Express

Reading Material

PW SKILLS

Topics Covered

- Concept understanding of Express Framework
  - Frameworks
  - Introduction to Express.js
    - Request Object
    - Response Object
  - Express Routes
    - Route chaining
  - Middleware in express js
  - Templating Engine

Frameworks

A framework is a software tool or platform that provides developers with pre-written code, components, and libraries to build applications. Popular frameworks include Ruby on Rails, Django, Laravel, React, Angular, and Vue.js. They help developers save time and effort by providing pre-built code and components that can be reused across different projects. Express.js is an open-source backend web application framework for building Restful APIs with Nodejs, offering a minimalist approach that allows developers to choose their own approach to building web applications.

Introduction to Express.js

Express.js is designed to be unopinionated, allowing developers to choose their own structure and components. Its main features include routing and middleware functions, which allow developers to map HTTP requests to specific handlers and modify or enhance the behavior of requests and responses.

Express.js is essential for building web applications due to its lightweight, unopinionated approach, speed, efficiency, vibrant community, ease of learning, modularity, and robust routing and middleware support. Its modular architecture and extensive plugin ecosystem allow developers to tailor their applications to meet specific requirements.

Simple Express Server

```
//Requiring Express in our file
const express = require('express');

//Creating instance of express
const app = express();

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

//Defining port
const port = 3000;

//Starting express by calling listen() method
app.listen(port, () => {
    console.log(`Server is listening on port ${port}`);
});
```

Request Object:
Represents the HTTP request and includes properties for query strings, parameters, body, headers, etc.

Common properties/methods:
1. req.params: Access route parameters.
2. req.query: Access query string parameters.
3. req.body: Access parsed request body.
4. req.headers: Access HTTP headers.
5. req.cookies: Access cookies sent by the client.
6. req.get(header): Retrieve value of a specific HTTP header.
7. req.ip: IP address of the client.
8. req.path: Path part of the request URL.
9. req.method: HTTP method used in the request.

Response Object:
Used to send responses to the client.

Common methods:
1. **res.send():** Send a string, buffer, JSON object, or HTML file.
2. **res.json():** Send a JSON response.
3. **res.render():** Render an HTML view using a template engine.
4. **res.status():** Set the status code of the response.
5. **res.redirect():** Redirect the client to a different URL.
6. **res.download():** Send a file as a download to the client.
7. **res.sendFile():** Send a file as a response to the client.
8. **res.cookie():** Set a cookie in the response.
9. **res.clearCookie():** Clear a cookie in the response.

Express Routes

Express applications define routes as a way to respond to client requests for specific URLs and HTTP methods. Routes consist of an HTTP method and a URL pattern, which determine which handler function should be executed to generate a response. Express.js provides a simple and expressive way to define routes for your application.

Basic route handling involves using the app.METHOD() functions, where METHOD is an HTTP request method in lowercase. Route parameters can be specified in the route path using colon notation (:). Route handlers are callback functions that execute when a route is matched, accessing the request (req) and response (res) objects.

Route chaining
Route chaining allows you to chain route handlers together using the app.route() method, organizing and simplifying code.

Middleware in express js

Middleware functions have access to the request (req), response (res), and next middleware function in the application's request-response cycle. They can modify the request and response objects, terminate the request-response cycle, or call the next middleware function in the stack.

Some commonly used built-in middleware in Express are -
1. **express.json() -** This middleware is used to parse JSON data in the request body. It makes the JSON data available as req.body.

2. **express.urlencoded() -** This middleware is used to parse URL-encoded data in the request body. It parses data sent via HTML forms.

3. **express.static() -** This middleware serves static files, such as HTML, CSS, and JavaScript, from a specified directory. It simplifies the serving of static assets.

1. **express.raw() -** This middleware is used to parse raw request data and make it available in req.body. It's often used for handling binary data.
2. **express.text() -** This middleware parses text data from the request body and makes it available in req.body.
3. **express.Router() -** While not exactly middleware, it allows you to create modular route handlers and is often included in lists of built-in middleware. You can define routes and route handlers using the Router object.

Error Handling in Express

Error handling is essential for building robust web applications. Express.js provides built-in error-handling middleware functions that catch errors during the execution of route handlers and middleware functions. These middleware functions are defined with four arguments (err, req, res, next) and are registered using the app.use() method.

Templating Engine

A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

Some popular template engines that work with Express are Pug, Mustache, and EJS. The Express application generator uses Jade as its default, but it also supports several others.
To render template files, set the following application setting properties, set in app.js in the default app created by the generator:

- views, the directory where the template files are located. Eg: app.set('views', './views'). This defaults to the views directory in the application root directory.
- view engine, the template engine to use. For example, to use the Pug template engine: app.set('view engine', 'pug').

Then install the corresponding template engine npm package; for example to install Pug:

```
npm install pug --save
```

After the view engine is set, you don't have to specify the engine or load the template engine module in your app; Express loads the module internally, as shown below (for the above example).

```
app.set('view engine', 'pug')
```

Create a Pug template file named index.pug in the views directory, with the following content:

```
html
  head
    title= title
  body
    h1= message
```

Then create a route to render the index.pug file. If the view engine property is not set, you must specify the extension of the view file. Otherwise, you can omit it.

```
app.get('/', (req, res) => {
  res.render('index', { title: 'Hey', message: 'Hello
there!' })
})
```

When you make a request to the home page, the index.pug file will be rendered as HTML.
Note: The view engine cache does not cache the contents of the template's output, only the underlying template itself. The view is still re-rendered with every request even when the cache is on.