Please review **Module 3** before starting this tutorial.

Understanding the Basics

CRUD Operations: The four fundamental operations of persistent storage are commonly referred to as Create, Read, Update, and Delete (CRUD). In the context of relational databases, the **C**reate, **R**ead, **U**pdate, and **D**elete (CRUD) operations are associated with specific SQL statements, namely INSERT, SELECT, UPDATE, and DELETE, respectively. The HTTP protocol offers a variety of methods that can be correlated with the CRUD operations, including POST, GET, PUT, and DELETE. In this context, when provided with a Uniform Resource Identifier (URI) such as http://localhost:3000/Books, it is possible to utilize the method specified in the HTTP request message to determine the appropriate operation to be implemented in accordance with the principles of Representational State Transfer (REST). This is the core principle of the REST architectural style in which a URI uses its methods to identify the intended operation rather than having a dedicated URI for each operation.

Resource Identifiers and REST Services: Every **resource identifier** has the potential to be uniquely distinct toward a specific service. In a nutshell, a single REST service has the capability to offer various resource identifiers that can be utilized by its **service consumers**. Consider, for example, a REST service called "Books". The responsibility of this service would be the encapsulation of resources associated with books. A resource identifier facilitating the retrieval of book data may potentially resemble **http://localhost:3000/Books**. A **method** refers to a particular type of function that is offered through a uniform contract for the purpose of handling resource identifiers and data.

In general, a REST **service capability** can be conceptualized as the combination of a uniform contract method and a resource identifier. Moreover, a REST **service contract** serves to construct a functional structure that contains one or more REST service capabilities that are associated to that functional context. The approach of combining a number service contracts is commonly employed to produce a **Web API**, which acts as a programmatic interface to a system that can be accessed using standard HTTP methods and headers. Web Application Programming Interfaces (APIs), together with the web services they have become vital components for facilitating the exchange of data between web-based applications. The following are examples of service contracts, encompassing a range of service capabilities provided under every contract.

- **HTTP GET** | **http://localhost:3000/Books**
    1. The utilization of the HTTP GET method facilitates the retrieval of a complete set of books, similar to executing the SQL query "**SELECT * FROM** …"
- **HTTP DELETE** | **http://localhost:3000/Books**
    1. The HTTP DELETE method is utilized to remove all books from the database, which is analogous to the "**DELETE FROM** ..." operation in SQL.
- **HTTP GET** | **http://localhost:3000/Books/ISBN/12345**
    1. The HTTP GET method is employed to acquire specific information of a book that corresponds to an ISBN of 12345. This operation is analogous to executing the SQL query "SELECT * FROM ... WHERE ISBN = 12345".
- **HTTP DELETE** | **http://localhost:3000/Books/ISBN/12345**
    1. The utilization of the HTTP DELETE method facilitates the deletion of specific book details (or record) associated with an ISBN of 12345, analogous to executing the SQL query "DELETE FROM ... WHERE ISBN = 12345".
- **HTTP POST** | **http://localhost:3000/Books/ISBN/123456**
    1. The use of the HTTP POST method enables the addition of a new book (or record). The supply of data for the new book (or record) is often provided within the body of the HTTP POST request message. This body contains the data, for example, in JSON format. The HTTP POST method is similar to the SQL statement "INSERT INTO ... VALUES (...)" in terms of functionality and purpose.

MySQL (or MariaDB): The open-source MySQL database engine uses relational database management system (DBMS) fundamentals. SQL, an acronym for Structured Query Language, is a widely used language employed for the purpose of constructing and manipulating databases. Developers can create structured tables with **columns** (or **fields**) and **records** using a DBMS. Columns represent entity characteristics, while records represent instances of these entities. **MySQL**, MariaDB, Oracle DB, Microsoft SQL Server, PostgreSQL, and SQLite are among the many DBMSs that exist.

Adopted From: Randy Connolly, Ricardo Hoar, "Fundamentals of Web Development" (2 Edition), 2017

Example of **Artist** table representing an **Artist** Entity

Primary key field

Fields

Field names

Records

ArtWorkID	Title	Artist	YearOfWork
345	The Death of Marat	David	1793
400	The School of Athens	Raphael	1510
408	Bacchus and Ariadne	Titian	1520
425	Girl with a Pearl Earring	Vermeer	1665
438	Starry Night	Van Gogh	1889

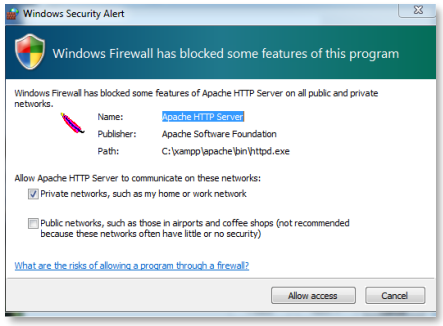
A primary key in a table indicates each record's uniqueness. An Artist table's ArtistID column is a primary key, ensuring that each artist has a unique identification. SQL simplifies writing data operations statements by abstracting data processing details. The **INSERT** command creates a record. A **SELECT** command retrieves one or more records. The **UPDATE** command modifies records. The **DELETE** command deletes records. The instructions follow the **CRUD** operations. Module 3 | Lecture 12 covers databases in particular. Click this link for database and SQL details. We will use a backend database to define HTTP CRUD procedures for our intended web services in this activity. We will setup and use a database using **MySQL** and **phpMyAdmin**. The open-source software stack XAMPP comprises all the libraries, programs, and tools for database development and management.

Getting Started

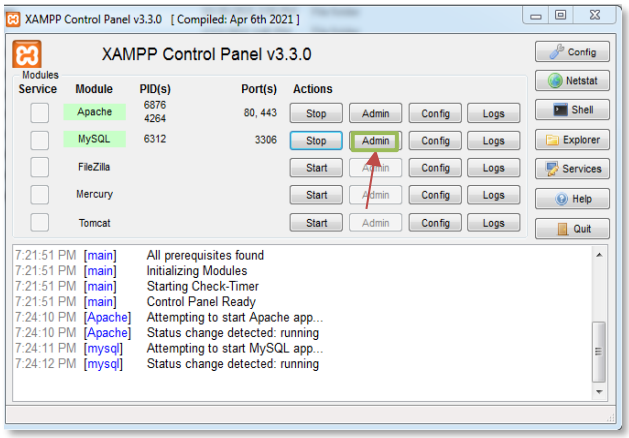
We will be using **Visual Studio Code** for the purpose of this activity. Please ensure that the latest Visual Studio Code is installed on your computing device. Some extensions or tools will be required for completing some of the steps in this activity.

PART I: SETTING UP A MYSQL DATABASE AND PHPMYADMIN INTERFACE

A. Downloading and Installing XAMPP Stack (Apache, MariaDB, PHP and Perl)

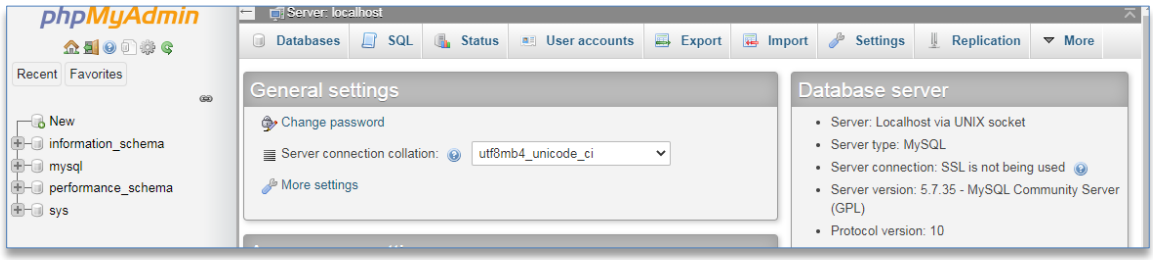


1. Using your browser, go to https://www.apachefriends.org/download.html
2. **Download** the relevant file for your OS (will use Windows OS for illustration)
   We will use **version 8.0.28 / PHP 8.0.28** for this activity
3. **Install** the XAMPP stack. XAMPP installation may prompt warnings about antivirus software and User Account Control. To continue installation, click "**Yes**" or "**OK**". Keep the default settings.
4. You may receive a pop-up window from the OS firewall to allow Apache Server to communicate on local private network. Click "**Allow access**" button.
5. The setup wizard will end with a panel named "Completing the XAMPP Setup Wizard." **Keep** "Do you want to start the Control Panel now?" **selected**. Then, click "**Finish**". Clicking this option launches the XAMPP control panel.
6. Click the **Start** button beneath the **Actions** column next to the following items to initiate the necessary components: (a) **Apache** is a popular open-source web server for backend services, and (b) **MySQL** (a popular relational DBMS for data management). Apache and MySQL will start in the background. Module port numbers (e.g., MySQL operates on port 3306) should be noted.
7. The **Admin** button will appear once MySQL is started. Click MySQL's "**Admin**" button. A browser window with **phpMyAdmin** should open automatically. PHP-based **phpMyAdmin** manages MySQL (or MariaDB) databases. Out task now is to create a simple database.

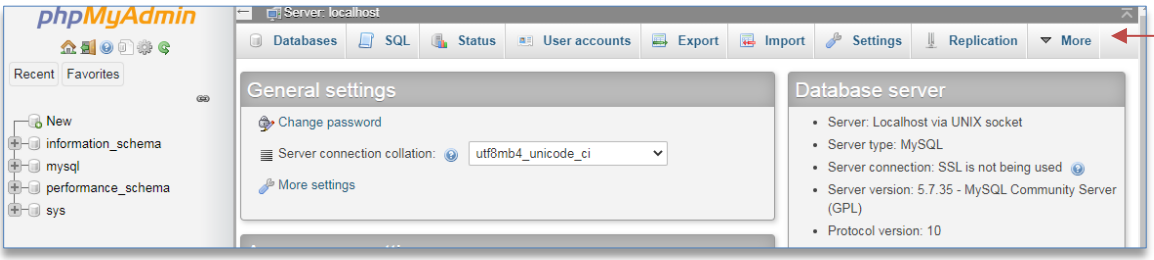


B. Creating a Backend Database

After completing Part A, you should be able to view the main phpMyAdmin interface as shown below.



The **left menu** displays all system databases built by default when MySQL is installed on an OS (e.g., Linux).



The **top menu** provides multiple MySQL DBMS features. Some of these menu items' primary features are listed below. Browse these options to learn about phpMyAdmin's tools or services.
Databases: manage existing databases (and also create new ones)

- **SQL**: allows you to run interactive SQL queries via simple HTML textbox
- **User accounts**: enables DBAs to create user accounts on the DBMS.
- **Export**: export databases from server into many formats such as SQL, JSON, YAML, CSV, etc.
- **Import**: imports databases from other sources such as SQL, spreadsheets, etc.

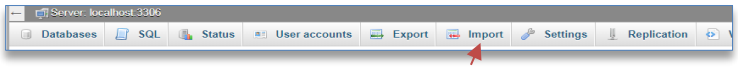
A database can be created by selecting the "New" option from the left navigation menu. The process is easy to use, allowing users to create tables as well as populate data directly through the interface. This eliminates the need to acquire knowledge or write SQL statements or queries. In order to facilitate this activity, we will proceed with the importing of a pre-existing SQL script that contains the database.

1. Our goal is to build a database to which a pre-existing database has been developed. This database stores 2023 population data estimates for major cities in Washington State. The database includes columns for city, population, population density, average people per square mile, and others. **Download "tut5-db.sql"** from Canvas | Module 3 | Tutorial 5 and save it locally. Use VSCode or an alternate code editor to inspect "tut5-db.sql". Close the file when done. Using phpMyAdmin, we would like to (a) **import tut5-db.sql** and then (b) **create** a user to access this database.

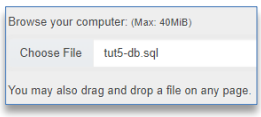
   As a preliminary step, it is necessary to be in the phpMyAdmin the main panel. Click on the **phpMyAdmin logo** or home icon in the top left corner of the interface.

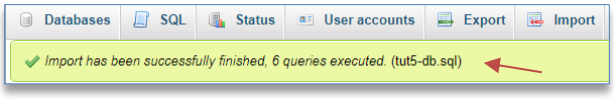
2. To import a database, click on **Import** from the top phpMyAdmin menu.

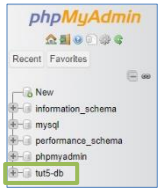
   

3. A webpage titled "**Importing into the Current Server**" will be shown. Within the "File to import" container, click the "**Choose File**" button and navigate to the location on your local machine where the "**tut5-db.sql**" file is saved. Scroll down and then click "**Go**". Keep all settings as default; no changes are needed.

4. Upon successful importing of the database, a notification indicating the successful completion of the import process should be displayed.

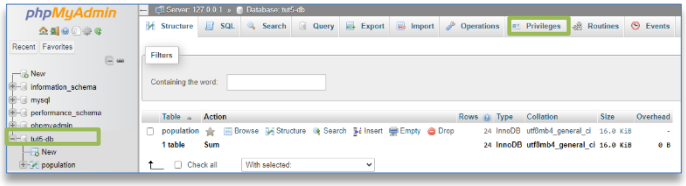
   



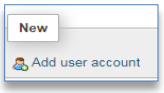
Additionally, from within the left menu of phpMyAdmin, one should now be able to view existing databases stored within MySQL. Access the '**tut5-db**' database by clicking on it from the left panel.

Our next step is creating an **external user** with the credentials and permissions needed to execute SQL queries from a web script, API, or program other than phpMyAdmin. A DBMS allows a database administrator to specify which queries an external user is able to perform. To associate a user with our '**tut5-db**' database, click on the database name to ensure you are the correct database session or environment.

5. To **add a user**, click on the '**tut5-db**' database after it has been created. Then, from the top menu, click on **Privileges**.



6. Click on "**Add user account**."



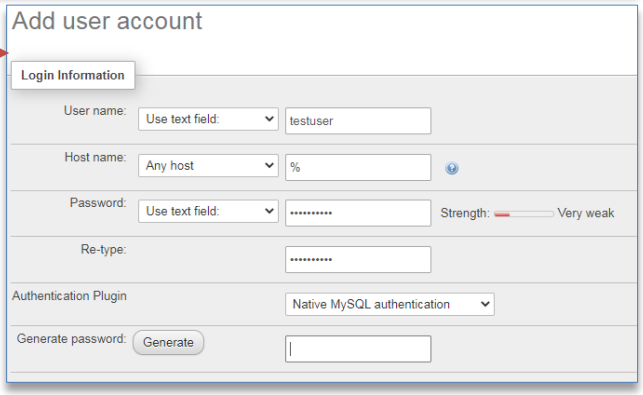
7. In the "Add user account" page, enter the following into the Login information container:

User name (use text filed): **testuser**

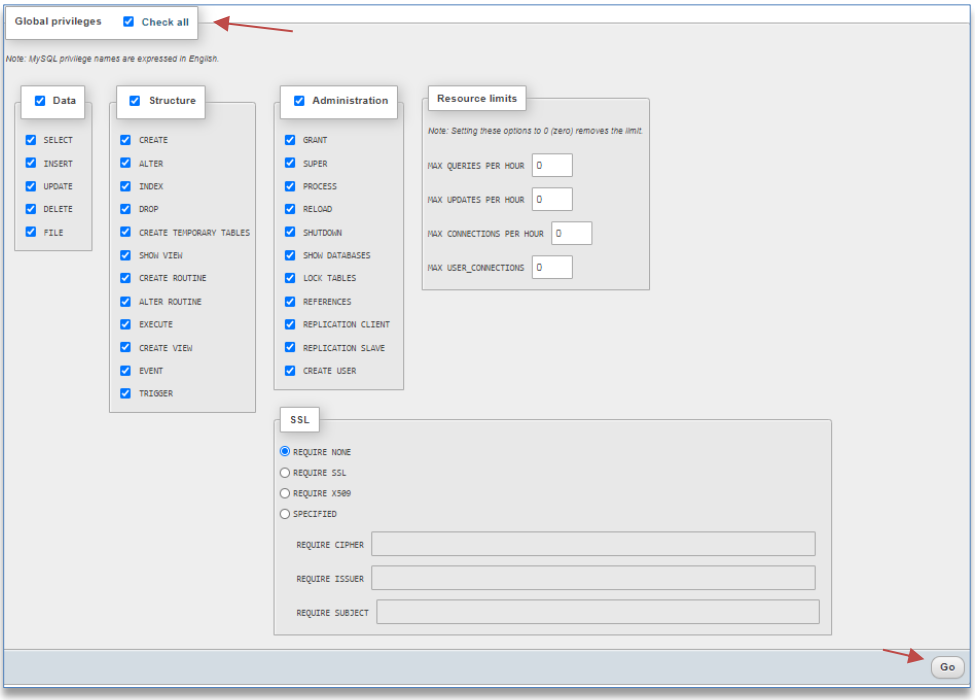
Host name (Any host): **%**

Password (use text filed): **mypassword**

Re-Type: **mypassword**



8. View each user account setup's options below. A database administrator or DBA can fully control database user actions. DBAs can identify user privileges in the Data section. The Structure allows DBAs restrict definition commands for users. The Administration component enables DBAs specify user access privileges. In the "**Global privileges**" section, please click on the option "**Check all**". Next, proceed by scrolling down the page and click on the '**Go button**'.



We are now ready to begin creating the backend REST web services that will interact with the database.

PART II: CREATING A NODE.JS BACKEND RESTFUL WEB SERVICE API

A. Create a Backend Node.js Workspace in Visual Studio Code (VSCode)

1. On your local machine, create a folder named **tutorial5**. Then, inside this folder, create a sub-folder called **backend**.
2. Launch **VS Code.** Create a **workspace** and associate it with the **tutorial5** folder.
3. Open a terminal window (top menu | **Terminal** | **New Terminal**).
4. **Change directory** to the **backend** directory (using cd command).
5. The backend solution will serve as the backend web services responsible for performing the CRUD operations on the MySQL database. In order to start the process, it is essential to initialize and install the required libraries for our backend project. Hence, from the terminal window in VSCode **within the backend directory**, **type the following commands**:
   - `npm init -y`
   - `npm i express nodemon mysql cors`

   ➔ '**-y**' instructs npm generator to use default values rather than asking questions
   ➔ **express** is a backend library for building RESTful web APIs
   ➔ **nodemon** is a utility that automatically restarts the server when source code changes
   ➔ **mysql** is a library that will enable an application to access mysql database
   ➔ **cors** is a library that serves as a middleware for enabling various CORS options

```
12    "dependencies": {
13      "cors": "^2.8.5",
14      "express": "^4.18.2",
15      "mysql": "^2.18.1",
16      "nodemon": "^3.0.1"
17    }
```
package.json after installing libraries

B. Create a Node.js Configuration Module for Connecting to MySQL DBMS

Our objective in this section is to develop a Node.js module in that will configure and establish MySQL connections. A pre-existing JavaScript file has been developed and made available via Canvas.

1. **Download** the "**config.js**" file from the Canvas | Module 3 | Tutorial 5.
2. Then, **save the file in the backend directory**.
3. "**config.js**" file contains comprehensive documentation. Hence, **examine the code** in order to understand how it works.
4. In the terminal window, **enter the following command**: `node config.js`
5. Terminal should now be connected to the MySQL database and a message "**Backend is now connected to: tut5-db**."

   ⚠ **Having Connectivity Problems**? It is advisable to check the configuration parameters, specifically the username and password. Additionally, verify that the MySQL server is operational via XAMPP, ensuring that the port (default is 3306) is being employed.

6. **Press** the keyboard buttons **Ctrl + C** to stop running the config.js in the terminal window.

C. Building Backend REST Web Service API

The objective of this section is to develop a backend Node.js module that will serve as the REST web service API for managing the CRUD operations on the MySQL database. A pre-existing file called "**index.js**" has already been developed with initial code for creating a server using Express library.

1. **Download** the "**index.js**" file from the Canvas | Module 3 | Tutorial 5.
2. Then, **save the file** in the backend directory.
3. "**index.js**" file contains comprehensive documentation. Hence, **examine the code** in order to understand how it works.
4. In the terminal window, **enter the following command**: `node index.js`
7. Instead of using node command to run our backend files, we wish to automatically trigger Node.js to start our backend application once changes are applied to the source code, specifically **index.js**. Hence, using the **Explorer** section on the left panel, **double click on package.json** to open the file in the editor and update the scripts element (lines 8 & 9).

```
5.  "main": "index.js",
6.    "type": "module",
7.    "scripts": {
8.      "test": "echo \"Error: no test specified\" && exit 1" ,
9.      "start": "nodemon index.js"
```

⚠ **Do not forget to add a comma at the end of line 8**. The '**npm start**' command will now cause the utility **nodemon** to monitor changes in the "main" module (or index.js – line 5) to be reflected on the terminal window automatically.

5. In the terminal, every time we run index.js, the MySQL debug information is displayed. Hence, open "**config.js**" and turn off the debugging mode in the **mysqlConfig** parameters, as shown below (should be line 34):

```
debug: false // Connection debugging mode is OFF
```

6. Now, in the terminal window, type in the command `npm start`
The server will be listening to port 3000 and any changes made in index.js will automatically trigger restarting the Node.js application. This behavior is achieved via the **nodemon** library that we installed.

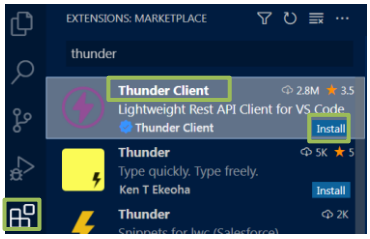
```
[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Express server is running and listening
Backend is now connected to: washingtondb.
```

7. Next, we would like to set up a route for the HTTP GET method in order to retrieve all the records included within the population table of the tut5-db database. In order to accomplish this task, it is necessary to incorporate the following code segment prior to Section C in index.js. **Save** the **index.js** file once completed.

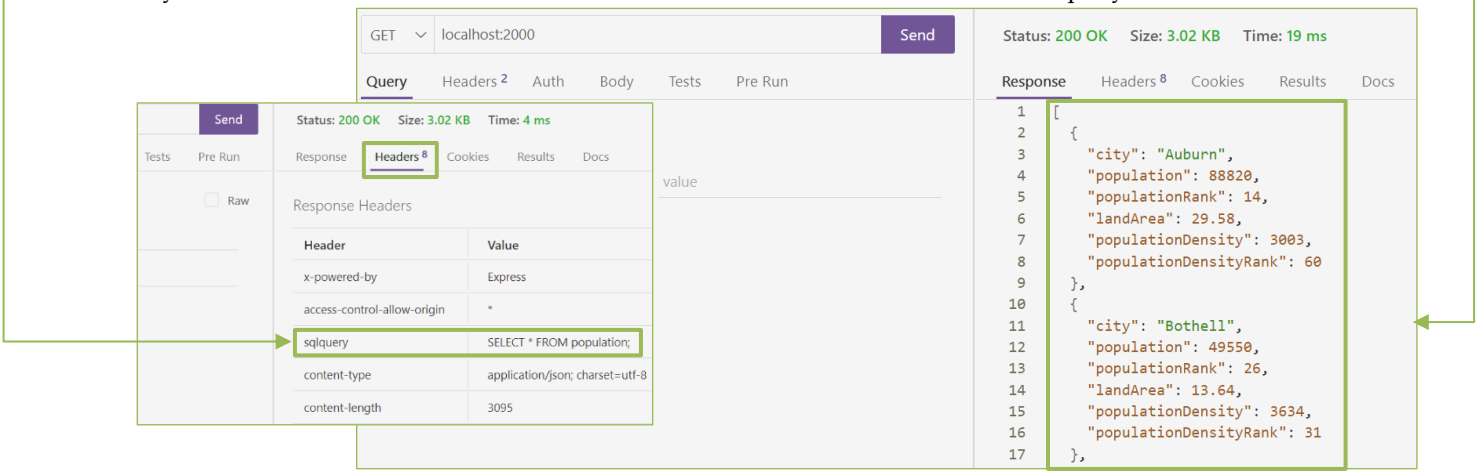
```
// ---------------------------------------------
// (1) Retrieve all records in population table
// root URI: http://localhost:port/
app.get('/', (request, response) => {
    const sqlQuery = "SELECT * FROM population;";
    dbConnection.query(sqlQuery, (err, result) => {
        if (err) {
            return response.status(400).json({Error: "Error in the SQL statement. Please check."});
        }
        response.setHeader('SQLQuery', sqlQuery); // send a custom header attribute
        return response.status(200).json(result);
    });
});
```

It should be noted that we are managing the HTTP response, ensuring that it includes the appropriate status code to indicate success (200) or error (400). Additionally, the response includes pertinent details such as an error message in the case of an error, or the actual result obtained from executing the SQL query which is stored within the result object. Furthermore, we include a header attribute named "SQL Query" which encapsulates the specific query employed for the given operation. All response outcomes have been programmed to be in JSON format.

8. We would like to test our first REST URI (or route). We can use the browser or preferably a HTTP client tool that can simulate or execute http client requests. Using VSCode, we would like to install an extension called "**Thunder Client**". From the left navigation menu, click on the extensions icon and search for that extension. Then, click Install to install it. A new icon on the menu will be added.



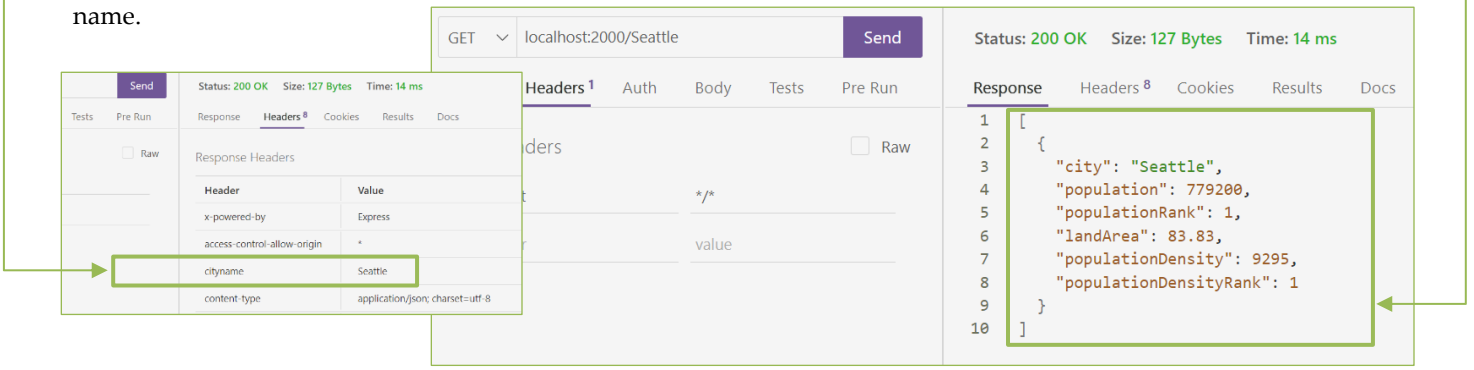
9. **Click on the Thunder Client** icon to activate it. Then, click on **New Request**.
10. In the address bar, **enter** http://localhost:**2000**/ and keep the HTTP method as **GET**. Then, click the **Send** button. You should then see the HTTP response on the right-hand side panel. Click on the **Headers** tab and you should also see the custom header we have created which contains the SQL query.



11. Let's create another route that will retrieve only one record by using a city name in the URI. Add the following code segment in index.js following the subsequent the code authored in Step 7.

```
// ---------------------------------------------
// (2) Retrieve one record by city name
// city URI: http://localhost:port/city
app.get('/:city', (request, response) => {
    const city = request.params.city;
    const sqlQuery = "SELECT * FROM population WHERE CITY = '" + city +"';";
    dbConnection.query(sqlQuery, (err, result) => {
        if (err) {
            return response.status(400).json({Error: "Error in the SQL statement. Please check."});
        }
        response.setHeader('CityName', city); // send a custom
        return response.status(200).json(result);
    });
});
```

12. Use Thunder Client to test the city URI from the previous step. In the address bar, **enter** http://localhost:3000/Seattle and keep the HTTP method as **GET**. Then, click the **Send** button. You should then see the HTTP response on the right-hand side panel. Click on the **Headers** tab and you should also see the custom header we have created which contains the city name.



13. Now, let's use a POST method instead of GET. The POST method can be used to create a new record in the database. To this extent, we need to construct the SQL statement that contains data for a new record (or city). Let's say we would like to add the population data for the city of Lacey. Our SQL statement should be as follows:

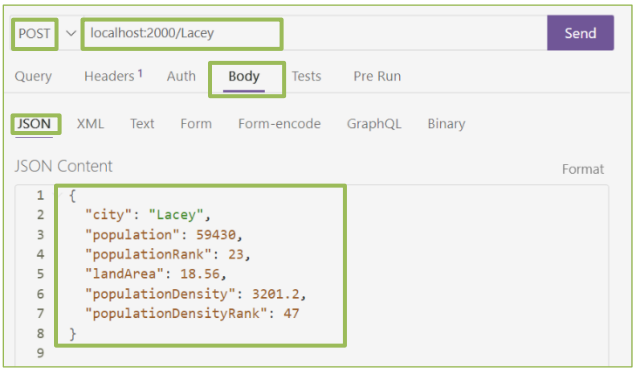
```
INSERT INTO POPULATION VALUES ('Lacey', 59430, 23, 18.56, 3201.2, 47);
```

Use http://localhost:3000/Lacey to check if the database has a record for the city Lacey. Response should be empty. Let's add Lacey's data to the database using HTTP POST. Hence, the same route (or URI) for city will be employed, but POST will be the HTTP method. The following code segment should be added to index.js following that from Step 11. Note that we would like to programmatically embed the data for the new record in the request body, not through the SQLQuery string directly.

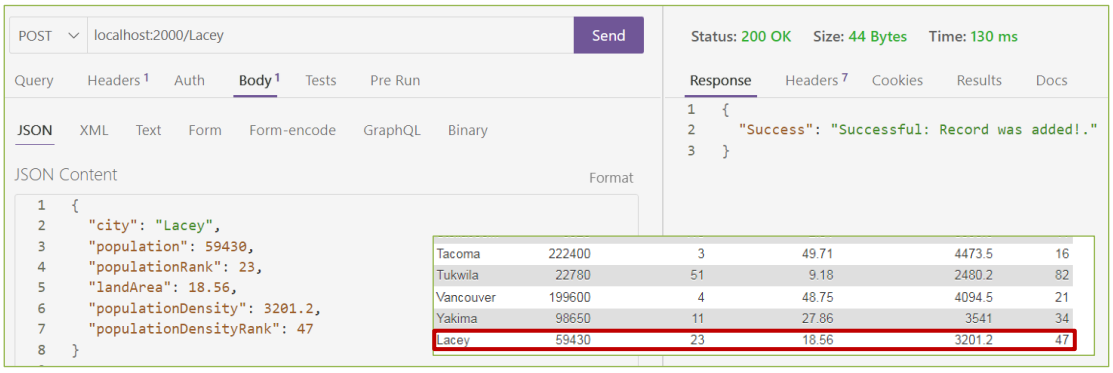
```
// ---------------------------------------------
// (3) insert a new record by city name
// city URI: http://localhost:port/city
app.post('/:city', (request, response) => {
    const sqlQuery = 'INSERT INTO POPULATION VALUES (?);';
    const values = [request.body.city, request.body.population, request.body.populationRank,
        request.body.landArea, request.body.populationDensity, request.body.populationDensityRank];
    dbConnection.query(sqlQuery, [values], (err, result) => {
        if (err) {
            return response.status(400).json({Error: "Failed: Record was not added."});
        }
        return response.status(200).json({Success: "Successful: Record was added!."});
    });
});
```

14. Use Thunder Client to test the city URI from the previous step using POST method. In the address bar, **enter** http://localhost:3000/Lacey and set the HTTP method as **POST**. Then, click on the **Body** tab and select **Json**. Then, in the textbox, enter the JSON body as shown below:

```
{
    "city" : "Lacey",
    "population" : 59430,
    "populationRank" : 23,
    "populationDensity": 18.56,
    "populationDensityRank": 3201.2,
    "landArea": 47
  }
```



15. Then, click the **Send** button. You should now see a HTTP 200 OK status. Complete request and response is shown below. You can also verify the result within the **phpMyAdmin** executing "SELECT * FROM Population".



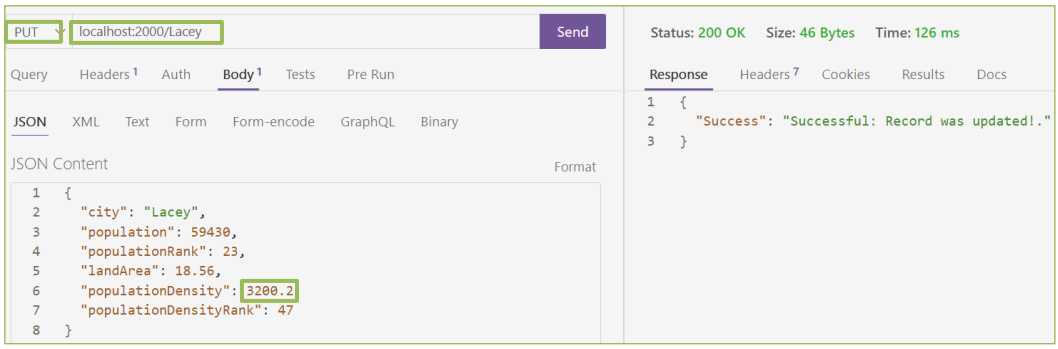
It should be noted that if the request is sent repeatedly, it will result in the addition of multiple records into the database. The presence of duplicate records in the database, resulting from the absence of a primary key, is the cause of this scenario. This presents a significant question what measures can be taken to prevent the insertion of a record in the event that it currently exists? Prior to processing a HTTP post, it is possible to use the URI http://localhost:3000/Lacey to examine the response. If response indicate the presence of a record (e.g., a non-empty response body), it is recommended to notify the client that the record already exists, rather than proceeding with a post request arbitrarily. From a development perspective, this increases the reusability of the functionalities or services that were implemented. The practical development of this can be achieved programmatically through the utilization of basic logic and control structures.

16. Now, let's use a **HTTP PUT method** instead, which can be used to **update an existing record**. The following code segment should be added to index.js following that from Step 13. For PUT, we will assume the client needs to supply all of the values, for simplicity.

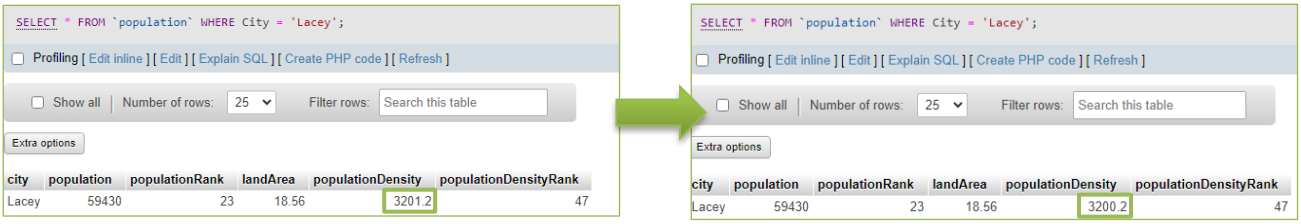
```
// ---------------------------------------------
// (4) update an existing record by city name
// city URI: http://localhost:port/city
app.put('/:city', (request, response) => {
    const city = request.params.city;
    const sqlQuery = `UPDATE POPULATION SET city = ?, population = ?,
    populationRank = ?, landArea = ?, populationDensity = ?, populationDensityRank = ?
    WHERE CITY = ? ;`;

    const values = [request.body.city, request.body.population, request.body.populationRank,
        request.body.landArea, request.body.populationDensity, request.body.populationDensityRank];

    console.log(sqlQuery); // for debugging purposes:
    dbConnection.query(sqlQuery, [...values, city], (err, result) => {
        if (err) {
            return response.status(400).json({Error: "Failed: Record was not added."});
        }
        return response.status(200).json({Success: "Successful: Record was updated!."});
        });
});
```

Note the **spread syntax** in **sqlQuery, [...values, city]** which expands the values array into individual elements. The values array expands into six elements and in addition to 'city', which are then used to replace the question marks in the value assigned to sqlQuery.

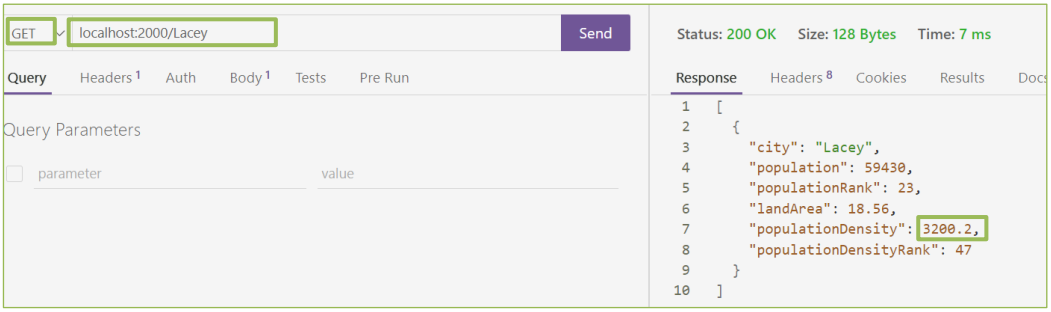
Below is a sample request via Thunder Client. The populationDesnsity value changes from **3201.2** to **3200.2**.



A phpMyAdmin inquiry before (left) and after (right) the successful REST service request using the PUT method.



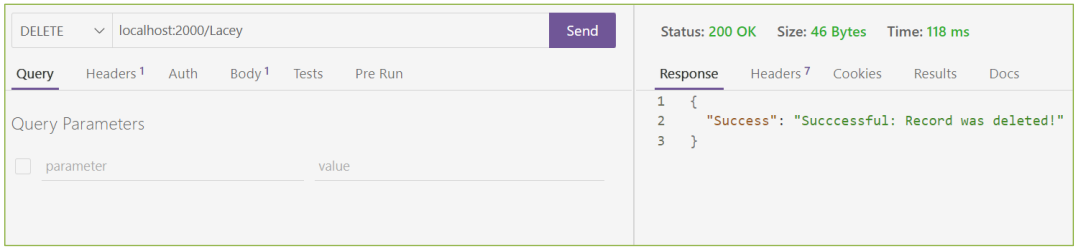
Below is running a HTTP GET request for retrieving the city data from the database.



17. Now, let's use a **HTTP DELETE method**, which can be used to **delete existing records**. The following code segment should be added to index.js following that from Step 13. For DELETE, we will attach a WHERE clause that will delete a single record only. A **DELETE FROM** without a WHERE clause will **delete all records** within the table!

```
// ------------------------------------------
// (5) Delete a record by city name
// city URI: http://localhost:port/city
app.delete('/:city', (request, response) => {
    const city = request.params.city;
    const sqlQuery = "DELETE FROM population WHERE CITY = ? ; ";
    dbConnection.query(sqlQuery, city, (err, result) => {
        if (err) {
            return response.status(400).json({ Error: "Failed: Record was not deleted" });
        }
        return response.status(200).json({ Success: "Succcessful: Record was deleted!" });
    });
});
```

Below is a sample request via Thunder Client. The record for Lacey should be deleted.



A phpMyAdmin inquiry following the successful REST service request using the DELETE method.