

CSCI 6220

Distributed Operating Systems

Homework 4

Pierce A. Lovesee

April 18th, 2022

Table of Contents

Introduction.....	3
Implementation.....	4
Python3 Virtual Environment.....	4
Flask and Flask-SQLAlchemy.....	4
Database Implementation.....	6
Implementations of GET Methods.....	6
Implementation of POST Method.....	7
Implementation of DELETE Method.....	8
Implementation of PUT Method.....	9
Application and Testing.....	10
Setup and Introduction.....	10
POST Invocations to Database.....	10
POST of “Arrays” Entity to Database.....	10
POST of “Queues” Entity to Database.....	11
POST of “Stacks” Entity to Database.....	11
Initial GET Invocation on Database to Spot-check Entries.....	12
GET Method for Selecting a Single and Specific ID.....	13
ID GET on ID: 1.....	13
ID GET on ID: 2.....	14
ID GET on ID: 3.....	15
PUT Method Invoked on Database.....	15
PUT Invocation to Change “Arrays” to “Welcome”.....	16
PUT Invocation to Change “Queues” to “to”.....	17
PUT Invocation to Change “Stacks” to “CSCI 6220”.....	18
DELETE Method Invoked on Database.....	19
Conclusion.....	20
Additional Screenshots.....	21
Work Cited.....	22

CSCI 6220

Homework 4 – REST API

Introduction

From its inception and standardization in 2002, Representational State Transfer Application Programming Interface, commonly known as REST API, has grown into an ubiquitous programming interface allowing for end-point access between virtually any combination of identical or heterogeneous systems ^[1]. The realized and potential uses for this interface are virtually limitless. This versatility is achieved through exposing the API from any application server in a secure, homogeneous, and stateless way to any subject client, be it a select group of clients or an open interface available to any client with the proper resources to access it ^[1]. Four of the most common HTTP Request Methods used in the API space are GET, POST, PUT, and DELETE ^[1]. These four methods have been implemented in this project to allow access from any given external client to perform actions on a simple, none hosted, database .

For this project, it was chosen to implement an example of a REST interface using the following technologies with the associated uses:

- **Python3** – Used in creating the end-points and associated behaviors for the the REST API as well as the structure and code base for the server
- **Python3 Virtual Environment** – Used to encapsulate the server instance, database, and handle all associated dependencies
- **Flash and Flask-SQLAlchemy** – Used to create the server, database and provide the interface between the server instance and database
- **Postman API Platform** – The chosen client application to allow POST, PUT, and DELETE HTTP methods to be invoked on server, thereby altering the database
- **JSON** – Used instead on XML for data being sent between the server and client; this was done to streamline the implementation as this is the default output with the above technologies

The design for the database was based on [Guru99's RESTful Web Services Tutorial](#) to help adhere to a similar end-product with the rest of the course section ^[1]. Several resources were referenced in creating the Python3 implementation with Flask and Flask-SQLAlchemy, all of which are referenced in the Work Cited and Reference Material section ^{[2] [3] [4] [5]}.

Implementation

Python3 Virtual Environment

In order to simplify the implementation, ensure observance of all dependencies for both the server and database, and accurately document dependencies for future reference, the project was run within the Python3 Virtual Environment (python venv). The environment was created in the present working directory and set as active.

```
pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ python3 -m venv .venv
pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ source .venv/bin/activate
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ pip3 install flask
```

Flask and Flask-SQLAlchemy

Flask and Flask-SQLAlchemy were installed within the Virtual Environment. These technologies were used to create and run the server instance as well as create and manage the database (Flask-SQLAlchemy).

```
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ pip3 install flask
```

```
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ pip3 install flask-sqlalchemy
```

With these two software packages being the two main dependencies for the implementation, a *requirements.txt* file was created for future reference with the Python code base.

```
Successfully installed SQLAlchemy-1.4.35 flask-sqlalchemy-2.5.1 greenlet-1.1.2
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ pip3 freeze > requirements.txt
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/
```

Flask Environment Variables, *FLASK_APP* and *FLASK_ENV*, were set within the Python Virtual Environment for streamlined testing of the implementation.

```
api/Tutorials$ pip3 freeze > requirements.txt
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ export FLASK_APP=application.py
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ export FLASK_ENV=development
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorials$ flask run
```

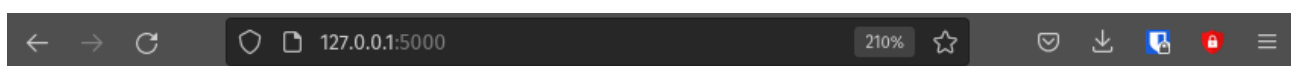
After importing the appropriate packages from Flask, a simple *app.route* was set within the Python source code to allow for testing before implementation. The *app.route*, was simply set for the root of the server to display a welcome message.

```
@app.route('/')  
def index():  
    return 'Hello - Welcome to the RESTful API tutorials'
```

The server was then ready to start and sanity test. The server address was set as <http://127.0.0.0:5000> on the local machine for access.

```
(.venv) pierce@pop-os:~/github/Distributed-Operating-Systems/RESTful_API/Python_API/  
api/Tutorials$ flask run  
* Serving Flask app 'application.py' (lazy loading)  
* Environment: development  
* Debug mode: on  
/home/pierce/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutoria  
ls/.venv/lib/python3.8/site-packages/flask_sqlalchemy/__init__.py:872: FSADeprecatio  
nWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disab  
led by default in the future. Set it to True or False to suppress this warning.  
  warnings.warn(FSADeprecationWarning(  
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)  
* Restarting with stat  
* Debugger is active!  
* Debugger PIN: 734-762-779
```

And upon accessing the root of the server on the local machine, the sanity check passed. It was then time to implement the database and HTTP Request Methods.



Hello - Welcome to the RESTful API tutorials

Status	Meth...	Domain	File	Initiator	Type	Transferred	Size	0 ms	160 ms
200	GET	127.0.0.1:5000	/	document	html	198 B (raced)	44 B	3 ms	

Database Implementation

It was decided to have each entity in the database be represented as an instance of a defined class *Tutorial* to match the general theme of the Guru99 tutorial. The *Tutorial* class includes three attributes: *id* (**PK**), *name*, and *description*. The `__repr__` method within the class defined to help with visualization while building the database. The architecture of the class is shown below for clarity:

```
# Model Objects we store in the DB
class Tutorial(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True,
                     nullable=False)
    description = db.Column(db.String(120))

    def __repr__(self):
        return f"{self.name} - {self.description}"
```

With the database entity class defined in the source code, the db was created in the Python3 environment, the *Tutorial* class was imported into the environment, and some test entities were added to the database.

```
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from application import db
/home/pierce/github/Distributed-Operating-Systems/RESTful_API/Python_API/api/Tutorial
ls/.venv/lib/python3.8/site-packages/flask_sqlalchemy/__init__.py:872: FSADeprecatio
nWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disab
led by default in the future. Set it to True or False to suppress this warning.
  warnings.warn(FSADeprecationWarning(
>>> db.create_all()
>>> from application import Tutorial
>>> tutorial = Tutorial(name="Arrays", description="A tutorial for introducing Array
s")
>>> tutorial
Arrays - A tutorial for introducing Arrays
>>> db.session.add(tutorial)
```

Implementations of GET Methods

Two separate methods for GET functions were implemented in the project – one for retrieving and returning all entities in the database to the client, and the other for returning a single entity at a time, called by *id* and returned to the client.

The first GET method is accessed on the server under *http://(root)/Tutorial*. To allow for JSON to iterate appropriately through the returned data, it was necessary to alter the data type returned from the Flask-SQLAlchemy query. The returned query was converted into a list, or array, of Python Dictionaries. With Python Dictionaries being an iteration-able datatype for JSON, the list of dictionaries is then returned to the client as a Tutorials Dictionary.

```
@app.route('/Tutorial')
def get_tutorials():
    tutorials = Tutorial.query.all()
    output = []
    for tutorial in tutorials:
        tutorial_data = {'name': tutorial.name,
                        'description': tutorial.description}
        output.append(tutorial_data)
    return {"Tutorials": output}
```

The second GET method is accessed on the server under *http://(root)/Tutorial/<id>*. As entities are added to the database, any given entity could be accessed and returned to the client using the associated *id*. This is done by passing the desired *id* to a query on the database. This query then returns either the entity to be displayed to the client, or a 404 – not found.

```
@app.route('/Tutorial/<id>')
def get_tutorial(id):
    tutorial = Tutorial.query.get_or_404(id)
    return {"name": tutorial.name, "description":
            tutorial.description}
```

Implementation of POST Method

The POST method allows for new entities to be added to the database. Unique *id*'s are automatically incremented for the new entities. The POST method is invoked with an API tool such as Postman API Platform. For this specific implementation, JSON was proffered over XML as the

```
@app.route('/Tutorial', methods=['POST'])
def add_tutorial():
    tutorial = Tutorial(name=request.json['name'],
                       description=request.json['description'])
    db.session.add(tutorial)
    db.session.commit()
    return {'id': tutorial.id}
```

means of communication between the server and client. This is primarily due to ease of use when working with Flash-SQLAlchemy.

When invoking the POST Method, a JSON file is passed to the server with two fields, *name* and *description*. These two fields are then parsed for their content and added to a new instance of the *Tutorial* class. This instance of the class is added to the database and changes are committed. A message showing the changes and the assigned *id* are returned.

Implementation of DELETE Method

The DELETE method allows for existing entities within the database to be removed. This method is invoked by passing a URI, or URL, indicating which *id* should be deleted from the database. This URI is sent using an API tool, such as Postman. Using the provided *id* a query is made on the database. If the *id* exists, the entity is deleted and the changes are committed. If the *id* does not appear to be in the database upon query, an error message is returned.

```
@app.route('/Tutorial/<id>', methods=['DELETE'])
def delete_tutorial(id):
    tutorial = Tutorial.query.get(id)
    if tutorial is None:
        return {"error": "not found"}
    db.session.delete(tutorial)
    db.session.commit()
    return {"message": ("Successfully deleted ID " + id +
        " from database.")}
```


Implementation of PUT Method

The PUT method allows for an existing entity within the database to be altered or updated while retaining its unique primary key, the *id*. This method is invoked by passing a URI indicating which *id* should be updated or changed in the database. The URI is sent using an API tool, such as Postman, to invoke the PUT operation on the database via the server. Included with the URI that identifies the specific entity, a JSON script is sent that includes the needed updates or changes. For this implementation, it was chosen to require every PUT invocation to include a name in the *name* field. Optionally, a change can also be made to the *description* field. Some error checking is performed to ensure these criteria are adhered to. Once the changes are received and implemented, the changes are committed to the database. For clarity, the altered entity's *id*, *name*, and *description* are returned for client viewing after the operation is complete.

```
@app.route('/Tutorial/<id>', methods=['PUT'])
def update_tutorial(id):
    data = request.get_json()
    if 'name' not in data:
        return ({'error': 'Bad Request', 'message':
                  'Name a field must be present'}, 400)
    tutorial = Tutorial.query.get_or_404(id)
    tutorial.name = data['name']
    if 'description' in data:
        tutorial.description = data['description']
    db.session.commit()
    return {"id": tutorial.id, "name": tutorial.name,
            "description": tutorial.description}
```

Application and Testing

Setup and Introduction

With the above implementation of REST API in place, it is now possible to perform operations on the database using the defined API end-points. For this phase of the project, following were in place and performing the following roles:

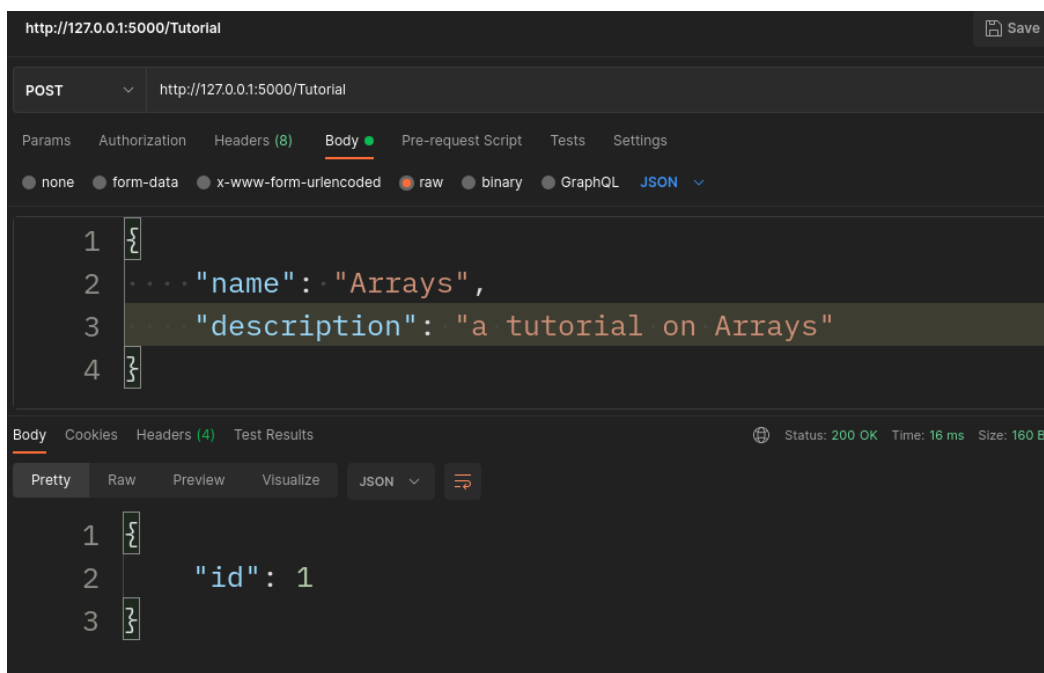
- **Server** – running within the Python3 Virtual Environment via Flask and based on the Python3 source code
- **Database** - implemented in Flask-SQLAlchemy and accessible within the the Python3 Virtual Environment
- **Client** – for this project, the chosen application to act as the client was Postman API Platform (as well as Firefox web browser for some GET invocations)

The database was initially cleared of test values and left completely blank.

POST Invocations to Database

To match the theme displayed in the Guru99 tutorial, three entities were initially added to the database, *Arrays*, *Queues*, and *Stacks* – each entity included a brief description to populate the the *description* field of the passed JSON file. These additions to the database were performed via three separate POST invocations on the server by the Postman client. To spot check, a GET operation was invoked on the server to ensure all data was included in the database.

POST of “Arrays” Entity to Database



POST of “Queues” Entity to Database

http://127.0.0.1:5000/Tutorial

POST http://127.0.0.1:5000/Tutorial

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   ... "name": "Queues",
3   ... "description": "a tutorial on Queues"
4 }
```

Body Cookies Headers (4) Test Results

Status: 200 OK Time: 15 ms Size: 160 B

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 2
3 }
```

POST of “Stacks” Entity to Database

http://127.0.0.1:5000/Tutorial

POST http://127.0.0.1:5000/Tutorial

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   ... "name": "Stacks",
3   ... "description": "a tutorial on Stacks"
4 }
```

Body Cookies Headers (4) Test Results

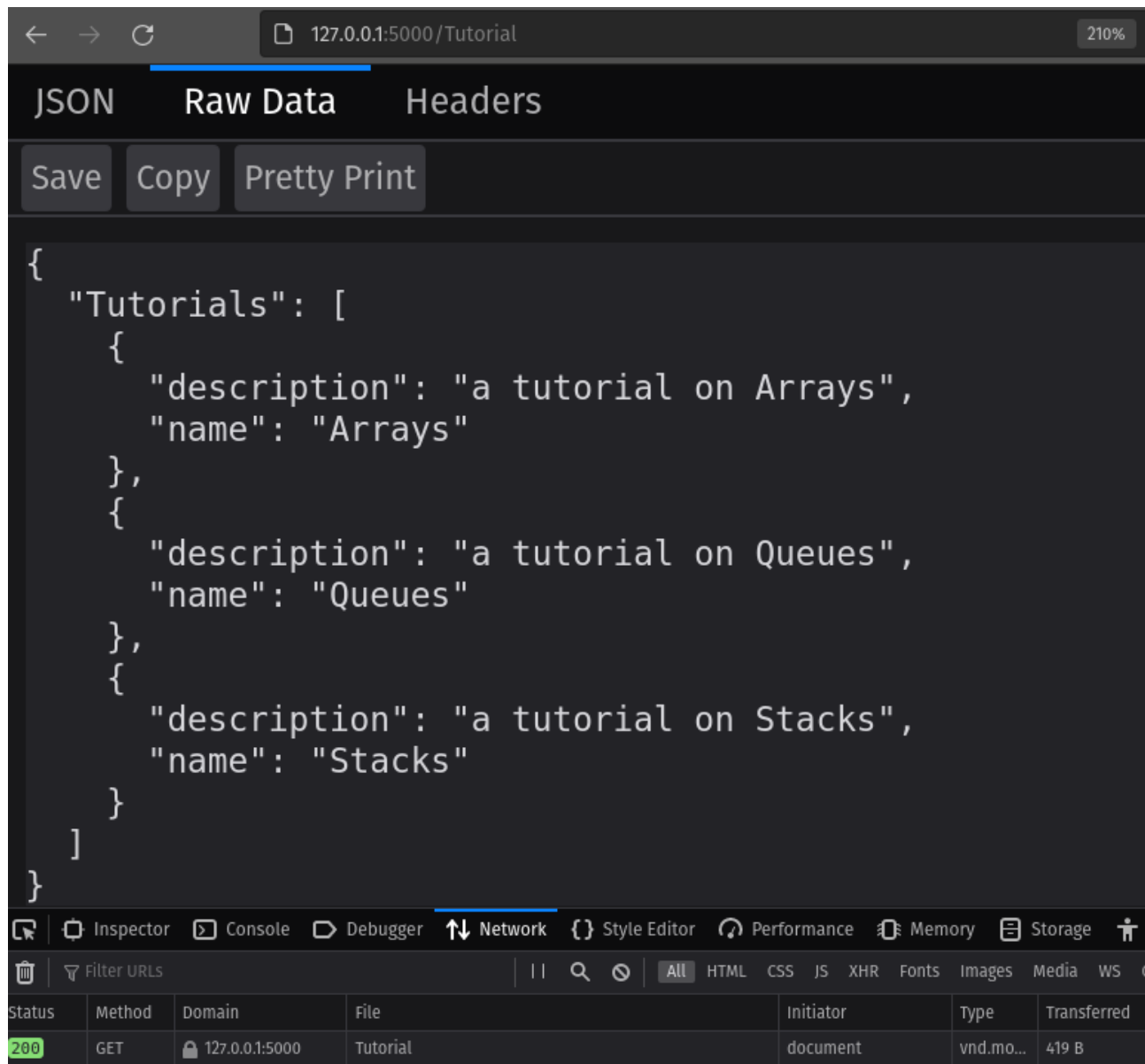
Status: 200 OK Time: 16 ms Size: 160 B

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 3
3 }
```

Initial GET Invocation on Database to Spot-check Entries

Upon addition of the above three entities to the database, the database was spot-checked to ensure the entities were added correctly. This was done by invoking a GET operation that returned the full contents of the database. This GET operation was simply invoked by navigating to <http://127.0.0.0:5000/Tutorial> from the local machine. Alternatively, Postman could be used to perform the GET invocation if desired.



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/Tutorial`. The page content is a JSON response, which is displayed in the 'Raw Data' tab. The JSON structure is as follows:

```
{
  "Tutorials": [
    {
      "description": "a tutorial on Arrays",
      "name": "Arrays"
    },
    {
      "description": "a tutorial on Queues",
      "name": "Queues"
    },
    {
      "description": "a tutorial on Stacks",
      "name": "Stacks"
    }
  ]
}
```

Below the JSON view, the browser's developer tools are open, showing the 'Network' tab. It displays a single request to `127.0.0.1:5000` with the method `GET` and the file `Tutorial`. The status is `200`, and the transferred size is `419 B`.

Status	Method	Domain	File	Initiator	Type	Transferred
200	GET	127.0.0.1:5000	Tutorial	document	vnd.mo...	419 B

GET Method for Selecting a Single and Specific ID

With the database populated with the initial values, it was then possible to showcase the second GET Method. Using a URI that indicates a specific *id* of desired entity, the application has the end-point capabilities to allow retrieval of single and specific entities within the database. This was easily shown using Firefox web browser and altering the last digit of the URI to indicate which entity should be returned.

ID GET on ID: 1

The screenshot shows a Firefox browser window with the address bar displaying `127.0.0.1:5000/Tutorial/1`. The page content is a JSON object: `{"description": "a tutorial on Arrays", "name": "Arrays"}`. The Network tab is open, showing a successful GET request to the same URL with a status code of 200. The response size is 211 B (raced).

Status	Meth...	Domain	File	Initiator	Type	Transferred
200	GET	127.0.0.1:5000	1	document	vnd....	211 B (raced)

ID GET on ID: 2

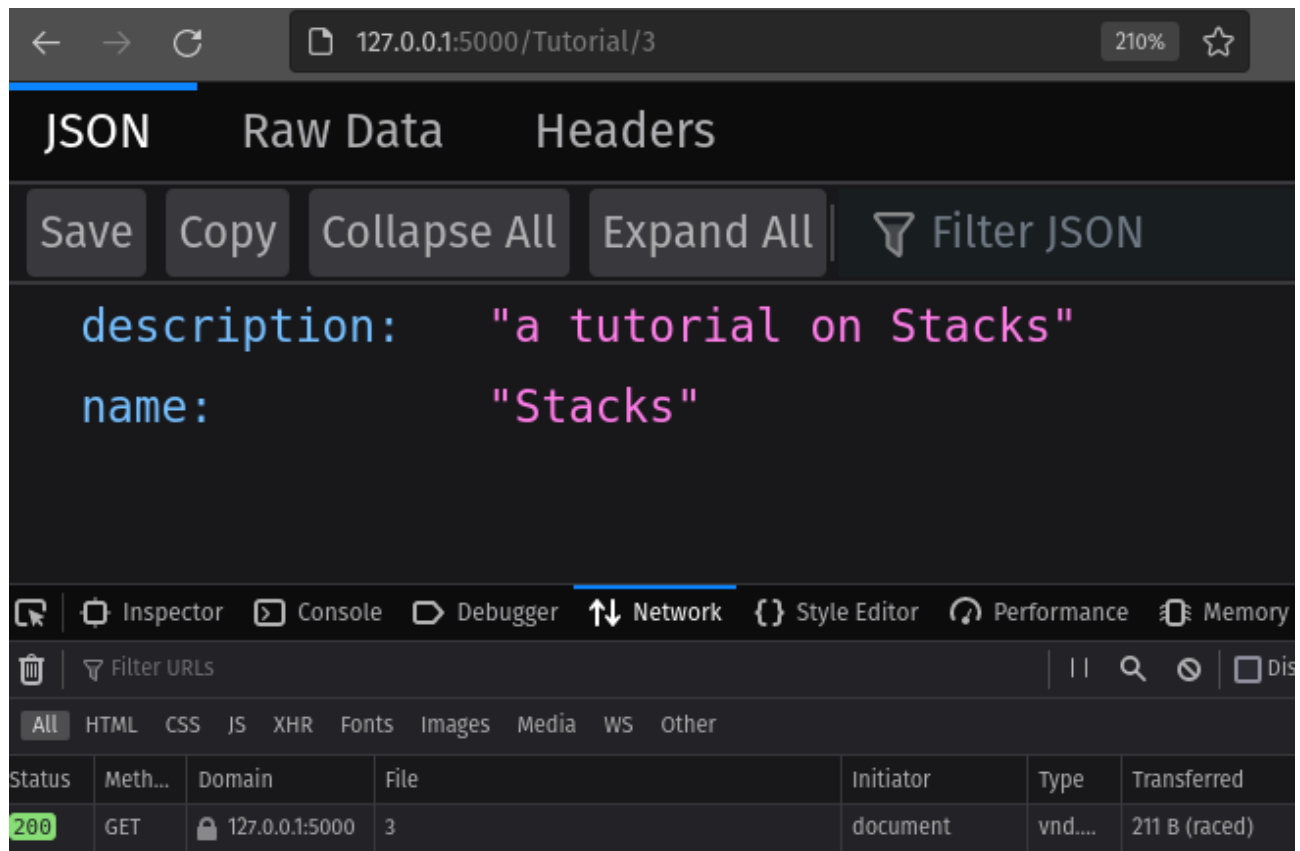
The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/Tutorial/2`. The main content area shows a JSON response with the following structure:

```
{  "description": "a tutorial on Queues",  "name": "Queues"}
```

The browser's developer tools are open, with the **Network** tab selected. The network log shows a single request with the following details:

Status	Meth...	Domain	File	Initiator	Type	Transferred
200	GET	127.0.0.1:5000	2	document	vnd....	211 B

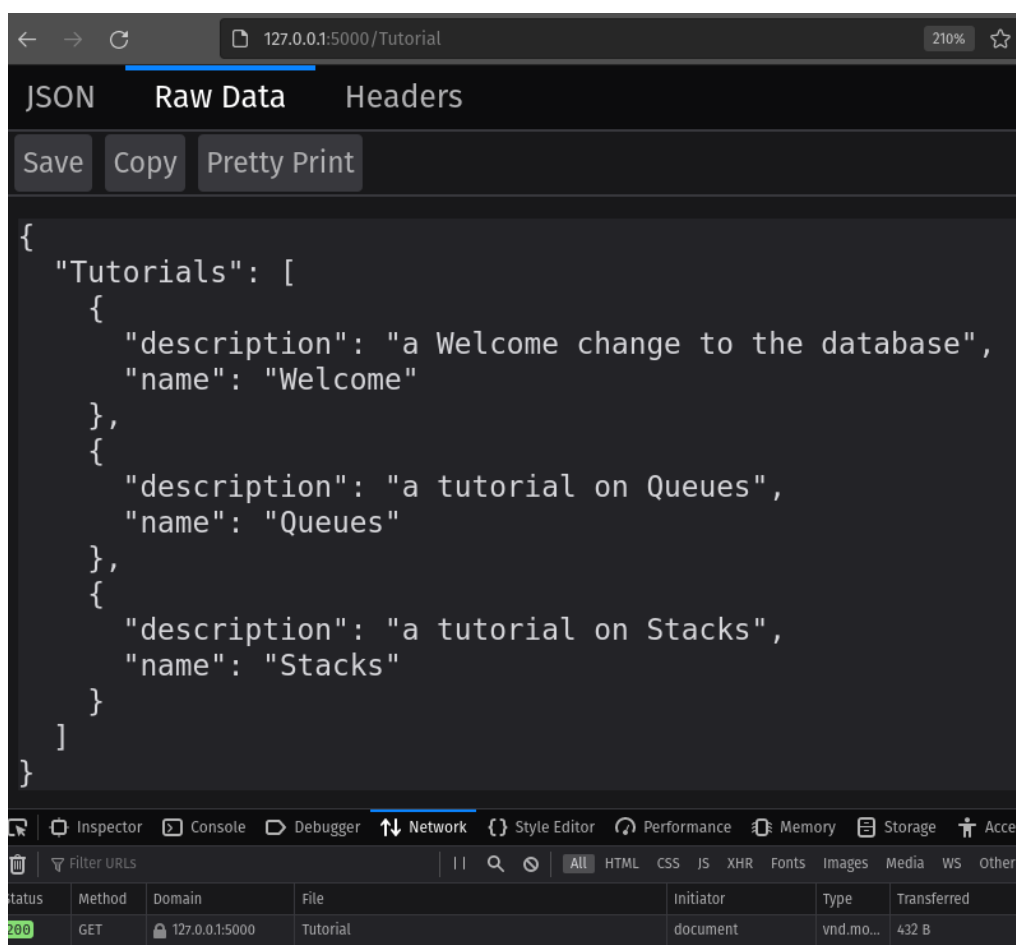
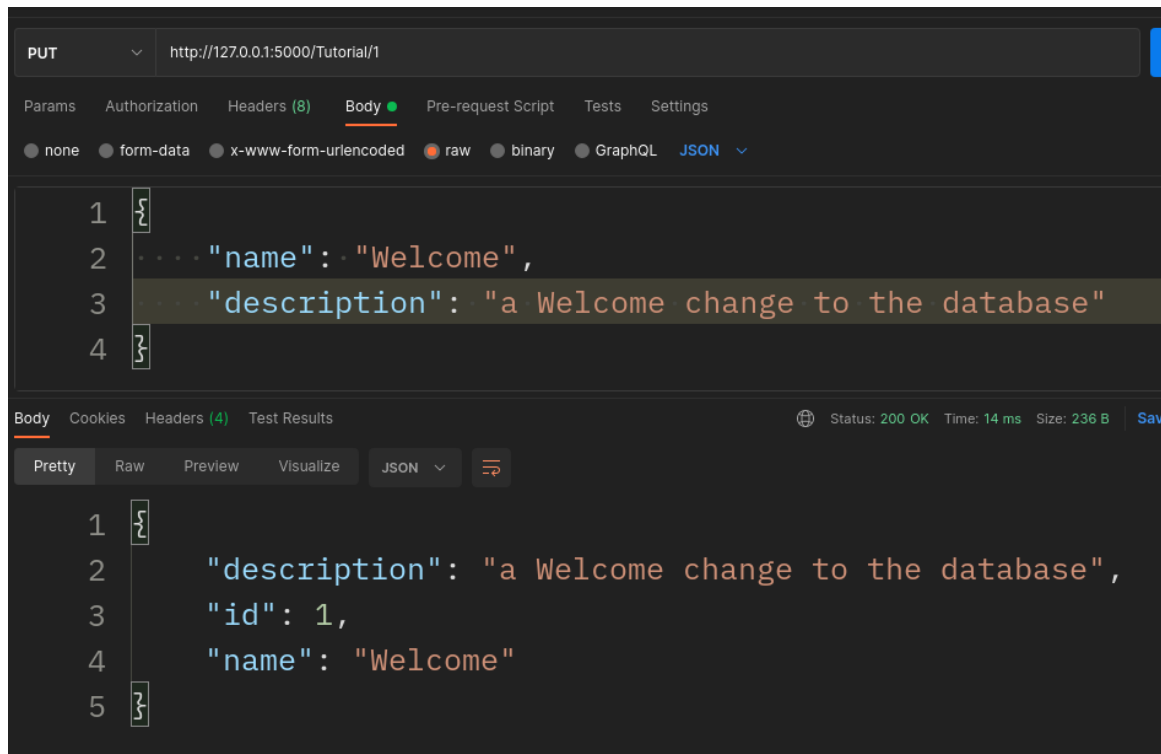
ID GET on ID: 3



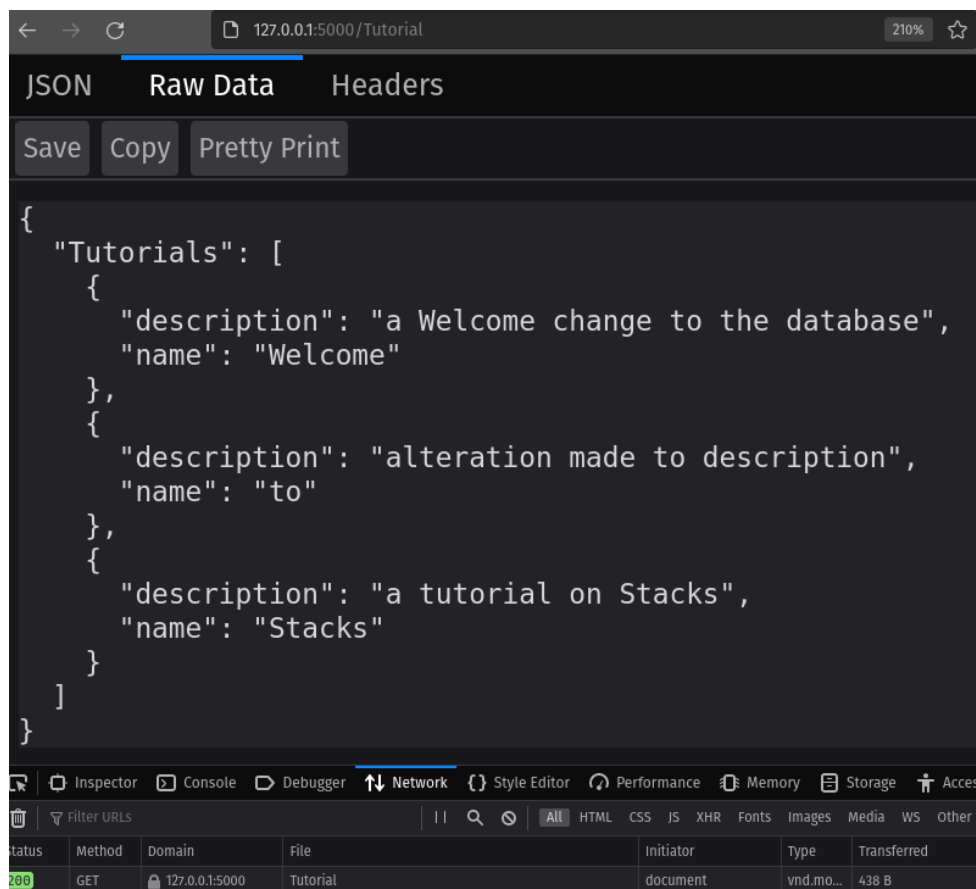
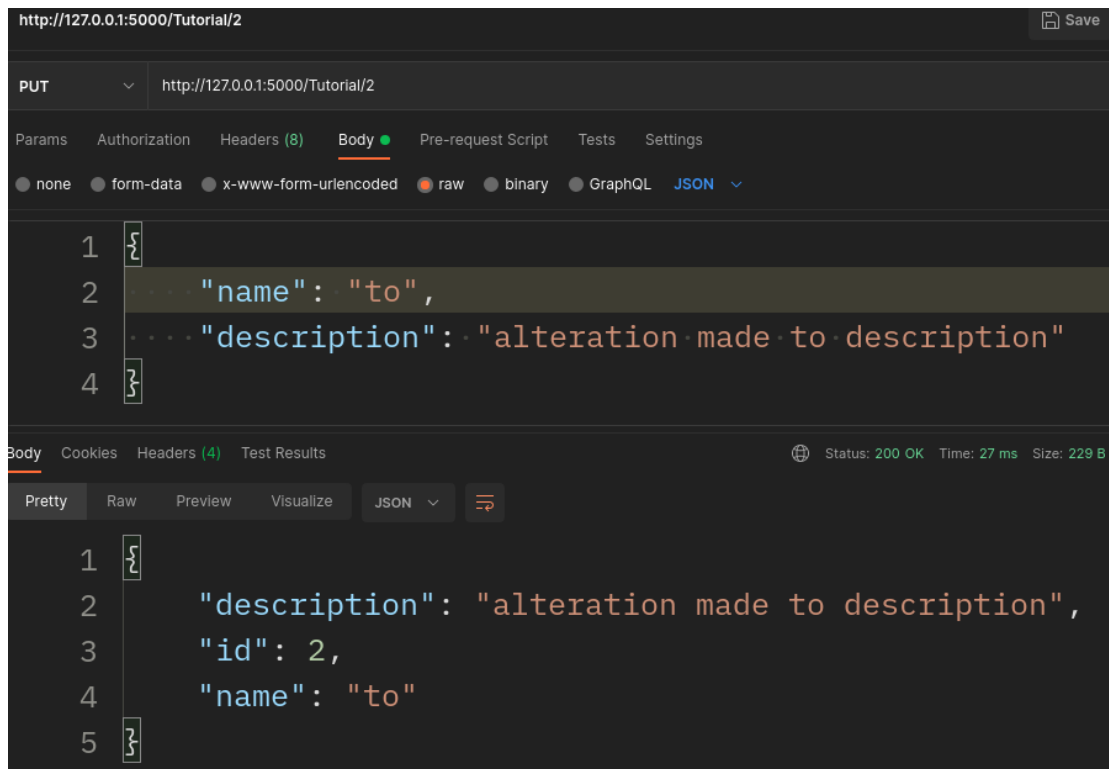
PUT Method Invoked on Database

With the database populated with the initial values, it was then possible to invoke the PUT method to further modify the database. In accordance with homework assignment statement the three existing entities in the database were changed via the PUT method – this changed the entities from “Arrays, Queues, Stacks” to “Welcome, to, CSCI 6220” respectively. Below are screenshots of each of the three PUT Invocations as well as a GET invocation on the entire database after each change. The PUT Invocations were done in ascending incremental order to show the full functionality of the PUT method – indeed changing the attributes for a given entity, while retaining the same unique *id* as apposed to deleting the entity and creating a new entity at the end of the database (an incorrect implementation of the PUT method).

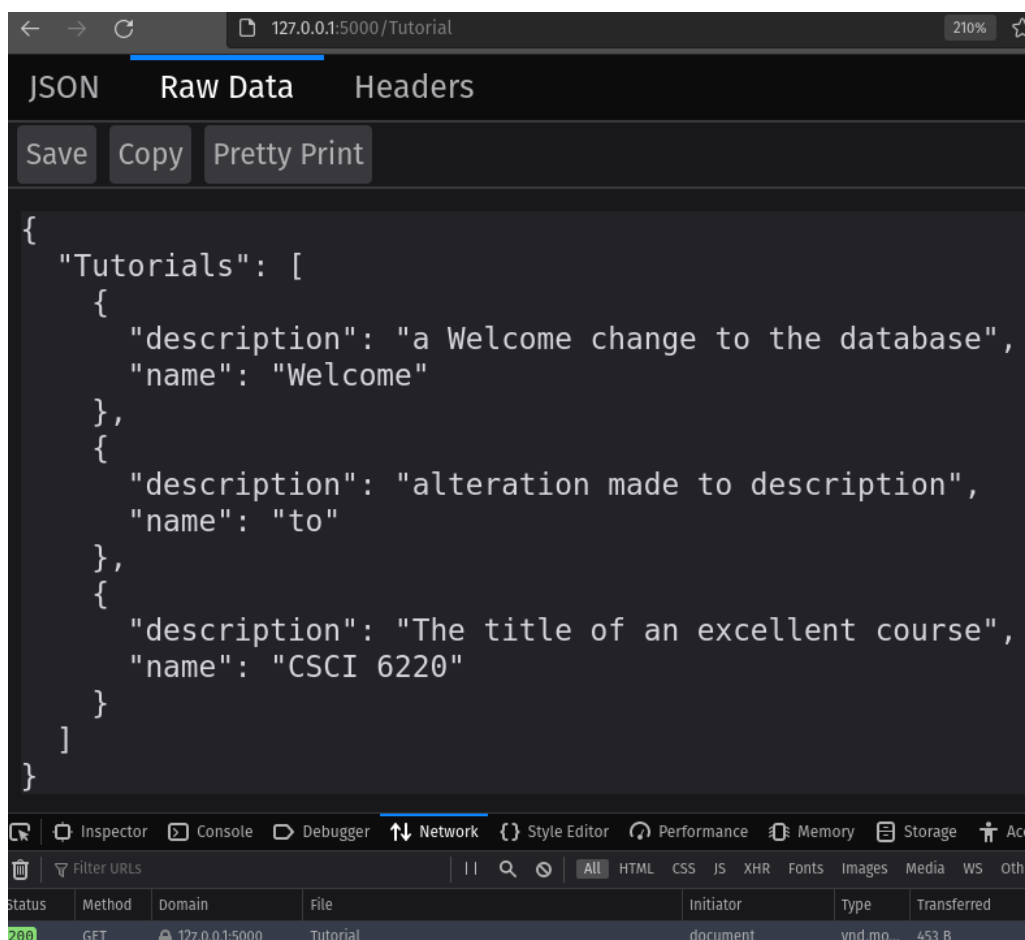
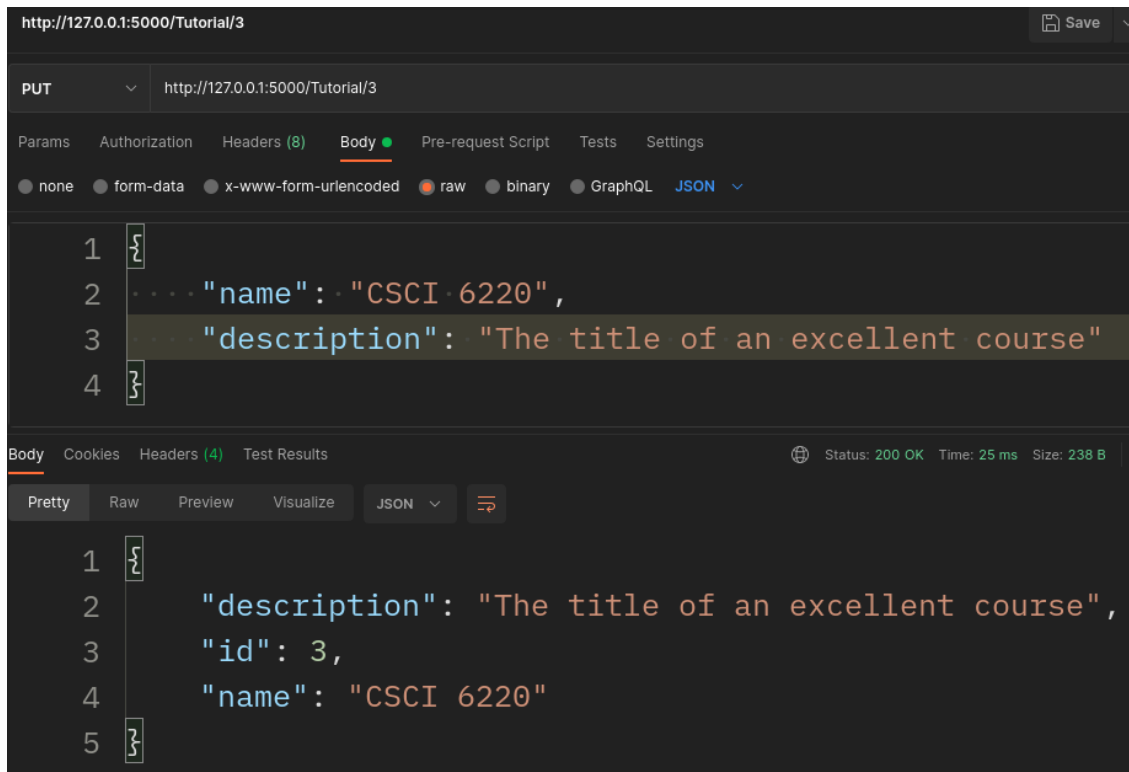
PUT Invocation to Change “Arrays” to “Welcome”



PUT Invocation to Change “Queues” to “to”

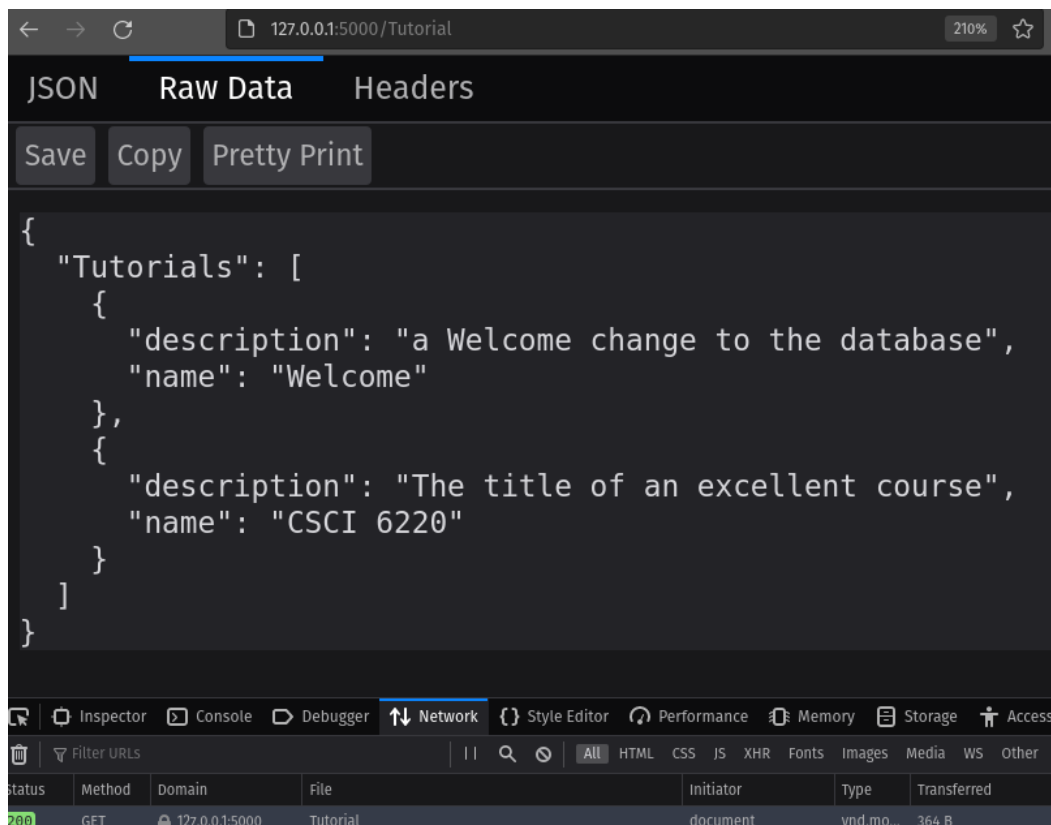
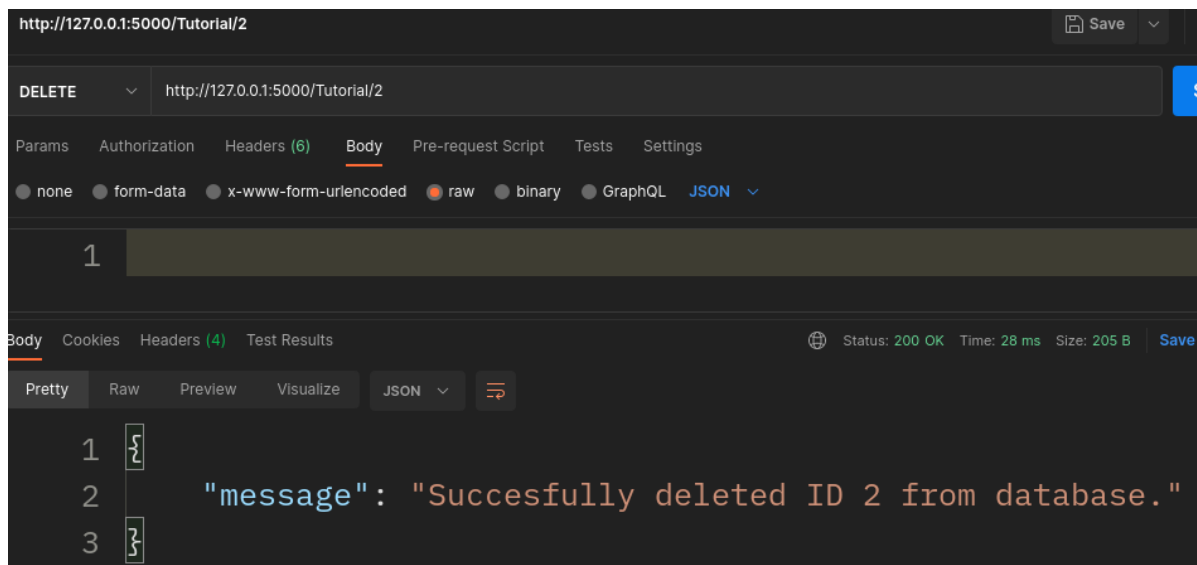


PUT Invocation to Change “Stacks” to “CSCI 6220”

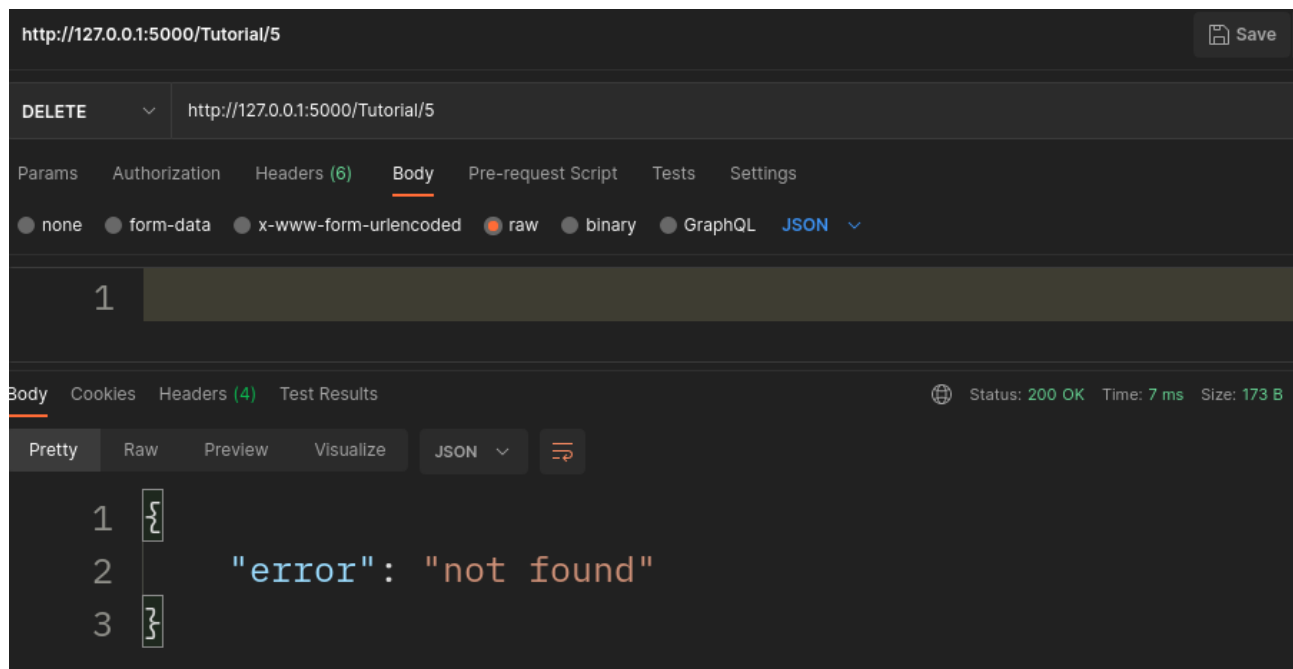


DELETE Method Invoked on Database

Lastly, with the populated database, it is possible to demonstrate the DELETE method. This method unsurprisingly offers the end-point capabilities to invoke the server to remove an entity from the database. This method is invoked using the Postman stand in client and supplying a DELETE HTTP Request with the ID for the desired deleted entity. Below is a screen show showing the deletion of the “to” entity from the database. There is then an additional screenshot taken after invoking the GET method on the database to show the still existing “Welcome” and “CSCI 6220” entities. A response message is displayed to the client after the DELETE method invocation detailing that the entity with the associated ID has been deleted.



The last thing to note on the DELETE method is the case in which DELETE is called on an *id* that does not exist in the database. In the event this happens, the supplied error message is returned to the client indicating that the given URI was not found.



Conclusion

The Representational State Transfer Application Programming Interface (REST API) has become a ubiquitous technology in today's distributed system landscape ^[1]. Apart from the many advantages the technology offers in terms of security and adaptability, one of the biggest advantages of REST API is the ability to quickly and easily create end-points that can be used by heterogeneous systems ^[1]. With the widely heterogeneous landscape of today computing field, this ability is a must.

Although the application presented here in the project is somewhat basic in nature, it effectively illustrates the ease of implementation of this technology and the ease in which it can be adapted to different systems. Four of the main HTTP Request Methods, POST, GET, DELETE, and PUT ^[1], used in REST API applications have been successfully implemented and demonstrated. An appropriate level of understanding of the topic has also been displayed by the student.

Additional Screenshots

The below screenshots show the feedback from the server with timestamps any time any HTTP Method was invoked on the server. These are added to show additional clarity of the server side activity during the Application and Testing phase of this project.

```
/home/pierce/github/Distributed-Operating-Systems/RESTful_API/Python_API/a
3.8/site-packages/flask_sqlalchemy/__init__.py:872: FSADeprecationWarning:
NS adds significant overhead and will be disabled by default in the future
suppress this warning.
  warnings.warn(FSADeprecationWarning(
127.0.0.1 - - [18/Apr/2022 21:19:27] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [18/Apr/2022 21:19:36] "GET /Tutorial/1 HTTP/1.1" 200 -
127.0.0.1 - - [18/Apr/2022 21:19:42] "GET /Tutorial/2 HTTP/1.1" 404 -
127.0.0.1 - - [18/Apr/2022 21:19:49] "GET /Tutorial/3 HTTP/1.1" 200 -
127.0.0.1 - - [18/Apr/2022 21:19:52] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [18/Apr/2022 21:20:47] "DELETE /Tutorial/1 HTTP/1.1" 200 -
127.0.0.1 - - [18/Apr/2022 21:20:53] "DELETE /Tutorial/3 HTTP/1.1" 200 -
127.0.0.1 - - [18/Apr/2022 21:20:57] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [18/Apr/2022 21:21:12] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [18/Apr/2022 21:21:19] "GET /Tutorial/1 HTTP/1.1" 404 -
127.0.0.1 - - [19/Apr/2022 00:10:11] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 00:10:11] "GET /favicon.ico HTTP/1.1" 404 -
* Detected change in '/home/pierce/github/Distributed-Operating-Systems/R
orials/application.py', reloading
* Restarting with stat
```

```
/home/pierce/github/Distributed-Operating-Systems/RESTful_API/Python_API/a
3.8/site-packages/flask_sqlalchemy/__init__.py:872: FSADeprecationWarning:
NS adds significant overhead and will be disabled by default in the future
suppress this warning.
  warnings.warn(FSADeprecationWarning(
127.0.0.1 - - [19/Apr/2022 01:16:17] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:16:54] "POST /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:17:13] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:17:20] "GET /Tutorial/1 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:17:26] "DELETE /Tutorial/1 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:17:40] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:18:35] "POST /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:18:43] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:24:13] "DELETE /1 HTTP/1.1" 404 -
127.0.0.1 - - [19/Apr/2022 01:24:24] "DELETE /Tutorial/1 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:24:31] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:30:55] "POST /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:31:44] "POST /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:35:47] "POST /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:41:51] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:50:02] "GET /Tutorial/1 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:51:48] "GET /Tutorial/2 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:53:23] "GET /Tutorial/3 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 01:58:44] "PUT /Tutorial/1 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 02:03:57] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 02:05:42] "PUT /Tutorial/2 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 02:06:50] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 02:10:47] "PUT /Tutorial/3 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 02:11:05] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 02:20:54] "DELETE /Tutorial/2 HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 02:22:16] "GET /Tutorial HTTP/1.1" 200 -
127.0.0.1 - - [19/Apr/2022 02:25:07] "DELETE /Tutorial/5 HTTP/1.1" 200 -
```

Work Cited and Reference Material

- [1] A. Walker, “RESTful Web Services Tutorial: What is REST API with Example,” *www.guru99.com*, 2020. <https://www.guru99.com/restful-web-services.html#restful-methods> (accessed Apr. 18, 2022).

- [2] “API — Flask Documentation (2.0.x),” *flask.palletsprojects.com*. <https://flask.palletsprojects.com/en/2.0.x/api/> (accessed Apr. 18, 2022).

- [3] B. Koiki, “Building Restful APIs With Flask and SQLAlchemy (Part 1),” *Medium*, Dec. 14, 2020. <https://betterprogramming.pub/building-restful-apis-with-flask-and-sqlalchemy-part-1-b192c5846ddd> (accessed Apr. 18, 2022).

- [4] “Flask-SQLAlchemy — Flask-SQLAlchemy Documentation (2.x),” *flask-sqlalchemy.palletsprojects.com*. <https://flask-sqlalchemy.palletsprojects.com/en/2.x/>

- [5] “venv — Creation of virtual environments — Python 3.8.1 documentation,” *Python.org*, 2019. <https://docs.python.org/3/library/venv.html>