Express JS

Express JS

- Express provides a minimal interface to build our applications. It provides us the tools that are required to build our app. It is flexible as there are numerous modules available on **npm**, which can be directly plugged into Express.
- Express was developed by **TJ Holowaychuk** and is maintained by the Node.js foundation and numerous open source contributors.

Express JS Environment

- npm install –g express //globally install
- npm install express

Advantage of ExpressJS

- Makes Node.js web application development fast and easy.
- Easy to configure and customize.
- Allows you to define routes of your application based on HTTP methods and URLs.
- Includes various middleware modules which you can use to perform additional tasks on request and response.
- Easy to integrate with different template engines like Jade, Vash, EJS etc.
- Allows you to define an error handling middleware.
- Easy to serve static files and resources of your application.
- Allows you to create REST API server.
- Easy to connect with databases such as MongoDB, Redis, MySQL,MSSQL

Creating a web server with express

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

// define routes here..

var server = app.listen(5000, function () {
    console.log('Node server is running..');
});
```

http://localhost:5000

How to Build REST API with Node.js from Scratch?

- REST or **REST**ful stands for *Representational **S**tate **T**ransfer.*
- It is an architectural style as well as an approach for communications purpose that is often used in various web services development.
- In simpler terms, it is an application program interface (API) which makes use of the HTTP requests to GET, PUT, POST and DELETE the data over WWW.
- REST architectural style helps in leveraging the lesser use of bandwidth which makes an application more suitable for the internet. It is often regarded as the "***language of the internet***".

The main functions used in any REST based architecture are:

- **GET** – Provides read-only access to a resource.
- **PUT** – Updates an existing resource
- **DELETE** – Removes a resource.
- **POST** – creates a new resource.

Principles of REST

- Well, there are six ground principles laid down by Dr. Fielding who was the one to define the REST API design in 2000. Below are the six guiding principles of REST:

Stateless

- Requests sent from a client to the server contains all the necessary information that is required to completely understand it. It can be a part of the URI, query-string parameters, body, or even headers. The URI is used for uniquely identifying the resource and the body holds the state of the requesting resource. Once the processing is done by the server, an appropriate response is sent back to the client through headers, status or response body.

Client-Server

- It has a uniform interface that separates the clients from the servers. Separating the concerns helps in improving the user interface’s portability across multiple platforms as well as enhance the scalability of the server components.

Uniform Interface

- To obtain the uniformity throughout the application, REST has defined four interface constraints which are:
- Resource identification
- Resource Manipulation using representations
- Self-descriptive messages
- Hypermedia as the engine of application state

Cacheable

- In order to provide a better performance, the applications are often made cacheable. It is done by labeling the response from the server as cacheable or non-cacheable either implicitly or explicitly. If the response is defined as cacheable, then the client cache can reuse the response data for equivalent responses in the future. It also helps in helps in preventing the reuse of the stale data.

Layered system

- The layers system architecture allows an application to be more stable by limiting the component behavior. This architecture enables load balancing and provides shared caches for promoting scalability. The layered architecture also helps in enhancing the application’s security as components in each layer cannot interact beyond the next immediate layer they are in.

Code on demand

- Code on Demand is an optional constraint and is used the least. It permits a clients code or applets to be downloaded and extended via the interface to be used within the application. In essence, it simplifies the clients by creating a smart application which doesn’t rely on its own code structure.
- Now that you know what is a REST API and what all you need to mind in order to deliver an efficient application, let’s dive deeper and see the process of building REST API using Node.js.

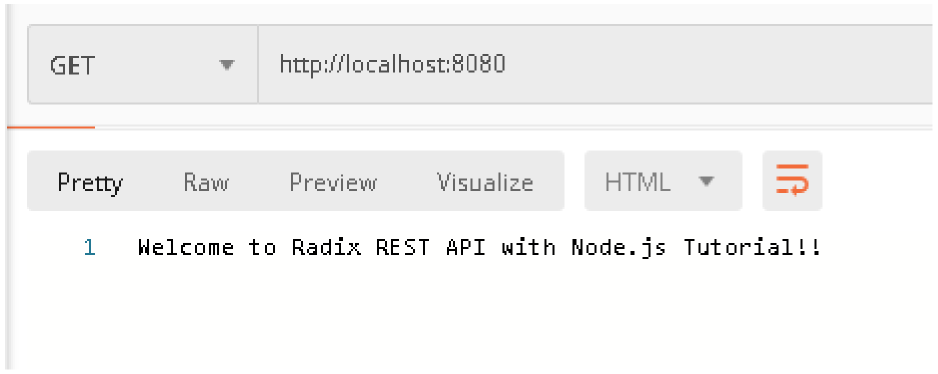
Building REST API using Node.js

- Node.js
- Express.js
- Joi

Sample Script

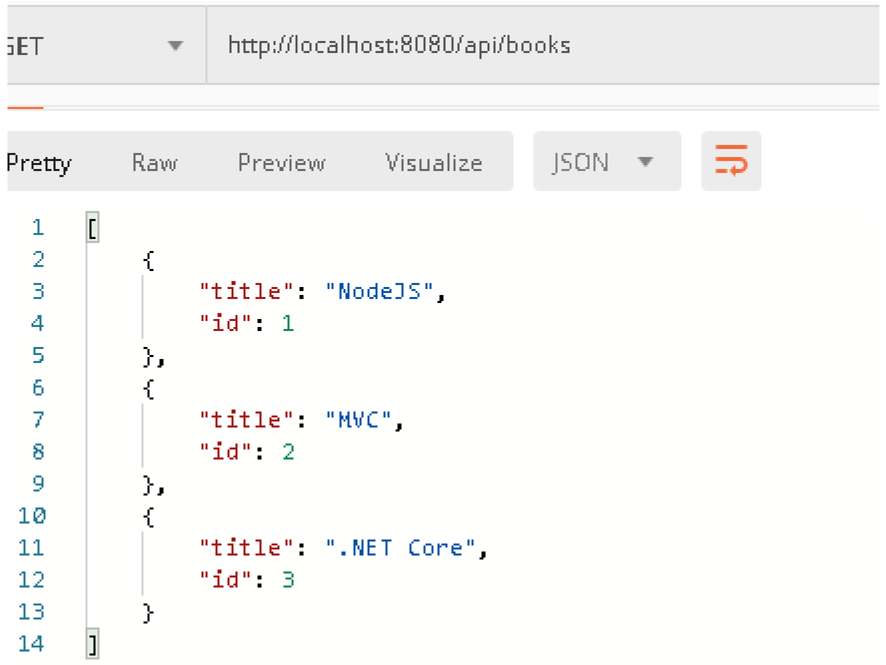
```
const express = require('express');
const Joi = require('joi'); //used for validation
const app = express();
app.use(express.json());

const books = [
{title: ‘Node.JS', id: 1},
{title: ‘MVC', id: 2},
{title: ‘.NET Core', id: 3}
]

//READ Request Handlers
app.get('/', (req, res) => {
res.send('Welcome to Radix REST API with Node.js Tutorial!!');
});

app.get('/api/books', (req,res)=> {
res.send(books);
});

app.get('/api/books/:id', (req, res) => {
const book = books.find(c => c.id === parseInt(req.params.id));

if (!book) res.status(404).send('<h2 style="font-
family: Malgun Gothic; color: darkred;">Ooops... Cant find what you are looking for!</h2>');
res.send(book);
});
```

```
//PORT ENVIRONMENT VARIABLE
const port = process.env.PORT || 8080;
app.listen(port, () => console.log(`Listening on port
 ${port}..`));
```


GET http://localhost:8080

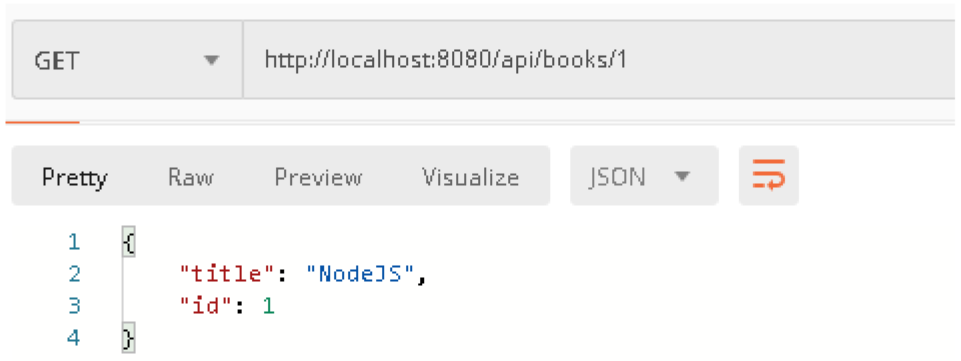
Pretty Raw Preview Visualize HTML

```
Welcome to Radix REST API with Node.js Tutorial!!
```


GET http://localhost:8080/api/books

Pretty Raw Preview Visualize JSON

```
[
    {
        "title": "NodeJS",
        "id": 1
    },
    {
        "title": "MVC",
        "id": 2
    },
    {
        "title": ".NET Core",
        "id": 3
    }
]
```


GET http://localhost:8080/api/books/1

Pretty Raw Preview Visualize JSON

```
{
    "title": "NodeJS",
    "id": 1
}
```

Validation JOI

```
function validateBook(book) {
const schema = {
title: Joi.string().min(3).required()
};
return Joi.validate(book, schema);

}
```

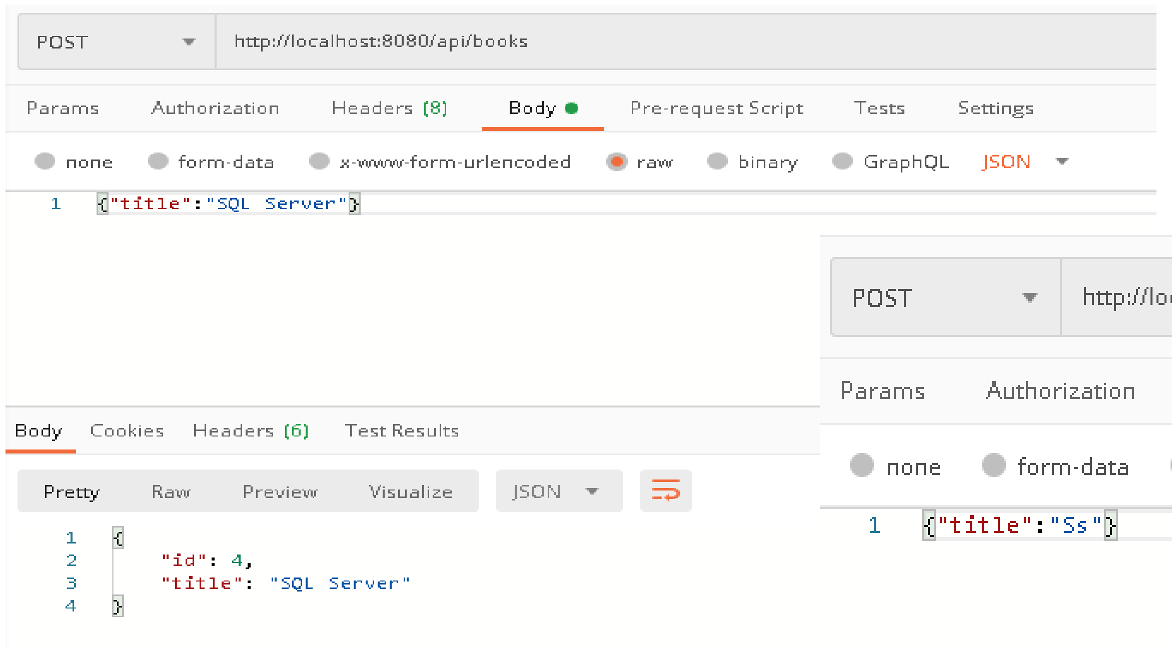
https://medium.com/@rossbulat/joi-for-node-exploring-javascript-object-schema-validation-50dd4b8e1b0f

https://dev.to/itnext/joi-awesome-code-validation-for-node-js-and-express-35pk

Post Method

```
//CREATE Request Handler
app.post('/api/books', (req, res)=> {
console.log(req.body.title);
const { error } = validateBook(req.body);
if (error){
res.status(400).send(error.details[0].message)
return;
}
const book = {
id: books.length + 1,
title: req.body.title
};
books.push(book);
res.send(book);
});
```

Post Method Output



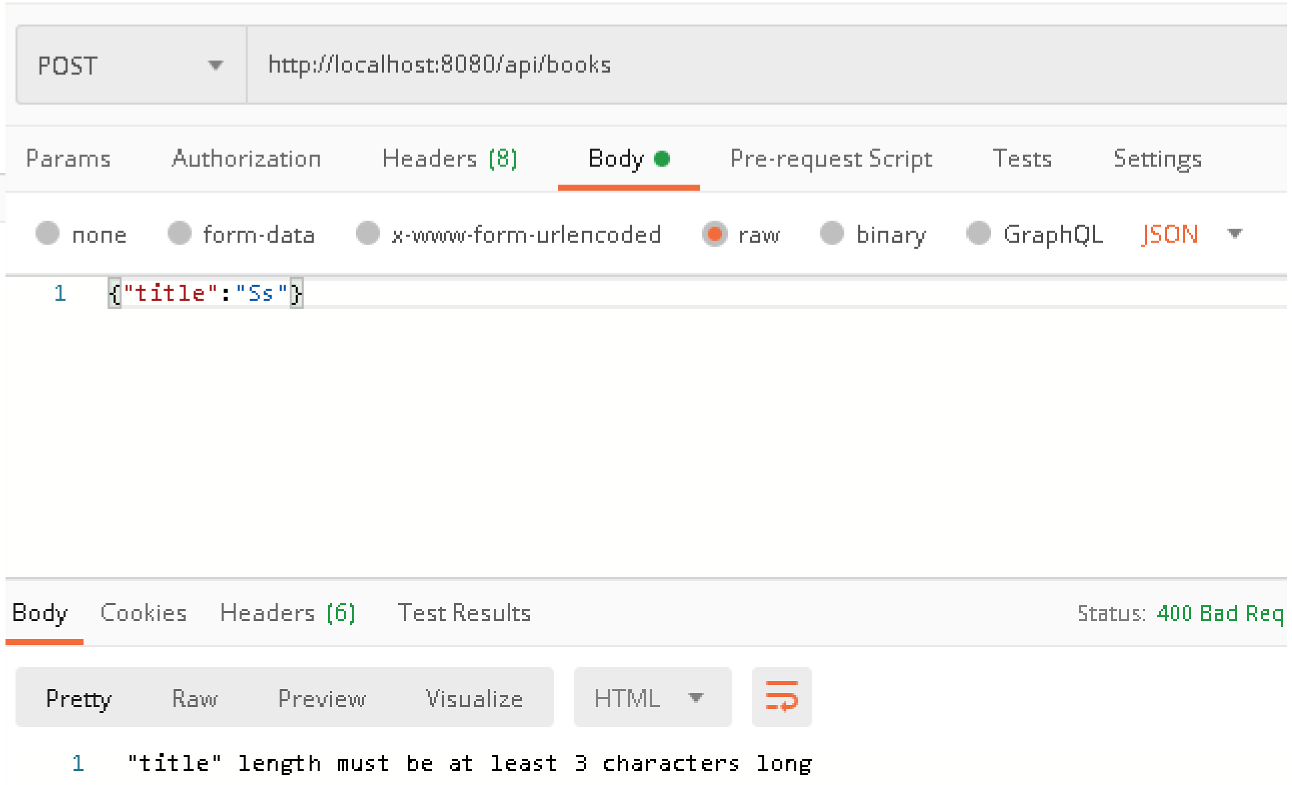
POST http://localhost:8080/api/books

Body: raw JSON

```
{"title":"SQL Server"}
```

Body: Pretty JSON

```
{
    "id": 4,
    "title": "SQL Server"
}
```



POST http://localhost:8080/api/books

Body: raw JSON

```
{"title":"Ss"}
```

Status: 400 Bad Req

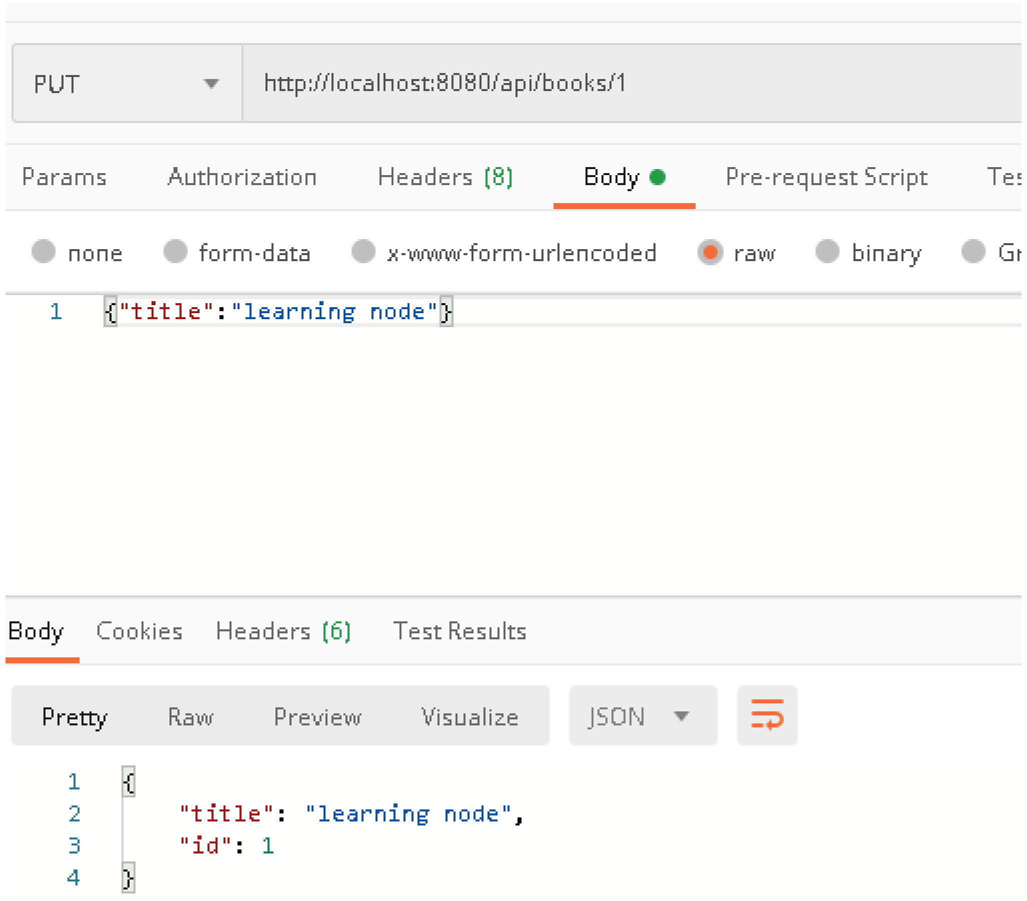
Body: Pretty HTML

```
"title" length must be at least 3 characters long
```

Put Method Output

```
//UPDATE Request Handler
app.put('/api/books/:id', (req, res) => {
const book = books.find(c=> c.id === parseInt(req.params.id));
if (!book) res.status(404).send('<h2 style="font-
family: Malgun Gothic; color: darkred;">Not Found!! </h2>');

const { error } = validateBook(req.body);
if (error){
res.status(400).send(error.details[0].message);
return;
}

book.title = req.body.title;
res.send(book);
});
```

Put Method Output



http://localhost:8080/api/books/1

Body: raw

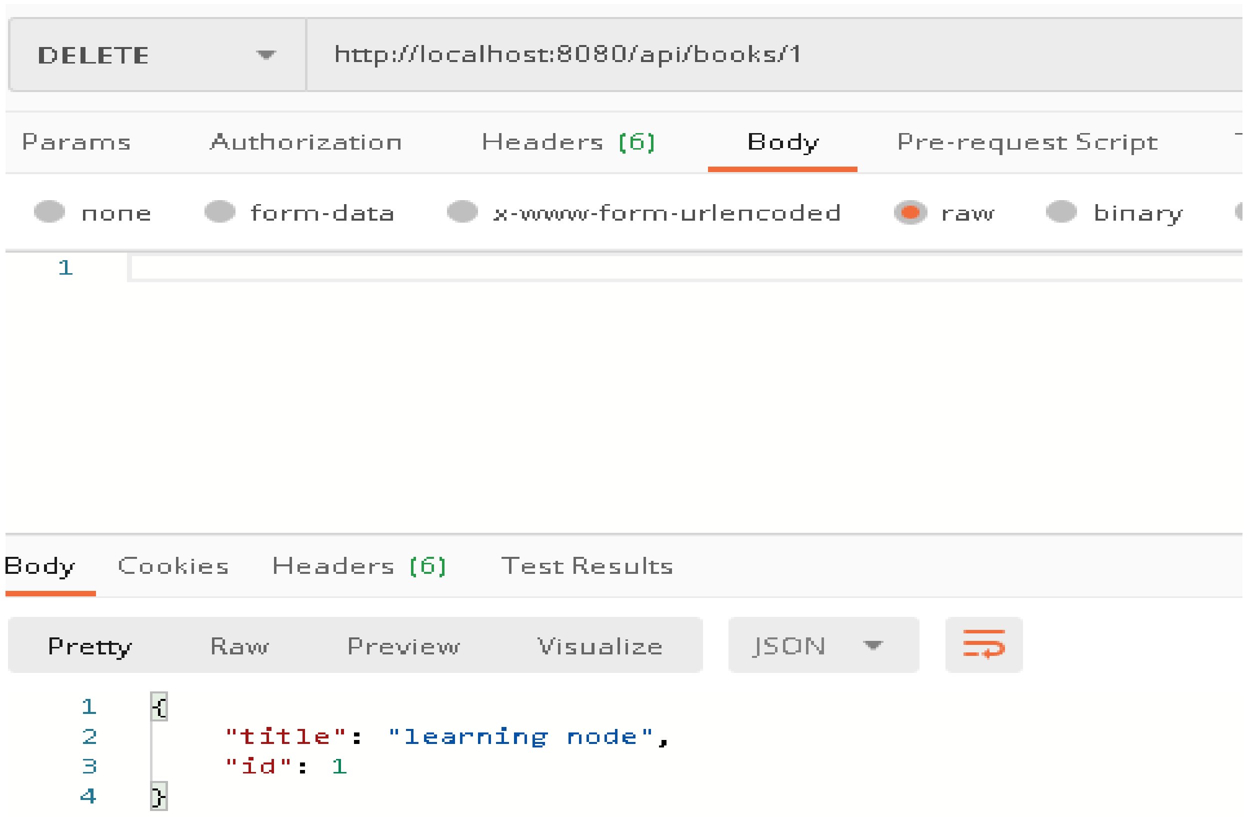
```
{"title":"learning node"}
```

```
{
    "title": "learning node",
    "id": 1
}
```

Delete Method

```
//DELETE Request Handler
app.delete('/api/books/:id', (req, res) => {

const book = books.find( c=> c.id === parseInt(req.params.id));
if(!book) res.status(404).send('<h2 style="font-
family: Malgun Gothic; color: darkred;"> Not Found!! </h2>');

const index = books.indexOf(book);
books.splice(index,1);

res.send(book);
});
```

Delete Method output

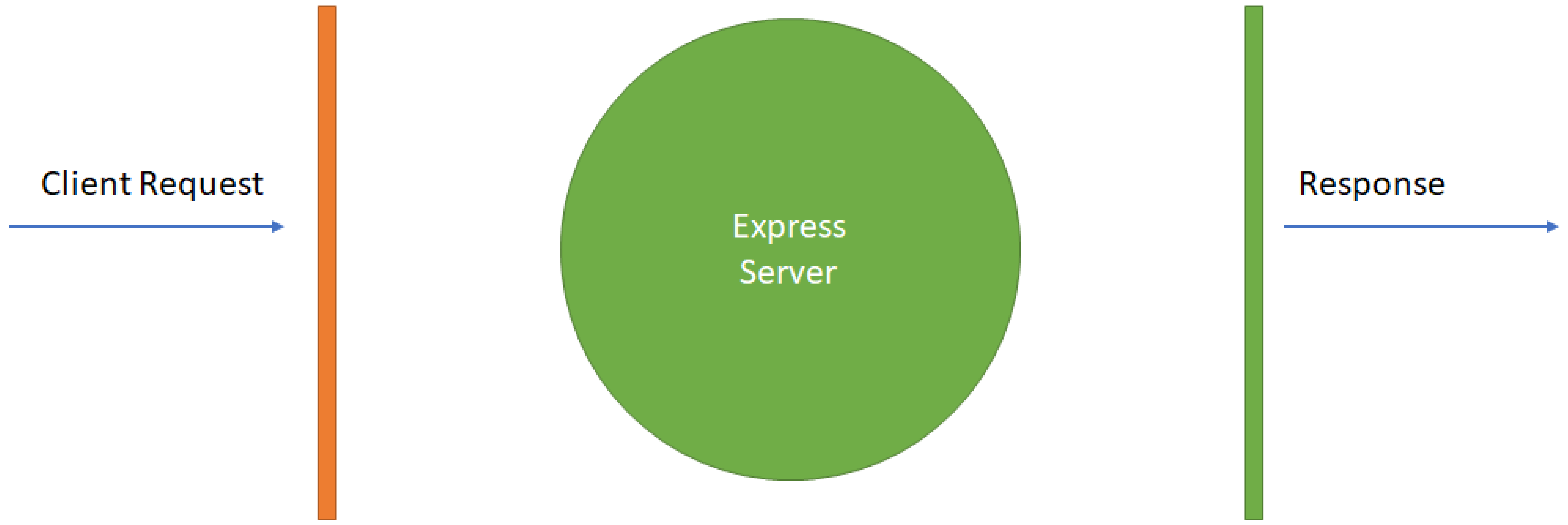


DELETE http://localhost:8080/api/books/1

```
{
    "title": "learning node",
    "id": 1
}
```

Middleware

- 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.
- As name suggests it comes in middle of something and that is request and response cycle
- Middleware has access to **request** and **response** object
- Middleware has access to **next** function of request-response life cycle


Client Request
Express
Server
Response

Middleware functions can perform the following tasks:

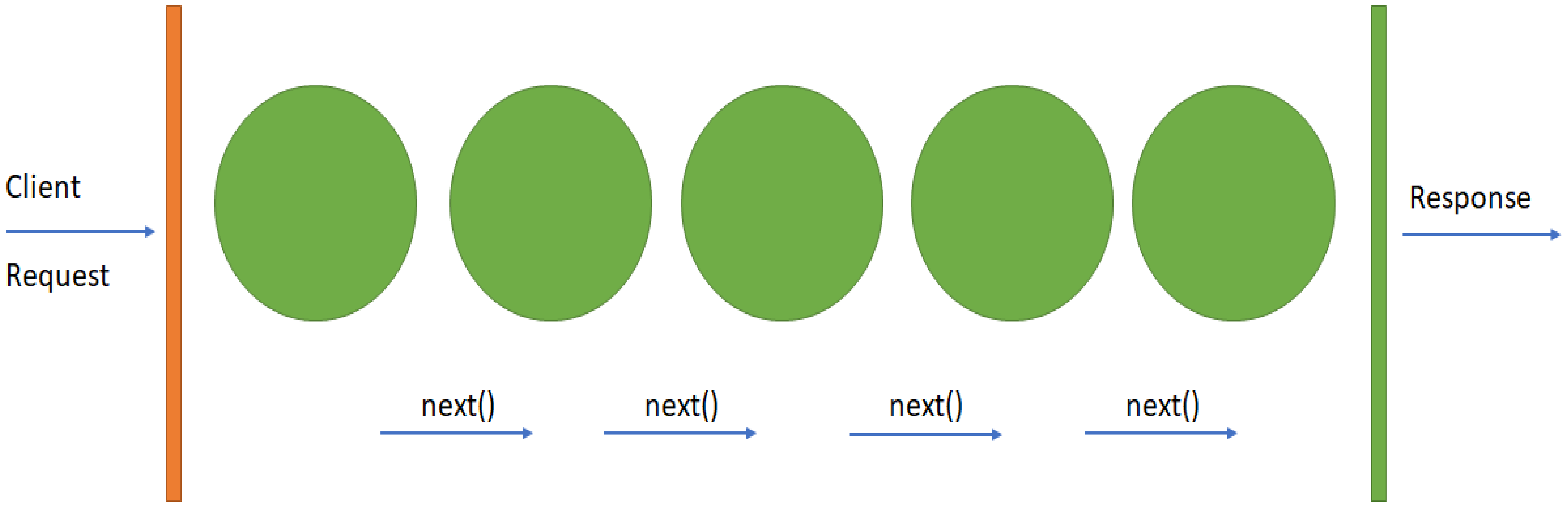
- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

What is this next()?

A middleware is basically a function that will the receive the Request and Response objects, just like your route Handlers do. As a third argument you have another function which you should call once your middleware code completed. This means you can wait for asynchronous database or network operations to finish before proceeding to the next step. This might look like the following:

All middleware has access to req,res and next



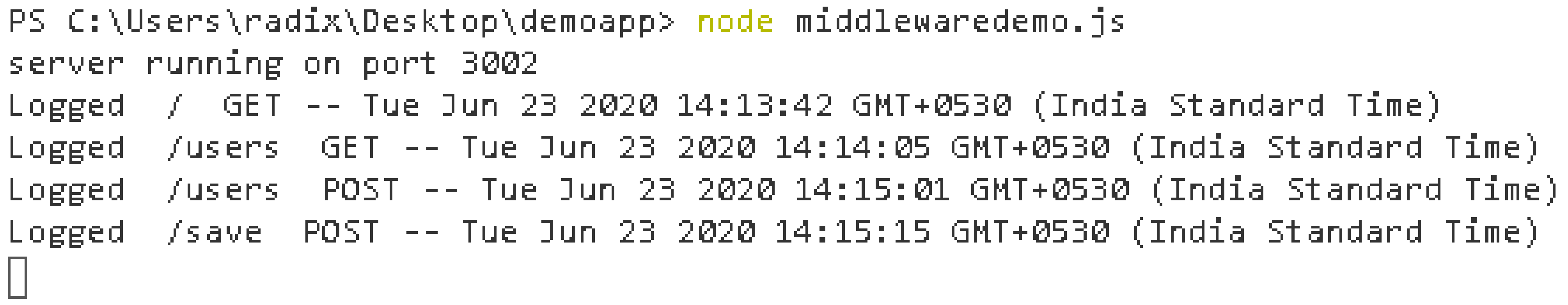
Types of express middleware

- Application level middleware app.use
- Router level middleware router.use
- Built-in middleware express.static,express.json,express.urlencoded
- Error handling middleware app.use(err,req,res,next)
- Thirdparty middleware bodyparser,cookieparser

Application level middleware

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

// custom middleware create
const LoggerMiddleware = (req,res,next) =>{
    console.log(`Logged  ${req.url}  ${req.method} -- ${new Date()}`)
    next();
}

const app = express()

// application level middleware
app.use(LoggerMiddleware);
// users route
app.get('/users',(req,res)=>{
    res.json({
        'status':true
    })
})

// save route
app.post('/save',(req,res)=>{
    res.json({
        'status':true
    })
})
app.listen(3002,(req,res)=>{
    console.log('server running on port 3002')
})
```

Output



```
PS C:\Users\radix\Desktop\demoapp> node middlewaredemo.js
server running on port 3002
Logged  /  GET -- Tue Jun 23 2020 14:13:42 GMT+0530 (India Standard Time)
Logged  /users  GET -- Tue Jun 23 2020 14:14:05 GMT+0530 (India Standard Time)
Logged  /users  POST -- Tue Jun 23 2020 14:15:01 GMT+0530 (India Standard Time)
Logged  /save  POST -- Tue Jun 23 2020 14:15:15 GMT+0530 (India Standard Time)
```

On every request logger middleware will be called and passed the request to the next middleware
With the help of next() function.

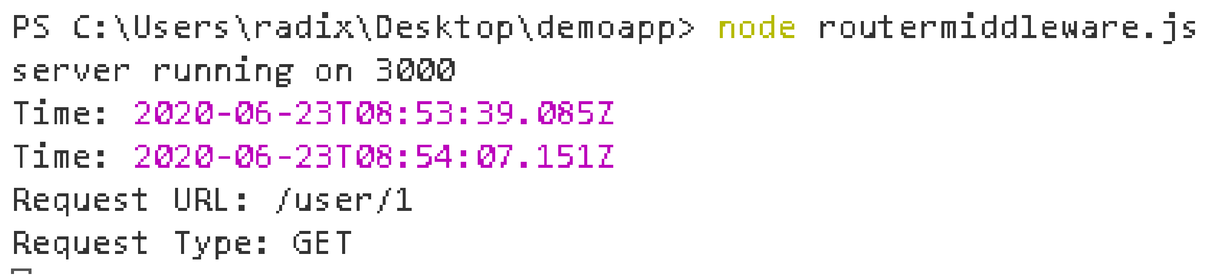
Router Level Middleware

```
//Router-level middleware works in the same way as application level
middleware, except it is bound to an instance of express.Router().
const router = express.Router()
```

Example

```
const express = require('express');
const app = express();
const router = express.Router()
router.use((req,res,next)=>{
   console.log("Time:",new  Date())
   next()
})
router.get("/user/:id",(req,res,next)=>{
   console.log('Request  URL:', req.originalUrl)
   next()
},(req,res,next)=>{
   console.log('Request  Type:', req.method)
   next()
},(req,res)=>{
   res.json({
      status:true,
      id:req.params.id
   })
})
app.use('/',router)

app.listen(3000,(req,res)=>{
   console.log('server  running on 3000')
})
```

Output



```
PS C:\Users\radix\Desktop\demoapp> node routermiddleware.js
server running on 3000
Time: 2020-06-23T08:53:39.085Z
Time: 2020-06-23T08:54:07.151Z
Request URL: /user/1
Request Type: GET
```

Error Handing Middleware

- Express JS comes with default error handling params, define error-handling middleware functions in the same way as other middleware functions, except error-handling functions have four arguments instead of three:

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

Third-party Middlewares

- In some cases we will be adding some extra features to our backend

  Example: body-parser

  All middlewares will populate the req.body property with the parsed body when the Content-Type request header.

Third Party Middleware

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

app.use(bodyParser.urlencoded({extended:false}))

app.use(bodyParser.json())

app.post('/save',(req,res)=>{
  res.json({
    "status":true,
    "payload":req.body
  })
}

app.listen(3000,(req,res)=>{
    console.log('server running on port')
})
```

Additional Third-Party Middleware

- https://expressjs.com/en/resources/middleware.html

Serving Static Files in Node.JS

- To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.

The function signature is:

express.static(root, [options])

For example, use the following code to serve images, CSS files, and JavaScript files in a directory named public:

```
app.use(express.static('public'))
```

Now, you can load the files that are in the public directory:

http://localhost:3000/images/kitten.jpg

http://localhost:3000/css/style.css

http://localhost:3000/js/app.js

http://localhost:3000/images/bg.png

http://localhost:3000/hello.html

Connecting MSSQL and Mongodb

- https://medium.com/@shubhamjajoo1/way-to-connect-nodejs-with-sql-server-6e20aba3f2d0
- https://medium.com/@joelrodrigues/how-to-access-mongodb-from-node-js-e37c587f226a
- https://mongodb.github.io/node-mongodb-native/api-articles/nodekoarticle1.html