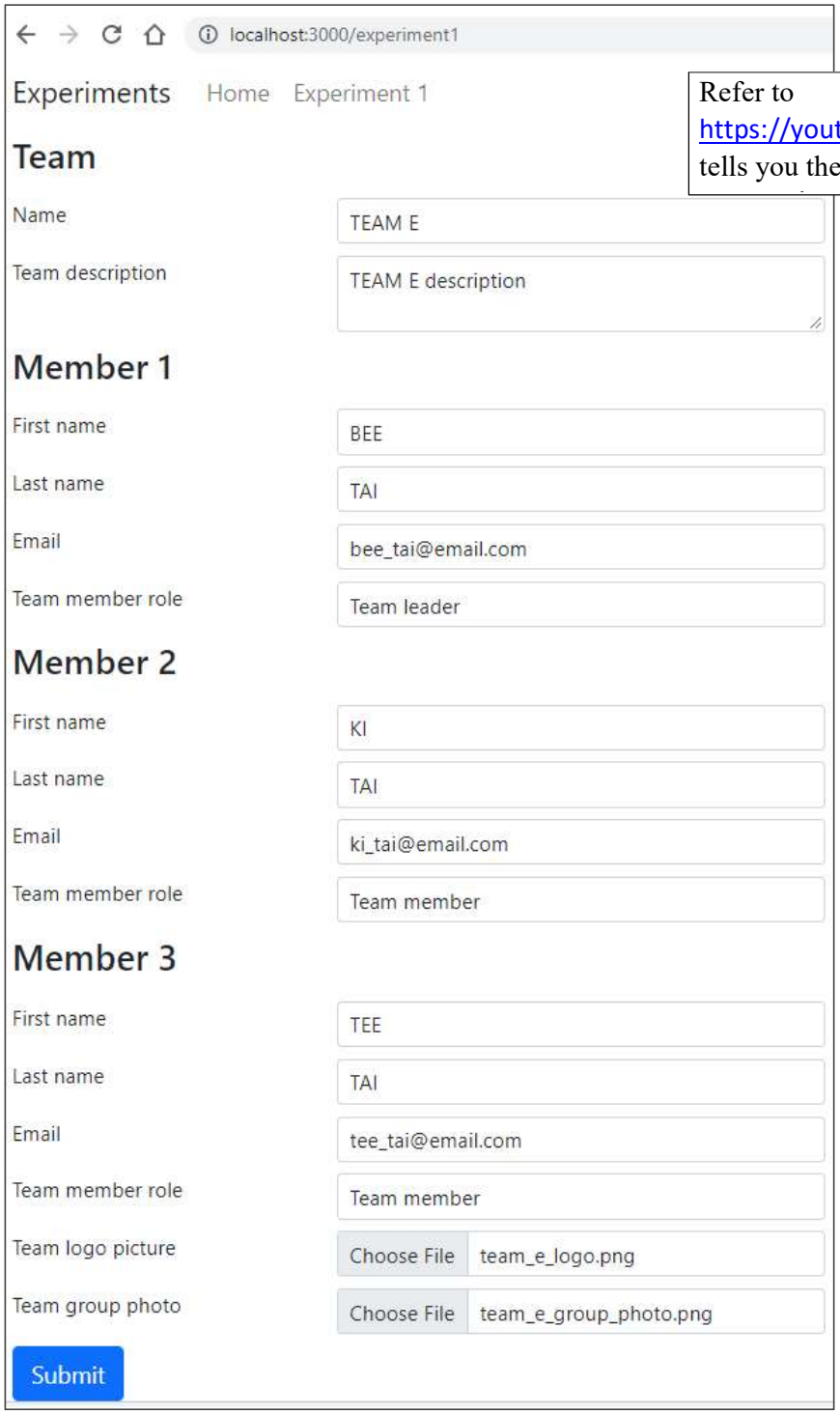
Table of Contents

What is inside this practical worksheet

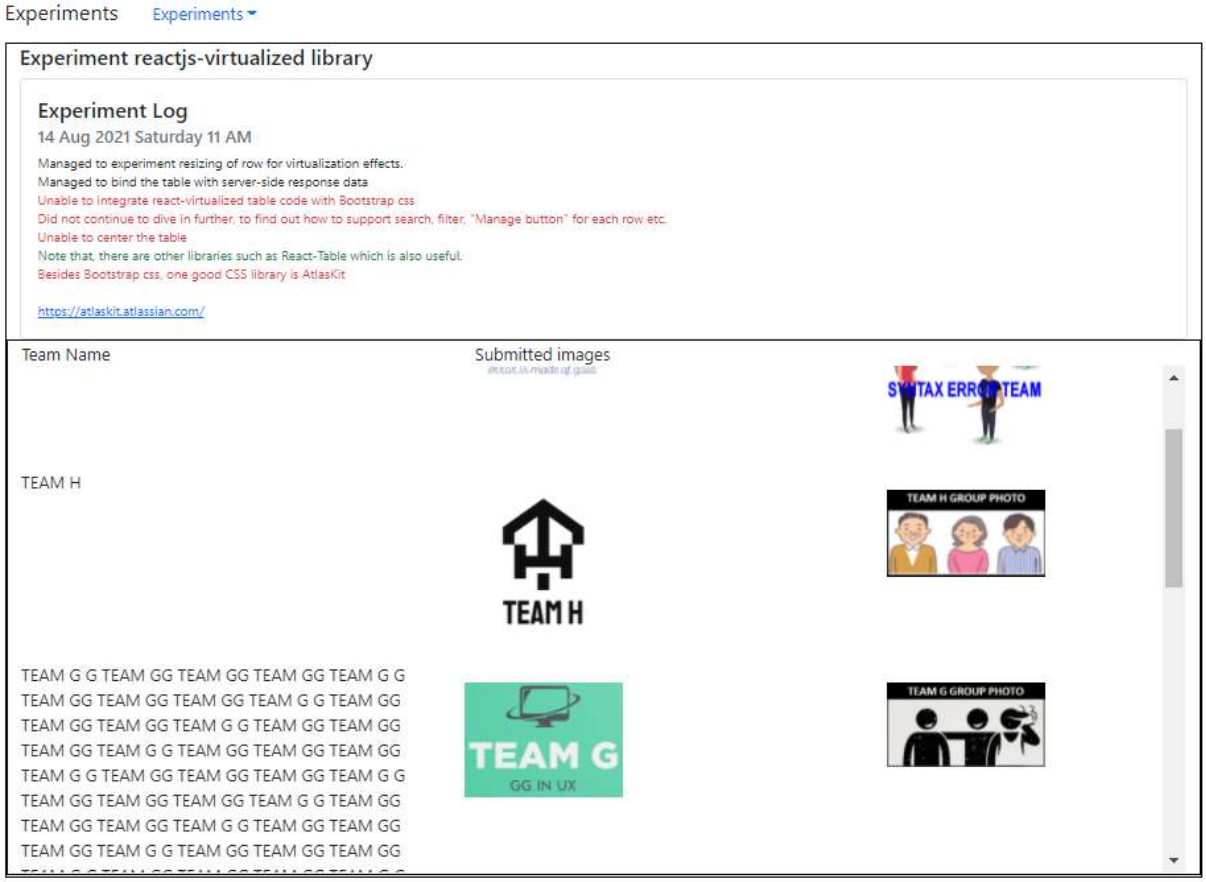
Refer to the video https://youtu.be/SctrhaGtTdg which covers the worksheet exercise learning goals and skill goals

You are going experiment and build a NodeJS backend system **from scratch**. The figure below describes how the frontend interface will work with the NodeJS backend system.



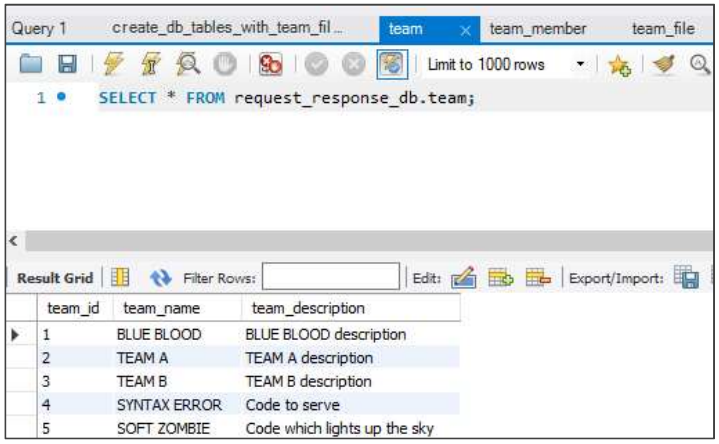
Refer to https://youtu.be/SctrhaGtTdg?t=1349 which tells you the expected outcome/result which

The backend system creates records inside the database and writes file data to Cloudinary storage. The figure below shows how the frontend interface logic can retrieve data from the backend system and renders the information to the user. You don't need to code the frontend project. The project files are available for you to test your backend system.



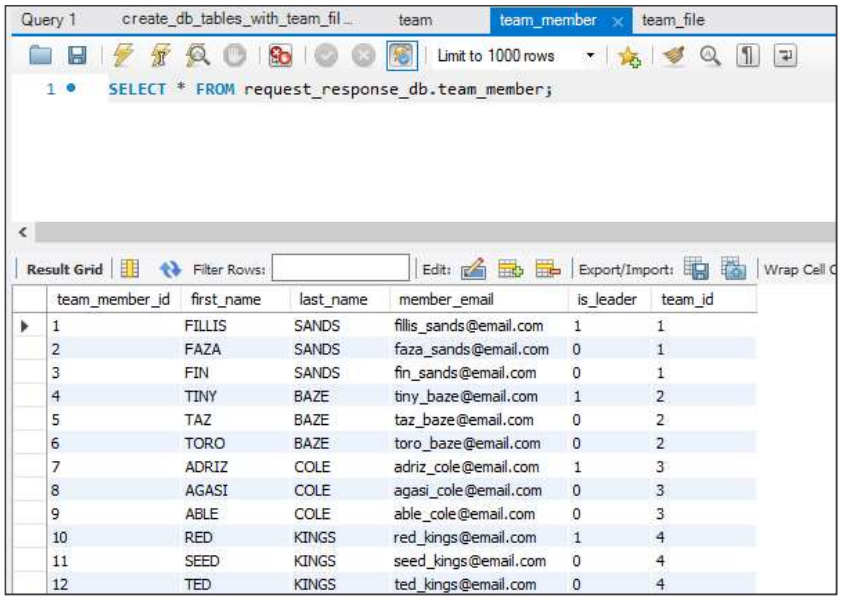
Backend System Development Fundamentals which operates on Parent Child record in database

The entire worksheet exercise shall work on three tables.



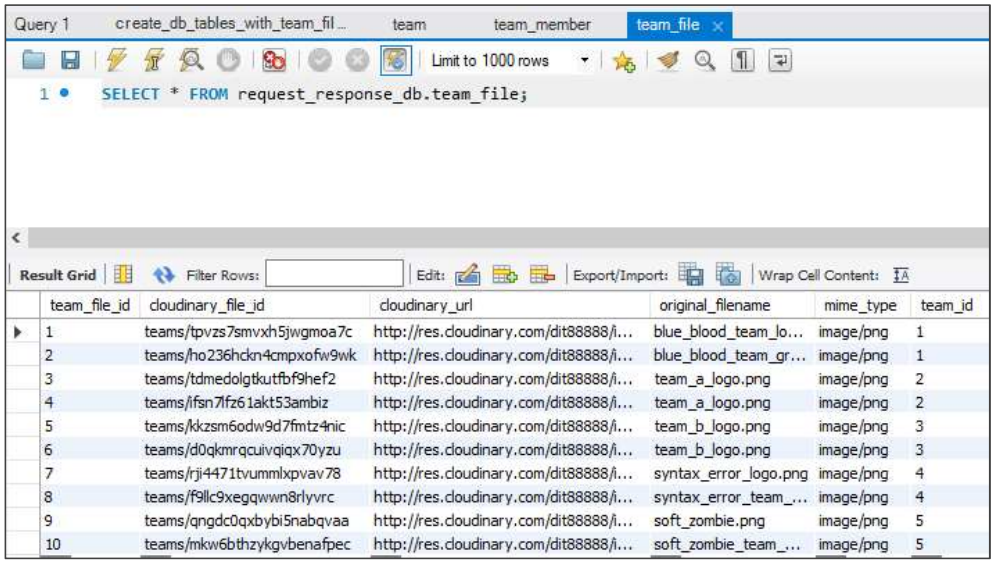
```
SELECT * FROM request_response_db.team;
```

team_id	team_name	team_description
1	BLUE BLOOD	BLUE BLOOD description
2	TEAM A	TEAM A description
3	TEAM B	TEAM B description
4	SYNTAX ERROR	Code to serve
5	SOFT ZOMBIE	Code which lights up the sky



```
SELECT * FROM request_response_db.team_member;
```

team_member_id	first_name	last_name	member_email	is_leader	team_id
1	FILLIS	SANDS	fillis_sands@email.com	1	1
2	FAZA	SANDS	faza_sands@email.com	0	1
3	FIN	SANDS	fin_sands@email.com	0	1
4	TINY	BAZE	tiny_baze@email.com	1	2
5	TAZ	BAZE	taz_baze@email.com	0	2
6	TORO	BAZE	toro_baze@email.com	0	2
7	ADRIZ	COLE	adriz_cole@email.com	1	3
8	AGASI	COLE	agasi_cole@email.com	0	3
9	ABLE	COLE	able_cole@email.com	0	3
10	RED	KINGS	red_kings@email.com	1	4
11	SEED	KINGS	seed_kings@email.com	0	4
12	TED	KINGS	ted_kings@email.com	0	4



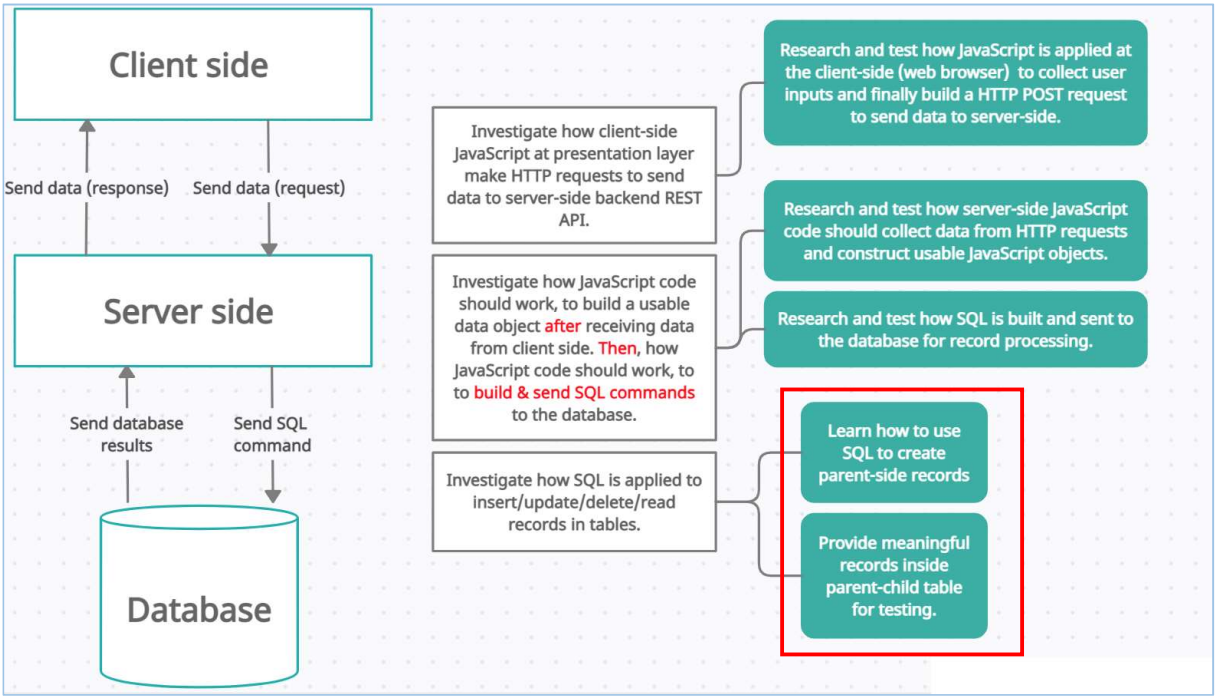
```
SELECT * FROM request_response_db.team_file;
```

team_file_id	cloudinary_file_id	cloudinary_url	original_filename	mime_type	team_id
1	teams/tpvzs7smvxh5jwgmoa7c	http://res.cloudinary.com/dit88888/i...	blue_blood_team_lo...	image/png	1
2	teams/ho236hckn4cmpxofw9wk	http://res.cloudinary.com/dit88888/i...	blue_blood_team_gr...	image/png	1
3	teams/tdmedolgtkutfbf9hef2	http://res.cloudinary.com/dit88888/i...	team_a_logo.png	image/png	2
4	teams/ifsn7lfz61akt53ambiz	http://res.cloudinary.com/dit88888/i...	team_a_logo.png	image/png	2
5	teams/kkzsm6odw9d7fmtz4nic	http://res.cloudinary.com/dit88888/i...	team_b_logo.png	image/png	3
6	teams/d0qkmrqcuivqiqx70yzu	http://res.cloudinary.com/dit88888/i...	team_b_logo.png	image/png	3
7	teams/rji4471tvummlxpvav78	http://res.cloudinary.com/dit88888/i...	syntax_error_logo.png	image/png	4
8	teams/f9llc9xegqwwn8rlyvrc	http://res.cloudinary.com/dit88888/i...	syntax_error_team_...	image/png	4
9	teams/qngdc0qxbybi5nabqvaa	http://res.cloudinary.com/dit88888/i...	soft_zombie.png	image/png	5
10	teams/mkw6bthzykgvbenafpec	http://res.cloudinary.com/dit88888/i...	soft_zombie_team_...	image/png	5

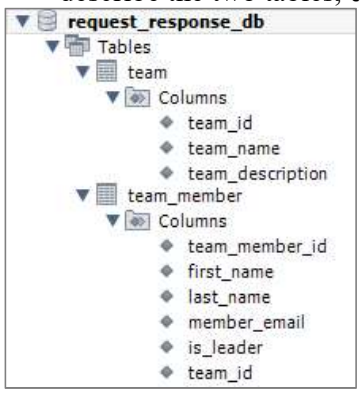
1. Familiarizing the Parent - Child record relationship in SQL

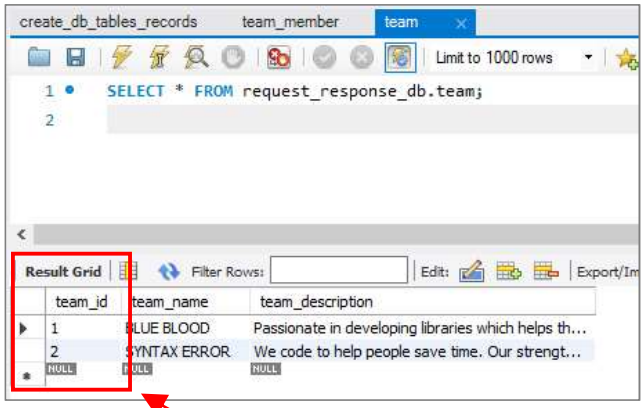
This section focuses on database SQL experiments.

- Parent - Child table relationship (using **team** and **team_member** table)
- Insert parent child record
- Select parent child record

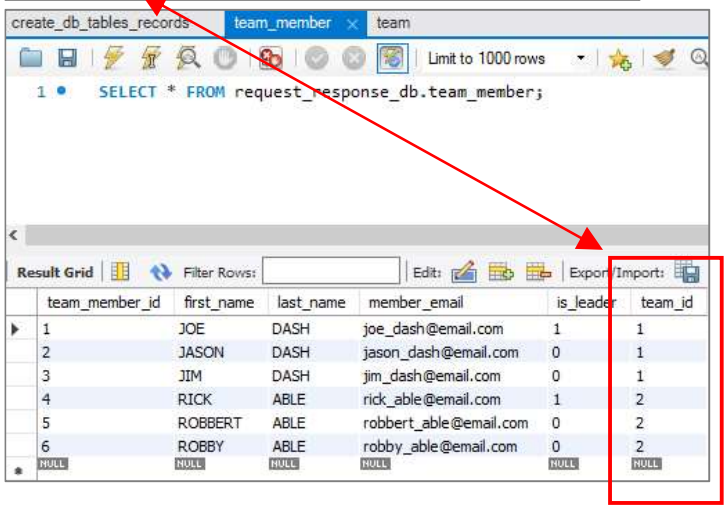


1. Execute the SQL script file, **create_db_tables_records.sql** which has the necessary SQL commands to create a new database **request_response_db**. The following three figures describe the two tables, **team** and **team_member** inside the database.





Parent table



Child table

The **team_member** table has 6 records. By visual inspection on the **team_id** field values of each **team_member** table record, you can easily notice that the first three records (JOE, JASON, JIM) are *linked* to the **team** record which describes "BLUE BLOOD" team. The next three records (RICK, ROBBERT and ROBBY) are *linked* to the team record which describes "SYNTAX ERROR".

2. Experiment INSERT parent-child records by using INSET SQL command first
You need to practice the SQL commands which will:

① Create one **team** record which describes 'SOFT ZOMBIE' team.

② Create three **team_member** records which describe AMANDA, AMY and ALICE who are team members of team 'SOFT ZOMBIE'.

Note that, the **team_member** record, AMANDA is a *team leader* for team 'SOFT ZOMBIE'. Other **team_member** records which describe ALICE and AMY are *normal team* members.

2.1 Apply the following SQL commands inside the MySQL Workbench environment to create the **parent-child records**.

```
INSERT INTO team(team_name, team_description)
 VALUES ('SOFT ZOMBIE','We develop softwares for people to edit cartoons. We aim to develop artificial intelligent tools to help users bring their imagination into the digital world in minutes.' );

INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
 VALUES ('AMANDA','KING','amanda_king@email.com',true, (SELECT team_id FROM team WHERE team_name='SOFT ZOMBIE') );

INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
 VALUES ('AMY','KING','amy_king@email.com',false, (SELECT team_id FROM team WHERE team_name='SOFT ZOMBIE') );

INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
 VALUES ('ALICE','KING','alice_king@email.com',false, (SELECT team_id FROM team WHERE team_name='SOFT ZOMBIE') );
```

The above SQL commands were derived from an article [INSERT with SELECT statement for columns with FOREIGN KEY constraint in MySQL with examples. | by Joshua Otwell | codeburst]

Note: There are other **more** useful techniques to create parent-child records inside the parent table and child table. The technique which uses SELECT technique is good enough to do the job for this worksheet exercise.

2.2 Understand the SQL command

When the database engine obtains the following INSERT SQL command:

```
INSERT INTO team(team_name, team_description) VALUES ('SOFT ZOMBIE','We develop softwares for people to edit cartoons. We aim to develop artificial intelligent tools to help users bring their imagination into the digital world in minutes.' );
```

The database engine recognizes the INSERT statement instruction, to create a new record inside the **team** table. The database engine automatically assigns a unique value of **3** for the new team record's **team_id** field.

team_id	team_name	team_description
1	BLUE BLOOD	Passionate in developing libraries which helps the community to build animation features.
2	SYNTAX ERROR	We code to help people save time. Our strength leans towards secure coding best practic...
3	SOFT ZOMBIE	We develop softwares for people to edit cartoons. We aim to develop artificial intelligent t...

When the database engine sees the next command:

```
INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
 VALUES ('AMANDA','KING','amanda_king@email.com',true, (SELECT team_id FROM team WHERE team_name='SOFT ZOMBIE') );
```

The database engine processes the highlighted command first.

The database engine will eventually treat the command as:

```
INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
 VALUES ('AMANDA','KING','amanda_king@email.com',true, 3 );
```

The highlighted section will be treated as 3 because the record which matches the query is associated to a unique **team_id** value of 3.

The value is assigned to the **team_id** field of the new **team_member** table record. The meaning of the value inside the **team_id** column which belongs to the **team_member** table, is *dependent* on the **team_id** primary key column of the **team** table. **Everytime** you look at the **team_member** table, you need to refer to the records inside the **team** table to find out, "to which team this AMANDA team member record belongs to?". The **team_id** column which belongs to the **team_member** table, is also known as "the **foreign key** column".

team_member_id	first_name	last_name	member_email	is_leader	team_id
1	JOE	DASH	joe_dash@email.com	1	1
2	JASON	DASH	jason_dash@email.com	0	1
3	JIM	DASH	jim_dash@email.com	0	1
4	RICK	ABLE	rick_able@email.com	1	2
5	ROBBERT	ABLE	robbert_able@email.com	0	2
6	ROBBY	ABLE	robby_able@email.com	0	2
7	AMANDA	KING	amanda_king@email.com	1	3

Eventually, when the database engine processes all the SQL commands, three child records are created in the **team_member** table. The three child records which describe team member information are associated to the parent team record which has a unique id value of 3.

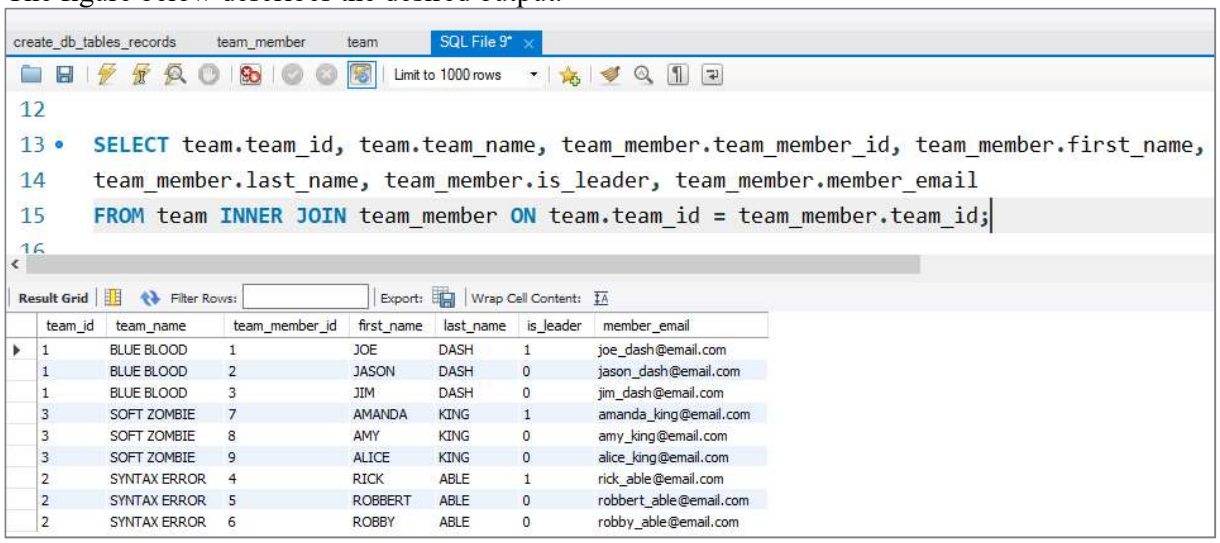
The database has captured the information that, there are three team members under the team, SOFT ZOMBIE.

2.3 Use SQL command to verify that the parent and child **table design** can indeed describe the team and team member information.

SELECT SQL statement is very useful to have an assurance that your parent and child table design is reliable. Check out the following SQL command which uses the INNER JOIN technique. Reference: https://www.w3schools.com/mysql/mysql_join_inner.asp

```
SELECT team.team_id, team.team_name, team_member.team_member_id,
team_member.first_name, team_member.last_name, team_member.is_leader,
team_member.member_email FROM team INNER JOIN team_member ON team.team_id =
team_member.team_id;
```

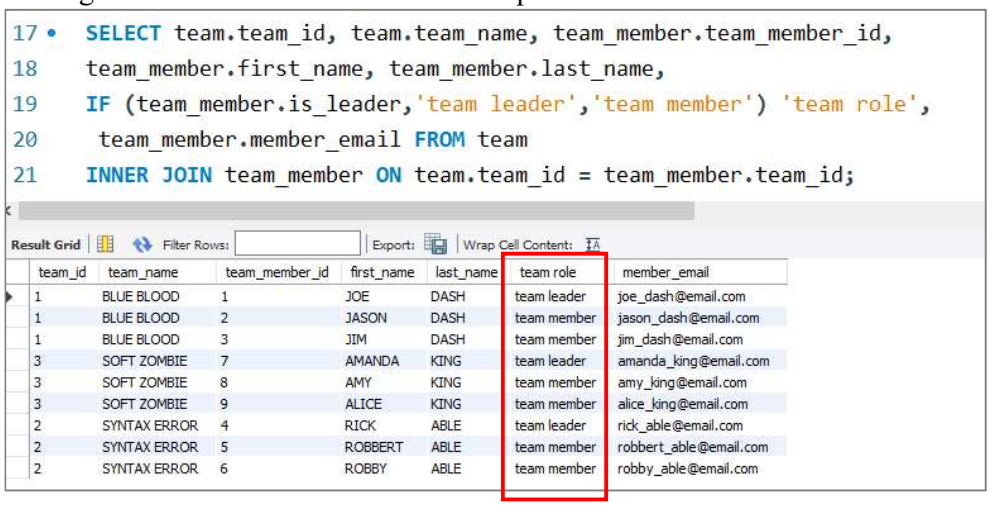
The figure below describes the desired output.



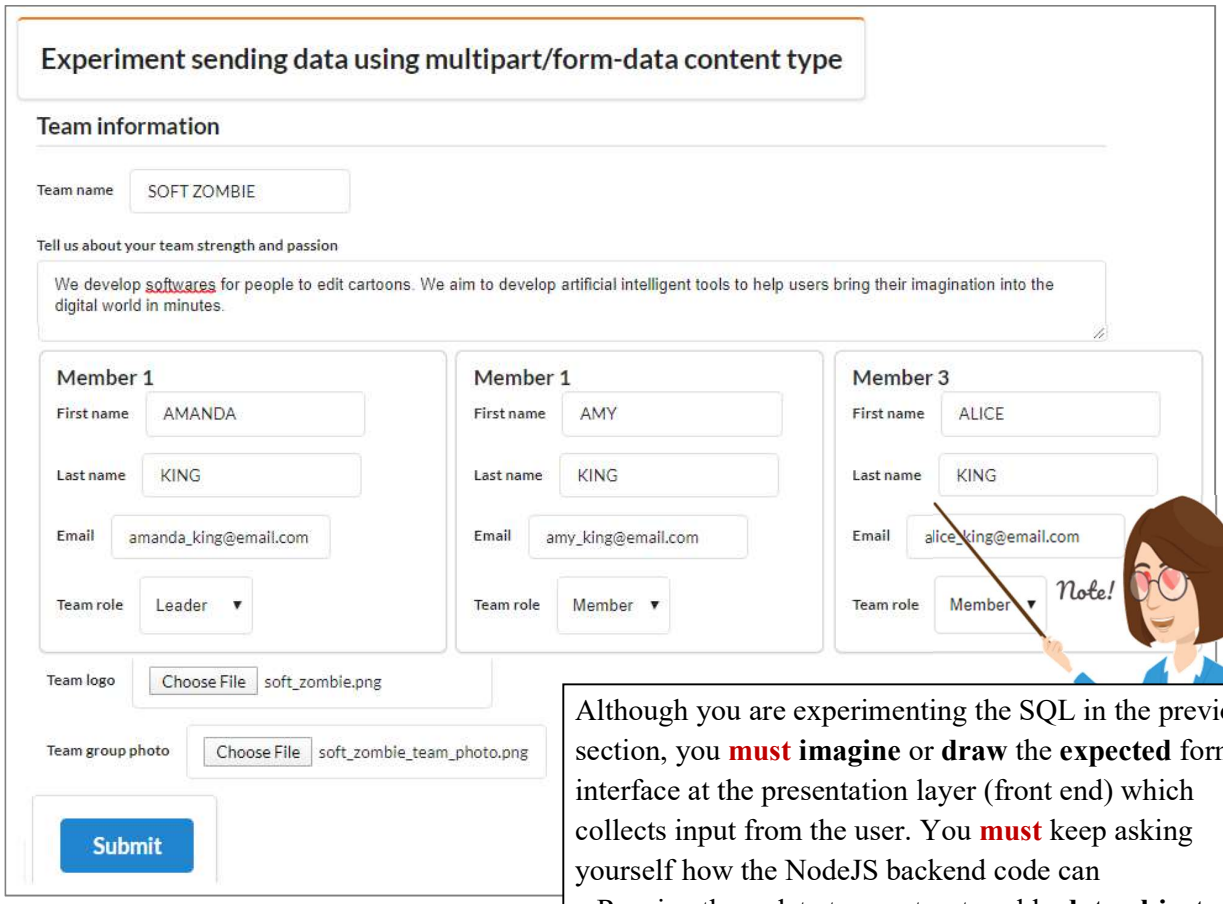
Maybe you need an output which is more "self-descriptive". You can refer to the tutorial at https://www.w3schools.com/sql/func_mysql_if.asp and implement the SQL IF technique inside the previous SQL statement.

```
SELECT team.team_id, team.team_name, team_member.team_member_id,
team_member.first_name, team_member.last_name, IF (team_member.is_leader,'team
leader','team member'), team_member.member_email FROM team INNER JOIN team_member
ON team.team_id = team_member.team_id;
```

The figure below describes the desired output.

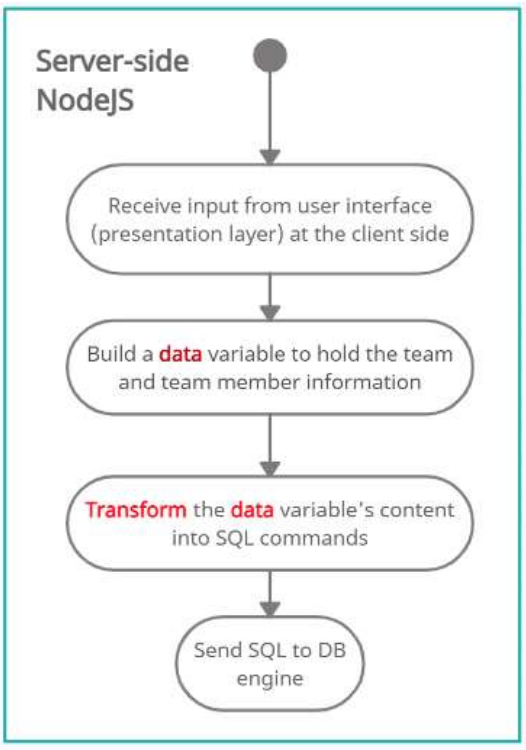


2. Experiment JavaScript data object and how it can be used to create records inside the database.



Although you are experimenting the SQL in the previous section, you **must** **imagine** or **draw** the **expected** form interface at the presentation layer (front end) which collects input from the user. You **must** keep asking yourself how the NodeJS backend code can

- Receive these data to construct usable **data object**
- Use the constructed data object to build the SQL.



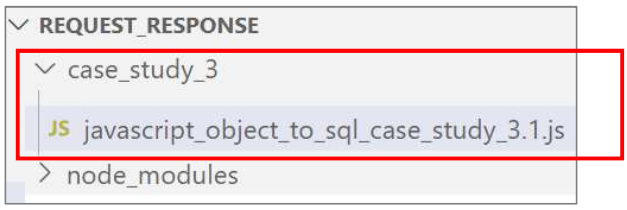
In this section:

①You will investigate how to build a **data** variable to hold the team and team information.

②You will make sure you can easily access information from this **data** variable to construct usable SQL commands.

Collecting data from the client-side is your **last** priority.

1. Create a directory inside the project, **case_study_3**. Then create a new JavaScript file, **javascript_object_to_sql_case_study_3.1.js**



This exercise is important. You need some fundamentals to appreciate the importance of this exercise. Watch the video at https://youtu.be/YztMpkuMP4Q to have some background fundamentals **before** you work on this exercise.

2. Refer to the two code listing below, provide the necessary JavaScript commands for the **buildCreateTeamAndTeamMemberSQL** function so that the **data** object's content can be translated into applicable SQL commands which you have been working on in the previous experiments.

```
//Assume that the, your NodeJS backend project
//is able to build a data object below which
//describes team and team member data.
//Construct the necessary SQL commands which are applicable
//for creating the team-team member parent-child records.
const data = {
  teamName: 'SOFT ZOMBIE',
  teamDescription: `We develop softwares for people to edit cartoons.
  We aim to develop artificial intelligent tools to help users
  bring their imagination into the digital world in minutes.`,
  firstName: ['AMANDA', 'AMY', 'ALICE'],
  lastName: ['KING', 'KING', 'KING'],
  email: [
    'amanda_king@email.com',
    'amy_king@email.com',
    'alice_king@email.com'
  ],
  isLeader: ['true', 'false', 'false']
}
console.log(buildCreateTeamAndTeamMemberSQL(data));
```

Assume that your NodeJS project's code can *structure* the team and team member information in this format within the **data** variable (refer to video https://youtu.be/YztMpkuMP4Q). You need to **be sure that**, such structure can be translated to the SQL commands which is recognizable by the database engine.

```
function buildCreateTeamAndTeamMemberSQL(data) {
  let sql = '';
  sql = sql + `INSERT INTO team(team_name, team_description)
   VALUES ('${data.teamName}','${data.teamDescription}' );\n`;
  for (let index = 0; index < 3; index++) {
    sql = sql +
      `INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
     VALUES ('${data.firstName[index]}','${data.lastName[index]}','${data.email[index]}',
     ${data.isLeader[index]},
     (SELECT team_id FROM team WHERE team_name='${data.teamName}') );\n`;
  }
  return sql;
}
```

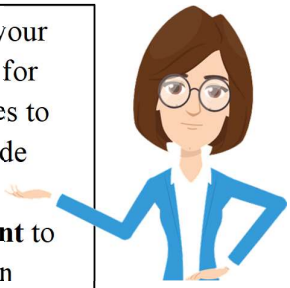
3. Use the command, **cd case_study_3**. Then, execute the JS file. If the **buildCreateTeamAndTeamMemberSQL** function logic is indeed working, you should see the following desired output in the terminal interface.
   You should see:
   ① 1 SQL command which inserts a **team** table record.
   ② 3 SQL commands which insert 3 **team_member** table record.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    2: powershell

PS D:\reactjs_nodejs_mysql\request_response\case_study_3> node javascript_object_to_sql_case_study_3.1.js
INSERT INTO team(team_name, team_description)
      VALUES ('SOFT ZOMBIE','We develop softwares for people to edit cartoons.
    We aim to develop artificial intelligent tools to help users
    bring their imagination into the digital world in minutes.' );
INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
          VALUES ('AMANDA','KING','amanda_king@email.com',
          true,
          (SELECT team_id FROM team WHERE team_name='SOFT ZOMBIE') );
INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
          VALUES ('AMY','KING','amy_king@email.com',
          false,
          (SELECT team_id FROM team WHERE team_name='SOFT ZOMBIE') );
INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
          VALUES ('ALICE','KING','alice_king@email.com',
          false,
          (SELECT team_id FROM team WHERE team_name='SOFT ZOMBIE') );
```

4. Then, you **copy** the SQL commands seen inside the terminal, and **paste** them into the MySQL workbench environment to test the command.

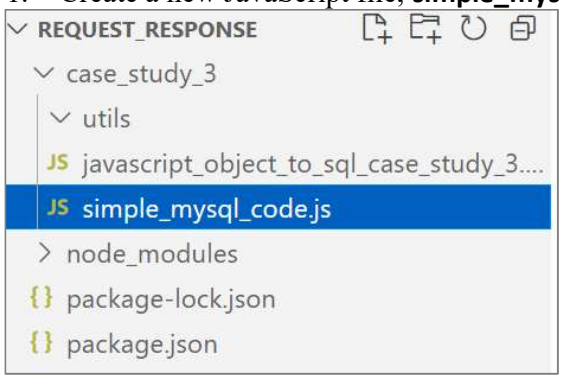
4.1 Inspect the records inside the **team** table and the **team_member** table. If you can see the parent-child records created in the respective **team** and **team_member** table, you have managed built a proper **buildCreateTeamAndTeamMemberSQL** function (method) logic which **translates a data object's content into usable SQL commands**.

Caution: When you involve database related objects in your code, which *sends* these SQL commands to the database for generating records, you still need to make a lot of changes to the code which you have been working on so far. The code inside the JavaScript function, **buildCreateTeamAndTeamMemberSQL** is **only sufficient** to assure you that, if the team and team member information organized in the **data** variable with this structure, you can access them by using code and build usable SQL. With this assurance, you can **proceed** to the next step, which is to work on database operations commands.



3. Experiment how JavaScript can work with objects obtained from MySQL package library to perform database operations

1. Create a new JavaScript file, **simple_mysql_code.js** inside the **case_study_3** directory.



Note: You need to frequently apply **DELETE from team_member** and **DELETE from team;** when you test the following commands inside the **simple_mysql_code.js** file.
Acquire database fundamentals through this video at **https://youtu.be/B9jcsp7acdg** before working on the following commands.

2. Watch the lesson video **https://youtu.be/ts9JC97iEoA** which explains the code below.
   The video covers:
   ① The concept on the **mysql** object variable which represents the MySQL database engine.
   ② The concept on the **pool** object variable which is created by calling the **createPool** method which belongs to the **mysql** object variable.
   ③ The concept on the **getConnection** method which belongs to the **pool** object variable.

3. Provide the commands for the **simple_mysql_code.js** by referring to the code listing below:

```
const mysql = require('mysql');
let dbConfig = {
  connectionLimit: 100,
  host: 'localhost',
  user: 'request_response_db_adminuser',
  password: 'password123',
  database: 'request_response_db',
  multipleStatements: false
};

const pool = mysql.createPool(dbConfig);
let count = 1;
```

Obtain an object which can communicate with the MySQL database from the mysql library. Then set the **mysql** variable to reference that object.
As a result, **mysql** variable has the internal logic to communicate with the database.

Create an object of information consisting of property-value pair information.
These property-value pair information can be recognized by the **mysql** object later to create a "channel" to the database.

You won't see any JavaScript commands using this **count** variable. But the variable will be used in subsequent experiments to experience the **asynchronous JavaScript behavior**.

```
pool.getConnection(function(error, connection) {
  if (error) {
    throw error;
  }
  connection.query(`INSERT INTO team(team_name, team_description)
    VALUES ('TEAM D','TEAM D DESCRIPTION')`, function(error, results) {
    console.log('INSERT TEAM D record has completed.');
  });

  connection.query(`SELECT * FROM team`, function(error, results) {
    if (error) {
      throw error;
    }
    console.log('SELECT team records has completed.');
  });


connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
      VALUES ('BRIZ','QUEENS','briz_queens@email.com',
      false, (SELECT team_id FROM team WHERE team_name='TEAM D') );`,
    function(error, result) {
      if (error) {
        throw error;
      }
      console.log('(INSERT member 1) INSERT team member briz queens for team D has completed.')
    });
 connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
      VALUES ('BRADDY','QUEENS','braddy_queens@email.com',
      false, (SELECT team_id FROM team WHERE team_name='TEAM D') );`,
    function(error, result) {
      if (error) {
        throw error;
      }
      console.log('(INSERT member 2) INSERT team member braddy queens for team D has completed.')
    });
```

```
  connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
      VALUES ('BOB','QUEENS','bob_queens@email.com',
      false, (SELECT team_id FROM team WHERE team_name='TEAM D') );`,
    function(error, result) {
      if (error) {
        throw error;
      }
      console.log('(INSERT member 3) INSERT team member bob queens for team D has completed.');
  });


}); // End of pool.getConnection
```

4. Execute the **simple_mysql_code.js** file. Refer to the video **https://youtu.be/PGsXbhyoGd8**

4. Experiment two major problems. Problem 1: How database integrity is affected (database content is no longer reliable and trustworthy) when your JavaScript code fails to provide a complete database operation. Problem 2: Asynchronous JavaScript behavior

Begin the experiment by having the following important assumption in your mind first.

The database was designed to capture a **complete** team-team member information whereby each team **must have** exactly **three** team member records. In this section, you are providing JavaScript commands which sends SQL instructions which eventually produce an undesired one team - two team member record data combination. This is due to the database engine's rejection on the last SQL instruction which tries to create the third team member record which has duplicate email information, '**bob_queens@email.com**'.

1. The video at **https://youtu.be/mAKv6S36sjg** covers the following topics:

1.1 Introduction

1.2 **https://youtu.be/mAKv6S36sjg?t=41** ➔ Backup your **simple_mysql_code.js** JavaScript file by copying the code to a new file, **simple_mysql_code_version_1.js**. You need to refer to the code listing below, and make changes to the string input so that you can hardcode the SQL to:

① Insert one new team record (TEAM C) into the **team** table.

② Insert three new team member records into the **team_member** table and associate these records to TEAM C parent record.

Important note:

You will notice that, the SQL string input for the last **query** method call, shall get the database to create a team member record which has a duplicate email information, 'bob_queens@email.com'.

When you test this JS file later, a runtime error is expected to occur because the **team_member** table has a unique constraint rule applied on the **member_email** field.

```
pool.getConnection(function(error, connection) {
  if (error) {
    throw error;
  }
  connection.query(`INSERT INTO team(team_name, team_description)
    VALUES ('TEAM C','TEAM C DESCRIPTION')`, function(error, results) {
    console.log('INSERT TEAM C record has completed.');
  });
  connection.query(`SELECT * FROM team`, function(error, results) {
    if (error) {
      throw error;
    }
    console.log('SELECT team records has completed.');
  });
  connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
      VALUES ('FRED','HANS','fred_hans@email.com',
      false, (SELECT team_id FROM team WHERE team_name='TEAM C') );`,
    function(error, result) {
      if (error) {
        throw error;
      }
      console.log('(INSERT member 1) INSERT team member fred hans for team C has completed.');
    });
```

```
connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
      VALUES ('FRANK','HANS','frank_hans@email.com',
      false, (SELECT team_id FROM team WHERE team_name='TEAM C') );`,
    function(error, result) {
      if (error) {
        throw error;
      }
      console.log('(INSERT member 2) INSERT team member frank hans for team C has completed.');
    });
  connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id
)
      VALUES ('BOB','QUEENS','bob_queens@email.com',
      false, (SELECT team_id FROM team WHERE team_name='TEAM C') );`,
    function(error, result) {
      if (error) {
        throw error;
      }
      console.log('(INSERT member 3) INSERT team member bob queens for team C has completed.');
    });
}); // End of pool.getConnection
```

Don't change the team member information which describes BOB QUEENS which has 'bob_queens@email.com'. You are going to **purposely force** the database engine to reject this SQL instruction.

1.3 The https://youtu.be/mAKv6S36sjg?t=230 ➔ Illustrates the intention of having the last SQL instruction which causes a database error in this experiment.

2. Test the JavaScript file to **experience the first problem**
https://youtu.be/mAKv6S36sjg?t=421
The video at https://youtu.be/mAKv6S36sjg?t=424 covers the fundamental knowledge which is required to appreciate the first key problem in your JavaScript code which handles database operation to create team C and team C member data:

https://youtu.be/mAKv6S36sjg?t=481 ➔ The problem begins when the **team_member** table has a unique constraint on the **member_email** column.
https://youtu.be/mAKv6S36sjg?t=515 ➔ Describes the database engine rejecting the SQL INSERT instruction to create the 3rd team member record.
https://youtu.be/mAKv6S36sjg?t=563 ➔ Describes the incomplete team-team member data capturing which has compromised the database integrity. Which means that, the database's data accuracy cannot be trusted anymore because it was designed to capture complete **1 team - 3 team member** record data.
☒ Database's integrity (database's accuracy in data) has been compromised

3.1 To appreciate **the second problem**, which is the JavaScript asynchronous behavior which can create a headache, you need to backup all the JavaScript commands which creates Team C and Team C member data into **simple_mysql_code_version_2.js**.

3.2 Replace all the code JavaScript file **simple_mysql_code.js** by using the code obtained from **simple_mysql_code_version_1.js** which creates Team D and Team D member data. We will use those commands to observe the **second problem**
Watch this supplement video **https://youtu.be/RnWFxpku2IE** first, before beginning the experiment for the second problem "Asynchronous JavaScript behavior"

	Simple_mysql_code.js code for observing count variable to "**feel**" asynchronous behavior while creating team D and team D member records in the database
1	`//----------`
2	`// Topic:`
3	`//----------`
4	`//Experiment two major problems.`
5	`//Problem 2: [Asynchronous JavaScript behavior]`
6	
7	`//The following code creates complete Team D and Team D member data.`
8	`//You have to keep deleting Team D and Team D member records before`
9	`//you retest the JS file.`
10	`//The code focuses on watching the count variable value changes to`
11	`//feel the asynchronous behavior`
12	`//----------`
13	`const mysql = require('mysql');`
14	`let dbConfig = {`
15	`  connectionLimit: 100,`
16	`  host: 'localhost',`
17	`  user: 'request_response_db_adminuser',`
18	`  password: 'password123',`
19	`  database: 'request_response_db',`
20	`  multipleStatements: false`
21	`};`
22	`const pool = mysql.createPool(dbConfig);`
23	`let count = 0;`
24	`pool.getConnection(function(error, connection) {`
25	`  if (error) {`
26	`  throw error;`
27	`  }`
28	`connection.query(`INSERT INTO team(team_name, team_description)`
29	`  VALUES ('TEAM D','TEAM D DESCRIPTION')`, function(error, results) {`
30	`  count = count + 1; //JavaScript engine already executed the line : 70`
31	`  console.log('INSERT TEAM D record has completed.');`
32	`  });`
33	`connection.query(`SELECT * FROM team`, function(error, results) {`
34	`  if (error) {`
35	`  throw error;`
36	`  }`
37	`  count = count + 1; //JavaScript engine already executed the line : 70`
38	`  console.log('SELECT team records has completed.');`

```
39      });
40  connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
41          VALUES ('BRIZ','QUEENS','briz_queens@email.com',
42          false, (SELECT team_id FROM team WHERE team_name='TEAM D') );`,
43        function(error, result) {
44          if (error) {
45            throw error;
46          }
47          count = count + 1; //JavaScript engine already executed the line : 79
48          console.log('(INSERT member 1) INSERT team member fred hans for team D has completed.');
49        });
50     connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
51        VALUES ('BRADDY','QUEENS','braddy_queens@email.com',
52        false, (SELECT team_id FROM team WHERE team_name='TEAM D') );`,
53        function(error, result) {
54          if (error) {
55            throw error;
56          }
57          count = count + 1; //JavaScript engine already executed the line :70
58          console.log('(INSERT member 2) INSERT team member frank hans for team D has completed.');
59        });
60  connection.query(`INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
61        VALUES ('BOB','QUEENS','bob_queens@email.com',
62        false, (SELECT team_id FROM team WHERE team_name='TEAM D') );`,
63        function(error, result) {
64          if (error) {
65            throw error;
66          }
67          count = count + 1; //JavaScript engine already executed the line : 702
68          console.log('(INSERT member 3) INSERT team member bob queens for team D has completed.');
69        });
70     console.log(count); //JavaScript engine executes this "far too early"
71
72  }); // End of pool.getConnection
```

https://youtu.be/mAKv6S36sjg?t=655 ➔ Advices you to delete the team D and the two team D record data by using some SQL DELETE instructions. So that, you can repeat the test on the JS file, which inserts the same team D and the team D data.

https://youtu.be/mAKv6S36sjg?t=696 ➔ Describes the necessary changes you need to have on your code inside the **simple_mysql_code.js.**

https://youtu.be/mAKv6S36sjg?t=904 ➔ Describes the experiment observations "JavaScript engine does not wait".

4. **Conclusion**

   https://youtu.be/mAKv6S36sjg?t=1097 explains the objective of the commands which you need to implement in the next session.

5. Strengthen your fundamentals on database design and the database's behavior on ROLLBACK, COMMIT feature by referring to this video https://youtu.be/l6qP18SJJRc

5. Using MySQL Workbench to practice on the concept START TRANSACTION, COMMIT and ROLLBACK

1. Ensure that the database table has the following records for the **team** and the **team_member** table. You can open the **create_database_tables_records_with_team_d_records.sql** to reset or prepare the database. The difference between **create_database_tables_records_with_team_d_records.sql** and **create_database_tables_records.sql** is, the **create_database_tables_records_with_team_d_records.sql** delete-create a new database which has complete team d and team d member records, so that learners can conveniently apply the SQL script file *before* testing how the backend code can work with database *when* duplicate team member email record is detected.

team_id	team_name	team_description
1	BLUE BLOOD	Passionate in developing libraries which helps the community to build animation features.
2	SYNTAX ERROR	We code to help people save time. Our strength leans towards secure coding best practic...
3	TEAM D	TEAM D DESCRIPTION

team_member_id	first_name	last_name	member_email	is_leader	team_id
1	JOE	DASH	joe_dash@email.com	1	1
2	JASON	DASH	jason_dash@email.com	0	1
3	JIM	DASH	jim_dash@email.com	0	1
4	RICK	ABLE	rick_able@email.com	1	2
5	ROBBERT	ABLE	robbert_able@email.com	0	2
6	ROBBY	ABLE	robby_able@email.com	0	2
7	BRIZ	QUEENS	briz_queens@email.com	0	3
8	BRADDY	QUEENS	braddy_queens@email.com	0	3
9	BOB	QUEENS	bob_queens@email.com	0	3

2. Open an SQL script file inside the MySQL Workbench environment. The script file name is, **create_team_C_team_C_member_records_with_rollback_feature.sql**

3. Refer to the **https://youtu.be/z0fXS2FzKw8** which covers:

① START TRANSACTION
② COMMIT
③ ROLLBACK

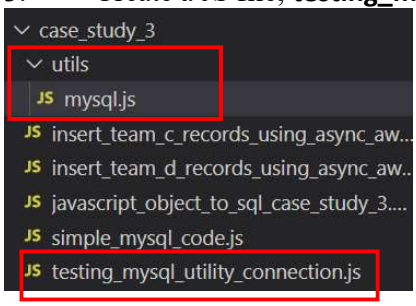
The video **https://youtu.be/z0fXS2FzKw8** uses the following are SQL commands inside the **create_team_C_team_C_member_records_with_rollback_feature.sql** file to demonstrate database engine behavior on START TRANSACTION, ROLLBACK and COMMIT. The characteristics *influences* how you code JavaScript commands inside a NodeJS project.

- **Do not** execute the SQL commands all at once. **Follow the video** demonstration to execute the commands line by line.
- Ensure that database **already has** one Team D and three Team D member records before experimenting the concepts covered inside the video.

```
-- topic:
-- Using MySQL Workbench to practice on the concept
-- START TRANSACTION, COMMIT and ROLLBACK

START TRANSACTION;
-- Once the DB engine sees the START TRANSACTION instruction:
-- All the 4 commands "after" the START TRANSACTION command
-- will be treated by the database engine as a single unit of work.
-- Which means that, if any of the SQL command has error,
-- you can execute the ROLLBACK command to "rollback" to the original state.
-- (nothing has happened)
INSERT INTO team(team_name, team_description)
    VALUES ('TEAM C','TEAM C DESCRIPTION' );
INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
       VALUES ('FRED','HANS','fred_hans@email.com',
       true,
       (SELECT team_id FROM team WHERE team_name='TEAM C') );
INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
       VALUES ('FRANK','HANS','frank_hans@email.com',
       false,
       (SELECT team_id FROM team WHERE team_name='TEAM C') );
INSERT INTO team_member(first_name,last_name, member_email,is_leader, team_id)
       VALUES ('BOB','QUEENS','bob_queens@email.com',
       false,
       (SELECT team_id FROM team WHERE team_name='TEAM C') );

-- If you want the database engine to ROLLBACK to the original state
-- (nothing has happened), then you need to use the SQL command
-- ROLLBACK;
ROLLBACK;

-- If you want the database engine to make all recent changes
-- permanent, then you need to use the SQL command
-- ROLLBACK;
COMMIT;
```

6. Produce a reliable JavaScript logic which has transactional feature and has synchronized database operation feature.

1. Inside the **case_study_3** directory, create a directory **utils**.
2. Create a JS file, **mysql.js** inside the **utils** directory.
3. Create a JS file, **testing_mysql_utility_connection.js** inside the **case_study_3** directory.



4. Refer to the code listing below. **Copy** the given commands into the **mysql.js** JS file.
   https://youtu.be/-ex3XO6Mkz4
   Caution: The author has carefully studied the official technical documentation of the mysql package library at the npm site and the GitHub site. Originally, the frequently used database object method usage such as pool.getConnection, connection.query and the connection.release are asynchronous. **You have experienced the asynchronous behavior in the previous exercise.** She has applied the Promise object technique inside the **mysql.js** file, so that both beginners and seasoned developers can call them **synchronously**. The code inside the **MySQL.JS** file is stable. She has helped you ***indirectly*** use MySQL database objects synchronously. But the code inside the **mysql.js** file is not an appropriate learning reference for beginners who are still *new* to Promise object unless they have tried Promise object techniques *countless times* and read the official documentation of mysql library package *countless times*.

```
//The mysql.js utility function (helper function)
//was created by referencing a precious article at
//https://medium.com/wenchin-rolls-around/example-of-using-transactions-with-async-await-via-mysql-connection-pool-9a37092f226f
const mysql = require('mysql');
let dbConfig = {
connectionLimit: 100,
host: 'localhost',
user: 'request_response_db_adminuser',
password: 'password123',
database: 'request_response_db',
multipleStatements: false
};
const pool = mysql.createPool(dbConfig);
  //Create an anonymous function and assign it to the connection
const connection = function() {
return new Promise(function(resolve, reject) {
   //Create a Promise object so that the calling program can wait for the getConnection method to finish.
  //The getConnection method is given a "call back function" so that the internal logic of the getConnection
  //executes the call back function's logic when an active connection is established with the database.
  pool.getConnection(function(error, connection) {
     if (error) {
```

```
      reject(error);
    }// End of if(error)
    console.log('MySQL pool connected: threadId ' + connection.threadId);
    //Create an anonymous function and assign it to the query. Note that, the query "belongs to"
    //the getConnection's callback function
    const query = function(sql, binding) {
          return new Promise(function(resolve, reject) {
            connection.query(sql, binding, function(error, results) {
              if (error) {
                  reject(error);
              }// End of if(error)
            resolve(results);
          });//End of Promise object "inside" the anonymous function which is given to the query
      });//End of anonymous function definition for the query
    };//End of const query = <anonymous function definition>
    //Create an anonymous function and assign it to the release. Note: The release "belongs" to the
    //getConnection's callback function
    const release = function() {
      //The anonymous function's body has a Promise object created, so that the calling program
      //can wait for the connection.release to finish
      return new Promise(function (resolve, reject) {
          if (error) {
            reject(error);
          }//End of if(error)
          console.log('MySQL pool released: threadId ' + connection.threadId);
          resolve(connection.release());
      });// End of Promise object "inside" the anonymous function which is assigned to the release
    };// End of the anonymous function's body which is assigned to the release.
      //End of const release = <anonymous function definition>
    resolve({ query, release });
    });//End of the Promise object definition "inside" the callback function which is
      //given (provided) to the getConnection
    });//End of the Promise object definition which is created inside the anonymous function which is
      //assigned to the connection
};// End of the anonymous function which is given to the connection
const query = function (sql, binding) {
          return new Promise((resolve, reject) => {
            pool.query(sql, binding, function (error, results, fields)  {
            if (error) {
              reject(error);
            }
          resolve(results);
          });// End of "call back function" which is provided to the query method.
      });// End of Promise object definition which is created inside the anonymous function
};// End of the anonymous function which is assigned to the query
module.exports = { pool, connection, query };
```

Online article reference of Wenchin author: https://medium.com/wenchin-rolls-around/example-of-using-transactions-with-async-await-via-mysql-connection-pool-9a37092f226f

5. Open the **testing_mysql_utility_connection.js**. Refer to the video lesson (30 minutes) **https://youtu.be/-ex3XO6Mkz4?t=39** to provide the commands (next page) **with understanding.**

```
const mysql = require('./utils/mysql');
const data = {
  teamName: 'TEAM D',
  teamDescription: `TEAM D Description`,
  firstName: ['BRIZ', 'BRADDY', 'BOB'],
  lastName: ['QUEENS', 'QUEENS', 'QUEENS'],
  email: [
    'briz_queens@email.com',
    'braddy_queens@email.com',
    'bob_queens@email.com'
  ],
  isLeader: ['true', 'false', 'false']
};

async function createTeamAndTeamMemberData(data) {
  const connection = await mysql.connection();
  try {
    await connection.query('START TRANSACTION');
    const createTeamResults = await connection.query(`INSERT INTO team
    (team_name,team_description)
    VALUES (? , ?)`, [data.teamName, data.teamDescription]);
    await connection.query(`INSERT INTO team_member
    (first_name,last_name,member_email,is_leader,team_id)
    VALUES (?,?,?,?,?)`, [data.firstName[0], data.lastName[0], data.email[0],
      data.isLeader[0], createTeamResults.insertId
    ]);
    await connection.query(`INSERT INTO team_member
    (first_name,last_name,member_email,is_leader,team_id)
    VALUES (?,?,?,?,?)`, [data.firstName[1], data.lastName[1], data.email[1],
      data.isLeader[1], createTeamResults.insertId
    ]);
    await connection.query(`INSERT INTO team_member
    (first_name,last_name,member_email,is_leader,team_id)
    VALUES (?,?,?,?,?)`, [data.firstName[2], data.lastName[2], data.email[2],
      data.isLeader[2], createTeamResults.insertId
    ]);

    await connection.query('COMMIT');

  } catch (error) {
    if (error) {
      console.log(error);
      await connection.query('ROLLBACK');
    }
  } finally {
    await connection.release();

  } // End of try-catch-finally block
} // End of createTeamAndTeamMemberData

// Call the createTeamAndTeamMemberData function
// Provide the data variable which has the team-team member data as input for the
// function call.
createTeamAndTeamMemberData(data);
```

This property-value structure was experimented in the previous exercises. You are involving this information structure here so that the **createTeamAndTeamMemberData** function logic can use the data structure to prepare a more dynamic SQL instruction. Which means, no more hard code SQL commands.

data.email[2] will be evaluated as 'bob_queens@email.com'
data.isLeader[0] will be treated as 'true'.
data.teamName will be evaluated as 'TEAM D'
data.firstName[1] will be treated as 'BRADDY'

There is a typo error in the video session. Use 'START TRANSACTION'

The database engine will reject this SQL instruction to create the first team member record.

This is done on purpose to: •Examine the rollback capability to protect the database integrity. •Learn inline IF technique.

This command prints the error details inside the terminal console. You need this command to inspect the database runtime error.

6. Execute the SQL script file, **create_db_tables_records.sql** to obtain a "fresh" database. https://youtu.be/-ex3XO6Mkz4?t=1352 The **team** table should have 2 records and the **team_member** table should have 6 records.

7. Test the JavaScript file. https://youtu.be/-ex3XO6Mkz4?t=1376 Remember to CTRL + C to manually terminate the program.

7.1 A runtime error which is caused by database engine. https://youtu.be/-ex3XO6Mkz4?t=1506 You should see the following output inside the terminal console. This is due to the **data.isLeader[0]** which gives a **string** value of "true". The database engine is expecting a numeric value of 1 for the **is_leader** field in the team member record. The **is_leader** column in the **team_member** table is a boolean data type column.

```
code: 'ER_TRUNCATED_WRONG_VALUE_FOR_FIELD',
errno: 1366,
sqlMessage: "Incorrect integer value: 'true' for column 'is_leader' at row 1",
sqlState: 'HY000',
index: 0,
sql: 'INSERT INTO team_member \n' +
  '        (first_name,last_name,member_email,is_leader,team_id) \n' +
  "        VALUES ('BRIZ','QUEENS','briz_queens@email.com','true',3)"
```

7.2 Please inspect the **team** table. The database engine **did not** make the TEAM D record creation **permanent** because the JavaScript engine has executed the `await connection.query('ROLLBACK')` command to tell the database engine to abort all the recent changes.

Observation notes:

① The database engine has received an SQL instruction to create a new TEAM D record in the **team** table due to the earlier JavaScript command.

② The database engine then, received the SQL command to create the first team member record *after* the new TEAM D record has been created.

③ When the database engine rejects the SQL command due to data type problem on the **is_leader** column for the **team_member** table, the database engine has responded a database error message to your JavaScript program.

④ The JavaScript engine has automatically **jumped** from the try block into the `catch(error)` block and executed the command `await connection.query('ROLLBACK')`.

8. Fix the code in the **testing_mysql_utility_connection.js** file by referring to the partial code listing below. Also, you will learn how **inline IF technique** is applied during this coding activity.
   https://youtu.be/-ex3XO6Mkz4?t=1580

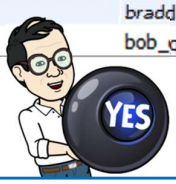
Inline IF technique is applied to convert "true" to 1 and "false" to 0.

```
await connection.query(`INSERT INTO team_member
(first_name,last_name,member_email,is_leader,team_id)
VALUES (?,?,?,?,?)`, [data.firstName[0], data.lastName[0], data.email[0],
   (data.isLeader[0] == 'true') ? 1 : 0,createTeamResults.insertId
]);
await connection.query(`INSERT INTO team_member
(first_name,last_name,member_email,is_leader,team_id)
VALUES (?,?,?,?,?)`, [data.firstName[1], data.lastName[1], data.email[1],
   (data.isLeader[1] == 'true') ? 1 : 0,createTeamResults.insertId
]);
await connection.query(`INSERT INTO team_member
(first_name,last_name,member_email,is_leader,team_id)
VALUES (?,?,?,?,?)`, [data.firstName[2], data.lastName[2], data.email[2],
   (data.isLeader[2] == 'true') ? 1 : 0,createTeamResults.insertId
]);
```

9. Test the JavaScript file. Manually terminate the program after the execution. Check the **team** and the **team_member** table. The figure below describes the expected results. https://youtu.be/-ex3XO6Mkz4?t=1756

team_id	team_name	team_description
1	BLUE BLOOD	Passionate in developing libraries which helps the community to build animation features.
2	SYNTAX ERROR	We code to help people save time. Our strength leans towards secure coding best practic...
4	TEAM D	TEAM D Description

team_member_id	first_name	last_name	member_email	is_leader	team_id
1	JOE	DASH	joe_dash@email.com	1	1
2	JASON	DASH	jason_dash@email.com	0	1
3	JIM	DASH	jim_dash@email.com	0	1
4	RICK	ABLE	rick_able@email.com	1	2
5	ROBBERT	ABLE	robbert_able@email.com	0	2
6	ROBBY	ABLE	robby_able@email.com	0	2
7	BRIZ	QUEENS	briz_queens@email.com	1	4
8	BRADDY	QUEENS	braddy_queens@email.com	0	4
9	BOB	QUEENS	bob_queens@email.com	0	4



Just a bit more to tidy up the files and code.

Refer to the video instruction at to tidy up the files and code. So that you are **ready** for the **next phase of the web application backend development topic**: The REST API which collects the team-team member data from the client-side web browser (or Postman).

7. Tidying up your work.

Video reference: https://youtu.be/FYRkSdiC7XI

1. Copy all the working code from the **testing_mysql_utility_connection.js** file into a new JS file, **insert_team_d_records_using_async_await_transaction.js**. Save the file. No changes are required. This file shall contain the working commands for your future revision or experiments on other database operation challenges.

2. Copy all the working code from the **testing_mysql_utility_connection.js** file into a new JS file, **insert_team_c_records_using_async_await_transaction.js**. Make changes to the code in the new JS file by referring to the following partial code listing.

```
const data = {
  teamName: 'TEAM C',
  teamDescription: `TEAM C Description`,
  firstName: ['FRED', 'FRANK', 'BOB'],
  lastName: ['HANS', 'HANS', 'QUEENS'],
  email: [
    'fred_hans@email.com',
    'frank_hans@email.com',
    'bob_queens@email.com'
  ],
  isLeader: ['true', 'false', 'false']
};
```

This **data** variable's information shall one database error within the **createTeamAndTeamMemberData** function. The database engine will detect the 3rd member record's 'bob_queens@email.com' will violate the unique constraint rule which was set on the **member_email** column.

Note: The **insert_team_c_records_using_async_await_transaction.js**. JavaScript file will cause a database engine rejection error because the **data** variable has "bob_queens@email.com" information which clashes with another email information inside an existing team member record that is associated to TEAM D.

```
    at processTicksAndRejections (node:internal/process/task_queues:96:5) {
  code: 'ER_DUP_ENTRY',
  errno: 1062,
  sqlMessage: "Duplicate entry 'bob_queens@email.com' for key 'team_member.member_email'",
  sqlState: '23000',
  index: 0,
  sql: 'INSERT INTO team_member \n' +
    '        (first_name,last_name,member_email,is_leader,team_id) \n' +
    "        VALUES ('BOB','QUEENS','bob_queens@email.com',0,5)"
}
```

8. Setup the directories and files inside the request_response NodeJS project before providing the code to turn the project into a web application project which has backend REST API features.

1. Watch https://youtu.be/Qo0xXk0xcDA and prepare directory and files. Acquire some concepts which has led to this directory and file structure.

1.1 Create **src** directory. Inside **src** directory, create **services** directory, **controllers** directory, **routes** directory and **utils** directory.

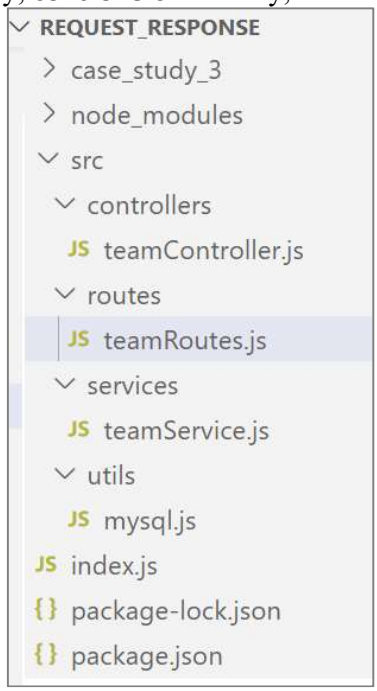
1.2 Create **index.js** at the **request_response** directory.

1.3 Create **teamService.js** inside **\src\services** directory.

1.4 Create **teamController.js** inside the **\src\controllers** directory.

1.5 Create **teamRoutes.js** inside the **\src\routes** directory.

1.4 Create **mysql.js** inside **\src\utils** directory.



2. Watch https://youtu.be/6XDmmJri9L8 and install the following libraries:

npm install express
npm install express-form-data
npm install os
npm install mysql
npm install cloudinary
npm install ejs

9. Begin developing the team data service module, team controller module, team routes module and index.js program entry file.

1 https://youtu.be/hY6EZdIIOmk

2 Refer to https://youtu.be/Qo0xXk0xcDA to provide **bare minimum** code for the **index.js** file.

3 Refer to https://youtu.be/Qo0xXk0xcDA?t=112 and provide code for the **mysql.js** file (You are **reusing** the tested code which is residing inside **case_study_3\utils\mysql.js**).

4 Watch the video **from beginning to end** https://youtu.be/z7FUO0qIW3A before you apply the code inside the **teamService.js** file. The **teamService.js** file has the method, **createTeamAndTeamMemberData** function.

5 Watch this video https://youtu.be/MY3NbXTDqkU **from beginning to end** which covers the following steps 6, 7 and 8:. You need the "big picture" before working on the next 3 files.

6 Provide code for the **teamController.js** file.

7 Provide code for the **teamRoutes.js** file.

8 Final touch up on the **index.js** file.

The code listing below shows the **index.js** file's bare minimum code. You will put more code inside the **index.js** file sometime later in the exercise. The main focus is you must **first** provide the database operation logic commands inside the **teamService.js** file.

index.js

```
const express = require('express');

const app = express();




const server = app.listen('4000', '127.0.0.1', function() {
   let host = server.address().address;
   let port = server.address().port;
   console.log('Web application listening at http://%s:%s', host, port)
})
```

Apply **node index.js** to be sure that, your bare minimum functional NodeJS express web application can start listening to a port 4000.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS D:\ST0505_ESDE\request_response> node index.js
Web application listening at http://127.0.0.1:4000
```

The video **https://youtu.be/z7FUO0qIW3A** covers the **teamService.js** file development activity, **module.exports** and **require**.

teamService.js

```
const mysql = require('../utils/mysql');
module.exports.createTeamAndTeamMemberData = async function(data) {
    const connection = await mysql.connection();
    try {
      await connection.query('START TRANSACTION');
      const createTeamResults = await connection.query(`INSERT INTO team (team_name,
            team_description) VALUES (? , ?)`, [data.teamName, data.teamDescription]);
      await connection.query(`INSERT INTO team_member
           (first_name,last_name,member_email,is_leader,team_id)
            VALUES (?,?,?,?,?)`, [data.firstName[0], data.lastName[0], data.email[0],
        (data.isLeader[0] == 'true') ? 1 : 0, createTeamResults.insertId
      ]);
      await connection.query(`INSERT INTO team_member
           (first_name,last_name,member_email,is_leader,team_id)
           VALUES (?,?,?,?,?)`, [data.firstName[1], data.lastName[1], data.email[1],
        (data.isLeader[1] == 'true') ? 1 : 0, createTeamResults.insertId
      ]);
      await connection.query(`INSERT INTO team_member
          (first_name,last_name,member_email,is_leader,team_id)
           VALUES (?,?,?,?,?)`, [data.firstName[2], data.lastName[2], data.email[2],
        (data.isLeader[2] == 'true') ? 1 : 0, createTeamResults.insertId
      ]);

      await connection.query('COMMIT');

    } catch (error) {
      if (error) {
        console.log(error);
        await connection.query('ROLLBACK');
      }
    } finally {
      await connection.release();

    } // End of try-catch-finally block
  } // End of module.exports.createTeamAndTeamMemberData
```

There following three sets of code-listing are applied in the respective **teamController.js**, **teamRoutes.js** and **index.js**. These code were used in the video session **https://youtu.be/MY3NbXTDqkU**

teamController.js

```
module.exports.createTeamAndTeamMemberData = async function(req, res, next) {
    console.log(req.headers['content-type']);
    console.log(req.body);
    const data = req.body;

    res.status(201).json({
      status: 'success',
      data: null
    });

  } // End of async function(req,res,next) for createTeamAndTeamMemberData
```

The given code here aims to inspect whether the backend system can indeed rebuild the received data from the frontend. Notice that, two **console.log(...)** calls were made to print the **request** object's **headers** and **body** property content at the terminal interface.

teamRoutes.js

```
const express = require('express');
const router = express.Router();
const teamController = require('../controllers/teamController');
const os = require("os");
const formData = require('express-form-data');

const options = {
  uploadDir: os.tmpdir(),
  autoClean: true
};
// parse data with connect-multiparty.
router.use('/teams/detail', formData.parse(options));
// delete from the request all empty files (size == 0)
router.use('/teams/detail', formData.format());
// change the file objects to fs.ReadStream
router.use('/teams/detail', formData.stream());
// union the body and the files
router.use('/teams/detail', formData.union());

router.post('/teams/detail', teamController.createTeamAndTeamMemberData);

module.exports = router;
```

The code inside the **teamController**.js file does not decide *when* the **createTeamAndTeamMemberData** method can be executed. The command at **teamRoutes.js** *will help* the backend system to decide when the **createTeamAndTeamMemberData** method can be executed. **https://youtu.be/MY3NbXTDqkU?t=567**

index.js

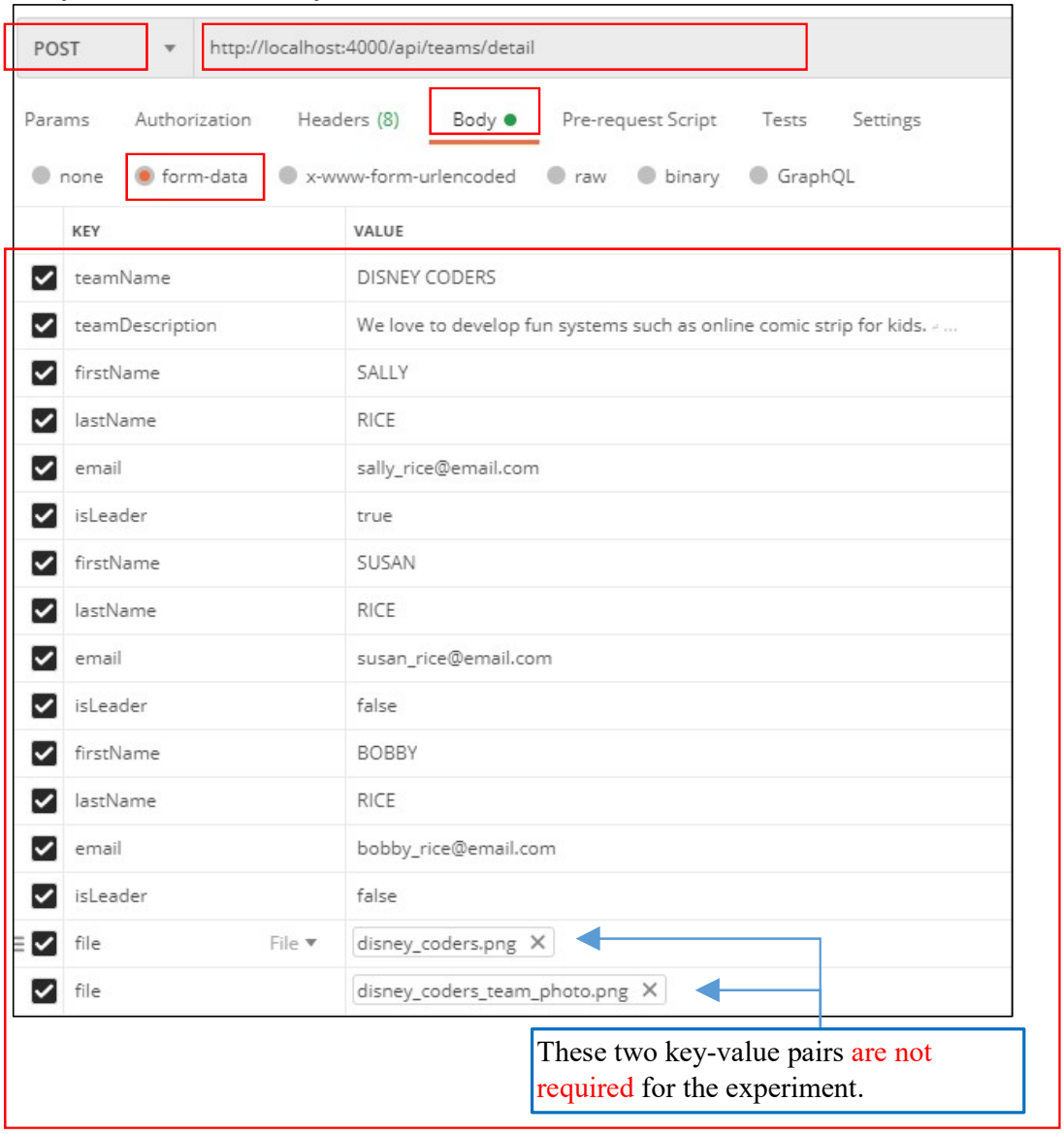
```
const express = require('express');
const teamRoutes = require('./src/routes/teamRoutes');
const app = express();

app.use('/api/', teamRoutes);

const server = app.listen('4000', '127.0.0.1', function() {
   let host = server.address().address;
   let port = server.address().port;
   console.log('Web application listening at http://%s:%s', host, port)
});
```

To appreciate the **app.use('/api/', teamRoutes)** inside the **index.js** file, please watch
https://youtu.be/MY3NbXTDqkU?t=1062

10. Test the backend system's REST API with Postman

1. Watch the video from beginning to end **https://youtu.be/vRbOoq15aAc** before testing your work.

2. Setup the Postman to make a POST method HTTP request to the REST API end point, **http://localhost:4000/api/teams/detail**



3. Test your work by referring to **https://youtu.be/vRbOoq15aAc?t=486**. Note that, the entire video shares the overall concept which you need to have before testing your work.

11. Promisifying your createTeamAndTeamMemberData function inside the teamService.js so that the code inside the teamController.js can apply try-catch block technique together with async-await technique

Watch the video from https://youtu.be/0ES3HWsFkw4 which walkthrough the process of **promisifying** **createTeamAndTeamMemberData** method inside the **teamService.js** to support the synchronous calls in the calling program at the **teamController.js** file.

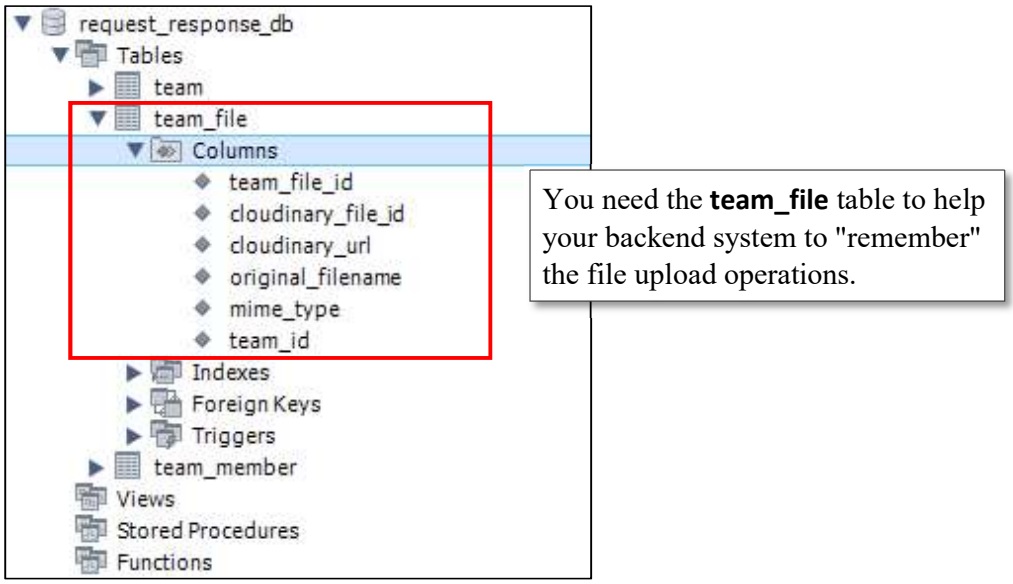
The code listing below describes the **final** code inside the **teamService.js** JavaScript file, which returns a promise which asks the calling program to **wait** for the **eventual** ①success results or ②error results.

teamService.js

```
const mysql = require('../utils/mysql');
module.exports.createTeamAndTeamMemberData = async function(data) {
  return new Promise(async function(resolve, reject) {
    const connection = await mysql.connection();
    try {
      await connection.query('START TRANSACTION');
      const createTeamResults = await connection.query(`INSERT INTO team (team_name,
        team_description)  VALUES (? , ?)`, [data.teamName, data.teamDescription]);
      await connection.query(`INSERT INTO team_member
        (first_name,last_name,member_email,is_leader,team_id)
        VALUES (?,?,?,?,?)`, [data.firstName[0], data.lastName[0], data.email[0],
        (data.isLeader[0] == 'true') ? 1 : 0, createTeamResults.insertId ]);
      await connection.query(`INSERT INTO team_member
        (first_name,last_name,member_email,is_leader,team_id)
        VALUES (?,?,?,?,?)`, [data.firstName[1], data.lastName[1], data.email[1],
        (data.isLeader[1] == 'true') ? 1 : 0, createTeamResults.insertId ]);
      await connection.query(`INSERT INTO team_member
        (first_name,last_name,member_email,is_leader,team_id)
        VALUES (?,?,?,?,?)`, [data.firstName[2], data.lastName[2], data.email[2],
         (data.isLeader[2] == 'true') ? 1 : 0, createTeamResults.insertId ]);

      await connection.query('COMMIT');
      resolve({ status: 'success', data: { teamId: createTeamResults.insertId } });
    } catch (error) {
      if (error) {
        console.log(error);
        await connection.query('ROLLBACK');
        reject({ status: 'fail', data: error });
      }
    } finally {
      await connection.release();

    } // End of try-catch-finally block
  }); //End of new Promise object definition
} // End of anonymous function async function(data)
```

12. Setup request_response project's libraries, teamRoutes.js file and recreate request_response_db database to have additional team_file table

1. Find and open the SQL script file, **create_db_tables_with_team_file_table.sql**. Execute the SQL script to obtain a fresh new database, **request_response_db** which has the following structure.
   Note that: There are no test records inside all the three tables.



2. Install the following library packages inside the **request_response** project directory.

Use **npm install streamifier**	I used this library within the **utils\cloudinary.js** to work with the cloudinary library so that the logic does not "create temporary files before sending digital content into Cloudinary online repository".
Use **npm install multer**	You use this library package inside the **teamRoutes.js** file so that your code can build a dynamic **multer** middleware function which constructs the necessary **body** and **files** property inside the Express Framework engine's internal Request object.
Use **npm install cloudinary**	I used this library inside the **utils\cloudinary.js** to provide the data stream upload functionality for this exercise. You are *indirectly* using the cloudinary library to upload content to the Cloudinary by calling the **uploadStreamToCloudinary** function.
Use **npm install cors**	Installing this library package for *possible* future usage.

3. The following commands were applied in the **teamRoutes.js** JavaScript file. You will need to make changes by going through two steps: **https://youtu.be/gy3Wq0-TrXw**

Step 1:Delete all the commands from line 4 to line 17 which are highlighted in the code listing (next page). **The reason is**, you will be using *another* library called **multer** to *rebuild* the data received from the client-side which is sent by using content type of multipart/form-data. You won't be using **express-form-data** library **anymore**, to build a Request object which represents the incoming HTTP request.

```
1.  const express = require('express');
2.  const router = express.Router();
3.  const teamController = require('../controllers/teamController');
4.  const os = require("os");
5.  const formData = require('express-form-data');
6.
7.  const options = {
8.      uploadDir: os.tmpdir(),
9.      autoClean: true
10. };
11. // parse data with connect-multiparty.
12. router.use('/teams/detail', formData.parse(options));
13. // delete from the request all empty files (size == 0)
14. router.use('/teams/detail', formData.format());
15. // change the file objects to fs.ReadStream
16. router.use('/teams/detail', formData.stream());
17. // union the body and the files
18. router.use('/teams/detail', formData.union());
19.
20. router.post('/teams/detail', teamController.createTeamAndTeamMemberData);
21.
22. module.exports = router;
```

Delete these commands which were used inside the **teamRoutes.js** JavaScript file.

The reason is, you **are not going** to use **express-form-data** library functionalities to support file data upload logic.

Step 2: Update the **teamRoutes.js** JavaScript file with the following commands:

```
1.  const express = require('express');
2.  const router = express.Router();
3.  const teamController = require('../controllers/teamController');
4.  const multer = require('multer');
5.
6.  const upload = multer({ storage: multer.memoryStorage() }).array('file', 2);
7.
8.  router.post('/teams/detail', upload, teamController.createTeamAndTeamMemberData);
9.
10. module.exports = router;
```

13. Study the teamRoutes.js code *before* providing code which supports the file data upload functionality inside the teamController.js file and teamService.js file.

To appreciate the code changes in the **teamRoutes.js**, you need to capture the following three points.

The incoming data for the POST HTTP request sent by the client-side (e.g. Postman) can be partially described by the following table. Note that, the table below has ignored many parts of the HTTP request details. The table below only focused on the important parts of the HTTP request details.

general information of client-side HTTP request	
request url	'http://localhost:4000/api/teams/detail'
request method	'POST'
request headers	
content type	'multipart/form-data'
request body	
teamName	'Team D'
teamDescription	Team D description
firstName	'BRIZ'
lastName	'QUEENS'
email	'briz_queens@email.com'
isLeader	'true'
firstName	'BRADDY'
lastName	'QUEENS'
email	'braddy_queens@email.com'
isLeader	'false'
firstName	'BOB'
lastName	'QUEENS'
email	'bob_queens@email.com'
isLeader	'false'
file	file data content of **team_d_logo.png**
file	file data content of **team_d_group_photo.png**

Before your backend logic (the code which you have placed inside the teamRoutes.js, teamController.js, teamService.js etc.) can handle the tabulated information which describes the HTTP request details, your backend Express framework engine will *internally* build a Request object. (Request object official notes: **https://expressjs.com/en/api.html#req**).

The figure below describes the internally built Request object which is created by the Express framework engine *before* the **teamRoutes.js** file's code is executed. Initially, the Request object has empty **body** property and empty **files** property. When the JavaScript engine processes the code inside the **teamRoutes.js** file, the Express engine executes the middleware function, **upload**. The middleware function, **upload** was dynamically created from the loaded **multer** library. Finally, the Request object within the Express engine will be populated with **body** property and **files** property data which can be easily accessible by your code inside the **createTeamAndTeamMemberData** function.

The Express Framework Engine's internal Request object does not have **body** property and **file** property

BEFORE

body	
Has no properties	Has no associated values
files	Has no file content and file description

```
const multer = require('multer')
const upload = multer({ storage: multer.memoryStorage() }).array('file', 2);
router.post('/teams/detail', upload, teamController.createTeamAndTeamMemberData);
```

The internal Request object has **body** and **files** property associated with data obtained from the incoming HTTP request details

AFTER

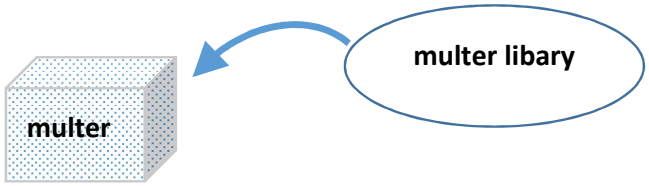
body			
teamName	'Team D'		
teamDescription	'Team D description'		
firstName	'BRIZ'	'BRADDY'	'BOB'
lastName	'QUEENS'	'QUEENS'	'QUEENS'
email	'briz_queens@email.com'	'braddy_queens@email.com'	'bob_queens@email.com'
isLeader	'true'	'false'	'false'

files

originalname	**mimetype**	**buffer**
'team_d_logo.png'	'image/png'	<Buffer 89.... 6e 20 ... 4555 more bytes>

originalname	**mimetype**	**buffer**
'team_d_group_photo.png'	'image/png'	<Buffer 89.... 6e 20 ... 118113 more bytes>

3 const multer = require('multer') command tells the JavaScript engine to load the **multer** library functionality into the variable **multer**.



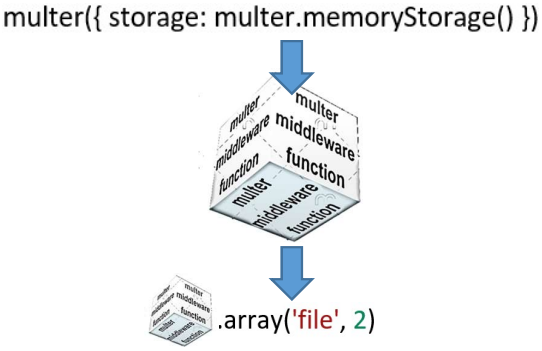
The entire expression at the right hand side of the assignment operator aims to *dynamically* create a middleware function, which has the capability of:

① Capturing incoming file information inside the system's memory (instead of saving to a file).
② Check-detect for any **file** property within the incoming HTTP request to extract file content information.
③ Limits the number of **file** content to two.

```
const upload = multer({ storage: multer.memoryStorage() }).array('file', 2);
```

The command above was pieced together through careful reading on the technical guide at https://www.npmjs.com/package/multer.

The JavaScript engine processes the expression `multer({ storage: multer.memoryStorage() })` first. This expression tells the JavaScript engine to:
①Dynamically creates an anonymous JavaScript *middleware* function, by calling the **multer(**  **)**.
②Pass the object of information  which has one property, **storage.** The **storage** property is associated with an empty MemoryStorage object created by the expression multer.memoryStorage().



Then, refer to the figure above, the JavaScript engine is calling the **array** method which belongs to the built middleware function.
After building the middleware function, the internal logic of the respective middleware function has an **array** method (The **array** method was seen inside the technical documentation guide for this **multer** library).
Before the middleware function is assigned to the **upload** variable, the **array** method was called to *educate* the middleware function to ①expect **file** property name inside the incoming HTTP request and ②do not accept more than 2 file content.

Finally, the middleware function is assigned to the **upload** variable. Therefore, you can see (treat) this **upload** variable as a middleware function.

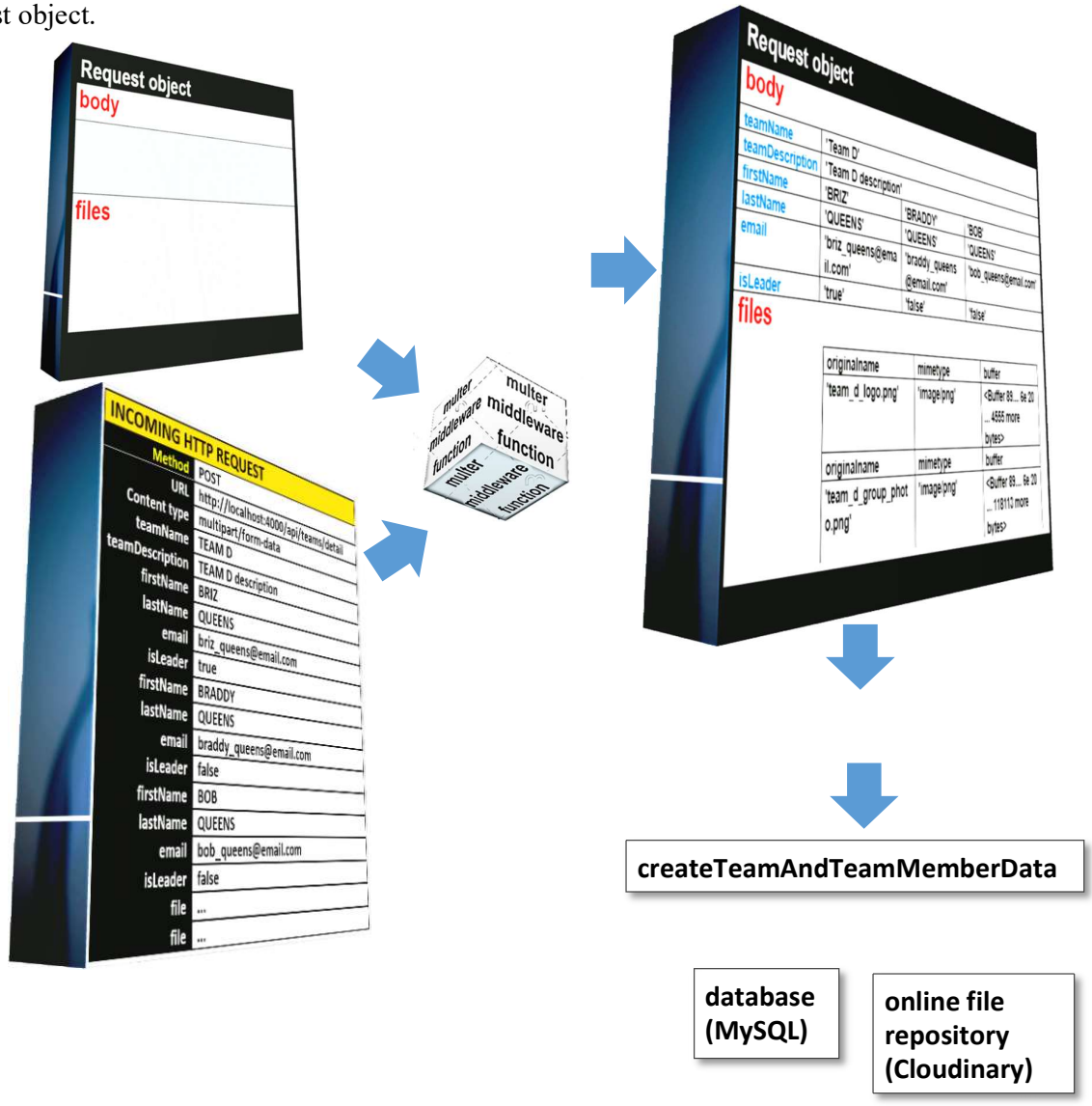
The **upload** middleware function which has been configured to "assemble" the **body** and **files** property content within the Express framework engine's Request object is not used yet.
Refer to the figure below. The figure describes the command below:

router.post('/teams/detail', **upload**, teamController.createTeamAndTeamMemberData);

The command tells the Express framework engine to:

① Process the **upload** middleware function first which builds the Request object to have **body** and **files** property content. (See figure below)

② Then, forward the Request object (which has data associated to the **body** and **files** property) to the **createTeamAndTeamMemberData** method to handle the data. The **createTeamAndTeamMemberData** function has the **req** parameter variable to reference the Request object.



14, Prepare request_response\src\utils\cloudiary.js utility JS file

The video reference **https://youtu.be/TcmPSMqLB9s** covers the following steps.

1. Create a new JS file, **cloudinary.js** at the **src\utils** directory.
2. Copy the following code listing (this page and next page) into the new JS file, **cloudinary.js**. The code inside the **teamService.js** will be *require* the functionality of this **cloudinary.js** file to communicate with the Cloudinary online file repository system and complete the file data upload operations.

cloudinary.js

```
const cloudinary = require('cloudinary').v2;
const streamifier = require('streamifier');
cloudinary.config({
   cloud_name: 'dit88xxx',
   api_key: '3848973511',
   api_secret: 'p6NEbaaaO7TD4Y4Q'});

//uploadStreamToCloudinary - useful if the backend uses the:
// ->direct stream the content to cloudinary without temp file technique
//strategy
module.exports.uploadStreamToCloudinary = function(buffer) {
     return new Promise(function(resolve, reject) {
       //Create a powerful, writable stream object which works with Cloudinary
          let streamDestination = cloudinary.uploader.upload_stream({ folder: 'teams',
          allowed_formats: 'png,jpg',
          resource_type: 'image' },
          function(error, result) {
            if (result) {
              //Inspect whether I can obtain the file storage id and the url from cloudinary
              //after a successful upload.
              //console.log({imageURL: result.url, publicId: result.public_id});
              let cloudinaryFileData = { url: result.url, publicId: result.public_id, status: 'success' };
              resolve({ status: 'success', data: cloudinaryFileData });
            }
            if (error) {
              reject({ status: 'fail', data: error });
            } // End of if..else block inside the anonymous function given to upload_stream
          });
       streamifier.createReadStream(buffer).pipe(streamDestination);
     }); //End of Promise
  } //End of uploadStreamToCloudinary
```

https://youtu.be/TcmPSMqLB9s?t=95

```
  //uploadFileToCloudinary - useful if the backend uses the:
  // ->create temp file first->then use temp file to upload
  //strategy
module.exports.uploadFileToCloudinary = function(file) {
    return new Promise((resolve, reject) => {
      //The following code will upload file to cloudinary
      cloudinary.uploader.upload(file.path, {
            folder: 'teams',
            allowed_formats: 'png,jpg',
            resource_type: 'image'
          })
          .then((result) => {
            //Inspect whether I can obtain the file storage id and the url from cloudinary
            //after a successful upload.
            //console.log({imageURL: result.url, publicId: result.public_id});
            let data = { url: result.url, publicId: result.public_id, status: 'success' };
            resolve({ status: 'success', data: data });
          }).catch((error) => {
            reject({ status: 'fail', data: error });
          }); //End of try..catch
    }); //End of Promise
  } //End of uploadFileToCloudinary
```

This function **will not be used** at all inside your NodeJS system.
The function remains inside the **cloudinary.js** to create awareness that there are *two* strategies for file upload operations.

15, Begin editing teamService.js file and teamController.js file to support file data upload and database operations on team data

Video reference: https://youtu.be/_wRmintSfU0

1. After preparing the **cloudinary.js** file, refer to the code listing below. Make changes to the commands *inside* the **createTeamAndTeamMemberData** function which is defined at the **teamService.js** JavaScript file.

teamService.js

```
const mysql = require('../utils/mysql');
const cloudinary = require('../utils/cloudinary');

module.exports.createTeamAndTeamMemberData = async function(req) {
    //Usually database operation functions do not take in the whole request object
    //as an input. For this createTeamAndTeamMemberData function, we don't have a choice
    //because the multer library internal logic creates body property to hold the text data
    //key-value pair portion and creates files property to hold the file data portion.
    data = req.body;

    return new Promise(async function(resolve, reject) {
      const connection = await mysql.connection();
      try {
        await connection.query('START TRANSACTION');
        const createTeamResults = await connection.query(`INSERT INTO team (team_name,team_descrip
tion)
    VALUES (? , ?)`, [data.teamName, data.teamDescription]);
        console.log(createTeamResults.insertId);
        await connection.query(`INSERT INTO team_member
    (first_name,last_name,member_email,is_leader,team_id)
    VALUES (?,?,?,?,?)`, [data.firstName[0], data.lastName[0], data.email[0],
          (data.isLeader[0] == 'true') ? 1 : 0, createTeamResults.insertId
        ]);
        await connection.query(`INSERT INTO team_member
    (first_name,last_name,member_email,is_leader,team_id)
    VALUES (?,?,?,?,?)`, [data.firstName[1], data.lastName[1], data.email[1],
          (data.isLeader[1] == 'true') ? 1 : 0, createTeamResults.insertId
        ]);
```

Create a **cloudinary** module object, which has functionalities to complete a file data upload operation to the Cloudinary.

The **req** input parameter variable references the Express framework engine's Request object. The **data** variable needs the **req.body** to reference the team and team member data.

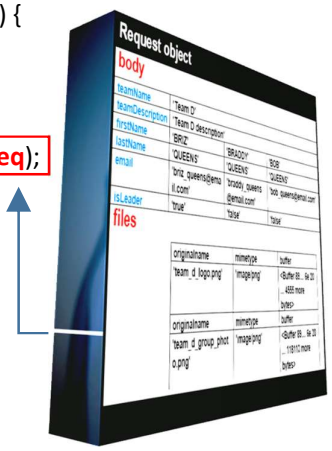
```
        await connection.query(`INSERT INTO team_member
    (first_name,last_name,member_email,is_leader,team_id)
    VALUES (?,?,?,?,?)`, [data.firstName[2], data.lastName[2], data.email[2],
          (data.isLeader[2] == 'true') ? 1 : 0, createTeamResults.insertId
        ]);
  let fileUploadResult = null;
  for (let fileIndex = 0; fileIndex < req.files.length; fileIndex++) {
  //console.log(req.files[fileIndex]);
  //upload file first
  fileUploadResult = await cloudinary.uploadStreamToCloudinary(req.files[fileIndex].buffer);
  await connection.query(`INSERT INTO team_file (cloudinary_file_id,cloudinary_url,
       original_filename, mime_type,
        team_id) VALUES (?,?,?,?,?)`,
       [fileUploadResult.data.publicId, fileUploadResult.data.url,
        req.files[fileIndex].originalname,
        req.files[fileIndex].mimetype, createTeamResults.insertId
       ]);
  }//End of for loop logic to stream 2 files to cloudinary file respository
        await connection.query('COMMIT');
        resolve({ status: 'success', data: { teamId: createTeamResults.insertId } });
      } catch (error) {
        if (error) {
          console.log(error);
          await connection.query('ROLLBACK');
          reject({ status: 'fail', data: error });
        }
      } finally {
        await connection.release();

      } // End of try-catch-finally block
    }); // End of Promise object definition
} // End of createTeamAndTeamMemberData function
// (to be exported out for calling program to use)
```

A for loop logic has been created, to call the **uploadStreamToCloudinary** method **twice** to complete two file upload operations. For each successful upload operation, a record is created inside the **team_file** table.

2. After making changes to the **teamService.js** file, refer to the code listing below. Make **minor** changes to the **teamController.js** file so that the logic can pass the *entire* Express Framework engine's Request object content into the **createTeamAndTeamMemberData** method to perform database operations and file upload operations.

teamController.js

```
const teamDataManager = require('../services/teamService');

module.exports.createTeamAndTeamMemberData = async function(req, res, next) {
    //console.log(req.body);
    //console.log(req.files);
    try {
     const results = await teamDataManager.createTeamAndTeamMemberData(req);
     console.log(results.data.teamId);
     res.status(201).json({
      status: 'success',
      data: null
     });
    } catch (error) {
       console.error(error)
       res.status(500).json({
          status: 'fail',
          data: null
       });

    }
  } // End of async function(req,res,next)
```



The **cloudinary.js** JavaScript file's code uses the JavaScript programming fundamentals such as pipe**,** stream object**,** method chaining and Buffer object etc**.** I did not cover the pipe, stream object and Buffer object concepts in this section of the worksheet practical exercise.

I have prepared the code for you to create the JavaScript file, **cloudinary.js.** The function, **uploadStreamToCloudinary** *directly* used the cloudinary library functionalities to upload one complete file data to cloudinary**.**

The code which you have written inside the **teamService.js** can *indirectly* use the **cloudinary** library's functionalities by calling the **uploadStreamToCloudinary** function using the **async** and **await** technique.

I have experimented, prepared and promisified the necessary code inside the **cloudinary.js** file, so that you can apply command such as `const cloudinary = require('../utils/cloudinary');` at the **teamService.js** file to:

① Load the functionality and build an module object, **cloudinary** which can communicate with the online file repository and complete one file upload operation.
② Apply **async await** technique to ensure the file upload operations can be in sequence. For example, upload and write the file data into the Cloudinary first, before creating one record inside the **team_file** table.

Learning advice: After this entire workbook exercise, please visit online blogs and videos to strengthen the concept of the **pipe** function, stream and **Buffer** object concepts, so that you can comprehend and reuse my code inside the **cloudinary.js** file with understanding. You can even appreciate:

① The code shared by other developers in the online tutorials, blogs and github repository which focus on the file upload and other file operation functionalities.

② The sample code examples and tutorials given at **npmjs.com** website which has library packages such as **busboy**, **multer**, **formidable**, **multiparty** etc.

The case study in this practice worksheet does not cover the above mentioned fundamentals. The following are some YouTube videos that I have curated which you must watch and practice the code at the same time to strengthen the fundamental concepts.

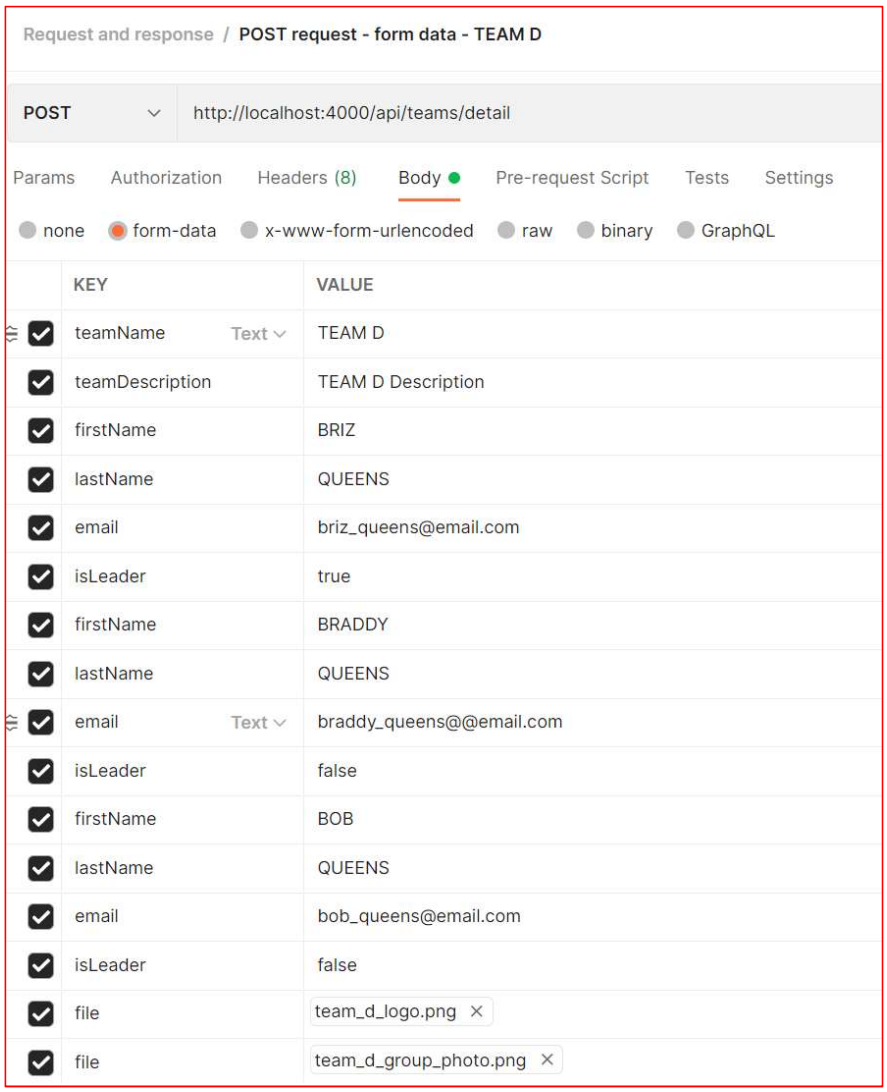
Video reference on the topic "Necessary fundamental file concepts before building any file upload logic in a backend system. **https://youtu.be/TazKKGA1-GM**

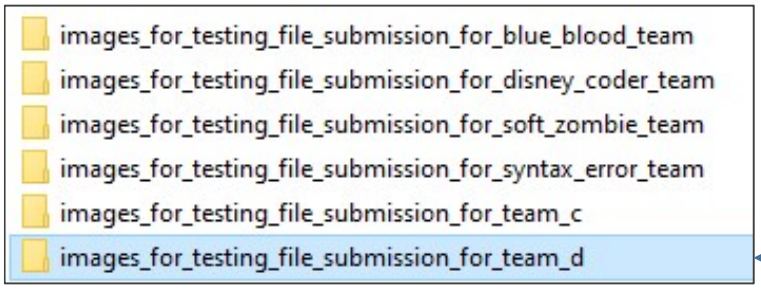
https://youtu.be/a8W90jDHSho by .NET Ninja covers the usefulness of **pipe** function. The author has involved the concept of stream object and buffer object in the video to produce file IO (input, output) operation what involves the **pipe** function.

https://youtu.be/2gEv05VmEC8 covers file upload which uses the formidable middleware instead of using the multer library.

16. Test the NodeJS project on file upload operation and create team data operation

1. Refer to the figure below. Setup a POST HTTP request at Postman.





You can find the required PNG files inside the directory, **images_for_testing_file_submission_for_team_d** folder.

2. Within the terminal interface, **node index.js**. Ensure that the console is focusing on **request_response** folder.

3. Send the POST HTTP request.

4. Check the Postman's response section. You should expect the following results after the NodeJS backend system has completed the database operations and file upload operations.

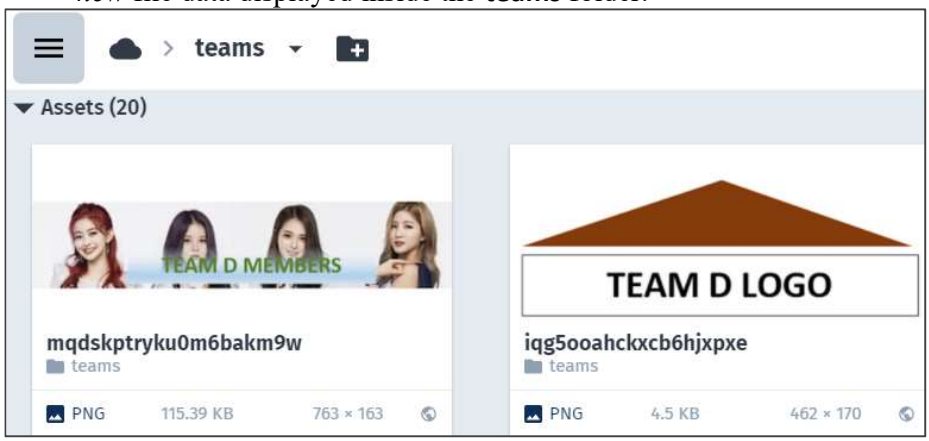
5. Refer to the figure below. The **teamService.js** file's logic should create 1 team record (**team** table), 3 team member records (**team_member** table) and 2 file description records (**team_file** table)

team_id	team_name	team_description
1	TEAM D	TEAM D Description

team_member_id	first_name	last_name	member_email	is_leader	team_id
1	BRIZ	QUEENS	briz_queens@email.com	1	1
2	BRADDY	QUEENS	braddy_queens@@email.com	0	1
3	BOB	QUEENS	bob_queens@email.com	0	1

team_file_id	cloudinary_file_id	cloudinary_url	original_filename	mime_type	team_id
1	teams/ftaqxssebgvyp0f9i56k	http://res.cloudinary.com/dit88888/image/uploa...	team_d_logo.png	image/png	1
2	teams/jp90wtq3pdlkv2u3w4dx	http://res.cloudinary.com/dit88888/image/uploa...	team_d_group_photo.png	image/png	1

6. Check the Cloudinary dashboard interface. Access the **teams** folder. You should expect two *new* file data displayed inside the **teams** folder.



16. Lesson on how the code works inside the teamRoutes.js teamController.js and teamService.js file to support database operations and file upload functionality.

The previous section exercise focuses on letting you "see" the *overall* results as soon as possible, while to let you feel the:

① Changes of the code inside the **teamRoutes.js**, **teamController.js** and **teamService.js** JavaScript file.
② Changes of the database design (**team_file** table).
③ Changes on the additional libraries (e.g. multer, streamifier)

You have provided a lot of JavaScript code in the previous few sessions. This section focuses on code explanation by using the Visual Studio Debug tool. The explanation covers how the additional commands (which you have provided), have worked together inside the NodeJS backend system to:

① Handle the multipart/form-data content type POST HTTP request which has two file data.
② Create records in the three tables.
③ Create two file data inside the Cloudinary's **teams** folder.

Compulsory video lesson: **https://youtu.be/9BMl8E8ggk0**

https://youtu.be/9BMl8E8ggk0	The starting part of the video shares how breakpoints are set in the **teamController.js** and **teamService.js** JavaScript file *before* launching the project in debug mode.
https://youtu.be/9BMl8E8ggk0?t=87	Setting up breakpoints at teamController.js
https://youtu.be/9BMl8E8ggk0?t=163	Setting up breakpoints at teamService.js
https://youtu.be/9BMl8E8ggk0?t=273	Prepare to use the **Debug Console** instead of using the Terminal Console. Choose **Run and Debug** function Familiarize with the **debug interface** **Launch** the project in debug mode Appreciate the existence of a horizontal **floating debug toolbar**
https://youtu.be/9BMl8E8ggk0?t=371	Prepare the database
https://youtu.be/9BMl8E8ggk0?t=421	Prepare the POST HTTP request at Postman
https://youtu.be/9BMl8E8ggk0?t=513	How to setup a file type key-value pair information
https://youtu.be/9BMl8E8ggk0?t=630	Observe how Postman **behaves** when sending a POST HTTP request to a system which is running in **debug mode**.
https://youtu.be/9BMl8E8ggk0?t=653	Using Visual Studio Debug tool, to appreciate the fundamental concept on the Express Framework engine's internal Request object.

https://youtu.be/9BMl8E8ggk0?t=773	Analyzing the internal Request object's **body** property and **files** property content structure by using the **Debug tool's Watch section**.
https://youtu.be/9BMl8E8ggk0?t=1236	Analyzing how the commands work together inside **teamRoutes.js** JavaScript file to prepare the Request object's **body** and **files** property.
https://youtu.be/9BMl8E8ggk0?t=1611	Last attempt to help you appreciate the routing logic inside the **teamRoutes.js** file.
https://youtu.be/9BMl8E8ggk0?t=2071	Observe the Debug tool's **Step Into** function (**F11**). Explanation on how the code works inside the **teamService.js** JavaScript file
https://youtu.be/9BMl8E8ggk0?t=2558	Explanation on the command which calls the **uploadStreamToCloudinary** method.
https://youtu.be/9BMl8E8ggk0?t=2711	Observe the Debug tool's **Step Over** function (F10) Appreciate the command which dynamically build the SQL which tells the database to create a new **team_file** record
https://youtu.be/9BMl8E8ggk0?t=3315	Using F10 or Step Over function to appreciate how the **fileIndex** variable is used to reference the next **buffer** property which belongs to the 2nd element of the **files** array property.
https://youtu.be/9BMl8E8ggk0?t=3742	**Using F5** *twice* to get the JavaScript engine finish processing all the commands to send a response back to the client (Postman)

https://youtu.be/5zz3Hv1J9tU	How to **stop** the project which is running in Debug mode.

17. Code your own cloudinary.js (Part 1) - What is Buffer object, Readable stream, Writable stream and pipe method

The **teamService.js** file has a function **createTeamAndTeamMemberData**. You have written a command:

```
cloudinary.uploadStreamToCloudinary(...);
```

inside the body of the **createTeamAndTeamMemberData** function.

Note that, before you can call the uploadStreamToCloudinary method, you need to create a module instance first. The module instance, **cloudinary** was created with the command:
cloudinary. (const cloudinary = require('..\utils\cloudinary'));
The command calls the function, **uploadStreamToCloudinary** inside the **\utils\cloudinary.js** through the module object (instance).

The previous practical exercise has demonstrated the file upload operation with the help of a JavaScript file, **\utils\cloudinary.js**. The previous practical exercise did not emphasize on "How to write the file data upload code which defines the **uploadStreamToCloudinary** function inside the **cloudinary.js** file". The previous practical exercise focuses on helping you to familiarize with the overall commands inside the **teamService.js**'s **createTeamAndTeamMemberData** function (method). As a result, the **\utils\cloudinary.js** file was prepared for you to complete overall logic of the **createTeamAndTeamMemberData** method.

The code inside the JavaScript file, **\utils\cloudinary.js**, are **learning code**.

They are not sufficient at all, to address real problems and objectives in real industry project development such as final year project.
Final year projects which involve real customers *do not share the same* file management needs. The use case is *often different*. As a result, you need to fully understand how the commands were written in the **\utils\cloudinary.js**.

In this practical session, you will starting writing your own **uploadStreamToCloudinary** function while learning fundamental concepts along the way. With those concepts, you are one step closer to have the ability in improving the file upload related code that addresses future problems.
It takes more than ten times of practice, to understand, improve and reuse my JavaScript commands for other **real** use case scenarios in the final year project, such as:

- Video upload
- Video streaming
- Document upload (pdf, word, excel etc.)
- Document download
- Document viewing

In this practical session, you will develop your own **experiment_cloudinary.js** utility file which as a function, **uploadStreamToCloudinary**. You will make changes to the **teamService.js** file's code to call the **uploadStreamToCloudinary** function which will be defined in your own **experiment _cloudinary.js** utility file. The **teamService.js**'s **createTeamAndTeamMemberData** function will not use the **cloudinary.js** file's code.

You can refer to the view list (next page) for the video **https://youtu.be/WJYtKi429Rk** which tries to break down the learning into several stages.

- The video guidance has been broken down into part 1 and part 2.
- The video session is lengthy because:
  ① Important concepts on Buffer object, Readable stream object, Writable stream object, file system IO are necessary.
  ② The programming activities involved "*testing a command first*" before "*using the command as part of the building blocks for the overall data upload logic*".

experiment_cloudinary.js **version 1**

Follow the video guide slowly to prepare the code here **line by line (step by step)**.
Avoid copy and paste all the commands because you need to personally practice and fully understand these commands yourself.

```
const cloudinary = require('cloudinary').v2;
const streamifier = require('streamifier');
const fs = require('fs');
cloudinary.config({
   cloud_name: 'ditaaxx',
   api_key: '384555583511',
   api_secret: 'p6NEbxxxxxO7TDyyyyQ',
   secure: true
});
const buffer = fs.readFileSync('team_d_logo.png');
console.log(buffer instanceof Buffer); // true
console.log(buffer); // '<Buffer 7b 0a 20 20 22 6e 61 6d 65 22 ...>'
//Experiment 1
const streamDestination = fs.createWriteStream('a.png');
const streamDataSource = fs.createReadStream('team_d_logo.png');
streamDataSource.pipe(streamDestination);
//Experiment 2 - Use the Buffer object, buffer as the source of the data
const streamDestination2 = fs.createWriteStream('b.png');
const streamDataSource2 = streamifier.createReadStream(buffer);
streamDataSource2.pipe(streamDestination2);
```

https://youtu.be/WJYtKi429Rk?t=83
Create **case_study_file_upload** folder inside the **request_response** folder. Prepare a new JS file, **experiment_cloudinary.js** Prepare a PNG file, **team_d_logo.png** inside the **case_study_file_upload** folder. Additional: Prepare a BMP file, **yellow_box.bmp** inside the **case_study_file_upload** folder
https://youtu.be/WJYtKi429Rk?t=120
Install the cloudinary and streamifier library packages.
https://youtu.be/WJYtKi429Rk?t=154
Provide commands inside the **experiment_cloudinary.js** file to *require* the necessary library packages.
https://youtu.be/WJYtKi429Rk?t=342
Preview on the desired outcome at the end of this exercise. This section demonstrates the results of the code which you will be experimenting in the subsequent sections of the video.
https://youtu.be/WJYtKi429Rk?t=483
Configuring the **cloudinary** object
https://youtu.be/WJYtKi429Rk?t=701
Introduction to Buffer object
https://youtu.be/WJYtKi429Rk?t=824
Appreciate the following command which prepares one Buffer object, **buffer** which contains **team_d_logo.png** digital data. const buffer = fs.readFileSync('team_d_logo.png');
https://youtu.be/WJYtKi429Rk?t=1225
Using two experiments to familiarize with Buffer object, **pipe** method, Readable stream and Writable stream object concepts.

experiment_cloudinary.js **version 2**

Follow the video guide slowly. Prepare the code here **step by step**.
Avoid copy and paste all the commands because you need to personally practice and fully understand these commands yourself.

```
const cloudinary = require('cloudinary').v2;
const streamifier = require('streamifier');
const fs = require('fs');
cloudinary.config({
   cloud_name: 'dit88xxx',
   api_key: '386663783511',
   api_secret: 'p6NEaa1122dZO7TD4Y4Q',
   secure: true
});

const buffer = fs.readFileSync('team_d_logo.png');
//const buffer = fs.readFileSync('yellow_box.bmp');
        let streamDestination = cloudinary.uploader.upload_stream({ folder: 'teams',
           allowed_formats: 'png,jpg', resource_type: 'image' },
           function(error, result) {
             if (result != null) {
                console.log(result);
```

```
        const cloudinaryFileData = { publicId: result.public_id, url: result.url };
        resolve({ status: 'success', data: cloudinaryFileData });
      } else {
        console.log(error);
        reject({ status: 'fail', data: error });
      }
    }); //End of cloudinary.uploader.upload_stream

streamifier.createReadStream(buffer).pipe(streamDestination);
```

https://youtu.be/WJYtKi429Rk?t=2025
Streaming data to Cloudinary A PowerPoint slide lecture which carefully explains how to use the **upload_stream** method which requires **two inputs**. The method is used to create a powerful-flexible Writable stream object which targets the Cloudinary. You create a **streamDestination** variable is used to reference that object.
https://youtu.be/jdHxK8ptGFw
Streaming data to Cloudinary An overview on the programming activity objectives which focuses on streaming data to Cloudinary
https://youtu.be/jdHxK8ptGFw?t=186
Streaming data to Cloudinary Breaking down the coding activity into *stages*.
https://youtu.be/jdHxK8ptGFw?t=490
Streaming data to Cloudinary Introduction on Cloudinary library's **upload_stream** method
https://youtu.be/jdHxK8ptGFw?t=597
Streaming data to Cloudinary Using the **upload_stream** method to create the **streamDestination** JavaScript object which does *two-way* communication with the Cloudinary system.
https://youtu.be/jdHxK8ptGFw?t=1045
Streaming data to Cloudinary Explanation on the callback function which is provided as an *input* when calling the **upload_stream** method.
https://youtu.be/jdHxK8ptGFw?t=1184
Streaming data to Cloudinary Create a Readable stream object and call the **pipe** method at the same time to begin streaming data file content to Cloudinary.
https://youtu.be/jdHxK8ptGFw?t=1324
Streaming data to Cloudinary ▪**Success based testing** on the code inside the **experiment_cloudinary.js** file by performing a successful upload operation. ▪Explanation on how the code works ▪Explanation on the important property-value pairs such as **public_id** and **url** properties inside the **result** object.

https://youtu.be/jdHxK8ptGFw?t=1601
Streaming data to Cloudinary ▪**Fault/Error based testing** on the code inside the **experiment_cloudinary.js** file, by performing a file upload operation which will *fail*. ▪Explanation on the important property-value pairs such as **message** property inside the **error** object.
https://youtu.be/jdHxK8ptGFw?t=1732
Streaming data to Cloudinary **Conclusion** for Part 1 which focuses on experimenting commands to **piece together** the Buffer object, **pipe** method, Readable stream and Writable stream object concepts.

18. Code your own cloudinary.js (Part 2) - Build a reusable uploadStreamToCloudinary function which can be used by the developers inside any NodeJS project which needs file data upload functionality.

experiment_cloudinary.js **version 3**

Follow the video guide slowly. Prepare the code here **step by step**.
Avoid copy and paste all the commands because you need to personally practice and fully understand these commands yourself.

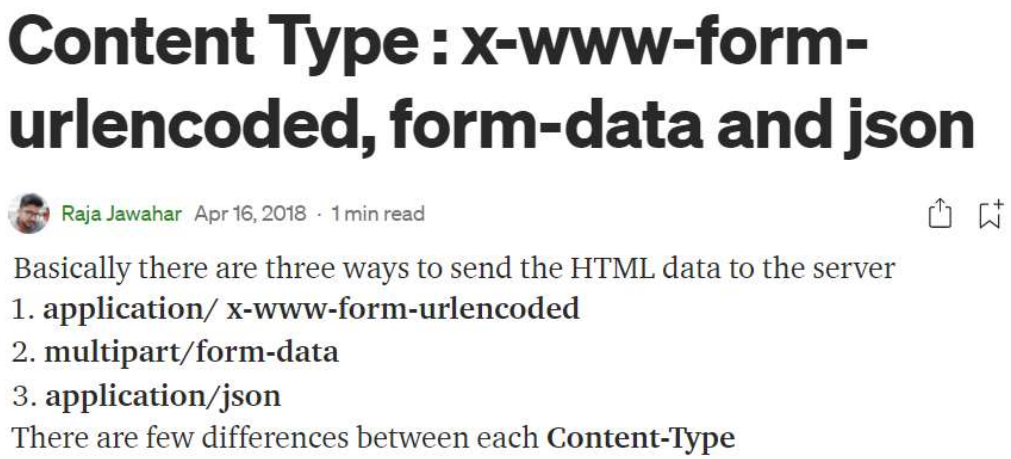
```
const cloudinary = require('cloudinary').v2;
const streamifier = require('streamifier');

cloudinary.config({
  cloud_name: 'dit88888',
  api_key: '384897343783511',
  api_secret: 'p6NEbofdJkdrf5g14dZO7TD4Y4Q',
  secure: true
});

module.exports.uploadStreamToCloudinary = function (buffer) {
  return new Promise(
    function (resolve, reject) {
      let streamDestination = cloudinary.uploader.upload_stream({ folder: 'teams', allowed_form
ats: 'png,jpg', resource_type: 'image' },
        function (error, result) {
          if (result != null) {
            console.log(result);
            const cloudinaryFileData = { publicId: result.public_id, url: result.url };
            resolve({ status: 'success', data: cloudinaryFileData });
          } else {
            console.log(error);
            reject({ status: 'fail', data: error });
          }
        });

      streamifier.createReadStream(buffer).pipe(streamDestination);
    }
  )
}

module.exports.uploadStreamToCloudinary(buffer);
```

Build your own uploadStreamToCloudinary inside experiment_cloudinary.js. Your NodeJS project will no longer use my cloudinary.js.
https://youtu.be/jdHxK8ptGFw?t=1782
Recap on all the commands inside the **experiment_cloudinary.js**, before transforming the code into a reusable function, **uploadStreamToCloudinary**.
https://youtu.be/jdHxK8ptGFw?t=1919
Begin defining the function, **uploadStreamToCloudinary** and the function's body.
https://youtu.be/jdHxK8ptGFw?t=1993
Begin defining the Promise object which is returned by the **uploadStreamToCloudinary** function logic. Carefully prepare the anonymous function code within the Promise object.

Apply the resolve and reject.
https://youtu.be/jdHxK8ptGFw?t=2760
Summarizing all the coding activity before proceeding towards integrating the function into a NodeJS project.
https://youtu.be/jdHxK8ptGFw?t=2865
Lecture session to recap the code inside the **teamService.js** JavaScript file.
https://youtu.be/jdHxK8ptGFw?t=3239
Begin integrating your own **uploadStreamToCloudinary** function into the NodeJS project. Make changes to the **teamService.js** code to use your **uploadStreamToCloudinary**. Your NodeJS project stops using my **cloudinary.js** and uses your **experiment_cloudinary.js**. **Final integration testing.**

Appendix A Three frequent ways to send HTML data to the server-side

The author has clearly shared the three frequent techniques for sending HTML data to the server side at,
https://medium.com/@rajajawahar77/content-type-x-www-form-urlencoded-form-data-and-json-e17c15926c69



Content Type : x-www-form-urlencoded, form-data and json

Raja Jawahar Apr 16, 2018 · 1 min read

Basically there are three ways to send the HTML data to the server
1. **application/ x-www-form-urlencoded**
2. **multipart/form-data**
3. **application/json**

There are few differences between each **Content-Type**

As a result, your NodeJS backend system's server-side JavaScript code need to deal with *how* the HTML data should *collected* and *rebuilt* inside the **body** property of the Express framework engine's internal Request object.

Appendix D Monitoring internal Request object's content by using a simple middleware function

A simple middleware concept was implemented in the lesson video, **https://youtu.be/bhHZMybsEuc** to monitor the changes inside the Express Framework engine's Request object.

The code listing below shows the code which was used in the video session to analyze the internal Request object.

```
const express = require('express');
const router = express.Router(); //Create a router object
const teamController = require('../controllers/teamController');
const multer = require('multer')

function inspectRequestObject (req,res,next){

 console.log(req);
 next();
}

const upload = multer({ storage: multer.memoryStorage() }).array('file', 2);

router.post('/teams/detail', [ inspectRequestObject ,upload,
inspectRequestObject ], teamController.createTeamAndTeamMemberData);

module.exports = router;
```

Middleware function was defined inside the **teamRoutes.j**s file to inspect the Request object (referenced by the **req** parameter variable)

Appendix E create_db_tables_with_team_file_table.sql SQL script file

The objective of the following script file is, to create three empty tables (additional team_file table). The script file is frequently applied in the video session and practice exercises which focuse on file upload concepts.

```
-- file name: create_db_tables_with_team_file_table.sql
DROP DATABASE IF EXISTS request_response_db;
Create Database request_response_db CHARACTER SET latin1 COLLATE latin1_general_ci;
Use request_response_db;
 CREATE TABLE team(
    team_id int NOT NULL AUTO_INCREMENT,
    team_name varchar(200) NOT NULL,
    team_description varchar(200) NOT NULL,
    PRIMARY KEY (team_id),
    UNIQUE(team_name)
)AUTO_INCREMENT=1;

CREATE TABLE team_member(
    team_member_id int NOT NULL AUTO_INCREMENT,
    first_name varchar(100) NOT NULL,
    last_name varchar(100) NOT NULL,
    member_email varchar(200) NOT NULL,
    is_leader boolean NOT NULL,
    team_id int NOT NULL,
    PRIMARY KEY (team_member_id),
    UNIQUE(member_email),
          FOREIGN KEY (team_id) REFERENCES team(team_id)
)AUTO_INCREMENT=1;

 CREATE TABLE team_file (
  team_file_id int NOT NULL AUTO_INCREMENT,
  cloudinary_file_id varchar(255) COLLATE latin1_general_ci NOT NULL,
  cloudinary_url varchar(255) COLLATE latin1_general_ci NOT NULL,
  original_filename varchar(255) COLLATE latin1_general_ci NOT NULL,
  mime_type varchar(100) COLLATE latin1_general_ci NOT NULL,
  team_id int NOT NULL,
  FOREIGN KEY (team_id) REFERENCES team(team_id) ,
  PRIMARY KEY (team_file_id)
);

DROP USER IF EXISTS 'request_response_db_adminuser'@'localhost';
CREATE USER 'request_response_db_adminuser'@'localhost' IDENTIFIED BY 'password123';
GRANT ALL PRIVILEGES ON request_response_db.* TO
'request_response_db_adminuser'@'localhost';
ALTER USER 'request_response_db_adminuser'@'localhost' IDENTIFIED WITH
mysql_native_password BY 'password123';
FLUSH PRIVILEGES;
```