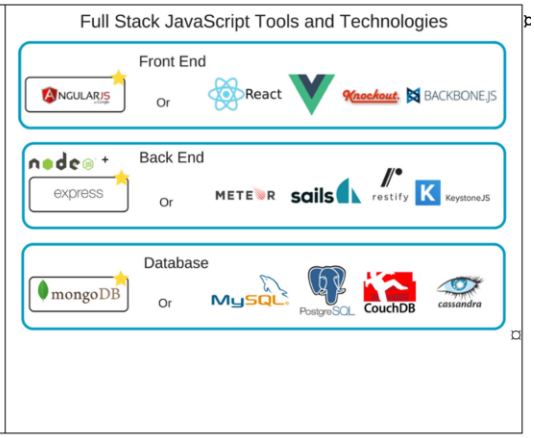
Setup & Run

1. npm install mssql
2. Node.js
3. Node Package Manager (NPM)
4. REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop
5. - To launch the REPL (Node shell), open command prompt (in Windows) or terminal (in Mac or UNIX/Linux) and type node as shown below. It will change the prompt to >
6. https://learn.microsoft.com/en-us/sql/connect/node-js/step-3-proof-of-concept-connecting-to-sql-using-node-js?view=sql-server-ver17
7.
8. https://www.tutorialsteacher.com/nodejs/access-sql-server-in-nodejs
9. **First node.js application with Visual Studio 2017**

Node.js is a server-side runtime environment for web development which is used to run JavaScript code built on Google's Chrome V8 engine with these features. 1. → Open source and cross-platform application. 2. → Asynchronous. It means request executes parallel rather than in queue. 3. → Its input/output operations are non-blocking. 4. → Event driven -- all tasks are performed accordance of the event. 5. → It creates HTTP server similar to SignalR Server.	

```
C:\> node server.js
Node.js web server at port 5000 is running..
```

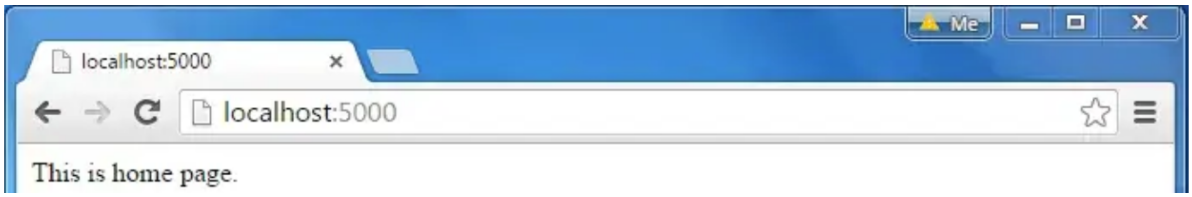
To test it, you can use the command-line program curl, which most Mac and Linux machines have pre-installed.

```
curl -i http://localhost:5000
```

You should see the following response.

```
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Tue, 8 Sep 2015 03:05:08 GMT
Connection: keep-alive
This is home page.
```

For Windows users, point your browser to http://localhost:5000 and see the following result.



- **Node.js**:

This is a JavaScript runtime environment that allows you to execute JavaScript code outside of a web browser. It's built on Chrome's V8 JavaScript engine and enables server-side programming with JavaScript, providing features like file system access, networking, and process management.

- **Express.js**:

This is a minimal and flexible web application framework built on top of Node.js. It simplifies the process of building web applications by providing a robust set of features for:

- **Routing**: Defining how the application responds to client requests to specific endpoints (URLs) and HTTP methods (GET, POST, PUT, DELETE).

- **Middleware**: Functions that have access to the request and response objects, and can modify them or terminate the request-response cycle. This allows for modular and reusable code for tasks like authentication, logging, and error handling.
- **HTTP utility methods**: Simplifying common HTTP operations.
- **Integrating with view rendering engines**: Generating dynamic HTML responses by inserting data into templates.

In essence, Node.js provides the foundation for running JavaScript on the server, while Express.js offers a structured and efficient way to build web applications and APIs on that foundation. Developers often use them together to create scalable and performant back-end systems.

A Quick Start

To build an easy Node.js API with Express.js in Visual Studio Code, following the guidance from Microsoft Learn resources:

- **Install Node.js and Visual Studio Code**: Ensure both Node.js (which includes npm) and Visual Studio Code are installed on your system.
- **Create a New Project Folder**: Open your preferred command line (e.g., Command Prompt, PowerShell, or VS Code's integrated terminal) and create a new directory for your project:

```
mkdir MyNodeApi
cd MyNodeApi
```

- **Initialize a Node.js Project**: Initialize a new Node.js project within your folder using npm:

```
npm init -y
```

This creates a package.json file.

- **Install Express.js**: Install the Express.js framework as a dependency:

```
npm install express
```

- **Create Your API File**: In Visual Studio Code, create a new file (e.g., app.js) in your project folder.
- **Write the Basic Express API**: Add the following code to app.js to create a simple API that responds to a GET request:

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

app.get('/', (req, res) => {
  res.send('Hello from your Node.js API!');
});

app.listen(port, () => {
  console.log(`API listening at http://localhost:${port}`);
});
```

- **Run the API**: Open the integrated terminal in Visual Studio Code (Terminal > New Terminal) and run your application:

```
node app.js
```

You should see the message "API listening at http://localhost:3000".

- **Test the API**: Open your web browser and navigate to http://localhost:3000. You should see the message "Hello from your Node.js API!".

This sets up a basic Node.js API using Express.js within Visual Studio Code, providing a foundation for further development.

Node Chat Example

A real-time chat application using Node.js typically involves the following components and steps:

1. Project Setup:

- **Initialize Node.js project:** Create a project directory and initialize it using npm init -y.
- **Install dependencies:** Install express for the web server and socket.io for real-time communication: npm install express socket.io.

2. Server-side (Node.js with Express and Socket.IO):

- Create index.js (or app.js): This file will contain the server-side logic.
- **Set up Express server:**

```
const express = require('express');
const app = express();
const http = require('http').createServer(app);
const io = require('socket.io')(http); // Initialize Socket.IO with the HTTP server

// Serve static files (e.g., index.html)
app.use(express.static(__dirname + '/public')); // Assuming public folder for client-side files

// Handle root route
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
});

// Start the server
const PORT = process.env.PORT || 3000;
http.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Handle Socket.IO connections.

```
io.on('connection', (socket) => {
  console.log('A user connected');

  // Listen for chat messages from clients
  socket.on('chat message', (msg) => {
    console.log('message: ' + msg);
    io.emit('chat message', msg); // Broadcast the message to all connected clients
  });

  // Handle disconnects
  socket.on('disconnect', () => {
    console.log('User disconnected');
  });
});
```

3. Client-side (HTML and JavaScript):

- Create public/index.html: This file will contain the front-end structure.

```
<!DOCTYPE html>
<html>
<head>
  <title>Node.js Chat</title>
  <style>
    /* Basic styling */
    #messages { list-style-type: none; margin: 0; padding: 0; }
    #messages li { padding: 5px 10px; }
    #messages li:nth-child(odd) { background: #eee; }
  </style>
</head>
<body>
  <ul id="messages"></ul>
  <form id="form" action="">
    <input id="input" autocomplete="off" /><button>Send</button>
  </form>

  <script src="/socket.io/socket.io.js"></script>
  <script>
    var socket = io(); // Connect to the Socket.IO server

    var form = document.getElementById('form');
    var input = document.getElementById('input');
    var messages = document.getElementById('messages');

    form.addEventListener('submit', function(e) {
      e.preventDefault();
      if (input.value) {
        socket.emit('chat message', input.value); // Send message to server
        input.value = '';
      }
    });

    socket.on('chat message', function(msg) {
      var item = document.createElement('li');
      item.textContent = msg;
      messages.appendChild(item);
      window.scrollTo(0, document.body.scrollHeight);
    });
  </script>
</body>
</html>
```

4. Running the Application:

- Start the server from your terminal in the project directory: node index.js.
- Open http://localhost:3000 (or your chosen port) in multiple browser tabs to simulate different chat users.

This example provides a basic real-time chat application where messages sent by one user are broadcast to all other connected users. More advanced features like user management, private messaging, and chat rooms can be implemented by extending this foundation with additional Socket.IO events and server-side logic.

End chat example

Performing CRUD (Create, Read, Update, Delete) operations in Node.js with Microsoft SQL Server typically involves the following steps:

- **Project Setup and Dependencies:**
  - Initialize a new Node.js project using npm init -y.
  - Install the mssql package, which is the official Node.js driver for SQL Server: npm install mssql.
  - Consider installing express for building a REST API and dotenv for managing environment variables (like database credentials).
- **SQL Server Configuration and Connection:**
  - Ensure SQL Server is running and accessible (e.g., TCP/IP enabled, port 1433 configured).
  - Obtain the necessary connection details: server name/host, database name, username, and password.
  - Create a configuration object in your Node.js application to store these details.
  - Establish a connection to the SQL Server database using mssql.connect(). It is recommended to use a connection pool for efficient resource management.
- **CRUD Operations Implementation:**
  - **Create (Insert):** Construct SQL INSERT statements or call stored procedures to add new records to tables. Use parameterized queries to prevent SQL injection vulnerabilities.
  - **Read (Select):** Write SQL SELECT queries to retrieve data from tables. You can fetch all records or filter by specific criteria.
  - **Update:** Utilize SQL UPDATE statements or stored procedures to modify existing records.

- **Delete:** Use SQL DELETE statements or stored procedures to remove records based on specified conditions.

- **API Endpoints (Optional but Recommended):**
  - If building a web application, use a framework like Express.js to create API endpoints (e.g., /api/items, /api/items/:id) that handle incoming requests for CRUD operations.
  - Link these endpoints to functions that execute the corresponding SQL operations using the mssql library.
- **Error Handling and Asynchronous Operations:**
  - Implement robust error handling for database operations, including connection errors, query execution errors, and data validation issues.
  - Utilize async/await for managing asynchronous database interactions, ensuring cleaner and more readable code.

Example Snippet (Conceptual - for illustration):

```
const sql = require('mssql');

const config = {
   user: 'your_username',
   password: 'your_password',
   server: 'your_server_name', // You can use 'localhost' or a specific IP
   database: 'your_database_name',
   options: {
      encrypt: true, // For Azure SQL Database or if using SSL
      trustServerCertificate: true // Change to false for production
   }
};

async function createRecord(data) {
   try {
      let pool = await sql.connect(config);
      let result = await pool.request()
         .input('name', sql.NVarChar, data.name)
         .input('value', sql.Int, data.value)
         .query('INSERT INTO YourTable (Name, Value) VALUES (@name, @value)');
      console.log('Record created:', result);
   } catch (err) {
      console.error('Error creating record:', err);
   } finally {
```

```
        sql.close();
    }
}

async function readRecords() {
    try {
        let pool = await sql.connect(config);
        let result = await pool.request().query('SELECT * FROM YourTable');
        console.log('Records:', result.recordset);
        return result.recordset;
    } catch (err) {
        console.error('Error reading records:', err);
    } finally {
        sql.close();
    }
}
```

End lab

Reference

1. Node.js Express.js
2. Express - Node.js web application framework
3. Node.js tutorial in Visual Studio Code
4. Get started with web development using Visual Studio Code - Training | Microsoft Learn
5. Connect to and query using Node.js and mssql npm package - Azure SQL Database | Microsoft Learn
6. endlist