ER Diagram and Database

Table of Contents

December 12, 2018

Contents

1 Executive Summary

This document's objective and purpose is to show the architecture of the database as well as show the design and implementation of the database for the Community Action Partnership of Dutchess County (CAP Dutchess). The CAP Dutchess database requires a database that can store, insert, update, delete data, and sort different surveys as well as different versions of those surveys. With that it must also be able to view the reports of data for their organizations records and purposes. The database will be key in storing survey data and projecting the data to better help the people of CAP Dutchess make their services even better.

Previously all the data that was being recorded for their surveys was all done by pen and paper. From this manually inputted data, information was extracted to create graphs and charts for statistics digitally. All of this can now quickly and easily be found, rather than manually recording and searching for the information on paper surveys. The database is also integrated with its own survey client and management application. The implementation of the database will coincide with the use of physical survey recordings, in case if the location that CAP Dutchess is servicing does not have any internet connection or devices.

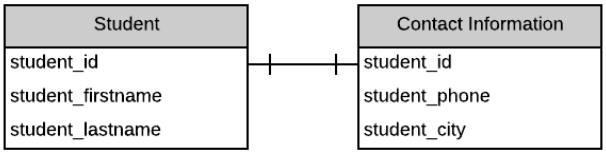
2 Entity Relationship Diagram

An Entity Relationship Diagram, also known as **ER Diagram** or **ERD** is a type of flowchart that depicts how "entities" such as people, objects, or concepts relate to one another within a system. These diagrams use an assortment of shapes and characters such as ovals, rectangles, diamonds and lines to display the connection between **entities**, **relationships** and **attributes**. The definitions of key terms regarding **ERD's** are as follows:

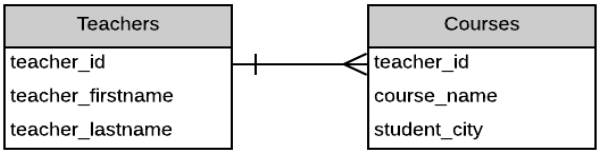
Entity: A definable thing—such as a person, object, concept or event—that can have data stored about it. At Marist, an example would be "Students" or "Staff".
Relationships: How entities act upon each other or are associated with each other. For example, if a student at Marist is registering for a course, the two entities that would be related to one another would be students and courses and the relationship would be registering.
Attributes: A property or characteristic of an entity. An example of this would be the **entity** "student" having a student_id, first_name, and last_name.
Cardinality: Defines the numerical attributes of the relationship between two entities or entity sets. These include the main relationships **one-to-one**, **one-to-many**, **many-to-many**.

In regards to cardiniality and for the purposes of this project, there are three main relationships that are defined below:

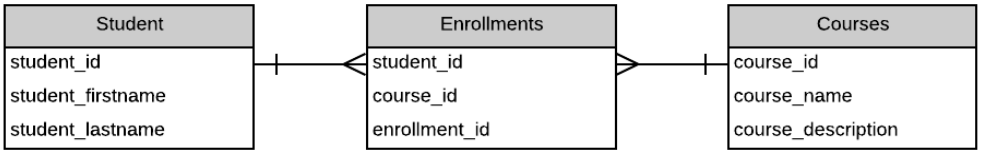
One-to-One: One record in a table is associated with one and only one record in another table. For example at Marist College, one student can only have one Marist ID, and this Marist ID is only for one student.



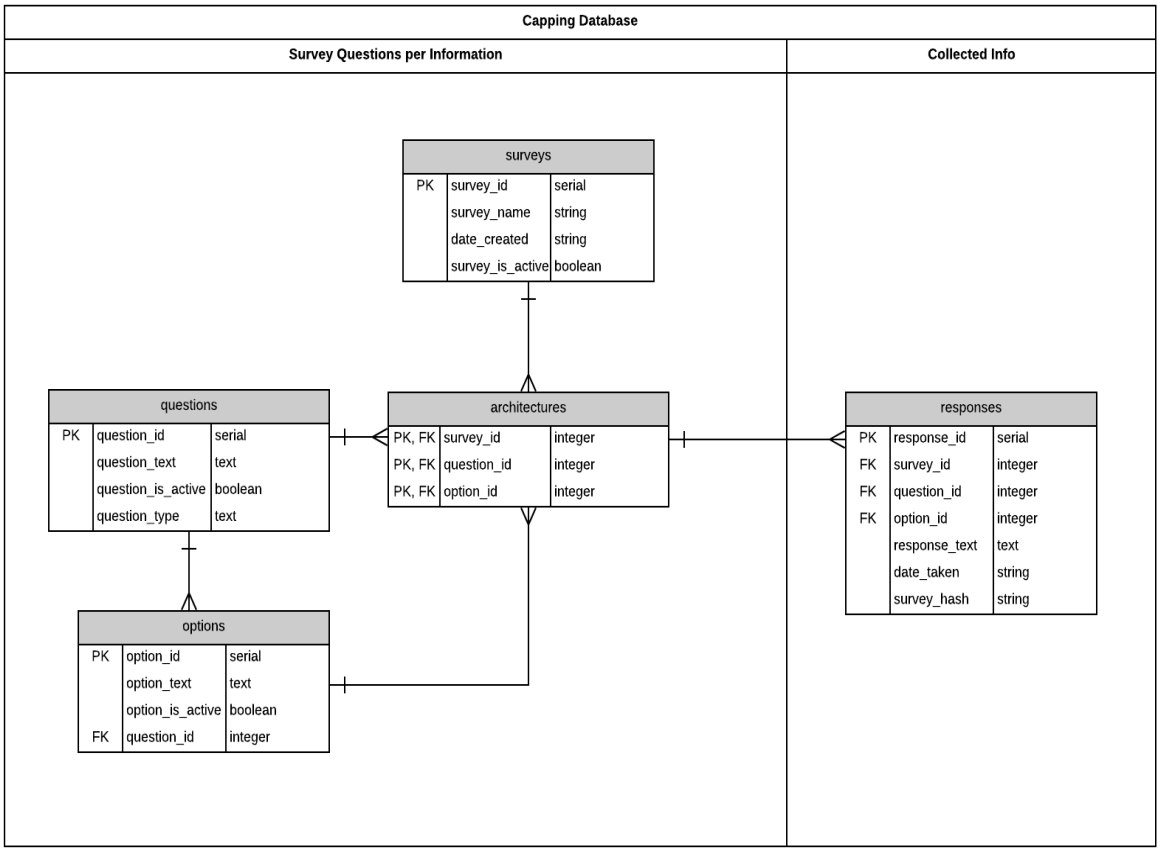
One-to-Many: One record in a table is when one record in a table can be associated with one or more records in another table. For example at Marist, a professor can teach multiple courses, but these courses will not have the same relationship with the professor.



Many-to-Many: When multiple records in a table are associated with multiple records in another table. Typically these types of relationships consist of three tables, two entities joined by a **join table**. For example at Marist, a student can be enrolled in many courses, and a course can contain many students.



ER Diagram

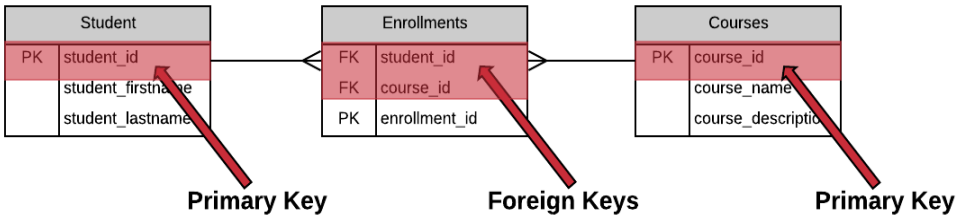

Capping Database
Survey Questions per Information
Collected Info
surveys
PK survey_id serial
survey_name string
date_created string
survey_is_active boolean
questions
PK question_id serial
question_text text
question_is_active boolean
question_type text
architectures
PK, FK survey_id integer
PK, FK question_id integer
PK, FK option_id integer
responses
PK response_id serial
FK survey_id integer
FK question_id integer
FK option_id integer
response_text text
date_taken string
survey_hash string
options
PK option_id serial
option_text text
option_is_active boolean
FK question_id integer

3 Primary and Foreign Keys

Primary and **foreign** keys are essential to creating a fully functioning and efficient relational database. Keys are a way to categorize and link the different attributes that are contained in the **entity** tables.

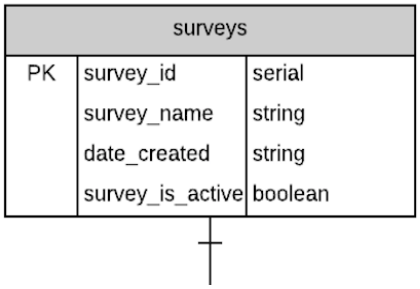
A **primary key** or (PK) is an attribute or combination of attributes that uniquely identifies one and only one instance of an entity. A primary key **must** 1.) be unique and 2.) contain a value (cannot be null).

A **foreign key** or (FK) is created any time an attribute relates to another entity in a one-to-one or one-to-many relationship.



4 Tables

4.1 surveys



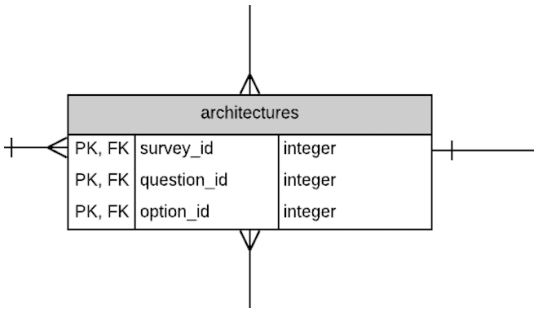
The Surveys table is used to contain the information regarding the survey itself. It includes the attributes:

- survey_id (PK)
- survey_name
- date_created

This table has relationships with two tables, and these include:

Table	Relationship	Other Table
surveys	Many-to-One	contact_info
surveys	One-to-Many	architectures

4.2 architectures



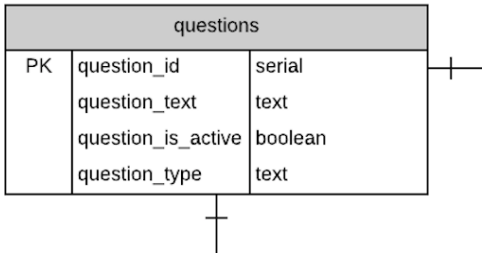
The architectures table is used as a **join** table to connect the responses from the collected information to the questions and options in the survey questions per information section. The attributes included in this table are:

- survey_id (PK, FK)
- question_id (PK, FK)
- option_id (PK, FK)

This table has relationships with four tables, and these include:

Table	Relationship	Other Table
architectures	Many-to-One	surveys
architectures	One-to-Many	responses
architectures	Many-to-One	questions
architectures	Many-to-One	options

4.3 questions



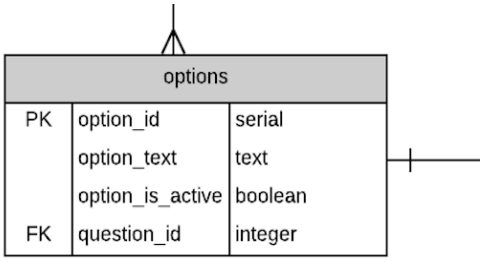
The questions table is used to record all of the information regarding the questions that are obtained from the survey editor. The attributes of the questions entity are:

- question_id (PK)
- question_num
- question_text
- question_is_active
- question_type

This table has relationships with two tables, and these include:

Table	Relationships	Other Table
questions	One-to-Many	architectures
questions	One-to-Many	options

4.4 options



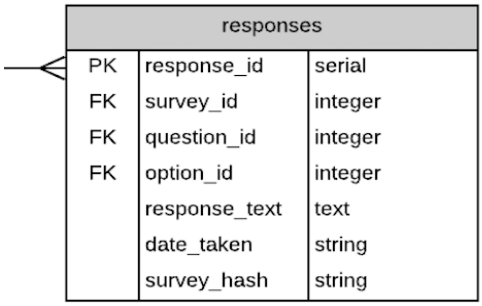
The options table is used to record the available answers for a given question, using the question_id. The table has the following attributes:

- option_id (PK)
- option_num
- option_text
- option_is_active
- option_id (FK)

This table has relationships with two tables, and these include:

Tables	Relationships	Other Table
options	Many-to-One	questions
options	One-to-Many	architectures

4.5 responses



The responses table is used to record all of the responses from participants to specific questions, based on question_id. Each question_id will have a option_id that is connected to it for a given survey_id. This table contains these attributes:

- response_id (PK)
- survey_id (FK)
- question_id (FK)
- option_id (FK)
- response_text
- date_taken

This contact_info table has relationship with one table, and it is:

Table	Relationship	Other Table
responses	Many-to-One	architecture

5 API Stored Procedures

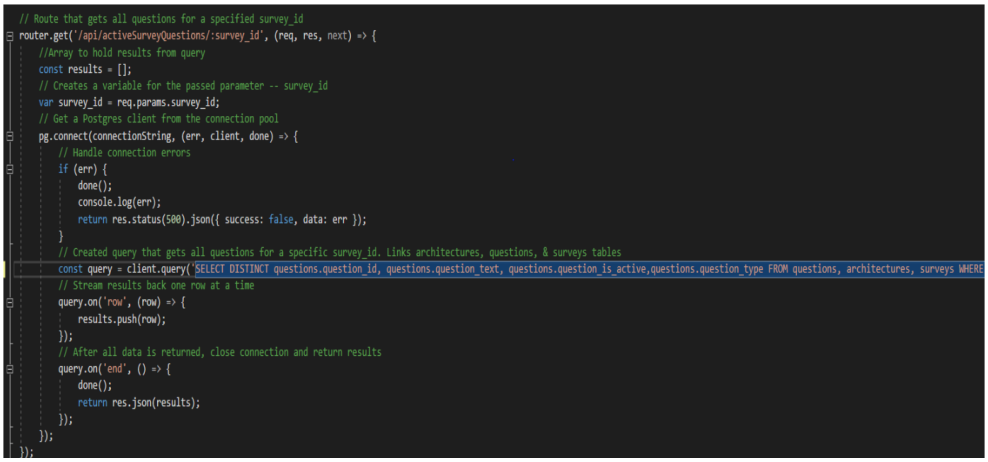
Stored Procedures are sets of SQL statements that are executed in node through the use of routes. Each route executes one SQL command in the database. These SQL commands can be combined our used in loops to execute a SQL command on multiple data items. Each of these API stored procedures our routes also have a corresponding survey service.ts function call that returns/posts a corresponding JSON observable.

5.1 Get All Active Survey Questions

SQL/Node Query:

```
const query = client.query(
'SELECT DISTINCT questions.question_id, questions.question_text,
questions.question_is_active,questions.question_type
FROM questions, architectures, surveys WHERE surveys.survey_id = ($1)
AND questions.question_is_active = true AND questions.question_id =
architectures.question_id AND architectures.survey_id =
surveys.survey_id
ORDER BY questions.question_id ASC', [survey_id]
);
```

Explanation: This route/query will get all of the all active survey questions given a specific survey_id.

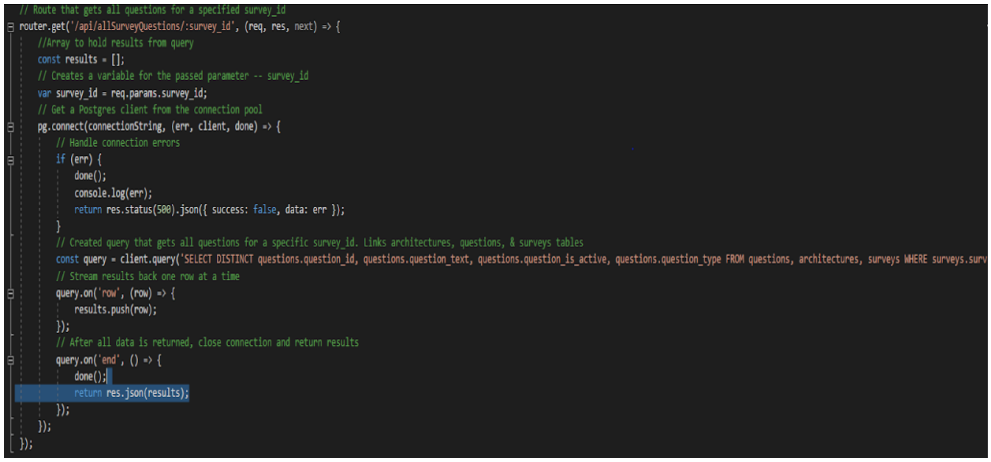


5.2 Get All Survey Questions

SQL/Node Query:

```
const query = client.query(
'SELECT DISTINCT questions.question_id, questions.question_text,
questions.question_is_active, questions.question_type
FROM questions, architectures, surveys WHERE surveys.survey_id = ($1)
AND questions.question_id = architectures.question_id
AND architectures.survey_id = surveys.survey_id
ORDER BY questions.question_id ASC', [survey_id]
);
```

Explanation: This route/SQL statement will get all survey questions regardless of whether or not they are active given a specific survey_id

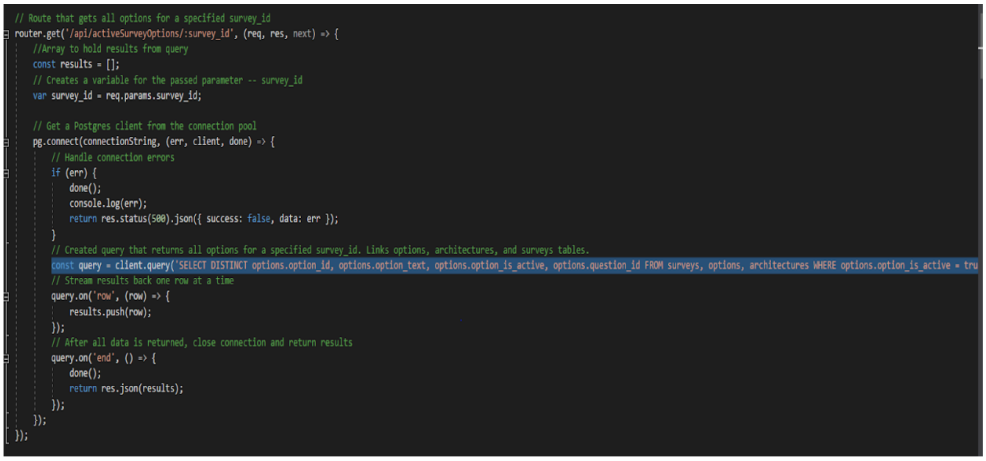


5.3 Get All Active Survey Options

SQL/Node Query:

```
const query = client.query(
'SELECT DISTINCT options.option_id, options.option_text,
options.option_is_active, options.question_id
FROM surveys, options, architectures
WHERE options.option_is_active = true
AND options.question_id = architectures.question_id
AND architectures.survey_id = surveys.survey_id
AND surveys.survey_id = ($1)
ORDER BY options.question_id, options.option_id ASC', [survey_id]
);
```

Explanation: This query will get all active options given a specific survey_id.

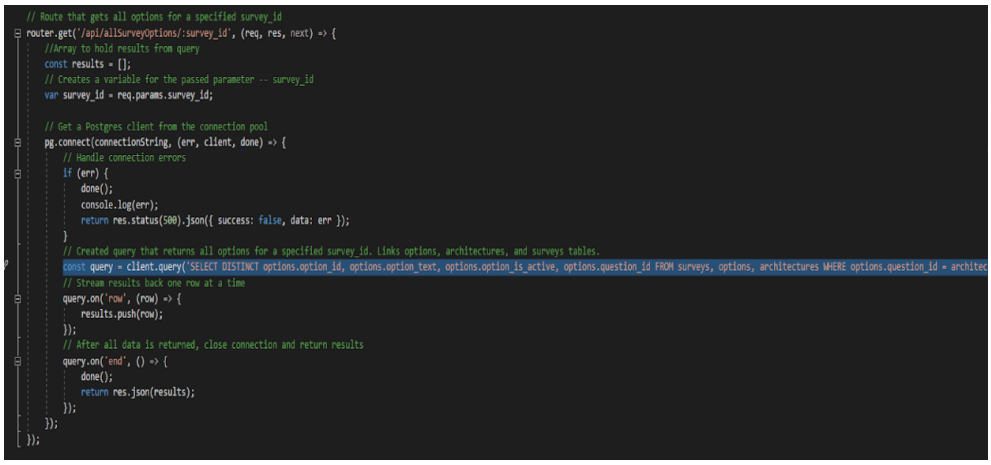


5.4 Get All survey options

SQL/Node Query:

```
const query = client.query(
'SELECT DISTINCT options.option_id, options.option_text,
options.option_is_active, options.question_id
FROM surveys, options, architectures
WHERE options.question_id = architectures.question_id
AND architectures.survey_id = surveys.survey_id
AND surveys.survey_id = ($1)
ORDER BY options.question_id, options.option_id ASC', [survey_id]
);
```

Explanation: This query will get all survey options given a specific survey_id regardless of whether they are active or inactive.

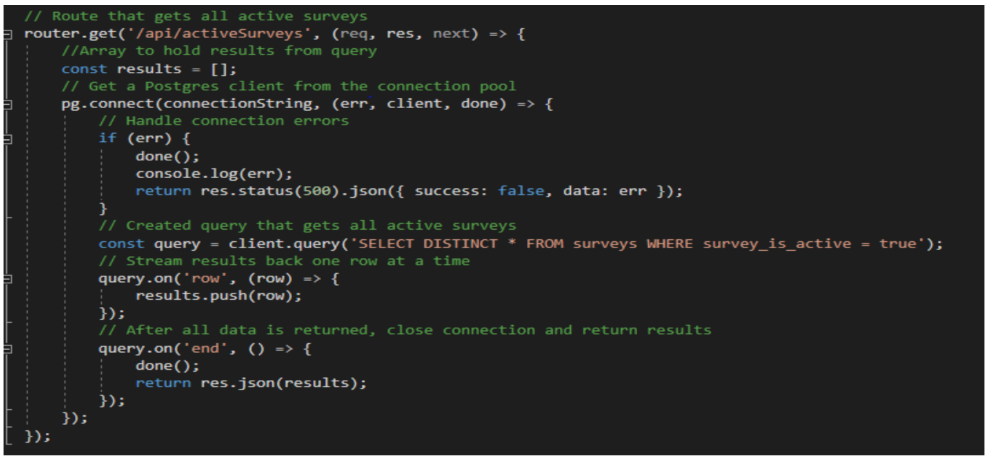


5.5 Get All Active Surveys

SQL/Node Query:

```
const query = client.query(
‘SELECT DISTINCT responses.response_id, responses.survey_id,
responses.question_id, responses.option_id,
responses.response_text, responses.date_taken,
responses.survey_hash
FROM responses, architectures, surveys
WHERE responses.survey_id = architectures.survey_id
AND architectures.survey_id = surveys.survey_id
AND surveys.survey_id = ($1)
ORDER BY response_id ASC’, [survey_id]
);
```

Explanation: This query will return all active surveys in the database.



5.6 Get All Surveys

SQL/Node Query:

```
const query = client.query(
‘SELECT *
FROM surveys
ORDER BY survey_id ASC’
);
```

Explanation: This query will get all of the surveys in the database regardless of whether they are active or inactive.

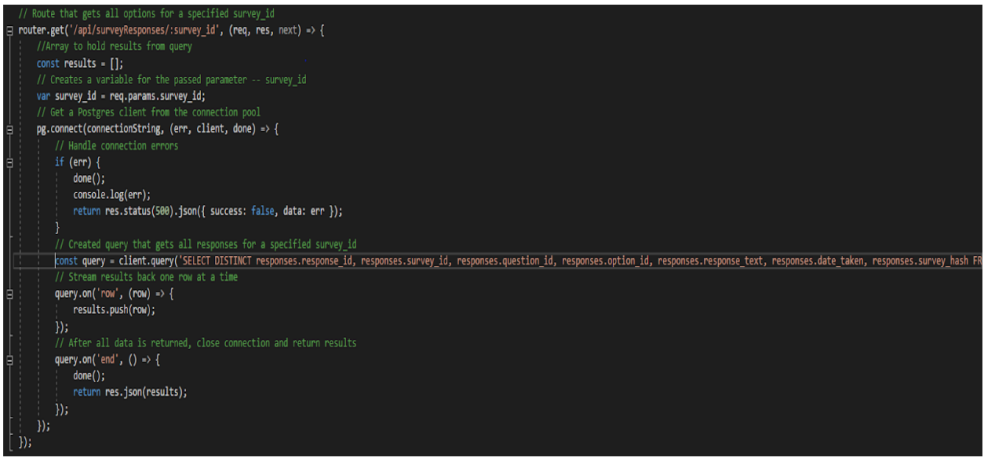
```
// Route that gets all surveys
router.get('/api/allSurveys', (req, res, next) => {
    //Array to hold results from query
    const results = [];
    // Get a Postgres client from the connection pool
    pg.connect(connectionString, (err, client, done) => {
        // Handle connection errors
        if (err) {
            done();
            console.log(err);
            return res.status(500).json({ success: false, data: err });
        }
        // Created query that gets all active surveys
        const query = client.query('SELECT * FROM surveys ORDER BY survey_id ASC');
        // Stream results back one row at a time
        query.on('row', (row) => {
            results.push(row);
        });
        // After all data is returned, close connection and return results
        query.on('end', () => {
            done();
            return res.json(results);
        });
    });
});
```

5.7 Get All Survey Responses

SQL/Node Query:

```
const query = client.query(
'SELECT DISTINCT responses.response_id, responses.survey_id,
responses.question_id, responses.option_id,
responses.response_text, responses.date_taken,
responses.survey_hash
FROM responses, architectures, surveys
WHERE responses.survey_id = architectures.survey_id
AND architectures.survey_id = surveys.survey_id
AND surveys.survey_id = ($1)
ORDER BY response_id ASC', [survey_id]
);
```

Explanation: This SQL query will get all survey responses given a specific survey_id.



5.8 Get All Survey Info

SQL/Node Query:

```
const query = client.query(
'SELECT s.survey_id, s.survey_name,
s.date_created, s.survey_is_active,
COUNT( DISTINCT r.survey_hash) as response_count
FROM responses r
INNER JOIN surveys s
ON r.survey_id = s.survey_id
GROUP BY s.survey_id'
);
```

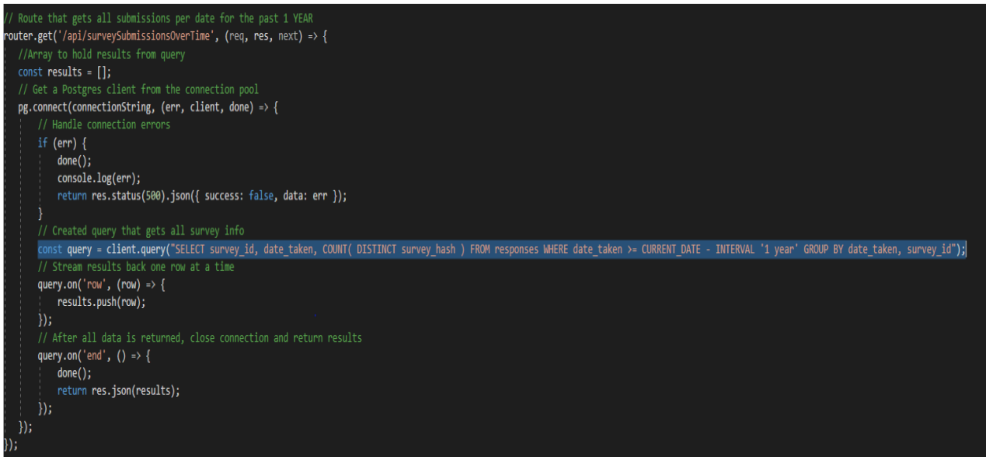
Explanation: This SQL query will return all information about a specific survey. It will return the survey's name, date created, whether it is active or inactive, how many distinct user's have taken that survey, and the number of responses for each survey.

5.9 Get All Survey Submissions Over Time

SQL/Node Query:

```
const query = client.query(
‘SELECT survey_id, date_taken,
COUNT( DISTINCT survey_hash )
FROM responses
WHERE date_taken >= CURRENT_DATE - INTERVAL ’1 year’
GROUP BY date_taken, survey_id"
);
```

Explanation: This SQL Query will give all submissions per survey per day over the interval of the past year.



5.10 Post New Survey Response

SQL/Node Query:

```
query = client.query(
‘INSERT INTO responses (survey_id, question_id, option_id,
response_text, survey_hash)
VALUES ($1, $2, $3, $4, $5)’, [req.body[i].survey_id,
req.body[i].question_id, req.body[i].option_id,
req.body[i].response_text, req.body[i].survey_hash]
);
```

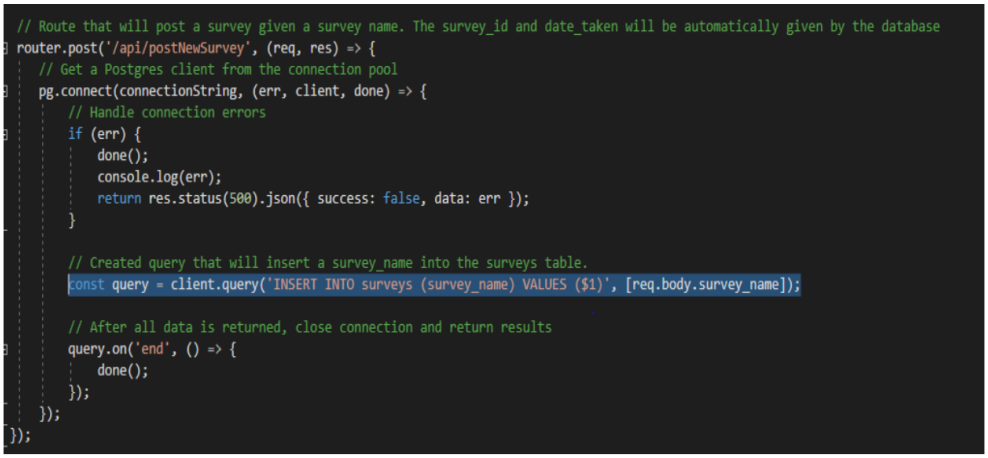
Explanation: This route will post an individual response. This means only one answer per post. We use a loop while indexing an array to post a complete survey worth of responses.

5.11 Post New Survey

SQL/Node Query:

```
const query = client.query(
‘INSERT INTO surveys (survey_name)
VALUES ($1)’, [req.body.survey_name]
);
```

Explanation: This route will take a survey name as a parameter and post it into the surveys table. The surveys table will automatically assign it a survey_id and date_created.

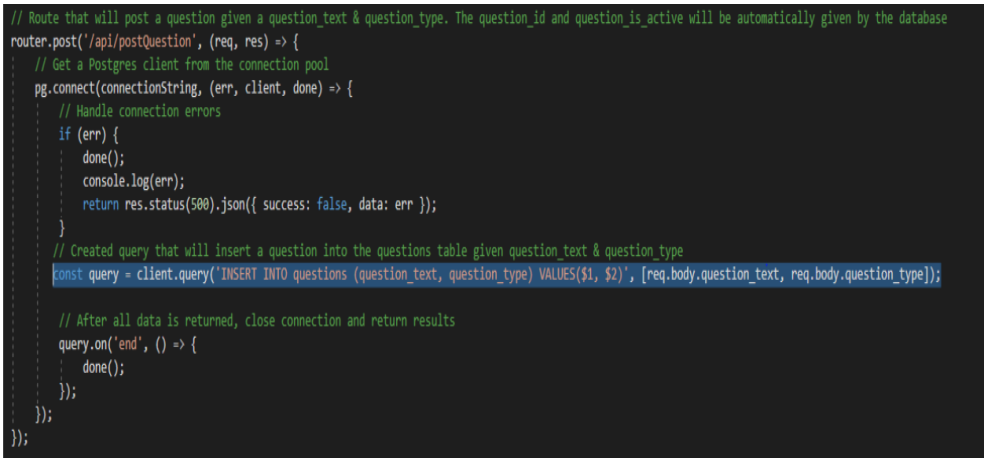


5.12 Post New Question

SQL/Node Query:

```
const query = client.query(
'INSERT INTO questions (question_text, question_type)
VALUES($1, $2)', [req.body.question_text, req.body.question_type]);
```

Explanation: This route will take an individual question and post it to the questions table. The route takes the parameters of a question_text and question_type. The question will automatically be assigned an id, and the default value of active.

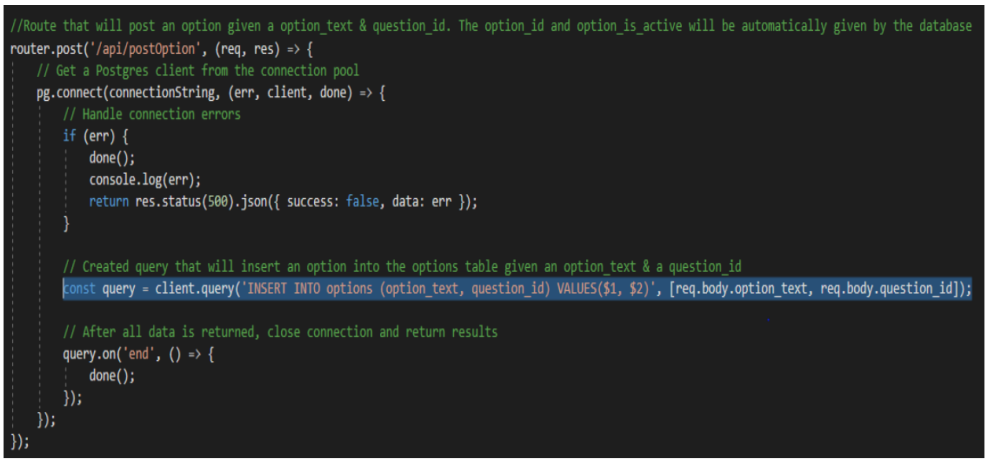


5.13 Post New Option

SQL/Node Query:

```
const query = client.query(
'INSERT INTO options (option_text, question_id)
VALUES($1, $2)', [req.body.option_text, req.body.question_id]
);
```

Explanation: This route will take the parameters of option_text, and the question_id that the option is associated with. The option is then posted to the options table where it is assigned an id and a default value of active.



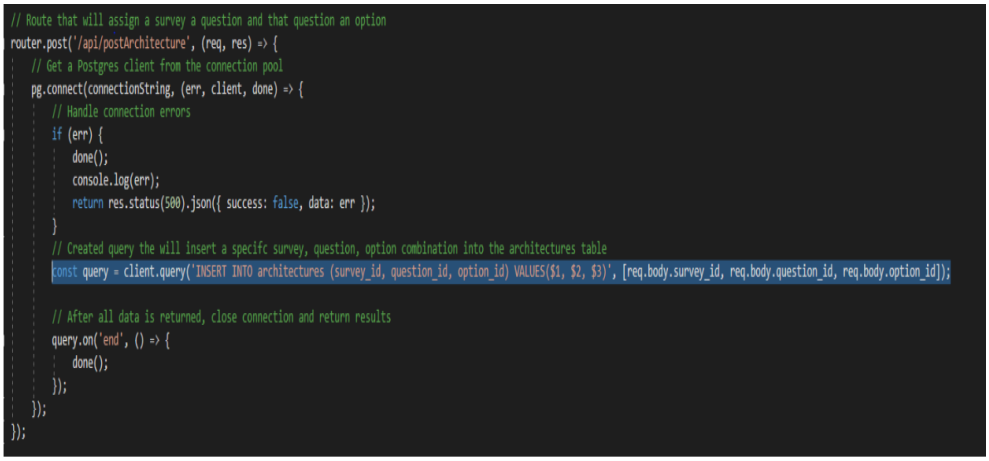
```
//Route that will post an option given a option_text & question_id. The option_id and option_is_active will be automatically given by the database
router.post('/api/postOption', (req, res) => {
    // Get a Postgres client from the connection pool
    pg.connect(connectionString, (err, client, done) => {
        // Handle connection errors
        if (err) {
            done();
            console.log(err);
            return res.status(500).json({ success: false, data: err });
        }

        // Created query that will insert an option into the options table given an option_text & a question_id
        const query = client.query('INSERT INTO options (option_text, question_id) VALUES($1, $2)', [req.body.option_text, req.body.question_id]);

        // After all data is returned, close connection and return results
        query.on('end', () => {
            done();
        });
    });
});
```

5.14 Post New Architecture

SQL/Node Query:

```
const query = client.query(
'INSERT INTO architectures (survey_id, question_id, option_id)
VALUES($1, $2, $3)', [req.body.survey_id, req.body.question_id,
req.body.option_id]
);
```

Explanation: This route will take the parameters of survey_id, question_id, and option_id. These will then be posted to the architectures table. The architecture table holds All the associations between a particular survey, the questions that survey has, and the options for those questions. This is where the tables of surveys, questions, and options come together to form a particular surveys architectures. Think of it as the full survey you would be given to fill out.

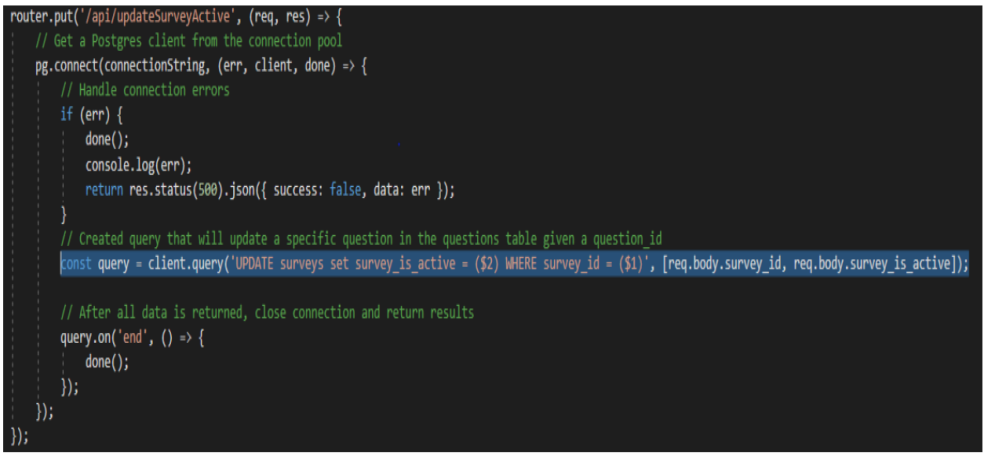


5.15 Updating the Active Status of a Survey

SQL/Node Query:

```
const query = client.query(
‘UPDATE surveys set survey_is_active = ($2)
WHERE survey_id = ($1)’, [req.body.survey_id,
req.body.survey_is_active]
);
```

Explanation: This route will take the parameters of a survey_id and survey_is_active. This route will allow you to set whether you wish a survey to be active or inactive.

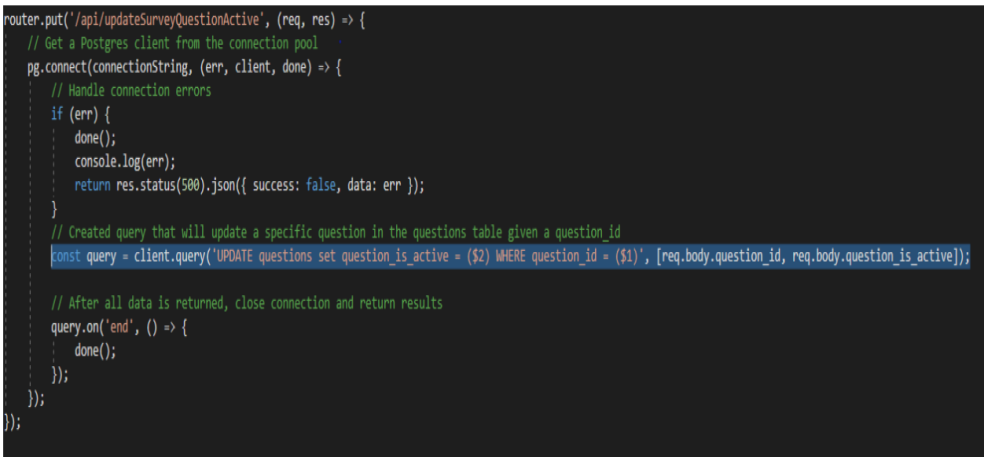


5.16 Updating the Active Status of a Question

SQL/Node Query:

```
const query = client.query(
‘UPDATE questions set question_is_active = ($2)
WHERE question_id = ($1)’, [req.body.question_id,
req.body.question_is_active]
);
```

Explanation: This route will take the parameters of a question_id and question_is_active. This route will allow you to set whether you wish a question to be active or inactive within a survey.

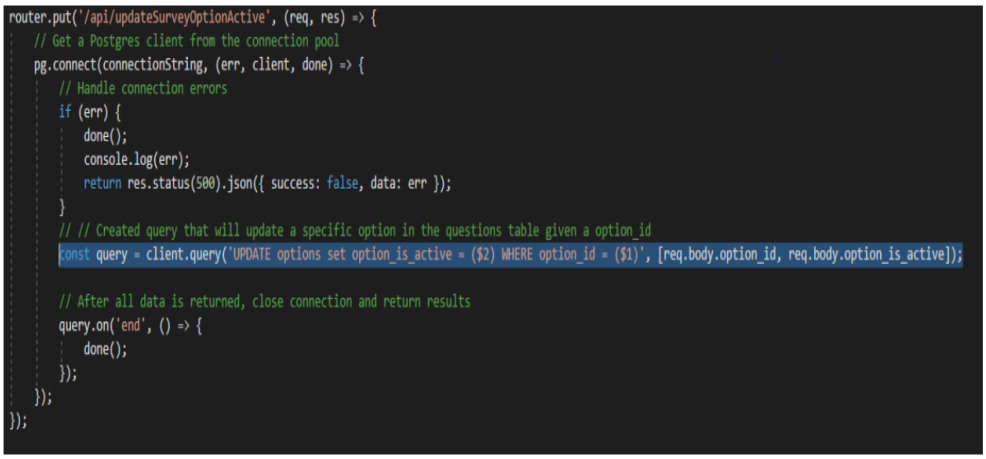


5.17 Updating the Active Status of an Option

SQL/Node Query:

```
const query = client.query(
‘UPDATE options set option_is_active = ($2)
WHERE option_id = ($1)’, [req.body.option_id,
req.body.option_is_active]
);
```

Explanation: This route will take the parameters of an option_id and option_is_active. This route will allow you to set whether wish an option to be active or for a question within a survey.



5.18 SQL Script for Creating the Database

```
CREATE DATABASE "CashCoalition"
    WITH
    OWNER = postgres
CREATE TABLE users
(
    user_id serial NOT NULL DEFAULT,
    user_name text DEFAULT NOT NULL,
    user_password text DEFAULT NOT NULL,
    CONSTRAINT "users_Pk" PRIMARY KEY (user_id),
    CONSTRAINT user_name_unique UNIQUE (user_name)

)
CREATE TABLE .surveys
(
    survey_id serial NOT NULL DEFAULT,
    survey_name text DEFAULT NOT NULL,
    date_created text DEFAULT NOT NULL DEFAULT CURRENT_TIMESTAMP,
    survey_is_active boolean NOT NULL DEFAULT true,
    CONSTRAINT surveys_pk PRIMARY KEY (survey_id)
)
CREATE TABLE questions
(
    question_id serial NOT NULL DEFAULT,
    question_text text DEFAULT NOT NULL,
    question_is_active boolean NOT NULL DEFAULT true,
    question_type text DEFAULT NOT NULL,
    CONSTRAINT questions_pk PRIMARY KEY (question_id)
)
CREATE TABLE public.options
(
    option_id serial NOT NULL DEFAULT,
    option_text text DEFAULT NOT NULL,
    option_is_active boolean NOT NULL DEFAULT true,
    question_id integer NOT NULL,
    CONSTRAINT options_pk PRIMARY KEY (option_id),
    CONSTRAINT questions_fk FOREIGN KEY (question_id)
        REFERENCES public.questions (question_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE architectures
(
    survey_id integer NOT NULL,
    question_id integer NOT NULL,
    option_id integer NOT NULL,
    CONSTRAINT architectures_pk PRIMARY KEY (survey_id, question_id, option_id),
    CONSTRAINT options_fk FOREIGN KEY (option_id)
        REFERENCES public.options (option_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
```

```
    CONSTRAINT questions_fk FOREIGN KEY (question_id)
        REFERENCES public.questions (question_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT surveys_fk FOREIGN KEY (survey_id)
        REFERENCES public.surveys (survey_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE responses
(
    response_id serial NOT NULL DEFAULT,
    survey_id integer NOT NULL,
    question_id integer NOT NULL,
    option_id integer NOT NULL,
    response_text text DEFAULT NOT NULL,
    date_taken text DEFAULT NOT NULL DEFAULT CURRENT_TIMESTAMP,
    survey_hash text DEFAULT NOT NULL,
    CONSTRAINT response_pk PRIMARY KEY (response_id, survey_id, question_id, option_id),
    CONSTRAINT architectures_fk FOREIGN KEY (survey_id, option_id, question_id)
        REFERENCES public.architectures (survey_id, option_id, question_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

6 SQL Statements

CREATE DATABASE - This will create an empty database with the given name.

OWNER - This has to do with setting an owner for the given database. This would be the psql username you create or wish to assign the owner privilege to.

CREATE TABLE - This will create a table with the given name. Within the parenthesis of this statement you specify each column you wish to give that table.

NOT NULL - This means that the column assigned with this definition cannot be given or hold a null value.

DEFAULT - This will set a default value for that column if a value is not specified directly in an insert query. serial - This is a data type that will auto increment to the next integer in the table. This is used to automatically assign different ids throughout the database.

CONSTRAINT - This is used to specify a primary key, foreign key(s), or items that should be unique within the database.

REFERENCES - This is used to signify that a column(s) within one table are specifically related to the column(s) of another table. For example, the options table has a column question_id and the questions table has a column question_id. These are the same set of ids, so the REFERENCES keyword is used to signify that these ids are correlated. If the table being referenced does not hold a specific id or an id is deleted within the referenced table, and is attempted to be added to the referencing table it will cause an error.

CURRENT_TIMESTAMP - This is a sql function that assigns the current date and time to a column. This is used scarcely through the database. This is used mainly to assign a default date/time to a few columns. These would be date_created and date_taken.