Express JS

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

```
app.METHOD(PATH, HANDLER)
```

Where:

- `app` is an instance of `express`.
- `METHOD` is an HTTP request method, in lowercase.
- `PATH` is a path on the server.
- `HANDLER` is the function executed when the route is matched.

The following examples illustrate defining simple routes.

Respond with `Hello World!` on the homepage:

```
app.get('/', function (req, res) {
  res.send('Hello World!')
})
```

Respond to POST request on the root route (`/`), the application's home page:

```
app.post('/', function (req, res) {
  res.send('Got a POST request')
})
```

Respond to a PUT request to the `/user` route:

```
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user')
})
```

Respond to a DELETE request to the `/user` route:

```
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user')
})
```

Server using ExpressJs

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

app.get('/', function(req,res){
     console.log('welcome');
     res.send('welcome to home page');
});

app.get('/students', function(req,res){
     const id = req.query.id;
     if (id === undefined){
     res.send('welcome to students page');
     } else{
          res.send('welcome to students page'+id);
     }

});

/*
app.get('/students/:qid',function(req,res){
     const id  = req.params.qid;
     res.send('student'+id);
});
*/

app.use('/students/:id', function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
})

app.listen(7000);
```

Middleware

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`.

Middleware functions can perform the following tasks:

- Execute any code.

- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

```
var express = require('express')
var app = express()

app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
```

This example shows a middleware function mounted on the `/user/:id` path. The function is executed for any type of HTTP request on the `/user/:id` path.

```
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
```

This example shows a route and its handler function (middleware system). The function handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', function (req, res, next) {
  res.send('USER')
})
```

Here is an example of loading a series of middleware functions at a mount point, with a mount path. It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the `/user/:id` path.

```
app.use('/user/:id', function (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}, function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
```

Error Handling

Express comes with a built-in error handler that takes care of any errors that might be encountered in the app. This default error-handling middleware function is added at the end of the middleware function stack.

If you pass an error to `next()` and you do not handle it in a custom error handler, it will be handled by the built-in error handler; the error will be written to the client with the stack trace.

If you call `next()` with an error after you have started writing the response (for example, if you encounter an error while streaming the response to the client) the Express default error handler closes the connection and fails the request.

Syntax for built-in error handler

```
function errorHandler (err, req, res, next) {


}
```

A program for error handling using built-in middleware

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

 // route handlers

  app.get('/', (req,res) => {
       var err = new Error("went wrong");
       next(err);
 });

 // other route handlers


 // an error handling middleware
 // at the end
 app.use(function(err,req,res,next){
      res.send("something went wrong");
 });
```