5. Working with Databases & Node.js

5.1 Introduction: Database and Database Management System (DBMS)

- **What is a Database?**
    - A database is an organized collection of structured information or data. This information is typically stored electronically within a computer system. Databases are designed to efficiently manage, retrieve, and update data.

- **What is a Database Management System (DBMS)?**
    - **Definition:**
        - A DBMS is software designed to facilitate easy, efficient, and reliable data processing and management within databases.
        - Databases are usually controlled by a Database Management System (DBMS).

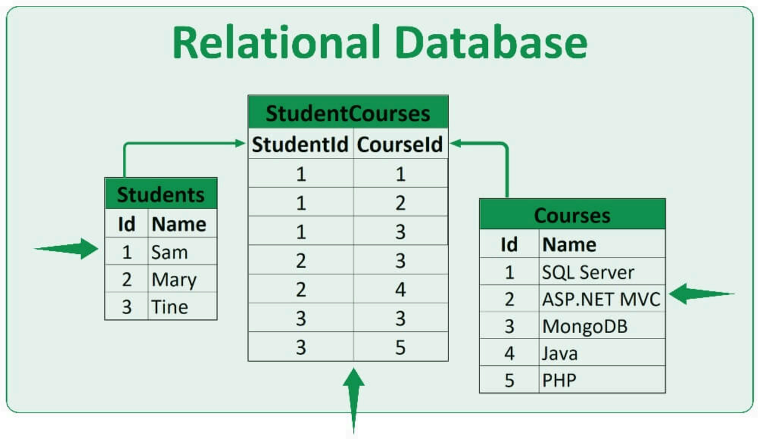
    - **Functions of a DBMS:**
        - **Creation of a Database:** DBMS helps in designing and creating databases as per defined schemas and structures.
        - **Storage of Data/Information:** It manages the storage of various types of data within the database.
        - **Retrieval of Information:** Enables users to efficiently retrieve specific data from the database using queries and commands.
        - **Updating the Database:** Allows for adding, modifying, or deleting data within the database.
        - **Control of Redundancy and Inconsistency:** Helps in minimizing data duplication and ensuring data consistency.
        - **Efficient Memory Management:** Optimizes the utilization of memory resources for data storage and retrieval.
        - **Access Control and Security:** Implements security measures to control access to the database and ensure data integrity.

All in all, the comprehensive database system encompasses both the data, the Database Management System (DBMS), and their associated applications, collectively forming the foundational components referred to as a database system, often shortened to just Database.

5.2 Main Categories of Database: Relational and Non-Relational Database

- **Relational Databases (SQL DB)**
  - **Definition:**
    - Relational databases, often referred to as SQL databases, are structured around multiple tables that interrelate with each other. The structure is akin to an organized set of spreadsheets where each table holds specific types of data. For instance, if you're dealing with an e-commerce platform, there might be a table for customers, another for orders, and so on. This structure allows for efficient management and retrieval of related data. A classic example of a relational database is MySQL.

  - **Table Structure and Relationships:**
    - Each table contains data specific to a particular domain or category, maintaining clear boundaries between different types of information.
    - Tables are intricately linked or related based on **primary key and foreign key**.
      - **A primary key** is a unique identifier within a table, ensuring each record has a distinct value. It uniquely identifies each row and enforces data integrity by preventing duplicate or null values.
      - **A foreign key** establishes a link between two tables, referencing the primary key of another table.
  - **Example**:



Students

Id	Name
1	Sam
2	Mary
3	Tine

StudentCourses

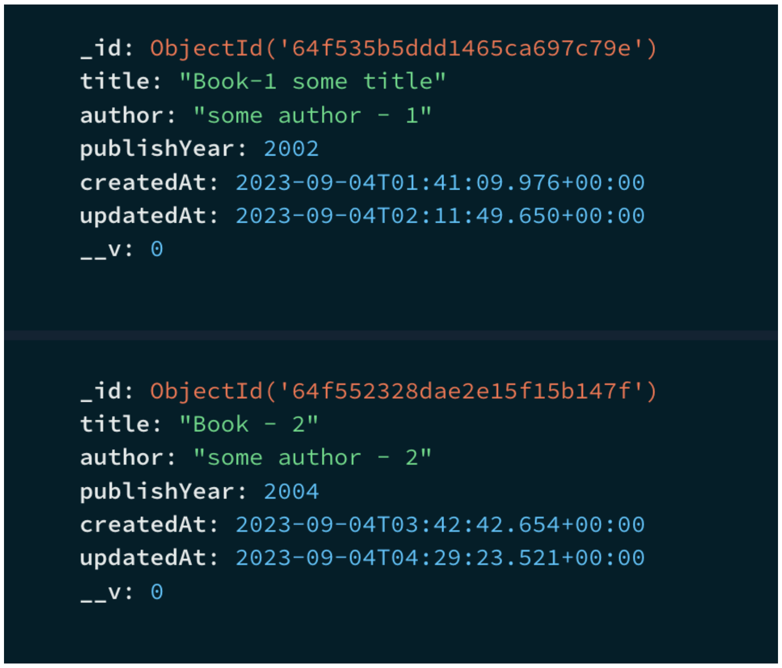
StudentId	CourseId
1	1
1	2
1	3
2	3
2	4
3	3
3	5

Courses

Id	Name
1	SQL Server
2	ASP.NET MVC
3	MongoDB
4	Java
5	PHP

    - In the above example, "Id" columns in the "Students" & "Courses" table are primary keys, and they are referenced onto the "StudentCourses" table as foreign keys.

- **Non-Relational Databases (NoSQL DB)**
  - **Definition:**
    - Non-relational databases, often termed NoSQL databases, diverge from the table-based structure. Instead, they primarily operate around objects or documents. There are no fixed table structures or rigid relationships between tables. One common format in NoSQL databases is the key-value pair model, similar to JSON (JavaScript Object Notation). An example of a non-relational database is MongoDB.
  - **Object-Oriented Approach:**
    - NoSQL databases work with collections of objects rather than structured tables.
    - They typically adopt a more flexible schema, allowing for easier scalability and adaptability.
  - **Example:**



```
_id: ObjectId('64f535b5ddd1465ca697c79e')
title: "Book-1 some title"
author: "some author - 1"
publishYear: 2002
createdAt: 2023-09-04T01:41:09.976+00:00
updatedAt: 2023-09-04T02:11:49.650+00:00
__v: 0
```

```
_id: ObjectId('64f552328dae2e15f15b147f')
title: "Book - 2"
author: "some author - 2"
publishYear: 2004
createdAt: 2023-09-04T03:42:42.654+00:00
updatedAt: 2023-09-04T04:29:23.521+00:00
__v: 0
```

- **Relational vs. Non-Relational: Key Differences**
  - **Structural Variance:**
    - Relational databases rely on tables with predefined structures and relationships.
    - Non-relational databases operate on flexible structures, using varied formats like key-value pairs or document-based storage.

    - **Data Modeling Philosophy:**
        - Relational databases follow the ACID (Atomicity, Consistency, Isolation, Durability) principles, ensuring strong data consistency.
        - Non-relational databases focus on BASE (Basically Available, Soft state, Eventually consistent) principles, emphasizing scalability and availability over strict consistency.

Choosing the Right Fit

The selection between relational and non-relational databases often hinges on the nature of the data and the requirements of the application. Relational databases excel in scenarios requiring structured data and complex queries, while non-relational databases offer scalability and flexibility for unstructured or evolving data models.

5.3 Working with Databases using NodeJS: HTTP Methods / Verbs and Their Functions

- **GET Method: Retrieving Data**
    - The GET method in Node.js allows the retrieval of data from a database. It's like browsing a library to find and read specific information without altering anything.

- **POST Method: Creating New Data**
    - When using the POST method, Node.js can create new data in the database. It's akin to adding a new book to the library's collection.

- **PUT and PATCH Methods: Updating Existing Data**
    - **PUT Method: Complete Replacement**
        - With PUT, Node.js updates existing data by completely replacing it with a new representation. Imagine swapping an entire book with a new edition in the library's catalog.
        - The client sends the entire updated data representation to the server.

    - **PATCH Method: Partial Updates**
        - The PATCH method updates data only if it exists and does so partially. It's like adding a new chapter to a book or modifying a section without rewriting the whole content.

        - Here, the client sends specific changes to be applied to the resource, rather than the entire updated data.

- **DELETE: Removing Data**
    - The DELETE method removes data from the database, similar to removing a book permanently from the library's shelves.

HTTP Verbs in Database Operations: Node.js Implementation

These HTTP verbs or methods are fundamental when working with databases via Node.js. They represent the various actions you can perform, such as retrieving, creating, updating, and deleting data. Incorporating these verbs in your Node.js application allows seamless interaction with the database, executing specific operations based on the HTTP request received.
By using these methods judiciously, you can effectively manage and manipulate data within your Node.js application, ensuring accurate and efficient database interactions.

5.4 Working with databases using NodeJS: SQL query and MySQL Database

- **SQL (Structured Query Language)**
    - SQL serves as the universal language for interacting with databases, including MySQL. It offers a standardized approach to perform various operations within a database.

- **SQL Statements: Managing Database Actions**
    - **CREATE**: This SQL statement creates a new table within your database. It's like setting up the blueprint for a new section in your library to categorize books.

    - **INSERT**: When you use INSERT, you're adding new data into a table within your database. Think of it as placing a new book onto a shelf in your library.

    - **SELECT**: SELECT retrieves data from a table within your database. It's similar to browsing the library catalog to find specific books based on certain criteria.

    - **UPDATE**: UPDATE alters existing data in a table within your database. It's like correcting the details of a book's entry in the library catalog.

    - **DELETE**: DELETE removes data from a table within your database. It's akin to removing a book permanently from the library's collection.

- **MySQL Database: Features and Installation**

    - **MySQL Overview**: MySQL is a widely used Relational Database Management System (RDBMS). It's renowned for being free, open-source, and suitable for applications of various scales, ranging from small projects to large enterprise-level systems.

    - **Installing MAMP for MySQL Access**: MAMP is a toolset that includes MySQL, Apache, and PHP.

        - It provides an easy way to set up and manage a local server environment, allowing you to develop and test your applications locally before deploying them.

        - You can **access MySQL through MAMP by installing** it via the link provided:

            - **MAMP for Windows**
            - **MAMP for Mac**

Node.js and MySQL: Powering Database Operations

With Node.js, you can leverage SQL queries to interact with a MySQL database seamlessly. These SQL statements enable the execution of specific actions like creating tables, inserting data, retrieving information, updating existing records, and deleting entries. By integrating Node.js with MySQL, you gain a powerful toolset to manage your database efficiently within your applications.

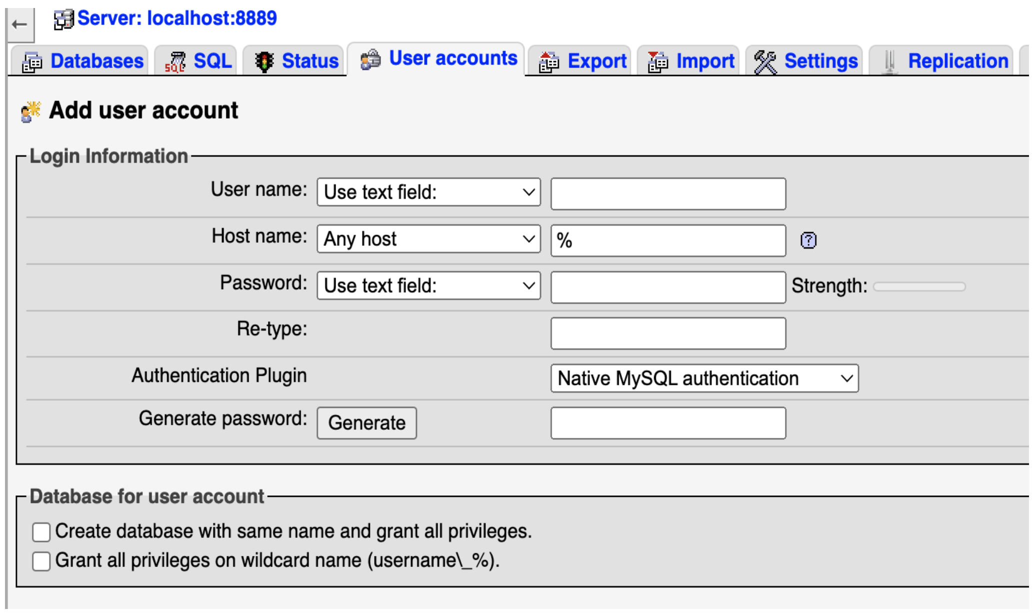
5.5 Working with databases using NodeJS: MySQL driver and connecting with MySQL database using Node

- **MySQL Driver: Bridging MySQL with Node.js**
  - A MySQL driver serves as a crucial intermediary, allowing seamless communication between a MySQL database and a Node.js application. Typically, developers rely on third-party modules installed via npm to establish this connection.

  - **npm Module for MySQL: npm i mysql2**

- **Connecting to MySQL Database from Node.js**

  - **1. MySQL Configuration:**
    - In MySQL using phpMyAdmin, first create a database and set up user credentials (username and password).



Add user account

Login Information

User name:	Use text field:		
Host name:	Any host	%	
Password:	Use text field:		Strength:
Re-type:			
Authentication Plugin		Native MySQL authentication	
Generate password:	Generate		

Database for user account

☐ Create database with same name and grant all privileges.
☐ Grant all privileges on wildcard name (username_%).

  - **2. Install and Import MySQL Module:**
    - Use “npm install mysql2” to install the MySQL driver.
    - Import the MySQL module within your Node.js application.

- **3. Create a Connection:**
  - Utilize the “createConnection” method provided by the MySQL module. Pass the database credentials (such as host, user, password, and database name) as an object.

- **4. Connect to the Database:**
  - Employ the “connect” method to establish the connection between your Node.js application and the MySQL database.

```
const express = require("express");
const app = express();

// STEP - 2 : Import mysql2 after installation
const mysql = require("mysql2");

// STEP - 3 : Pass Credentials to the createConnection method
const connection = mysql.createConnection(
  {
    host: "localhost",
    user: "Exact-Info-You-Used-When-Creating-The-User&DB",
    password: "Exact-Info-You-Used-When-Creating-The-User&DB",
    database: "Exact-Info-You-Used-When-Creating-The-User&DB",
  }
);

// STEP - 4 : Connect to MySQL DB
connection.connect((err) => {
    if (err) console.log(err);
    else console.log("Connected to MySQL");
});

app.listen(3001, () => console.log("listening on port 3001"));
```

Powering Node.js with MySQL: Connecting the Dots

The MySQL driver, installed via npm, acts as the crucial bridge between your Node.js application and the MySQL database. By following these steps, you set up a secure connection, allowing your Node.js scripts to interact with the MySQL database seamlessly. Once connected, you can perform various database operations using SQL queries within your Node.js codebase.

Note: Always ensure the secure handling of credentials and establish secure connections to prevent unauthorized access or potential security breaches.

5.6 Creating Tables in a Database: Using SQL's CREATE Statement

The "**CREATE**" statement in SQL is fundamental for establishing a new table within a database. It defines the table's structure, including its columns and their respective data types.

- **CREATE Statement Syntax:**
  - The syntax for creating a table using the "**CREATE TABLE**" statement involves specifying the table name, its columns, and their data types.
  - **Syntax:**

```sql
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ...
);
```

  - **Example:**
    - Suppose you're creating a table to store information about users in a database:

```
CREATE TABLE users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE
);
```

- **Explanation:**
  - “**CREATE TABLE**” initiates the creation process of a new table.
  - “**users**” is the name of the table being created.
  - Columns are defined within parentheses:
    - “**user_id**”: An integer column designated as the primary key and set to auto-increment for unique identification.
    - **“username**”: A column for storing usernames, restricted to 50 characters and cannot be empty (NOT NULL).
    - “**email**”: A column for storing email addresses, limited to 100 characters and ensuring uniqueness (UNIQUE).

Deploying the CREATE Statement

By utilizing the “**CREATE TABLE**” statement and adhering to the specified syntax, you can effectively craft new tables within your database using SQL. These tables serve as structured repositories for storing and organizing data, playing a pivotal role in managing information within your database.

Note: It's crucial to design tables with appropriate data types, constraints, and relationships based on the specific requirements of your application to ensure efficient data storage and retrieval.

5.7 Inserting Data into Tables using SQL's INSERT Statement

The “**INSERT**” SQL statement is crucial for adding new data entries into existing tables within a database.

- **INSERT Statement Syntax:**
    - To insert data into a table, you use the “**INSERT INTO**” statement, specifying the table name and the columns where you want to insert data, followed by the “**VALUES**” keyword with the actual values to be inserted.

    - **Syntax:**

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- **HTTP Method for Data Insertion in Node.js:**
    - In Node.js, the HTTP method typically used for data insertion is “**POST**”. This method is employed to send data to the server to create or update resources, making it suitable for inserting data into databases.

- **Enabling Data Insertion for the app:**
    - **1. Create a Frontend Interface:**
        - Design an HTML page featuring a form that interacts with the backend.
        - Use the “**<form>**” tag within the HTML page to collect user input.

    - **2. Form Attributes:**
        - Set the form's “**action**” attribute to the route in your Node.js backend responsible for handling data insertion (“**POST**” method).
        - Use the **method="post"** attribute to specify the HTTP method as **POST**.

    - **3. Retrieving Data in Node.js:**
        - Utilize middleware in your Node.js application to handle incoming form data (“**req.body**”).

- **4. Executing the SQL:**
    - Pass the incoming data received from the frontend with the help of "**req.body**" to the appropriate "**INSERT**" sql statement.

By creating this frontend interface and ensuring proper communication between the HTML form and the Node.js backend, our app users can input data seamlessly into the database.

5.8 Retrieving Data from Tables using SQL's SELECT Statement

The "**SELECT**" SQL statement is essential for fetching data from tables within a database.

- **SELECT Statement Syntax:**
    - To retrieve data, you use the "**SELECT**" statement followed by the columns you want to fetch from a specific table.
    - **Syntax:**

```sql
SELECT * FROM table_name;
```

  - The * signifies selecting all columns from the specified table.

- **HTTP Method for Data Retrieval in Node.js:**
    - In Node.js, the HTTP method primarily used for data retrieval is "**GET**". This method is employed to request data from a specified resource, making it suitable for fetching data from databases.

- **Selecting Primary Keys and Using Them as Foreign Keys in another table:**
    - If you are inserting data to multiple tables, you need to utilize primary keys and foreign keys, since that's how the tables are connected. Therefore, you

have to select the primary key for the inserted data and pass it to the other table, so it can be inserted as a foreign key.

- The following 3 steps explain how you can effectively retrieve the primary key value and use it as a foreign key in other tables, enabling relationships between different entities within your database.
- **1. Retrieve Primary Key:**
  - Use the "**SELECT * FROM <table_name>**" statement to retrieve data from the main table.
  - Target the primary key using the appropriate parameter (usually named **rows**) in your code to access the primary key value.
- **2. Assign Primary Key Value:**
  - Extract the primary key value from the retrieved data, often accessed through the first index of the returned array or object.
  - Store this primary key value in a variable within your Node.js application.
- **3. Insert as Foreign Key:**
  - Use this stored variable (the primary key value) to insert it as a foreign key into another table.
  - Remember that in MySQL, all values need to be treated as strings, so ensure the primary key value is surrounded by quotation marks to be interpreted as a string when inserting it into the other table.

5.9 Managing Database Operations in Node.js: SQL Queries for Update and Delete

- **UPDATE SQL Statement: Modifying Data**
  - The "**UPDATE**" statement in SQL is utilized to modify existing data within tables.
  - Use the "**UPDATE**" statement to set new values to specific columns based on a condition.
  - **Syntax:**

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- **HTTP Method for Modifying Data in Node.js:**
  - The HTTP method commonly associated with updating data in Node.js is "**PUT**".

- **DELETE SQL Statement: Removing Data**
  - The "**DELETE**" statement in SQL is used to eliminate specific data from a table.
  - DELETE removes records based on specified conditions.
  - **Syntax:**

```
DELETE FROM table_name WHERE condition;
```

- **HTTP Method for Deleting Data in Node.js:**
  - For removing data in Node.js, the corresponding HTTP method is "**DELETE**".

Summary

In exploring the intersection of Node.js and database operations through SQL queries, it's evident that these technologies harmonize to enable robust data management. From fundamental queries like SELECT and INSERT for retrieving and adding data, to more complex operations involving UPDATE and DELETE for modifying and removing records, Node.js seamlessly integrates with SQL to power these functionalities. Corresponding HTTP verbs in Node.js align with these operations, delineating clear pathways for **C**reate, **R**ead, **U**pdate, and **D**elete (**CRUD**) actions—**GET** for reading, **POST** for creating, **PUT** for updating, and **DELETE** for deleting data. Through the synergy of Node.js's backend capabilities and SQL's database manipulation prowess, a comprehensive toolkit emerges, empowering developers to build dynamic, responsive applications while efficiently managing data flows.