**Documentation**

**Middleware: -** Middleware is a software that acts as an intermediary between two applications or services to facilitate their communication.

A component which is designed to modify an HTTP request and/or response but does not (usually) serve the response in its entirety, designed to be chained together to form a pipeline of behavioural changes during request processing.

Middleware is about how our application responds to incoming requests. Middleware investigates the incoming request and make decisions based on this request i.e., either it can modify that request and pass it on to the next middleware or it can also directly respond to that request on its own.

Middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application’s request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

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.

Graphical user interface, text, application

Description automatically generated with medium confidence

Video Link: - <https://youtu.be/qkMJL0FwiyE>

Editorial Link: - <https://expressjs.com/en/guide/writing-middleware.html>

**Express: -** Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

*Routing:* - Routing refers to how an application’s endpoints (URIs) respond to client request.

* Each endpoint is the location from which APIs can access the resources they need to carry out their function. When an API requests information from a web application or web server, it will receive a response. The place that APIs send requests and where the resource live, is called an endpoint.

**ENDPOINTS = SERVER\_URL + ENDPOINT PATH**

e.g., api.google.com + /user i.e., api.google.com/user

or

api.google.com/login

* **HTTP Request Types**

*Get: - requests resource (HTML, CSS, Image, JS).*

*Post: - used to submit an entity to the specified resource.*

*Put: - replaces all current representations of the target resource.*

*Delete: - delete the target resource.*

*Patch: - apply partial modifications to a response.*

* **HTTP Response Status**

*2XX -> Success*

*3XX -> Redirections*

*4XX -> Client Error*

*5XX -> Server Error*

**Middleware Functions in express: -**

Link: - <https://expressjs.com/en/4x/api.html>

***Parsing***: - Parsing means analysing and converting a program into an internal format that a runtime environment can actually run, for example the JavaScript engine inside browsers.

1. **express.json([options]): -** It parses incoming requests with JSON payloads and is based on body-parser. Returns middleware that only parses JSON and only looks at requests where the *Content-Type* header matches the *type* option.

A new body object containing the parsed data is populated on the request object after the middleware (i.e. req.body), or an empty object ({}) if there was no body to parse, the Content-Type was not matched, or an error occurred.

*type*: - “application/json”

1. **express.raw([options]): -** It parses incoming request payloads into a Buffer and is based on body-parser. Returns middleware that parses all bodies as a Buffer and only looks at 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 middleware (i.e. req.body), or an empty object ({}) if there was no body to parse, the Content-Type was not matched, or an error occurred.

*type*:-“application/octet-stream”

1. **express.Router([Options]): -** Creates a new router object.
2. **express.urlencoded([options]): -** It parses the incoming request with urlencoded payloads and is based on *body-parser*. Returns middleware that only parses urlencoded bodies and only looks at requests where the *Content-Type* header matches the *type* option.

A new body object containing the parsed data is populated on the request object after the middleware (i.e. req.body), or an empty object ({}) if there was no body to parse, the Content-Type was not matched, or an error occurred. This object will contain key-value pairs, where the value can be a string or array (when extended is false), or any type (when extended is true).

*type*: - “application/x-www-form-urlencoded”.

1. **express.text([options]): -** It parses incoming request payloads into a string and is based on body-parser. Returns middleware that parses all bodies as a string and only looks at requests where the Content-Type header matches the type option.

A new body string containing the parsed data is populated on the request object after the middleware (i.e., req.body), or an empty object ({}) if there was no body to parse, the Content-Type was not matched, or an error occurred.

*type: -* “text/plain”

**Environment Variables Video Link: -** <https://www.youtube.com/watch?v=z4zDueqJEsc>

**CORS (Cross-Origin Resource Sharing): -** Cors is a mechanism which uses additional http headers to tell the browser whether a specific web app can share resources with another web app.

*Note*: - But both web apps should have different origins.

Origin = Protocol + Domain name + Port

e.g., <http://mydomain.com:8080>

Two origins are different if any of the three (Protocol/Domain name/Port) are different.

e.g., <http://mydomain.com:8080>, <https://mydomain.com:8080>, <http://mydomain.com:8070>, <http://mydomain2.com:8080> all of these are different origins.

**Video Link: -** <https://www.youtube.com/watch?v=woXBXJgGQvQ>

**Buffer: -** Buffers were introduced to help developers to deal with binary data. In an ecosystem that traditionally deals with strings rather than binary.

Buffers are deeply linked with streams, when the stream processor receives data faster than it can digest then it puts the data into a buffer.

Stream in node.js simply means a sequence of data being moved from one place to another.

The whole concept is you have a huge data to process, but you don’t need to wait for all the data to be available before you start processing it.

A simple visualization is when we watch a YouTube video, most of the time the watching speed is slower than downloading speed. Downloading is done simultaneously with watching, so here buffer comes into the picture.

Code:-

Let buf = Buffer.from(“dvdgvsfbfs”);