/not_found')
def not_found():
    return jsonify(message='That resource was not found'), 404
```

Adding status code to our API



```
from flask import Flask, jsonify, request
```

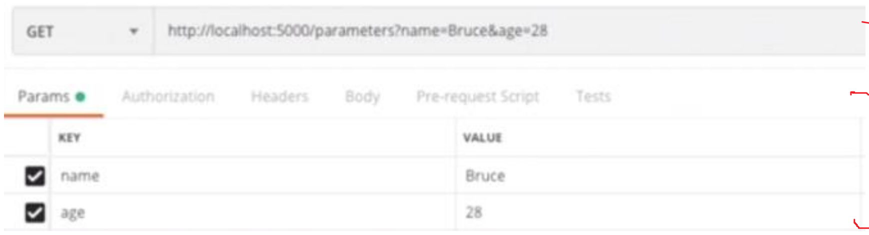
In order to work with URL parameters and other HTML request related stuffs.



```
@app.route('/parameters')
def parameters():
    name = request.args.get('name')
    age = int(request.args.get('age'))
    if age < 18:
        return jsonify(message="Sorry " + name + ", you are not old enough."), 401
    else:
        return jsonify(message="Welcome " + name + ", you are old enough!")
```

A simple function which grabs the name and age from the URL parameters by sending
requests, and in return checks and the 'age' parameter and returns the JSON data as per
in the code.

401 - Unauthorized Status code



GET http://localhost:5000/parameters?name=Bruce&age=28

	KEY	VALUE
✓	name	Bruce
✓	age	28

This is how we put parameters in HTTP request in the URL/Endpoint

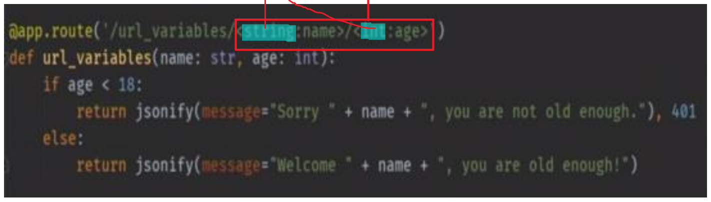
PostMan - has the feature of adding Key and Value in GUI

URL Variables and conversion filters:

Specifying var type

Variable in URL , instead of adding 'Key' and 'Val' pairs in the endpoint by the above method

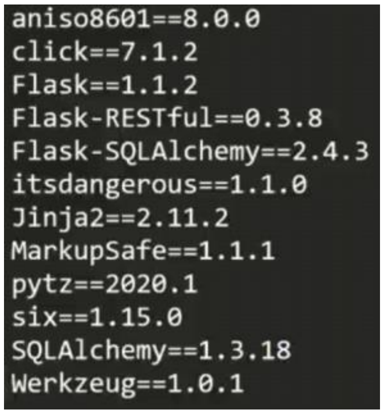
- This method is used in recent modern api's



```
@app.route('/url_variables/<string:name>/<int:age>')
def url_variables(name: str, age: int):
    if age < 18:
        return jsonify(message="Sorry " + name + ", you are not old enough."), 401
    else:
        return jsonify(message="Welcome " + name + ", you are old enough!")
```

Packages used in course (beginning):

Click, Flask, MarkupSafe, Werkzeug, itsdangerous

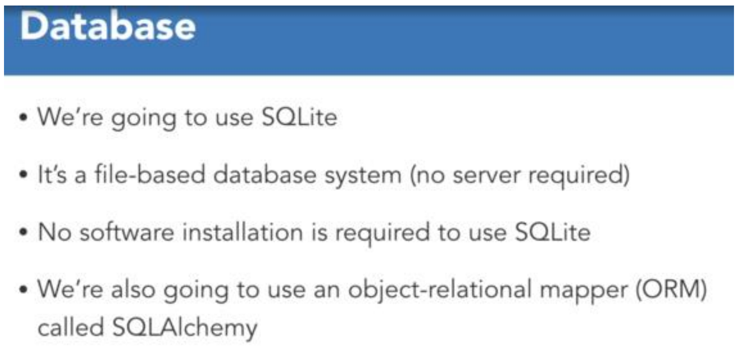


```
aniso8601==8.0.0
click==7.1.2
Flask==1.1.2
Flask-RESTful==0.3.8
Flask-SQLAlchemy==2.4.3
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
pytz==2020.1
six==1.15.0
SQLAlchemy==1.3.18
Werkzeug==1.0.1
```

==> Packages used in Tech With Tim course.

Database

Saturday, January 23, 2021 12:10 PM



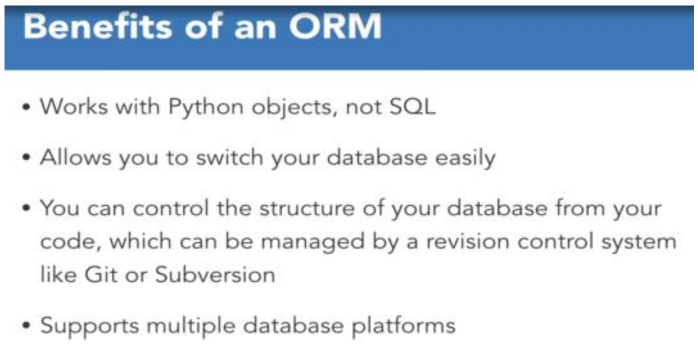
Database

- We're going to use SQLite
- It's a file-based database system (no server required)
- No software installation is required to use SQLite
- We're also going to use an object-relational mapper (ORM) called SQLAlchemy

Flask-SQLAlchemy

package for SQLAlchemy

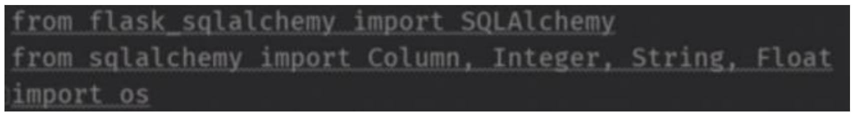
SQLite - File based DB , instead of server based one (which involved installing and
Managing a SQL server in the machine).



Benefits of an ORM

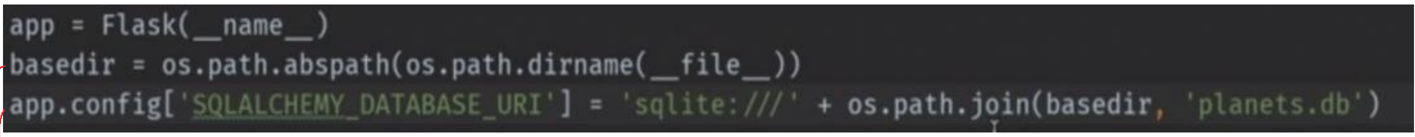
- Works with Python objects, not SQL
- Allows you to switch your database easily
- You can control the structure of your database from your code, which can be managed by a revision control system like Git or Subversion
- Supports multiple database platforms

- **Setting Up:**



```
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import Column, Integer, String, Float
import os
```

Imports - to be done to setup SQLAlchemy

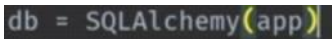


```
app = Flask(__name__)
basedir = os.path.abspath(os.path.dirname(__file__))
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + os.path.join(basedir, 'planets.db')
```

Getting current working directory

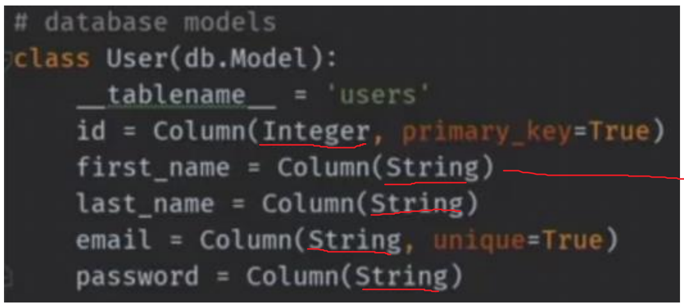
Creating/Configuring a DB file in that base/working directory (Config function in the default Flask Package is used)

- **Creating ORM class models:**



```
db = SQLAlchemy(app)
```

==> SQLAlchemy constructor



```
# database models
class User(db.Model):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    first_name = Column(String)
    last_name = Column(String)
    email = Column(String, unique=True)
    password = Column(String)
```

* These classes are called 'DB models' since these classes will be converted to SQL tables by ORM

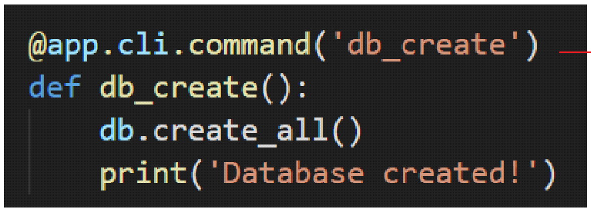
__tablename__ used to control table

These are the way to specify the type of var in the table

'unique' makes it to have unique value instead of multiple values.

Creating DB

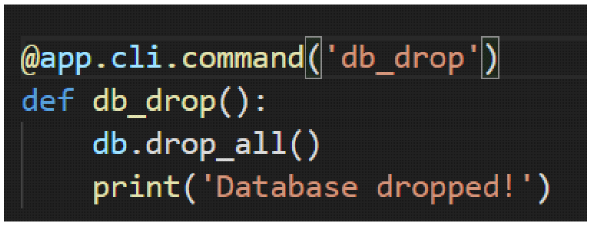
Sunday, January 24, 2021 8:08 AM



```
@app.cli.command('db_create')
def db_create():
    db.create_all()
    print('Database created!')
```

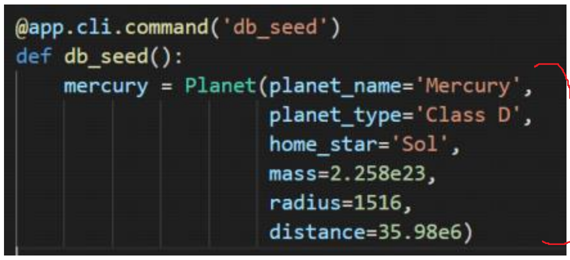
Command Line Interface feature in FLASK.

Deleting DB



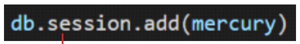
```
@app.cli.command('db_drop')
def db_drop():
    db.drop_all()
    print('Database dropped!')
```

Seeding (adding values) to DB



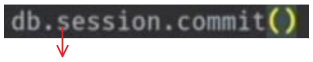
```
@app.cli.command('db_seed')
def db_seed():
    mercury = Planet(planet_name='Mercury',
                     planet_type='Class D',
                     home_star='Sol',
                     mass=2.258e23,
                     radius=1516,
                     distance=35.98e6)
```

Creating Class Objects/Instance in order to add it to the DB



```
db.session.add(mercury)
```

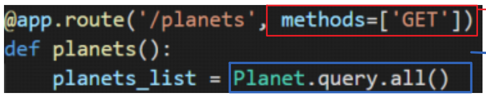
This command/line adds the mercury object to the DB table.



```
db.session.commit()
```

Final Command/line , this command commits the changes to the DB (saves the changes in DB)

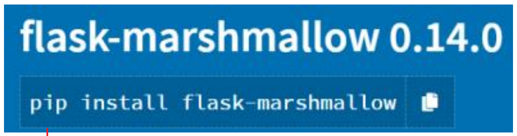
Getting Data from DB using EndPoint



```
@app.route('/planets', methods=['GET'])
def planets():
    planets_list = Planet.query.all()
```

Specifying 'methods' for the ENDPOINT.

SQLAlchemy command which Queries all data from the (Planet table in this case) from the specified table.



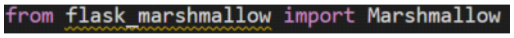
flask-marshmallow 0.14.0

```
pip install flask-marshmallow
```

Module to serialize SQLAlchemy resulted objects.

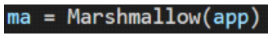
SettingUp Marshmallow

Sunday, January 24, 2021 8:47 AM



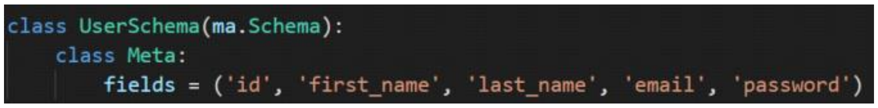
```
from flask_marshmallow import Marshmallow
```

==> Importing 'flask-marshmallow'



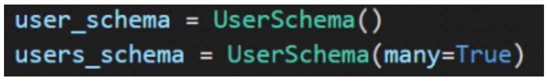
```
ma = Marshmallow(app)
```

==> Creating marshmallow instance



```
class UserSchema(ma.Schema):
    class Meta:
        fields = ('id', 'first_name', 'last_name', 'email', 'password')
```

==> Classes to create schema of objects , and creating innerclass with the table fields

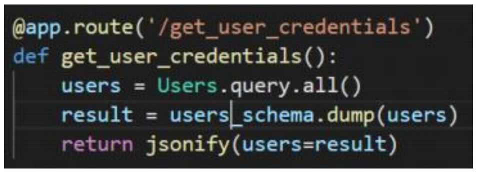


```
user_schema = UserSchema()
users_schema = UserSchema(many=True)
```

==> Instance to retrieve single data fields

==> Instance to retrieve a collection of data from the table/DB

Getting and serializing data



```
@app.route('/get_user_credentials')
def get_user_credentials():
    users = Users.query.all()
    result = users_schema.dump(users)
    return jsonify(users=result)
```

==> Gets all the data from the table

==> Serialize the returned value from the query to JSON
==> Returns the result which is JSON serialized

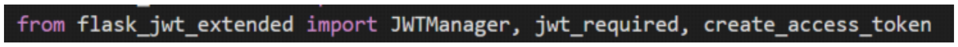
JSON web tokens

Monday, January 25, 2021 11:16 AM

Package : Flask-JWT-Extended

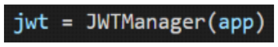
Website : jwt.io

Import :

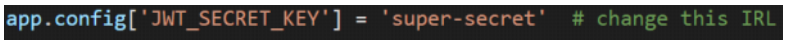


```
from flask_jwt_extended import JWTManager, jwt_required, create_access_token
```

Initializing:

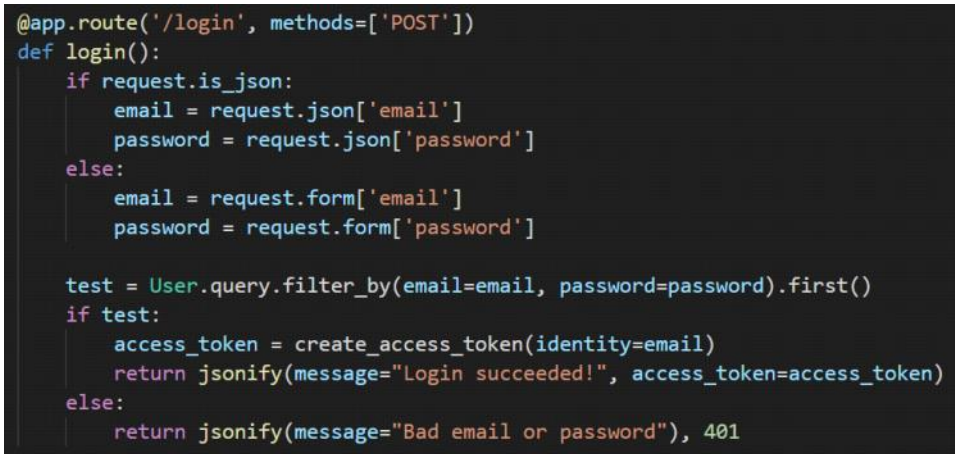


```
jwt = JWTManager(app)
```



```
app.config['JWT_SECRET_KEY'] = 'super-secret'  # change this IRL
```

Authorizing



```
@app.route('/login', methods=['POST'])
def login():
    if request.is_json:
        email = request.json['email']
        password = request.json['password']
    else:
        email = request.form['email']
        password = request.form['password']

    test = User.query.filter_by(email=email, password=password).first()
    if test:
        access_token = create_access_token(identity=email)
        return jsonify(message="Login succeeded!", access_token=access_token)
    else:
        return jsonify(message="Bad email or password"), 401
```

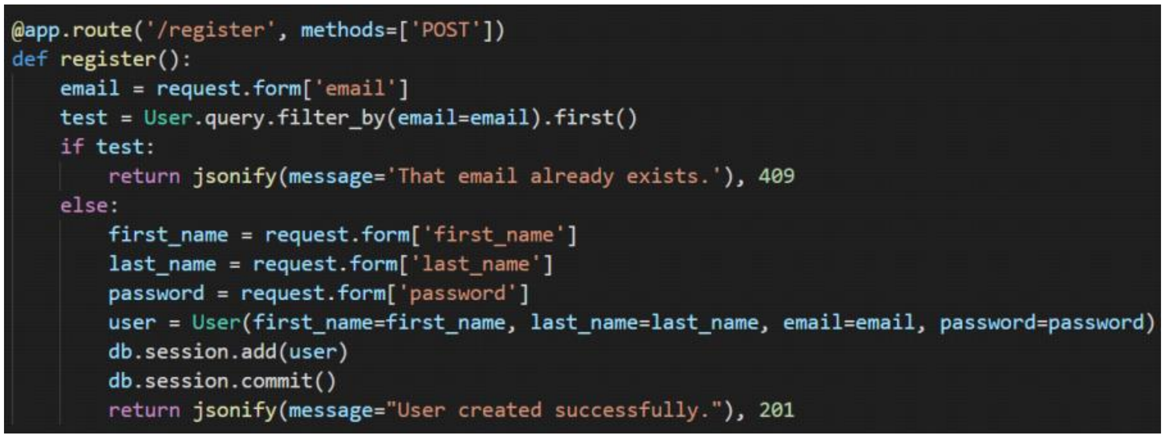
==> request POST method which posts JSON data

==> request POST method which posts HTML form data

==> Getting user data from the DB to check the user login

==> Creates access token based on the identity , this is a JWT func

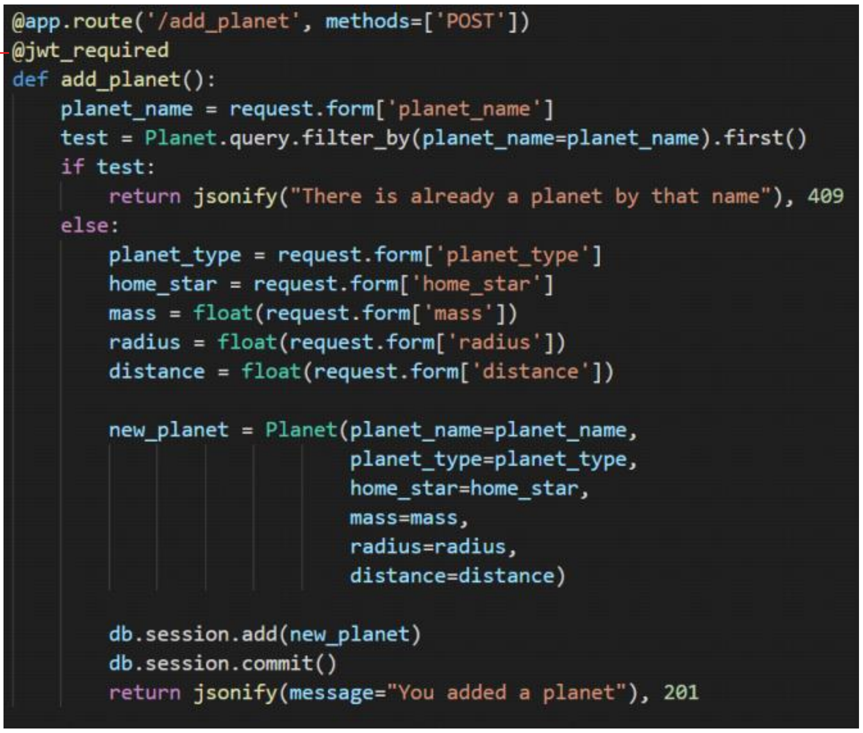
A simple function to register new users



```
@app.route('/register', methods=['POST'])
def register():
    email = request.form['email']
    test = User.query.filter_by(email=email).first()
    if test:
        return jsonify(message='That email already exists.'), 409
    else:
        first_name = request.form['first_name']
        last_name = request.form['last_name']
        password = request.form['password']
        user = User(first_name=first_name, last_name=last_name, email=email, password=password)
        db.session.add(user)
        db.session.commit()
        return jsonify(message="User created successfully."), 201
```

How to secure an EndPoint using JSON tokens

This decorator make the EndPoint secure by asking token which we created in login EndPoint

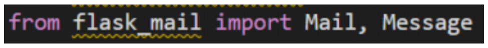


```
@app.route('/add_planet', methods=['POST'])
@jwt_required
def add_planet():
    planet_name = request.form['planet_name']
    test = Planet.query.filter_by(planet_name=planet_name).first()
    if test:
        return jsonify("There is already a planet by that name"), 409
    else:
        planet_type = request.form['planet_type']
        home_star = request.form['home_star']
        mass = float(request.form['mass'])
        radius = float(request.form['radius'])
        distance = float(request.form['distance'])

        new_planet = Planet(planet_name=planet_name,
                            planet_type=planet_type,
                            home_star=home_star,
                            mass=mass,
                            radius=radius,
                            distance=distance)

        db.session.add(new_planet)
        db.session.commit()
        return jsonify(message="You added a planet"), 201
```

Mailing Users Example

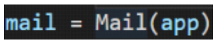
Monday, January 25, 2021 7:02 PM

Import :

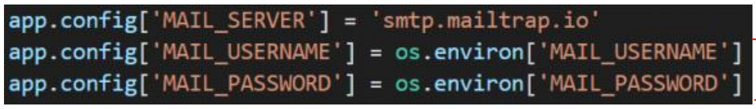


```python
from flask_mail import Mail, Message
```

Config:



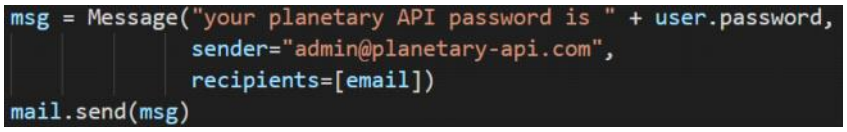
```python
mail = Mail(app)
```



```python
app.config['MAIL_SERVER'] = 'smtp.mailtrap.io'
app.config['MAIL_USERNAME'] = os.environ['MAIL_USERNAME']
app.config['MAIL_PASSWORD'] = os.environ['MAIL_PASSWORD']
```

==> URL of the mail server (in this case online server from mailstrap.io is used)

==> User credentials of the sender from env variables



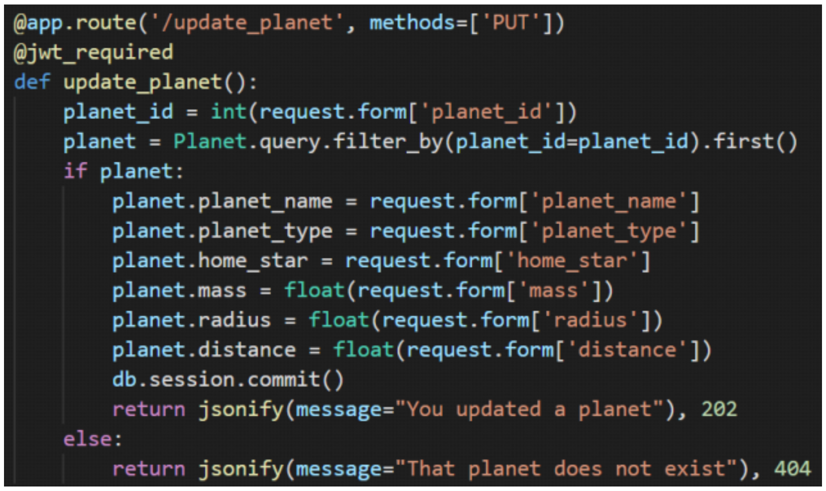
```python
msg = Message("your planetary API password is " + user.password,
              sender="admin@planetary-api.com",
              recipients=[email])
mail.send(msg)
```

==> Flask message instance , (the body of the mail is added in this instance)

==> Send instance to send the mail to the recipients.

Updating values in DB by route

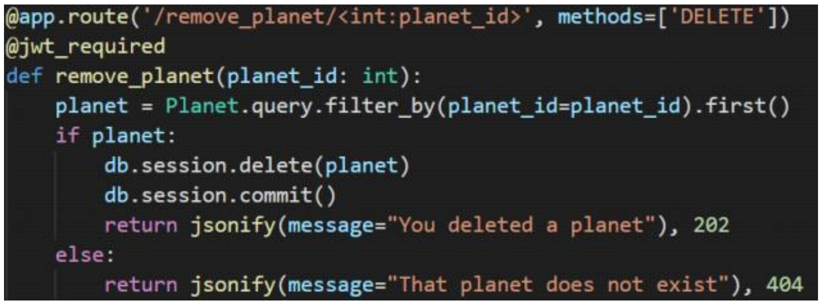
Tuesday, January 26, 2021 11:57 AM



```python
@app.route('/update_planet', methods=['PUT'])
@jwt_required
def update_planet():
    planet_id = int(request.form['planet_id'])
    planet = Planet.query.filter_by(planet_id=planet_id).first()
    if planet:
        planet.planet_name = request.form['planet_name']
        planet.planet_type = request.form['planet_type']
        planet.home_star = request.form['home_star']
        planet.mass = float(request.form['mass'])
        planet.radius = float(request.form['radius'])
        planet.distance = float(request.form['distance'])
        db.session.commit()
        return jsonify(message="You updated a planet"), 202
    else:
        return jsonify(message="That planet does not exist"), 404
```

==> Just commiting the changes will update the values in the table if the
Data exists in the table. (skipping session.add)

Deleting an object in DB

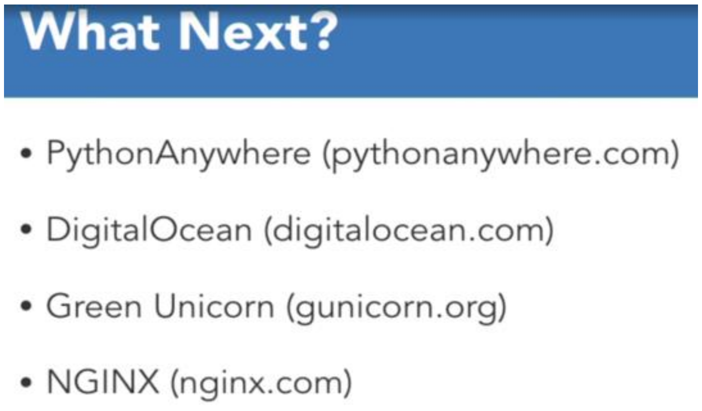


```python
@app.route('/remove_planet/<int:planet_id>', methods=['DELETE'])
@jwt_required
def remove_planet(planet_id: int):
    planet = Planet.query.filter_by(planet_id=planet_id).first()
    if planet:
        db.session.delete(planet)
        db.session.commit()
        return jsonify(message="You deleted a planet"), 202
    else:
        return jsonify(message="That planet does not exist"), 404
```

==> Deleting and commiting changes to the DB by 'delete' URL route

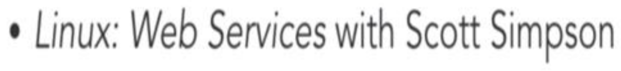
Deployment

Tuesday, January 26, 2021 12:09 PM



What Next?

- PythonAnywhere (pythonanywhere.com)
- DigitalOcean (digitalocean.com)
- Green Unicorn (gunicorn.org)
- NGINX (nginx.com)



- *Linux: Web Services* with Scott Simpson