Introduction to Node.js and Express.js

What is Node.js?

Node.js is an open-source, cross-platform runtime environment that allows developers to run **JavaScript** on the server side. Traditionally, JavaScript was used only for client-side scripting (in browsers), but Node.js extends JavaScript's capability to the backend.

Key Features of Node.js:

1. **Built on V8 Engine**: Node.js uses Google Chrome's V8 JavaScript engine, making it fast and efficient.
2. **Non-blocking I/O**: Node.js uses an **event-driven, non-blocking I/O model**, which makes it lightweight and scalable.
3. **Single-threaded Event Loop**: Handles multiple requests efficiently using a single thread.
4. **Cross-platform**: Supports Windows, Linux, macOS, etc.
5. **NPM (Node Package Manager)**: Comes with Node.js, providing access to thousands of open-source libraries.

Common Use Cases of Node.js:

- Real-time applications (e.g., chat apps, gaming servers)
- REST APIs
- Data streaming applications
- IoT applications
- Server-side web applications

What Can Node.js Do?

Node.js can generate dynamic page content
Node.js can create, open, read, update, write, delete, and close files on the server
Node.js can collect form data
Node.js can add, delete, modify data in your database

REPL Feature in Node.js

It's a feature of Node.js

Read - Reads user’s input, parses the input into Javascript Data Structure and stores it in
the memory.
Eval - Takes and evaluates the data structure.
Print - Prints the results.
Loop - Loops the above command until the user presses ctrl twice.

REPL stands for Read-Eval-Print-Loop. It is a feature of Node.js that allows developers to execute JavaScript code and see the results in a command-line interface (CLI). The REPL
feature in Node.js provides an interactive shell where the developer can enter JavaScript
code and see the results immediately. It is similar to the JavaScript console in a web
browser, but it runs on the server side.

Using the REPL feature in Node.js, developers can quickly test and debug JavaScript code,
explore and experiment with Node.js modules and APIs, and run scripts and programs.
The REPL feature is also useful for learning the basics of Node.js and JavaScript.

To start the REPL feature in Node.js, you can simply open a command-line terminal and
type "node" followed by the enter key. This will open the REPL shell, where you can type
JavaScript code and see the results. The REPL shell also provides several useful
commands, such as the ability to load and run JavaScript files, and to control the input
and output.

What is NPM?

npm (short for Node Package Manager) is a package manager for the JavaScript programming language.
It is the default package manager for Node.js. npm makes it easy for JavaScript developers to share, reuse, and update code.

npm is a command-line tool that allows developers to install, update, and manage packages (collections of JavaScript code) in their projects. These packages can be third-
party libraries, frameworks, or tools, or they can be packages that the developer has
created themselves.
You can find list of packages on https://www.npmjs.com/
10 Packages for Node Js developers: https://www.turing.com/blog/top-npm-packages
for-node-js-developers/

Run a Nodejs file

To run a node js file every time you do any changes you need to run in the terminal as
"node index.js"

Nodemon is a **Node.js tool** that automatically **restarts your application** whenever it detects changes in your files. It is very useful during development as it eliminates the need to manually restart the server every time you make changes to the code.

Installing Nodemon

Instead of running node index.js every time, install the nodemon package
npm i nodemon (installs nodemon)

nodemon i --global (runs globally)

npm init -y (is used to **initialize a new Node.js project** by creating a `package.json` file in the current directory. The `-y` flag automatically fills the file with default values, saving time compared to the interactive `npm init`.

nodemon -v (checks the version)

Now while running node js run "nodemon index.js" only and from next time just ctrl+s will do the work.

1. Core Node.js Commands

Command	Purpose
`node <filename>`	Executes a Node.js script (e.g., `node app.js`).
`node`	Launches the interactive Node.js REPL (shell).
`node --version` or `node -v`	Displays the installed version of Node.js.
`node --watch <file>`	Automatically restarts when file changes are detected (Node 18+).
`node --check <file>`	Checks for syntax errors without running the code.

---

2. npm (Node Package Manager) Commands

Command	Purpose
`npm init`	Initializes a new project by creating a `package.json` file.
`npm install <package>` or `npm i`	Installs a package locally (e.g., `npm i express`).
`npm install -g <package>`	Installs a package globally. Example: `npm i -g nodemon`.
`npm uninstall <package>`	Removes an installed package.
`npm update`	Updates all locally installed packages to the latest version.
`npm list`	Lists all installed packages and their versions.

Command	Purpose
`npm audit`	Scans for vulnerabilities in the dependencies.
`npm run <script>`	Runs a custom script from `package.json` (e.g., `npm run dev`).
`npm cache clean --force`	Clears npm's cache to fix dependency issues.

---

3. Debugging Commands

Command	Purpose
`node --inspect <file>`	Enables debugging for the file (Chrome DevTools).
`node --inspect-brk <file>`	Breaks execution on the first line for debugging.
`console.log(<value>)`	Outputs information to the terminal for debugging.
`debugger`	Sets a breakpoint in the code.

Most Popular Commands for Development Workflow

1. **Start a project:**

   ```
   npm init
   ```

2. **Install dependencies:**

   ```
   npm install express
   ```

3. **Run the app:**

```
node app.js
```

4. **Restart app on changes:**

```
nodemon app.js
```

5. **Debugging:**

```
node --inspect app.js
```

6. **Check vulnerabilities:**

```
npm audit
```

What is Express.js?

Express.js is a **web application framework** built for **Node.js**. It simplifies building server-side applications and APIs. Express.js provides robust features and a minimalistic structure to streamline web development.

Key Features of Express.js:

1. **Minimal and Flexible**: Provides core features for web applications without dictating the project structure.
2. **Routing**: Simplifies handling URLs and HTTP methods like `GET`, `POST`, `PUT`, and `DELETE`.
3. **Middleware Support**: Helps manage request processing, such as logging, parsing, or authentication.
4. **Templating Engines**: Supports templating engines like EJS, Pug, and Handlebars for rendering dynamic web pages.
5. **REST API Development**: Ideal for building RESTful APIs quickly and efficiently.

Why Use Express.js?

- Reduces boilerplate code for setting up servers in Node.js.

- Improves developer productivity with routing and middleware capabilities.
- Highly customizable and integrates easily with other Node.js libraries.

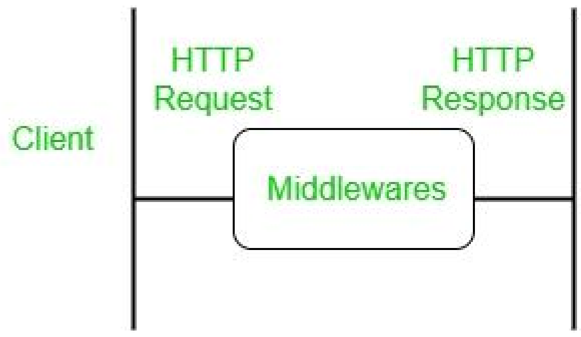
How Node.js and Express.js Work Together

1. **Node.js** acts as the runtime environment, enabling JavaScript to run on the server.
2. **Express.js** simplifies server creation and provides tools for routing, middleware, and API development.

Example Workflow:

- Node.js starts the server.
- Express.js handles routing, requests, and responses.

Middleware and routing



What is Middleware in Express JS?

Middleware is a request handler that allows you to intercept and manipulate requests and responses before they reach route handlers. They are the functions that are invoked by the Express.js routing layer.

It is a flexible tool that helps add functionalities like logging, authentication, error handling, and more to Express applications.

Syntax

The basic syntax for the middleware functions is

app.get(path, (req, res, next) => {}, (req, res) => {})

How the App Works?

The first line imports Express in our file, we have access to it through the variable

Express. We use it to create an application and assign it to var app.

app.get(route, callback)

This function tells what to do when a get request at the given route is called. The

callback function has 2 parameters, ***request(req)*** and ***response(res)***. The

request **object(req)** represents the HTTP request and has properties for the request

query string, parameters, body, HTTP headers, etc. Similarly, the response object

represents the HTTP response that the Express app sends when it receives an HTTP

request.

res.send()

This function takes an object as input and it sends this to the requesting client. Here we are sending the string *"Hello World!"*.

app.listen(port, [host], [backlog], [callback]])

This function binds and listens for connections on the specified host and port.

Types of Middleware

Express JS offers different types of middleware and you should choose the middleware on the basis of functionality required.

- **Application-level middleware:** Bound to the entire application using **app.use()** or **app.METHOD()** and executes for all routes.

  ```
  app.use((req, res, next) => {
      console.log('Application-level Middleware');
      next();
  });
  ```

- **Router-level middleware**: Associated with specific routes using **router.use()** or **router.METHOD()** and executes for routes defined within that router.

  ```
  const router = express.Router();


  router.use((req, res, next) => {
      console.log('Router-level Middleware');
      next();
  ```

```
});


router.get('/user', (req, res) => {
    res.send('User Route');
});


app.use('/api', router);
```

- **Error-handling middleware:** Handles errors during the request-response cycle. Defined with four parameters (err, req, res, next).

  ```
  app.use((err, req, res, next) => {
      console.error(err.stack);
      res.status(500).send('500 page not found!');
  });
  ```

- **Built-in middleware:** Provided by Express (e.g., express.static, express.json, etc.).

  ```
  app.use(express.json()); // Parse JSON requests
  app.use(express.urlencoded({ extended: true })); // Parse URL-encoded data
  ```

- **Third-party middleware**: Developed by external packages (e.g., body-parser, morgan, etc.).

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

app.use(morgan('tiny')); // Logs incoming requests
```

Express.js Routing

Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds to a client request to a particular route, URI or path and a specific HTTP request method (GET, POST, etc.). It can handle different types of HTTP requests.

The basic syntax for routing in Express:

app.METHOD(PATH, HANDLER);

- `app` → Instance of Express.
- `METHOD` → HTTP method (e.g., `get`, `post`, `put`, `delete`).
- `PATH` → The URL path or route (e.g., `/`, `/users`).
- `HANDLER` → A callback function executed when the route is matched.

Express JS Routing -First Challenge in Express JS

```
var express = require('express');

var app = express();

app.get('/', function (req, res) {

res.send('<html><body><h1>Hello World</h1></body></html>');
});
app.post('/submit-data', function (req, res) {
res.send('POST Request');
});
app.put('/update-data', function (req, res) {
res.send('PUT Request');
});
app.delete('/delete-data', function (req, res) {
res.send('DELETE Request');
});
var server = app.listen(5000, function () {
console.log('Node server is running..');
});
```

The **var express = require('express')** line imports the Express.js module, which is used to

create the web server. The **var app = express()** line creates an instance of the Express

application.

The **app.get()** function creates a route for a GET request to the root path **'/'**, which

responds with the HTML content of a "Hello World" message.

The **app.post()**, **app.put()**, and **app.delete()** functions create routes for HTTP POST, PUT,

and DELETE requests respectively. These routes respond with strings indicating the type

of request made.

The **var server = app.listen(5000, function () {...})** line starts the web server on port

5000 and logs a message to the console when the server starts running.

Callback Function in Express.js

1. What is a Callback Function in Express.js?

A callback function in Express.js is executed after an HTTP request matches a specific route or after a middleware is invoked. It is used to handle requests and send responses.

---

2. Basic Usage in Routes

In a route, the callback function processes the incoming request (`req`) and prepares the outgoing response (`res`).

Example:

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

// Route with a callback function

app.get('/', (req, res) => {

  res.send('Hello, World!');

});

// Start the server

app.listen(3000, () => console.log('Server running on
http://localhost:3000'));
```

- `req` **(Request)**: Contains information about the HTTP request, such as headers, parameters, and body.
- `res` **(Response)**: Used to send a response back to the client.
- `next` **(Optional)**: Passes control to the next middleware or route handler.

Await in express js

In Express.js, using async/await allows you to work with asynchronous code in a more readable and maintainable manner compared to traditional callback-based approaches or .then()/.catch() methods. Since Express is built on top of Node.js and is heavily asynchronous (due to I/O operations like database queries, file handling, etc.), async/await is a natural fit for handling asynchronous operations within route handlers and middleware.

Async function in Express js

In **Express.js**, async functions are used to handle asynchronous operations cleanly and avoid callback hell. They allow you to use await to pause execution until a Promise resolves, making code easier to read and maintain.

what do ypu mean by promise?

A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises help manage asynchronous tasks in a more structured and readable way, avoiding the infamous "callback hell."

1. Why Use Promises?

Promises are designed to:

- Handle **asynchronous operations** such as fetching data, reading files, or waiting for events.
- Make code more readable and maintainable than using nested callbacks.
- Simplify error handling by providing a single place to manage errors.

---

2. The Lifecycle of a Promise

A Promise can be in one of three states:

1. **Pending**: The initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation completed successfully, and a value is available.
3. **Rejected**: The operation failed, and a reason (error) is available.

Once a Promise transitions to either **fulfilled** or **rejected**, it is said to be **settled** and cannot change states.

---

3. Creating a Promise

You create a Promise using the `Promise` constructor, which takes a function as an argument. This function is called the **executor function** and has two parameters:

- **`resolve`:** A function to mark the Promise as fulfilled.
- **`reject`:** A function to mark the Promise as rejected.

Example:

```
const myPromise = new Promise((resolve, reject) => {

  const success = true;

  if (success) {

    resolve('Operation succeeded!');

  } else {

    reject('Operation failed.');

  }

});
```

Introduction

Passport.js is a popular authentication center for Node.js, designed to easily add authentication mechanisms to your application. It is highly flexible and modular, allowing you to authenticate requests using a variety of methods including local username and password, OAuth, OAuth2, and more

Features of Passport.js

- **Modularity**: It supports over 500 methods of authentication, including social login providers like Google, Facebook, and Twitter, as well as traditional methods like username and password

- **Middleware integration**: Designed to work seamlessly with Express and other web frameworks, it is used as middleware, making it easy to integrate into your existing applications
- **Session management**: Supports session management, helps you manage user sessions and provides options for organizing and recording user information.
- **Customizable**: Highly customizable to meet the specific needs of your application, and allows you to define custom validation functions and authentication logic.

Installation

To install Passport.js, use npm:
Additionally, you may need to install specific strategies depending on your authentication needs. For example, for local authentication:

```
// npm init -y
//npm install express passport passport-local express-session bcrypt connect-flash mongoose
```

Here's what each package does:

express: A fast web framework for Node.js, used for building web applications and APIs.

passport: Middleware for handling authentication in Node.js applications. It supports various authentication strategies.

passport-local: Passport strategy for username and password authentication (local authentication).

express-session: Middleware for handling session management, which is required by Passport to store user sessions.

bcrypt: A library for hashing and verifying passwords securely (important for storing passwords safely).

connect-flash: Middleware for passing flash messages between requests (useful for showing error or success messages after login attempts).

mongoose: An ODM (Object Data Modeling) library for MongoDB, used to interact with the database in a more structured way.

Why use Passport.js?

- **Easy to integrate**: It simplifies the process of adding authentication to your Node.js application, especially when using a popular web framework like Express.
- **Multiple methods**: With support for over 500 authentication methods, it can handle almost any authentication requirement from traditional username and password management to social login
- **Flexibility**: Its modular design allows you to select and configure only the methods you need, making your application lightweight and focused.
- **Community and Support**: It has a large and dynamic community, which provides examples, tutorials, and plenty of support to help you get started and solve any problems you may encounter.

Authentication and Security of passport. js

Explain the use of passport. js for authentication in Express applications.

Authentication is an important aspect of maintaining the overall security of the application. As Express. js is the server-side language, it is important to secure the application with authentication. So we can use Passport.js which is the authentication middleware that has a modular way to implement the authentication strategies which includes the OAuth, username/password, and more authentication features.

Example :

```
// Import necessary modules

const passport = require('passport');

const LocalStrategy = require('passport-local').Strategy;

const User = require('./models/User');  // Assume this is a user
model that connects to your database



// Define local strategy

passport.use(new LocalStrategy(

  function(username, password, done) {

    User.findOne({ username: username }, function(err, user) {

      if (err) return done(err);  // If there's an error with the
database

      if (!user) return done(null, false, { message: 'Incorrect
username.' });  // User not found

      if (!user.verifyPassword(password)) return done(null, false,
{ message: 'Incorrect password.' });  // Invalid password

      return done(null, user);  // Successful authentication
    });
  }
));

// Express routes to handle login
app.post('/login', passport.authenticate('local', {
  successRedirect: '/dashboard',
  failureRedirect: '/login',
  failureFlash: true
}));
```

Mongoose is a **MongoDB Object Data Modeling (ODM) library** for Node.js. It provides a schema-based solution to model application data and interact with MongoDB databases. Mongoose simplifies operations like validation, query building, and business logic integration with MongoDB.

Key Features of Mongoose

1. **Schema Definition**: Allows defining the structure of documents in a collection.
2. **Data Validation**: Ensures data integrity with built-in validation.
3. **Middleware**: Supports pre- and post-hooks for operations like saving, deleting, etc.

4. **Query Building**: Simplifies writing complex queries.
5. **Plugins**: Extensible via plugins for added functionality (e.g., timestamps, pagination).

Security with Passport.js

Authentication involves sensitive user information, so it's important to implement security best practices:

- **Use HTTPS**: Always encrypt data in transit by using HTTPS.
- **Password Hashing**: Never store passwords in plaintext. Use a strong hashing algorithm (e.g., bcrypt) to hash passwords.
- **Session Security**: Use secure session cookies (`httpOnly`, `secure`, `sameSite`) to prevent attacks like session hijacking.
- **Rate Limiting**: Protect against brute-force attacks by limiting the number of login attempts per user/IP.

Error handling

Express has a built-in error-handling middleware. You can define this middleware at the **end of the middleware stack** to catch errors thrown in routes or other middleware.

Example of error handling :

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

const app = express();

// Example route that throws an error

app.get('/', (req, res) => {

  throw new Error('Something went wrong!');
});

// Error handling middleware (defined at the end)

app.use((err, req, res, next) => {

  console.error(err.stack);  // Log the error stack

  res.status(500).json({

    message: err.message || 'Internal Server Error'

  });

});

app.listen(3000, () => {

  console.log('Server is running on port 3000');

});
```

logging

Logging in **Express.js** is a crucial part of building robust and maintainable applications. Proper logging helps developers monitor application behavior, debug issues, and track important events or errors that occur during the application's runtime. In Express, logging can be handled through custom solutions (like console.log), or using more advanced libraries like **Winston** or **Morgan**.

Winston is a highly configurable logging library that provides advanced features like log levels, logging to multiple destinations (console, files, remote services), and structured logging.

```
npm install winston
```

Example of logging :

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

const winston = require('winston');

const app = express();


// Create a Winston logger instance

const logger = winston.createLogger({

  level: 'info',

  format: winston.format.combine(

    winston.format.timestamp(),

    winston.format.json()  // Log in JSON format (can also use
winston.format.simple() for plain text)

  ),

  transports: [

    new winston.transports.Console({

      format: winston.format.combine(

        winston.format.colorize(),  // Add color to console output

        winston.format.simple()

      ),
    }),
    new winston.transports.File({ filename: 'logs/app.log' })  //
Log to file
  ],
});

// Custom middleware to log requests
app.use((req, res, next) => {
  logger.info(`Request: ${req.method} ${req.url}`);
  next();
});

// Example route
app.get('/', (req, res) => {
  logger.info('GET / route hit');
  res.send('Hello, World!');
});

// Example error route
app.get('/error', (req, res) => {
  logger.error('Error occurred in /error route');

  res.status(500).send('Something went wrong!');

});


app.listen(3000, () => {

  logger.info('Server is running on port 3000');

});
```