Middleware!

More often than note, you're going to want many of your endpoints to have the same functionality. Imagine if we wanted to print a console.log every time a user hit our endpoint, or what if you wanted to have a body parser for all of your endpoints that take in a body. You could copy and paste the same code over and over and over again, but there's a better way! Middleware is ultimately just a function (or series of functions) that sits between our server's port and the endpoints!

A middleware method is one that looks something along the lines of:

```
const methodForMiddleware = (req, res, next) => {
  // Whatever you want to do!
  next()
}
```

Notice that there's an extra parameter called next! Ultimately that takes us to whatever "next" function there is in line of middleware. We'll see how that works a little further down. To use a middleware function you simply need to write:

```
...
app.use(methodForMiddleware)
...
```

You may be wondering "Hey, we've used app.use" before! What's this about?" and you'd be absolutely right! We do use it for routing (and fun story, our router is also middleware). But before we get into actually coding out some middleware, let's see how they work. Let's see how this works with two middlewares:

```
...

const methodForMiddlewareOne = (req, res, next) => {
  console.log("In middleware method 1")
  next()
}

const methodForMiddlewareTwo = (req, res, next) => {
  console.log("In middleware method 2")
  next()
```

```
}

app.use(methodForMiddlewareOne)
app.use(methodForMiddlewareTwo)
app.use(somePredefinedRouter)
```

When we run this code, what will happen is that when a request comes in, before we end up sending our code to the router, we hop into `methodForMiddlewareOne`, execute the code, and then move to the next middleware we said to use next, which just so happens to be `methodForMiddlewareTwo`, so before we even get to our route, we'll see the console printing:

```
In middleware method 1
In middleware method 2
```

Let's try it with some real code. We can start with taking our code from the end of previous lecture (with office character and parks and rec character routes)! We've now added `methodForMiddlewareOne` and `methodForMiddlewareTwo`, and implemented them with `app.use`.

```
const express = require("express");
const officeRouter = require("./routes/office/officeRoute");
const parksAndRecRouter = require("./routes/parksAndRec/parksAndRecRoute");

const app = express();

const methodForMiddlewareOne = (req, res, next) => {
  console.log("In middleware method 1");
  next();
};

const methodForMiddlewareTwo = (req, res, next) => {
  console.log("In middleware method 2");
  next();
};

app.use(methodForMiddlewareOne);
app.use(methodForMiddlewareTwo);
app.use("/office", officeRouter);
app.use("/parksAndRec", parksAndRecRouter);

const port = 3000;
app.listen(port);
console.log("Now listening on port " + port);
```

Notice that when you run the code and make a call, the console prints:

```
In middleware method 1
In middleware method 2
```

Notice that these get run in order. Middleware functions get run in the order in which they are used. Try removing the `next()` on line 9. Notice anything? What's happening is that we're not moving to the next function defined. You'll likely forget a next here or there, but this architecture can be extremely powerful because you can ultimately control the flow of the request (e.g. authentication).

Logging:

One of most common uses of middleware is logging. You'll often want to record what endpoints are being called, with what parameters, and when. So let's remove our methods for middleware, and write a quick logging method:

```
const express = require("express");
const officeRouter = require("./routes/office/officeRoute");
const parksAndRecRouter = require("./routes/parksAndRec/parksAndRecRoute");

const app = express();

const logger = (req, res, next) => {
  const requestedRoute = req.method + ":" + req.url;
  const timeOfRequest = new Date();

  const logBody = {
    requestedRoute,
    timeOfRequest,
  };

  console.log(logBody);
  next();
};

app.use(logger);
app.use("/office", officeRouter);
app.use("/parksAndRec", parksAndRecRouter);

const port = 3000;
app.listen(port);
console.log("Now listening on port " + port);
```

We now have a logger written, so that every single time we make a call to our API, we log what's being called and when! What's neat is that this will catch literally anything! Try requesting a route that doesn't exist! What makes adding this middleware nice is that we don't need to place a `console.log` (or call our `logger` function) at the beginning of ever function we route to.

Before moving on, let's keep our index clean and add our logger to our library:

```
├── index.js
├── lib
|   └── bodyParser.js
|   └── logger.js
├── package-lock.json
├── package.json
└── routes
    ├── office
    |   ├── office.js
    |   └── officeRoute.js
    └── parksAndRec
        ├── parksAndRecRoute.js
        └── parksNRec.js
```

`/lib/logger.js`

```
const logger = (req, res, next) => {
  const requestedRoute = req.method + ":" + req.url;
  const timeOfRequest = new Date();

  const logBody = {
    requestedRoute,
    timeOfRequest,
  };

  console.log(logBody);
  next();
};

module.exports = logger;
```

`index.js`

```
const express = require("express");
const officeRouter = require("./routes/office/officeRoute");
const parksAndRecRouter = require("./routes/parksAndRec/parksAndRecRoute");
const logger = require("./lib/logger");

```

```
const app = express();

app.use(logger);
app.use("/office", officeRouter);
app.use("/parksAndRec", parksAndRecRouter);

const port = 3000;
app.listen(port);
console.log("Now listening on port " + port);
```

Body Parsing

When printing out that tree, you may have remembered that we have our `bodyParser` file. This is some constantly repeated code that we have when we try to parse incoming json. Let's write some code to turn that into middleware too!

`/lib/bodyParser.js`

```
const bodyParser = (req) =>
  new Promise((resolve) => {
    let chunks = [];
    req.on("data", (chunk) => {
      console.log("Got chunk: ", chunk.toString());
      chunks.push(chunk);
    });
    req.on("end", () => {
      console.log("got everything!");
      resolve(Buffer.concat(chunks));
    });
  });

const parseJSON = async (req, res, next) => {
  const body = await bodyParser(req);
  const json = JSON.parse(body);
  req.body = json;
  next();
};

module.exports = parseJSON;
```

And let's make sure to go change `getMultipleCharacters` in `parksAndRec/parksAndRecRoute.js`:

```
const express = require("express");
const parksAndRecCharacters = require("./parksNRec");

const getParksAndRecCharacters = (req, res) => {
  res.send(parksAndRecCharacters);
};

const getParksAndRecCharacter = (req, res) => {
  const characterName = req.params.characterName;
  res.send({
    characterName: characterName,
    actorName: parksAndRecCharacters[characterName],
  });
};

const getMultipleCharacters = async (req, res) => {
  const response = req.body.map((characterName) => ({
    character: characterName,
    actor: parksAndRecCharacters[characterName],
  }));

  res.send(response);
};

const parksAndRecRouter = express.Router();

parksAndRecRouter.get("/", getParksAndRecCharacters);
parksAndRecRouter.get("/:characterName", getParksAndRecCharacter);
parksAndRecRouter.post("/", getMultipleCharacters);

module.exports = parksAndRecRouter;
```

Not only did we remove the extra code calling the body parser, but when we saved it as `req.body` we can access it within the route as if it were always there!! Try giving that endpoint a call! Works like a charm, and our code looks much cleaner.

Before we move on, there's one minor issue. While we know our code works for our post, try getting all the characters from `/parksAndRec`. Your response probably looks something along the lines of:

```
(node:61562) UnhandledPromiseRejectionWarning: SyntaxError: Unexpected end of JSON input
    at JSON.parse (<anonymous>)
    ...
```

Notice the `unexpected end of JSON input`. Well, with a traditional `GET /parksAndRec` call, we're not adding in any input whatsoever. Clearly our middleware here is being too good and watching everything. We want it only to be middleware for routes that take a body. Luckily for us, we can do that! First we need to remove the middleware from `index.js`:

```
const express = require("express");
const officeRouter = require("./routes/office/officeRoute");
const parksAndRecRouter = require("./routes/parksAndRec/parksAndRecRoute");
const logger = require("./lib/logger");

const app = express();

app.use(logger);
app.use("/office", officeRouter);
app.use("/parksAndRec", parksAndRecRouter);

const port = 3000;
app.listen(port);
console.log("Now listening on port " + port);
```

Next, in our `parksAndRecRoutes.js` (where our post route lives), we can use the middleware in there. It looks a little different than what we've just seen. Try to spot it:

```
const express = require("express");
const parksAndRecCharacters = require("./parksNRec");
const parseJSON = require("../../lib/bodyParser");

const getParksAndRecCharacters = (req, res) => {
  res.send(parksAndRecCharacters);
};

const getParksAndRecCharacter = (req, res) => {
  const characterName = req.params.characterName;
  res.send({
    characterName: characterName,
    actorName: parksAndRecCharacters[characterName],
  });
};

const getMultipleCharacters = async (req, res) => {
  const response = req.body.map((characterName) => ({
    character: characterName,
    actor: parksAndRecCharacters[characterName],
  }));

```

```
  res.send(response);
};

const parksAndRecRouter = express.Router();

parksAndRecRouter
  .route("/")
  .get(getParksAndRecCharacters)
  .post(parseJSON, getMultipleCharacters);
parksAndRecRouter.get("/:characterName", getParksAndRecCharacter);

module.exports = parksAndRecRouter;
```

What did we do? First, notice that we changed the construction of our `parksAndRecRouter`. The router is incredibly powerful, and buy defining an overarching route, we can then decide which specific functions to apply when a given method is called on that route.

Secondly, notice that in our `.post(parseJSON, getMultipleCharacters)` we're adding our middleware as the first parameter, and then our destination function as our second. **Curiously enough**, you can add as many middleware functions as you please, just so long as your destination function is last.

This functionality allows for us to create a middleware that is code that'll be reused regularly, but not necessarily for every single incoming request. You don't need to add route specific middleware if you don't like, but it does keep your functions looking clean.

Middleware Packages via NPM:

Writing your own middleware is good to know. However, it's significantly easier to use middleware that others have already written (and there have been a lot). Let's take our middleware files and update them!

Morgan

A solid middleware to use is morgan. It's maintained by the same folks who maintain express. Just like any other package, install it with:

```
npm i morgan
```

Let's then remove the logger code we wrote in `lib/logger` and change that to be:

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

const logger = morgan("tiny");

module.exports = logger;
```

If we wanted to further customize morgan, we could, but using the tiny parameter gives us some minimal output (if you are more curious about the kinds of outputs you can have, read through the morgan documentation via their github link).

Body-Parser

We put a lot of effort into our body parser, but as it turns out, we can just use the body-parser node package:

```
npm i body-parser
```

And now we can take all of that body parsing code, and replace it with:

```
const bodyParser = require("body-parser");

module.exports = bodyParser;
```

There's a smidge more we'll need to do for body-parser, and that's to update what we're passing into the middleware in `parksAndRecRoute.js`:

```
const parseJSON = require("../../lib/middleware/bodyParser");
const express = require("express");
const parksAndRecCharacters = require("./parksNRec");

const getParksAndRecCharacters = (req, res) => {
  console.log("MIDDLE WARE WHEN? ");
  res.send(parksAndRecCharacters);
};

const getParksAndRecCharacter = (req, res) => {
  const characterName = req.params.characterName;
  res.send({
    characterName: characterName,
    actorName: parksAndRecCharacters[characterName],
  });
};

const getMultipleCharacters = async (req, res) => {
  const body = req.body;

  const response = body.map((characterName) => ({
    character: characterName,
    actor: parksAndRecCharacters[characterName],
  }));
```

```
25
26    res.send(response);
27  };
28
29  const parksAndRecRouter = express.Router();
30
31  parksAndRecRouter
32    .route("/")
33    .get(getParksAndRecCharacters)
34    .post(parseJSON.json(), getMultipleCharacters);
35
36  parksAndRecRouter.get("/:characterName", getParksAndRecCharacter);
37
38  module.exports = parksAndRecRouter;
```

Notice on Line 34 we're using `parseJSON.json()`? That's because many middlewares are factories in that they offer many options, and we can select those options like `.json()` or by passing in a string parameter above like `"tiny"`.

SwaggerUI

Now that we've updated our middlware packages, let's have our application serve up some actual HTML documentation so that folks can actually understand how to use our API! We'll do this with SwaggerUI. This module allows for us to easily serve up swagger pages (that we'll have to define) so that users can go and see how our endpoints work!

```
1  npm i swagger-ui-express
```

In order for swagger to know how our API works, we'll need to create a swagger template file we can just call `swagger.js` (note: swagger template files can be in many formats - we're going to stick with a more javascript approach, but note that on the site's documentation, they tend to use .yml files). We should put that in the `/lib` directory, and while we're at it, let's put our middleware in its own separate directory too:

```
1  .
2  ├── index.js
3  ├── lib
4  |   ├── middleware
5  |   |   ├── bodyParser.js
6  |   |   └── logger.js
7  |   └── swagger.js
8  ├── package-lock.json
9  ├── package.json
```

```
└── routes
    ├── office
    |   ├── office.js
    |   └── officeRoute.js
    └── parksAndRec
        ├── parksAndRecRoute.js
        └── parksNRec.js
```

For our swagger file, we'll need some basic information:

```
const swaggerDocument = {
  openapi: "3.0.1",
  info: {
    version: "1.0.0",
    title: "My application's API Document",
    description: "This is how you use my application!",
    termsOfService: "Nah",
    contact: {
      name: "Your name here",
      email: "someemail@umsl.edu",
      url: "www.google.com/somewebsite",
    },
    license: {
      name: "Apache 2.0",
      url: "https://www.apache.org/licenses/LICENSE-2.0.html",
    },
  },
};

module.exports = swaggerDocument;
```

Now, in order for us to access this swagger, we'll need to actually use it with our express! Back in `index.js` we need to add our swagger to our `app.use`:

```
const express = require("express");
const officeRouter = require("./routes/office/officeRoute");
const parksAndRecRouter = require("./routes/parksAndRec/parksAndRecRoute");
const logger = require("./lib/middleware/logger");
const app = express();
const swaggerUI = require("swagger-ui-express");
const swaggerDoc = require("./lib/swagger");

app.use(logger);
```

```
app.use("/api-docs", swaggerUI.serve, swaggerUI.setup(swaggerDoc));
app.use("/office", officeRouter);
app.use("/parksAndRec", parksAndRecRouter);

const port = 3000;
app.listen(port);
console.log("Now listening on port " + port);
console.log(`Swagger docs at localhost:${port}/api-docs`);
```

Now, when you go to your swagger page, you should see a very basic site! Let's add a route in `swagger.js` to make this useful:

```
const swaggerDocument = {
  openapi: "3.0.1",
  info: {
    version: "1.0.0",
    title: "My application's API Document",
    description: "This is how you use my application!",
    termsOfService: "Nah",
    contact: {
      name: "Matt Lane",
      email: "mjlny2@umsl.edu",
      url: "www.google.com/mattlane",
    },
    license: {
      name: "Apache 2.0",
      url: "https://www.apache.org/licenses/LICENSE-2.0.html",
    },
  },
  paths: {
    "/office": {
      get: {
        description: "GET THE OFFICE CHARACTERS",
        responses: {
          "200": {
            description: "200 - ok - got the characters",
            content: {
              "application/json": {
                example: {
                  "Michael Scott": "Steve Carell",
                  "Dwight Schrute": "Rainn Wilson",
                  "Jim Halpert": "John Krasinski",
                  "Pam Beesly": "Jenna Fischer",
                  "Ryan Howard": "B.J. Novak",
                  "Andy Bernard": "Ed Helms",
```

```
                  "Robert California": "James Spader",
                  "Stanley Hudson": "Leslie David Baker",
                  "Kevin Malone": "Brian Baumgartner",
                  "Meredith Palmer": "Kate Flannery",
                  "Angela Martin": "Angela Kinsey",
                  "Oscar Martinez": "Oscar Nunez",
                  "Phyllis Lapin": "Phyllis Smith",
                  "Roy Anderson": "David Denman",
                  "Jan Levinson": "Melora Hardin",
                  "Kelly Kapoor": "Mindy Kaling",
                  "Toby Flenderson": "Paul Lieberstein",
                  "Creed Bratton": "Creed Bratton",
                  "Darryl Philbin": "Craig Robinson",
                  "Erin Hannon": "Ellie Kemper",
                  "Gabe Lewis": "Zach Woods",
                  "Holly Flax": "Amy Ryan",
                },
              },
            },
          },
        },
      },
    },
  },
};

module.exports = swaggerDocument;

```

Now we have a basic path, go back to your swagger ui! Now you can actually click `try it out` and get a legitimate response back!

Though one thing to note is that GET's are relatively easy to create a path for, doing something with a body can be slightly more frustrating. Let's add a path for our parks and rec route:

```
const swaggerDocument = {
  openapi: "3.0.1",
  info: {
    version: "1.0.0",
    title: "My application's API Document",
    description: "This is how you use my application!",
    termsOfService: "Nah",
    contact: {
```

```
      name: "Matt Lane",
      email: "mjlny2@umsl.edu",
      url: "www.google.com/mattlane",
    },
    license: {
      name: "Apache 2.0",
      url: "https://www.apache.org/licenses/LICENSE-2.0.html",
    },
  },
  paths: {
    "/office": {
      get: {
        description: "GET THE OFFICE CHARACTERS",
        responses: {
          "200": {
            description: "200 - ok - got the characters",
            content: {
              "application/json": {
                example: {
                  "Michael Scott": "Steve Carell",
                  "Dwight Schrute": "Rainn Wilson",
                  "Jim Halpert": "John Krasinski",
                  "Pam Beesly": "Jenna Fischer",
                  "Ryan Howard": "B.J. Novak",
                  "Andy Bernard": "Ed Helms",
                  "Robert California": "James Spader",
                  "Stanley Hudson": "Leslie David Baker",
                  "Kevin Malone": "Brian Baumgartner",
                  "Meredith Palmer": "Kate Flannery",
                  "Angela Martin": "Angela Kinsey",
                  "Oscar Martinez": "Oscar Nunez",
                  "Phyllis Lapin": "Phyllis Smith",
                  "Roy Anderson": "David Denman",
                  "Jan Levinson": "Melora Hardin",
                  "Kelly Kapoor": "Mindy Kaling",
                  "Toby Flenderson": "Paul Lieberstein",
                  "Creed Bratton": "Creed Bratton",
                  "Darryl Philbin": "Craig Robinson",
                  "Erin Hannon": "Ellie Kemper",
                  "Gabe Lewis": "Zach Woods",
                  "Holly Flax": "Amy Ryan",
                },
              },
            },
          },
        },
```

```
      },
    },
    "/parksAndRec": {
      post: {
        summary: "Get multiple characters back based on their character
names",
        consumes: "application/json",
        produces: "application/json",
        requestBody: {
          description: "an array of names to send",
          required: true,
          content: {
            "application/json": {
              schema: {
                type: "array",
                items: {
                  type: "string",
                  example: "Ann Perkins",
                },
              },
            },
          },
        },

        responses: {
          "200": {
            description: "A response with the desired characters and actors",
          },
        },
      },
    },
  },
};

module.exports = swaggerDocument;

```

For our parks and rec path, we actually have to create a requestBody that we can serve up so that the user will understand what to send to the API!