# next()

It seems like our middleware was successful — it logged out

The middleware stack is processed in the order they appear in the application file, such that middleware defined later happens after middleware defined before. It’s important to note that this is regardless of method — an app.use() that occurs after an app.get() will get called after the app.get(). Observe the following code:

app.use((req, res, next) => {

console.log("A sorcerer approaches!");

next();

});

app.get('/magic/:spellname', (req, res, next) => {

console.log("The sorcerer is casting a spell!");

next();

});

app.get('/magic/:spellname', (req, res, next) => {

console.log(`The sorcerer has cast ${req.params.spellname}`);

res.status(200).send();

});

app.get('/magic/:spellname', (req, res, next) => {

console.log("The sorcerer is leaving!");

});

// Accessing http://localhost:4001/magic/fireball

// Console Output:

// "A sorcerer approaches!"

// "The sorcerer is casting a spell!"

// "The sorcerer has cast fireball"

In the above code, the routes are called in the order that they appear in the file, provided the previous route called next() and thus passed control to the next middleware. We can see that the final matching call was not printed. This is because the previous middleware did not invoke the next() function to run the following middleware.

An Express middleware is a function with three parameters: (req, res, next). The sequence is expressed by a set of callback functions invoked progressively after each middleware performs its purpose. The third argument to a middleware function, next, should get explicitly called as the last part of the middleware’s body. This will hand off the processing of the request and the construction of the response to the next middleware in the stack.

**Instructions**

**1.**

Add a call to next after your logging statement so that the middleware will be executed before moving on to your routes.

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

# Request And Response Parameters

Recall the function signature of an Express middleware, i.e., (req, res, next). You might recognize this signature as being the very same that we’ve used for Express routes in the past. Well there’s a perfectly good reason for that: Express routes are middleware. Every route created in Express is also a middleware function handling the request and response objects at that part of the stack. Express routes also have the option of sending a response body and status code and closing the connection. These two features are a byproduct of Express routes being middleware, because all Express middleware functions have access to the request, the response, and the next middleware in the stack.

# Route-Level app.use() - Single Path

Now that we’ve managed to refactor our duplicate code into middleware functions, we should be noticing that our code contains much less repetition than before. Unfortunately, we still have duplicate code in some of our routes. Since this code isn’t shared by all of our routes, the previous syntax of app.use() won’t work. Let’s see what the [Express documentation](https://expressjs.com/en/4x/api.html) for app.use() has to say about this use case. This is the app.use() function signature:

app.use([path,] callback [, callback...])

In documentation for many programming languages, optional arguments for functions are placed in square brackets ([]). This means that app.use() takes an optional path parameter as its first argument. We can now write middleware that will run for every request at a specific path.

app.use('/sorcerer', (req, res, next) => {

console.log('User has hit endpoint /sorcerer');

next();

});

In the example above the console will print 'User has hit endpoint /sorcerer', if someone visits our web page’s ‘/sorcerer’ endpoint. Since the method app.use() was used, it won’t matter if the user is performing a GET,a POST, or any other kind of HTTP request. Since the path was given as an argument to app.use(), this middleware function will not execute if the user hits a different path (for instance: '/spells' or '/sorcerer/:sorcerer\_id').

**Instructions**

**1.**

We’re going to refactor all the logic that checks the existence of a jelly bean into a new middleware function. Currently, this logic is used in every route that begins with beans/:beanName and looks like this:

const beanName = req.params.beanName;

if (!jellybeanBag[beanName]) {

console.log('Response Sent');

return res.status(404).send('Bean with that name does not exist');

}

We check to see if the bean with the supplied name exists in jellybeanBag, and we send a 404 response if it does not. The return statement ensures that we break out of the middleware and don’t try any operations on a nonexistent jelly bean.

Create a new app.use call after your logging middleware. It should be called for all /beans/:beanName routes. You can leave the callback empty at this point.

Checkpoint 2 Passed

**2.**

Copy all the checking logic (from const beanName through the if statement) from a route into your middleware callback. Remove those lines from every route that uses them.

Checkpoint 3 Passed

**3.**

After the checking logic, we’re going to attach the correct bean object to the request by setting a bean property on the request (req.bean). Set it equal to the correct bean from the bean object. For good measure, also attach the bean name to the request as req.beanName.

After these properties are set, be sure to call next.

Checkpoint 4 Passed

Hint

The syntax for setting a property of the request object is

req.propertyName = value;

**4.**

You can now remove the duplicate checking logic from all /beans/:beanName routes. To make sure that all your routes still work if we remove const beanName = req.params.beanName; from them, make sure that you use req.beanName any place where you need to access the bean by name. For instance, inside app.delete, you’ll have to change

jellybeanBag[beanName] = null;

to

jellybeanBag[req.beanName] = null;

Check your routes to make sure that they use req.beanName.

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

**Route-Level app.use() - Multiple Paths**

We learned that app.use() takes a path parameter, but we never fully investigated what that path parameter could be. Let’s take another look at the Express documentation for app.use():

“*argument*: path

*description*: The path for which the middleware function is invoked; can be any of:

* A string representing a path.
* A path pattern.
* A regular expression pattern to match paths.
* An array of combinations of any of the above. “

So app.use() can take an array of paths! That seems like a handy way to rewrite the code from our last exercise so that we don’t have to put the same code in two different routes with different paths.

**Instructions**

**1.**

Now we’ll add some more advanced middleware. You might have noticed that in each PUT and POST route, there is code that looks like this:

let bodyData = '';

req.on('data', (data) => {

bodyData += data;

});

req.on('end', () => {

// ...

});

You don’t need to worry too much about how this code works right now since we’ll eventually be replacing it with a better solution, but it is used for combining the HTTP request body into a single string. The req.on('end' .. callback will be called once the whole request has been received. We are going to move this logic to middleware so that it attaches the body to the request object once it’s fully received and then calls next.

Open a new call to app.use below the previous middleware. Make sure that it matches all routes for '/beans/' and '/beans/:beanName' using the array of routes syntax. You can leave your callback function body empty for now.

Checkpoint 2 Passed

**2.**

Now, copy the lines from the bodyData variable declaration to the end of the first req.on call into your middleware callback.

Checkpoint 3 Passed

Hint

Copy these lines into your middleware:

let bodyData = '';

req.on('data', (data) => {

bodyData += data;

});

**3.**

The next step will be a bit different from the routes that are already present. Add req.on('end', () => {}). Complete the callback by adding the following lines inside the body of the callback function:

if (bodyData) {

req.body = JSON.parse(bodyData);

}

This will parse the request body into a JavaScript object and attach it to the request object. Finish the middleware by calling next at the end of the req.on('end') callback function outside of the if statement.

Checkpoint 4 Passed

**4.**

Now to refactor! You can remove the lines

let bodyData = '';

req.on('data', (data) => {

bodyData += data;

});

from all your routes. Then you can remove the req.on('end' ...) method calls, but you’ll need to preserve the callback functions’ internal logic. You can simply remove the lines with req.on(... and the }); line at the end of the method call. Do this for all routes that have this duplicate code.

Checkpoint 5 Passed

**5.**

To finish refactoring, you can replace all instances of JSON.parse(bodyData) in the same routes and replace them with req.body since the body has already been parsed!

Checkpoint 6 Passed

**6.**

Great job, you removed duplicate code from four routes!

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bag with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

// Add your code below:

app.use(['/beans/', '/beans/:beanName'], (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    if (bodyData) {

      req.body = JSON.parse(bodyData);

    }

    next();

  });

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!req.bean) {

    return res.status(404).send('Bag with that name does not exist');

  }

  req.bean = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!req.bean) {

    return res.status(404).send('Bag with that name does not exist');

  }

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  req.bean = null;

  res.send(jellybeanBag[newName]);

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

# Middleware Stacks

Recall that middleware is just a function with a specific signature, namely (req, res, next). We have, for the most part, been using anonymous function definitions for this because our middleware has only been relevant to the route invoking it. There is nothing stopping us from defining functions and using them as middleware though. That is to say:

const logging = (req, res, next) => {

console.log(req);

next();

};

app.use(logging);

is a valid and reasonable way to introduce logging throughout all paths. It is also modifiable so that you can remove the app.use() line and replace it with a specific route method, or sprinkle it throughout the application without it being universal.

Up until this point we’ve only been giving each middleware-accepting method a single callback. With modular pieces like this, it is useful to know that methods such as app.use(), app.get(), app.post(), and so on all can take multiple callbacks as additional parameters. This results in code that looks like the following:

const authenticate = (req, res, next) => {

...

};

const validateData = (req, res, next) => {

...

};

const getSpell = (req, res, next) => {

res.status(200).send(getSpellById(req.params.id));

};

const createSpell = (req, res, next) => {

createSpellFromRequest(req);

res.status(201).send();

};

const updateSpell = (req, res, next) => {

updateSpellFromRequest(req);

res.status(204).send();

}

app.get('/spells/:id', authenticate, getSpell);

app.post('/spells', authenticate, validateData, createSpell);

app.put('/spells/:id', authenticate, validateData, updateSpell);

In the above code sample, we created reusable middleware for authentication and data validation. We use the authenticate() middleware to verify a user is logged in before proceeding with the request and we use the validateData() middleware before performing the appropriate create or update function. Additional middleware can be placed at any point in this chain.

**Instructions**

**1.**

Since we don’t need any request body for GET or DELETE routes, let’s refactor the behavior of our body-parsing middleware to use the in-route middleware stack. Start by saving the body-parsing middleware to a const variable bodyParser and removing the app.use call handling body parsing for ['/beans/', '/beans/:beanName'].

Checkpoint 2 Passed

Hint

To extract a middleware function and save to a variable, you can use this refactoring pattern:

Initial code:

app.use((req, res, next) => {

res.send('Cool data!');

});

Refactored:

const sendCoolResponse = (req, res, next) => {

res.send('Cool data!');

});

app.get(sendCoolResponse);

**2.**

Now, insert the bodyParser as the first callback for all routes handling POST requests.

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

const bodyParser = (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    if (bodyData) {

      req.body = JSON.parse(bodyData);

    }

    next();

  });

};

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', bodyParser, (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/remove', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

# Open-Source Middleware: Logging

Knowing how to write middleware, we should now feel inspired to solve all the problems that come at us by writing code. It’s encouraging to know how to fix an issue. If we find a solution we don’t need to write, however, it will allow us to work faster and more intelligently to focus on the problems that differentiate our application from others.

To illustrate: if we needed to write a web server from scratch every time we wanted to build a web application, we’d waste a lot of time solving problems that have been solved countless times before and ignoring perfectly good pre-existing solutions. Luckily for us web developers, Express already exists as an open-source package that we can install and use to build upon. There is a huge ecosystem of Javascript packages that will solve so many of the problems that developers frequently run into.

In the workspace you’ll see what code looks like using unnecessary custom solutions and lots of lines calling console.log(). It’s not bad code, but it introduces complexity that could be avoided. Time spent thinking about and writing code that accomplishes common tasks is time that could be better spent on thinking about and writing code that is unique to your application.

We will replace the logging code in the workspace with [morgan](https://github.com/expressjs/morgan" \t "_blank), an open-source library for logging information about the HTTP request-response cycle in a server application. morgan() is a function that will return a middleware function, to reiterate: the return value of morgan() will be a function, that function will have the function signature (req, res, next) that can be inserted into an app.use(), and that function will be called before all following middleware functions. Morgan takes an argument to describe the formatting of the logging output. For example, morgan('tiny') will return a middleware function that does a “tiny” amount of logging. With morgan in place, we’ll be able to remove the existing logging code. Once we see how fast it is to add logging with morgan, we won’t have to spend time in the future trying to figure out how to replicate that functionality.

**Instructions**

**1.**

Require morgan at the top of the app where you import Express, and save it to a const morgan.

Checkpoint 2 Passed

**2.**

Replace your logging middleware with morgan('tiny').

Checkpoint 3 Passed

Hint

Use morgan('tiny') inside an app.use call.

**3.**

Morgan will log response codes after the response is sent, so you can get rid of all the remaining console.log statements that log 'Response Sent'. At the end of this refactor, you should only have one console.log left in your code (inside app.listen).

const express = require('express');

const morgan = require('morgan');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

const bodyParser = (req, res, next) => {

  let queryData = '';

  req.on('data', (data) => {

    data = data.toString();

    queryData += data;

  });

  req.on('end', () => {

    if (queryData) {

      req.body = JSON.parse(queryData);

    }

    next();

  });

};

// Logging Middleware

app.use(morgan('tiny'));

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

});

app.post('/beans/', bodyParser, (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

});

app.post('/beans/:beanName/add', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

});

app.post('/beans/:beanName/remove', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

});

app.put('/beans/:beanName/name', bodyParser, (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

# Documentation

With software we’ve personally written, invocation is a simple process. We already know what the code does, what it expects, and may have some notion how things could go wrong. Losing this intuition is the biggest downside to using open-source packages.

This is not meant to be discouraging. The best open-source packages have extremely well written documentation. Documentation is a resource, presented by the package’s author(s), that includes information about what software is, what it does, and how to use it. We’ve seen the Express documentation in this course, and now we’re going to look at the [morgan documentation](https://github.com/expressjs/morgan" \l "api" \t "_blank).

**Instructions**

**1.**

Morgan provides a number of pre-defined formats. Let’s change from using 'tiny' to one with a bit more information. We want one with the HTTP method, URL, status code, response time, content length, and one that changes colors of the status code output based on the code. Try to find the logging format that will create output as we expect and replace 'tiny' with that format name.

Hint

If you look through the docs, you’ll notice ‘dev’ meets our criteria. Let’s use that!

app.use(morgan('dev'));

# Open-Source Middleware: Body Parsing

Being able to use open-source middleware can certainly make our jobs as programmers a lot easier. Not only does it prevent us from having to write the same code every time we want to accomplish a common task, it allows us to perform some tasks that would take a lot of research for us to implement.

When we implement middleware, we take in the req object, so that we can see information about the request. This object includes a good deal of important information about the request that we can use to inform our response, however for some requests it misses a fundamental piece. An HTTP request can include a body, a set of information to be transmitted to the server for processing. This is useful when the end user needs to send information to the server. If you’ve ever uploaded a post onto a social media website or filled out a registration form chances are you’ve sent an HTTP request with a body. The lucky thing about using open-source middleware is that even though parsing the body of an HTTP request is a tricky operation requiring knowledge about network data transfer concepts, we easily manage it by importing a library to do it for us.

If we look at our bodyParser, we see a simplified version of how one might perform request body parsing. Let’s see if there’s a better way that doesn’t involve us trying to create our own body-parser. Maybe we can find a library that does it for us?

Take a look at [body-parser](https://github.com/expressjs/body-parser#body-parser). “Node.js body parsing middleware”, that’s just what we needed! Let’s see if we can use this dependency instead of trying to manage our own body-parsing library.

**Instructions**

**1.**

Our bodyParser function is okay for now, but there are bound to be edge cases and all sorts of request bodies it can’t handle well. Let’s replace it with a well-maintained open-source package, body-parser. Require 'body-parser' at the top of the app in the same way, and save it to a const bodyParser.

Checkpoint 2 Passed

Hint

The syntax for importing 'body-parser' should mirror your syntax for importing 'morgan'.

**2.**

Remove the bodyParser middleware that you previously wrote. You can also now remove it from the middleware stacks for all PUT and POST routes. bodyParser will automatically attach the parsed body object to req.body.

Open a new app.use call directly after your morgan logging middleware. bodyParser has multiple [methods](https://github.com/expressjs/body-parser#api) for returning middleware functions. For now, let’s use bodyParser.json() to parse all request bodies in JSON format.

const express = require('express');

const app = express();

const morgan = require('morgan');

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use(morgan('dev'));

// Body parsing middleware

app.use(bodyParser.json());

app.use('/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

});

app.post('/beans/', (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

# Error-Handling Middleware

We’re almost finished with our Code Quality Checklist, there’s just one last problem to fix! When an error is thrown somewhere in our code, we want to be able to communicate that there was a problem to the user. A programming error is never something to be ashamed of. It’s simply another situation for which we should be prepared.

Error handling middleware needs to be the last app.use() in your file. If an error happens in any of our routes, we want to make sure it gets passed to our error handler. The middleware stack progresses through routes as they are presented in a file, therefore the error handler should sit at the bottom of the file. How do we write it?

app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).send('Something broke!');

});

Based on the code above, we can see that error-handling middleware is written much like other kinds of middleware. The biggest difference is that there is an additional parameter in our callback function, err. This represents the error object, and we can use it to investigate the error and perform different tasks depending on what kind of error was thrown. For now, we only want to send an HTTP 500 status response to the user.

Express has its own error-handler, which catches errors that we haven’t handled. But if we anticipate an operation might fail, we can invoke our error-handling middleware. We do this by passing an error object as an argument to next(). Usually, next() is called without arguments and will proceed through the middleware stack as expected. When called with an error as the first argument, however, it will call any applicable error-handling middleware.

app.use((req, res, next) => {

const newValue = possiblyProblematicOperation();

if (newValue === undefined) {

let undefinedError = new Error('newValue was not defined!');

return next(undefinedError);

}

next();

});

app.use((err, req, res, next) => {

const status = err.status || 500;

res.status(status).send(err.message);

});

In this segment we assign the return value of the function possiblyProblematicOperation() to newValue. Then we check to see if this function returned anything at all. If it didn’t, we create a new Error and pass it to next(). This prompts the error-handling middleware to send a response back to the user, but many other error-handling techniques could be employed (like logging, re-attempting the failed operation, and/or emailing the developer).

**Instructions**

**1.**

Add a very simple error handler as the last handler in your file, immediately before app.listen. The callback function should have four arguments. It should set the status of the response equal to the status property of the error object if it exists or set it to 500 by default. Finally, your error handler should send back the error object’s message property.

If you want to see your errors in the terminal console as you test, log out the error or it’s message property inside your error handler.

Checkpoint 2 Passed

**2.**

Now, refactor the routes that send error responses (any that are greater than or equal to 400) to use this error handler. This means instead of a line like this

return res.status(404).send('<error message>');

You should instead create a new Error with the correct error message, set its .status property, and then call next and pass in the error. Be sure to still return the next call so that the route/middleware callback breaks out and the error handler takes over.

Checkpoint 3 Passed

Hint

An example of this refactoring might look like this:

Before:

return res.status(404).send('error!');

After:

const err = new Error('error!');

err.status = 400;

return next(err);

# Discovering Open-Source Middleware

While it’s good to know how to write error-handling middleware, it’s a natural curiosity that causes us to ask “isn’t error-handling a common task? Has someone written middleware that performs it for us?” Let’s take a look at the [list of Express middleware](https://expressjs.com/en/resources/middleware.html). This list of middleware includes many things the creators of Express maintain, some of which was included in Express in previous versions. The movement on the Express team’s part to identify separate functionality and modularize their code into independent factors allows developers like us to only take what we need. In this way, they can make major updates to each middleware individually and programmers who do not use that middleware won’t have to worry about their version of Express being out of date.

Can you find something on that list that will help us handle errors?

**Instructions**

**1.**

Require the error handling package that you found. Replace your catch-all error handler with the middleware created by the error handler package. You can use the default settings.

Checkpoint 2 Passed

Hint

Looking through the Express website, you’ll find [this](https://github.com/expressjs/errorhandler) package.

const express = require('express');

const app = express();

const morgan = require('morgan');

const bodyParser = require('body-parser');

const errorHandler = require('errorhandler')

app.use(errorHandler());

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Body-parsing Middleware

app.use(bodyParser.json());

// Logging Middleware

app.use(morgan('dev'));

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

});

app.post('/beans/', (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bean with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

});

app.use((err, req, res, next) => {

  res.status(500).send(err);

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

# Review

We’ve accomplished a lot! We learned what middleware is and we’ve used it to write cleaner, readable, adaptable, and maintainable code. We’ve written functions that are context aware and can have overlapping functionality without duplicating code. We can serve data by route, with each possible endpoint being treated as a separate relative of the family of our application. We learned to link these middleware using next() to continue to the next middleware in the stack. We’ve reduced complexity in our codebase by relying on external, open-source middleware. We are truly harnessing the power of the Express web server, the Node environment, and our knowledge of Javascript. Let’s review those skills.

In the workspace there is another codebase with a set of familiar problems. Custom middleware to accomplish tasks we could be importing a module for. Duplicated code throughout the different routes. Improperly managed middleware stack missing next() calls. You will need everything learned in this lesson, but it’s nothing you haven’t done before.

**Instructions**

**1.**

There’s something missing from the custom body-parsing middleware function, fix it to move on.

Checkpoint 2 Passed

**2.**

Now our app should work, but let’s replace the custom body-parsing with the body-parser package. Use bodyParser.json() for all routes.

Checkpoint 3 Passed

**3.**

The following existence checking logic appears at all /cards/:cardId routes. Refactor it to a middleware function that matches all /cards/:cardId routes. If the card exists, add it to the req object as req.cardIndex and refactor routes to use req.cardIndex where necessary.

const cardId = Number(req.params.cardId);

const cardIndex = cards.findIndex(card => card.id === cardId);

if (cardIndex === -1) {

return res.status(404).send('Card not found');

}

Checkpoint 4 Passed

**4.**

You probably noticed that these lines get replicated for POST and PUT requests

const newCard = req.body;

const validSuits = ['Clubs', 'Diamonds', 'Hearts', 'Spades'];

const validRanks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace'];

if (validSuits.indexOf(newCard.suit) === -1 || validRanks.indexOf(newCard.rank) === -1) {

return res.status(400).send('Invalid card!');

}

Create a custom middleware function named validateCard that replicated this logic. Use it in the middleware stack for the POST and PUT routes.

Checkpoint 5 Passed

**5.**

Congratulations! You’ve fixed the broken functionality and greatly increased the code quality using your middleware skills!

const express = require('express');

const app = express();

const morgan = require('morgan');

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const cards = [

  {

    id: 1,

    suit: 'Clubs',

    rank: '2'

  },

  {

    id: 2,

    suit: 'Diamonds',

    rank: 'Jack'

  },

  {

    id: 3,

    suit: 'Hearts',

    rank: '10'

  }

];

let nextId = 4;

// Logging

if (!process.env.IS\_TEST\_ENV) {

  app.use(morgan('short'));

}

// Parsing

app.use(bodyParser.json());

// Find card

app.use('/cards/:cardId', (req, res, next) => {

  const cardId = Number(req.params.cardId);

  const cardIndex = cards.findIndex(card => card.id === cardId);

  if (cardIndex === -1) {

    return res.status(404).send('Card not found');

  }

  req.cardIndex = cardIndex;

  next();

});

const validateCard = (req, res, next) => {

  const newCard = req.body;

  const validSuits = ['Clubs', 'Diamonds', 'Hearts', 'Spades'];

  const validRanks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace'];

  if (validSuits.indexOf(newCard.suit) === -1 || validRanks.indexOf(newCard.rank) === -1) {

    return res.status(400).send('Invalid card!');

  }

  next();

};

// Get all Cards

app.get('/cards/', (req, res, next) => {

  res.send(cards);

});

// Create a new Card

app.post('/cards/', validateCard, (req, res, next) => {

  const newCard = req.body;

  newCard.id = nextId++;

  cards.push(newCard);

  res.status(201).send(newCard);

});

// Get a single Card

app.get('/cards/:cardId', (req, res, next) => {

  res.send(cards[req.cardIndex]);

});

// Update a Card

app.put('/cards/:cardId', validateCard, (req, res, next) => {

  const newCard = req.body;

  const cardId = Number(req.params.cardId);

  if (!newCard.id || newCard.id !== cardId) {

    newCard.id = cardId;

  }

  cards[req.cardIndex] = newCard;

  res.send(newCard);

});

// Delete a Card

app.delete('/cards/:cardId', (req, res, next) => {

  cards.splice(req.cardIndex, 1);

  res.status(204).send();

});

// Start the server

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

# Introduction

When building interfaces with Express, we will run into the pattern of expecting certain information in a requested URL and using that information to identify the data that is being requested. To give an example:

app.get('/sorcerers/:sorcererName', (req, res, next) => {

const sorcerer = Sorcerers[req.params.sorcererName];

res.send(sorcerer.info);

});

app.get('/sorcerers/:sorcererName/spellhistory', (req, res, next) => {

const sorcerer = Sorcerers[req.params.sorcererName];

const spellHistory = Spells[sorcerer.id].history;

res.send(spellHistory);

});

In the above code we need to extract the request parameter :sorcererName from the url in both instances, and end up duplicating the necessary code so that it appears in both routes. When working with routes that require parameters, we might find ourselves in a position where multiple different routes require the same parameter and use it to identify the same piece of data. While writing this duplicate code will get the job done, copy-and-pasting functionality does leave a bitter taste in the mouth of the principled developer. We should investigate if there is a better way to accomplish this.

const express = require('express');

const app = express();

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const spiceRack = [

  {

    id: 1,

    name: 'cardamom',

    grams: 45,

  },

  {

    id: 2,

    name: 'pimentn',

    grams: 20,

  },

  {

    id: 3,

    name: 'cumin',

    grams: 450,

  },

  {

    id: 4,

    name: 'sichuan peppercorns',

    grams: 107,

  }

];

let nextSpiceId = 5;

app.use(bodyParser.json());

// Add your code here:

app.param('spiceId', (req, res, next, id) => {

  const spiceId = Number(id);

  const spiceIndex = spiceRack.findIndex(spice => spice.id === spiceId);

  if (spiceIndex !== -1){

    req.spiceIndex = spiceIndex;

    next();

  } else {

    res.sendStatus(404);

  }

})

app.get('/spices/', (req, res, next) => {

  res.send(spiceRack);

});

app.post('/spices/', (req, res, next) => {

  const newSpice = req.body.spice;

  if (newSpice.name  && newSpice.grams) {

    newSpice.id = nextSpiceId++;

    spiceRack.push(newSpice);

    res.send(newSpice);

  } else {

    res.status(400).send();

  }

});

app.get('/spices/:spiceId', (req, res, next) => {

  res.send(spiceRack[req.spiceIndex]);

});

app.put('/spices/:spiceId', (req, res, next) => {

  spiceRack[req.spiceIndex] = req.body.spice;

  res.send(spiceRack[req.spiceIndex]);

});

app.delete('/spices/:spiceId', (req, res, next) => {

  spiceRack.splice(req.spiceIndex, 1);

  res.status(204).send();

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

# Merge Parameters

Complexity is all around us. Buildings are made from bricks and many droplets of water make a cloud. When we want to create something complex in software, we model out our base components and use composition to create these relationships.

When we’re building web endpoints, we might want to access some property of a complex object. In order to do this in Express, we can design a nested router. This would be a router that is invoked within another router. In order to pass parameters the parent router has access to, we pass a special configuration object when defining the router.

const sorcererRouter = express.Router();

const familiarRouter = express.Router({mergeParams: true});

sorcererRouter.use('/:sorcererId/familiars', familiarRouter);

sorcererRouter.get('/', (req, res, next) => {

res.status(200).send(Sorcerers);

next();

});

sorcererRouter.param('sorcererId', (req, res, next, id) => {

const sorcerer = getSorcererById(id);

req.sorcerer = sorcerer;

next();

});

familiarRouter.get('/', (req, res, next) => {

res.status(200).send(`Sorcerer ${req.sorcerer} has familiars ${getFamiliars(sorcerer)}`);

});

app.use('/sorcerer', sorcererRouter);

In the code above we define two endpoints: /sorcerer and /sorcerer/:sorcererId/familiars. The familiars are nested into the sorcerer endpoint — indicating the relationship that a sorcerer has multiple familiars. Take careful note of the {mergeParameters: true} argument that gets passed when creating the familiarRouter. This argument tells Express that the familiarRouter should have access to parents passed into its parent router, that is, the sorcererRouter. We then tell express that the path for the familiarRouter is the same as the path for the sorcererRouter with the additional path /:sorcererId/familiars. We then can create a family of routes (a router) built by appending routes to familiarRouter‘s base: /sorcerer/:sorcererId/familiars.

**Instructions**

**1.**

Let’s make our spices API more flexible and allow spices to be associated with different spice racks. The goal for this exercise will be to ensure that when new spices are created or updated, they will be associated with the correct spice rack.

In the workspace, you have a new root **app.js** file and a **spicesRouter.js** with code from the last exercise. **app.js** will handle interactions retrieving, creating, updating, and deleting spice racks, and **spicesRouter.js** will be nested to handle individual spices with the spice racks. Each file has a param method call (app.param in **app.js**, router.param in **spicesRouter.js**).

To begin, let’s hook the router up to the main application. At the end of **app.js**, use the spicesRouter for all /spice-racks/:spiceRackId/spices routes.

Checkpoint 2 Passed

**2.**

Now, let’s make sure that the spicesRouter is merging parameters from parent **app.js** Express instance. Add the proper options to the .Router() method at the top of your **spicesRouter.js** file.

Checkpoint 3 Passed

**3.**

Okay, now let’s make sure that newly created spices inside **spicesRouter.js** are associated with the correct spice rack. Inside your .post() route, make sure to set the newSpice.spiceRackId equal to the req.params.spiceRackId that the parent router attached if mergeParams has performed as expected. Don’t forget to convert the spiceRackId to a number before attaching it. Make sure to set this before it is pushed onto the spices array.

# Review

router.param is a powerful tool that we can use to keep our code from repeating core functionality through routes. This is a pattern we want to frequently follow: identify multiple pieces of code that accomplish the same goal, put it into a single component, let that component do that thing (and update it when we want the thing it does to change — in a single place).

Let’s try applying that knowledge again, to another codebase. If you look at the workspace you’ll find the same problem of data-lookup happening, based on a URL parameter, multiple times in different places. Try combining that logic in a single place using router.param.

**Instructions**

**1.**

Refactor the current application to use an app.param to handle all routes with snackId. It should set the req.snackIndex if it exists and send the proper 404 response if not. Make sure to fix all routes to use the req.snackIndex and remove duplicate code.

Hint

Review the previous exercise if you want to remember the steps to refactor a route to use app.param.

# Code Challenge

Code Challenge

**Instructions**

**1.**

This server currently sends the same response for ‘/whatis/apple’, ‘/whatis/orange’, and ‘/whatis/banana’. Write a function to send the same response, and DRY the code by using that function as a callback for all three routes.

Checkpoint 2 Passed

Hint

const sendFruitResponse = (req, res, next) => {

res.send('fruit');

}

app.get('/whatis/apple', sendFruitResponse);

app.get('/whatis/orange', sendFruitResponse);

app.get('/whatis/banana', sendFruitResponse);

const express = require('express');

const app = express();

// Finish this function and use it in the routes below

const sendFruitResponse = (req, res, next) => {

  res.send('fruit');

}

app.get('/whatis/apple', sendFruitResponse);

app.get('/whatis/orange', sendFruitResponse);

app.get('/whatis/banana', sendFruitResponse);

# Code Challenge

Code Challenge

**Instructions**

**1.**

The current moodleware middleware function should attach a mood property to the request object on every request and then continue on with the rest of the route matching. Something is missing! Fix it so that it moves on after attaching the req.mood.

Hint

const moodleware = (req, res, next) => {

const randomMoodIndex = Math.floor(Math.random() \* moods.length);

req.mood = moods[randomMoodIndex];

// Don't forget next!

next();

}

const express = require('express');

const app = express();

const moods = ['happy', 'exuberant', 'fanatical about middleware'];

const moodleware = (req, res, next) => {

  const randomMoodIndex = Math.floor(Math.random() \* moods.length);

  req.mood = moods[randomMoodIndex];

  next();

}

app.use(moodleware);

app.get('/randomMood', (req, res, next) => {

  res.send(req.mood);

});

1. Complete the timeMiddleware function to DRY the current application by attaching the currentTime to the request body as date. Use this in all routes and eliminate the duplicate code.

const express = require('express');

const app = express();

const database = {

  snacks: ['chips', 'apple', 'cookies'],

  sides: ['beans and rice', 'cole slaw', 'broccoli'],

  appetizers: ['oysters', 'dumplings', 'smoked almonds'],

};

// Add your code here:

const timeMiddleware = (req, res, next) => {

  req.date = Date.now();

  next();

};

app.use(timeMiddleware);

app.get('/snacks', (req, res, next) => {

  res.send(`Snacks as of ${req.date}: ${database.snacks}`);

});

app.get('/sides', (req, res, next) => {

  res.send(`Sides as of ${req.date}: ${database.sides}`);

});

app.get('/appetizers', (req, res, next) => {

  res.send(`Appetizers as of ${req.date}: ${database.appetizers}`);

});

**1.**

The server will handle requests for existing food indices as expected, but it will call next with an error in response to a request for a non-existent food.

Complete the errorHandler function as an Express [error handling function.](http://expressjs.com/en/guide/error-handling.html) It should set the response status equal to the error object’s status property if it exists, and 500 by default if there is no status on the error object. It should send the message property of the error object as its response.

Hint

const errorHandler = (err, req, res, next) => {

if (!err.status) {

err.status = 500;

}

res.status(err.status).send(err.message);

}

const express = require('express');

const app = express();

const foods = ['pasta carbonara', 'bnh m', 'cucumber salad'];

app.get('/foods/:index', (req, res, next) => {

  if (foods[req.params.index]) {

    res.send(req.params.index);

  } else {

    const err = Error('Invalid index!');

    err.status = 404;

    next(err);

  }

});

const errorHandler = (err, req, res, next) => {

  if (!err.status) {

    err.status = 500;

  }

  res.status(err.status).send(err.message);

}

app.use(errorHandler);

**1.**

Import the [morgan](https://github.com/expressjs/morgan" \t "_blank) logging middleware to log all requests. Use the [predefined format](https://github.com/expressjs/morgan#predefined-formats) ‘short’ instead of ‘tiny’.

Hint

const morgan = require('morgan');

app.use(morgan('short'));

const express = require('express');

const app = express();

// Add your code below:

const morgan = require('morgan');

app.use(morgan('short'));

// Add your code above

app.get('/say-hi', (req, res, next) => {

  res.send('Hi!');

});

app.get('/say-bye', (req, res, next) => {

  res.send('Bye!');

});

**1.**

Use app.use and the indexExists middleware function for all /data/:index paths.

Hint

app.use('/data/:index', indexExists);

const express = require('express');

const app = express();

const data = [1, 2, 3, 4, 5];

const indexExists = (req, res, next) => {

  const index = req.params.index;

  if (data[index]) {

    next();

  } else {

    res.status(404).send();

  }

};

// Add your code below:

app.use('/data/:index', indexExists);

// Add your code above

app.get('/data/:index', (req, res, next) => {

  res.send(data[req.params.index]);

});

app.put('/data/:index', (req, res, next) => {

  data[req.params.index] = req.body.number;

  res.send(data[req.params.index]);

});

app.delete('/data/:index', (req, res, next) => {

  data.splice(req.params.index, 1);

  res.status(204).send();

});

app.get('/', (req, res, next) => {

  res.send(data);

});

**1.**

Use the Express [middleware resources](https://expressjs.com/en/resources/middleware.html) to find an appropriate middleware package to handle [CORS requests](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing).

Import the correct middleware package and use it for all requests. You can use the default settings for this middleware, no need to use any customization!

Hint

const cors = require('cors');

app.use(cors());

const express = require('express');

const app = express();

const cors = require('cors');

app.use(cors());

**1.**

The current API has public data that should be accessible to anybody, and private data that should be only accessed by admin users. Currently, users can access any data, but the isAdmin middleware function should be able to help. Use this function in any route that accesses the secretData object.

Hint

app.use(['/:userId/phone-numbers', '/:userId/fav-sites'], isAdmin);

// Or:

app.get('/:userId/phone-numbers', isAdmin, (req, res, next) => {

res.send(secretData.coolPhoneNumbers);

});

app.get('/:userId/fav-sites', isAdmin, (req, res, next) => {

res.send(secretData.favSites);

});

const express = require('express');

const app = express();

const secretData = {

  adminUsers: ['1', '2', '51'],

  coolPhoneNumbers: ['555-555-CODE', '555-EXP-RESS', 'MID-DLE-WARE'],

  favSites: ['codecademy.com', 'expressjs.com']

}

const publicData = {

  colors: ['green', 'orange'],

  numbers: [1, 2, 3, 4, 5]

}

const isAdmin = (req, res, next) => {

  const id = req.params.userId;

  if (!secretData.adminUsers.includes(id)) {

    res.status(401).send(); // Unauthorized

  } else {

    next();

  }

}

app.get('/:userId/colors', (req, res, next) => {

  res.send(publicData.colors);

});

app.get('/:userId/numbers', (req, res, next) => {

  res.send(publicData.numbers);

});

app.get('/:userId/phone-numbers', isAdmin, (req, res, next) => {

  res.send(secretData.coolPhoneNumbers);

});

app.get('/:userId/fav-sites', isAdmin, (req, res, next) => {

  res.send(secretData.favSites);

});

**1.**

Use Router.param to DRY this appleRouter. Attach the apple variety to the request body if it exists, send a 404 response if it does not. In each individual route, send the appropriate property from that apple object.

Hint

appleRouter.param('variety', (req, res, next, variety) => {

if (!apples[variety]) {

res.status(404).send();

} else {

req.variety = apples[variety];

next();

}

});

appleRouter.get('/:variety/description', (req, res, next) => {

res.send(req.variety.description);

});

appleRouter.get('/:variety/price-range', (req, res, next) => {

res.send(req.variety.priceRange);

});

appleRouter.get('/:variety/color', (req, res, next) => {

res.send(req.variety.color);

});

const express = require('express');

const appleRouter = express.Router();

const apples = {

  mcintosh: {

    description: 'Classic, juicy, bright',

    priceRange: 'medium',

    color: 'green and red'

  },

  honeycrisp: {

    description: 'Crisp, sweet!',

    priceRange: 'pricey',

    color: 'red and yellow'

  },

  goldenDelicious: {

    description: 'rich, custardy',

    priceRange: 'cheap',

    color: 'yellow'

  }

}

appleRouter.param('variety', (req, res, next, variety) => {

  if (!apples[variety]) {

    res.status(404).send();

  } else {

    req.variety = apples[variety];

    next();

  }

});

appleRouter.get('/:variety/description', (req, res, next) => {

  res.send(req.variety.description);

});

appleRouter.get('/:variety/price-range', (req, res, next) => {

  res.send(req.variety.priceRange);

});

appleRouter.get('/:variety/color', (req, res, next) => {

  res.send(req.variety.color);

});