

Nathan Dinh
04/14/2024

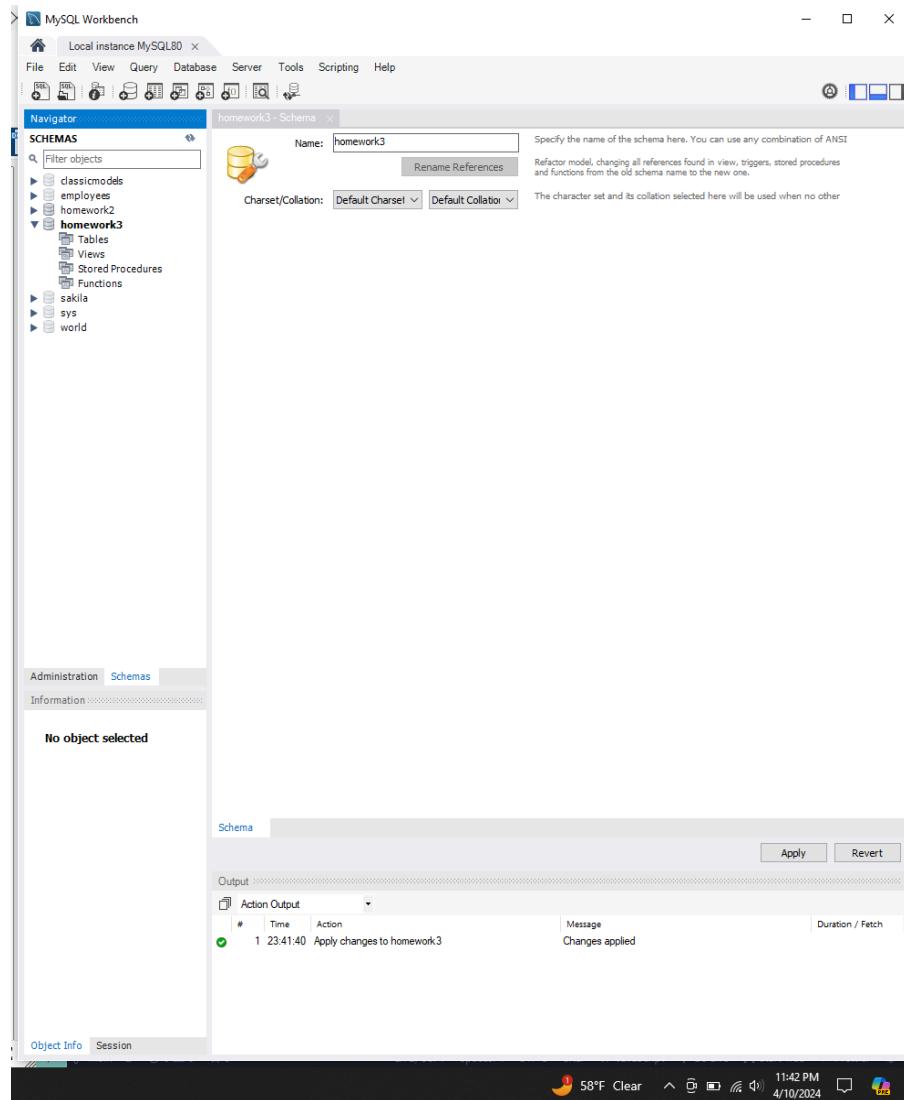
How To Deploy MySQL on AWS RDS and Express API on Render

The code is available on GitHub: <https://github.com/nathan-dinh-dev/mysql-api>
Please review the git commits to follow the coding steps

**** Skip Part 1 & 2 if you already had database schema & express API code already**
**** Part 5 is for the Deployment Strategy**

Part 1: Database Schema Creation

Step 1: Create a “homework3” schema



Schema Creation

Step 2: Create a small library database application (Books and Authors Tables)

Books table has 4 columns which are BookID (primary key), AuthorID (foreign key), Title, and Genre

Authors table has 3 columns which are AuthorID (primary key), AuthorName, and Country

The screenshot shows the MySQL Workbench interface with the following details:

- Schemas:** Local instance MySQL80, homework3
- SQL File 3:** Contains the SQL code for creating the Books and Authors tables.

```
1 -- Create Books Table
2 -- CREATE TABLE Books (
3   -- BookID INT PRIMARY KEY,
4   -- AuthorID INT,
5   -- Title VARCHAR(255),
6   -- Genre VARCHAR(255)
7   --
8
9   -- Create Authors Table
10 -- CREATE TABLE Authors (
11   -- AuthorID INT PRIMARY KEY,
12   -- AuthorName VARCHAR(255),
13   -- Country VARCHAR(255)
14   --
15 )
```

- Information:** No object selected
- Action Output:** Shows the results of the executed queries.

#	Time	Action	Message	Duration / Fetch
1	23:41:40	Apply changes to homework3	Changes applied	
2	00:54:27	CREATE TABLE Books (BookID INT PRIMARY KEY, A...	0 row(s) affected	0.047 sec
3	00:54:53	CREATE TABLE Authors (AuthorID INT PRIMARY KEY,...	0 row(s) affected	0.031 sec

- Object Info:** Session

Create Books and Authors' tables

Step 3: Insert Sample Data into Authors & Books tables

```

15
16      -- INSERT INTO Authors (AuthorID, AuthorName, Country)
17      -- VALUES
18      --     (1, "John Doe", "USA"),
19      --     (2, "Jane Smith", "UK")
20
21      -- INSERT INTO Books (BookID, AuthorID, Title, Genre)
22      -- VALUES
23      --     (1, 1, "The Great Novel", "Fiction"),
24      --     (2, 2, "Science 101", "Education"),
25      --     (3, 1, "Another Story", "Fiction")
26
27 •  Select * from Authors;

```

Result Grid | Filter Rows: | Edit: | Export/Import:

	AuthorID	AuthorName	Country
▶	1	John Doe	USA
2	Jane Smith	UK	
*	NULL	NULL	NULL

Authors Table

```

15
16      -- INSERT INTO Authors (AuthorID, AuthorName, Country)
17      -- VALUES
18      --     (1, "John Doe", "USA"),
19      --     (2, "Jane Smith", "UK")
20
21      -- INSERT INTO Books (BookID, AuthorID, Title, Genre)
22      -- VALUES
23      --     (1, 1, "The Great Novel", "Fiction"),
24      --     (2, 2, "Science 101", "Education"),
25      --     (3, 1, "Another Story", "Fiction")
26
27 •  Select * from Books;

```

Result Grid | Filter Rows: | Edit: | Export/Import:

	BookID	AuthorID	Title	Genre
▶	1	1	The Great Novel	Fiction
2	2	Science 101	Education	
3	1	Another Story	Fiction	
*	NULL	NULL	NULL	NULL

Books Table

Part 2: Database Connection

Step 1: Create a JavaScript file responsible for database information to make connections, such as the host, user, password,...

```
import "dotenv/config";
```

```
const config = {
  db: {
    host: process.env.MYSQL_HOST,
    user: process.env.MYSQL_USER,
    password: process.env.MYSQL_PASSWORD,
    database: process.env.MYSQL_DB,
    port: process.env.MYSQL_PORT,
    connectTimeout: 60000,
  },
  listPerPage: 10,
};

export default config;
```

The .env file will store sensitive information for the database connection. To ensure a secure connection, this file will be utilized to conceal all sensitive information.

Step 2: Make a connection to the database with an exception handler and print out the retrieved data to the console to confirm a successful database connection.

```
import config from "../config.js";
import mysql from "mysql2/promise";

try {
  console.log(config.db);

  const connection = await mysql.createConnection(config.db);

  const [results, fields] = await connection.query("SELECT * FROM `books`");

  console.log(results);
  console.log(fields);
} catch (error) {
  console.log(error);
}
```

The screenshot shows a terminal window in Visual Studio Code. The terminal tab is selected at the top. The command `node db.js` is run, which triggers an error message:

```
[`BookID` INT NOT NULL PRIMARY KEY,
`AuthorID` INT,
`Title` VARCHAR(255),
`Genre` VARCHAR(255)
]
PS C:\Users\nghia\Desktop\Learning Web Coding\mysql-api\services> node db.js
● Error: Access denied for user 'root'@'localhost' (using password: YES)
    at Object.createConnection (C:\Users\nghia\Desktop\Learning Web Coding\mysql-a
pi\node_modules\mysql2\promise.js:253:31)
    at file:///C:/Users/nghia/Desktop/Learning%20Web%20Coding/mysql-api/services/d
b.js:5:34
    at ModuleJob.run (node:internal/modules/esm/module_job:194:25) {
  code: 'ER_ACCESS_DENIED_ERROR',
  errno: 1045,
  sqlState: '28000'
}
PS C:\Users\nghia\Desktop\Learning Web Coding\mysql-api\services>
```

The terminal interface includes status indicators like Ln 14, Col 1, Spaces: 2, UTF-8, CRLF, and various icons for Go Live, Start Tree, Prettier, and system notifications. The bottom bar also shows the date and time (3:05 AM, 4/11/2024).

Testing Error Handler When Wrong Password Input

The screenshot shows a Visual Studio Code (VS Code) interface. The top navigation bar has tabs for index.js, db.js (which is currently active), config.js, package.json, .gitignore, and other files. Below the tabs, the code editor displays db.js with the following content:

```
import config from "../config.js";
import mysql from "mysql2/promise";

try {
  const connection = await mysql.createConnection(config.db);

  const [results, fields] = await connection.query("SELECT * FROM `books`")

  console.log(results);
  console.log(fields);
} catch (error) {
  console.log(error);
}
```

The bottom panel shows the terminal window with the following output:

```
`Genre` VARCHAR(255)
]
PS C:\Users\nggia\Desktop\Learning Web Coding\mysql-api\services> node db.js
[ [
  [Object: null prototype] {
    BookID: 1,
    AuthorID: 1,
    Title: 'The Great Novel',
    Genre: 'Fiction'
  },
  [Object: null prototype] {
    BookID: 2,
    AuthorID: 2,
    Title: 'Science 101',
    Genre: 'Education'
  },
  [Object: null prototype] {
    BookID: 3,
    AuthorID: 1,
    Title: 'Another Story',
    Genre: 'Fiction'
  }
]
[
  `BookID` INT NOT NULL PRIMARY KEY,
  `AuthorID` INT,
  `Title` VARCHAR(255),
  `Genre` VARCHAR(255)
]
```

The terminal also lists two open sessions: "node mysql..." and "node services".

The Success Result From the Terminal

Part 3: Building APIs

During this part, I used JavaScript with Express as a node.js framework to develop APIs that perform CRUD operations on the MySQL database.

Step 1: Install npm packages such as axios, cors, dotenv, express, mysql2, and nodemon for the application

Run this in vscode terminal: "*npm install axios cors dotenv express mysql2 nodemon*"

Axios: A JavaScript library used to make HTTP requests from node.js. It simplifies the process of sending asynchronous HTTP requests to REST endpoints and performing CRUD operations.

CORS: CORS stands for Cross-Origin Resource Sharing. It's a security feature implemented by web browsers to control requests made to a different domain than the one that served the web page. When developing web applications, CORS issues often arise when making requests from one domain to another. It is used in this app to make requests to localhost:5000.

dotenv: dotenv is a popular npm package that loads environment variables from a .env file into process.env. It is used in this app to hide all sensitive information that is needed for the connection such as password,...

Express: Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It is used in this app to provide a set of middleware functions and routes to handle HTTP requests and responses.

mysql2: mysql2 is a fast MySQL driver for Node.js that provides support for promises, async/await, and other modern JavaScript features. It allows developers to interact with MySQL databases using non-blocking, asynchronous methods, making it efficient for handling database operations in Node.js applications.

Nodemon: Nodemon is a utility tool that monitors changes in your Node.js application and automatically restarts the server whenever changes are detected. It's commonly used during the development phase to streamline the development process by eliminating the need to manually restart the server after making changes to the code.

Step 2: Create Express API which is listening on port 5000

```
import express from "express";
import cors from "cors";
import "dotenv/config";
import router from "./apiRouter.js";

const app = express();

const PORT = process.env.PORT || 5000;

const corsOptions = {
  origin: "http://localhost:5000",
  optionsSuccessStatus: 200,
};

// Middleware
```

```

app.use(cors(corsOptions));

// parse requests of content-type - application/json
app.use(express.json());

// parse requests of content-type - application/x-www-form-urlencoded
app.use(
  express.urlencoded({
    extended: true,
  })
);

// API routes
app.use("/", router);

app.get("/testAPI", (req, res) => {
  res.json({ message: "ok" });
});

app.listen(PORT, () => {
  console.log("Listening on Port", PORT);
});

```

Step 3: Create class DbService which holds CRUD methods

1. Async function for when receiving GET request

```

// GET method
async getAllData(tableName) {
  try {
    if (tableName === "books") {
      const [results, fields] = await connection.query(
        "SELECT * FROM `books`"
      );

      return results;
    } else if (tableName === "authors") {
      const [results, fields] = await connection.query(
        "SELECT * FROM `authors`"
      );
    }
  }
}

```

```

        return results;
    }
} catch (err) {
    console.log(err.message);
}
}

```

2. Async functions for POST requests

```
// POST method

async insertNewBook(bookID, authorID, title, genre) {
    try {
        const query =
            "INSERT INTO `books` (`BookID`, `AuthorID`, `Title`,
`Genre`) VALUES (?, ?, ?, ?)";
        const [results, fields] = await connection.query(query, [
            bookID,
            authorID,
            title,
            genre,
        ]);
        return results;
    } catch (error) {
        console.log(error);
    }
}
```

```
async insertNewAuthor(authorID, authorName, country) {
    try {
        const query =
            "INSERT INTO `authors` (`AuthorID`, `AuthorName`, `Country`)
VALUES (?, ?, ?)";
        const [results, fields] = await connection.query(query, [
            authorID,
            authorName,
            country,
        ]);
        return results;
    } catch (error) {
        console.log(error);
    }
}
```

```
}
```

3. Async functions for PUT method

```
// PUT method

async updateAuthor({ authorID, authorName, country }) {
  try {
    let query = "";
    if (authorName && country)
      query = `UPDATE authors SET AuthorName="${authorName}", Country="${country}" WHERE authorID=${authorID} LIMIT 1`;
    else if (authorName)
      query = `UPDATE authors SET AuthorName="${authorName}" WHERE authorID=${authorID} LIMIT 1`;
    else if (country)
      query = `UPDATE authors SET Country="${country}" WHERE authorID=${authorID} LIMIT 1`;

    const [results, fields] = await connection.query(query);
    return results;
  } catch (error) {
    console.log(error);
  }
}

async updateBook({ bookID, authorID, title, genre }) {
  try {
    let query = "";
    if (bookID && authorID && title && genre)
      query = `UPDATE books SET AuthorID=${authorID}, Title="${title}", Genre="${genre}" WHERE bookID=${bookID} LIMIT 1`;
    else if (authorID && title)
      query = `UPDATE books SET AuthorID=${authorID}, Title="${title}" WHERE bookID=${bookID} LIMIT 1`;
    else if (authorID && genre)
      query = `UPDATE authors SET AuthorID=${authorID}, Genre="${genre}" WHERE bookID=${bookID} LIMIT 1`;
    else if (genre && title)
      query = `UPDATE books SET Genre="${genre}", Title="${title}" WHERE bookID=${bookID} LIMIT 1`;
  }
}
```

```

        else if (authorID)
            query = `UPDATE books SET AuthorID=${authorID} WHERE
bookID=${bookID} LIMIT 1`;
        else if (genre)
            query = `UPDATE books SET Genre="${genre}" WHERE bookID=${bookID}
LIMIT 1`;
        else if (title)
            query = `UPDATE books SET Title="${title}" WHERE bookID=${bookID}
LIMIT 1`;

        const [results, fields] = await connection.query(query);
        return results;
    } catch (error) {
        console.log(error);
    }
}

```

4. Async functions for DELETE requests

```

// Delete Method
async deleteBook({ bookID, authorID, title, genre }) {
    try {
        let query = "";
        if (bookID) query = `DELETE FROM books WHERE BookID=${bookID}
LIMIT 1`;
        else if (authorID)
            query = `DELETE FROM books WHERE AuthorID=${authorID} LIMIT
1`;
        else if (genre)
            query = `DELETE FROM books WHERE Genre="${genre}" LIMIT 1`;
        else if (title)
            query = `DELETE FROM books WHERE Title="${title}" LIMIT 1`;

        const [results, fields] = await connection.query(query);
        return results;
    } catch (error) {
        console.log(error);
    }
}

```

```

async deleteAuthor({ authorID, authorName, country }) {
  try {
    let query = "";
    if (authorID)
      query = `DELETE FROM authors WHERE AuthorID=${authorID}
LIMIT 1`;
    else if (authorName)
      query = `DELETE FROM authors WHERE
AuthorName="${authorName}" LIMIT 1`;
    else if (country)
      query = `DELETE FROM authors WHERE Country="${country}"

LIMIT 1`;

    const [results, fields] = await connection.query(query);
    return results;
  } catch (error) {
    console.log(error);
  }
}
}

```

Step 4: Create apiRouter.js which handles POST, GET, PUT, and DELETE requests

1. Create a new row in the database using **POST** requests

```
// Create a new row in the database
```

Add a new book

```

router.post("/insert/book", (req, res) => {
  const { bookID, authorID, title, genre } = req.body;
  const db = new DbService();
  const result = db.insertNewBook(bookID, authorID, title, genre);

  result
    .then((data) => res.json({ message: "ok", data: data }))
    .catch((err) => console.log(err));
});

```

Add a new author:

```

router.post("/insert/author", (req, res) => {
  const { authorID, authorName, country } = req.body;

```

```

const db = new DbService();
const result = db.insertNewAuthor(authorID, authorName, country);

result
  .then((data) => res.json({ message: "ok", data: data }))
  .catch((err) => console.log(err));
}) ;

```

2. Retrieve data from database using **GET** requests

Get all books in the books table:

```

// Retrieve data from database
router.get("/get", (req, res) => {});

router.get("/get/books/all", (req, res) => {
  const db = new DbService();

  const result = db.getAllData("books");

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err));
}) ;

```

Get all authors in authors table:

```

router.get("/get/authors/all", (req, res) => {
  const db = new DbService();

  const result = db.getAllData("authors");

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err));
}) ;

```

3. Update tables in the database using **PUT** requests

Update Author's table

```

router.put("/update/author", (req, res) => {
  const db = new DbService();

  const result = db.updateAuthor(req.body);

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err.message));
});

```

Update Books table

```

router.put("/update/book", (req, res) => {
  const db = new DbService();

  const result = db.updateBook(req.body);

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err.message));
});

```

4. Delete row in the database using DELETE requests

Delete a book:

```

// Delete database
router.delete("/delete/book", (req, res) => {
  const db = new DbService();

  const result = db.deleteBook(req.body);

  result
    .then((data) => res.json({ data: data }))
    .catch((err) => console.log(err.message));
});

```

Delete an author:

```

router.delete("/delete/author", (req, res) => {
  const db = new DbService();

  const result = db.deleteAuthor(req.body);
}

```

```

result
  .then( (data) => res.json({ data: data }) )
  .catch( (err) => console.log(err.message) );
} );

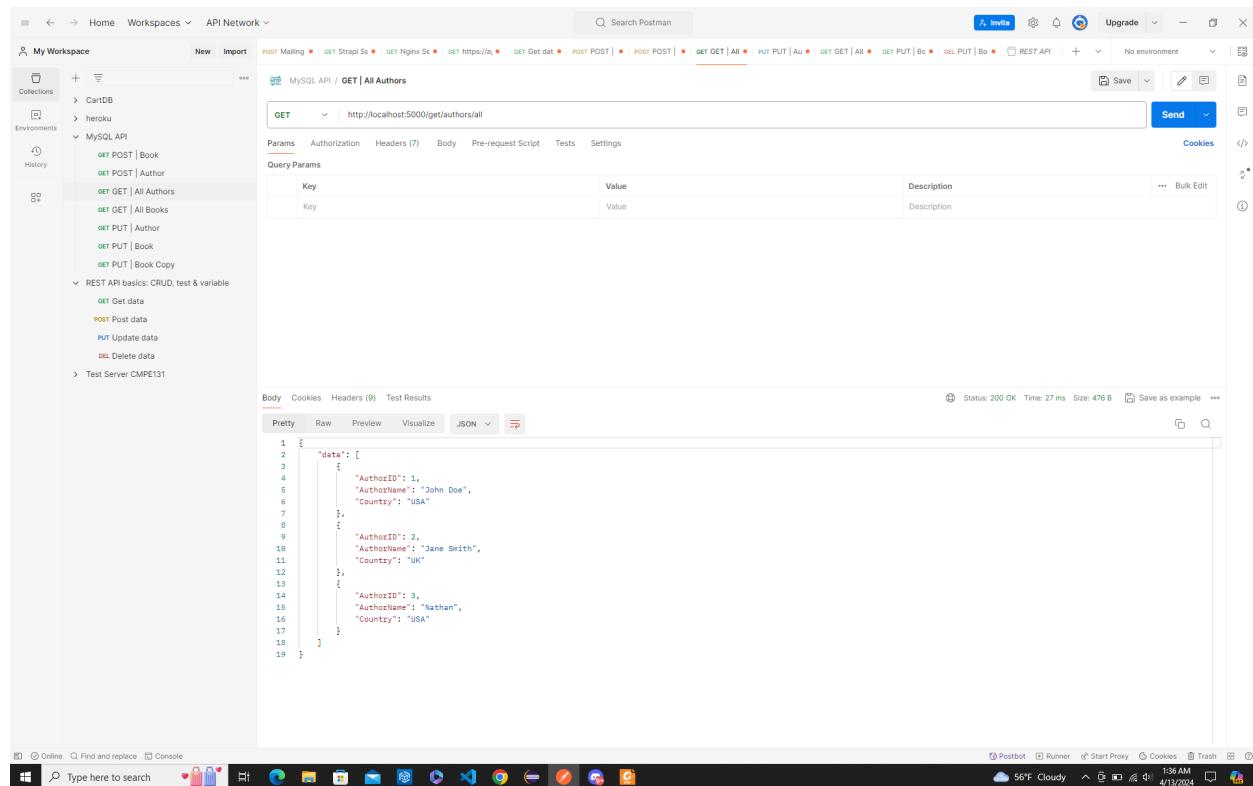
```

Part 4: API Testing

I used Postman for testing API in this assignment.

1. Testing **GET** requests

Test Success GET request to retrieve all books and all authors
 Expected to receive all data in JSON file



The screenshot shows the Postman application interface. On the left, there's a sidebar with 'My Workspace' containing collections like 'CarDB', 'heroku', and 'MySQL API'. Under 'MySQL API', several requests are listed: 'GET POST | Book', 'GET POST | Author', 'GET GET | All Authors', 'GET GET | All Books', 'GET PUT | Author', 'GET PUT | Book', and 'GET PUT | Book Copy'. Below these, under 'REST API basics: CRUD, test & variable', are 'GET Get data', 'POST Post data', 'PUT Update data', and 'DEL Delete data'. At the bottom of the sidebar, it says 'Test Server CMP131'.

The main area shows a 'GET | All Authors' request. The URL is set to `http://localhost:5000/get/authors/all`. The 'Params' tab is selected, showing a single query parameter 'Key' with value 'Value'. The 'Body' tab shows a JSON response:

```

1 {
2   "data": [
3     {
4       "AuthorID": 1,
5       "AuthorName": "John Doe",
6       "Country": "USA"
7     },
8     {
9       "AuthorID": 2,
10      "AuthorName": "Jane Smith",
11      "Country": "UK"
12    },
13    {
14      "AuthorID": 3,
15      "AuthorName": "Nathan",
16      "Country": "USA"
17    }
18  ]
19 }

```

The status bar at the bottom indicates 'Status: 200 OK Time: 27 ms Size: 476 B Save as example'. The taskbar at the bottom of the screen shows various icons for other applications.

MySQL API | GET | All Books

```

1 {
2   "data": [
3     {
4       "BookID": 1,
5       "AuthorID": 1,
6       "Title": "The Great Novel",
7       "Genre": "Fiction"
8     },
9     {
10       "BookID": 2,
11       "AuthorID": 2,
12       "Title": "Science 101",
13       "Genre": "Education"
14     },
15     {
16       "BookID": 3,
17       "AuthorID": 1,
18       "Title": "Another Story",
19       "Genre": "Fiction"
20     },
21     {
22       "BookID": 4,
23       "AuthorID": null,
24       "Title": "Test API",
25       "Genre": "Test API"
26     }
]

```

Success GET requests

Test Error GET requests

Expected to receive ERROR message “Cannot GET request”

MySQL API | GET | All Authors

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Error</title>
6   </head>
7   <body>
8     <p>Error: Cannot GET /get/authors/1</p>
9   </body>
10  </html>

```

Error GET request

2. Testing POST requests

Test Success POST request

Expected to receive all data in JSON:

```
{  
    "message": "ok",  
    "data": {  
        "fieldCount": 0,  
        "affectedRows": 1,  
        "insertId": 0,  
        "info": "",  
        "serverStatus": 2,  
        "warningStatus": 0,  
        "changedRows": 0  
    }  
}
```

The screenshot shows the Postman interface with a successful POST request to `http://localhost:5000/insert/book`. The request body is defined with the following parameters:

Key	Value	Description
bookID	5	
authorID	3	
title	API	
genre	Programming	

The response body is displayed in the Body tab:

```
1 {  
2     "message": "ok",  
3     "data": {  
4         "fieldCount": 0,  
5         "affectedRows": 1,  
6         "insertId": 0,  
7         "info": "",  
8         "serverStatus": 2,  
9         "warningStatus": 0,  
10        "changedRows": 0  
11    }  
12 }
```

Success POST requests

Test Error POST request with NULL primary key

Expected to receive error JSON message

```
{  
    "message": "ok",  
    "data": {
```

```

        "message": "Incorrect integer value: 'null' for column 'BookID' at row 1",
        "code": "ER_TRUNCATED_WRONG_VALUE_FOR_FIELD",
        "errno": 1366,
        "sql": "INSERT INTO `books` (`BookID`, `AuthorID`, `Title`, `Genre`) VALUES ('null', '3', 'API', 'Programming')",
        "sqlState": "HY000",
        "sqlMessage": "Incorrect integer value: 'null' for column 'BookID' at row 1"
    }
}

```

The screenshot shows the Postman interface with a failed POST request to `http://localhost:5000/insert/book`. The request body is set to `x-www-form-urlencoded` and contains the following parameters:

Key	Value	Description
bookID	null	
authorID	3	
title	API	
genre	Programming	

The response body is a JSON object containing error details:

```

1 {
2     "message": "ok",
3     "data": null,
4     "error": {
5         "message": "Incorrect integer value: 'null' for column 'BookID' at row 1",
6         "code": "ER_TRUNCATED_WRONG_VALUE_FOR_FIELD",
7         "errno": 1366,
8         "sql": "INSERT INTO `books` (`BookID`, `AuthorID`, `Title`, `Genre`) VALUES ('null', '3', 'API', 'Programming')",
9         "sqlState": "HY000",
10        "sqlMessage": "Incorrect integer value: 'null' for column 'BookID' at row 1"
11    }
12 }

```

Error POST requests

3. Testing PUT requests

Test Success POST request

Expected to receive all data in JSON:

```
{
    "data": {
        "fieldCount": 0,
        "affectedRows": 1,
        "insertId": 0,
        "info": "Rows matched: 1  Changed: 1  Warnings: 0",
        "serverStatus": 2,
        "warningStatus": 0,
    }
}
```

```

        "changedRows": 0
    }
}

```

The screenshot shows the Postman interface with a collection named "MySQL API". A PUT request is being made to `http://localhost:5000/update/author`. The request body is set to "x-www-form-urlencoded" and contains three parameters: `authorID` (value 3), `authorName` (value Nathan), and `country` (value USA). The response tab shows a status of 200 OK with a JSON response body:

```

1 {
2     "data": {
3         "fieldCount": 0,
4         "affectedRows": 1,
5         "insertId": 0,
6         "info": "Rows matched: 1 Changed: 0 Warnings: 0",
7         "serverStatus": 2,
8         "warningStatus": 0,
9         "changedRows": 0
10    }
11 }

```

Success PUT requests

Test Error PUT request with not found primary key

Expected to receive error JSON message

```

{
    "data": {
        "fieldCount": 0,
        "affectedRows": 0,
        "insertId": 0,
        "info": "Rows matched: 0 Changed: 0 Warnings: 0",
        "serverStatus": 2,
        "warningStatus": 0,
        "changedRows": 0
    }
}

```

The screenshot shows the Postman interface with a collection named "MySQL API". A PUT request is being tested against the URL `http://localhost:5000/update/author`. The request body is set to "x-www-form-urlencoded" and contains the following key-value pairs:

Key	Value
authorID	5
authorName	Nathan
country	USA

The response details show a status of 200 OK, a time of 10 ms, and a size of 458 B. The JSON response body is:

```

1 {
2   "data": [
3     {"fieldCount": 0,
4      "affectedRows": 0,
5      "insertId": 0,
6      "info": "Rows matched: 0  Changed: 0  Warnings: 0",
7      "serverStatus": 2,
8      "warningStatus": 0,
9      "changedRows": 0
10    }
11  }

```

Error PUT requests

Explanation: There is no row found that match the record in the table to make changes

`"info": "Rows matched: 0 Changed: 0 Warnings: 0",`

Test Error PUT request with empty query

Expected to receive error JSON message

```
{
  "data": {
    "message": "Query was empty",
    "code": "ER_EMPTY_QUERY",
    "errno": 1065,
    "sqlState": "42000",
    "sqlMessage": "Query was empty"
  }
}
```

The screenshot shows the Postman interface with the following details:

- Collection:** MySQL API
- Request Type:** PUT
- URL:** http://localhost:5000/update/author
- Body (x-www-form-urlencoded):**

Key	Value	Description
authorID	3	
authorName	Nathan	
country	USA	
- Response Body (Pretty JSON):**

```

1 {
2     "data": [
3         {
4             "message": "Query was empty",
5             "code": "ER_EMPTY_QUERY",
6             "errno": 1065,
7             "sqlState": "42000",
8             "sqlMessage": "Query was empty"
9         }
10 }

```
- Status:** 200 OK
- Time:** 15 ms
- Size:** 427 B

Error PUT requests

4. Testing **DELETE** requests

Test Success **DELETE** request with **BookID = 3**

Expected to receive all data in JSON:

```
{
    "data": {
        "fieldCount": 0,
        "affectedRows": 1,
        "insertId": 0,
        "info": "",
        "serverStatus": 2,
        "warningStatus": 0,
        "changedRows": 0
    }
}
```

The screenshot shows the Postman interface with a successful DELETE request. The request URL is `http://localhost:5000/delete/book`. The request body is set to `x-www-form-urlencoded` and contains a key `bookID` with value `3`. The response status is `200 OK`, and the response body is a JSON object with the following content:

```

1
2   "data": {
3     "fieldCount": 0,
4     "affectedRows": 1,
5     "insertId": 0,
6     "info": "",
7     "serverStatus": 2,
8     "warningStatus": 0,
9     "changedRows": 0
10  }
11

```

Success **DELETE** requests

Test Error **DELETE** request with empty query

Expected to receive error JSON message

```
{
  "data": {
    "message": "Query was empty",
    "code": "ER_EMPTY_QUERY",
    "errno": 1065,
    "sqlState": "42000",
    "sqlMessage": "Query was empty"
  }
}
```

The screenshot shows the Postman interface with a collection named 'MySQL API'. A DELETE request is being made to `http://localhost:5000/delete/book`. The request body is set to 'x-www-form-urlencoded' and contains a key 'bookID' with a value of '3'. The response status is 200 OK, and the JSON body is:

```

1 {
2   "data": [
3     {
4       "message": "Query was empty",
5       "code": "ER_EMPTY_QUERY",
6       "errno": 1065,
7       "sqlState": "42000",
8       "sqlMessage": "Query was empty"
9     }
10 ]

```

Error PUT requests

Part 5: Deployment Strategy

A. Deploy MySQL on AWS RDS DB Instance

Step 1: Create an AWS account with root user and login to aws console

Step 2: Go to RDS dashboard and click on the orange button “Create Database”

mysql-api - Web Service - Rename RDS | us-east-2

us-east-2.console.aws.amazon.com/rds/home?region=us-east-2

Online Courses | Learn Paraphrasing Tool | Codeforces | MDN Web Docs | HTML elements reference | one.SJSU | Most Popular | Professional Certifications | Airbnb JavaScript Study Group | Parchment Exchange | Hoppscotch | OpenCourseWare | CSS transitions: Animation | Webinar Popmenu | All Bookmarks

AWS Services Search [Alt+S]

Amazon RDS

Introducing Aurora I/O-Optimized
Aurora's I/O-Optimized is a new cluster storage configuration that offers predictable pricing for all applications and improved price-performance, with up to 40% costs savings for I/O-intensive applications.

Resources

You are using the following Amazon RDS resources in the US East (Ohio) region (used/quota)

- DB Instances (0/40)
 - Allocated storage (0 TB/100 TB)
 - Increase DB instances limit
- DB Clusters (0/40)
- Reserved instances (0/40)
- Snapshots (1)
 - Manual
 - DB Cluster (0/100)
 - DB Instance (1/100)
- Automated
- Recent events (0)
- Event subscriptions (0/20)

Parameter groups (1)

- Default (1)
- Custom (0/100)

Option groups (1)

- Default (1)
- Custom (0/20)

Subnet groups (1/50)

- Supported platforms VPC
- Default network vpc-0998b03a9ee9acb5

Recommended services

Customers like you also use these services.

- CloudFront Global Content Delivery Network
- AWS Organizations Central governance and management across AWS accounts.
- Amazon Simple Email Service Email Sending and Receiving Service
- Cloud9 A Cloud IDE for Writing, Running, and Debugging Code

Recommended for you

Test Your DR Strategy in Minutes
Amazon Aurora Global Database now supports planned managed failover, making disaster recovery drills a breeze. [Learn more](#)

Amazon RDS Backup and Restore using AWS Backup
Learn how to backup and restore Amazon RDS databases using AWS Backup in just 10 minutes. [Learn more](#)

Time-Series Tables in PostgreSQL
Step-by-step guide to design high-performance time series data tables on Amazon RDS for PostgreSQL. [Learn more](#)

Build RDS Operational Tasks
Watch how to enable users to perform common tasks such as snapshots or restart DB instances in Amazon RDS. [Learn more](#)

Additional Information

© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

CloudShell Feedback Type here to search 48°F Rain 104 AM 4/14/2024

Step 4: For testing/learning/practicing, I chose Easy Create with MySQL engine

mysql-api - Web Service - Rename RDS | us-east-2

us-east-2.console.aws.amazon.com/rds/home?region=us-east-2#launch-dbinstance

Online Courses | Learn Paraphrasing Tool | Codeforces | MDN Web Docs | HTML elements reference | one.SJSU | Most Popular | Professional Certifications | Airbnb JavaScript Study Group | Parchment Exchange | Hoppscotch | OpenCourseWare | CSS transitions: Animation | Webinar Popmenu | All Bookmarks

AWS Services Search [Alt+S]

Your database might take a few minutes to launch. You can use settings from mysql-hw3 to simplify configuration of suggested database add-ons while we finish creating your DB for you.

RDS > Create database

Create database

Choose a database creation method [Info](#)

Standard create
You set all of the configuration options, including ones for availability, security, backups, and maintenance.

Easy create
Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

Configuration

Engine type [Info](#)

Aurora (MySQL Compatible) 

Aurora (PostgreSQL Compatible) 

MySQL 

MariaDB 

PostgreSQL 

Oracle 

Microsoft SQL Server 

Edition

MySQL

MySQL

MySQL is the most popular open source database in the world. MySQL on RDS offers the rich features of the MySQL community edition with the flexibility to easily scale compute resources or storage capacity for your database.

- Supports database size up to 64 TiB.
- Supports General Purpose, Memory Optimized, and Burstable Performance Instance classes.
- Supports automated backup and point-in-time recovery.
- Supports up to 15 Read Replicas per instance, within a single Region or 5 read replicas cross-region.

CloudShell Feedback Type here to search 48°F Light rain 101 AM 4/14/2024

Step 5: Set the root name as admin and your password

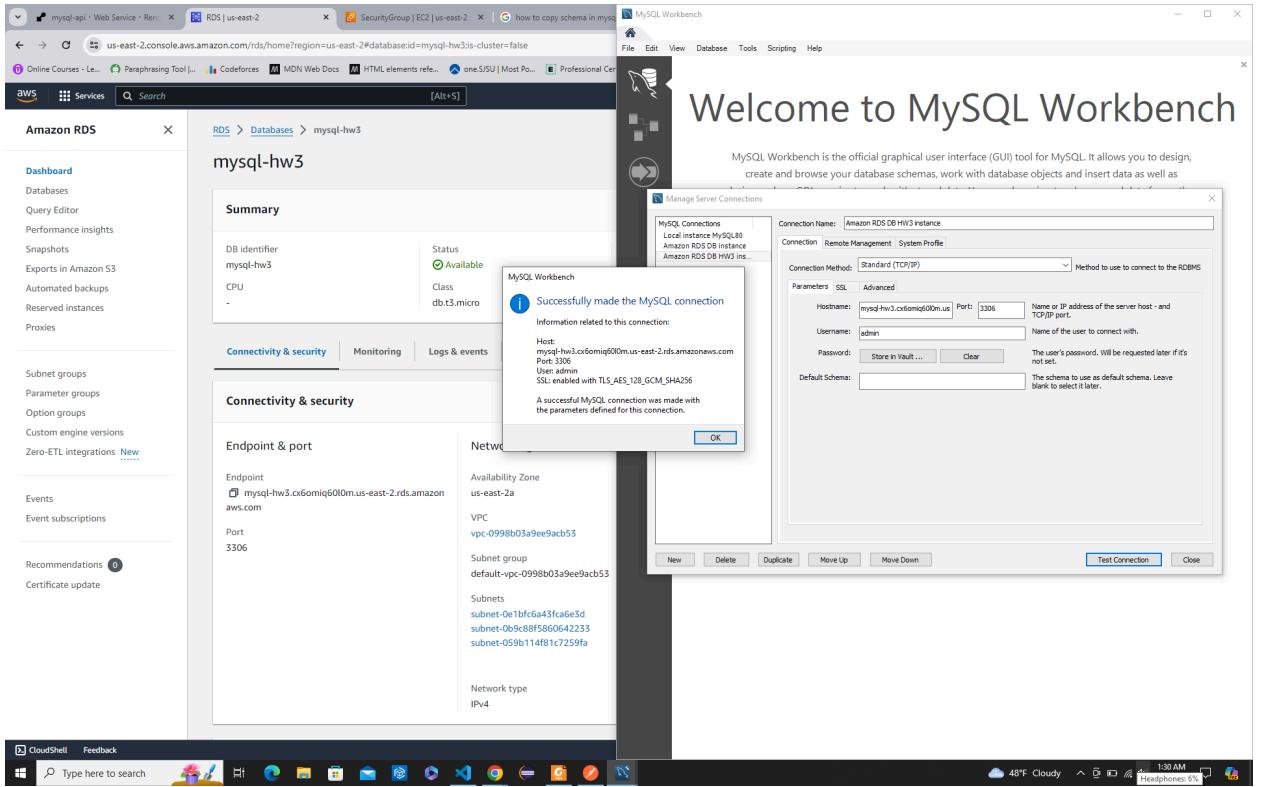
Step 7: After creating, save the port (default is 3306) and the endpoint for linking the MySQL later on

The screenshot shows the AWS RDS console for a MySQL database named 'mysql-hw3'. The 'Summary' tab is selected, displaying details like DB identifier, Status (Configuring-enhanced-monitoring), Role (Instance), Engine (MySQL Community), and Region & AZ (us-east-2a). The 'Connectivity & security' tab is also visible, showing the Endpoint (mysql-hw3.cx6omiq60l0m.us-east-2.rds.amazonaws.com), Port (3306), Networking (Availability Zone: us-east-2a, VPC: vpc-0998b03a9ee9acb53, Subnet group: default-vpc-0998b03a9ee9acb53, Subnets: subnet-0e1bfc6a43fc46e3d, subnet-0b9c88f5860642233, subnet-059b11481c7259fa), and Security (VPC security groups: default (sg-0c15bd8e8b1185f01), Active). The status bar at the bottom indicates it's 48°F Light rain, 11:14 AM, 4/14/2024.

Step 8: Change the access rule to public and EC2 rule to access from any IP address for testing purposes:

- Go to EC2 Dashboard
- Go to the Security Groups tab
- Select and only select the RDS database security group. You'll see the security group detail at the bottom
- Click Inbound tab
- Click Edit button
- Add-Type:MYSQL/Aurora;Protocol:TCP;Range:3306;Source:0.0.0.0/0

Step 9: Test MySQL connection



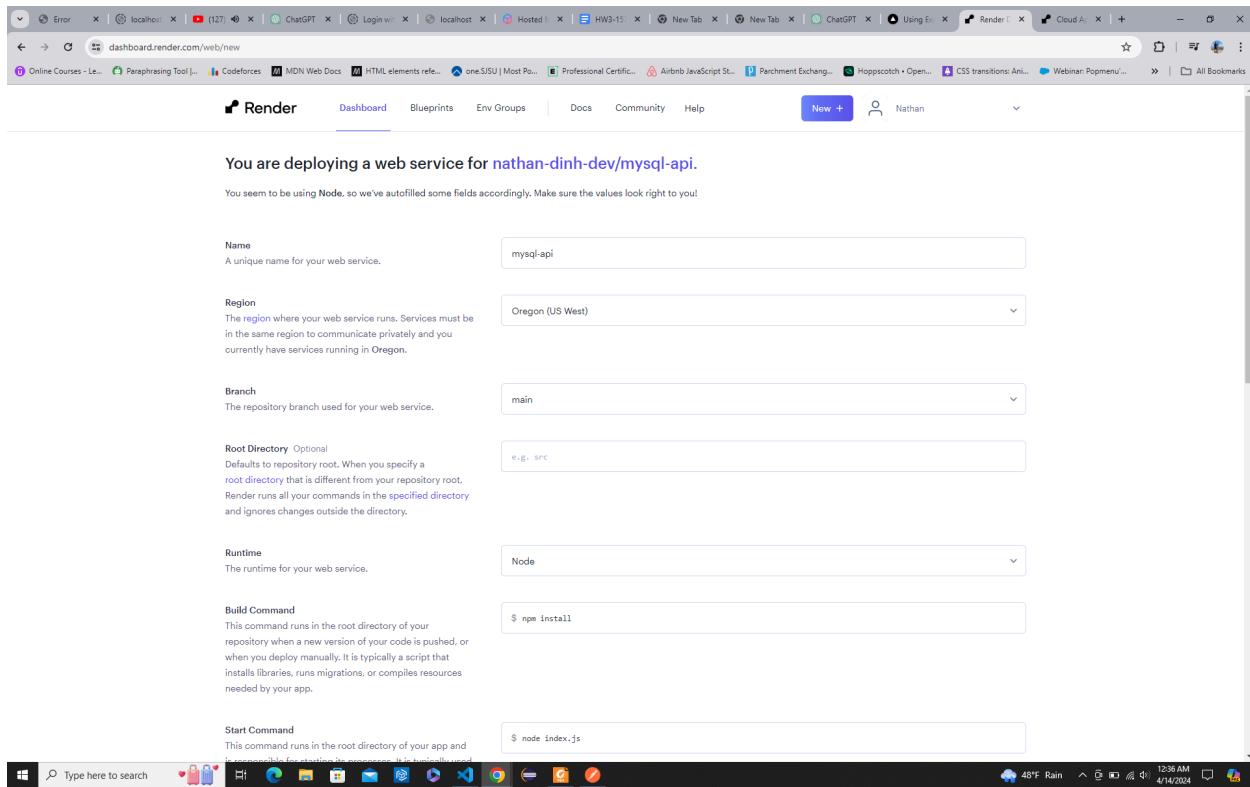
B. Deploy your NodeJs API on [Render](#) that links to your deployed MySQL

Step 1: Create an account on Render

Step 2: Create a new Render Web Service

Step 3: Authorize Render to access your GitHub repositories

Step 4: Setting the deployment configuration



Note: You need to override:

Build Command: npm install

Start Command: node index.js

Added Environment Keys which are from MySQL AWS that already deployed

Example in my application:

MYSQL_DB="homework3"

MYSQL_HOST="mysql-hw3.cx6omiq60l0m.us-east-2.rds.amazonaws.com" // This
need to be same as the one you created on AWS RDS

MYSQL_PORT=3306

MYSQL_USER="admin"

MYSQL_PASSWORD="*****"

Step 5: Setting deployment commands and choosing an instance type

Step 6: Overriding the default Node version

Step 7: Deployed successfully

The screenshot shows the Render dashboard for the mysql-api service. The service is listed as a WEB SERVICE named mysql-api, running on Node.js and currently free. A message indicates that the instance will spin down if inactive for 50 seconds. The Logs section shows deployment logs for the main branch, including messages about Node.js version 20.12.1, Bun version 1.1.0, and the service starting on port 10000. The URL <https://mysql-api-wb0w.onrender.com> is displayed at the bottom.

This is my API endpoint: <https://mysql-api-wb0w.onrender.com>

Final Step:

Test deployed API with **GET**, **POST**, **PUT**, and **DELETE** using Postman

The screenshot shows a Postman collection named "MySQL API" with various requests for authors and books. One specific GET request is highlighted for the endpoint <https://mysql-api-wb0w.onrender.com/get/authors/all>. The response body is a JSON array of authors:

```

[{"id": 1, "AuthorName": "John Doe", "Country": "USA"}, {"id": 2, "AuthorName": "Jane Smith", "Country": "UK"}, {"id": 3, "AuthorName": "Nathan", "Country": "USA"}]

```

GET Request

POST request

Method: POST
URL: https://myred-api-wb0w.onrender.com/insert/author

Key	Value	Description
authorID	3	
authorName	Nathan	
country	USA	

Body Cookies Headers (15) Test Results

```
{
  "message": "ok",
  "data": [
    {
      "authorID": "3",
      "authorName": "Nathan",
      "country": "USA"
    }
  ],
  "code": "00_OUP_ENTRY",
  "errno": "1062",
  "sql": "INSERT INTO `authors`(`AuthorID`, `AuthorName`, `Country`) VALUES ('3', 'Nathan', 'USA')",
  "sqlstate": "23000",
  "errMessage": "Duplicate entry '3' for key 'authors.PRIMARY'"
}
```

PUT request

Method: PUT
URL: https://myred-api-wb0w.onrender.com/update/author

Key	Value	Description
authorID	3	
authorName	Nathan	
country	USA	

Body Cookies Headers (15) Test Results

```
{
  "data": [
    {
      "authorID": "3",
      "authorName": "Nathan",
      "country": "USA"
    }
  ],
  "affectedRows": 1,
  "insertId": 1,
  "info": "Rows matched: 1  Changed: 0  Warnings: 0",
  "serverStatus": 2,
  "warningStatus": 0,
  "changePlugin": 0
}
```

The screenshot shows the Postman application interface. In the left sidebar, under 'My Workspace', there is a 'Collections' section with 'CarDB' and 'MySQL API'. Under 'MySQL API', several requests are listed: GET POST | Book, GET | Author, GET | All Authors, GET | All Books, PUT | Author, PUT | Book, and DELETE | Book. The 'DELETE | Book' request is selected. The main panel shows a 'DELETE' request to 'https://mysql-api-ws0le.onrender.com/delete/book'. The 'Body' tab is selected, showing a key-value pair 'bookID' with a value of '3'. Below the request, the 'Test Results' section displays a JSON response:

```
1 {  
2   "data": {  
3     "id": 1,  
4     "title": "The Great Gatsby",  
5     "author": "F. Scott Fitzgerald",  
6     "publishedYear": 1925,  
7     "insertedId": 1,  
8     "isbn": "9780451522257",  
9     "rating": 4.5,  
10    "warningStatus": 0,  
11    "changedRows": 0  
12  }  
13}
```

DELETE request

Conclusion: API - MySQL database deployed successfully.

References

MySQL2 npm package documentation: <https://sidorares.github.io/node-mysql2/docs>
AWS rds for deploying MySQL: <https://aws.amazon.com/rds/mysql/>