Express JS

Introduction :—

Express is a framework of Node JS that is used to write APIs, web-application and Server side programming. Express is minimal and flexible node Js web app framework, designed to develop webs and APIs faster.

Command to install express globally is :—

npm install express —save

---

const express = require('express')

different method of express is

1. **express.json() :—** This is a built in middle ware function in express, passes incoming requests with JSON payloads and is based on body-parser. return middle ware that only parses json and only looks at request where the content type header matches the type option. we can get the parsed data using **request.body** property.

   ```
   json = express.JSON()
   ```

2. **express.raw() :—** returns middle ware that parses all bodies as a buffer and only looks at the requests where the content type header matches the type option. A new **body buffer** containing the parsed data is populated on the request object after the middle ware **request.body** or an {} if no body was parsed.

   ```
   var raw = express.raw()
   ```

3. **express.Router() :—** creates a new router object

```
var router = express.Router([options])
```

4. **express.static() :—** used to serve all the static files like images, css, HTML content and more. The **root** argument specifies the root directory from which to serve static assets.

```
express.static(root,[options,])
```

5. **express.urlencoded() :—** it is a built in middleware in express and parses the incoming request with urlencoded payloads and is based upon body-parser.

```
express.urlencoded([options])
```

const app = express()

different methods of app are

1. **app.get() :—** app.get(name) . returns the value of name app setting, where name is one of the string in the app setting table.

```
app.get('title')
//undefined

app.set("title","My site")
app.get('title')
//==> My site
```

   app.get(route, cb) .. routes HTTP GET requests to the specified path with specified callback functions

```
app.get('/', (res,req)=>{
res.send("connection stablished");
})
```

2. **app.post() :—** Routes HTTP POST requests to the specified path with the specified callback functions.

```
app.post('/post',(req,res)=>{
  res.send("data posted")
})
```

3. **app.put() :—** Routes HTTP PUT request to the specified path with the specified callback functions.

```
app.put('/put',(req,res)=>{
  res.send("data updated")
})
```

4. **app.delete() :—** Routes HTTP DELETE request to the specified path with callback function.

```
app.delete('/delete',(req,res)=>{
 res.send("Data deleted")
})
```

5. **app.set(name,value) :—** Assigns setting name to value. you may store any value you want. but certain names can be used to configure the behaviour of the server. it indicates configuration within our server file, which must be place before define any route or middleware.

```
app.set('title','App')
app.get('title')
```

6. **app.use() :—** mounts the specified middle ware function or functions at the specified path, the middle ware functions is executed when the base of the request is passed. if we omits the path then the middleware will run for every routes defined below that middle ware definitions

```
app.use('path', middleware)
//RUNS FOR SPECIFIED PATH
```

```
app.use('path', [middleware1 , middleware2,..])
//RUNS FOR ALL THE ROUTES
app.use(middleware)
app.use( [middleware1 , middleware2,..])
```

7. **app.route() :—** returns an instance of a single routes, which you can then use to handle HTTP verbs with middleware. use app.route() to avoid duplicate route names. means mapping the URL to specific function that will handle the logic for that URL.

```
app.route('/home/data').get((req,res)=>{
res.send("read")
}).post((req,res)=>{
res.send("added")
}).put((req,res)=>{
res.send("updated")
}).delete((req,res)=>{
res.send("deleted")
})
```

8. **app.all() :—** This method is like the standard methods, except it matches all HTTP verbs. it is used for all types of HTTP request.

```
app.all('path',(req,res,next)=>{
console.log("reading values using all methods")
})
```

9. **app.listen() :—** it binds and listens for connection on the specified host and port. This method is identical to node's http.server.listen() If port is omitted or is 0, the operating system will assign an arbitrary unused port, which is useful for cases like automated tasks.

```
app.listen(portNumber)

app.listen(portNumber,()=>{
console.log("Listening to port "+portNumber)
})
```

app.get(route, (request, respond)⇒{

//functionality of that route.

})

different methods of respond

1. **respond.send() :—** sends the HTTP response. This method perform many useful tasks for simple non-streaming responses;, for example, it automatically assigns Content-Length HTTP response header field and provides automatic HEAD and HTTP cache freshness support.

```
res.send(Buffer.from("message"))
res.send({some : "JSON"})
res.send('<p>Some HTML </p>')
res.status(200).send("successful")
res.status(404).send("Not found")
```

2. **respond.status() :—** sets the HTTP status for the response. it is a chainable alias of node's response.statusCode.

```
res.status(code)
```

3. **respond.sendFile() :—** this basically transfers the file at the given path and it sets the Content-Type response HTTP header field based on the filename extension. The method invokes the callback function `fn(err)` when the transfer is complete or when an error occurs. If the callback function is specified and an error occurs, the callback function must explicitly handle the response process either by ending the request-response cycle, or by passing control to the next route.

```
app.get('/file/:name', function (req, res, next) {
  var options = {
    root: path.join(__dirname, 'public'),
    dotfiles: 'deny',
    headers: {
      'x-timestamp': Date.now(),
      'x-sent': true
    }
  }
```

```
var fileName = req.params.name
res.sendFile(fileName, options, function (err) {
  if (err) {
    next(err)
  } else {
    console.log('Sent:', fileName)
  }
})
```

4. **respond.json() :—** sends a JSON response. This method sends a response that is the parameter converted to a JSON string using JSON.stringify(). The parameter can be any JSON type, including object, array, string, Boolean, number, or null, and you can also use it to convert other values to JSON.

```
res.json(null)
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```

5. **respond.jsonp() :—** sends a JSON response with JSONP support. This method is identical to `res.json()`, except that it opts-in to JSONP callback support.

```
res.jsonp(null)
// => callback(null)

res.jsonp({ user: 'tobi' })
// => callback({ "user": "tobi" })

res.status(500).jsonp({ error: 'message' })
// => callback({ "error": "message" })
```

6. **respond.render() :—** Renders a view and sends the rendered HTML string to the client. The view argument is a string that is the file path of the view file to render. This can be an absolute path or a path relative to the view setting. if the path doesn’t contain a file extension, then the view engine setting determines the file extension. if the path does contain a file extension, then express will load the module for the specified template engine via require and render it using the loaded module’s__express function.

```
// send the rendered view to the client
res.render('index')

// if a callback is specified, the rendered HTML string has to be sent explicitly
res.render('index', function (err, html) {
  res.send(html)
})

// pass a local variable to the view
res.render('user', { name: 'Tobi' }, function (err, html) {
  // ...
})
```

7. **respond.get() :—** Returns the HTTP response header specified by field. The match is case-sensitive.

```
res.get("content type")
```

8. **respond.end() :—** Ends the response process. use to quickly end the response without any data. if you need to respond with data, instead use method such as send() and json().

```
res.end("MESSAGE")
```

9. **respond.sendStatus() :—** sets a response HTTP status code to statusCode and sends the registered status message as the text response body. if an unknown status code is specified, the response body will just be the code number.

```
res.sendStatus(400)
```

10. **respond.download() :—** Transfers the file at path as an attachment. Typically browsers will prompt the user for download. By default content-disposition header “filename = ” parameter is derived from the path argument but can be overridden with filename parameter.

```
res.download("path to the file")
```

```
res.download("path to the file 1","path to the file 2",function (err){
if(err){
    // handled it
 }else{
    // handle it
 }
});
```

11. **respond.location() :—** Sets the response `Location` HTTP header to the specified `path` parameter. A `path` value of “back” has a special meaning, it refers to the URL specified in the `Referer` header of the request. If the `Referer` header was not specified, it refers to “/”.

```
res.location('/foo/bar')
res.location('http://example.com')
res.location('back')
```

12. **respond.attachment() :—** Indicate to a web browser or other user agent that an outgoing file download sent in this response should be "Saved as..." rather than "Opened", and optionally specify the name for the newly downloaded file on disk. Specifically, this sets the "Content-Disposition" header of the current response to "attachment". If a `filename` is given, then the "Content-Type" will be automatically set based on the extension of the file (e.g. `.jpg` or `.html`), and the "Content-Disposition" header will be set to "filename=`filename`".

```
res.attachment([Filename])
```

13. **respond.redirect() :—** redirects to the URL derived from the specified path, with specified status, a positive integer that corresponds to an HTTP status code. if not specified, status default to 302 ‘found’.

```
res.redirect('/foo/bar')
res.redirect('http://example.com')
res.redirect(301, 'http://example.com')
res.redirect('../login')
```

14. **respond.append() :—** Appends the specified **value** to the HTTP response header field. if header is not already set, it creates the header with the specified **value.** the

value parameter can be string or an array.

```
res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>'])
res.append('Set-Cookie', 'foo=bar; Path=/; HttpOnly')
res.append('Warning', '199 Miscellaneous warning')
```

15. **respond.format() :—** Performs content-negotiation on the accept HTTP header on the request object, when present. it uses req.accepts() to select a handler for the request, based on the acceptable types ordered by their quality values. if header is not specified, the first callback is invoked. when no match found, the server responds with 406 “Not acceptable” or invokes default callback.

```
res.format({
  'text/plain': function () {
    res.send('hey')
  },

  'text/html': function () {
    res.send('<p>hey</p>')
  },

  'application/json': function () {
    res.send({ message: 'hey' })
  },

  default: function () {
    // log the request and respond with 406
    res.status(406).send('Not Acceptable')
  }
})

// canonicalized MIME types
res.format({
  text: function () {
    res.send('hey')
  },

  html: function () {
    res.send('<p>hey</p>')
  },

  json: function () {
    res.send({ message: 'hey' })
  }
})
```

16. **respond.type() :—** sets the Content-Type response header to the specified **type.** This method is pretty forgiving, but note that if type contains “/” , res.type() assumes it is a MIME type and interprets it literally.

```
res.type('.html');
res.type('html');
res.type('json');
res.type('application/json');
res.type('png');
```

17. **respond.vary() :—** Adds the field to the Vary response header, if it is not there already. The Vary header indicates which headers it’s basically used for content negotiation.

```
res.vary('User-Agent')
```

18. **respond.cookie() :—**
19. **respond.clearCookie() :—** Clears the cookie specified by name. for details about the option object.

```
res.clearCookie('Name',{path : '/admin'})
```