CS 61:
Database Systems

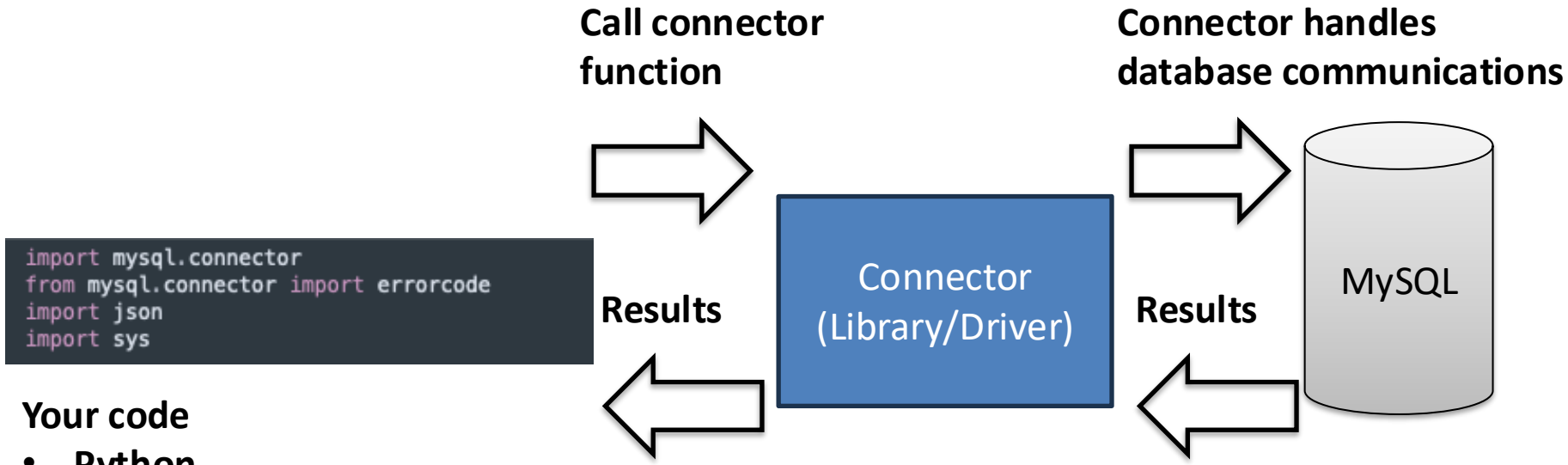
Access via programming languages

Agenda

1. Direct database access

2. Web APIs

Use a library to connect to the database from your code



Your code

- **Python**
- **Java**
- **C**
- **JavaScript (web)**

Connectors

- **Python: mysql.connector**
- **Java: JDBC**
- **C: libmysqlclient**
- **JavaScript: mysql2**

install: pip3 install mysql-connector-python
import mysql.connector

Simple example: Python program to get restaurant data from the command line

```
$ python3 get_restaurants.py nobu remote
```

sys.argv[0]: Python program we will write

sys.argv[1]: restaurant name for which we will get data

sys.argv[2]: Optional, database to use

- **localhost**
- **remote (CS server)**

localhost is default in this example

Four steps:

1. Read database credentials from file (do not hardcode credentials)
2. Get database connection using credentials
3. Query database for data
4. Close connection

What types of databases?

- **Production**
- **QA**
- **Dev**
- **Local**

Step 1: Read database credentials from file

db.json

Avoid hardcoding your credentials in your code, instead read from a config file (db.json here)

```
{
    "remote": {
        "user":"your username",
        "password":"your password",
        "host":"some_server_name.cs.dartmouth.edu",
        "database":"nyc_data"
        },
    "localhost": {
        "user":"your username",
        "password":"your password",
        "host":"localhost",
        "database":"nyc_data"

    }
}
```

Credentials to two databases:

- **remote (CS server)**
- **localhost (your local machine's database)**

Update this file with your credentials to these databases

Add this file to .gitignore!

Bots scan for credentials

Better: user prompt for username/password rather than put in config file

Even better: use API and put in environment variable

Step 1: Read database credentials from file

In main

Get database from command line args (localhost is default)

get_restaurants.py

```
if __name__ == '__main__':
    #check usage
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print("Usage: python get_restaurants.py restaurant_name <localhost|remote host> ")

    #Step 1: read database credentials from file
    #check which database to use (usually "localhost" or "remote")
    database = "localhost"
    if len(sys.argv) == 3:
        database = sys.argv[2]
    print(f'Reading database credentials from file: {credentials_filename} for database: {database}')
    credentials = read_database_credentials(credentials_filename, database)
    print("\tDone")

    #Step 2: get database connection using credentials from file
    print(f'Connecting to database {credentials["database"]} on {credentials["host"]}')
    cnx = get_database_connection(credentials)
    print("\tDone, got good connection") #would exit in get_database_connection if not successful

    #Step 3: query database for restaurant data
    restaurant = sys.argv[1]
    print("\nFetching data about",restaurant)
    get_restaurant_data(cnx,restaurant)

    #Step 4: close connection to database and exit
    cnx.close()
```

Read db.json file and get credentials for the database (either localhost or sys.argv[2])

Step 1: Read database credentials from file

get_restaurants.py

filename = db.json, database = localhost or sys.argv[2]

```
credentials_filename = "db.json" #file holding username and password to database

def read_database_credentials(credentials_filename, database="localhost"):
    #Read credentials from config file specified by credentials_filename
    #Input: name of file holding credentials
    #Output: dictionary of credentials with entries for username, password, host, and database
    #        exit if file not found or database name is invalid

    #read credentials file
    credentials_file = open(credentials_filename,'r')
    credentials = json.load(credentials_file)
    credentials_file.close()

    #return credentials for the database parameter (normally "localhost" or "remote")
    return credentials[database]
```

Read json from db.json file

Return for localhost or remote

```
{ "user":"your username",
  "password":"your password",
  "host":"localhost",
  "database":"nyc_data"}
```

Step 2: Get database connection using credentials

In main

get_restaurants.py

```
if __name__ == '__main__':
    #check usage
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print("Usage: python get_restaurants.py restaurant_name <localhost|remote host> ")

    #Step 1: read database credentials from file
    #check which database to use (usually "localhost" or "remote")
    database = "localhost"
    if len(sys.argv) == 3:
        database = sys.argv[2]
    print(f'Reading database credentials from file: {credentials_filename} for database: {database}')
    credentials = read_database_credentials(credentials_filename, database)
    print("\tDone")

    #Step 2: get database connection using credentials from file
    print(f'Connecting to database {credentials["database"]} on {credentials["host"]}')
    cnx = get_database_connection(credentials)
    print("\tDone, got good connection") #would exit in get_database_connection if not successful

    #Step 3: query database for restaurant data
    restaurant = sys.argv[1]
    print("\nFetching data about",restaurant)
    get_restaurant_data(cnx,restaurant)

    #Step 4: close connection to database and exit
    cnx.close()
```

Get database connection

Step 2: Get database connection using credentials

get_restaurants.py

```
def get_database_connection(credentials):
    #Get a connection to database
    #Input: dictionary with username, password, host, and database
    #Output: database connection or exit if connection not successful

    try:
        #make connection to database on host server using credentials provided
        cnx = mysql.connector.connect(user=credentials["user"], password=credentials["password"],
                                      host=credentials["host"], database=credentials["database"])
        return cnx
    #catch exceptions, alert user to problem, then exit
    except mysql.connector.Error as err:
        if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
            print("Something is wrong with your user name or password")
        elif err.errno == errorcode.ER_BAD_DB_ERROR:
            print("Database does not exist")
        else:
            print(err)
        cnx.close()
        exit()
```

{ "user":"your username",
"password":"your password",
"host":"localhost",
"database":"nyc_data"}

Get database connection and return connection if successful

Handle errors/exit
mysql.connector can tell you about the type of error

Step 3: Query the database for data

In main

get_restaurants.py

```
if __name__ == '__main__':
    #check usage
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print("Usage: python get_restaurants.py restaurant_name <localhost|remote host> ")

    #Step 1: read database credentials from file
    #check which database to use (usually "localhost" or "remote")
    database = "localhost"
    if len(sys.argv) == 3:
        database = sys.argv[2]
    print(f'Reading database credentials from file: {credentials_filename} for database: {database}')
    credentials = read_database_credentials(credentials_filename, database)
    print("\tDone")

    #Step 2: get database connection using credentials from file
    print(f'Connecting to database {credentials["database"]} on {credentials["host"]}')
    cnx = get_database_connection(credentials)
    print("\tDone, got good connection") #would exit in get_database_connection if not successful

    #Step 3: query database for restaurant data
    restaurant = sys.argv[1]
    print("\nFetching data about",restaurant)
    get_restaurant_data(cnx,restaurant)

    #Step 4: close connection to database and exit
    cnx.close()
```

Query database for data

Step 3: Query the database for data

```python
def get_restaurant_data(cnx, restaurant_name):
    # Query the database for database restaurants like restaurant_name
    # Input:
    #   cnx: connection to database
    #   restaurant_name: name of restaurant to fetch from database
    # Output: results of query printed to screen
    # Reference: https://pynative.com/python-mysql-execute-parameterized-query-using-prepared-statement/

    # get cursor from database connection and execute query using parameterized query and prepared statement
    cursor = cnx.cursor(prepared=True)  # use a prepared statement to avoid SQL injection!
    query = ("SELECT RestaurantID, RestaurantName, Boro, CuisineDescription, InspectionAvg "
             + "FROM Restaurants r JOIN Cuisine c USING (CuisineID) "
             + "WHERE RestaurantName LIKE %s")  # %s is a parameter
    cursor.execute(query, ('%' + restaurant_name + '%',))  # second item must be tuple, one entry per parameter

    # get column names
    cols = cursor.description  # column names plus other attributes stored in description, one tuple per column
    column_names = []  # store column names in this list
    for c in cols:  # loop over all columns
        column_names.append(c[0])  # column name is first item in tuple, extract it and append to column_names list

    # fetch ALL rows at once into a list of tuples
    rows = cursor.fetchall()

    # print rows, each row is a tuple of attributes
    # unlike iterating over the cursor, rows is a regular list
    # so you CAN loop over it multiple times if needed
    for row in rows:
        for i in range(len(column_names)):
            print(column_names[i] + ":" + str(row[i]))  # prints database attribute: value
        print()

    # close cursor
    cursor.close()
```

Get cursor (iterator!)

SQL query

Execute SQL passing parameters

Get column (attribute) names

fetchall returns all rows
Also:
fetchone
fetchmany(num)

Loop over rows/cols and print

Close cursor

Problem: business logic is hidden inside python code
We can do better!

Step 4: Close connection

In main

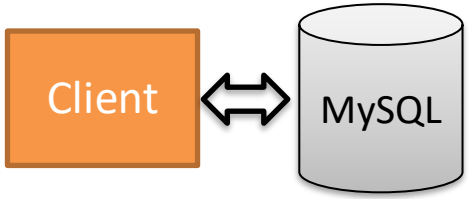
get_restaurants.py

```
if __name__ == '__main__':
    #check usage
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print("Usage: python get_restaurants.py restaurant_name <localhost|remote host> ")

    #Step 1: read database credentials from file
    #check which database to use (usually "localhost" or "remote")
    database = "localhost"
    if len(sys.argv) == 3:
        database = sys.argv[2]
    print(f'Reading database credentials from file: {credentials_filename} for database: {database}')
    credentials = read_database_credentials(credentials_filename, database)
    print("\tDone")

    #Step 2: get database connection using credentials from file
    print(f'Connecting to database {credentials["database"]} on {credentials["host"]}')
    cnx = get_database_connection(credentials)
    print("\tDone, got good connection") #would exit in get_database_connection if not successful

    #Step 3: query database for restaurant data
    restaurant = sys.argv[1]
    print("\nFetching data about",restaurant)
    get_restaurant_data(cnx,restaurant)

    #Step 4: close connection to database and exit
    cnx.close()
```

Close connection

get_restaurants.py is example of client-side Python code directly querying the database

get_restaurants.py



1. Download get_restaurants.py and db.json from course web page
2. Edit db.json with your credentials
3. Run:

   python get_restaurants.py restaurant_name <localhost|remote>

   - Replace restaurant_name with your own choice (e.g., Nobu or 'Rosa Mexicano')
   - Provide either localhost or remote (localhost default)

   python3 get_restaurants.py nobu remote
   python3 get_restaurants.py 'rosa mexicano' localhost

   Fetches data about matching restaurant names

There are several problems with this approach

Problems:

Business logic is hidden inside Python code

- What if database changes (MySQL -> Postgress)?
- What it structure of database changes?

If changes are made, must update all copies of an application that uses the database!

- There might be many applications that use the database
- All copies of each application will need to be updated!

The database credentials are stored locally, so everyone can see them!

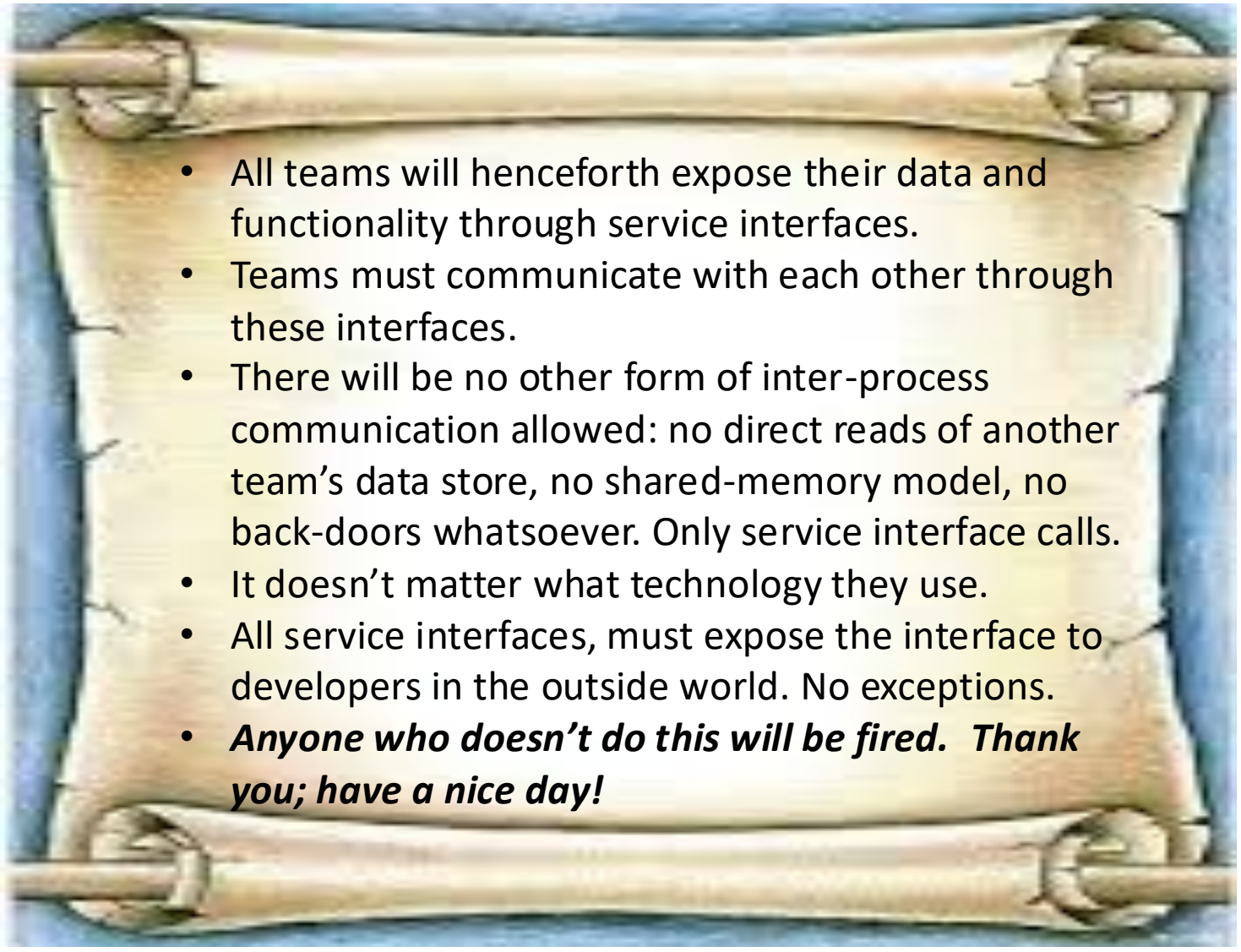
A better approach, create an API that applications can call

Agenda

1. Direct database access

2. Web APIs

A (possibly apocryphal) letter from Jeff Bezos to Amazon developers



- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. Only service interface calls.
- It doesn't matter what technology they use.
- All service interfaces, must expose the interface to developers in the outside world. No exceptions.
- ***Anyone who doesn't do this will be fired. Thank you; have a nice day!***

In short: don't do what we just did with Python!

Create an API instead

APIs are the backbone of micro services architectures

Source: https://apievangelist.com/2012/01/12/the-secret-to-amazons-success-internal-apis/, lightly edited for space

https://medium.com/@benbob/the-gospel-of-dogfooding-can-i-hear-an-amen-brother-1af4d82cf221

To transfer data between computers we use pre-defined protocols

Network protocols

- Network protocols define how data will be exchanged so everyone knows the "rules"
- There are dozens of protocols used for different purposes:
  - TCP/IP, FTP
  - Wi-Fi, Bluetooth
- HyperText Transfer Protocol (HTTP) is the protocol commonly used by the World Wide Web to retrieve web pages (HTML)
- We use a Uniform Resource Locator (URL) to specify what page we want to get:

http://www.cs.dartmouth.edu/~tjp/cs60/index.html

Protocol: how we will talk (http)

Computer that has data

Your computer is: localhost

Directory where data located

File (assume index.html or index.php if not provided)

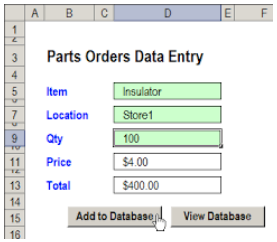
RESTful APIs rely on four HTTP “verbs” to implement CRUD operations

Client side

Server side

Smart phone apps

Web browser



“Thick client” apps

API

URL/api/restaurants

Clients make RESTful calls over network to API listening on server

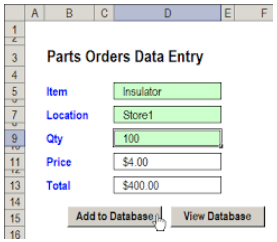
Server has database credentials

RESTful APIs rely on four HTTP "verbs" to implement CRUD operations

Client side

Smart phone apps

Web browser



"Thick client" apps

Server side

API
URL/api/restaurants

Clients make RESTful calls over network to API listening on server

Server has database credentials

POST
/api/restaurants

Create: use POST
Include params to create a new restaurant

GET
/api/restaurants
/api/restaurants/:id

Read: use GET
If no id passed in URL, get all restaurants, otherwise get data for RestaurantID = :id

PUT
/api/restaurants/:id

Update: use PUT
Update restaurant with RestaurantID = :id

DELETE
/api/restaurants/:id

Delete: use DELETE
Delete restaurant with RestaurantID = :id

By convention HTTP verb tells API what CRUD operation to perform

Calls are stateless (all information needed is provided in each call)

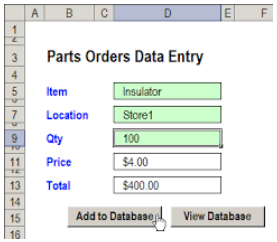
A URL + verb is called an *endpoint*

RESTful APIs rely on four HTTP "verbs" to implement CRUD operations

Client side

Smart phone apps

Web browser



"Thick client" apps

Server side

API
URL/api/restaurants

Clients make RESTful calls over network to API listening on server

Server has database credentials

POST
/api/restaurants

Create: use POST
Include params to create a new restaurant

GET
/api/restaurants
/api/restaurants/:id

Read: use GET
If no id passed in URL, get all restaurants, otherwise get data for RestaurantID = :id

PUT
/api/restaurants/:id

Update: use PUT
Update restaurant with RestaurantID = :id

DELETE
/api/restaurants/:id

Delete: use DELETE
Delete restaurant with RestaurantID = :id

APIs access database and return results to client side

MySQL

APIs access database as user with minimal required rights

Use URLs to identify *things* (nouns) and HTTP methods to identify *actions* (verbs)

Client makes a request to an *endpoint* on a server using *URL* and *verb*

GET	URL/api/restaurants	→ get all restaurants
GET	URL/api/restaurants/3	→ get restaurant with id 3
POST	URL/api/restaurants	→ create a new restaurant
PUT	URL/api/restaurants/3	→ update restaurant with id 3
DELETE	URL/api/restaurants/3	→ delete restaurant with id 3

Each of these is a different endpoint

JSON data is sent to the server in a request body

Server responds with a status code

- 200 OK — Success
- 201 Created — New resource created
- 400 Bad Request — Invalid input
- 401 Unauthorized — Not logged in
- 403 Forbidden — Logged in but no access
- 404 Not Found — Resource doesn't exist
- 500 Server Error — Something broke

URL is the web server's address
On your laptop address called *localhost*

http://localhost/ is the root or home directory of a web server on your local machine

HTTP uses port 80 by default, we will use port 8080 to avoid conflicts with other software you might be running

http://localhost:8080/ will be home
http://localhost:8080/api/restaurants is an *end point*

Create a new API running on your laptop using Flask

Flask: a lightweight, "micro" web framework written in Python, designed for building web applications and APIs quickly and easily

- Install: pip3 install flask
- Documentation: https://flask.palletsprojects.com/en/stable/

```
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip3 install flask
```

Create new directory for this API

Create new file: app.py

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "API is running!"

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

Import Flask

Variable that references the Flask application

Route: http://localhost:8080 will end up here running *home()* function

***home()* just returns string "API is running"**

Start this web application running in main

Create a new API running on your laptop using Flask

Flask: a lightweight, "micro" web framework written in Python, designed for building web applications and APIs quickly and easily

- Install: pip3 install flask
- Documentation: https://flask.palletsprojects.com/en/stable/

Create new directory for this API

Create new file: app.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "API is running!"

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

Server is now listening for connections to http://localhost:8080/

Start server running your API

```
$ cd <api directory>
$ flask --app app.py run --port 8080
```

If you called your Python program app.py, you can omit --app app.py
flask run –port 8080

Create a new API running on your laptop using Flask

Flask: a lightweight, "micro" web framework written in Python, designed for building web applications and APIs quickly and easily

- Install: pip3 install flask
- Documentation: https://flask.palletsprojects.com/en/stable/

Create new directory for this API

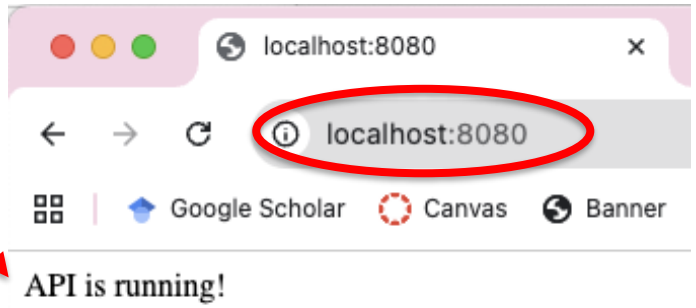
Create new file: app.py

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "API is running!"

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

Server is now listening for connections to http://localhost:8080/

Start server running your API

```
$ cd <api directory>
$ flask --app app.py run --port 8080
```



Web browser

Add a route that gets and returns data from the database

```python
from flask import Flask, jsonify
import mysql.connector
import json

app = Flask(__name__)

# Load credentials from the JSON file
with open('db.json') as config_file:
    credentials = json.load(config_file)
credentials = credentials['localhost']

def get_db_connection():
    return mysql.connector.connect(host=credentials['host'], user=credentials['user'],
            password=credentials['password'], database=credentials['database'])


@app.route('/')
def home():
    return "API is running!"
```

Import Flask, jsonify (json helper) and MySQL connector

Read database credentials

Function to get a connection using mysql.connector (change if using different database)

Add a route that gets and returns data from the database

```python
@app.route('/')
def home():
    return "API is running!"


@app.route('/restaurants', methods=['GET'])
def get_restaurants():
    try:
        cnx = get_db_connection()
        cursor = cnx.cursor()
        query = ("SELECT RestaurantID, RestaurantName, Boro, CuisineDescription, InspectionAvg "
                 "FROM Restaurants r JOIN Cuisine c USING (CuisineID) "
                 "ORDER BY RestaurantID "
                 "LIMIT 10")
        cursor.execute(query)
        rows = cursor.fetchall()

        return jsonify(rows), 200

    except Exception as e:
        # Log the error (optional, but recommended for debugging)
        print(f"Database error: {e}")

        # Return a JSON error message and a 500 status code
        return jsonify({"error": "Internal Server Error", "message": str(e)}), 500

    finally:
        # Ensure the connection is closed even if an error occurs
        if 'cursor' in locals():
            cursor.close()
        if 'cnx' in locals():
            cnx.close()


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

Add a new route /restaurants to list the top 10 restaurants by ID

Open connection and get cursor

SQL query for data

Execute query and get all results

Return json data to caller with status 200 OK

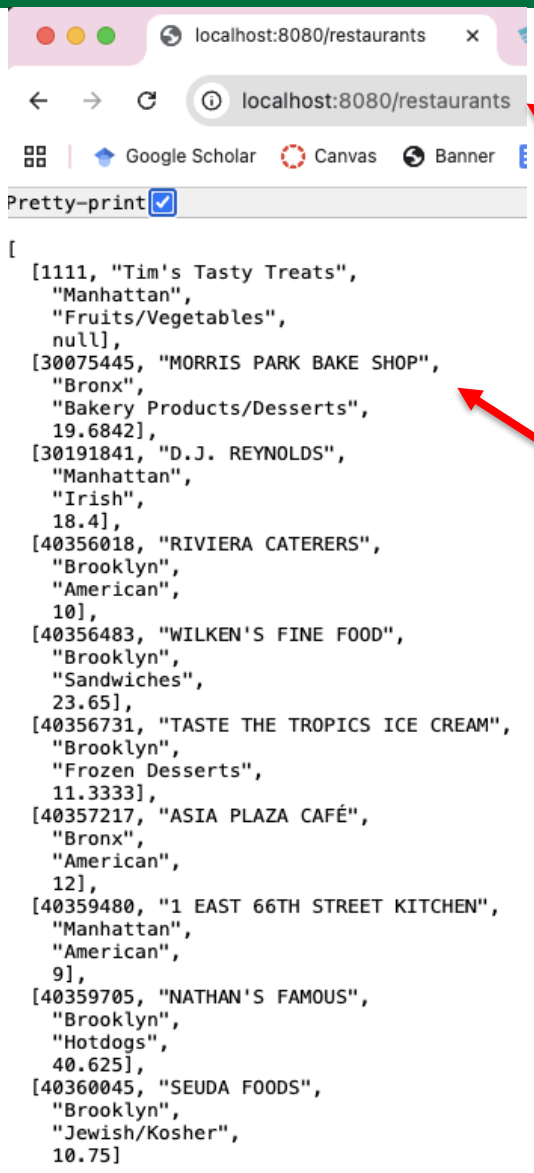
Handle exceptions, return status 500 Server Error

Close cursor and connection

NOTE: only GET requests are allowed, others blocked

Cannot make POST, PUT, or DELETE request

Try the new route from a browser



New route:
http://localhost/restaurants

JSON data returned from data according to SQL

All web browsers, mobile apps, thick client apps can call this route for data

All SQL is in one place

Clients know they can call the API and will get data

Clients do not need to know anything about the database

Try the new route from a client-side Python program

```python
import requests

'''
Demo to show that we can call an API from Python.

Author: Tim Pierson, Dartmouth CS61, Spring 2026
        Modified from Gemini example
'''


if __name__ == "__main__":
    # Replace with your actual local or hosted URL
    url = "http://localhost:8080/restaurants"

    try:
        # 1. Send the GET request
        response = requests.get(url)

        # 2. Check the status code we set in the Flask route
        if response.status_code == 200:
            restaurants = response.json() # Automatically parses the 'jsonify' data
            print(f"Retrieved {len(restaurants)} restaurants:")
            for r in restaurants:
                print(f"ID: {r[0]} | Name: {r[1]} | Cuisine: {r[3]}")

        elif response.status_code == 500:
            print("Server Error:", response.json().get("message"))

        else:
            print(f"Unexpected Error: {response.status_code}")

    except requests.exceptions.ConnectionError:
        print("Could not connect to the server. Make sure your Flask app is running!")

```

Use Python *requests* library to make a HTTP GET request from the API

Make HTTP GET request and see API response

If response == 200 OK Print results

Try the new route from a client-side Python program

```python
import requests

'''
Demo to show that we can call an API from Python.

Author: Tim Pierson, Dartmouth CS61, Spring 2026
        Modified from Gemini example
'''


if __name__ == "__main__":
    # Replace with your actual local or hosted URL
    url = "http://localhost:8080/restaurants"

    try:
        # 1. Send the GET request
        response = requests.get(url)

        # 2. Check the status code we set in the Flask route
        if response.status_code == 200:
            restaurants = response.json() # Automatically parses the 'jsonify' data
            print(f"Retrieved {len(restaurants)} restaurants:")
```

Use Python *requests* library to make a HTTP GET request from the API

Make HTTP GET request and see API response

If response == 200 OK
Print results

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

d84607y@MacBook-Pro-5 flaskExample % python3 call_api.py
Retrieved 10 restaurants:
ID: 1111 | Name: Tim's Tasty Treats | Cuisine: Fruits/Vegetables
ID: 30075445 | Name: MORRIS PARK BAKE SHOP | Cuisine: Bakery Products/Desserts
ID: 30191841 | Name: D.J. REYNOLDS | Cuisine: Irish
ID: 40356018 | Name: RIVIERA CATERERS | Cuisine: American
ID: 40356483 | Name: WILKEN'S FINE FOOD | Cuisine: Sandwiches
ID: 40356731 | Name: TASTE THE TROPICS ICE CREAM | Cuisine: Frozen Desserts
ID: 40357217 | Name: ASIA PLAZA CAFÉ | Cuisine: American
ID: 40359480 | Name: 1 EAST 66TH STREET KITCHEN | Cuisine: American
ID: 40359705 | Name: NATHAN'S FAMOUS | Cuisine: Hotdogs
ID: 40360045 | Name: SEUDA FOODS | Cuisine: Jewish/Kosher
```

Try the new route from a client-side Python program

```python
import requests

'''
Demo to show that we can call an API from Python.

Author: Tim Pierson, Dartmouth CS61, Spring 2026
        Modified from Gemini example
'''


if __name__ == "__main__":
    # Replace with your actual local or hosted URL
    url = "http://localhost:8080/restaurants"

    try:
        # 1. Send the GET request
        response = requests.get(url)

        # 2. Check the status code we set in the Flask route
        if response.status_code == 200:
            restaurants = response.json() # Automatically parses the 'jsonify' data
            print(f"Retrieved {len(restaurants)} restaurants:")
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```
d84607y@MacBook-Pro-5 flaskExample % python3 call_api.py
Retrieved 10 restaurants:
ID: 1111 | Name: Tim's Tasty Treats | Cuisine: Fruits/Vegetables
ID: 30075445 | Name: MORRIS PARK BAKE SHOP | Cuisine: Bakery Products/Desserts
ID: 30191841 | Name: D.J. REYNOLDS | Cuisine: Irish
ID: 40356018 | Name: RIVIERA CATERERS | Cuisine: American
ID: 40356483 | Name: WILKEN'S FINE FOOD | Cuisine: Sandwiches
ID: 40356731 | Name: TASTE THE TROPICS ICE CREAM | Cuisine: Frozen Desserts
ID: 40357217 | Name: ASIA PLAZA CAFÉ | Cuisine: American
ID: 40359480 | Name: 1 EAST 66TH STREET KITCHEN | Cuisine: American
ID: 40359705 | Name: NATHAN'S FAMOUS | Cuisine: Hotdogs
ID: 40360045 | Name: SEUDA FOODS | Cuisine: Jewish/Kosher
```

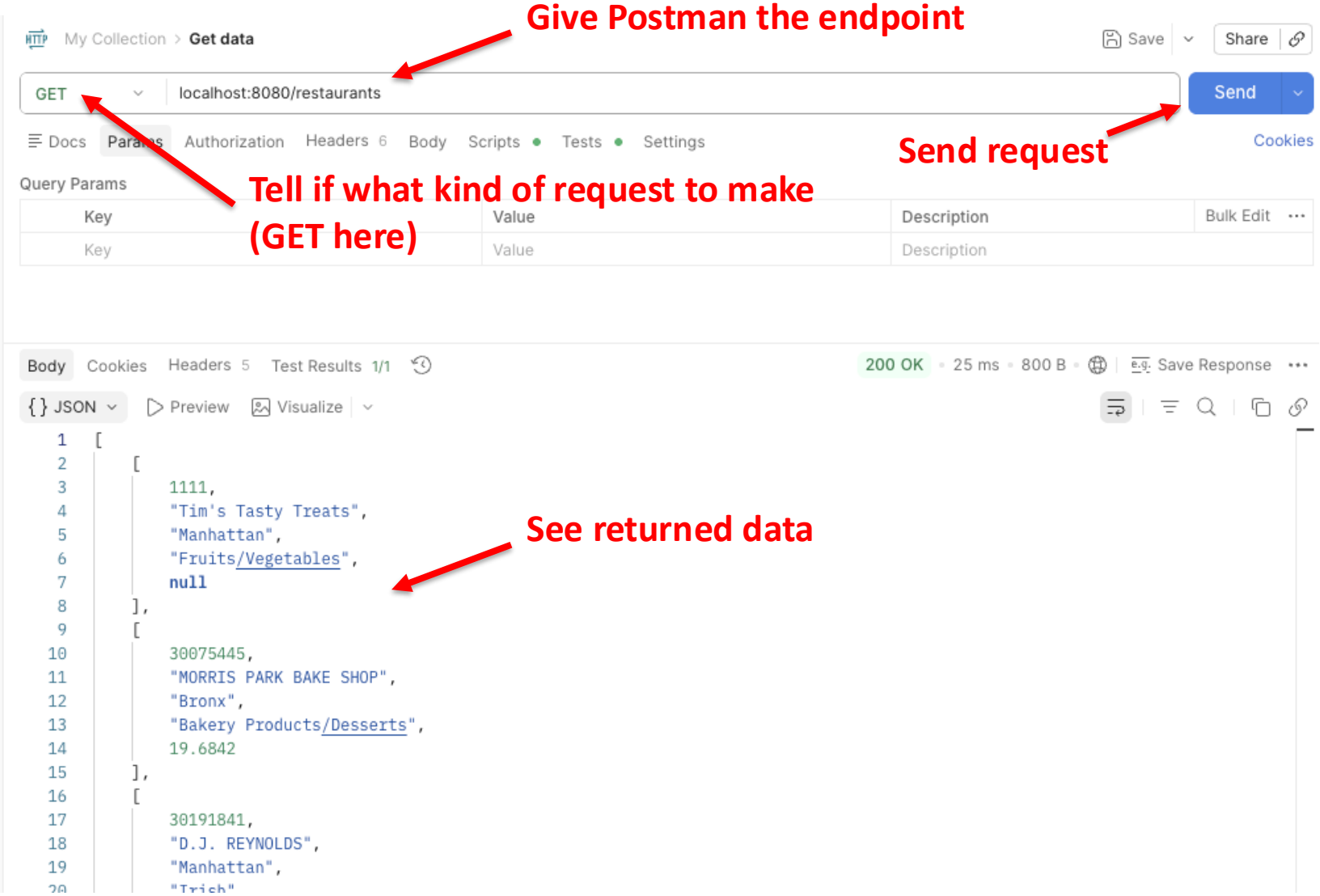
Big picture:
SQL is now in one place, the API

API can be called by browser or Python program (or other programs

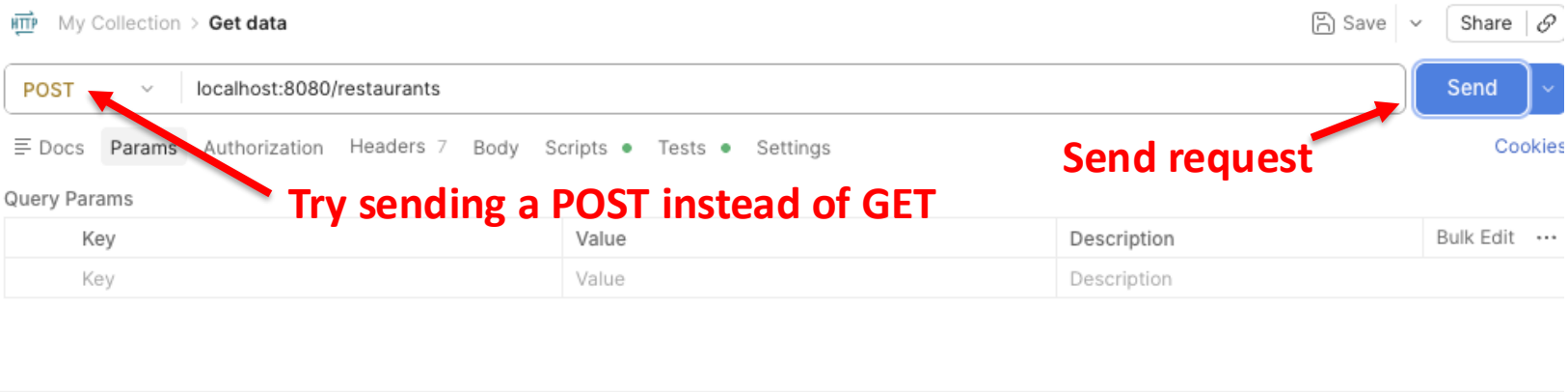
If update SQL, just update the API route and all users are updated

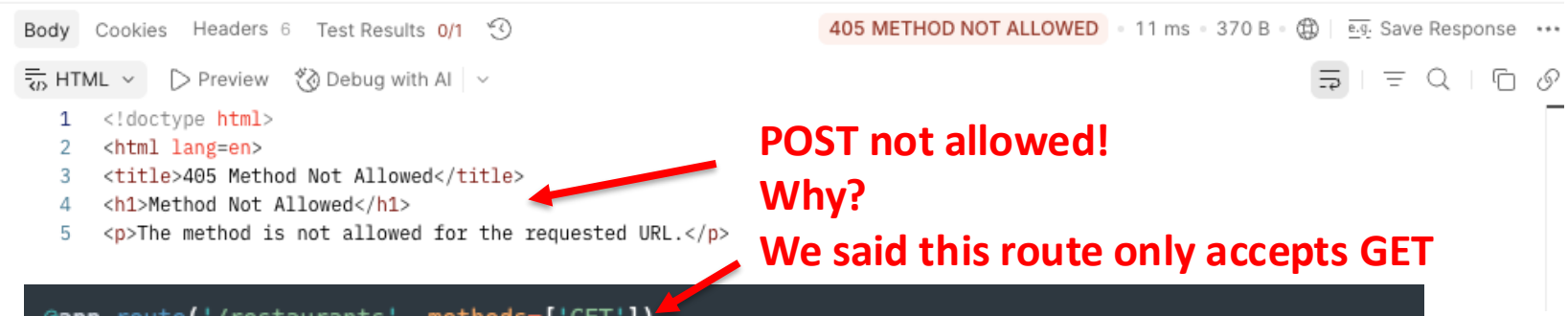
Could you change database and clients would not notice?
Yes!
Just change routes (endpoints)

Postman is a great debugging tool!



Postman lets you easily send GET, POST, PUT, and DELETE requests






```python
@app.route('/restaurants', methods=['GET'])
def get_restaurants():
    try:
        cnx = get_db_connection()
        cursor = cnx.cursor()
        query = ("SELECT RestaurantID, RestaurantName, Boro, CuisineDescription, InspectionAvg "
                 "FROM Restaurants r JOIN Cuisine c USING (CuisineID) "
                 "ORDER BY RestaurantID "
                 "LIMIT 10")
        cursor.execute(query)
        rows = cursor.fetchall()

        return jsonify(rows), 200
```

Add a GET route that takes a parameter

Get data for one restaurant by ID **https://localhost:8080/restaurants/1111**

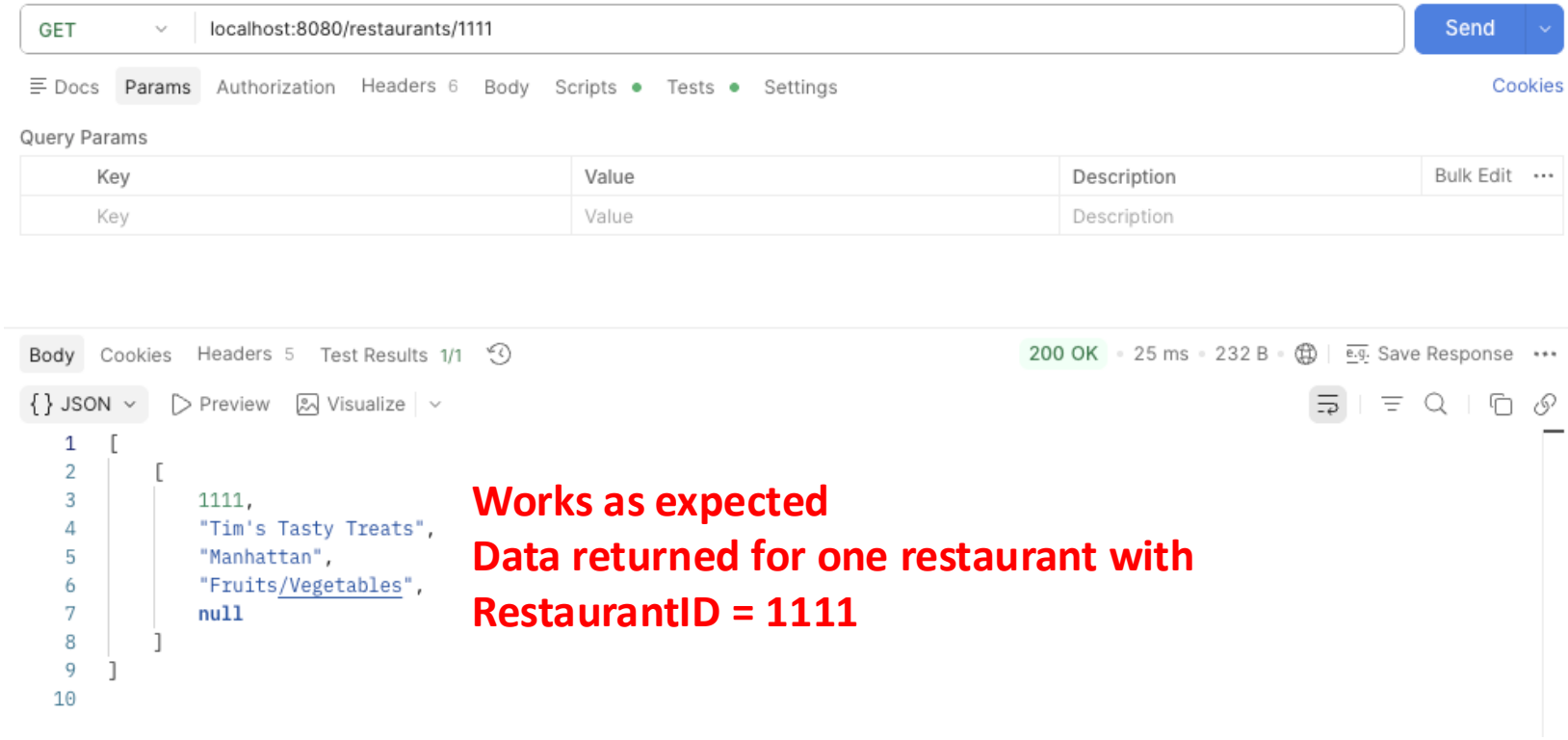
```python
@app.route('/restaurants/<int:id>', methods=['GET'])
def get_restaurant(id):
    try:
        cnx = get_db_connection()
        cursor = cnx.cursor(prepared=True)  # use a prepared statement to avoid SQL injection!
        query = ("SELECT RestaurantID, RestaurantName, Boro, CuisineDescription, InspectionAvg "
                 "FROM Restaurants r JOIN Cuisine c USING (CuisineID) "
                 "WHERE RestaurantID = %s")
        cursor.execute(query, (id,))
        rows = cursor.fetchall()

        return jsonify(rows), 200

    except Exception as e:
        # Log the error (optional, but recommended for debugging)
        print(f"Database error: {e}")

        # Return a JSON error message and a 500 status code
        return jsonify({"error": "Internal Server Error", "message": str(e)}), 500

    finally:
        # Ensure the connection is closed even if an error occurs
        if 'cursor' in locals():
            cursor.close()
        if 'cnx' in locals():
            cnx.close()
```

id will be 1111 here

Code like restaurants route (the one without ID)

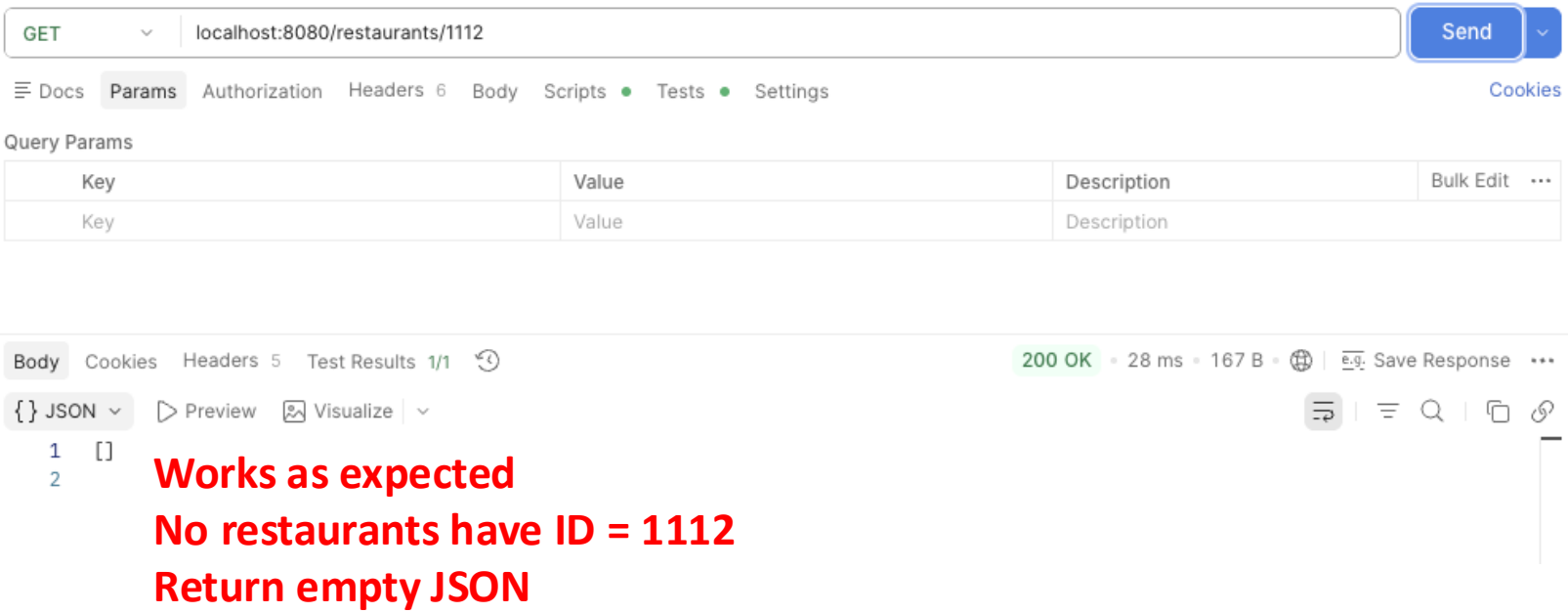
- **SQL only gets data for RestaurantID=id**
- **NOTE: we use a parameterized query for security (more on that next class)**

Test with Postman using RestaurantID=1111



Works as expected
Data returned for one restaurant with RestaurantID = 1111

Test with Postman using RestaurantID=1112 (not a restaurant)



Works as expected

No restaurants have ID = 1112

Return empty JSON

Add a POST route to add a new restaurant

```python
# POST a new restaurant
@app.route('/restaurants', methods=['POST'])
def add_restaurant():
    try:
        data = request.get_json() #data provided by client
        # Basic validation
        if not data:
            return jsonify({"error": "No data provided"}), 400 # status 400 = Bad Request
        if data['Boro'] not in ['Manhattan', 'Brooklyn', 'Queens', 'Bronx', 'Staten Island']:
            return jsonify({"error": "Invalid boro"}), 400 # status 400 = Bad Request

        cnx = get_db_connection()
        cursor = cnx.cursor(prepared=True)
        query = ("INSERT INTO Restaurants (RestaurantID, RestaurantName, Boro, Building, Street, "
            "ZipCode, Phone, Latitude, Longitude, CuisineID) "
            "VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)")
        values = (data['RestaurantID'], data['RestaurantName'], data['Boro'], data['Building'], data['Street'],
            data['ZipCode'], data['Phone'], data['Latitude'], data['Longitude'], data['CuisineID'],)
        cursor.execute(query, values)
        cnx.commit()
        new_id = cursor.lastrowid

        return jsonify({"id": new_id, "message": "Restaurant created"}), 201 #status 201 = Created
    except Exception as e:
        # Log the error (optional, but recommended for debugging)
        print(f"Database error: {e}")

        # Return a JSON error message and a 500 status code
        return jsonify({"error": "Internal Server Error", "message": str(e)}), 500 #status 500 = Server Error

    finally:
        # Ensure the connection is closed even if an error occurs
        if 'cursor' in locals():
            cursor.close()
        if 'cnx' in locals():
            cnx.close()
```

POST requests to http://localhost:8080/restaurants sent to this route

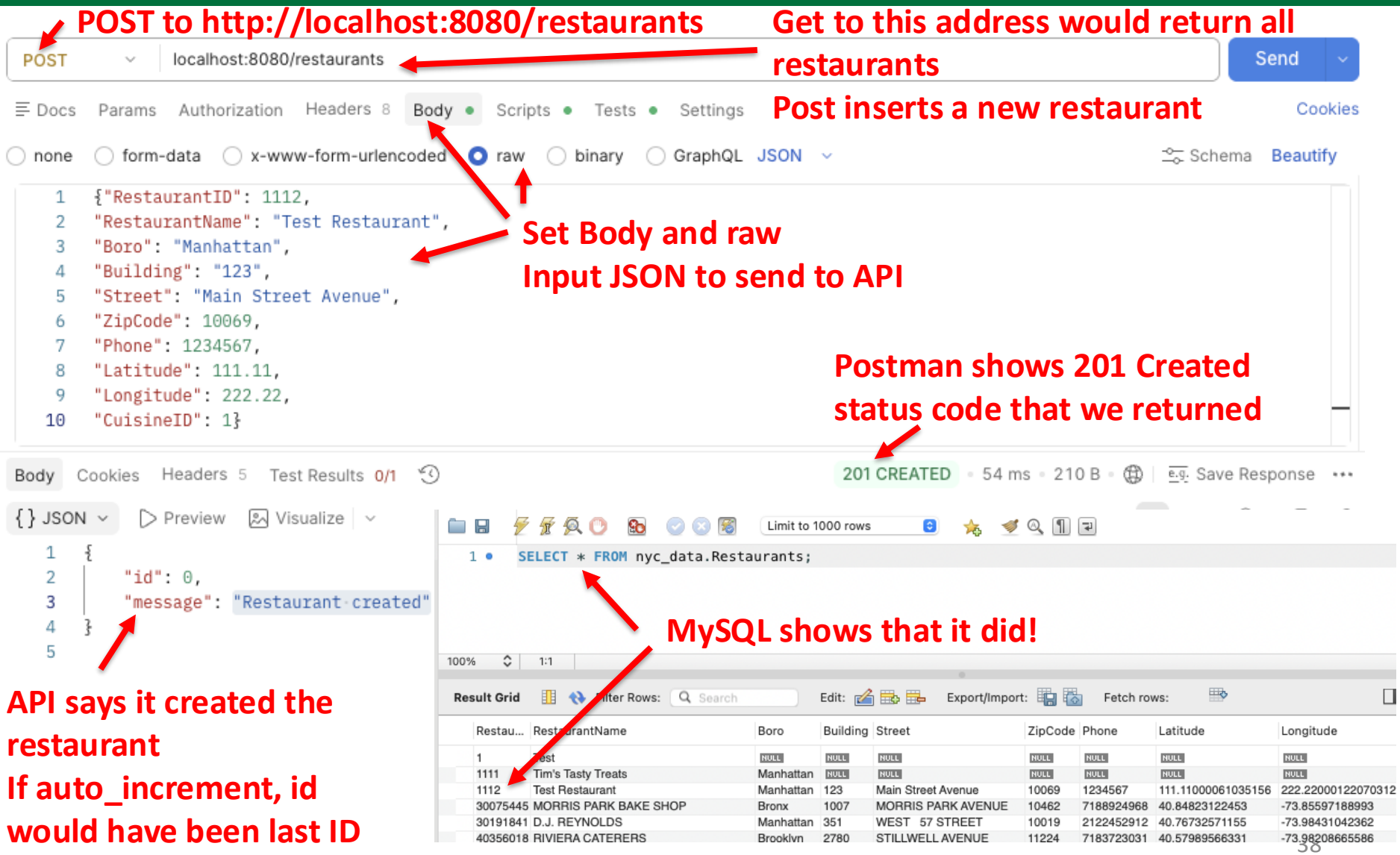
Get JSON data sent by client

Some *basic* validation checks here in boro

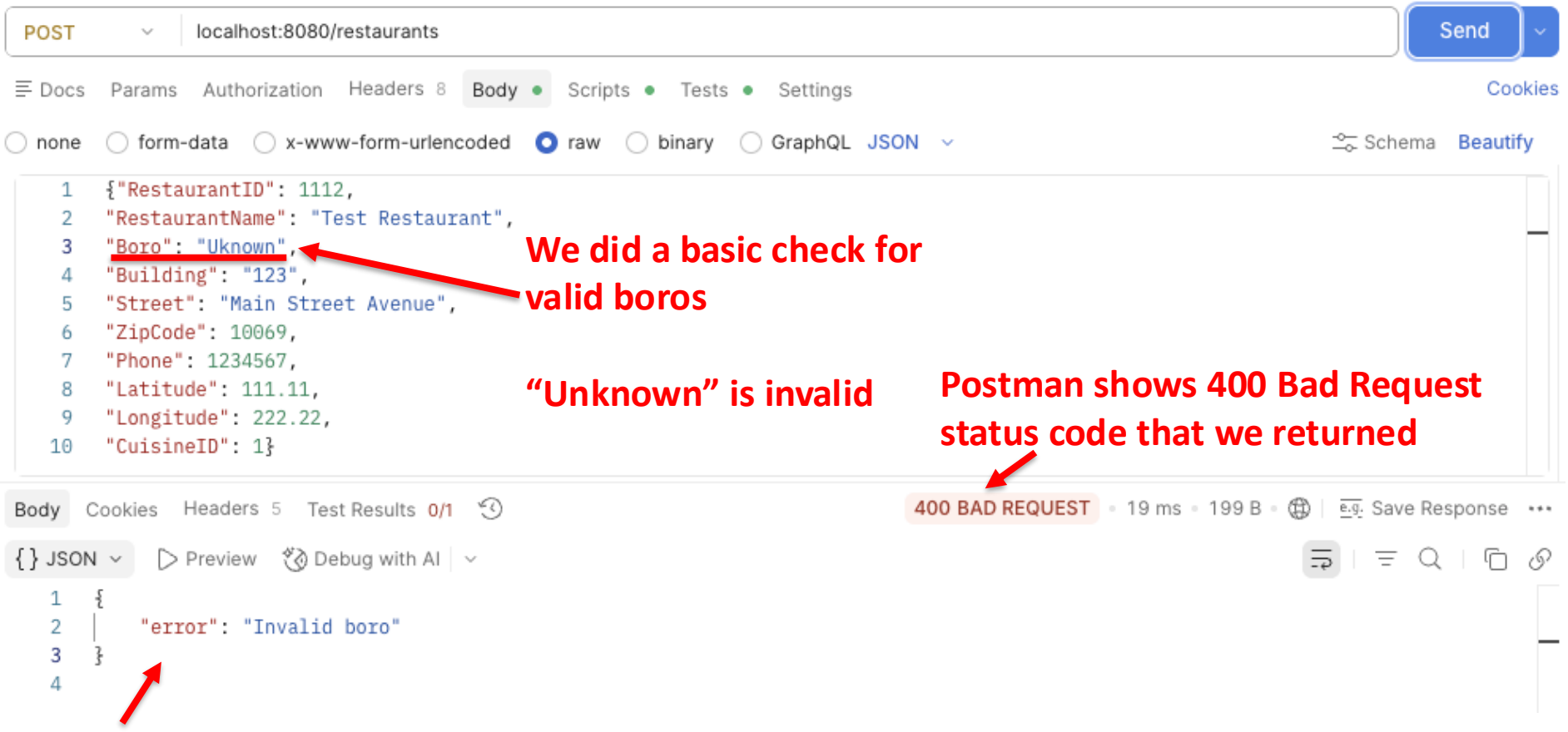
Execute SQL INSERT statement and commit

Auto_increment would have return the last ID, but our table is not auto_increment

Test with Postman using valid data



Test with Postman with invalid Boro



POST using Python to create new restaurant by sending data in the body

```python
url = "http://localhost:8080/restaurants"

try:
    # 1. Send the POST request
    data = {"RestaurantID": 1112,
            "RestaurantName": "Test Restaurant",
            "Boro": "Manhattan",
            "Building": "123",
            "Street": "Main Street Avenue",
            "ZipCode": 10069,
            "Phone": 1234567,
            "Latitude": 111.11,
            "Longitude": 222.22,
            "CuisineID": 1
            }
    response = requests.post(url, json=data)

    # 2. Check the status code we set in the Flask route
    if response.status_code == 201:
        resp = response.json() # Automatically parses the 'jsonify' data
        print(f"Received: {resp}")

    elif response.status_code == 500:
        print("Server Error:", response.json().get("message"))

    else:
        print(f"Unexpected Error: {response.status_code}")

except requests.exceptions.ConnectionError:
    print("Could not connect to the server. Make sure your Flask app is running!")
```

Set JSON parameters to send to server

Make POST sending data to create new restaurant

Expect status code 201

Practice

Create a DELETE route for /restaurants/:id

For example: localhost:8080/restaurants/1112

- Get the ID to delete
- Create a SQL command that will delete a restaurant with that ID
- Execute SQL command
- cnx.commit to finalize
- Get how many rows where affected with:
  affected = cursor.rowcount
- Return a JSON message:
  - If no rows were affected:
    - {"error": "Restaurant not found"}
    - Return Not Found status code
  - Else:
    - {"message": "Restaurant deleted"}
    - Return OK status code

For next time: Should anyone be able to call this route?