
node
JS
express

An Express History on Express

Express

- Express is a web framework for Node.js.
- It is self-described as a "fast, unopinionated, minimalist web framework."
- Express provides enough functionality to make implementing HTTP requests and responses easy.
- It does not obscure the underlying native HTTP functionality, allowing developers to have greater control.

A History of Express

- Express was initially created in early 2010.
- Originally, it aimed to provide a small and robust server for making application routing and template rendering easier.
- Over time, Express evolved into a pluggable framework capable of handling a variety of use cases.
- However, the central philosophy of being fast, unopinionated, and minimalist remains.

Routes: Which way do we go?

Routing

- Routing methods in Express are generally in the form router.METHOD(PATH, HANDLER).
- METHOD defines the kind of request (e.g., GET, POST, etc.) that the route will handle.
- PATH defines the URL path that the route will respond to.
- HANDLER is a function that holds the logic to handle the request.

```
app.get('/users', (req, res) => {
  // Logic to handle GET request to '/users' path
});

app.post('/login', (req, res) => {
  // Logic to handle POST request to '/login' path
});
```

Handlers in Express Routing

```
app.get('/users', (req, res) => {
  // req: Request object
  // res: Response object
  // Logic to handle the request
});

app.post('/login', (req, res, next) => {
  // req: Request object
  // res: Response object
  // next: Next middleware function
  // Logic to handle the request
});
```

Handlers in Express are functions applied to the given route that can take up to three parameters:

1. request: An object housing all of the request information, including:
    a. URL that was matched
    b. Body of the request (if applicable)
    c. Parameters for a request (if any)
    d. Query string information (if any)

2. response: An object containing methods for responding to a request.

3. next (optional): Instead of responding to the request directly, the next parameter allows passing control to the next middleware function in the stack.

Routing Example

```
// Include Express
var express = require('express');

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

// Define a route on `/hello/world`
app.get('/hello/world', function(request, response) {
  console.log('Got request for "/hello/world"');
  response.send('Hello there!');
});

// Have the application listen on a specific port
app.listen(3000, function() {
  console.log('Example app listening on port 3000!');
});
```

Response Status

```
// Returning HTTP statuses in Express

// Define a route on '/hello'
app.get('/hello', function(request, response) {
  // ... route logic ...
});

// If no routes are matched, return a 404
app.get('*', function(request, response) {
  response.status(404).send('Uh oh! Page not found!');
});
```

- HTTP statuses can be returned in Express using the response.status(status) method.
- In the example code, the app.get('*') route is used to handle unmatched routes and returns a 404 status with a corresponding message.

Parameterized Routing

```
// Parameterized route with :id variable
app.get('/users/:id', function(request, response) {
  const userId = request.params.id;
  // ... route logic ...
});

// Parameterized route with multiple variables
app.get('/users/:id/posts/:postId', function(request, response) {
  const userId = request.params.id;
  const postId = request.params.postId;
  // ... route logic ...
});
```

- Parameterized routes contain one or more parts that are variables.
- In Express, these variables are denoted on the route with a colon (:).
- The variables in the route are put onto the params member of the request.
- In the example code, the :id and :postId are variables in the route.
- The values of these variables can be accessed through the params member of the request object.

Patterns in Express

Routes support the following patterns:

- Wildcards (*): Matches any character or group of characters in a route segment.
- Character repetition (+): Matches one or more occurrences of a character or group of characters.
- Optional characters (a?): Matches a character or group of characters that are optional.
- Optional group ((ab)?): Matches an optional group of characters, such as "ab" or an empty string.

Patterns in Express - Example

```
// Wildcard route
app.get('/users/*', function(request, response) {
  // ... route logic ...
});

// Character repetition route
app.get('/repeated+', function(request, response) {
  // ... route logic ...
});

// Optional characters route
app.get('/optional(a)?', function(request, response) {
  // ... route logic ...
});

// Optional group route
app.get('/optionalgroup(ab)?', function(request, response) {
  // ... route logic ...
});
```

In the example code, various pattern-based routes are demonstrated using wildcards, character repetition, optional characters, and optional groups.

Express and path-to-regexp Library

- Express relies on a library called "path-to-regexp" for handling route patterns.
- Groups directly after a leading slash in a route pattern do not work as expected.
- Using parentheses to define a group in this case will throw an error, complaining about an invalid regular expression group.

```
// Throws an error due to invalid regular expression group
app.get('/(hel)?lo', function(req, res) {
  // ... route logic ...
});
```

- In the example code above, the use of parentheses to define a group directly after a leading slash in the route pattern will result in an error.

Aliasing Routes in Express

-Express allows you to supply an array of paths for a single handler.

- This feature makes it easy to alias one route for another if they perform the same action.

```
app.get(['/hello', '/hi', '/hola'], function(request, response) {
  // ... route logic ...
});
```

- In the example code, the handler is associated with multiple paths: '/hello', '/hi', and '/hola'.
- All these paths will trigger the same route handler, allowing for route aliasing.

Uh Oh We Have a Problem - Route Conflicts in Express

- Problem:
  - Consider the following routes:

```
app.get('/hello/:name', function(request, response) {
  // ... route logic ...
});

app.get('/hello/world', function(request, response) {
  // ... route logic ...
});
```

What is our issue here?

Uh Oh We Have a Problem - Route Conflicts in Express

- Problem:
  - Consider the following routes:

```
app.get('/hello/:name', function(request, response) {
  // ... route logic ...
});

app.get('/hello/world', function(request, response) {
  // ... route logic ...
});
```

Issue: When calling /hello/world, the first route (/hello/:name) is executed instead of the intended route (/hello/world).

Route Conflicts in Express - Solution

- Solution:
  - Rearrange the order of the route handlers:

```
app.get('/hello/world', function(request, response) {
  // ... route logic for '/hello/world' ...
});

app.get('/hello/:name', function(request, response) {
  // ... route logic for '/hello/:name' ...
});
```

- By placing the /hello/world route handler before the /hello/:name route handler, the correct route will be matched and executed.
- Important Note:
  - The order of route handlers matters in Express. The first matching route handler is executed, so it's essential to arrange them accordingly.

Exercise: Routinng

1. Route: Apple or Ale
    a. Accepts 'apple' or 'ale' as input
    b. Returns "Apple or Ale?"
2. Route: Whoa!
    a. Accepts the word 'whoa' with an arbitrary number of 'os' and 'as'
    b. Returns "I know, right?!"
3. Route: User Greeting
    a. Takes a first name and last name as parameters
    b. Returns a greeting for that user
4. Route: Reverse Word
    a. Takes a word as a parameter
    b. Returns the word reversed
5. Default Route
    a. Executes if none of the above routes match
    b. Handles any other request or inpu

Query Strings

- Query strings provide extra information at the end of a URL.
- Information is in key-value pairs.
- Express puts this information into request.query.
- Example:
  - URL: http://my-cool-site.com/page?foo=bar&baz=quux
  - request.query:
    - { ‘foo’:’bar’, ‘baz’: ‘quux’ }
- Note:
  - Query strings allow passing additional information in the URL, which can be accessed through request.query in Express.

Exercise: Query String

- Add a route to the previous example that returns a friendly greeting for firstName and lastName query parameters on the route /hello.

Middleware

- Middleware adds useful functions after the request is received but before the route is handled.
- Common use cases include logging, authentication, and parsing.
- Middleware improves code organization, reusability, and maintainability.
- It allows for separation of concerns and modular handling of common tasks.
- Middleware functions can be chained together to control the flow of operations.

Middleware Example

```
// Import required modules
const express = require('express');

// Create an instance of Express
const app = express();

// Logging middleware
app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
});

// Route handler
app.get('/hello', (req, res) => {
  res.send('Hello, World!');
});

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

In this Express example:

- We have a logging middleware and a route handler.
- The logging middleware, defined using app.use(), logs the timestamp, request method, and URL for each incoming request.
- The next() function passes control to the next middleware or route handler.
- When a request is made to /hello, the route handler responds with "Hello, World!".
- Logging middleware helps with debugging, monitoring, and tracking incoming requests in the console.

By implementing logging middleware, you can easily track and monitor the details of incoming requests in your Express application.

Exercise - Middleware

Create middleware for our earlier examples to make a log of incoming requests. Include the original route and a timestamp. Have the log write to a filed call “log.txt” in your project directory.

Response Encoding - JSON

If you want to send JSON data as a response in Express, you can use res.json() method, which automatically sets the Content-Type header to application/json. However, if you want to be explicit, you can use the following approach:

```
app.get('/some/route', function(req, res) {
  var obj = { 'hello': 'there' };
  // Get the default JSON replacer from Express (defaults to undefined)
  var replacer = app.get('json replacer');
  // Get the default spacing for JSON stringification (defaults to undefined)
  var spaces = app.get('json spaces');
  // Stringify your response object
  var jsonString = JSON.stringify(obj, replacer, spaces);

  res.set('Content-Type', 'application/json'); // Optional: Set Content-Type header explicitly
  res.send(jsonString);
});
```

Response Coding - HTML

If you want to explicitly set the Content-Type to text/html and send an HTML response using res.send() in Express, you can use the following code:

```
app.get('/page', function(req, res) {
  res.set('Content-Type', 'text/html');
  res.send('<h1>Hello!</h1>');
});
```

Response Encoding - Plain Text

To send plaintext as a response in Express and explicitly set the Content-Type to text/plain. This does not happen automatically and must explicitly be set like so:

```
app.get('/page', function(req, res) {
  res.set('Content-Type', 'text/plain');
  // This will print the string literal to the browser instead of rendering
  res.send('<h1>Hello!</h1>');
});
```