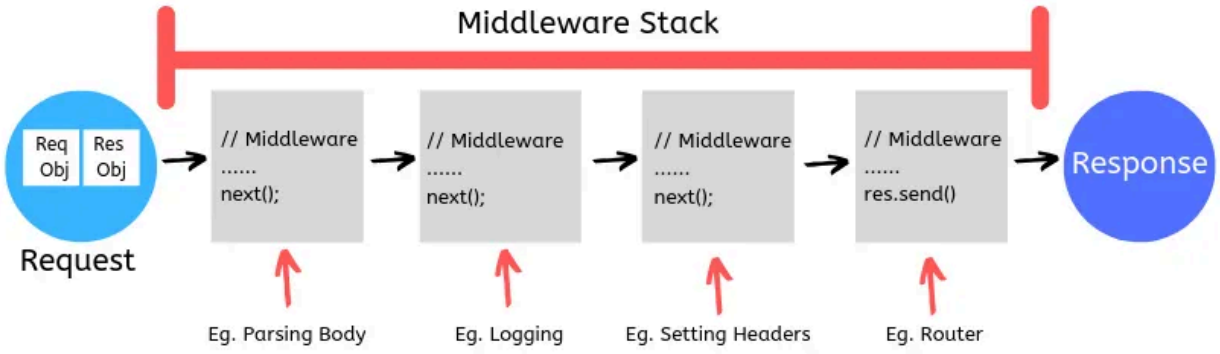
Middleware

[Middleware in Express - GeeksforGeeks]
Middleware are software tools that act as intermediaries between different applications, systems, or services, facilitating their communication and interaction. They ensure that data and requests can be exchanged smoothly and efficiently, even if the systems involved are built using different technologies.

Middleware in Express refers to functions that process requests before reaching the route handlers. These functions can modify the request and response objects, end the request-response cycle, or call the next middleware function. Middleware functions are executed in the order they are defined. They can perform tasks like authentication, logging, or error handling. Middleware helps separate concerns and manage complex routes efficiently.



How Middleware Works in Express.js?

In Express.js, middleware functions are executed sequentially in the order they are added to the application. Here's how the typical flow works:

1. Request arrives at the server.
2. Middleware functions are applied to the request, one by one.
3. Each middleware can either:
    a. Send a response and end the request-response cycle.
    b. Call next() to pass control to the next middleware.
4. If no middleware ends the cycle, the route handler is reached, and a final response is sent.

Middleware functions can perform the following tasks:

- Make changes to the request and the response objects.
- End the request cycle by sending back a response to the client.

- Call the next middleware function in the stack.
- It can be used to add logging and authentication functionality.
- It can be used for error handling and parsing the request data.
- It helps for Optimization and better performance.

Types of Middleware

Application-level Middleware

Application-level middleware is bound to the entire Express application using app.use() or app.METHOD(). It executes for all routes in the application/**every incoming request**, regardless of the specific path or HTTP method.

```
app.use(express.json()); // Parses JSON data for every incoming request
app.use((req, res, next) => {
  console.log('Request received:', req.method, req.url);
  next();
});
```

Router-level middleware:-

Router-level middleware in Express.js refers to middleware functions that are bound to a specific instance of the Express Router. Unlike application-level middleware, which is applied globally to the entire application, router-level middleware is specific to a particular set of routes defined by an instance of the Router.

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

// Router-level middleware
router.use((req, res, next) => {
  console.log('This middleware runs for routes defined by the router.');
  next(); // Pass control to the next middleware or route handler
});
// Mount the router at a specific path
app.use('/myapp', router);
```

In this example, the router-level middleware is applied to all routes defined within the router instance. The app.use('/myapp', router) line specifies that the router is mounted at the path '/myapp', so the middleware and routes defined within the router will only be active for requests that start with '/myapp'.

Error-handling middleware:-

Error handling middleware refers to a type of middleware that is specifically designed to handle errors that occur during the processing of a request. When an error occurs in an Express application, it can be caught by an error handling middleware, allowing you to handle and respond to the error in a centralized and organized way.
This middleware is essential for sending a consistent error response and avoiding unhandled exceptions that might crash the server.

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

You must provide **four** arguments to identify it as an error-handling middleware function. Even if you don't need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors.

Built-in middleware:-

Built-in middleware are middleware functions that are included with the Express framework by **default**. These middleware functions provide common and essential functionalities that can be used in web applications.
Here are some examples of built-in middleware in Express.js:
1. express.json():
This middleware parses incoming JSON data from the request body and makes it available in req.body.
2. express.urlencoded():
This middleware parses incoming URL-encoded form data from the request body and makes it available in req.body.
3. express.static():
This middleware serves static files (e.g., images, CSS, JavaScript) from a specified directory.

Third-party middleware:-

Third-party middleware is developed by external developers and packaged as npm modules. These middleware packages add additional functionality to your application, such as request logging, security features, or data validation.

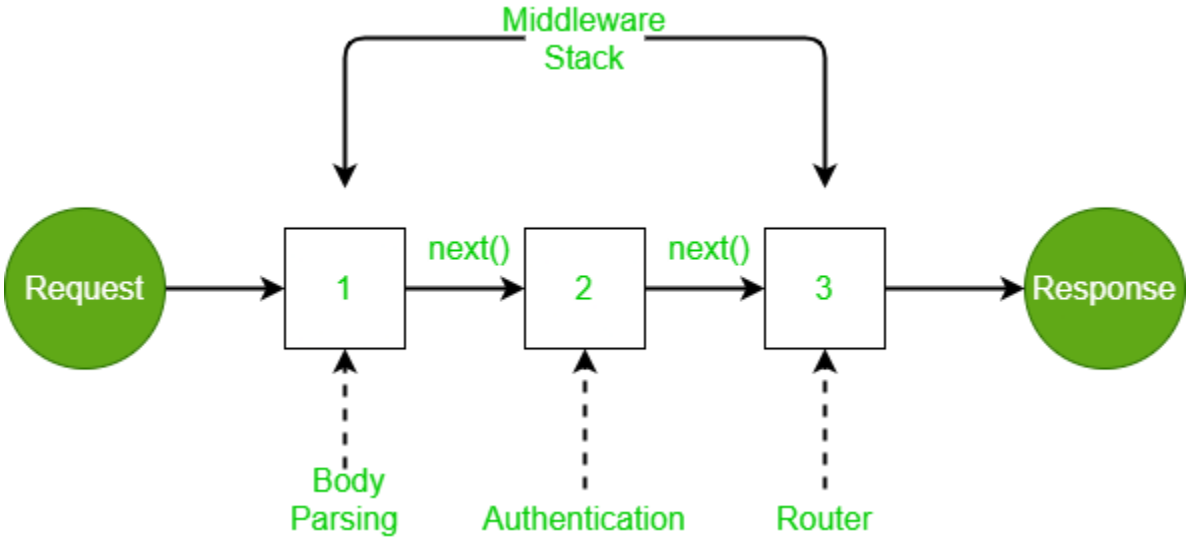
Here are a few examples of popular third-party middleware in Express.js:
1. cookie-parser:
Middleware for parsing cookies attached to incoming requests and making the parsed data available in req.cookies.
2. body-parser:
A middleware that parses incoming request bodies in different formats, such as JSON and URL-encoded data, and makes the parsed data available in req.body.
3. express-session:
Middleware for handling user sessions, allowing you to store and retrieve data associated with a user across multiple requests.
4.multer:
Middleware for handling multipart/form-data, commonly used for file uploads.
5.passport:
Middleware for validating and sanitizing input data in request bodies, query parameters, and headers.

Middleware Chaining

Middleware can be chained from one to another, Hence creating a chain of functions that are executed in order. The last function sends the response back to the browser. So, before sending the response back to the browser the different middleware processes the request.

The next() function in the express is responsible for calling the next middleware function if there is one.

Modified requests will be available to each middleware via the next function



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

// Middleware 1: Log request method and URL
app.use((req, res, next) => {
    console.log(`${req.method} request to ${req.url}`);
    next();
});

// Middleware 2: Add a custom header
app.use((req, res, next) => {
    console.log('Adding custom header');
    res.setHeader('X-Custom-Header', 'Middleware Chaining Example');
    next();
});

// Route handler
app.get('/', (req, res) => {
    console.log('Handling request');
    res.send('Hello, World!');
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Browser Storage

[A Developer's Guide to Browser Storage: Local Storage, Session Storage, and Cookies - DEV Community]

When developing web applications, storing data in the browser is essential for maintaining **user sessions, saving preferences, and improving performance**. Three common methods for client-side storage are Local Storage, Session Storage, and Cookies.

localStorage

localStorage is storage data that is only stored in the browser where:

- Stored data is **persistent**. This means this data remains in the browser's memory even if the user refreshes the web page or closes and reopens the browser window.
- Stored data is onl**y accessible through client-side JavaScript** and cannot be accessed by the server.
- Stored data is not automatically sent to the server with every HTTP request. This means that the server will not have access to the data unless it is specifically requested.

sessionStorage

sessionStorage behaves similarly to localStorage in that the data stored with it is only accessible through client-side JavaScript and is not automatically sent to the server with every HTTP request. However, sessionStorage has one key difference: **the data stored with it is automatically deleted when the user closes the browser tab or window where the data was stored.**

Cookies

Cookies are another form of data that can be stored in the browser. Some key properties of cookies include:

- **Persistence**. The data stored in cookies is persistent, which means it will remain in the browser even if the user refreshes the web page or closes and reopens the browser window.

- **Accessibility/Security**. Cookies can be marked as HttpOnly which means that they can only be accessed by the server and not by JavaScript running in the browser. This helps to protect the security of the data stored in the cookie.
- **Automatic transmission**. Unlike localStorage or sessionStorage, the data stored in cookies is automatically sent to the server with every HTTP request.

Points	Local	Session	Cookie
Storage Limit	5 MB	5-10 MB	4KB
Validity	Lifetime	Browser/Tab close	Depends on the type
HTTP request	Not sent with HTTP requests, so it's good for storing data that doesn't need to be transferred to the server	Not sent with HTTP requests, so it's good for storing data that doesn't need to be transferred to the server	Automatically sent with each HTTP request to the server, making them useful for tracking sessions or managing authentication.
Example	Saving user preferences like theme settings or cart items that should persist even after the browser is closed.	Keeping form data or user selections that should only persist until the user navigates away or closes the browser tab.	Storing user login tokens, session IDs, or tracking information that must be sent to the server with every request.