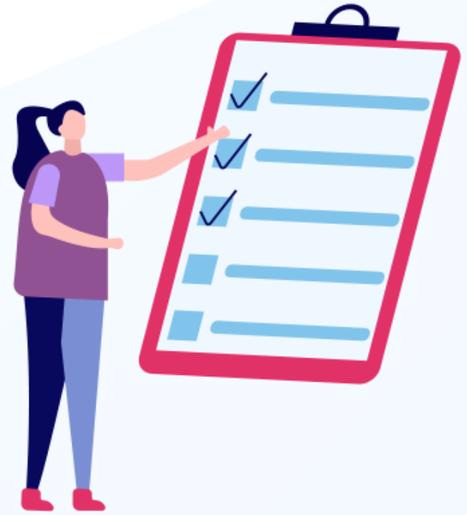
Develop a Reliable Backend with Node and Express

Configuration, Middleware, and Request Handling in Express.js

Learning Objectives

By the end of this lesson, you will be able to:

- Assess the functions app.set(), app.get(), app.enable(), and app.disable() by exploring their configurations within the application environment
- Analyze the different types of middleware to understand their roles and functionalities within the application's operational flow
- Describe the various request handlers in terms of how they manage incoming requests and execute specific functions based on request type and URL
- Analyze other attributes and methods within frameworks or libraries by considering their roles in managing data, sessions, headers, and responses for web applications



A Day in the Life of a MERN Stack Developer

Mr. Robin is working as a MERN stack developer in an organization looking to establish and maintain consistency in a product's performance and functional attributes with great database management.

To achieve all of the above, along with some additional concepts, you will be learning a few concepts in this lesson that will help create a backend for a web application framework.

Configuration

app.set()

The setting name and values are assigned using the app.set() function.

```
app.set(name, value)
```

The server's behavior can be configured using some specific identifiers.

app.get()

Using app.get() in web development associates a URL path with a callback function. When that path receives an HTTP GET request, the linked callback function executes to handle it.

```
app.get(path, callback)
```

Parameters

The following are the two types of parameters of the app.get() function :

app.enable()

The app.enable() function in frameworks like Express.js is utilized to set certain application settings to true.

```
app.enable(name)
```

This method permits the activation of distinct features or functionalities in your application by switching a boolean value linked to a specific setting or identifier.

app.disable()

The boolean setting name is set to false using the app.disable() function.

```
app.disable(name)
```

Assign false boolean values to several Express.js system parameters using the app.disable(name) function.

app.enabled()

The status of the setting name property is checked by the app.enabled() method.

```
app.enabled(name)
```

It determines whether a setting name's value is True or not.

app.disabled()

The boolean values of the setting name are returned by the app.disabled() function.

```
app.disabled(name)
```

If the specified setting is disabled, it returns True; otherwise, it returns False.

Configuration Commands

Problem Statement:

Duration: 20 min.

You have been assigned a task to demonstrate the use of configuration commands.

ASSISTED PRACTICE

Assisted Practice: Guidelines

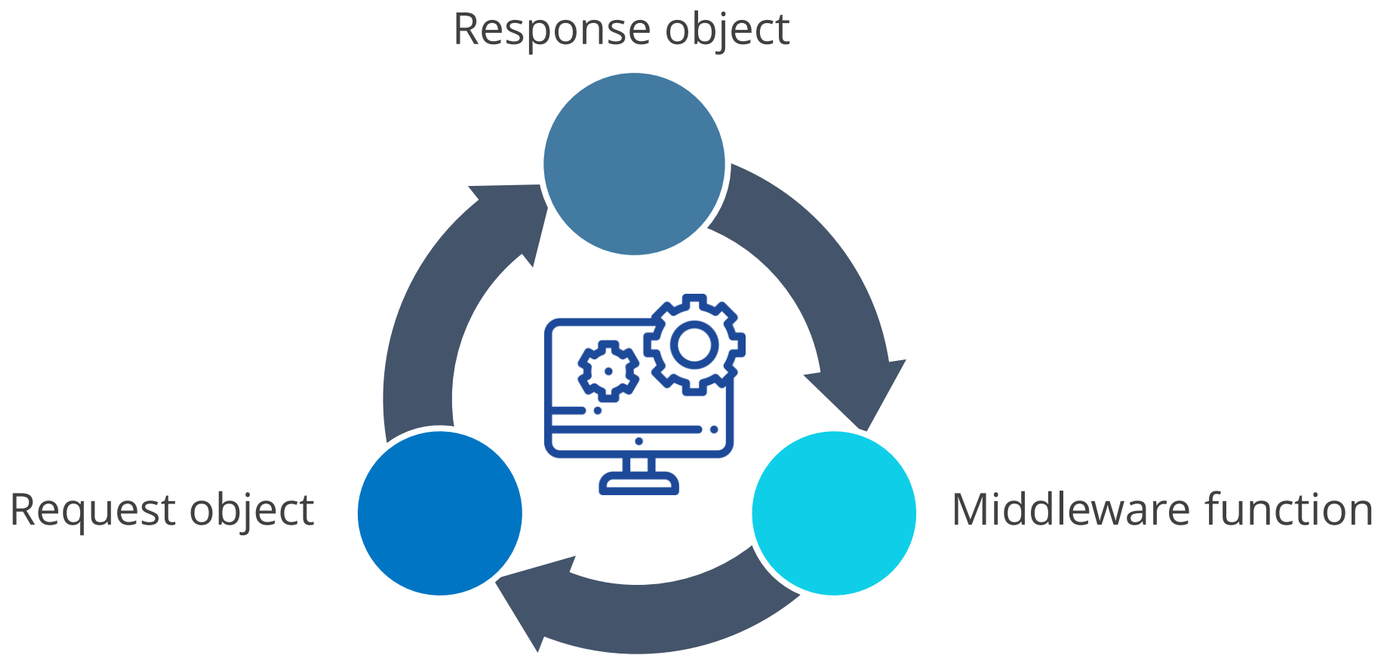
Steps to be followed:

1. Set up the working directory
2. Use app.set() and app.get() methods in Express.js
3. Use app.enable() and app.disable() methods in Express.js
4. Use app.enabled() and app.disabled() methods in Express.js

Types of Middleware

Middleware

In Express, middleware refers to functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle.

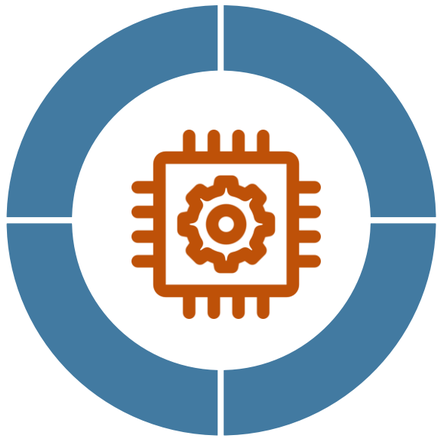


A significant portion of an Express application involves the invocation of middleware functions.

Middleware

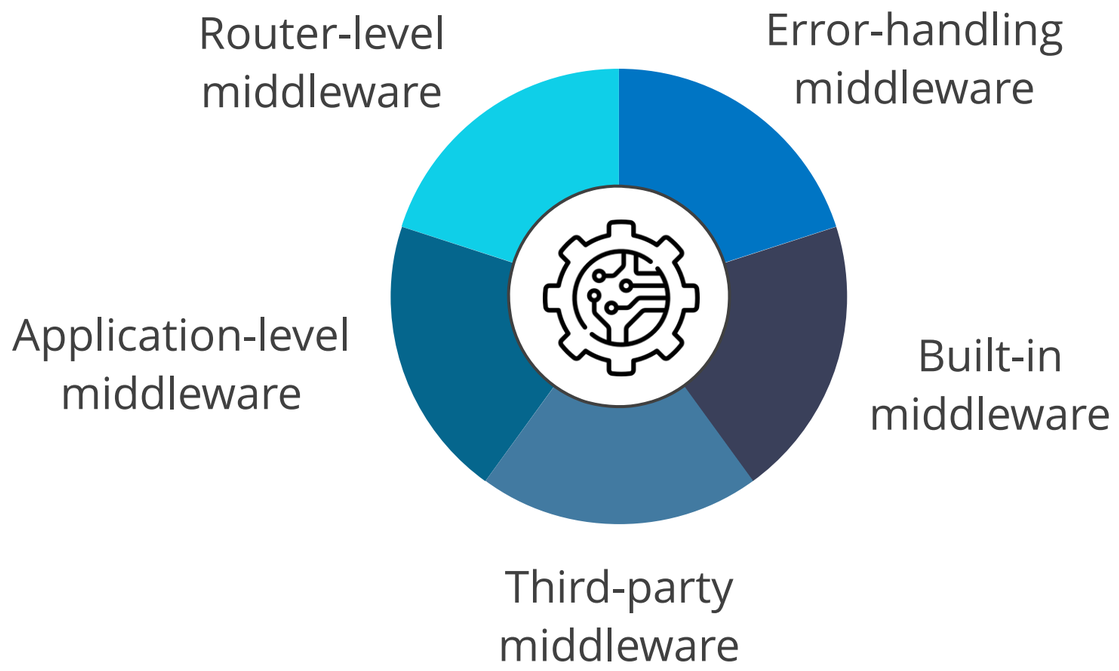
Middleware functions can carry out the following tasks:

- Change the request and response objects
- Stop the cycle of requests and responses
- Execute any program
- Make a call to the middleware function at the bottom of the stack



Types of Middleware

An Express application can utilize the following types of middleware:



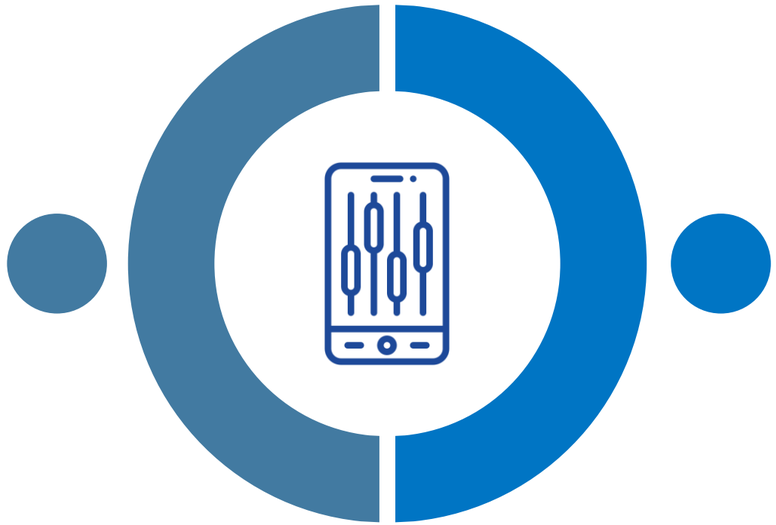
Application-Level Middleware

Application-level middleware generally runs throughout the whole application, that is, for all the routes in an app object.

Application-Level Middleware

Bind application-level middleware to an instance of the app object, which represents the Express application, using the following two functions:

app.use()



app.method()

In app.method(), method refers to the HTTP method handled by the middleware function.

Application-Level Middleware

Example of a middleware function that does not have a mount path:

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

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

When the app receives a request, the function gets called.

Application-Level Middleware

A middleware function is mounted on the **/user/:id** path.

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

For any type of HTTP request on the **/user/:id** path, the function gets called.

Application-Level Middleware

The following example depicts a route and its associated handler function:

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

GET requests to the **/user/:id** path are handled by this function.

Application-Level Middleware

Following is an example of a mount path being used to load a series of middleware functions:

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

The above example depicts a middleware substack that prints the requested information to the **/user/:id** path for any HTTP request.

Application-Level Middleware

Route handlers allow the user to define multiple paths for a single path.

```
app.get('/user/:id', (req, res, next) => {
  console.log('ID:', req.params.id)
  next()
}, (req, res, next) => {
  res.send('User Info')
})

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', (req, res, next) => {
  res.send(req.params.id)
})
```

In this example, a middleware substack handles GET requests to the **/user/:id** path.

Application-Level Middleware

An array with a middleware substack that handles GET requests to the **/user/:id** path is shown below:

```
// if the user ID is 0, skip to the next route
  if (req.params.id === '0') next('route')
  // otherwise pass the control to the next middleware function in
this stack
  else next()
}, (req, res, next) => {
  // send a regular response
  res.send('regular')
})

// handler for the /user/:id path, which sends a special response
app.get('/user/:id', (req, res, next) => {
  res.send('special')
})
```

Application-Level Middleware

Call next('route') to pass control to the next route to skip the rest of the middleware functions.

The next('route') will only work in middleware functions loaded via the app.METHOD() or router.METHOD() functions.

Application-Level Middleware

An example where a middleware substack handles GET requests to the **/user/:id** path:

```
function logOriginalUrl (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}

function logMethod (req, res, next) {
  console.log('Request Type:', req.method)
  next()
}

const logStuff = [logOriginalUrl, logMethod]
app.get('/user/:id', logStuff, (req, res, next) => {
  res.send('User Info')
})
```

Router-Level Middleware

Router-level middleware function is bound to an instance of a router or express.Router().

```
const router = express.Router()
```

Use the router to load router-level middleware using the router.use() and router.METHOD() functions.

Router-Level Middleware

The code below uses router-level middleware to replicate the middleware system:

```
const express = require('express')
const app = express()
const router = express.Router()

// a middleware function with no mount path. This code is executed for
every request to the router
router.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})

// a middleware sub-stack shows request info for any type of HTTP
request to the /user/:id path
router.use('/user/:id', (req, res, next) => {
  console.log('Request URL:', req.originalUrl)
  next()
}, (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

Router-Level Middleware

```
// a middleware sub-stack that handles GET requests to the /user/:id path
router.get('/user/:id', (req, res, next) => {
  // if the user ID is 0, skip to the next router
  if (req.params.id === '0') next('route')
  // otherwise pass control to the next middleware function in this stack
  else next()
}, (req, res, next) => {
  // render a regular page
  res.render('regular')
})

// handler for the /user/:id path, which renders a special page
router.get('/user/:id', (req, res, next) => {
  console.log(req.params.id)
  res.render('special')
})

// mount the router on the app
app.use('/', router)
```

Router-Level Middleware

The next('router') returns control to the router instance, bypassing the rest of the router's middleware functions.

Router-Level Middleware

A middleware substack manages the GET requests for the **/user/:id** path.

```
const express = require('express')
const app = express()
const router = express.Router()
// predicate the router with a check and bail out when needed
router.use((req, res, next) => {
  if (!req.headers['x-auth']) return next('router')
  next()
})
router.get('/user/:id', (req, res) => {
  res.send('hello, user!')
})
// use the router and 401 anything falling through
app.use('/admin', router, (req, res) => {
  res.sendStatus(401)
})
```

Error-Handling Middleware

Error-handling middleware deals with identifying and handling errors that happen during runtime, as the name would imply.

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

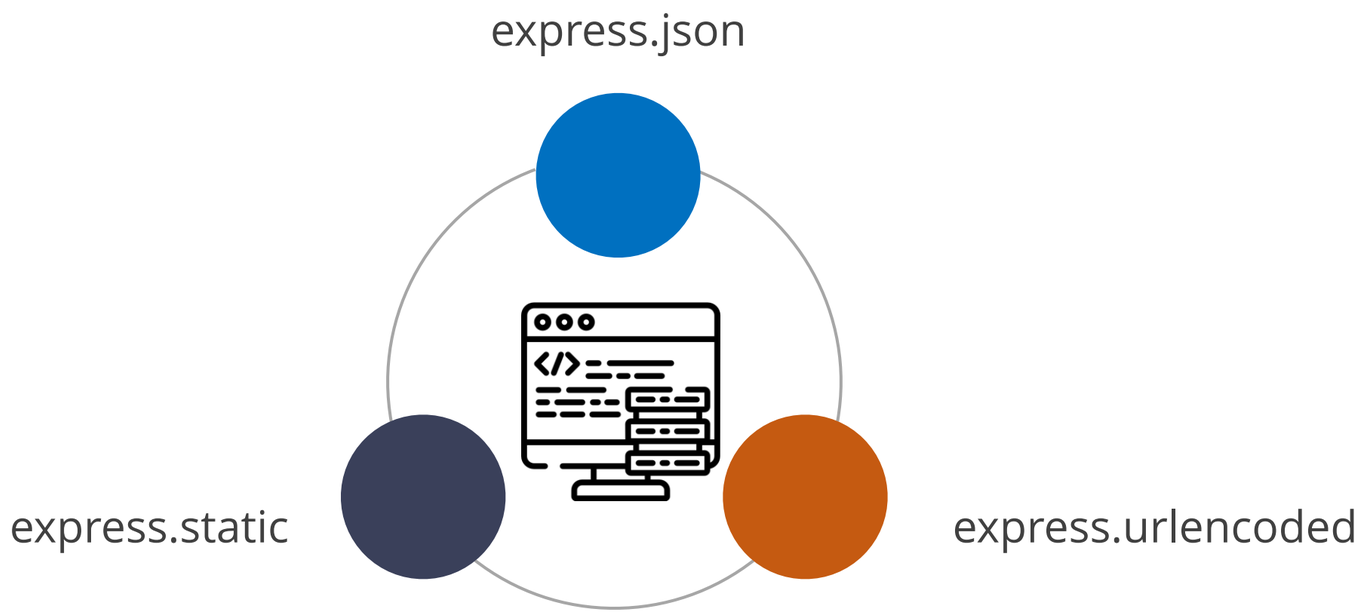
Instead of three, as indicated above, it has four arguments.

Built-in Middleware

Due to the functionalities being pre-bundled with Express, in built-in middleware there is no need to install any additional modules.

Built-in Middleware

The following are some built-in middleware functions offered by Express.js:



Third-Party Middleware

Express apps' capabilities can be increased by utilizing the third-party middleware.

These middleware can be easily integrated into an Express application to add additional functionalities, enhance security, enable logging, or perform various other tasks.

Third-Party Middleware

Use third-party middleware to enhance the functionality of Express applications.

Install the necessary Node.js modules, then load it in the application at the application or router level.

Third-Party Middleware

An illustration of how to setup and load the middleware function’s cookie-parser is shown below:

```
$ npm install cookie-parser

const express = require('express')
const app = express()
const cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

Middleware in Express.js App

Problem Statement:

Duration: 20 min.

You have been assigned a task to demonstrate the use of middleware in Express.js app.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Set up Express.js
2. Demonstrate Application-level middleware in Express.js
3. Demonstrate Router-level middleware in Express.js
4. Demonstrate Error-handling middleware in Express.js
5. Demonstrate Third-party middleware in Express.js

Request Handlers

Request Handlers

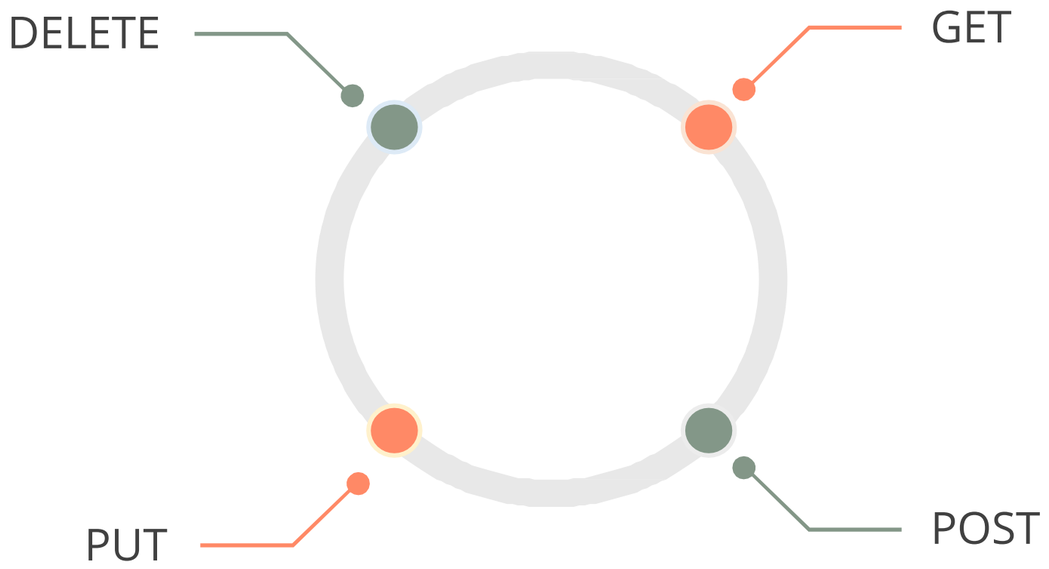
A request handler is a function that is called whenever the server receives a specific request. It accepts the request and the response as arguments:

Syntax:

```
app.method('URI Template', callback){...})
```

Request Handlers

Request handler takes a URL parameter as the first argument and a callback function as the second. The following methods are included:



Request Handlers

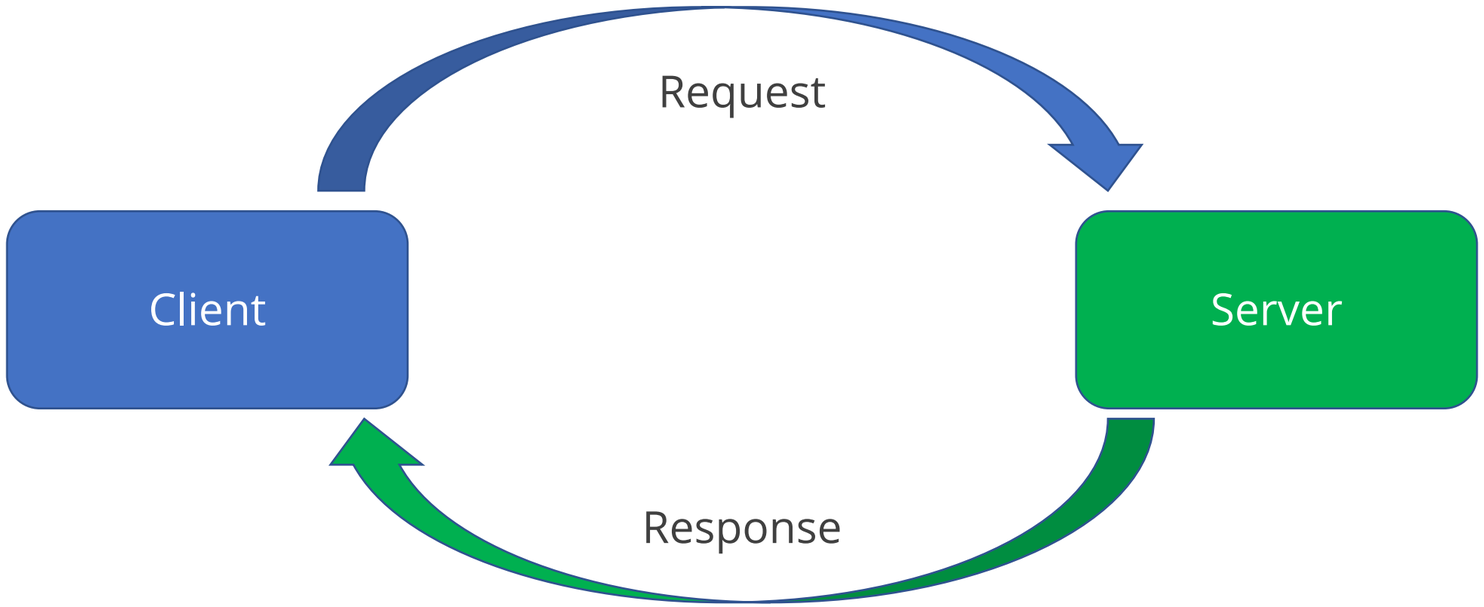
The request handler is responsible for processing the request and writing to the response object.

```
app.get('/user',function(req, res){res.send('Welcome user!');
});
```

This is analogous to the callback in the core **http.createServer()** method of Node.js.

Request

Express request is a wrapper around the Node.js http.request object.



These are some of the properties of the request object.

req.params

The req.params method retrieve the URL path and query strings from a specific request handler.

```
var express = require("express");var app = express();
app.get('/params/:type/:name/:value’,
function(req, res) {
console.log(req.params);res.end();
})
app.listen(3030);
http://localhost:3030/params/express/solo/121
```

This is essentially an array of key-value pairs.

req.body

The req.body object contains text parameters extracted from the request body.

```
var express = require('express')
var bodyParser = require('body-parser')
var app = express()
app.post('/body', function(req, res){
console.log(req.body);
res.end(JSON.stringify(req.body)+'\n');});
```

It is widely used in body-parser middleware.

req.files

The req.files method handles multipart file uploads.

```
app.post('/upload', function(req, res){
 console.log(req.files.archive);
res.end();
 })
```

It is used in conjunction with the express.bodyParser() and express.multipart() middleware.

req.route

The req.route object contains information about the currently matched route.

```
app.get('/params/:role/:name/:status', function(req, res) {
 console.log(req.route);
res.end();
});
```

req.route

The following information is present in the req.route object:

Path	URL pattern for requests
Method	The HTTP request method
Keys	Parameter list in the URL pattern
Regexp	Pattern generated for the path
Params	Object of req.Params

req.cookies

The req.cookies object is used to retrieve the request cookies.

```
req.cookies.session_id
```

The express.cookieParser() middleware enables this.

req.query

The req.query object contains the property for each query string parameter in the route.

```
req.query
```

req.query

Parameter: The req.query object doesn't take any parameters when accessed in an Express route handler.

Return Value: req.query returns an object containing the parsed query parameters from the URL**.**

req.query

Example:

```
var express = require('express');
var app = express();
var PORT = 3000;

app.get('/profile', function (req, res) {
  console.log(req.query.name);
  res.send('Profile page'); // Sending a response to the
client
});

app.listen(PORT, function(err){
    if (err) console.log(err);
    console.log("Server listening on PORT", PORT);
});
```

req.query

Open the browser and navigate to http://localhost:3000/profile?name=XYZ

```
Output:
Server listening on PORT 3000 XYZ
```

req.get()

The req.get() function returns the specified HTTP request header field.

```
req.get( field )
```

It is case-insensitive and interchangeable with the Referrer and Referrer fields.

req.get()

The HTTP request header field is specified by the field parameter.

```
var express = require('express');
var app = express();
var PORT = 3000;

app.get('/', function (req, res) {
    console.log(req.get('Content-Type'));
    res.end();
});

app.listen(PORT, function(err){
    if (err) console.log(err);
    console.log("Server listening on PORT", PORT);
});
```

Return Value: String

req.get()

Make a GET request to http://localhost:3000/ with the header 'content-type: text/plain‘.

```
Server listening on PORT 3000text/plain
```

One should get the above output on the console.

Other Attributes and Methods

Request Object Properties

Name	Purpose
req.app	It stores a reference to the Express application instance using the middleware.
req.baseurl	It identifies the URL path where a router instance was mounted.
req.body	It holds key-value pairs of data from the request body. It is undefined by default and populated when you use body-parsing middleware such as body-parser.
req.cookies	It contains cookies sent by the request when cookie-parser middleware is used.
req.fresh	It indicates that the request is fresh, as opposed to req.stale meaning the requested resource has not been modified.

Request Object Properties

Name	Purpose
req.hostname	It retrieves the hostname specified in the host HTTP header of the request, representing the hostname part of the URL.
req.ip	It provides the remote IP address of the client making the request, derived from **X-Forwarded-For** header or **req.connection.remoteAddress.**
req.ips	It contains an array of IP addresses from the **X-Forwarded-For** header, revealing a chain of client IP addresses behind proxies or load balancers.
req.originalurl	It holds the unchanged original URL string of the incoming request as received from the client, unaffected by any internal routing or modifications.

Request Object Properties

Name	Purpose
req.params	It holds route parameters extracted from the URL. It contains key-value pairs where the keys correspond to the route parameters defined in the route path.
req.path	It represents the path part of the URL from the request. It holds the path that matched the route, without any query parameters.
req.protocol	It contains the protocol used in the request, such as http or https, representing the protocol portion of the URL.
req.query	It contains the parsed query parameters from the URL. It provides access to the query string parameters sent in the URL, parsed into an object of key-value pairs for easy access.

Request Object Properties

Name	Purpose
req.route	It provides information about the currently matched route, such as its path pattern and HTTP method. It holds details specific to the matched route within the application.
req.secure	It is a boolean property indicating whether the request was made over a secure (HTTPS) connection. It evaluates to true if the connection is secure and false otherwise.
req.signedcookies:	It contains the signed cookies sent by the client in the request. Similar to **req.cookies**, but specifically holds signed cookies that are signed using a secret.
req.stale	It is a boolean property that determines if the requested resource is considered **stale** based on ETag or Last-Modified headers, indicating whether the resource has been modified since the client's last request.

Request Object Properties

Name	Purpose
req.subdomains	It contains an array of subdomains in the domain name of the request. It holds the subdomain parts parsed from the hostname.
req.xhr	It is a boolean property indicating whether the request was made using **XMLHttpRequest (XHR)**. It evaluates to true if the request is an AJAX request made via JavaScript using **XMLHttpRequest,** and false otherwise.

req.accepts(type)

The req.accepts(type) method determines whether the specified content types are acceptable.

Examples:

```
req.accepts('html');
//=>?html?
req.accepts('text/html');
// => ?text/html?
```

req.is(type)

The req.is(type) method returns true if the **Content-Type** HTTP header field of the incoming request matches the MIME type specified by the type parameter.

Examples:

```
// With Content-Type: text/html; charset=utf-8
req.is('html');
req.is('text/html');
req.is('text/*');
// => true
```

Working with Request Handlers

Problem Statement:

Duration: 10 min.

You have been assigned a task to demonstrate the use of request parameters.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Demonstrate request.query parameter in Express.js
2. Demonstrate req.params() parameter in Express.js
3. Demonstrate req.header() and req.get() parameters in Express.js

Key Takeaways

- The setting name and values are assigned using the app.set() function.
- The middleware function's response to the request uses the lowercase representation of HTTP method.
- Express applications can gain functionality by utilizing third-party middleware.
- A request handler is a function that is called whenever the server receives a specific request.

Thank You