Node JS

- open-source and cross-platform JavaScript runtime environment

- Single Threaded (handle many connections at once without waiting for one to finish before starting another)

Usage

- Real-time applications (chats, gaming, collaboration tools)
- APIs and microservices
- Data streaming applications
- Command-line tools
- Server-side web applications

Create a Server

```
const { createServer } = require('http');

const port = 3000;

const server = createServer((req, res) => {

 res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Welcome to node server');});

server.listen(port, hostname, () => {  console.log(`Server running at http://${hostname}:${port}/`);});
```

```
> node server.js
```

Watch

Watch changes on the code and restart the server immediately

> node –watch server.js

V8 Engine

- **Interpretation and Compiling**
- **Efficient Memory Management and Garbage Collection**
- **Support for Modern JavaScript (ECMAScript) Features - Promises, async/await, classes, and arrow functions**

Workload

- **Event Driven**
- **Non Blocking I/O**
- **Single Threaded**

Asynchronous Programming

Synchronous: Tasks execute sequentially. A task must complete before the next one starts (blocking).

Asynchronous: Tasks start and run independently. The program can continue executing other code while waiting for a result (non-blocking).

The Need for Asynchronous Code in Node.js:

Node.js is designed for I/O-intensive operations (networking, file system, database access).

Asynchronous operations prevent the single-threaded Node.js environment from blocking while waiting for I/O.

Callbacks:

A function passed as an argument to another function, which is executed once the asynchronous operation is complete.

Example: fs.readFile(file, callbackFunction)

Asynchronous Programming

- **Do other work while waiting for tasks**

  Sync - tasks order : 1->2->3->4

  Async - tasks order: 1->3->4->2

Timers

setTimeout():

Executes a function once after a specified delay.

setInterval():

Executes a function repeatedly at a fixed interval.

setImmediate():

Executes a function immediately after the current I/O phase of the event loop completes.

Promises

The Problem with Callbacks (Callback Hell):

Nested callbacks can make code difficult to read, debug, and maintain.

Promises (ES6):

A cleaner way to handle asynchronous results. A Promise represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

States: Pending, Fulfilled (resolved), Rejected.

Promise Chaining:

Uses .then() for successful completion and .catch() for errors.

Allows sequential asynchronous operations to be executed and managed cleanly.

Async and Await

Built on Promises:

async/await is syntactic sugar for working with Promises. It makes asynchronous code look and behave more like synchronous code.

async Keyword:

Used to declare an asynchronous function. An async function always returns a Promise.

await Keyword:

Used inside an async function to pause execution until a Promise resolves.

Error Handling:

Use standard try...catch blocks for error handling, simplifying error management compared to traditional Promises.

Modules

- CommonJS
- ES Module

CommonJS

export.fnName = functionName();

module.exports ={ fn1, fn2}

to import use :

require

ES Module

export function fnName()

to import use:

import

Project Creation

> npm init -y

- Name
- Version
- Main
- Scripts
- Keywords
- author
- License
- description

PM2

Process Management for production

```
> npm install -g pm2

 # Start an application

pm2 start app.js

# List all running applications

pm2 list

# Monitor resources

pm2 monit

# View application logs

pm2 logs

# Stop an application

pm2 stop app_name

# Restart an application

pm2 restart app_name

# Delete an application from PM2

pm2 delete app_name
```

Framework

Productivity: Frameworks provide pre-built solutions for common tasks like routing, middleware management, and templating.

Standardization: They establish patterns and structures that make code more maintainable and easier to understand.

Community: Popular frameworks have large communities, extensive documentation, and many third-party plugins or extensions.

Security: Well-maintained frameworks often include built-in security features and best practices.

Performance: Many frameworks are optimized for performance and provide tools for caching, load balancing, and more

Express

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

const app = express();

const port = 8080;

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

  res.send('Hello World from Express.js!');

});

app.listen(port, () => {

  console.log(`Express server running at
http://localhost:${port}`);

});
```

Routes

.get, .post, .put, .patch

- Query (?)

- params (:)

Express

```
app.get('/search/:bookId', (req, res) => {

  res.send(`Book ID: ${req.params.bookId}`);

// for more parameter values - http://localhost:8000/search/02

});

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

  const { bookId } = req.query;

  res.send(`Book ID: ${bookId}`);

// for less parameter values -
http://localhost:8000/search?bookId=02

});
```

Middlewares

```
// Middleware to parse JSON request bodies

app.use(express.json());

// Middleware to parse URL-encoded request bodies

app.use(express.urlencoded({ extended: true }));

// Middleware to serve static files from a directory

app.use(express.static('public'));

// middleware usage

app.use((req, res,) => {console.log('Time:', Date.now());});
```

Routes

Like Menus in web page

Routes the corresponding API

to endpoint

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

const router = express.Router();

// Router-level middleware

router.use((req, res) => {

  console.log('Router specific middleware');

});

router.get('/:id', (req, res) => {

  res.send('Event Info);

});

router.get('/:number', (req, res) => {

  res.send('Event Info);

});

// Add the router to the app
```

Project Structures

server.js or index.js

.env

- config

- routes

- controllers

- middlewares

- models