Notes: Basic Node and Express (FreeCodeCamp Backend Development & APIs)

What is Express.js?

Express is a **lightweight web application framework** and one of the most popular npm packages. It makes creating servers and handling routing much easier than using raw Node.js. Express handles directing users to correct pages when they visit specific endpoints like `/blog` or `/api/users`.

1. Setting Up Express Server

Basic Server Setup

Express routes follow the structure: `app.METHOD(PATH, HANDLER)`

- **METHOD**: HTTP method (get, post, put, delete)
- **PATH**: Route path on server (string or regex)
- **HANDLER**: Function executed when route matches

```
// Basic Express server setup
let express = require('express');
let app = express();

// Basic route - serves string to GET requests at root path
app.get('/', function(req, res) {
    res.send('Hello Express');
});

// Start server listening on port
app.listen(3000);
```

Key Points:

- Handler functions take `(req, res)` parameters
- `req` = request object, `res` = response object

- Use `res.send()` to send string responses

2. Serving HTML Files

Serving Static HTML

Use `res.sendFile(path)` to respond with files. Requires absolute file path.

```
// Serve HTML file
app.get('/', function (req, res) {
    const path = __dirname + '/views/index.html';
    res.sendFile(path);
});
```

Important:

- `__dirname` is Node global variable for current directory
- Method sets appropriate headers based on file type
- Express evaluates routes from top to bottom

3. Static Assets with Middleware

Serving Static Files

Middleware are functions that intercept route handlers. `express.static()` serves static assets.

```
// Mount static middleware for CSS, JS, images
app.use('/public', express.static(__dirname + '/public'));
```

How it works:

- Files in `/public` directory become accessible via `/public/filename`
- Middleware executes for all requests when no path specified
- Use `app.use(path, middleware)` to mount middleware

4. JSON API Responses

Creating JSON Endpoints

REST APIs serve data (usually JSON) instead of HTML pages.

```
// Serve JSON response
app.get('/json', function (req, res) {
    res.json({"message": "Hello json"});
});
```

Key Features:

- `res.json()` converts JavaScript objects to JSON strings
- Sets appropriate headers for JSON content type
- Closes request-response cycle

5. Environment Variables with dotenv

Using .env Files

Environment variables configure app behavior without code changes.

```
// Load environment variables
require('dotenv').config();

// Use environment variables
app.get('/json', function (req, res) {
    if (process.env.MESSAGE_STYLE === "uppercase") {
        res.json({"message": "Hello json".toUpperCase()});
    } else {
        res.json({"message": "Hello json"});
    }
});
```

Environment Variable Rules:

- Create `.env` file in project root
- Variables are UPPERCASE with underscores: `MESSAGE_STYLE=uppercase`
- No spaces around equals sign
- Access via `process.env.VARIABLE_NAME`
- Variables are passed as strings

6. Middleware Functions

Root-Level Request Logger

Middleware functions take 3 parameters: `(req, res, next)`

```
// Custom logging middleware
app.use((req, res, next) => {
    console.log(`${req.method} ${req.path} - ${req.ip}`);
    next();
});
```

Middleware Concepts:

- Execute code that can modify req/res objects
- Must call `next()` to continue to next function
- Can end cycle by sending response
- Mounted with `app.use()` for all routes
- Order matters - middleware executes in order defined

7. Chaining Middleware

Time Server Example

Multiple middleware functions can be chained in single route.

```
// Chain middleware and handler
app.get('/now', (req, res, next) => {
    // First middleware adds time to request
    req.time = new Date().toString();
    next();
}, (req, res) => {
    // Final handler sends response
    res.json({time: req.time});
});
```

Benefits:

- Split server operations into smaller units
- Better app structure and code reuse
- Can perform validation at each step
- Can block execution or pass to error handlers

8. Route Parameters

Dynamic Route Segments

Route parameters are named URL segments delimited by slashes.

```
// Echo server with route parameter
app.get('/:word/echo', (req, res) => {
    res.json({ echo: req.params.word });
});
```

Parameter Access:

- Route: `/user/:userId/book/:bookId`
- URL: `/user/546/book/6754`
- Access: `req.params.userId` and `req.params.bookId`

9. Query Parameters

Query String Processing

Query strings encode data after route path using `?field=value&field2=value2`

```
// Handle query parameters
app.get('/name', (req, res) => {
    res.json({ name: `${req.query.first} ${req.query.last}` });
});
```

Query String Example:

- Route: `/library`
- URL: `/library?userId=546&bookId=6754`
- Access: `req.query.userId` and `req.query.bookId`

10. POST Requests & Body Parser

Handling POST Data

POST requests send data in request body (hidden from URL).

```
// Setup body parser middleware
let bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: false }));

// Handle POST request
app.post('/name', (req, res) => {
    res.json({ name: `${req.body.first} ${req.body.last}` });
});
```

Body Parser Configuration:

- `extended: false` uses classic querystring library
- `extended: true` uses qs library (more flexible)

- Must mount before routes that need it
- POST data appears in `req.body` object

HTTP Method Conventions

- **POST**: Create new resource
- **GET**: Read existing resource without modification
- **PUT/PATCH**: Update resource using sent data
- **DELETE**: Delete resource

11. Complete Working Example

```
let express = require('express');
let app = express();
require('dotenv').config();
let bodyParser = require('body-parser');

// Middleware setup
app.use('/public', express.static(__dirname + '/public'));
app.use(bodyParser.urlencoded({ extended: false }));

// Root-level logger middleware
app.use((req, res, next) => {
    console.log(`${req.method} ${req.path} - ${req.ip}`);
    next();
});

// Routes
app.get('/', function (req, res) {
    const path = __dirname + '/views/index.html';
    res.sendFile(path);
});

app.get('/json', function (req, res) {
    if (process.env.MESSAGE_STYLE === "uppercase") {
        res.json({"message": "Hello json".toUpperCase()});
    } else {
        res.json({"message": "Hello json"});
    }
});

app.get('/now', (req, res, next) => {
    req.time = new Date().toString();
    next();
}, (req, res) => {
    res.json({time: req.time});
});

app.get('/:word/echo', (req, res) => {
    res.json({ echo: req.params.word });
});

app.get('/name', (req, res) => {
    res.json({ name: `${req.query.first} ${req.query.last}` });
});

app.post('/name', (req, res) => {
    res.json({ name: `${req.body.first} ${req.body.last}` });
});

module.exports = app;
```

Key Development Tips

1. **Server Restart**: Restart server after file changes (`Ctrl + C` then `node server.js`)
2. **Auto-restart**: Use `nodemon` or Node's `--watch` flag
3. **Console Logging**: Use `console.log()` for debugging
4. **Route Order**: Express evaluates routes top to bottom
5. **Middleware Order**: Mount middleware before routes that depend on it

Summary

This module covers fundamental Express.js concepts:

- Creating servers and handling routes
- Serving HTML files and static assets
- Building JSON APIs
- Using environment variables for configuration
- Implementing middleware for logging and request processing
- Handling route parameters and query strings
- Processing POST requests with body parser

These are essential building blocks for any Node.js backend application using Express framework.