- ⏱️ 9 minutes
- ✅ Completed

# Express Middleware

In a previous reading, we briefly introduced the urlencoded middleware function. Middleware functions are a critical part of a robust Express server. In this reading you will:

1. Understand that the request pipeline in an Express application is composed of a series of middleware functions.
2. Write a custom middleware function that validates user-provided data (submitted via an HTML form), sets an array of user-friendly validation error messages on the Request object when necessary, and passes control to the next middleware function.

## Middleware overview

Express middleware is kind of a misnomer: because of the "middle" in "middleware", you might assume that middleware is anything that sits between the client and the Express server. However, according to the Express documentation on using middleware: "An Express application is essentially a series of middleware function calls." Let's dive into what this means.

For starters, start up a demo server by running the following commands:

```
mkdir middleware-demo
cd middleware-demo
npm init --yes
npm install express@^4.0.0
npm install nodemon@^2.0.0 --save-dev
touch index.js
```

Then, in your `index.js`, handle get requests to the root path by responding with "Hello World!":

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

const app = express();

app.get("/", (req, res) => {
  res.send("Hello World!");
});
```

```
// Define a port and start listening for connections.
const port = 3000;

app.listen(port, () => console.log(`Listening on port
${port}...`));
```

Set up a `start` script in your `package.json`:

```
{
  "name": "middleware-demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.2"
  }
}
```

Then start up your server by running `npm start`

## Anatomy of a middleware function

In Express, a middleware function is a function that takes three arguments, in this specific order:

1. `req`- the request object
2. `res`- the response object
3. `next`- according to the Express documentation on using middleware: "the next middleware function in the application's request-response cycle"

These arguments might seem a little familiar. Up to this point, you've been handling all requests with a callback function that takes a `req` and `res` argument.

For example, take a look at the callback function that you just set up to send back "Hello World!": it takes `req` and `res` as the first two arguments. There is, in fact, an optional `next` argument that you could have passed into this function as well.

The `next` argument will be discussed in more depth later in this reading.

This means that all of the callback functions that you've been writing this whole time to handle requests and send back responses are actually middleware functions.

# Series of middleware functions

As a reminder, the documentation mentioned that "an Express application is a *series* of middleware function calls." To explore what "series" means there, let's set up another middleware function.

Here's the goal: let's set up a middleware function that logs the time of each request.

Remember, a middleware function takes three arguments: `req`, `res`, and `next`.
In `index.js`, create a middleware function `logTime` that console logs the current time formatted as an ISO string. At the end of the middleware function, invoke the `next` function, which represents the next middleware function:

```
const logTime = (req, res, next) => {
  console.log("Current time: ", new Date().toISOString());
  next();
};
```

Now, update the `app.get('/')` route so that it calls `logTime` before it invokes the anonymous callback function that sends back "Hello World!":

```
app.get("/", logTime, (req, res) => {
  res.send("Hello World!");
});
```

To confirm that this is working, refresh `localhost:3000` and check that your server logs are showing the current time of each request.

Let's recap what just happened:

1. When the user lands on `localhost:3000`, a GET request is made to the "/" route of the Express server.
2. The first middleware function this route invokes is `logTime`. In `logTime`, the current time is logged. At the end of `logTime`, it invokes `next`, which represents the next middleware function.
3. The next middleware function in this example is the anonymous callback function that runs `res.send("Hello World!")`.

You could invoke as many middleware functions as you'd like. In addition, because the `req` and `res` objects are passed through every one of the middleware functions, you could store values in the `req` object for the next middleware function to use.
Let's explore this by creating another middleware function called `passOnMessage`:

```
const passOnMessage = (req, res, next) => {
  console.log("Passing on a message!");
  req.passedMessage = "Hello from passOnMessage!";
  next();
};
```

Then, let's add this middleware function to the `app.get('/')` route and then console.log the `req.passedMessage` in one of the later middleware functions:

```
app.get("/", logTime, passOnMessage, (req, res) => {
  console.log("Passed Message: ", req.passedMessage);
  res.send("Hello World!");
});
```

*In the example above, the `passedMessage` was added to the `req` object so that it could be used in a later middleware function. Alternatively, you might instead want to store properties inside of the res.local object so that you don't accidentally override an existing property in the `req` object.*

Instead of passing each middleware function in separate arguments, you could also pass them all in as one array argument:

```
app.get("/", [logTime, passOnMessage], (req, res) => {
  console.log("Passed Message: ", req.passedMessage);
  res.send("Hello World!");
});
```

The order does matter. Try changing up the order of the middleware functions and see the order of the console.log statements.

## Application-level middleware

To be clear, with the current set up, `logTime` and `passOnMessage` will only be executed for the `app.get('/')` route. For example, let's say you set up another route:

```
app.get("/bye", (req, res) => {
  res.send("Bye World.");
});
```

Because that route does not currently take in `logTime` as one of its arguments, it would not invoke the `logTime` middleware function. To fix this, you could simply pass in the `logTime` function, but if there was a middleware function that you wanted to execute for every single route, this could be pretty tedious.

Setting up an application-level middleware function that runs for every single route is simple. In fact, the `express.urlencoded` middleware you set up in the previous reading was an application-level middleware.

To do this, remove `logTime` from the `app.get('/')` arguments. Instead, add it as an application-wide middleware by writing `app.use(logTime)`. After doing this, your `index.js` file should look like this:

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

const app = express();

const logTime = (req, res, next) => {
  console.log("Current time: ", new Date().toISOString());
  next();
};

app.use(logTime);

const passOnMessage = (req, res, next) => {
```

```
  console.log("Passing on a message!");
  req.passedMessage = "Hello from passOnMessage!";
  next();
};

app.get("/", passOnMessage, (req, res) => {
  console.log("Passed Message: ", req.passedMessage);
  res.send("Hello World!");
});

app.get("/bye", (req, res) => {
  res.send("Bye World.");
});

// Define a port and start listening for connections.
const port = 3000;

app.listen(port, () => console.log(`Listening on port
${port}...`));
```

Now, whenever you navigate to either `localhost:3000` or `localhost:3000/bye`, the `logTime` middleware function will be executed. Note how the `passOnMessage` is only executed for the `app.get('/')` route.

# Data validations with middleware

In the previous reading, you set up data validations in your Express server in the "Guest List" example.

Let's pick up where that example left off and move the data validations into a middleware function.

At this point, your `index.js` should look like this:

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

// Create the Express app.
const app = express();

// Set the pug view engine.
app.set("view engine", "pug");
app.use(express.urlencoded());

const guests = [];

// Define a route.
```

```
app.get("/", (req, res) => {
  res.render("index", { title: "Guest List", guests });
});

app.get("/guest", (req, res) => {
  res.render("guest-form", { title: "Guest Form" });
});

app.post("/guest", (req, res) => {
  const { fullname, email, numGuests } = req.body;
  const numGuestsNum = parseInt(numGuests, 10);
  const errors = [];

  if (!fullname) {
    errors.push("Please fill out the full name field.");
  }

  if (!email) {
    errors.push("Please fill out the email field.");
  }

  if (!numGuests || numGuests < 1) {
    errors.push("Please fill out a valid number for number of
guests.");
  }

  if (errors.length > 0) {
    res.render("guest-form", {
      title: "Guest Form",
      errors,
      email,
      fullname,
      numGuests
    });
    return;
  }

  const guest = {
    fullname,
    email,
    numGuests: numGuestsNum
  };
  guests.push(guest);
  res.redirect("/");
});

// Define a port and start listening for connections.
const port = 8081;
```

```
app.listen(port, () => console.log(`Listening on port
${port}...`));
```

You might be wondering why you would want to move the validation logic into a middleware function. Suppose you now wanted to add a route that would allow the user to update a `guest` on the guest list.

In that update route, you probably want to enforce the same validations. You could simply copy and paste over all of those validations, but having it in a middleware function would keep your code DRY.

To start, create a function called `validateGuest` and move all of the validation logic into that function.
Because this will be a middleware function, be sure to accept `req`, `res`, and `next` as arguments in the function.
Finally, your `validateGuest` functions should pass on error messages to a later function so the later function can render the error messages back to the client.
When you're done with your `validateGuest` function, it should look something like this:

```
const validateGuest = (req, res, next) => {
  const { fullname, email, numGuests } = req.body;
  const numGuestsNum = parseInt(numGuests, 10);
  const errors = [];

  if (!fullname) {
    errors.push("Please fill out the full name field.");
  }

  if (!email) {
    errors.push("Please fill out the email field.");
  }

  if (!numGuests || numGuests < 1) {
    errors.push("Please fill out a valid number for number of
guests.");
  }

  req.errors = errors;
  next();
};
```

Notice how the `errors` array is passed on to the next middleware function by being added to the `req` object. Update your `app.post('/guest')` route so that it now uses the `validateGuest` middleware function and so that it checks `req.errors` for error messages:

```
app.post("/guest", validateGuest, (req, res) => {
  const { fullname, email, numGuests } = req.body;
  if (req.errors.length > 0) {
    res.render("guest-form", {
      title: "Guest Form",
```

```
      errors: req.errors,
      email,
      fullname,
      numGuests
    });
    return;
  }

  const guest = {
    fullname,
    email,
    numGuests
  };
  guests.push(guest);
  res.redirect("/");
});
```

In summary, moving validations into a middleware allows you to concisely reuse validations across different routes. In production-level projects, you'll likely use a validation library called express-validator, which follows the same pattern of validating data in middleware functions and then passing on error messages through the `req` object.

The express-validator library gives you a wide range of pre-built validations so that you don't have to implement validation logic from scratch. For example, it comes with a pre-built validation for checking whether an input field's is in a proper email format: `check('email').isEmail()`. You'll get a chance to explore the `express-validator` middleware functions more in today's project!

# What you learned

In this reading, you learned:

1. that the request pipeline in an Express application is composed of a series of middleware functions
2. how to write a custom middleware function that validates user-provided data (submitted via an HTML form), sets an array of user-friendly validation error messages on the Request object when necessary, and passes control to the next middleware function.