**Introduction to express JS.**

Express is a popular node JS framework. It is very developer friendly. We can require express module. It gives use a function then we need to call this function. After calling this function it gives us a instance object. We can now work with this object like in raw node js we use server.listen(port). Similarly we can use app.listen(port). Then server will be run using express js.

**Row node server:**

const http = require("http");

// here server is event emiter

const server = http.createServer((req, res) => {

  if (req.url === "/") res.writeHead("this is main port");

  if (req.url === "/about") res.write("this is about page");

});

//when a new connection is established means server is hitting

// then this "connection" event will be   fired

server.on("connection", () => {

  console.log("a new connection established");

});

// Note: rather doing connection events. we can pass a callback function to the createServer for each connection request for    each connection request. it will be fired when a new connection is established

// here is ther server register listener

server.listen(4000, () => {

  console.log("server listening on port ", 4000);

});

In raw node project, all the request hit into a callback inside createServer() function. Then check the request method and route. Based on route we perform operation. If a project have 500 api request. It check 500 if else condition then perform task based which condition meet up with route and method. It is huge task for a large application and not scalable, maintainable, extendable

**Express server demo:**

// require express module

const express = require("express");

// call the express function and get a instace of server

const app = express();

// git specific api route wit method and pass a callback function

app.get("/", (req, res) => {

  console.log("this is base api");

});

app.post("/", (req, res) => {

  console.log("this is post method with " / " route");

});

// start the express server

app.listen(4000, () => {

  console.log("Server is running on port 4000");

});

Similarly we can require express module. It gives us a function. When we call this function it gives a server instance object. Through this instance variable we can create api like app.get(‘/users’, (req, res) => { });

Here we use this instance. And define method with route. We pass ‘route’ and callback to the method. When a request is come to the express server first it check the method is called then it check with the route. If matches then it enter into the callback. Now we can do something and response data using res.send().

Express gives use scalable system, modular, maintainable, readable, developer experience etc.

Express have five components:

* Express function
* App object
* Req and res
* And routing

**Introduction to express JS.**

**express() function:**

const express = require("express");

const app = express();

app.use(express.json());

app.get("/", (req, res) => {

  res.send("this is base route");

});

app.post("/", (req, res) => {

  res.send("this is post route ");

});

app.listen(5000, () => {

  console.log("server listening on port: ", 5000);

});

**app.use() and express.json() inside app.use() function**

app.use() function use for which services we want to use my project. Like if we want to use express.json() function inside app.use() function. Our express server will be able to accept ‘application/json’ data from the client. It will not accept the plain text from client request. By using express.json() function inside app.use() method. It enable server to accept the json data from the client body. It parse the json data from the client body. If we don’t use it. Our express will not be able to accept the json data from the client body. Express.json() method return something that used by app.use() function.

So ensure in the client side header that client added the headers as content-type: application/json. So must pass the headers object with content-type as application/json. Otherwise our express server will not be able to accept this data into server.

**app.use(express.raw())**

if we want to received stream buffer raw data we can use express.raw() function. It gives use data as stream of buffer.

If we want to get orginal data we can use toString() method.

**app.use(express.text())**

if in client side header has not any content-type. By default it pass text data to the server. But if we want to use this text data from client we need to use express.plain() function.

app.use(express.urlencoded())

if we pass urlencoded data from the client we should pass content-type in client side as well as we need to use in the server side app.use(express.urlencoded()) middleware.

**express.static()**

if we want to make a folder as public for direct access. Means we want to make a folder which we can access directly by url like localhost:5000/public/masud.jpg.. we can access this directly by making it static folder.

Here public is a folder which we have made as static folder. Now can access the content of this folder directly by url. Anywhere in the project and in client side url

**app.use(express.static(‘dir of the folder’, options(optional)));**

role of option like suppose we have public folder and a view folder inside public folder. Express application search index.html file first by default. But if we pass option to the static function. Express go to this function first. Like localhost:5000/view/ no need to define like locahost:5000/public/view/index.html.

const userRouter = express.Router();

**app.use(userRouter);**

this express.Router() object enable us to create multiple router object with handle multiple router efficiently. We can have multiple router. We can use then router variable like

userRouter.get(), userRouter.post()… adminRouter.get() etc.

const express = require("express");

const app = express();

// app.use(express.json());

// app.use(express.text());

// app.use(express.raw());

// app.use(express.urlencoded());

// enable use to create public static folder

app.use(express.static(`${\_\_dirname}/public/`));

// enable use to create multiple router object and use it

const userRouter = express.Router();

userRouter.get("/", function (req, res) {

  console.log("this is using the router object");

});

app.get("/", (req, res) => {

  res.send("this is base route");

  console.log(req.body);

});

app.post("/", (req, res) => {

  res.send("this is post route ");

});

app.listen(5000, () => {

  console.log("server listening on port: ", 5000);

});

**Application object or app object**

**Const app = express()**

By using app.local.title = ‘this is my app tiitle’. We can access this title entire our application by using app.local.title. in other js file we need use req.app.local.title. we can define our local variable by using app.local variable

**app.mountPath : it return the parent path of this sub path**

used for sub app as mounted. We have multiple app object. This is the beauty of express application. Const app = express(); const admin = express() … and more we have. We know that object reference type. So here we call express function and express function give us a reference object. So app and admin hold different reference. Now we can define our app entry point that which sub route will be called. Suppose we have used app.use(‘/admin’, adminRoutes); when any request start with ‘/admin’ then program go to this routes only. By pass the app route.this is very important concept in express application. We can find the mounted path using console.log(admin.mountPath); it gives the mounted path

const admin = express();

app.use("/admin", admin);

admin.get("/verify", (req, res) => {

  console.log(

    "when /admin is called then the program will come to the admin object and search the sub path"

  );

  console.log(admin.mountpath);

});

**Admin.mount event : it will be fired when a sub route will be fired. This mount event gives parent app. // refers to the parent app. Means ‘app’ which is our main server.**

admin.on("mount", (parent) => {

  console.log("parent is fired", parent);

});

**App.all() function**

We can create universal route by using app.all() function. When any user hit a route with any method like get, post, put, delete or any method. This route will be hit if route is ok. App.all(‘/masud’, (req, res) => {

Console.log(‘this is masud rana from express app’);

});

If any client hit ‘/masud’ route with any method. This route will be hit. It enable us to create a universal route

app.all("/masud", (req, res) => {

  console.log("this is masud rana inside univesal route path");

});

**app.disalbed(‘case sensitive routing’) or app.enable()**

these function enable us to setting up some functionality that is by default used. We know that by default app route case insensitive but we can setting up as sensitive by using app.enable(‘case sensitive routing’) function

our route path ‘/aobut’. But any client hit ‘About’ then will not get any response. Because now our routing is case sensitive.

**App.get() with app.set()..**

It similar to the app.locals variable. If we set app.set(‘title’, ‘this is my express title’) then If we define app.get(‘title’); if any client hit this ‘title’ route. Then he will get ‘this is express title’ value. But mainly it used for setting.

**app.listen(port, callback)**

this is used for start the express server. It takes port number and a callback

**Middleware:** middleware kinds of interceptor. It intercept the main req and do something before going to the main request route. Then it back to the main request. Middleware has access req and res object. It takes req, res, next parameter. Before going to the main route request. We can do something and modify req object.

In middleware we need to call next(). It means now it will be gone to the main request route.

**app.param()**

this function is used likely middleware. It takes params and a callback req, res, next, params variable

app.param("id", (req, res, next, id) => {

  //when a id param is provided to a route. this app.param function will be called. users/234234

  //after that we can do something like we can modify the req object. after working this function. now we can shift to the main route req

  const user = {

    name: "masud rana",

    age: "23",

  };

  req.user = user;

  next(); // must call next() to go to the main route req.

  // now we access this user to the main route

});

app.get("/:id", (req, res) => {

  console.log("user comming from ", req.user.name);

  // res.send("this is base route");

  // console.log(app.locals.title);

  // console.log(req.body);

  res.send(req.user);

});

**app.path()**

app.use(‘/user’, admin);

it return the path. Const admin = express(); admin.use(‘/admin’, admin);

first when client hit ‘/user’, route it go the admin router. In admin route we define a middleware. Now it go to the admin router. Inside admin route if we console admin.path() function it gives us ‘/user/admin’.

app.use("/blog", blog);

blog.use("/admin", admin);

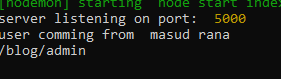
admin.get("/", (req, res) => {

  // here we can check the entire route path using admin.path()

  console.log(admin.path());

  res.send("this is route like blog/admin home route");

});



**App.route()**

When we define our api like app.get(‘user’,callback) app.post(‘/user’,callback) app.put(‘/user’,callback) app.delete(‘/user’,callback) or similar route path and different method. When route path large like ‘/user/course/section’. Each time we need to write route name with method. It is very silly hard. Instead we can use app.route() function. It takes the common route path. Then we define with dot notation with method like

app

  .route("/user")

  .get((req, res) => {

    console.log("this is get route");

  })

  .post((req, res) => {

    console.log("this is post route");

  })

  .put((req, res) => {

    console.log("this is put route");

  });

**Note:** ejs is similar to the html

**app.set(‘view engine’, ‘ejs’)** by using this. We can use ejs template. Views is the default folder for ejs. We can create sub folder to it.

**Request Object**

Client request mainly http or https request. Request represents the http request.

Request know as req ( we can receive it as other names. Everyone use this req. it is convention. Nothing else).

When a client make a request to the server. Automatically this request have been included some things. It may have query. Or client manually pass some things.

**req.baseUrl** gives the url in which route our app is mounted.

**req.orginalUrl**

**req.path** : it gives route path

**req.hostname :** it gives the domain name

**req.ip** :return client ip address

**req.method :** return the request method like get, post, put, delete etc

**req.params :** return all the params as object

**req.query :** return query as object

**req.body :** return client body object

**req.cookies** :return cookies

**req.signCookies** :return sign cookies

**req.secure :**return request is secure or not. If http return false. If https return true

**these are important property of request object.**

**Req.baseUrl**

const express = require("express");

const app = express();

app.get("/", (req, res) => {

  console.log(req.baseUrl);

  res.send("Hello world!");

});

app.listen(5000, () => {

  console.log("listening on port ", 5000);

});

If a route mounted in a sub route. Then we can see the baseUrl value. But in the main app we cannot see the baseUrl. In the above example we can’t see the baseUrl. Because it is the main app. And path is ‘/’. We know that baseUrl return something when we console it inside a sub route where sub route is mounted.

baseUrl holo eirokom j. j route a ami akhon asi. Mane ei api ta mounted hoise. Sei route ta amader k dibe baseUrl.

const express = require("express");

const app = express();

const admin = express();

app.use("/admin", admin);

admin.get("/dashboard", (req, res) => {

  console.log(

    "this is the sub app and mounted app.here we can see the baseUrl property"

  );

  console.log(req.baseUrl);

});

Here we get ‘/admin’ as answer. Because admin.get() api is mounted under the ‘/admin’ route. This is the base url of this admin route. Mainly req.baseUrl return this base url.

But req.orginalUrl return whole url. Means it return ‘/admin/dashboard’. It return route path to the mounted route or api.

**Req.cookiess**  note: cookies has in client. So if we want to access this cookies in the server side we need to parse this cookies. Express has a cookies parser module we can use it.

npm I cookie-parser. Then we can use this module by require.

**Response Object**

Response header read client. response data maybe json, html, text, form data etc.

Server can set cookies to the client response object. Res represent the http response. By convention we use res name. response has some properties. res.app return the server reference. We can set many custom properties like data. Res.headersSent return true or false when response is passed or not.

**Res.cookie** set by name and value

**res.end()** cancel the response only. We don’t pass the data to the client

**res.send()** ;send the http response. We use when need to pass data to the client. and end the response.

**Res.json()** it response the data that we pass to this function as JSONParse. It parse the response data and send to the client

**Res.status = 500** by doing this we can set status code to the client

**res.render()** : used for render view template like ejs

**res.redirect()** to help client to redirect a specific location

**res.render() with app.set(‘view engine’, ‘ejs’)**

const express = require("express");

const app = express();

app.set("view engine", "ejs");

app.get("/about", (req, res) => {

  res.render("pages/about");

});

app.listen(5000, () => console.log("listening on port 5000"));

we can pass the location of the ejs file to the render() method. Then we can see the rendered page. We can pass object to the ejs file by pass object as second parameter into render() method

receive the object property value:

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Document</title>

  </head>

  <body>

    about page: <%= title %>

  </body>

</html>

Object pass:

app.get("/about", (req, res) => {

  console.log(res.headersSent);

  res.render("pages/about", { title: "About" });

  console.log(res.headersSent);

});

**Res.end()** finished the response without passing any data. **Res.send()** finished response by passing data

**res.status(200)**

**res.cookie(‘name’, ‘value’)**

res.cookie("token", "masud rana", );

  res.end();

we can set cookie from client side. We can send options object for httpOnly, expires date etc

**res.location(‘/test’)** it setlocation into header for client

const express = require("express");

const app = express();

app.set("view engine", "ejs");

app.get("/test", (req, res) => {

  console.log("this is test");

  res.end();

});

app.get("/about", (req, res) => {

  // console.log(res.headersSent);

  // res.render("pages/about", { title: "About" });

  // console.log(res.headersSent);

  res.status(200);

  res.cookie("token", "masud rana");

  // res.location("/about"); just set location into header

  // res.location("/test");

  //res.location("/test"); redirect to this /test route

  res.redirect("/test");

  res.end();

  // res.json({

  //   name: "masud Rana",

  // });

});

app.listen(5000, () => console.log("listening on port 5000"));

**res.redirect(‘/test’)** redirect to this route path

**res.set(‘header name’, ‘value’)** res.set() method set header key value pair.

**res.get(‘title’)** it return the header property which name is ‘title’

 res.set("Title", "About");

  res.get("Title");

**Middleware**

Express js works based on middleware. Middleware is a simple function which has access to the req, res, object and one next() function. Middleware work like a cycle. When a http request is coming to the server, then server can call middleware function if needed any middleware function with the api route. Then middleware do some task and call the next() function if any middleware is persist. Or any middleware can call res object without calling any next() function.

Any middleware can through any error. If any middleware throw any error. Server will be stop there. Next middleware will not pass any request. We need to handle error

If we have not any error handing, even error is found. The program will be crashed.

Each middleware can call res object without calling the next() function.

Some important characteristic of middleware

* Executes any js code or anything
* Can change ‘req’ and ‘res’ objects
* Can end request or response cycle
* Call next middleware by next()
* Throw & catch errors

Types of middleware

* Application level middleware by using app.use()
* Router level middleware like router.use();
* Error-handling middleware
* Built-in middleware express.json(), express.raw()
* Third party middleware

Middleware is just a function which have access req, res, and next function.

app.get("/about", (req, res) => {

  res.send("about");

});

Here (req,res) => { res.send(“about”); }); one kind of middleware. Because this is a function.

If we want to use any middleware we should notify the express app. We need to use app.use() function by passing middleware function. When we will use app.use(myMiddleware); when any api is called. First program run this middleware.

const express = require("express");

const app = express();

const myMiddleware = (req, res, next) => {

  console.log("i am logged in 1");

  next();

};

const myMiddleware2 = (req, res, next) => {

  console.log("i am logged in 2");

  next();

};

app.use(myMiddleware);

app.use(myMiddleware2);

app.get("/about", (req, res) => {

  res.send("about");

});

app.listen(5000, () => console.log("listening on port 5000"));

even I hit the ‘/about’ route. First it go to the middleware. After executing middleware we need to call next() function otherwise it stuck on this middleware. After calling next() inside both middleware it come to the ‘/about’ route.

* **Above example is application level middleware.** Where we directly write middleware inside our application.
* const express = require("express");
* const app = express();
* const admin = express.Router();
* const myMiddleware = (req, res, next) => {
* console.log("i am logged in 1");
* next();
* };
* const myMiddleware2 = (req, res, next) => {
* console.log("i am logged in 2");
* next();
* };
* // middleware
* // app.use(myMiddleware); // application level middleware
* // app.use(myMiddleware2);
* app.use("/admin", admin); //router level middleware
* admin.get("/dashboard", (req, res, next) => {
* console.log("this is called only when /admin/dashboard is called");
* });
* app.get("/about", (req, res) => {
* res.send("about");
* });
* **app.listen(5000, () => console.log("listening on port 5000"));**
* **Above example is router level middleware**

Below is **error handling middleware**

admin.get("/dashboard", (req, res, next) => {

  console.log("this is called only when /admin/dashboard is called");

  // throw a error to the error handling middleware

  throw new Error("this is custom error masud rana passed");

  // you must catch this error by error handling middleware

  // otherwise your program will crash  and you won't be able to continue.

});

app.get("/about", (req, res) => {

  res.send("about");

});

const errorHandler = (err, req, res, next) => {

  console.log("without error this middleware is called");

};

app.use(errorHandler);

app.listen(5000, () => console.log("listening on port 5000"));

when any error is thrown to the server, then program executing will be gone to the error handler middleware. Only when we will throw any error then this middleware will be called.

Without any error. This middleware will not call. This middleware takes four parameters err, req, res, next. First it takes ‘err’ parameter. Express server mainly know which middleware need to call, it find by middleware parameter. When express server see that a function has 4 parameter with err object. Then express understand that this is the error handling middleware. this middleware occurs when any error is thrown to the server. So if want to use this error handling we must provide error handling.