Backend Development with Node.js Lab 2

1 Middleware

You can continue with the project folder from the previous lab exercise or create a new project from scratch as we did before in a new directory, for e.g. `secondapp`

If you creating from scratch, then just repeat all the previous steps in this directory, for e.g. type:

```
npm init
```

Press Enter to accept the default for all the questions except for the one below, for which you should enter a new value:

```
entry point (index.js): app.js
```

Press Enter to the final `Is this Ok` question and you will be returned to the prompt.

Open this folder in Visual Studio Code (or the particular IDE you are using).
Then install Express.js in this directory by typing:

```
npm install express
```

The source code files for this lab can be obtained from `labcode / middleware` folder from the downloaded zip.

1.1 Application level Middleware processing flow

In Express.js, middleware is one of the most fundamental and powerful concepts. It lies at the core of how Express apps handle requests and responses.

Middleware functions are functions that have access to the following three objects: `(req, res, next)` where:

- req → the request object (HTTP request made by client)
- res → the response object (HTTP response to be sent to client)
- next → a callback function that passes control to the next middleware in the stack

Middleware functions sit between the incoming request and the outgoing response, processing or modifying them as needed.
When a request hits an Express app, Express processes it through a series (stack) of middleware functions, in the order they are defined. Each middleware can:

a) Perform some task (e.g., logging, parsing, authentication)
b) Modify the request or response objects
c) End the request-response cycle (e.g., send a response)
d) Call `next()` to hand control to the next middleware in line

If a middleware in a series or chain of middleware functions does not call `next()`, the request stops there and no further middleware runs.

At some point the request will reach the destination route handler. After that the response from the route handler (from the `send()`) flows back through the middleware (in reverse order)

An Express application can use the following types of middleware:

a) Application-level middleware
b) Router-level middleware
c) Error-handling middleware
d) Built-in middleware
e) Third-party middleware

You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

Create a file `app.js` into this project folder (you had earlier specified this as the main entry point for this project in package.json).

Place it with the content from `app-v1.js`

Run it with `nodemon app.js`

Trace the flow of incoming request processing to the following URLs:

- root URL (/)
- things URL (/things)

from the console log output statements on the server side.

First global middleware
```
app.use((req, res, next) => {
  console.log("Processing Middleware 1 at " + new Date());
  next();
});
```
`app.use(...)` with no path means run this for every request (all paths, all HTTP methods).
The function signature `(req, res, next)` makes it a standard middleware.
`next()` hands control to the next middleware/route. If you forgot `next()`, the request would hang.

Second global middleware
```
app.use((req, res, next) => {
  console.log('Processing Middleware 2');
  next();
});
```
Also runs for every request, after Middleware 1 (order matters). Calls next() to continue the chain which then goes onto the single root URL route path handler.

Middleware mounted on a specific path (`things`)
```
app.use('/things', function(req, res, next){
   console.log("Processing middleware for things at " + new Date());
   next();
});
```
Only runs for every request for the \things path, after the 2 other global middleware completes.. Calls next() to continue the chain which then goes onto the \things URL route path handler.

The request flow (for GET /)

a) Request enters the app.
b) Global Middleware 1 runs → logs with timestamp → next().
c) Global Middleware 2 runs → logs "Processing Middleware 2" → next().
d) Express finds the matching route GET / → handler runs

The request flow (for GET /things)

a) Request enters the app.
b) Global Middleware 1 runs → logs with timestamp → next().
c) Global Middleware 2 runs → logs "Processing Middleware 2" → next().
d) Middleware for /things runs → logs "Processing middleware for things"
e) Express finds the matching route GET /things → handler runs

We will see a few more complex examples of implementing application level middleware.

Update `app.js` with the content from `app-v2.js`

Make requests to the following URLs and confirm the response returned and console log displayed.

http://localhost:3000/user/5

http://localhost:3000/books/catstory

http://localhost:3000/order/0

http://localhost:3000/order/5

Section	Type	Description	Example URL	Output
`/user/:id`	`app.use()` middleware stack	Logs URL and method for any `/user/:id` request before reaching handler	`/user/101`	Logs URL + method, sends "Response for user with ID: 101"
`/books/:title`	Route-level middleware	Logs the book title then sends a response	`/books/alchemist`	Logs "Books title: alchemist", sends "Response for books"
`/order/:id`	Conditional route handling	Uses `next('route')` to skip stack for certain IDs	`/order/0`	"special response"
			`/order/9`	"regular response"

1.2 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() and provides a different perspective on defining and using middleware. Its use cases include grouping route-specific middleware, API versioning, and organizing routes into logical groups

Aspect	Application-Level Middleware	Router-Level Middleware
Scope of effect	Applies **globally** to the entire app; every incoming request passes through it (unless restricted by path).	Applies **only** to a specific router instance or group of routes.
Attachment point	Defined directly on the `app` object using `app.use()` or `app.METHOD()`.	Defined on a `Router` instance using `router.use()` or `router.METHOD()`, which is later mounted to a path on the app.
Typical use cases	Logging, authentication checks, CORS setup, body parsing, global error handling, etc.	Route grouping (e.g., `/users`, `/admin`), performing middleware specific to that group like input validation, access control, or parameter checking.
Example syntax	`app.use((req, res, next) => { ... });`	`router.use((req, res, next) => { ... });`
Mounting	Automatically active once declared.	Must be mounted on an app using `app.use('/path', router)`.

Aspect	Application-Level Middleware	Router-Level Middleware
Effect on routing	Runs for all routes, unless limited by path.	Runs only when the router's base path is matched.

Update `app.js` with the content from `app-v3.js`

Make requests to the following URLs and confirm the response returned and console log displayed.

http://localhost:3000/users

http://localhost:3000/users/3

http://localhost:3000/admin

http://localhost:3000/admin/settings

http://localhost:3000/books/catstory

1.3 Error handling middleware

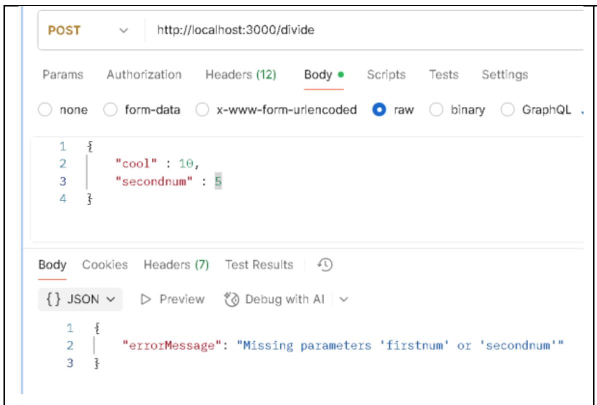
Error-handling middleware always takes four arguments (`err, req, res, next`) to handle errors that occur during request processing. You must provide four arguments to identify it as an error-handling middleware function. Even if you don't need to use the `next` object, you must specify it to maintain the signature. Otherwise, the `next` object will be interpreted as regular middleware and will fail to handle errors.

The middleware
- Should be defined after other `app.use()` and route calls
- Can be used to centralize error handling logic
- Can forward errors to the next error handler using next(err)

Typical use cases:
a) Catch synchronous errors (e.g. invalid JSON, missing fields, arithmetic errors, etc)
b) Catch asynchronous errors (e.g. errors in reading from database, rejected promises)
c) Handle unmatched route paths with a 404 response
d) Centralized error response formatting and logging in the final middleware definition to return appropriate error message (in JSON or other suitable format)

Update `app.js` with the content from `app-v4.js`

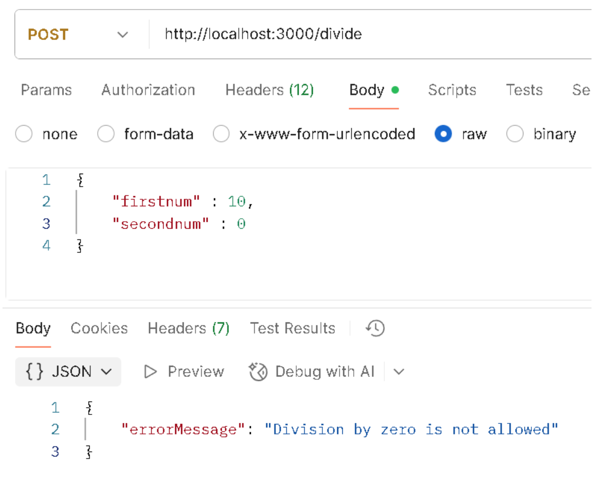
Experiment with sending different combination of valid and invalid values via a POST to `/divide` and check on the response returned and the error message in the console log output on the server side.



http://localhost:3000/divide

```
{
  "cool" : 10,
  "secondnum" : 5
}
```

```
{
  "errorMessage": "Missing parameters 'firstnum' or 'secondnum'"
}
```



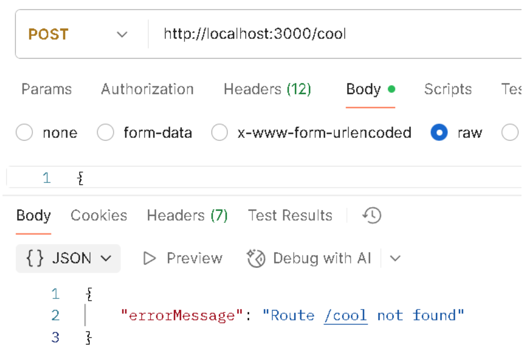
http://localhost:3000/divide

```
{
  "firstnum" : 10,
  "secondnum" : 0
}
```

```
{
  "errorMessage": "Division by zero is not allowed"
}
```

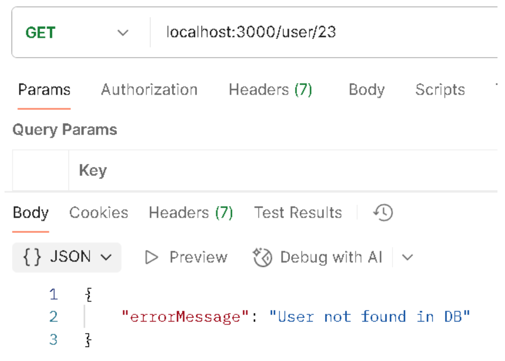
Send a request to the route path with a handler that simulates handling error related to a database error: localhost:3000/user/23

Finally, try sending requests to non-existent route paths, and verify that the appropriate error response returned and the error message in the console log output on the server side.



http://localhost:3000/cool

```
{
  "errorMessage": "Route /cool not found"
}
```



localhost:3000/user/23

```
{
  "errorMessage": "User not found in DB"
}
```

1.4 Defining custom middleware

You can also write your own middleware to implement your customized functionality in the middleware stack.

A typical use case of custom middleware is to perform some form of custom logging of incoming requests, as analysis of request histories is common data analytic activity performed for security and performance reasons.

Update `app.js` with the content from `app-v5.js`

We can define the custom middleware function as a standalone function or directly as an arrow function within `app.use()`). If you define the middleware function as a separate, standalone function, that you will need to load it using app.use().

The order of middleware loading is important: middleware functions that are loaded first are also executed first. So, we need to ensure our custom middleware is loaded first in the middleware stack sequence before it calls next() to pass onto the next middleware / route handler.

Send a few sample requests to the sample routes defined with the correct method, and verify that the details are logged correctly to a file `requests.log` in the same directory as app.js

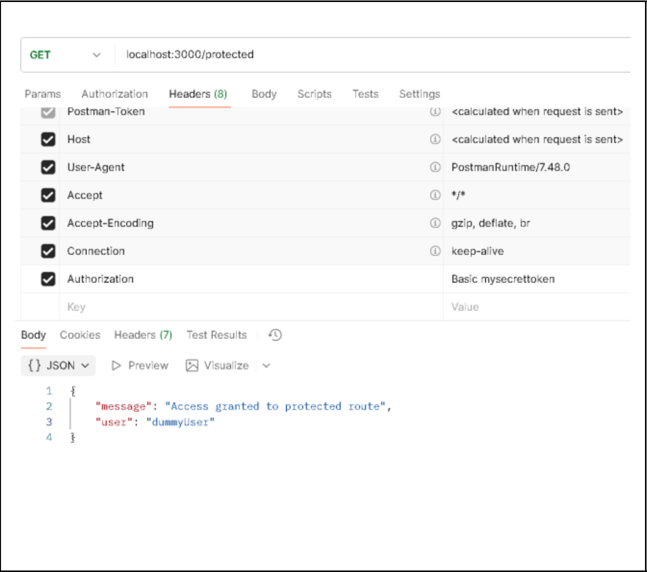
Another typical use case of custom middleware is to perform some form of custom authentication on requests. This custom middleware

a) Intercepts requests on selected routes only.
b) Checks for the presence of an Authorization header with a specific expected value (e.g., Bearer mysecrettoken).
c) Returns a dummy JSON response on successful authentication or a 403 Forbidden error with a JSON error message if the token is missing or incorrect.

Update `app.js` with the content from `app-v6.js`

Attempt accessing the public route and verify it succeeds without any issue.

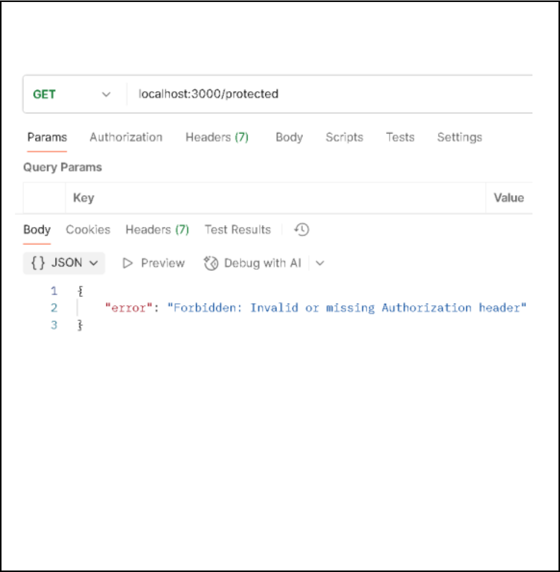
Attempt accessing the protected route with the correct Authorization header set up in Postman and verify that it succeeds. Then try accessing it without this header setup and verify that an error message with the correct status code is returned.


GET localhost:3000/protected

Headers (8)

Postman-Token	<calculated when request is sent>
Host	<calculated when request is sent>
User-Agent	PostmanRuntime/7.48.0
Accept	*/*
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
Authorization	Basic mysecrettoken

```
{
  "message": "Access granted to protected route",
  "user": "dummyUser"
}
```


GET localhost:3000/protected

```
{
  "error": "Forbidden: Invalid or missing Authorization header"
}
```

2 END