Node.js and Express

1. What is Node.js?

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It allows you to run JavaScript code outside the browser, enabling developers to write server-side applications in JavaScript. Before Node.js, JavaScript was mostly used for front-end scripting in browsers, but Node.js extends its utility to the server side, allowing for building scalable, high-performance applications.

Key Features of Node.js:

- **Asynchronous and Non-blocking I/O**: Node.js is designed to handle asynchronous, event-driven programming. Non-blocking means Node.js can perform I/O operations (like reading files, querying databases, or making HTTP requests) without waiting for each to complete before moving on to the next.
- **Single-threaded Event Loop**: Despite being single-threaded, Node.js handles multiple connections concurrently using an event loop. This enables high concurrency and can efficiently handle a large number of simultaneous requests.
- **Built-in Libraries and NPM**: Node.js comes with a rich set of built-in libraries, and its package manager, **npm (Node Package Manager)**, allows developers to use thousands of open-source libraries to speed up development.
- **Scalability**: Node.js is often used in applications that require scalability, such as real-time applications, APIs, and microservices.

2. What is Express?

Express.js is a lightweight, minimal web application framework for Node.js. It provides a simple way to set up web servers, handle routing, manage HTTP requests, and streamline the creation of web applications and RESTful APIs.

Key Features of Express:

- **Routing**: Express provides a powerful routing system that maps HTTP requests (like GET, POST, PUT, DELETE) to specific functions that will handle those requests.
- **Middleware**: Middleware is a core concept in Express. It refers to functions that have access to the request, response, and the next middleware function in the application’s request-response cycle. Middleware can modify the request and response objects, or terminate the request-response cycle.
- **Simplified Setup**: Express simplifies the process of setting up a server, handling requests, and generating responses. It also simplifies error handling.
- **Template Engines**: Express supports various template engines like **EJS**, **Pug**, and **Handlebars**, which allow for dynamic rendering of HTML views.
- **Extensibility**: Express is minimal by design, and you can add more complex features like user authentication, session management, logging, etc., by integrating third-party middleware.

3. How Node.js and Express Work Together

In a typical Node.js and Express application, **Node.js** handles the basic HTTP server functionality, while **Express** provides a higher-level framework for managing routes, requests, and middleware.

1. **Setup**: You install Express in a Node.js project using npm. Once installed, you can create an Express app and configure it with routes and middleware.
2. **Request Handling**: When a request is sent to the server (e.g., when a user visits a webpage or an API endpoint), Express uses its routing mechanism to determine which handler function should respond to the request.
3. **Middleware Execution**: As the request passes through the middleware functions, each one can perform tasks like validating data, authenticating the user, or logging information about the request.
4. **Response**: Once the necessary processing is done, the final response is sent back to the client (typically in the form of HTML, JSON, or other data formats).

4. Example Code (Express with Node.js)

Here's a simple example of an Express application running on Node.js:

```
// Import the required modules
const express = require("express");
const app = express();
const port = 3000;

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

// Simple route that responds with a message
app.get("/", (req, res) => {
  res.send("Hello, World!");
});

// A route that handles POST requests
app.post("/data", (req, res) => {
  const receivedData = req.body;
  res.json({ message: "Data received", data: receivedData });
});

// A route with a dynamic parameter
app.get("/user/:id", (req, res) => {
  const userId = req.params.id;
  res.send(`User ID is ${userId}`);
});

// Start the server
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

5. Express Middleware and Routing

Middleware functions are essential in Express for processing incoming requests. Middleware functions have access to the request (`req`), response (`res`), and the next middleware in the pipeline.

For example:

```
// Example of logging middleware
app.use((req, res, next) => {
  console.log(`${req.method} request for ${req.url}`);
  next(); // Pass the request to the next middleware
});
```

- **Routing** defines which logic should run based on the HTTP method (GET, POST, etc.) and the URL path.

  ```
  app.get("/home", (req, res) => {
    res.send("Welcome to Home!");
  });
  ```

Middleware

In Node.js, specifically with Express, **middleware** refers to functions that have access to the request, response, and the next middleware function in the application’s request-response cycle. Middlewares are used for tasks such as handling HTTP requests, modifying the request/response objects, logging, authentication, error handling, etc.

Middleware functions can be used globally for all routes, or locally for specific routes. They are executed sequentially, and can either terminate the request-response cycle or pass control to the next middleware using the `next()` function.

Types of Middleware in Express:

1. **Application-level middleware**: This is defined at the application level and is executed for all routes or specific routes.
2. **Router-level middleware**: This is middleware defined for specific routers.
3. **Error-handling middleware**: This is used for handling errors, and it has four parameters (err, req, res, next).
4. **Built-in middleware**: Express provides built-in middlewares like `express.json()`, `express.urlencoded()`, etc.
5. **Third-party middleware**: Middleware developed by third parties, like `morgan` for logging requests.

Example 1: Simple Middleware Function

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

// Custom middleware
```

```
const myMiddleware = (req, res, next) => {
  console.log("Middleware executed!");
  next(); // Pass control to the next middleware
};

// Apply the middleware globally for all routes
app.use(myMiddleware);

// A route handler
app.get("/", (req, res) => {
  res.send("Hello World");
});

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

In this example:

- The `myMiddleware` function logs a message to the console every time a request is made.
- `app.use(myMiddleware)` makes this middleware apply to all incoming requests.
- The `next()` function tells Express to continue processing the request and move to the next middleware or route handler.

Example 2: Middleware for JSON Body Parsing

Express has built-in middleware for handling JSON data in request bodies, which can be used as follows:

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

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

// Route handler
app.post("/data", (req, res) => {
  console.log(req.body); // The parsed JSON data will be available here
  res.send("Data received");
});

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

Here:

- `express.json()` is a built-in middleware that parses incoming JSON requests and makes the parsed data available on `req.body`.

Example 3: Middleware for Logging Requests

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

// Third-party middleware for logging requests
const morgan = require("morgan");

// Use morgan for logging HTTP requests
app.use(morgan("dev"));

// A route handler
app.get("/", (req, res) => {
  res.send("Hello, Morgan!");
});

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

Here:

- `morgan('dev')` is a third-party logging middleware that logs HTTP requests in a concise format.
- Every request made to the server will be logged in the terminal.

Example 4: Route-level Middleware

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

// Middleware for a specific route
const authenticate = (req, res, next) => {
  if (req.headers["auth-token"] === "12345") {
    next(); // Authentication passed
  } else {
    res.status(401).send("Unauthorized");
  }
};

// Route with authentication middleware
app.get("/secure", authenticate, (req, res) => {
  res.send("This is a secure page");
});

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

Here:

- The `authenticate` middleware is applied only to the `/secure` route.

- If the request does not include the correct `auth-token` header, a `401 Unauthorized` error is sent.
