Info 2312 Final Project

Summary

By Douglas Oak

100356690

Online Database Builder and Viewer

**Purpose:**

The purpose of the online database builder and viewer (ODBV), is to provide a web service that allow users to quickly load data into a MySQL database using a simple GUI and to view that data. In the future, further tools are intended to allow for online data viewing and manipulation.

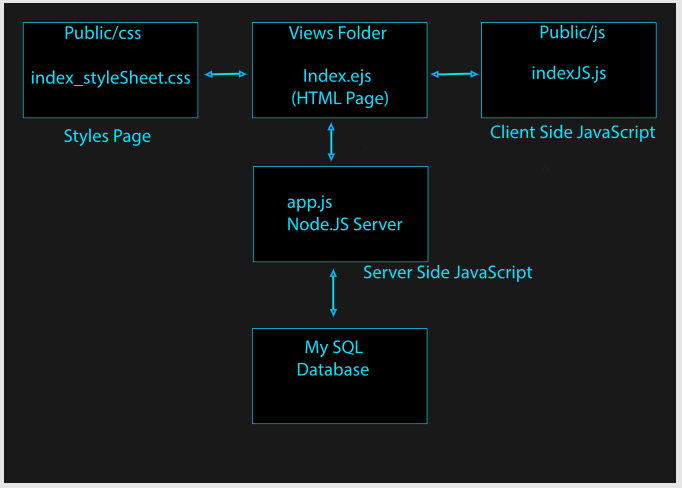
**Features:**

At present, users can add or drop a database. View the names of all databases created. Show tables within the database and view the data within those tables. Users can also load a csv into their webpage. CSVs with both headers and non-headers can be loaded and are differentiated by the user; through the use of a radio button on the webpage. Doing so, brings up the database UI creation tool. The creation tools allow users to customize their header names using text boxes. A row of radio buttons beside each text box, allows users to categorize data as primary keys, strings or numbers. Users can also select the delete radio option to not include data all together.

**Languages/Framework/Packages:**

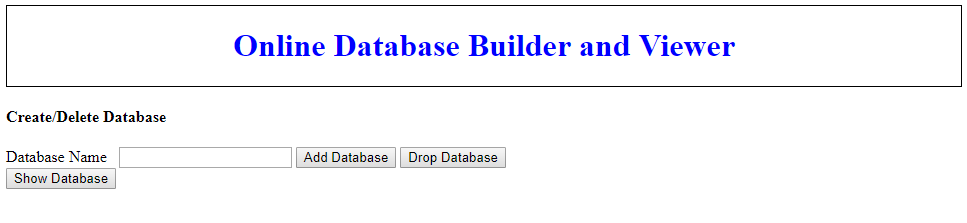
The application uses JavaScript ES6. The server is written using the NodeJS 6.4.1 framework. Packages used were body-parse 1.18.3, ejs 2.6.1, express 4.16.4 and mysql 2.16.0. Communication between server and client was handled via JSON strings transmitted using AJAX.

**Structure:**

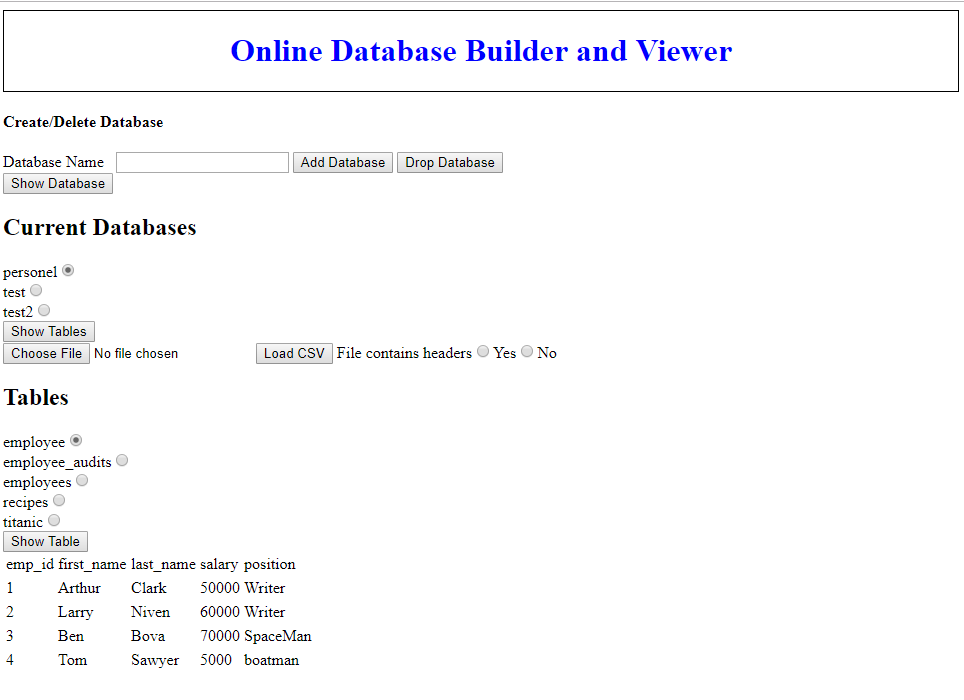


The application is divided into layers; a client layer, a server layer and the database layer. The client layer is controlled by the files Index.ejs, index\_styleSheet.css and indexJS.js. The Server layer is controlled by the app.js file and controls all communication between the client and the database. The database layer is a MySQL data base.

**UI:**

The user interface is a single HTML webpage served to the browser by the NodeJS server. 

Much of the initial HTML document, consists a header bar, a text box for the user to insert a database name, and 3 buttons for adding a database, dropping a database and showing any databases currently on the MySQl server. Clicking on the Show Database button, activates functions in the indexJS.js file which then writes response HTML into ID labeled empty div boxes for the user. These functions are used to create new text boxes, buttons and to dynamically create tables and to display response messages. The webpage also contains a number of invisible text boxes used to store information on the client side for further processing later.



**Application Operation:**

The user primarily controls the application through the clicking of buttons tagged with Onclick = “[function name]()” calls. The button clicks call functions from the IndexJS.js file. The functions communicate with the NodeJS server using Ajax calls tagged to specific routes. Ajax is provided vis link to a JQuery server serving JQuery 3.3.1.min.js. Information is passed/received from/to the server via JSON strings.

Once the data reaches the server, it is often unpacked, converted into SQL strings and then passed onto the database server. Data returned from the database server is then repacked into a JSON string and passed back up to the client. Once at the client, JavaScript is used to write HTML code to display the data or options for interacting with the data. In addition, a number of functions will store their data in hidden text boxes for later use by other asynchronous functions.

**JavaScript Functions:**

**showDatabases()**

ShowDatabases() creates a list of databases available. It is activated by clicking the “Show Databases” button.

ShowDatabases operates by calling the '/showdatabases' route. This function does not transmit any data to the server. It causes the server to create a string with the value “SHOW DATABASES”, connect to the database server and then transmit the before mentioned string.

Information is then returned as a list of names. The names are that of internal databases. Internal database names “information\_schema”, “mysql”, “performance\_schema” and “sys” are then removed from this list and then the new list is transmitted the client side using a comma delimited string. Note :This was the first time I had done this, and I did not yet how to transmit a list via JSON.

Once the above string is returned, to the client side showDatabases() function, the function then parses the string into an array. The function when writes a header tag and iterates through the database name list, outputting an HTML string into the “databaselist” div that names the database along with a radio button that allows the database to be selected.

Finally, the last of databases is saved to a hidden text box on the page for future use.

**Add/drop Dbase() functions**

Add and drop database functions respectively us the contents of the database name text box to add or remove data bases from the database.

Activated by the Add and drop database buttons, these functions load the value of the database text box into a JSON sting and pass that value via the add/drop Dbase routes to the server. The server then concatenates the database name onto the end of a “CREATE DATABASE “ string and then passes that string via a database query to the database. Finally, both functions call the showdatabases() function to update the client’s browser.

If the database name is empty, drop database sets the name to ‘default’ in order to prevent wiping out of all databases.

**showTables()**

ShowTables() displays a list of tables in the selected database.

The show tables function loads and parses the database name list. It then iterates through the list and checks to see which associated radio button is selected. That name is then passed, via a JSON string; via the ‘/showtables’ route. The database name is then appended to a “SHOW TABLES FROM “ string and passed to the database. The response is a list of tables that is loaded into a new array and stringified into a JASON sting and passed onto the client. This list is then parsed and iterated through, where it is loaded into a list along with a radio button in the “showtables” div. This is list is then stored on the client page using a new hidden text box.

**displayTable()**

The display table function displays the contents of a database table in the browser window via an HTML table. Upon pressing the “display table” button. The function first loads up the stored list of database names. It iterates through these names to generate the ID tag of the database radio buttons; loading the “selected” value into the data base name variable. It then loads the stored list of table names and iterates through that list to determine the “selected” table as above. That name is also loading into the variable for the query table name. These names are then concatenated into a string that reads “[dbase name].[table name]”.

This table is then passed to the server via the “/showtable” route.

Once server side, it is concatenated to a “SELECT \* FROM “ string; which is then transmitted to the database.

What returns from the database is a list of (key : data) pairs where each piece of data is referred to by its header. This is first unpacked by creating a list of headers. This list is then used to guide a nested loop which iterates through the raw data in the outside loop and iterates through the headers in the inside loop to pack the return data into a list which represents a table. The first position of the list contains a column count. This list is then returned to the client.

The client starts by loading the column count. It then uses to column count to iterate through the list and generate the code for an HTML table. Once this string is generated, it is loaded into the “show table” div.

**Loading .CSV Functions:**

**TCUI()**

The “To CSV User Interface” function allows a user to load a .CSV file, select whether or not the file has headers and then create an interactive HTML table for the user on the page. The list contains text boxes for each header name. If the user selects the radio button indicating that the CSV has no headers, these fields are blank. Beside this, is text from the first row of data in the csv file providing a sample of what the data looks like. Underneath are radio buttons allowing the user to set each row as a:

Primary key, number, string, to not include the data in the database.

Finally, the function saves the data into a hidden page box as a list of string rows.

**UploadToDb()**

**Client Side:**

This function uses the user settings and header names to convert the loaded csv file into a database for the user.

This function starts by loading the raw csv data into an array. The first string of this raw data array is used to generate a max number of columns to be created. This number then drives a FOR loop that iterates through each field in the UI and creates a “data selection guidance array” which is essentially a list of column positions.

Using the dataSelectionGuidenceArray to guide the selection of data from the raw data array and the UI, a list is generated which consists of a NEW number of columns, a list of column names from the UI text boxes; the ones which haven’t been selected for non-inclusion. A list containing datatypes comes next, then comes all the data and finally ,tacked onto the end, are the names of the database, the information is being added to, and the name of the new table; which was collected from the table name box. This new list is then transmitted to the server via the ‘/loadcsv’ route.

**Server Side:**

Once server side, the list is unpacked and the SQL command dynamically generated. The server starts by loading the number of columns, the database name and the table name from the ends of the list. The columns number is them used to drive a FOR loop that creates a header list and a linked datatype list. A “CREATE TABLE” string is then generated that contains “CREATE TABLE [dbase name].[table name] ( [header1 name] [datatype], [header2 name][datatype] …);

Next the “INSERT INTO” string is created. Concatenated to this string is [dbase name].[table name] VALUES. Then the iterated data is added, preceded by a “(“ and ended with a “),” .

Once finished, the end comas are clipped off of the end of both strings and semi-colons are added to the end of each. Finally, the “CREATE TABLE” string is transmitted to the data base followed by the “INSERT INTO” string.

Finally, the showTables() function is called in order to update the page.

**Areas for Further Development**

There is an obvious development of sophistication with this project’s code as one goes from start to finish. The earliest functions are clunky and overly verbose while the later functions show marginal iterated improvement. A basic first step would be to rewrite all the code from start to finish; allying the lessons learned during initial writing.

Secondly, the code is quite brittle, and the application could easily be crashed by an unknowing user. Add to this, the code contains no filtering to protect against SQL injection. This can be improved by incorporating checks on user input, adding error handling code and setting some defaults.

Finally, the application was written as a proof of concept and no unified purpose was incorporated into the creation of features. A more focused purpose would lead to a more polished product.