Express JS

Alright, let's craft some easy-to-read markdown notes on Express.js and EJS, and then explore how learning Express.js can be helpful in bug bounty hunting and penetration testing.

Express.js Notes

What is Express.js?

- Imagine building a house. You could lay every single brick yourself, which would take a long time and be quite complicated.
- Express.js is like a set of pre-made tools and blueprints for building web applications with Node.js.
- **It's a framework:** This means it provides a basic structure and a bunch of helpful features so you don't have to start from scratch.
- **It simplifies Node.js:** Node.js is powerful for handling server-side tasks, but Express.js makes common web development tasks much easier.
- **Key benefits:**
  - Makes routing (handling different web addresses) simple.
  - Provides easy ways to handle requests and send responses.
  - Has middleware, which are like helpful assistants that can process requests.
  - Lots of community support and available packages to extend its functionality.
- **Use cases:** Building websites, web applications, APIs (ways for different software to talk to each other).

Getting Started with Express.js

1. **Make sure you have Node.js and npm (or yarn) installed:** These are the foundation. You can check by running `node -v` and `npm -v` in your terminal.
2. **Create a project folder:** Make a new folder for your project (e.g., `my-express-app`).
3. **Initialize your project:** Open your terminal in the project folder and run `npm init -y` (or `yarn init -y`). This creates a `package.json` file, which keeps track of your project's dependencies.
4. **Install Express.js:** In your terminal, run `npm install express` (or `yarn add express`). This downloads and adds Express.js to your project.
5. **Create your first Express.js app (e.g., `server.js`):**

```
// Import the Express.js library
const express = require('express');

// Create an Express application
const app = express();

// Define the port the server will listen on
const port = 3000;

// Define a route for the homepage (/)
app.get('/', (req, res) => {
  res.send('Hello World!'); // Send "Hello World!" as the response
});

// Start the server and listen on the specified port
app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`);
});
```

6. **Run your app:** In your terminal, run `node server.js`. You should see the message "Server listening at http://localhost:3000". Open your web browser and go to `http://localhost:3000`. You should see "Hello World!".

Handling Requests

- When you type a web address in your browser and press Enter, your browser sends a **request** to a server.
- Express.js helps you define how your server should handle different types of requests.
- **HTTP Methods (Verbs):** These tell the server what the client wants to do. Common ones include:
  - `GET`: Requesting data from the server (e.g., loading a webpage).
  - `POST`: Sending data to the server to create something new (e.g., submitting a form).
  - `PUT`: Sending data to the server to update something existing.
  - `DELETE`: Asking the server to delete something.
- **Request Object (`req`):** When a request comes in, Express.js creates a `req` object that contains information about the request, such as:
  - `req.url`: The requested URL.
  - `req.method`: The HTTP method used (GET, POST, etc.).
  - `req.headers`: Information about the browser and the request.
  - `req.body`: Data sent in the request body (often for POST or PUT requests).
  - `req.params`: Parameters from the URL (we'll see this in routing).

- `req.query` : Data sent in the URL as query strings (we'll see this later).

Sending a Response

- After your server handles a request, it usually sends back a **response** to the client (the browser).
- Express.js provides a `res` object to help you build and send responses.
- **Common `res` methods:**
  - `res.send(body)` : Sends a simple response, which can be text, HTML, or other data.
  - `res.json(object)` : Sends a JSON (JavaScript Object Notation) response, which is a common format for APIs.
  - `res.sendFile(path)` : Sends a file as the response (e.g., an HTML file, an image).
  - `res.render(view, [locals])` : Renders a template file (like an EJS file) and sends the resulting HTML.
  - `res.redirect(url)` : Sends a redirect response, telling the browser to go to a different URL.
  - `res.status(code)` : Sets the HTTP status code of the response (e.g., 200 for success, 404 for not found).
  - `res.setHeader(name, value)` : Sets a specific HTTP header in the response.

Routing

- **Routing** is the process of determining how an application responds to client requests to specific endpoints (URLs or paths).
- In Express.js, you define routes using the HTTP methods (`app.get()` , `app.post()` , `app.put()` , `app.delete()` , etc.) and a path.

```
app.get('/about', (req, res) => {
  res.send('This is the about page.');
});

app.post('/submit', (req, res) => {
  // Handle the submitted data here
  res.send('Data received!');
});
```

- The first argument to these methods is the **path** (the part of the URL after the domain name).
- The second argument is a **route handler function**. This function is executed when a request matching the method and path is received. It takes two arguments: the `req`

(request) object and the `res` (response) object.

Nodemon

- **Nodemon** is a utility that automatically restarts your Node.js application when it detects file changes.
- This is very helpful during development because you don't have to manually stop and restart your server every time you make a change.
- **Installation:** Install it globally using npm: `npm install -g nodemon` (or `yarn global add nodemon`).
- **Usage:** Instead of `node server.js`, run your app with `nodemon server.js`. Now, whenever you save a file in your project, Nodemon will automatically restart the server.

Path Parameters

- **Path parameters** are dynamic parts of a URL that allow you to capture values from the URL itself.
- They are defined in the route path with a colon (`:`).

```
app.get('/users/:userId', (req, res) => {
  const userId = req.params.userId; // Access the userId parameter
  res.send(`User ID: ${userId}`);
});
```

- If you go to `/users/123` in your browser, the server will respond with "User ID: 123". The value `123` is captured in the `userId` parameter.
- You can have multiple path parameters: `/products/:productId/details/:detailId`.

Query Strings

- **Query strings** are used to send data to the server through the URL. They appear after a question mark (`?`) and consist of key-value pairs separated by ampersands (`&`).

```
/search?query=express&sort=date
```

- In Express.js, you can access query string parameters using `req.query`.

```
app.get('/search', (req, res) => {
  const query = req.query.query; // Access the 'query' parameter
  const sort = req.query.sort;   // Access the 'sort' parameter
```

```
  res.send(`Searching for: ${query}, sorted by: ${sort}`);
});
```

- If you go to `/search?query=node&sort=relevance`, the server will respond with "Searching for: node, sorted by: relevance".

Middleware

- **Middleware** functions are functions that have access to the `req` object, the `res` object, and the next middleware function in the application's request-response cycle.
- They can perform various tasks like:
  - Logging requests.
  - Authenticating users.
  - Parsing request bodies (e.g., for form data or JSON).
  - Adding headers.
  - Handling errors.
- **Types of Middleware:**
  - **Application-level middleware:** Bound to the `app` object using `app.use()`.
  - **Route-level middleware:** Specific to certain routes.
  - **Error-handling middleware:** Special middleware with four arguments (`err`, `req`, `res`, `next`) used to handle errors.
  - **Third-party middleware:** Provided by external packages (e.g., `body-parser` for parsing request bodies).

```
// Application-level middleware (runs for every request)
app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next(); // Pass control to the next middleware function
});

// Route-level middleware (only runs for the /admin route)
const adminAuth = (req, res, next) => {
  // Check if the user is an admin
  const isAdmin = true; // Replace with actual authentication logic
  if (isAdmin) {
    next(); // Allow access to the route
  } else {
    res.status(403).send('Unauthorized');
  }
};

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

```
  res.send('Admin dashboard');
});
```

- The `next()` function is crucial; it tells Express.js to move on to the next middleware function in the chain. If you don't call `next()`, the request will be left hanging.

Serving Static Files

- Web applications often need to serve static files like HTML, CSS, JavaScript, images, etc.
- Express.js provides the `express.static()` middleware for this.

```
app.use(express.static('public')); // Serve files from the 'public'
directory
```

- You would create a folder named `public` in your project directory and put your static files inside it. Then, you can access them in your browser without needing specific routes (e.g., `http://localhost:3000/style.css`, `http://localhost:3000/images/logo.png`).

Error Handling

- It's important to handle errors gracefully in your application.
- You can use error-handling middleware, which is defined like regular middleware but with four arguments: `(err, req, res, next)`.

```
app.get('/error-example', (req, res, next) => {
  try {
    throw new Error('Something went wrong!');
  } catch (error) {
    next(error); // Pass the error to the error-handling middleware
  }
});

// Error-handling middleware (defined last)
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

- When an error occurs in a route handler or regular middleware, you can pass it to `next()`. Express.js will then skip any remaining regular middleware and go directly to the error-handling middleware.

EJS Notes

What is Templating?

- Imagine you have a website with dynamic content (content that changes based on data). You don't want to manually create separate HTML files for every possible piece of data.
- **Templating** is a way to create dynamic web pages using template files that contain placeholders for data.
- A **template engine** takes these template files and data, and then generates the final HTML that is sent to the browser.
- **Benefits of Templating:**
  - Separates presentation (HTML) from application logic (JavaScript).
  - Makes it easier to manage and update the look and feel of your application.
  - Allows you to dynamically insert data into your HTML.
  - Supports control flow (like loops and conditional statements) within your HTML.

Using EJS

- **EJS (Embedded JavaScript templates)** is a simple and popular templating engine for JavaScript.
- It allows you to embed JavaScript code directly within your HTML files.

Views Directory

- By convention, Express.js applications often keep their template files (like EJS files) in a directory named `views` at the root of the project.
- You need to tell Express.js where to find these view files using the `app.set('views', './views')` setting.

```
const express = require('express');
const app = express();
const port = 3000;

// Set the view engine to EJS
app.set('view engine', 'ejs');

// Tell Express where the view files are located
app.set('views', './views');

app.get('/', (req, res) => {
  // Render the 'index.ejs' file and pass some data
  res.render('index', { title: 'My Homepage', message: 'Welcome!' });
```

```
});

app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`);
});
```

- In this example, Express.js will look for a file named `index.ejs` inside the `./views` directory.

Interpolation Syntax

- EJS uses special tags to embed JavaScript within your HTML:
  - `<%= value %>`: Outputs the value directly into the HTML (HTML-escaped to prevent potential security issues like cross-site scripting - XSS).
  - `<%- value %>`: Outputs the raw, unescaped value into the HTML. Use this with caution as it can be a security risk if the data comes from untrusted sources.
  - `<% JavaScript code %>`: Executes JavaScript code without outputting anything directly into the HTML. This is used for control flow (if statements, loops, etc.).
  - `<%# comment %>`: EJS comment, not rendered in the output.
- **Example `index.ejs`:**

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
</head>
<body>
  <h1><%= message %></h1>

  <p>Current time: <%= new Date() %></p>
</body>
</html>
```

- When the `/` route is accessed, Express.js will render `index.ejs` and replace `<%= title %>` with 'My Homepage', `<%= message %>` with 'Welcome!', and `<%= new Date() %>` with the current date and time.

Passing Data to EJS

- You pass data to your EJS templates as an object in the second argument of the `res.render()` method.

```
res.render('profile', { name: 'Alice', age: 30, hobbies: ['reading',
'hiking'] });
```

- In your `profile.ejs` file, you can then access these variables:

```
<h1>Profile</h1>
<p>Name: <%= name %></p>
<p>Age: <%= age %></p>
<ul>
  <% hobbies.forEach(hobby => { %>
    <li><%= hobby %></li>
  <% }); %>
</ul>
```

Conditional Statements

- You can use `<% if (condition) { %> ... <% } %>`, `<% else if (condition) { %> ... <% } %>`, and `<% else { %> ... <% } %>` for conditional rendering.

```
<% if (userLoggedIn) { %>
  <button>Logout</button>
<% } else { %>
  <button>Login</button>
<% } %>
```

Loops

- EJS supports JavaScript loops like `forEach`, `for`, `while`, etc., using the `<% %>` tags.

```
<h2>Products</h2>
<ul>
  <% products.forEach(product => { %>
    <li><%= product.name %> - $<%= product.price %></li>
  <% }); %>
</ul>

<h2>Numbers</h2>
<ul>
  <% for (let i = 0; i < 5; i++) { %>
    <li>Number: <%= i %></li>
```

```
  <% } %>
</ul>
```

Partials

- **Partials** are reusable EJS templates that you can include in other templates. This helps to keep your code organized and avoids repetition.
- You might have a `header.ejs` and a `footer.ejs` that you want to include on multiple pages.
- To use partials, you might need a library like `express-ejs-layouts` or you can implement a simple inclusion mechanism yourself. With `express-ejs-layouts`, you can define a layout template and then inject content into specific sections.

  Example using `express-ejs-layouts`:

  1. **Install:** `npm install express-ejs-layouts` (or `yarn add express-ejs-layouts`)

Okay, let's complete the EJS notes with the `express-ejs-layouts` example and then move on to how Express.js can help with bug bounty hunting and penetration testing.

(Continuing EJS Notes - Partials with `express-ejs-layouts`)

3. **Configure in your `server.js`:**

```
const express = require('express');
const app = express();
const port = 3000;
const expressLayouts = require('express-ejs-layouts');

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

// Use express-ejs-layouts middleware
app.use(expressLayouts);
app.set('layout', 'layouts/main'); // Optional: set a default layout

app.get('/', (req, res) => {
  res.render('index', { title: 'Homepage', content: 'Welcome to the
homepage!' });
});

app.get('/about', (req, res) => {
  res.render('about', { title: 'About Us', content: 'Learn more about our
company.' });
```

```
});

app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`);
});
```

4. **Create a layout file (e.g., `views/layouts/main.ejs`):**

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel="stylesheet" href="/style.css">
</head>
<body>
  <%- body %>
  <footer>
    <p>&copy; 2025 My Website</p>
  </footer>
</body>
</html>
```

5. **Create your view files (e.g., `views/index.ejs` and `views/about.ejs`):**

`views/index.ejs`:

```
<h1><%= title %></h1>
<p><%= content %></p>
```

`views/about.ejs`:

```
<h1><%= title %></h1>
<p><%= content %></p>
```

- In this setup, `express-ejs-layouts` takes your individual view files (`index.ejs`, `about.ejs`) and injects their content into the `<%- body %>` section of the `main.ejs` layout. You can also define other named sections in your layout to inject specific parts of your views.

Folder Structure (Common for Express.js with EJS)

```
my-express-app/
├── node_modules/         (Dependencies installed by npm/yarn)
├── views/                (EJS template files)
│   ├── index.ejs
│   ├── about.ejs
│   └── layouts/          (Layout files, if using express-ejs-layouts)
│       └── main.ejs
├── public/               (Static files like CSS, JavaScript, images)
│   ├── style.css
│   └── script.js
│   └── images/
│       └── logo.png
├── server.js             (Main application file)
├── package.json          (Project configuration and dependencies)
└── package-lock.json     (Detailed dependency information)
```

This is a common, but not the only possible, structure. As your application grows, you might organize your routes, middleware, and models into separate folders for better maintainability.

How Learning Express.js Can Help in Bug Bounty and Penetration Testing

Understanding Express.js can be significantly beneficial in bug bounty hunting and penetration testing of web applications built with this framework. Here's how:

1. **Understanding Application Logic and Structure:**
   - Knowing Express.js helps you understand how the application is likely structured, how routes are defined, how requests are handled, and how data flows. This allows you to more effectively map the application's attack surface.
   - You can anticipate common patterns and vulnerabilities that arise from specific Express.js features or common development practices.
2. **Identifying Routing Vulnerabilities:**
   - Knowledge of Express.js routing enables you to identify potential issues like insecure direct object references (IDOR) where you might be able to access resources by manipulating URL parameters.
   - You can test for improper handling of different HTTP methods on specific routes. For example, a `POST` request might be unexpectedly accepted on a `GET`-only route.
3. **Analyzing Request Handling and Input Validation:**

- Understanding how Express.js handles request parameters (`req.params`, `req.query`, `req.body`) allows you to craft malicious inputs to test for vulnerabilities like:
  - **SQL Injection:** If data from these parameters is directly used in database queries without proper sanitization.
  - **Cross-Site Scripting (XSS):** If user-provided data is rendered in EJS templates without proper escaping (especially when using `<%- %>`).
  - **Command Injection:** If user input is used to execute system commands.

4. **Examining Response Handling:**
   - Knowing how Express.js sends responses helps you analyze the data being returned by the server. This can reveal sensitive information, error messages that disclose internal workings, or clues about the backend technologies used.
   - You can look for insecure headers or missing security headers.
5. **Identifying Middleware Vulnerabilities:**
   - Understanding Express.js middleware is crucial. Misconfigured or vulnerable middleware can introduce security flaws. For example:
     - **Authentication/Authorization bypasses:** If custom authentication middleware has logical flaws.
     - **Exposure of sensitive data:** If middleware logs sensitive information.
6. **Analyzing Templating Engine Vulnerabilities (EJS):**
   - If the application uses EJS, understanding its syntax and potential pitfalls is vital. You can test for Server-Side Template Injection (SSTI) vulnerabilities by injecting malicious JavaScript code into template parameters. Improper use of `<%- %>` is a common entry point for SSTI.
7. **Testing API Endpoints:**
   - Express.js is commonly used to build RESTful APIs. Knowing how these APIs are structured (routes, request methods, data formats like JSON) allows you to effectively test their security, including authentication, authorization, rate limiting, and input validation.
8. **Understanding Common Libraries and Their Vulnerabilities:**
   - Express.js applications often rely on third-party middleware and libraries. Familiarity with common ones (e.g., `body-parser`, `cookie-session`, `passport`) and their known vulnerabilities can guide your testing efforts.
9. **Source Code Analysis (If Available):**
   - If you have access to the application's source code, your knowledge of Express.js and EJS will be invaluable for identifying potential security flaws through static analysis. You can look for insecure coding practices, improper data handling, and misconfigurations.

In summary, learning Express.js provides you with a foundational understanding of how many modern web applications are built on the Node.js platform. This knowledge empowers you to

approach security testing with a more informed perspective, allowing you to identify and exploit vulnerabilities that might be missed by someone unfamiliar with the framework.