Date:- 8/3/2025

Module 04

Page 59

Express.js

1. Define Express.js and explain its role in web development.

→

- Express.js is a fast, minimal and flexible web framework for Node.js used to build web Applications and APIs.
- It simplifies backend development by providing a structured way to handle routing, middleware, request handling and HTTP methods.

→ Role in web Development :-

- Simplifies server-side development with built-in features.
- Provides a robust routing systems to manages different API endpoints.
- Supporting middleware for request processing authentication and error handling.
- Used for building RESTful APIs and full stack applications with frontend framework like React, Angular & vue.js.

2. Describe the basic structure of an Express.js applications.

→

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

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

```
    res.send("Hello, world!');
});

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

→ A Basic Express.js applications :-

1. Import Express
2. creating an Express App
3. Defining Routes
4. starting the server

3. Write a basic Express.js server that listen on port 3000 and responds with "Hello, Express!" when accessed via a GET request to the root route ("/").

→

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

app.get('/', (req, res) => {
    res.send('Hello, Express!');
});

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

4. Explain the concept of routing in Express.js and how it maps HTTP requests of specific endpoints.

→

- Routing in Express.js :-

- Routing in Express.js refers to defining URL endpoints that handle HTTP requests (GET, POST, PUT, DELETE etc.) It allows the server to respond differently based on the URL path and request method.

- How Routing Works :-

- Each route is defind using a specific HTTP Method (app.get(); app.post())
- Express maps incoming requesting to the appropriate route handler.
- Routes can include dynamic parameters for handling different inputs.

- Example :-

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

app.get('/', (req, res) => {
    res.send('welcome to the Homepage!');
});

app.get('/about', (req, res) => {
    res.send('About Page');
});
```

```
app.get('/user/:name', (req, res) => {
    res.send('Hello, ${req.params.name}');
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

5. Discuss the importance of middleware in Express.js and provide example of common middleware functions.

→

- Middleware functions in Express.js are functions that execute during the request response cycle.
- They can modify the request, response or terminate the request cycle before reaching the final route handler.

→ Importance of Middleware :-

- Request Processing
- Code Reusability
- Error handling
- Security.

→ Common Middleware Functions in Express.js

1. Built-in Middleware :-

- Express.json()
- express.urlencoded({ extended: true })
- express.static('public')

2. Third-party Middleware :-

- cors, Helmet, morgan

3. Custom Middleware Example :-

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

const loggerMiddleware = (req, res, next) => {
    console.log('Request : ${req.method} ${req.url}');
    next();
};

app.use(loggerMiddleware);
app.get('/', (req, res) => {
    res.send('Hello, Express with Middleware!');
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

6. Explain the purpose of a template engine in web development and its integration with Express.js.

→

- A Temple engine is used in web development to generate dynamic HTML pages. Instead of writting static HTML files, a template engine allows embedding variable, loops and conditional statements within HTML files.

→ Purpose :-

- Generates dynamic content by inserting data into templates.
- Separates logic from presentation, making code more organized.
- Improves maintainability by using reusable components.
- Enhances flexibility by allowing loops and conditional rendering.

→ Integrating EJS with Express.js :-

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

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

app.get('/', (req, res) => {
    res.render('index', {title: 'welcome',
    message: 'Hello, Express with EJS!'});
});

app.listen(3000, () => console.log('Server
    running on Port 3000'));
```

7. Write an Express.js route handler that retrieves data from a mock database and sends it as a JSON response when a specific endpoint ("/data") is accessed via an HTTP GET request.

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

const mockDatas = [
    {id:1, name: 'Alice', age:25},
    {id:2, name: 'Jil Dumgsani', age:20},
    {id:3, name: 'Sujal Lad', age:8774'
];

app.get('/datas', (req, res) => {
    res.json(mokDatas);
});

app.listen(3000, () => console.log('Server
    running on port 3000'));
```

8. Develop an Express.js application that handles a POST request to an endpoint ("/login") with username and password parameters in the request body, and responds with a success message if the credentials are valid.

→

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

app.use(express.json());
const validUser = {username: 'admin',
                   password: '2055'};
```

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  if (username === validUser.username
      && password === validUser.password
  ) {
    res.json({ message: 'Login
        Successful!' });
  } else {
    res.status(401).json({ error:'
      Invalid Credentials' });
  }
});

app.listen(3000, () => console.log('Server
    running on port 3000'));
```

9. Describe how Express.js handles HTTP methods such as GET, POST, PUT & DELETE.

→

- Express.js provides method to handle different HTTP requests (GET, POST, PUT, DELE which are used in RESTful APIs for CRUD operations (create, Read, Update, delete).

i) GET Request :- used to retrive data from the server.

```
app.get('/users', (req, res) => {
  res.send('Fetching all users...');
});
```

iii. POST Request :- used to send data to the server for creating a new resource.

```
app.post('/users', (req, res) => {
    const newUser = req.body;
    res.status(201).json({ message: 'User created!', data: newUser });
});
```

iii) PUT Request :- Used to update an existing resources on the server.

```
app.put('/users/:id', (req, res) => {
    const userId = req.params.id;
    res.json({ message: `User ${userId} updated!` });
});
```

iv) DELETE Request :- Used to Remove a resource from the server.

```
app.delete('/users/:id', (req, res) => {
    const userId = req.params.id;
    res.json({ message: `User ${userId} deleted!` });
});
```

10. Develop a RESTful API using Express.js for a blogging platform with endpoints for creating, reading, updating and deleting blog points, as well as endpoints for commenting on posts and liking posts.

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

app.use(express.json());

let posts = [];

// Create a blog post (POST)
app.post('/posts', (req, res) => {
    const post = { id: posts.length + 1,
        title: req.body.title,
        content: req.body.content,
        likes: 0,
        comments: [] };
    posts.push(post);
    res.status(201).json({ message: 'Post
        created!', post });
});

// Get all blog posts (GET)
app.get('/posts', (req, res) => {
    res.json(posts);
});

// Get a specific post by ID (GET)
app.get('/posts/:id', (req, res) => {
    const post = posts.find(p => p.id ===
        req.params.id);
    if (!post) return
        res.status(404).json({ message: 'Post
        not found' });
    res.json(post);
});
```

```
//Get a specific post by ID (GET)
app.get('/posts/:id', (req, res) => {
    const post = posts.find(p => p.id
```

```
//Update a blog post (PUT)
app.put('/posts/:id', (req, res) => {
    let posts=posts.find(p => p.id ===
                req.params.id);
    if (!post) return res.status(404).json
        ({message:'post not found'});

    post.title = req.body.title || post.title;
    post.content = req.body.content ||
                Post.content;
    res.json({message:'Post updated!', post});
});
```

```
//delete a blog post (DELETE)
app.delete('/posts/:id', (req, res) => {
    posts = posts.filter(p => p.id !== req.params.
        id);
    res.json({message:'Post deleted!'});
});
```

```
//add a comment to a Post (POST)
app.post('/posts/:id/comments', (req, res)=>{
    let post = posts.find(p => p.id ==
                req.params.id);
    if(!post) return res.status(404).json
        ({message:'post not found'});
    post.comments.push(req.body.comment)
    res.json({message:'comment added!'
                post});
});
```

// like a post (PUT)
app.put('/posts/:id/like', (req, res) => {
  let post = posts.find(p => p.id ==
    req.params.id);
  if (!post) return res.status(404).json
  ({message: 'Post not found'});

  post.likes += 1;
  res.json({message: 'post liked', post
});

app.listen(3000, () => console.log('Blog API
running on Port 3000'));