**Middleware and modules**

 Estimated Time: 1 hour

[Middleware](http://expressjs.com/en/guide/using-middleware.html) is the organizing principle of Express apps. An Express app is a stack of middleware functions that requests are sent through.

Each function in the middleware stack gets access to a request object, a response object, and a next function, which can be called to pass control to the next middleware function in the stack.

**const** express = require('express');

**const** app = express();

**const** myMiddlewareFunc = (req, res, next) => {

console.log(req.url);

**if** (someConditionIsTrue(req)) {

*// return a response*

**return** res.json({msg: 'someMessage'});

}

**else** {

*// call next to trigger next middleware function*

next();

}

}

app.use(myMiddlewareFunc);

app.get('/api/foo', (req, res) => {

**return** res.json({foo: 'bar'});

});

app.get('/api/bar', (req, res) => {

**return** res.json({bar: 'foo'});

})

app.listen(8080);

In this example app, we have two routes. If a user makes a request to /api/foo, we expect the response to be the JSON object {foo: 'bar'}, and for requests to /api/bar, {bar: 'foo'}.

We've created a middleware function (myMiddlewareFunc) and configured our app to use it (app.use(myMiddlewareFunc). When a request comes in to our app (for either of our endpoints), the request will first go through our middleware function, and depending on what happens there, may or may not end up at our route function.

Having a look at myMiddlewareFunc, this function takes three arguments (req, res, and next). Sometimes you'll see request instead of req or response instead of res, and although we could name these parameters tom, bill, and sue, it's best to stick to convention. This function logs req.url, then checks if some condition is true about the request object. If it is, the middleware function returns a response object, which means that the request will never make it to the main route function. We'll see a concrete example of this in a moment, where we check to see if the user is authenticated, and if not, send an error message.

If the condition is false, we call the next function, which will pass control to the next middleware function in the stack. Here, since we only have a single middleware function, if a user makes a request to /api/foo and the condition is false, control will pass to the route function for /api/foo.

**Crucially, all middleware functions must do one of two things: either return a response or call next()**. If a middleware function does not end with one of these behaviors, it will hang and your app will be blocked.

In Express apps, we can distinguish between *built-in* middleware (for instance, express.static, which is used to expose folders containing static assets like CSS and client-side JavaScript), *third-party* middleware (like [body-parser](https://www.npmjs.com/package/body-parser), which we saw in the previous lesson), and *custom* middleware, which we write ourselves.

The beauty of the middleware-centric approach to server-side web development is that it encourages us to write modular, reusable, functional code. Let's explore some examples.