Introduction to SQL

Assigned: Thursday, February 20, 2020 *Due 5 PM, Thursday, February 20, 2020*

Our goal today is to begin learning SQL (for Structured Query Language). We'll make our web services API from last week persistent. To do this, we'll use Sqlite for the relational database.

I chose Sqlite for this exercise because it's ubiquitous. Sqlite is the world's "Most Widely Deployed and Used Database Engine" (https://sqlite.org/mostdeployed.html). It's already on your development computer. So we can avoid installation complications and get going immediately.

Sqlite is really a *library* that allows any application to be extended with database functionality. It's mostly used as a backend data storage layer for applications.

Although Sqlite is extremely powerful it isn't appropriate for all applications, specifically those with massive datasets and many concurrent writers. For such applications *client-server* databases such as PostGres or MySQL are a better fit.

This lab is organized into two sections. The first part is a tutorial. It shows with examples how to install sqlite, connect to a database, create tables, run queries, and update the database. In the second part, the problem set, you will use the concepts from the tutorial to add a database layer to the web services API.

Setup and Installation

To check whether sqlite3 is already installed on your system, run the magic shell command below:

In [1]:
```
# determine if sqlite3 is installed
! which sqlite3
'''
#for windows
try:
    !where sqlite3
except:
    pass'''
```

```
/anaconda3/envs/ipk3/bin/sqlite3
```

Out[1]:
```
'\n#for windows\ntry:\n    !where sqlite3\nexcept:\n    pass'
```

The above command will print the location of the `sqlite3` executable. If you see some output, you're good to go. If not, follow the instructions here (https://www.sqlite.org/download.html).

We'll use the SQLAlchemy (https://www.sqlalchemy.org) library to access Sqlite (and other databases) from Python. SQLAlchemy is an abstraction and adaptation layer that runs above most relational databases. (A Sqlite-specific library for Python also exists but it does not support any other database; SQLAlchemy works with most popular SQL databases.)

SQLAlchemy provides two main ways to access databases: **core** and **ORM** (for Object Relational Mapping). We'll use the Core interface in this lab. With the Core interface, you write queries in *raw* SQL. The ORM interface is a higher-level abstraction. We'll explore the ORM interface next week.

You can install SQLAlchemy with the following commands:

In [2]:
```
# install sqlalchemy with conda
import sys

!conda install --yes --prefix {sys.prefix} sqlalchemy
```

```
Collecting package metadata (current_repodata.json): done
Solving environment: / ^C
-
```

Connect to a Sqlite Database

With SQLAlchemy installed, you can connect to a database with the `create_engine` function. An `engine` is the central entry point for communicating with a specific database.

The first argument to `create_engine` specifies the kind of database to open. Here we'll open an *in-memory* sqlite database. The database won't be stored to disk. Instead it will disappear everytime the calling process (i.e., this Jupyter notebook) exits. A non-persistent database can be useful while learning: you get a blank database everytime you run the application.

The second argument `echo=True` tells SQLAlchemy to verbosely print to the console all the SQL statements it generates.

For more about the `create_engine` function see here (https://docs.sqlalchemy.org/en/latest/core/engines.html#sqlite).

In [3]:
```
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import inspect
```

In [4]:
```
# Type of database to connect to. The following specifies an in-memory s
qlite.
db_name = 'sqlite:///:memory:'

# use a URL like the following to open (and create if necessary)
# a file-backed sqlite database:

# db_name = 'sqlite:///my_sqlite_db.sqlite3'
# the above will create a database relative to current working director
y. See
# the documentation for how to create a database in a different locatio
n.


# create an engine
engine = create_engine(db_name, echo=True)
print(sqlalchemy.__version__)
print(engine)
```

```
1.3.13
Engine(sqlite:///:memory:)
```

Create Your First Table

At this point you'll have an empty database created in memory (not persistent storage). To do anything useful with a relational database, you have to first create a table (or tables) and insert some rows into those tables.

Creating a table defines the names and types of the attributes (columns) of the database.

We'll create a simple table called `cities`. It will have the following columns:

```
{
    "id": 'the primary key for the table',
    "name": 'the city name, a text string',
    "lat": 'latitude, a floating point number',
    "lng": 'longitude, a floating point number',
    "country": 'country, a text string',
    "population": 'city population, an integer'
}
```

To create the `cities` table, use the `create table` SQL statement as shown below. The `engine.execute()` method is used to send the raw SQL to the connected database.

In [5]:
```
# drop a table cities in case it existed already
drop_table_statement = """drop table if exists cities"""
engine.execute(drop_table_statement)

# sql statement
create_table_stmt = """create table cities(
  id integer primary key,
  name text not null,
  lat float not null,
  lng float not null,
  state text not null,
  country text not null,
  population integer not null
);
"""
engine.execute(create_table_stmt)
```

```
2020-02-27 09:45:12,507 INFO sqlalchemy.engine.base.Engine SELECT CAST
('test plain returns' AS VARCHAR(60)) AS anon_1
2020-02-27 09:45:12,513 INFO sqlalchemy.engine.base.Engine ()
2020-02-27 09:45:12,529 INFO sqlalchemy.engine.base.Engine SELECT CAST
('test unicode returns' AS VARCHAR(60)) AS anon_1
2020-02-27 09:45:12,537 INFO sqlalchemy.engine.base.Engine ()
2020-02-27 09:45:12,550 INFO sqlalchemy.engine.base.Engine drop table i
f exists cities
2020-02-27 09:45:12,559 INFO sqlalchemy.engine.base.Engine ()
2020-02-27 09:45:12,566 INFO sqlalchemy.engine.base.Engine COMMIT
2020-02-27 09:45:12,573 INFO sqlalchemy.engine.base.Engine create table
cities(
  id integer primary key,
  name text not null,
  lat float not null,
  lng float not null,
  state text not null,
  country text not null,
  population integer not null
);

2020-02-27 09:45:12,579 INFO sqlalchemy.engine.base.Engine ()
2020-02-27 09:45:12,588 INFO sqlalchemy.engine.base.Engine COMMIT
```

Out[5]:
```
<sqlalchemy.engine.result.ResultProxy at 0x10b7cf460>
```

You now have an empty database created in memory (not persistent storage). The following code block shows how to retrieve information about the `users` table you just created:

In [6]:
```
engine.table_names()
```

```
2020-02-27 09:45:14,936 INFO sqlalchemy.engine.base.Engine SELECT name
FROM sqlite_master WHERE type='table' ORDER BY name
2020-02-27 09:45:14,942 INFO sqlalchemy.engine.base.Engine ()
```

Out[6]:
```
['cities']
```

In [7]:
```
# inspect the database
inspector = inspect(engine)

# Get table information
print(inspector.get_table_names())

# Get column information
for col in inspector.get_columns('cities'):
    print(col)
```

```
2020-02-27 09:45:16,380 INFO sqlalchemy.engine.base.Engine SELECT name
FROM sqlite_master WHERE type='table' ORDER BY name
2020-02-27 09:45:16,390 INFO sqlalchemy.engine.base.Engine ()
['cities']
2020-02-27 09:45:16,396 INFO sqlalchemy.engine.base.Engine PRAGMA main.
table_info("cities")
2020-02-27 09:45:16,400 INFO sqlalchemy.engine.base.Engine ()
{'name': 'id', 'type': INTEGER(), 'nullable': True, 'default': None, 'a
utoincrement': 'auto', 'primary_key': 1}
{'name': 'name', 'type': TEXT(), 'nullable': False, 'default': None, 'a
utoincrement': 'auto', 'primary_key': 0}
{'name': 'lat', 'type': FLOAT(), 'nullable': False, 'default': None, 'a
utoincrement': 'auto', 'primary_key': 0}
{'name': 'lng', 'type': FLOAT(), 'nullable': False, 'default': None, 'a
utoincrement': 'auto', 'primary_key': 0}
{'name': 'state', 'type': TEXT(), 'nullable': False, 'default': None,
'autoincrement': 'auto', 'primary_key': 0}
{'name': 'country', 'type': TEXT(), 'nullable': False, 'default': None,
'autoincrement': 'auto', 'primary_key': 0}
{'name': 'population', 'type': INTEGER(), 'nullable': False, 'default':
None, 'autoincrement': 'auto', 'primary_key': 0}
```

Insert City Data Into the Database

Next, we will use a SQL `insert` statement to add some cities to the database. (The data for this example came from SimpleMaps (https://simplemaps.com/data/us-cities).)

Note the use of `?` placeholders in the `insert` statement. These are positional parameters that are filled in with actual values at runtime.

In [8]:

```
cities = [
    ("Ammon", 43.4748, -111.9559, "Idaho", "USA", 15252),
    ("Idaho Falls", 43.4878, -112.0359, "Idaho", "USA", 96166),
    ("Iona", 43.5252, -111.931, "Idaho", "USA", 2213),
    ("Island Park",     44.5251, -111.3581, "Idaho", "USA", 272),
    ("Ririe", 43.6326, -111.7716,"Idaho", "USA",643),
    ("Sugar City", 43.8757,     -111.7518, "Idaho" ,"USA", 1361),
    ("Teton", 43.8872, -111.6726, "Idaho", "USA", 714),
]

insert_statement = """
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)
"""

for c in cities:
    print(f"inserting {c[0]}")
    # insert into db; note unpacking of tuple (*c)
    engine.execute(insert_statement, *c)
```

```
inserting Ammon
2020-02-27 09:45:23,935 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2020-02-27 09:45:23,940 INFO sqlalchemy.engine.base.Engine ('Ammon', 4
3.4748, -111.9559, 'Idaho', 'USA', 15252)
2020-02-27 09:45:23,946 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Idaho Falls
2020-02-27 09:45:23,952 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2020-02-27 09:45:23,955 INFO sqlalchemy.engine.base.Engine ('Idaho Fall
s', 43.4878, -112.0359, 'Idaho', 'USA', 96166)
2020-02-27 09:45:23,958 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Iona
2020-02-27 09:45:23,966 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2020-02-27 09:45:23,969 INFO sqlalchemy.engine.base.Engine ('Iona', 43.
5252, -111.931, 'Idaho', 'USA', 2213)
2020-02-27 09:45:23,974 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Island Park
2020-02-27 09:45:23,978 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2020-02-27 09:45:23,981 INFO sqlalchemy.engine.base.Engine ('Island Par
k', 44.5251, -111.3581, 'Idaho', 'USA', 272)
2020-02-27 09:45:23,989 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Ririe
2020-02-27 09:45:24,006 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2020-02-27 09:45:24,011 INFO sqlalchemy.engine.base.Engine ('Ririe', 4
3.6326, -111.7716, 'Idaho', 'USA', 643)
2020-02-27 09:45:24,016 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Sugar City
2020-02-27 09:45:24,023 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2020-02-27 09:45:24,031 INFO sqlalchemy.engine.base.Engine ('Sugar Cit
y', 43.8757, -111.7518, 'Idaho', 'USA', 1361)
2020-02-27 09:45:24,041 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Teton
2020-02-27 09:45:24,046 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2020-02-27 09:45:24,052 INFO sqlalchemy.engine.base.Engine ('Teton', 4
3.8872, -111.6726, 'Idaho', 'USA', 714)
2020-02-27 09:45:24,057 INFO sqlalchemy.engine.base.Engine COMMIT
```

Retrieve Selected Rows

We can now retrieve a subset of the cities with the SQL `select` statement. Here we use a `where` retrieve cities with a `population` less than `1000`.

In the second `select` We also use a `count` function to compute the number cities with a population greater than `1000`.

In [10]:
```
print(engine.table_names())

c = engine.execute('select * from cities where population < 1000')

for row in c:
    print(dict(row))

cnt = engine.execute('select count(name) from cities where population >
 10000')
for row in cnt:
    print(dict(row))
```

```
2020-02-27 09:45:35,039 INFO sqlalchemy.engine.base.Engine SELECT name
FROM sqlite_master WHERE type='table' ORDER BY name
2020-02-27 09:45:35,043 INFO sqlalchemy.engine.base.Engine ()
['cities']
2020-02-27 09:45:35,049 INFO sqlalchemy.engine.base.Engine select * fro
m cities where population < 1000
2020-02-27 09:45:35,056 INFO sqlalchemy.engine.base.Engine ()
{'id': 4, 'name': 'Island Park', 'lat': 44.5251, 'lng': -111.3581, 'sta
te': 'Idaho', 'country': 'USA', 'population': 272}
{'id': 5, 'name': 'Ririe', 'lat': 43.6326, 'lng': -111.7716, 'state':
'Idaho', 'country': 'USA', 'population': 643}
{'id': 7, 'name': 'Teton', 'lat': 43.8872, 'lng': -111.6726, 'state':
'Idaho', 'country': 'USA', 'population': 714}
2020-02-27 09:45:35,066 INFO sqlalchemy.engine.base.Engine select count
(name) from cities where population > 10000
2020-02-27 09:45:35,071 INFO sqlalchemy.engine.base.Engine ()
{'count(name)': 2}
```

Update Selected Rows

In this example we shall change the 'State' from 'Idaho' to 'ID' for all cities with a `population` greater than `10000`. To do this we will utilize a SQL `update` command:

In [11]:
```
update_statement = """
update cities
set state = ?
where population > ?
"""

engine.execute(update_statement, 'ID', 10000) #state is ID

# read updated rows to see that the state attribute was changed
cs = engine.execute('select id, population, state from cities where popu
lation > 10000')

# print out each row
for row in cs:
    print(row)
```

```
2020-02-27 09:45:37,337 INFO sqlalchemy.engine.base.Engine
update cities
set state = ?
where population > ?

2020-02-27 09:45:37,341 INFO sqlalchemy.engine.base.Engine ('ID', 1000
0)
2020-02-27 09:45:37,348 INFO sqlalchemy.engine.base.Engine COMMIT
2020-02-27 09:45:37,354 INFO sqlalchemy.engine.base.Engine select id, p
opulation, state from cities where population > 10000
2020-02-27 09:45:37,357 INFO sqlalchemy.engine.base.Engine ()
(1, 15252, 'ID')
(2, 96166, 'ID')
```

Problem Set

50 Points

The above tutorial should have provided you with enough background to get started with the homework, which is to migrate your web app `users` resource to sqlite. The problems below are identical to those from last week except you will be retrieving, creating, and updating a database table instead of using an in-memory list of users.

Setup: Migrate Web Service Users to Sqlite

Before proceeding to the problem set, update `api.py` to initialize and connect to the database. Please follow these steps:

1. Install Flask-SQLAlchemy (http://flask-sqlalchemy.pocoo.org/2.3/#), a library that simplifies access to SQLAlchemy from within Flask:

    ```
    conda install Flask-SQLAlchemy
    ```

2. Import `flask_sqlalchemy` into `api.py`.

    ```
    from flask_sqlalchemy import SQLAlchemy
    ```

3. Update `flask` startup code by replacing it with this:

```
def init_db():

        # create a global variable __db__ that you can use from route handle
rs
        global __db__

        # use in-memory database for debugging
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///:memory:'

        # app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db.sqlite'

        __db__ = SQLAlchemy(app)
        engine = __db__.engine

        # put your database initialization statements here
```

Problem 1: List Users

10 Points

Modify `api.py` to retrieve the collection of users. Essentially, you will convert your existing handler that returns data from the variable `USERS` to read from the database.

Run the test below to show that your code is correct.

In [49]:
```
import unittest
import requests
import json

# The base URL for all HTTP requests
BASE = 'http://localhost:5000/users'

# set Content-Type to application/json for all HTTP requests
headers={'Content-Type': 'application/json'}

class Problem1Test(unittest.TestCase):

    # test
    def test_users_get_collection(self):
        r = requests.get(BASE, headers = headers)
        self.assertEqual(r.status_code, 200)

        j = r.json()
        self.assertEqual(type(j), list)
        self.assertGreater(len(j), 0)

        # extract the first element of the list
        first = j[0]

        # check all attributes exist
        self.assertIn('id', first)
        self.assertIn('first', first)
        self.assertIn('last', first)
        self.assertIn('email', first)



# Run the unit tests
unittest.main(defaultTest="Problem1Test", argv=['ignored', '-v'], exit=F
alse)
```

```
test_users_get_collection (__main__.Problem1Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.066s

OK
```

Out[49]:
```
<unittest.main.TestProgram at 0x10cb39f40>
```

Problem 2: Retrieve a Single User

10 Points

Modify the method `GET /users/{id}` to retrieve a specific user by ID to use the database instead of the `USERS` list.

This method shall return an HTTP status code of `200` on success and `404` (not found) if the user with the specified ID does not exist. See the unit tests below.

In [60]:
```
class Problem2Test(unittest.TestCase):

    def test_users_get_member(self):

        r = requests.get(BASE + '/0')
        self.assertEqual(r.status_code, 200)
        print(r.headers)
        j = r.json()

        self.assertIs(type(j), dict)
        self.assertEqual(j['id'], 0)
        self.assertIn('first', j)
        self.assertIn('last', j)
        self.assertIn('email', j)

    def test_users_wont_get_nonexistent_member(self):

        r = requests.get(BASE + '/1000')
        self.assertEqual(r.status_code, 404)

# Run the unit tests
unittest.main(defaultTest="Problem2Test", argv=['ignored', '-v'], exit=F
alse)
```

```
test_users_get_member (__main__.Problem2Test) ... ok
test_users_wont_get_nonexistent_member (__main__.Problem2Test) ...

{'Content-Type': 'application/json', 'Content-Length': '122', 'Server':
'Werkzeug/1.0.0 Python/3.7.3', 'Date': 'Thu, 27 Feb 2020 18:40:47 GMT'}

ok

----------------------------------------------------------------------
Ran 2 tests in 0.082s

OK
```

Out[60]:
```
<unittest.main.TestProgram at 0x10cb31160>
```

Problem 3: Create a User

10 Points

Modify the `POST /users` method to save the user to the database.

All of these parameters are required and your code should enforce this. If validation succeeds, add the new user to the `USERS` list and give it a unique ID.

Return HTTP status code `201` (created) if the operation succeeds and `422` (Unprocessable Entity) if validation fails.

The created user will be returned as JSON if the operation succeeds.

In [65]:
```
class Problem3Test(unittest.TestCase):


    def test_users_create(self):
        data = json.dumps({'first': 'Sammy', 'last': 'Davis', 'email':
'sammy@cuny.edu'})

        r = requests.post(BASE, headers = headers, data = data)
        self.assertEqual(r.status_code, 201)

    def test_wont_create_user_without_first_name(self):
        # simple validation (missing parameters)
        data = json.dumps({'last': 'Davis', 'email': 'sammy@cuny.edu'})

        r = requests.post(BASE, headers = headers, data = data)
        self.assertEqual(r.status_code, 422)


# Run the unit tests
unittest.main(defaultTest="Problem3Test", argv=['ignored', '-v'], exit=F
alse)
```

```
test_users_create (__main__.Problem3Test) ... ok
test_wont_create_user_without_first_name (__main__.Problem3Test) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.046s

OK
```

Out[65]:
```
<unittest.main.TestProgram at 0x10cb830a0>
```

Problem 4: Update a User

10 Points

Change the method that handles user updates (`PATCH/PUT /users/<id>`) so that it writes the update to the database.

In [83]:
```
class Problem4Test(unittest.TestCase):

    def test_users_update_member(self):
        data = json.dumps({'first': 'testing'})
        r = requests.patch(BASE + '/0', headers = headers, data = data)
        self.assertEqual(r.status_code, 200)

        j = r.json()
        self.assertIs(type(j), dict)
        self.assertEqual(j['id'], 0)
        self.assertEqual(j['first'], 'testing')

        # now retrieve the same object to ensure that it was really upda
ted
        r = requests.get(BASE + '/0', headers = headers, data = data)
        self.assertEqual(r.status_code, 200)

        j = r.json()
        self.assertEqual(j['first'], 'testing')


    def test_users_update_member_not_found(self):
        data = json.dumps({'first': 'testing'})
        r = requests.patch(BASE + '/1000', headers = headers, data = dat
a)
        self.assertEqual(r.status_code, 404)



# Run the unit tests
unittest.main(defaultTest="Problem4Test", argv=['ignored', '-v'], exit=F
alse)
```

```
test_users_update_member (__main__.Problem4Test) ... ok
test_users_update_member_not_found (__main__.Problem4Test) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.062s

OK
```

Out[83]:
```
<unittest.main.TestProgram at 0x10cc055e0>
```

Problem 5: Deactivate a User

Modify the handler for `POST /users/<id>/deactivate` so that it persists the deactivation to the database.

In [84]:
```
class Problem5Test(unittest.TestCase):

    def test_users_deactivate_member(self):

        r = requests.post(BASE + '/0/deactivate', headers = headers)
        self.assertEqual(r.status_code, 200)

        j = r.json()
        self.assertIs(type(j), dict)
        self.assertEqual(j['active'], False)


# Run the unit tests
unittest.main(defaultTest="Problem5Test", argv=['ignored', '-v'], exit=F
alse)
```

```
test_users_deactivate_member (__main__.Problem5Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.026s

OK
```

Out[84]:
```
<unittest.main.TestProgram at 0x10cbde400>
```

In [ ]:

```
import os
from flask import Flask, request, Response, jsonify, abort
from functools import wraps
from flask_sqlalchemy import SQLAlchemy

# dummy users

USERS = [
    (0, 'Joe','Bloggs',
        'joe@bloggs.com', 'student',  True),
    (1, 'Ben', 'Bitdiddle',
        'ben@cuny.edu', 'student',  True),
    (2, 'Alissa P','Hacker',
         'missalissa@cuny.edu', 'professor',  True)
]
'''
USERS = {
    {'id': 0, 'first': 'Joe', 'last': 'Bloggs',
        'email': 'joe@bloggs.com', 'role': 'student', 'active': True},
    {'id': 1, 'first': 'Ben', 'last': 'Bitdiddle',
        'email': 'ben@cuny.edu', 'role': 'student', 'active': True},
    {'id': 2, 'first': 'Alissa P', 'last': 'Hacker',
        'email': 'missalissa@cuny.edu', 'role': 'professor', 'active': T
rue},
}'''

# Custom error handler. Raise this exception
# to return a status_code, message, and body
class InvalidUsage(Exception):
    status_code = 400

    def __init__(self, message, status_code=None, payload=None):
        Exception.__init__(self)
        self.message = message
        if status_code is not None:
            self.status_code = status_code
        self.payload = payload

    def to_dict(self):
        rv = dict(self.payload or ())
        rv['message'] = self.message
        return rv


print(__name__)

app = Flask(__name__)

# set the default error handler
@app.errorhandler(InvalidUsage)
def handle_invalid_usage(error):
    response = jsonify(error.to_dict())
    response.status_code = error.status_code
    return response

def init_db():
```

```
    # create a global variable __db__ that you can use from route handle
rs
    global __db__

    # use in-memory database for debugging
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///:memory:'

    # app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db.sqlite'
    __db__ = SQLAlchemy(app)
    engine = __db__.engine

    # put your database initialization statements here
    create_table_stmt = """create table MyDataBase(
      id integer primary key,
      first text not null,
      last text not null,
      email text not null,
      role text ,
      active boolean
    );
    """
    # create the users table
    engine.execute(create_table_stmt)

    # insert each item from USERS list into the users table

    insert_statement = """
    insert into MyDataBase ('id', 'first', 'last',
        'email', 'role', 'active')
      values(?, ?, ?, ?, ?, ?)
    """

    for u in USERS:
        print(f"inserting {u}")
        # insert into db; note unpacking of tuple (*u)
        engine.execute(insert_statement, *u)


if __name__ == 'api':
    # save database handle in module-level global
    init_db()
    app.run(debug=True)


# Problem 1
@app.route("/users", methods=["GET"])
def get_users():
    u = __db__.engine.execute('select * from MyDataBase')
    users = [dict(user) for user in u]
    return jsonify(users)


# Problem 2
@app.route("/users/<int:post_id>", methods=["GET"])
def users_get_member(post_id):
    u = __db__.engine.execute(f'select * from MyDataBase WHERE id = {pos
t_id}')
```

```
    target = u.fetchone()
    if target: #Check for whether target is empty if have invalid id.
        return jsonify(dict(target))
    return Response(None,404)

# Your code here...
# E.g.,
# @app.route("/users", methods=["GET"])

# Problem 3
@app.route("/users", methods=["POST"])
def add_user():
    new_user = request.get_json() #input from client to server
    myArray = [] #use list-array since each row is a tuple
    if 'first' in new_user.keys():
        myArray.append(new_user['first'])
        myArray.append(new_user['last'])
        myArray.append(new_user['email'])

        add_statement = """
            insert into MyDataBase ('first', 'last', 'email')
             values(?, ?, ?)
        """

        __db__.engine.execute(add_statement,myArray)

        response = Response(None, 201)
        return response

    return Response(None, 422)

# Problem 4
@app.route("/users/<int:post_id>", methods=["PUT","PATCH"])
def update_user(post_id):# client provides post_id

    myDict = request.get_json()#returns dictionary

#     update_statement = f"""
#        UPDATE MyDataBase
#        SET first = {myDict['first']}, last = {myDict['last']}, email =
{myDict['email']},
#        role = {myDict['role']}, active = {myDict['active']}
#     """
    update_statement = f"""
       UPDATE MyDataBase
       SET first = ?
       WHERE id = {post_id}
    """
    engineRP = __db__.engine.execute(update_statement,myDict['first'])#T
HIS IS THE DISPATCHER TO THE SERVER
    u = __db__.engine.execute(f'select * from MyDataBase WHERE id = {pos
t_id}')
    target = u.fetchone()
    if target: #Check for whether target is empty if have invalid id.
        return jsonify(dict(target))
    return Response(None,404)
```

```
    #response = Response(None, 200)
    #return jsonify(dict(engineRP.fetchone())) #returns to client
# Problem 5

@app.route("/users/<int:post_id>/deactivate", methods=["POST"])
def deactivate_user(post_id):

    update_statement = f"""
       UPDATE MyDataBase
       SET active = ?
       WHERE id = {post_id}
    """
    engineRP = __db__.engine.execute(update_statement,[False])#THIS IS T
HE DISPATCHER TO THE SERVER
    u = __db__.engine.execute(f'select * from MyDataBase WHERE id = {pos
t_id}')
    target = u.fetchone()
    if target: #Check for whether target is empty if have invalid id.
        return jsonify(dict(target))
    return Response(None,404)



'''


@app.route("/users", methods=["GET"])
def get_users():
    return jsonify(USERS)


# Problem 2
@app.route("/users/<int:post_id>", methods=["GET"])
def profile(post_id):
    for user in USERS:
        if post_id == user['id']:
            return jsonify(user)
    return Response(None,404)
# Problem 3
@app.route("/users", methods=["POST"])
def add_user():
    new_user = request.get_json()
    myDict = {}
    if 'first' in new_user.keys():
        myDict['first'] = new_user['first']
        myDict['last'] = new_user['last']
        myDict['email'] = new_user['email']

        #user role true
        #myDict['role'] = new_user['role']
        #myDict['status'] = new_user['status']

        #new user TRUE
        myDict['id'] = len(USERS)

        #response
        USERS.append(myDict)
```

```
        response = Response(None, 201)

        return response

    else:
        response = Response(None, 422)
        return response

# Problem 4
@app.route("/users/<int:post_id>", methods=["PUT","PATCH"])
def update_user(post_id):
    for dictionary in USERS:
        if post_id == dictionary['id']:
            user = request.get_json()
            dictionary['first'] = user['first']
            response = Response(None, 200)
            return jsonify(dictionary)

    response = Response(None, 404)
    return response

# Problem 5

@app.route("/users/<int:post_id>/deactivate", methods=["POST"])
def deactivate_user(post_id):
    for dictionary in USERS:
        if post_id == dictionary['id']:
            dictionary['active']=False
            response = Response(None, 200)
            return jsonify(dictionary)
    response = Response(None, 404)
    return response
'''
```