21/07/2022	Node JS
Previous day : **Port Number** **Http verbs** **Parts of url**	**Lecture Flow :** **Dynamic path value** **Query Parameter** **Status code**
Topic and Explanation	**Referrance**
Fetching Query and Route Parameters in Express.js **Introduction :** Express.js is a popular backend web application framework for Node.js. While setting up your routes you might encounter scenarios where you need to fetch the parameters specified in the URL path to retrieve a particular set of data. This article will take you through how to extract values from URL route and query parameters. We recommend you have some basic experience with Javascript and NodeJS and have setup simple routes using Express.js before you go on with this guide. **Extract Query Parameters :** A Query String is the part that comes after the Question mark ? in a URL. Let's take an example URL. **https://example.com/api/users?type=admin&status=active** The URL above fetches a list of users with a type of admin and status is active .To fetch these values in Express you can access them using req.query object in your route handler.	**Link 1 :** https://dev.to/davidamunga/fetching-query-and-route-parameters-in-expressjs-1d92 **Link 2:** https://masteringjs.io/tutorials/express/body **Link 3:** https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
const express = require('express')
const app = express()
app.get('/users', (req, res) => {
  const type = req.query.type;
  const status = req.query.status;
})
app.listen(8080);
```

Extract Route Parameters

On some routes we might want to fetch parameters that are specified within the URL path. Extracting these parameters is similar to the Query parameters. Let's take an example URL
.

https://example.com/api/users/1

This URL fetches a User resource with and id of 1. Using Express we can write a single route handler that can fetch any User with any provided id . Lets take a look at an example.

```
const express = require('express')
const app = express()
app.get('/users/:id', (req, res) =>
{
  const id = req.params.id;
})
app.listen(8080);
```

In this example, you can notice that the route specified is /users/:id , where :id represents the provided user id in the URL. Whenever we navigate to the URL with a value passed in that part of the path gets populated as the id parameter. We can fetch the id within the route in the req.params object.

The POST Method

POST is used to send data to a server to create/update a resource.
The data sent to the server with POST is stored in the request body of the HTTP request:

Some notes on POST requests:

POST requests are never cached
POST requests do not remain in the browser history
POST requests cannot be bookmarked
POST requests have no restrictions on data length

Example :

```
const express = require('express')
//commonJS Modules //import express from 'express' -Es6
const mockData = require('../Day 3/mock/posts.js') //object
const app = express()

//add middlewares
app.use(express.json()) //accept json form of data

// app.methodname(pathname, funcToHandleReq)
app.get('/', (request, response) => {
  console.log("Get Call executed")

response.sendFile('/Users/jmahirakz/FSD-JG-March-2022/BackEnd/Day 3/index.html')
})

//GET Operations
app.get('/posts', (req, res) => {
```

```
    const queryParams = req.query
    const { userID, title } =
queryParams

    let filteredPosts =
mockData.posts
    if (userID) {
      filteredPosts =
filteredPosts.filter((post) => {
        return post.userId ===
Number(userID) ? true : false
      })
    }

    if (title) {
      filteredPosts =
filteredPosts.filter((post) => {
        return post.title === title
? true : false
      })
    }

    res.json(filteredPosts) //from
DB
  })

  app.get('/posts/:postID', (req,
res) => {
    //return posts with ID - postID
    const paramObj = req.params
    const { postID } = paramObj
//{postID: '1'} - destructuring
    const post =
mockData.posts.find((post) => {
      return post.id ===
Number(postID) ? true : false
    })
```

```
    if (post) {
      res.json(post)
    } else {
      res.status(404).json({ status:
'No Data Found' }) //send response
along with different status code
    }
  })

  //add posts
app.post('/posts', (req, res) => {
    const postData = req.body
    mockData.posts.push(postData)
    res.status(201).send(postData)
  })

  app.listen(8000, () => {
    console.log("Server Started
Successfully at Port 8000")
  })
```

HTTP response status codes :

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

1 : Informational responses (100–199)
2 : Successful responses (200–299)
3 : Redirection messages (300–399)
4 : Client error responses (400–499)
5 : Server error responses (500–599)

Example :

100 Continue

This interim response indicates that the client should continue the request or ignore the response if the request is already finished.

102 Processing (WebDAV) This code indicates that the server has received and is processing the request, but no response is available yet. **200 OK** The request succeeded. The result meaning of "success" depends on the HTTP method: GET: The resource has been fetched and transmitted in the message body. HEAD: The representation headers are included in the response without any message body. PUT or POST: The resource describing the result of the action is transmitted in the message body. TRACE: The message body contains the request message as received by the server. **202 Accepted** The request has been received but not yet acted upon. It is noncommittal, since there is no way in HTTP to later send an asynchronous response indicating the outcome of the request. It is intended for cases where another process or server handles the request, or for batch processing. **300 Multiple Choices** The request has more than one possible response. The user agent or user should choose one of them. (There is no standardized way of choosing one of the responses, but HTML links to the possibilities are recommended so the user can pick.) **400 Bad Request** The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).	

402 Payment Required Experimental This response code is reserved for future use. The initial aim for creating this code was using it for digital payment systems, however this status code is used very rarely and no standard convention exists. **404 Not Found** The server can not find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource itself does not exist. Servers may also send this response instead of 403 Forbidden to hide the existence of a resource from an unauthorized client. This response code is probably the most well known due to its frequent occurrence on the web. **405 Method Not Allowed** The request method is known by the server but is not supported by the target resource. For example, an API may not allow calling DELETE to remove a resource. **406 Not Acceptable** This response is sent when the web server, after performing server-driven content negotiation, doesn't find any content that conforms to the criteria given by the user agent. **503 Service Unavailable** The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded. Note that together with this response, a user-friendly page explaining the problem should be sent. This response should be used for temporary conditions and the Retry-After HTTP header should, if possible, contain the estimated time before the recovery of the service. The webmaster must also take care about the caching-related headers that are sent along with this response, as these temporary	

condition responses should usually not be cached.	
Lecture Recording : https://university.attainu.com/library	**Summary :** **Dynamic path value → Query Parameter → Status code**