

Building RESTful API in Node.js

Building a RESTful API in Node.js involves creating endpoints that clients can use to perform CRUD (Create, Read, Update, Delete) operations on resources.

Steps involved in building a REST API in Node.js using Express

- **Set up a new Node.js project:** Start by creating a new directory for your project and initializing a new Node.js project using npm init.

- **Install Express:** Install Express and save it as a dependency in your project.

```
npm install express
```

- **Create the main application file:** Create a new file (e.g., App.js) and set up the basic Express application.

```
const express = require('express');  
const app = express();  
const PORT = 3000;
```

```
app.listen(PORT, () => {  
  console.log(`Server running on port ${PORT}`);  
});
```

- **Define routes:** Define routes for your API. Routes correspond to different endpoints that clients can access to perform operations on resources. For example, you can create a route to get all users:

```
app.get('/users', (req, res) => {  
  // Logic to fetch all users from the database  
  res.json({ message: 'Get all users' });  
});
```

- **Add middleware:** Use middleware to handle common tasks such as parsing JSON bodies and handling CORS (Cross-Origin Resource Sharing) requests.

```
app.use(express.json());  
app.use(cors());
```

- **Implement CRUD operations:** Implement CRUD operations for your resources. For example, to create a new user:

```
app.post('/users', (req, res) => {  
  const newUser = req.body;  
  // Logic to save the new user to the database  
  res.status(201).json(newUser);  
});
```

- **Use a router for modularization:** As your API grows, consider using Express's Router to modularize your routes and keep your code organized.

```
const usersRouter = require('./routes/users');
app.use('/users', usersRouter);
```
- **Error handling:** Implement error handling middleware to catch and handle errors that occur during request processing.

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Internal Server Error' });
});
```
- **Connect to a database:** If your API needs to interact with a database, use a database library like Mongoose (for MongoDB) or Sequelize (for SQL databases) to connect to your database and perform operations.
- **Testing your API:** Test your API using tools like Postman or curl to send requests to your endpoints and verify that they work as expected.
- **Deployment:** Deploy your API to a production environment. You can use platforms like Heroku, AWS, or Azure to host your Node.js application.
- **Documentation:** Document your API using tools like Swagger or API Blueprint to make it easier for developers to understand and use your API.

Building a RESTful API in Node.js involves designing clean and consistent APIs, implementing CRUD operations, handling errors, and ensuring security and scalability. With Express, you can quickly build robust APIs that meet the needs of your application.

Example: RESTful API with CRUD (Create, Read, Update, Delete) operations for a user resource

App.js

```
const express = require('express');
const app = express();
const PORT = 3000;

// Middleware to parse JSON bodies
app.use(express.json());

// Sample data
let users = [
  { id: 1, name: 'shivam' },
  { id: 2, name: 'manisha' },
  { id: 3, name: 'neeraj' }
];

// GET /users - Get all users
app.get('/users', (req, res) => {
  res.json(users);
});


// GET /users/:id - Get user by ID
app.get('/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ error: 'User not found' });
  res.json(user);
});

// POST /users - Create a new user
app.post('/users', (req, res) => {
  const newUser = { id: users.length + 1, name: req.body.name };
  users.push(newUser);
  res.status(201).json(newUser);
});
```

```
// PUT /users/:id - Update user by ID
app.put('/users/:id', (req, res) => {
  const userIndex = users.findIndex(u => u.id === parseInt(req.params.id));
  if (userIndex === -1) return res.status(404).json({ error: 'User not found' });
  users[userIndex].name = req.body.name;
  res.json(users[userIndex]);
});

// DELETE /users/:id - Delete user by ID
app.delete('/users/:id', (req, res) => {
  const userIndex = users.findIndex(u => u.id === parseInt(req.params.id));
  if (userIndex === -1) return res.status(404).json({ error: 'User not found' });
  const deletedUser = users.splice(userIndex, 1);
  res.json(deletedUser);
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

The logo for 'Learn 2 Earn Labs' is centered on the page. It features the word 'LEARN' in blue, a large red number '2' inside a hexagon, the word 'EARN' in blue, and the word 'LABS' in grey below them. A horizontal line with circles at both ends passes behind the '2'.

Connecting Node.js with a database

- Connecting Node.js with a database involves establishing communication between a Node.js application and a database management system (DBMS) to perform CRUD operations (Create, Read, Update, Delete) on the data stored in the database.
- Node.js, being a server-side JavaScript runtime, provides an efficient environment for building web applications, APIs, and other server-side tasks.

To interact with a database from a Node.js application, you need to follow the following steps:

- **Choose a Database:** There are various types of databases available, including relational databases like MySQL, PostgreSQL, SQLite, non-relational or NoSQL databases like MongoDB, CouchDB, and others. Choose a database that best fits the requirements of your application.
- **Install Database Driver/Module:** Node.js does not come with built-in support for specific databases. You need to install a database driver or module that allows Node.js to interact with your chosen database. These drivers/modules are available through Node Package Manager (npm). For example, mysql for MySQL, pg for PostgreSQL, mongodb for MongoDB.
- **Configure Database Connection:** You need to create a connection to your database by providing connection details such as hostname, port, username, password, and database name. This information varies depending on the type of database you're using.
- **Write Queries:** Once the connection is established, you can write SQL queries (for relational databases) or use an object-oriented approach (for NoSQL databases) to perform CRUD operations. Node.js provides various methods and libraries to execute queries against the database.
- **Handle Errors and Close Connection:** It's crucial to handle errors that may occur during the database operations. Also, remember to close the database connection when it's no longer needed to prevent resource leaks.

Sample Example:

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

// Create a connection
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'username',
  password: 'password',
  database: 'mydatabase'
});

// Connect to the database
connection.connect((err) => {
  if (err) {
    console.error('Error connecting to database:', err);
    return;
  }
  console.log('Connected to the database');
});

// Perform database operations
connection.query('SELECT * FROM mytable', (err, results) => {
  if (err) {
    console.error('Error executing query:', err);
    return;
  }
  console.log('Query results:', results);
});

// Close the connection
connection.end();
```

You can go through with above sample example to illustrate the process. In real-world applications, you may use connection pooling, ORM (Object-Relational Mapping) libraries, or other advanced techniques to manage database interactions efficiently and securely.

Importance of Connecting Node.js with a Database

- **Data Persistence:** In most web applications, there's a need to store and retrieve data persistently. Databases provide a structured way to store and manage data.
- **Dynamic Content Generation:** Node.js allows for dynamic content generation on the server-side. By connecting with a database, you can fetch and manipulate data to generate dynamic responses.
- **User Authentication and Authorization:** Databases play a crucial role in storing user credentials and session data for authentication and authorization purposes.
- **Data Analysis and Reporting:** Data stored in databases can be analyzed and used to generate reports and insights, which are vital for decision-making in businesses and organizations.

Node.js Database Drivers

- **Drivers/Modules:** Node.js doesn't include built-in support for specific databases. Instead, you rely on third-party drivers or modules to interact with databases.
- **npm:** Node Package Manager (npm) hosts a vast ecosystem of modules for various databases. You can install these modules using npm and include them in your Node.js project.
- **Examples:** Popular database drivers/modules include mysql for MySQL, pg for PostgreSQL, mongodb for MongoDB, mongoose for MongoDB with an ORM layer, and many more.

Connection Configuration

- **Connection Details:** To establish a connection, you need to provide details such as hostname, port, username, password, and database name.
- **Security Considerations:** It's crucial to follow security best practices, such as using environment variables to store sensitive information like database credentials, and using techniques like connection pooling to manage database connections securely.
- **Connection Pooling:** Connection pooling helps manage multiple connections to the database efficiently, reducing overhead and improving performance.

Query Execution

- **SQL vs. NoSQL:** Depending on the type of database you're using (SQL or NoSQL), the query execution approach varies. SQL databases require SQL queries, while NoSQL databases may use object-oriented methods or specialized query languages.
- **CRUD Operations:** You perform CRUD operations (Create, Read, Update, Delete) to interact with the database. Node.js provides methods to execute these operations against the database.
- **Handling Asynchronous Operations:** Database operations in Node.js are typically asynchronous. You need to handle callbacks, promises, or use async/await syntax to manage asynchronous code effectively.

Error Handling and Connection Management

- **Error Handling:** Proper error handling is essential to handle potential errors that may occur during database operations, such as network issues, query failures, or authentication errors.
- **Connection Management:** Always close the database connection when it's no longer needed to prevent resource leaks. You can use built-in methods to close connections or rely on libraries that provide connection pooling and automatic connection management.

LABS

Example : Connect nodejs with MySQL database

First, use or install a Compatible Client library

`npm install mysql2`

Now use the following code to connect nodejs with MySQL

App.js

```
// Import the mysql module
const mysql = require('mysql2');

// Create a connection to the MySQL database
const connection = mysql.createConnection({
  host: 'localhost',
  port: 3306,
  user: 'root',
  password: 'android',
  database: 'mernstack'
});

// Connect to the database
connection.connect((err) => {
  if (err) {
    console.error('Error connecting to MySQL database: ' + err.stack);
    return;
  }
  console.log('Connected to MySQL database as ID ' + connection.threadId);
});
```

mysql2 over the mysql library

- **Performance:** mysql2 is known for its better performance compared to mysql. It achieves this by providing support for promises, streams, and a better handling of connection pooling. mysql2 also has better support for prepared statements, which can improve performance in certain scenarios.
- **Streaming Support:** mysql2 provides native support for streaming query results, which allows you to handle large datasets more efficiently. This is particularly useful when dealing with large amounts of data, as it reduces memory usage and improves application performance.
- **Promise Support:** While both libraries support callbacks, mysql2 also provides support for promises out of the box. This makes it easier to work with modern JavaScript async/await syntax and allows for cleaner and more readable code, especially in asynchronous contexts.
- **Improved Prepared Statements:** mysql2 offers better support for prepared statements, allowing you to execute parameterized queries more efficiently. This can help prevent SQL injection attacks and improve overall security.
- **Active Development:** mysql2 is actively maintained and developed, with new features and improvements being added regularly. This ensures that you have access to the latest advancements and bug fixes in the library.
- **Compatibility:** While mysql2 is compatible with most applications using the mysql library, it also provides additional features and enhancements that can benefit your application.

Hence you can understand that, mysql2 is generally considered a better choice for new Node.js projects or projects where performance and modern JavaScript features are a priority. However, the choice between mysql and mysql2 ultimately depends on your specific requirements and preferences.

Example : creating table in mysql database using nodejs

App.js

```
// Import the mysql module
const mysql = require('mysql2');

// Create a connection to the MySQL database
const connection = mysql.createConnection({
  host: 'localhost', // Remove the port number from here
  port: 3306, // Add port here
  user: 'root',
  password: 'android',
  database: 'mernstack'
});

// Connect to the database
connection.connect((err) => {
  if (err) {
    console.error('Error connecting to database:', err);
    return;
  }
  console.log('Connected to the database');
  // Create the student table
  const createTableQuery = `CREATE TABLE IF NOT EXISTS students (id INT
  AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, city
  VARCHAR(255) NOT NULL )`;
  connection.query(createTableQuery, (err, result) => {
    if (err) {
      console.error('Error creating student table:', err);
      return;
    }
    console.log('Student table created successfully');
    // Close the connection after executing the query
    connection.end();
  });
});
```

Example : Inserting values/records into mysql table using nodejs

App.js

```
// Import the mysql module
const mysql = require('mysql2');

// Create a connection to the MySQL database
const connection = mysql.createConnection({
  host: 'localhost', // Remove the port number from here
  port: 3306, // Add port here
  user: 'root',
  password: 'android',
  database: 'mernstack'
});

// Connect to the database
connection.connect((err) => {
  if (err) {
    console.error('Error connecting to database:', err);
    return;
  }
  console.log('Connected to the database');
  // Insert values into the student table
  const insertQuery = `INSERT INTO students (name, city) VALUES ('Nisha Raghav',
'Ghaziabad'), ('Prashant Sharma', 'Jaipur'), ('Tulika Gupta', 'Gurgaon')`;
  connection.query(insertQuery, (err, result) => {
    if (err) {
      console.error('Error inserting values into student table:', err);
      return;
    }
    console.log('Values inserted into student table successfully');

    // Close the connection after executing the query
    connection.end();
  });
});
```

Example : Updating values/records into mysql table using nodejs

App.js

```
// Import the mysql module
const mysql = require('mysql2');

// Create a connection to the MySQL database
const connection = mysql.createConnection({
  host: 'localhost', // Remove the port number from here
  port: 3306, // Add port here
  user: 'root',
  password: 'android',
  database: 'mernstack'
});

// Connect to the database
connection.connect((err) => {
  if (err) {
    console.error('Error connecting to database:', err);
    return;
  }
  console.log('Connected to the database');

  // Update values in the student table
  const updateQuery = `UPDATE students SET city = 'Mumbai' WHERE name = 'Prashant Sharma'`;

  connection.query(updateQuery, (err, result) => {
    if (err) {
      console.error('Error updating values in student table:', err);
      return;
    }
    console.log('Values updated in student table successfully');

    // Close the connection after executing the query
    connection.end();
  });
});
```

Example : Delete values/records into mysql table using nodejs

App.js

```
// Import the mysql module
const mysql = require('mysql2');

// Create a connection to the MySQL database
const connection = mysql.createConnection({
  host: 'localhost', // Remove the port number from here
  port: 3306, // Add port here
  user: 'root',
  password: 'android',
  database: 'mernstack'
});

// Connect to the database
connection.connect((err) => {
  if (err) {
    console.error('Error connecting to database:', err);
    return;
  }
  console.log('Connected to the database');

  // Delete a record from the student table
  const deleteQuery = `DELETE FROM students WHERE name = 'Prashant Sharma';

  connection.query(deleteQuery, (err, result) => {
    if (err) {
      console.error('Error deleting record from student table:', err);
      return;
    }
    console.log('Record deleted from student table successfully');

    // Close the connection after executing the query
    connection.end();
  });
});
```