Node.js

What is Node.js?

- Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine.
- It allows you to run JavaScript code on the server-side.

Key Concepts:

- **Non-blocking I/O:**
  - Node.js is designed to handle asynchronous operations efficiently.
  - It uses callbacks, promises, and async/await for handling async tasks.
- **Event Loop:**
  - Node.js relies on an event-driven, single-threaded architecture.
  - It processes events and callbacks in a loop, making it highly scalable.
- **Modules:**
  - Node.js uses a CommonJS module system (require/exports) to organize code.
  - You can create and use your own modules.

Core Modules:

- **fs (File System):**
  - Used for file I/O operations like reading and writing files.
- **http/https:**
  - For creating HTTP/HTTPS servers and handling web requests.
- **path:**
  - Helps with file and directory path manipulation.

NPM (Node Package Manager):

- **NPM:** package manager for Node.js.
  - Used to install and manage third-party libraries and packages.

Express.js:

- A popular web application framework for Node.js.
- Simplifies the creation of APIs and web applications.

RESTful APIs:

- **Create RESTful APIs using Express.js:**
  1. Set Up Your Node.js Project
  2. Install Express.js `npm install express --save`
  3. Create an Express App (e.g. 'app.js')

```
const express = require('express');
const app = express();
const port = process.env.PORT || 3000; // Choose your desired port

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

  4. Define Routes

```
// GET request to retrieve all items
app.get('/api/items', (req, res) => {
  // logic to retrieve & send all items as JSON response
});

// GET request to retrieve a specific item by ID
app.get('/api/items/:id', (req, res) => {
  // logic to retrieve & send a specific item by ID as JSON response
});

// POST request to create a new item
app.post('/api/items', (req, res) => {
  // logic to create a new item & send a JSON response with the created item
});

// PUT request to update an existing item by ID
app.put('/api/items/:id', (req, res) => {
  // logic to update existing item by ID & send a JSON response w updated item
});

// DELETE request to delete an item by ID
app.delete('/api/items/:id', (req, res) => {
  // logic to delete an item by ID & send a JSON response indicating success
});
```

  5. Middleware

```
// middleware function for JSON request parsing
app.use(express.json());

// custom middleware for logging requests
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

6. Implement CRUD Operations (database, in-memory array, or any other data source)
7. Start the Server

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

8. Test Your API (e.g., Postman)

- **HTTP methods (GET, POST, PUT, DELETE):**

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

// Define a route for a GET request
app.get('/', (req, res) => {
  res.send('Hello, World!');
});

// Define a route for a POST request
app.post('/api/users', (req, res) => {
  // Handle the POST request here
});

// Define a dynamic route with parameters for GET requests
app.get('/api/users/:id', (req, res) => {
  const userId = req.params.id;
  // Use userId to fetch user data and send a response
});

// Handle a PUT request
app.put('/api/users/:id', (req, res) => {
  const userId = req.params.id;
  // Handle the PUT request to update user data with the given userId
});

// Handle a DELETE request
app.delete('/api/users/:id', (req, res) => {
  const userId = req.params.id;
  // Handle the DELETE request to delete the user with the given userId
});

// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Middleware:

- Middleware functions in Express.js are used for **request/response processing**. Here are some examples:
  - **logging**

```
app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
});
```

  - **authentication**

```
// Custom authentication middleware
const authenticate = (req, res, next) => {
  if (req.isAuthenticated()) {
    // User is authenticated, allow access
    return next();
  }
  // User is not authenticated, redirect to login page or send an error
response
  res.status(401).json({ message: 'Authentication required' });
};

// Use the authentication middleware for a specific route
app.get('/secured', authenticate, (req, res) => {
  // This route is protected and only accessible to authenticated users
  res.send('Welcome to the secured route!');
});
```

  - **error handling middleware**

```
// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Something went wrong!' });
});
```

Asynchronous Programming:

- A programming paradigm (set of concepts, practices, and theories) that lets your program do multiple things at once, so it doesn't get stuck waiting for one task to finish before moving on to the next one. Here are some examples:
    - **callback:** A function passed as an argument, used for asynchronous actions

```
function fetchData(callback) {
  setTimeout(() => {
    const data = 'Some data';
    callback(data);
  }, 1000);
}

fetchData((data) => {
  console.log(data);
});
```

    - **promise:** A way to handle asynchronous operations with success or failure

```
// Simulates an asynchronous operation with a Promise.
// Resolves with 'Some data' on success, or rejects w error message on failure
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = 'Some data';
      resolve(data); // Operation successful
      // or reject('Error occurred'); // Operation failed
    }, 1000);
  });
}

// handling errors with promise
fetchData()
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.error(error);
  });
```

- **async/await:** Simplifies asynchronous code with keywords "async" and "await"

```
// handling errors with async/await
async function fetchData() {
  try {
    const data = await fetchDataPromise();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

// Simulates an asynchronous operation with a Promise.
// Resolves with 'Some data' on success, or rejects w error message on failure
function fetchDataPromise() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = 'Some data';
      resolve(data); // Operation successful
      // or reject('Error occurred'); // Operation failed
    }, 1000);
  });
}

fetchData();
```

Package.json:

- Holds information about the project (name, version, etc.), lists the components (dependencies), and provides instructions (scripts) for building and running the project.
- It is essential for collaboration and managing project details.

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "description": "A simple Node.js application",
  "main": "app.js",
  "scripts": {
    "start": "node app.js",
    "test": "mocha"
  },
  "dependencies": {
    "express": "^4.17.1",
    "axios": "^0.21.1"
  },
  "devDependencies": {
    "mocha": "^8.4.0"
  },
  "author": "Max",
  "license": "DTWebSolutions"
}
```

Debugging:

- Process of identifying and fixing issues or errors in your code
- Fun Fact: In JavaScript and Node.js, you can use built-in debugging tools or third-party solutions (e.g., debug, node-inspect)
- Types of debugging in Node.js:
    - **Built-in Debugging:**
        - **console.log:** Print variable values, object properties, and messages to the console to understand the flow of your code

```
console.log('This is a debug message');
console.log('Variable value:', someVariable);
```

        - **Debugger Statements:** Triggers the built-in Node.js debugger, allowing you to inspect variables, set breakpoints, and step through your code

```
function someFunction() {
  const value = 42;
  debugger; // trigger debugger here
  // rest of the code
}
```

          This debugger allows you to:
            1. Inspect variable values, like 'value'.
            2. View the call stack to see how you got here.
            3. Interactively evaluate expressions in the debugger console.
            4. Set additional breakpoints for detailed inspection.
            5. Step through the code to understand its flow.

        - **Node.js Inpector:** Built-in inspector for debugging by running your application with the `--inspect` or `--inspect-brk` flag and using a tool (e.g., Google Chrome DevTools) to connect to it

    - **Third-Party Debugging Tools:**
        1. Install **node-inspect**: `npm install -g node-inspect`
        2. Run Your Node.js Application with **node-inspect**: `node-inspect my-node-app.js`
        3. Interact with the Debugger:
            a. Open the provided URL in a web browser (commonly http://127.0.0.1:9229/)
            b. There will be a graphical debugging interface that allows you to set breakpoints, inspect variables, evaluate expressions, and step through your code
        4. Debug Your Application: `node-inspect app.js`

Testing:

- **Mocha:**
  - Flexible and widely-used JavaScript testing framework
  - Provides a test runner that lets you define and run tests
  - Works well with various assertion libraries (e.g., Chai)
  - Supports asynchronous testing
  - Allows you to structure your tests using **BDD (Behavior-Driven Development)** or **TDD (Test-Driven Development)** styles
- Example of a unit test in Mocha:

```
const assert = require('assert');
const myFunction = require('./myFunction'); // Import the function to test

describe('myFunction', () => {
  it('should return 3 when given 1 and 2', () => {
    assert.strictEqual(myFunction(1, 2), 3);
  });
});
```

- **Jest:**
  - All-in-one JavaScript testing framework
  - Built on top of Jasmine (common JavaScript and TypeScript testing framework)
  - Widely used in the React and JavaScript communities
  - Includes its own assertion library
  - Provides a built-in mocking system
  - Specifically designed for ease of use
  - Particularly useful for testing React components
- Writing unit tests in Jest:

```
const myFunction = require('./myFunction'); // Import the function to test

test('myFunction adds 1 + 2 to equal 3', () => {
  expect(myFunction(1, 2)).toBe(3);
});
```

- **Behavior-Driven Development (BDD):**
  - Approach to software development that focuses on how a program should behave from the user's perspective
  - Emphasizes clear, human-readable descriptions of features and expected outcomes
  - Encourages collaboration between developers, testers, and non-technical stakeholders
  - Common BDD tools include "Given-When-Then" scenarios to define behavior

- **Test-Driven Development (TDD):**
  - Development approach where tests are written before the actual code
  - Starts with creating a test that defines the expected behavior of a feature
  - Developers then write code to make the test pass and ensure the feature works correctly
  - Helps catch and prevent bugs early in the development process

Security:

- It is essential to protect your applications and user data from various threats and vulnerabilities
- Here are common examples:
    - **Injection Attacks:**
        - **SQL Injection:** Attacker manipulates database queries through user input
        - **NoSQL Injection:** Similar to SQL Injection, but for NoSQL databases
        - **Command Injection:** Unauthorized commands run through user inputs
    - **Cross-Site Scripting (XSS):** Occur when user inputs are not properly sanitized, allowing malicious scripts to be executed in users' browsers
    - **Cross-Site Request Forgery (CSRF):** attacks trick users into performing actions without their consent when they are logged into an application
    - **Cross-Origin Resource Sharing (CORS):** web application doesn't properly restrict which origins can access its resources, and it allows attackers to make unauthorized cross-origin requests, potentially leading to data exposure or unauthorized actions
- **Helmet.js:**
    - Node.js library that helps secure your web applications by setting various HTTP headers to protect against common vulnerabilities

Deployment:

- Making your application accessible on the internet
- Deployment Options (Description & Free Tier Details):
    - **Heroku:**
        - **Description:** A Platform-as-a-Service (PaaS) provider that simplifies deployment, scaling, and management of web applications
        - **Free Tier:** Named “Hobby” which includes 550 hours of runtime per month for one dyno (container) and free access to Heroku Postgres with 10,000 rows
    - **Netlify:**
        - **Description:** A modern web hosting and automation platform that specializes in hosting static websites and serverless functions
        - **Free Tier:** Has unlimited bandwidth, but includes limitations on the number of build minutes, concurrent builds, and serverless function execution
    - **Amazon Web Services:**
        - **Description:** A comprehensive cloud platform that provides various services for hosting, scaling, and managing applications
        - **Free Tier:** Has a limited number of free services for 12 months, including AWS Lambda, Amazon S3, and EC2. Each service may have specific usage limits.
    - **Microsoft Azure:**
        - **Description:** Microsoft's cloud platform offering a wide range of services for deploying and managing applications
        - **Free Tier:** Has specific usage limits on services like Azure Functions, Azure App Service, and Azure Storage. Some services are free for the first 12 months.
    - **Google Cloud:**
        - **Description:** Provides cloud computing, data storage, and machine learning services
        - **Free Tier:** Has specific usage limits for services like Google Cloud Functions, Google App Engine, and Google Cloud Storage
    - **Docker:**
        - **Description:** A containerization platform that allows you to package applications and their dependencies into containers
        - **Free Tier:** Docker itself is an open-source project, and using Docker containers is generally free. However, you may incur costs when hosting Docker containers in cloud services like AWS, Azure, or Google Cloud.

- **Process Manager:**
    - **Process Manager 2 (PM2):** A widely used process manager for Node.js, which provides features like process monitoring, automatic restarts, load balancing, and log management

    - **Forever:** A process manager for Node.js that ensures applications keep running by automatically restarting them in case of failures
    - **systemd:** On Linux systems, **systemd** can be used to manage Node.js processes as services, with process control, logging, and system-wide management
    - **Nodemon:** Even though it is a development tool, it can be used to monitor and restart Node.js applications during development and testing

Performance Optimization:

- The process of making your applications or websites run faster and use fewer resources
- Some key techniques for improving performance in Node.js applications:
    - **Minimizing I/O Operations:**
        - Limit database queries and file system operations
        - Batch and cache data when possible to reduce unnecessary I/O
    - **Asynchronous Programming:**
        - Utilize Node.js's non-blocking, asynchronous nature to handle multiple operations concurrently, enhancing responsiveness
    - **Caching:**
        - Implement caching mechanisms to store frequently accessed data in memory, reducing the need for repeated computations or database queries
    - **Load Balancing:**
        - Distribute incoming requests across multiple server instances to **evenly distribute the load** and **prevent overloading a single server**
    - **Compression:**
        - Compress responses before sending them to clients to reduce bandwidth usage and to improve load times
    - **Optimizing Code:**
        - Write efficient code, profile and analyze performance, and make code improvements where performance issues are identified
    - **Database Indexing:**
        - Properly index database tables to speed up data retrieval and query performance