**EXPRESS JS**

Trainer **:** Saravanan Durai

MERN and MEAN STACK Developer

**Express.js**

Express.js is a popular Node.js web framework that allows you to build web applications quickly and efficiently.

**Key Features**

1. Routing: Define routes for handling HTTP requests.

2. Middleware: Use middleware functions to perform tasks such as authentication and data parsing.

3. Templating: Use templating engines to render dynamic HTML templates.

**Benefits**

1. Fast and flexible: Express.js is lightweight and flexible, making it ideal for building web applications.

2. Large ecosystem: Express.js has a large ecosystem of middleware and plugins.

3. Easy to learn: Express.js has a simple and intuitive API.

**Use Cases**

1. Web applications: Build web applications with Express.js, including RESTful APIs and web servers.

2. API development: Use Express.js to build RESTful APIs and microservices.

3. Real-time applications: Build real-time applications with Express.js and WebSockets.

**Example**

const express = require('express');

const app = express();

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

res.send('Hello World!');

});

app.listen(3000, () => {

console.log('Server listening on port 3000');

});

Express.js is a powerful and flexible framework for building web applications and APIs.

**Nodemon**

Nodemon is a tool that automatically restarts your Node.js application when it detects changes in your code.

**Installation**

npm install nodemon --save-dev

**Usage**

1. Run your application with Nodemon:

**bash**

nodemon app.js

2. Configure Nodemon: You can configure Nodemon using a nodemon.json file or command-line options.

**Benefits**

1. Automatic restarts: Nodemon automatically restarts your application when it detects changes.

2. Faster development: Nodemon saves you time by eliminating the need to manually restart your application.

**Features**

1. File watching: Nodemon watches for changes in your files and restarts your application accordingly.

2. Customizable: Nodemon is highly customizable, allowing you to specify which files to watch and how to restart your application.

**Dotenv**

Dotenv is a popular library for loading environment variables from a .env file.

**Installation**

npm install dotenv

**Usage**

**1. Create a .env file in the root of your project:**

PORT=3000

DB\_URL=mongodb://localhost:27017

**2. Load the .env file in your Express.js application:**

require('dotenv').config();

const express = require('express');

const app = express();

const port = process.env.PORT;

const dbUrl = process.env.DB\_URL;

app.listen(port, () => {

console.log(`Server listening on port ${port}`);

});

**Benefits**

1. Environment-specific configuration: Store environment-specific variables in separate .env files.

2. Security: Keep sensitive information, such as database credentials, out of version control.

**Best Practices**

1. Add .env to .gitignore: Prevent sensitive information from being committed to version control.

2. Use different .env files: Use separate .env files for different environments (e.g., development, production).

Dotenv simplifies environment variable management in Express.js applications, making it easier to switch between different environments and keep sensitive information secure.

**Body Parser**

Body Parser is a middleware that parses the body of incoming HTTP requests, allowing you to access the request body data.

**Installation**

npm install body-parser

**Usage**

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

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

**Options**

1. json(): Parses JSON request bodies.

2. urlencoded(): Parses URL-encoded request bodies.

3. extended: If true, uses the qs library to parse URL-encoded data.

**Accessing Request Body**

After parsing the request body, you can access the data using req.body:

app.post('/api/data', (req, res) => {

console.log(req.body);

res.json(req.body);

});

**Express.js Built-in Body Parser**

As of Express.js 4.16.0, the express.json() and express.urlencoded() middleware are built-in, making Body Parser optional:

app.use(express.json());

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

Body Parser is a useful middleware for handling request body data in Express.js applications.

**CORS**

**Enabling CORS in Express.js**

To enable CORS in Express.js, you can use the cors middleware.

**Installation**

npm install cors

**Usage**

const express = require('express');

const cors = require('cors');

const app = express();

app.use(cors());

**Options**

You can configure CORS options by passing an object to the cors middleware:

const corsOptions = {

origin: 'http://example.com',

methods: ['GET', 'POST'],

allowedHeaders: ['Content-Type', 'Authorization'],

preflightContinue: false,

optionsSuccessStatus: 200

};

app.use(cors(corsOptions));

**Common Options**

1. origin: Specifies which origins are allowed to access the resource.

2. methods: Specifies which HTTP methods are allowed.

3. allowedHeaders: Specifies which headers are allowed.

Example with Specific Routes

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

res.json({ data: 'Hello World!' });

});

By using the cors middleware, you can easily enable CORS in your Express.js application and configure it to suit your needs.

**EJS**

EJS (Embedded JavaScript) is a templating engine that allows you to render dynamic HTML templates with Express.js.

**Installation**

npm install ejs

**Usage**

1. Set up EJS as the templating engine in your Express.js application:

const express = require('express');

const app = express();

app.set('view engine', 'ejs');

2. Create an EJS template file (e.g., index.ejs):

<!DOCTYPE html>

<html>

<head>

<title><%= title %></title>

</head>

<body>

<h1><%= message %></h1>

</body>

</html>

3. Render the EJS template in your Express.js route:

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

res.render('index', { title: 'Hello World', message: 'Welcome to my website!' });

});

**Benefits**

1. Dynamic templating: Render dynamic HTML templates with EJS.

2. Easy to use: EJS has a simple and intuitive syntax.

**Features**

1. Embedded JavaScript: Use JavaScript syntax to render dynamic data in your templates.

2. Conditionals and loops: Use conditionals and loops to control the flow of your templates.

EJS is a popular templating engine for Express.js, allowing you to render dynamic HTML templates with ease.

**JSON Web Tokens (JWT)**

JSON Web Tokens (JWT) is a popular authentication mechanism for securing web applications.

**Installation**

npm install jsonwebtoken

**Generating a Token**

const jwt = require('jsonwebtoken');

const token = jwt.sign({ userId: 1 }, 'secretKey', { expiresIn: '1h' });

**Verifying a Token**

jwt.verify(token, 'secretKey', (err, decoded) => {

if (err) {

console.log('Invalid token');

} else {

console.log(decoded); // { userId: 1 }

}

});

**Using JWT with Express.js**

1. Generate a token: Generate a token when a user logs in.

2. Verify the token: Verify the token on each request to protected routes.

**Example**

const express = require('express');

const jwt = require('jsonwebtoken');

const app = express();

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

const user = { id: 1 };

const token = jwt.sign(user, 'secretKey');

res.json({ token });

});

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

res.json({ message: 'Hello World!' });

});

function authenticate(req, res, next) {

const token = req.header('Authorization');

jwt.verify(token, 'secretKey', (err, decoded) => {

if (err) {

res.status(401).json({ message: 'Invalid token' });

} else {

req.user = decoded;

next();

}

});

}

**Benefits**

1. Stateless authentication: JWT is a stateless authentication mechanism, making it scalable and efficient.

2. Secure: JWT is digitally signed, making it secure and tamper-proof.

JSON Web Tokens (JWT) is a popular authentication mechanism for securing web applications, and it can be easily integrated with Express.js.

**Express File Upload**

Express File Upload is a popular middleware for handling file uploads in Express.js applications.

**Features**

1. Multipart/form-data support: Handles file uploads from forms with multipart/form-data encoding.

2. File validation: Supports file type, size, and other validation rules.

3. File storage: Allows you to store uploaded files in memory or on disk.

**Installation**

To install Express File Upload, run the following command:

**bash**

npm install express-fileupload

**Usage**

Here's an example of how to use Express File Upload:

**Program**

const express = require('express');

const fileUpload = require('express-fileupload');

const app = express();

app.use(fileUpload());

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

if (!req.files || Object.keys(req.files).length === 0) {

return res.status(400).send('No files were uploaded.');

}

const file = req.files.file;

file.mv('/path/to/upload/folder/' + file.name, (err) => {

if (err) {

return res.status(500).send(err);

}

res.send('File uploaded!');

});

});

Documentation

For more information on Express File Upload, including options, events, and examples, please refer to the official documentation:

https://www.npmjs.com/package/express-fileupload

Express File Upload is a powerful and flexible middleware for handling file uploads in Express.js applications.

**Mongoose with Express.js**

Mongoose is a popular MongoDB ORM (Object Relational Mapping) library for Node.js.

**Installation**

npm install mongoose

**Connecting to MongoDB**

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/mydatabase', { useNewUrlParser: true, useUnifiedTopology: true });

**Defining a Model**

const userSchema = new mongoose.Schema({

name: String,

email: String

});

const User = mongoose.model('User', userSchema);

**CRUD Operations**

1. Create: User.create({ name: 'John Doe', email: 'john@example.com' })

2. Read: User.find().then(users => console.log(users))

3. Update: User.findByIdAndUpdate(id, { name: 'Jane Doe' })

4. Delete: User.findByIdAndRemove(id)

**Using Mongoose with Express.js**

const express = require('express');

const app = express();

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/mydatabase');

const User = mongoose.model('User', {

name: String,

email: String

});

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

User.find().then(users => res.json(users));

});

Benefits

1. Easy MongoDB interaction: Mongoose provides a simple and intuitive way to interact with MongoDB.

2. Schema-based: Mongoose uses schemas to define the structure of your data.

Mongoose is a powerful tool for working with MongoDB in Node.js applications, and it pairs well with Express.js.