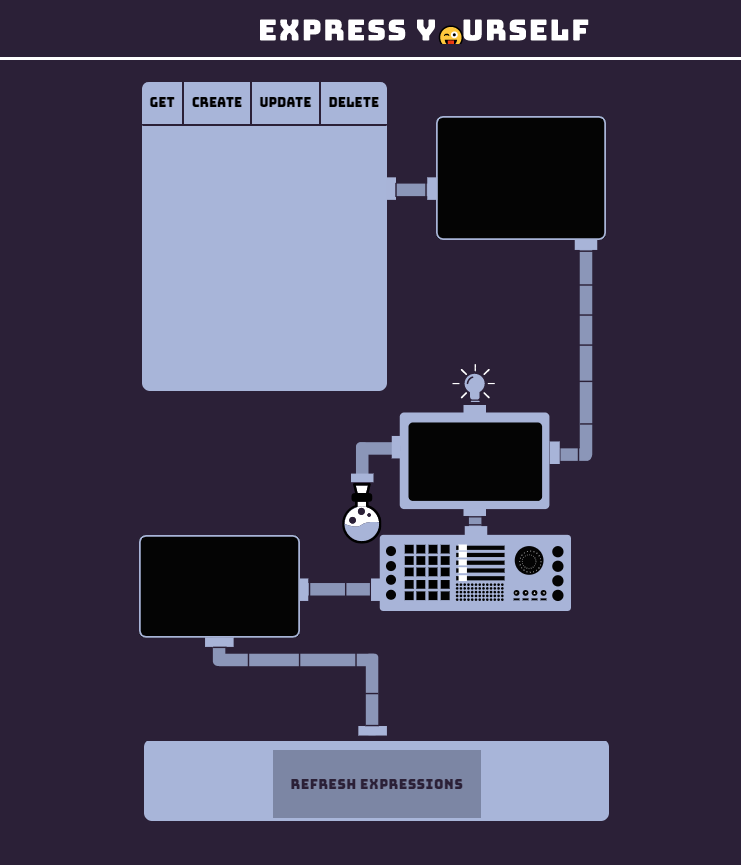
**Introduction**

A huge portion of the Internet’s data travels over HTTP/HTTPS through request-response cycles between [clients and servers](https://en.wikipedia.org/wiki/Client%E2%80%93server_model). Every time that you use a website, your browser sends requests to one or many servers requesting resources. Every image, meme, post, and video is requested by a client and sent in a response from a server.

Express is a powerful but flexible Javascript framework for creating web servers and [APIs](https://en.wikipedia.org/wiki/Web_API). It can be used for everything from simple static file servers to JSON APIs to full production servers.

In this lesson, you will be fixing a machine called Express Yourself in the browser. The machine is supposed to provide functionality for clients to interact with various Expressions: JavaScript objects each containing ids, names, and emojis. Currently, it looks nice, but nothing works since there is no server in place! You will be learning all the necessary skills to implement an API allowing clients to Create, Read, Update, and Delete Expressions. These four functionalities together are known as [CRUD](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete), and they form the backbone of many real-life APIs.



**Starting A Server**

Express is a Node module, so in order to use it, we will need to import it into our program file. To create a server, the imported express function must be invoked.

const express = require('express');

const app = express();

On the first line, we import the Express library with require. When invoked on the second line, it returns an instance of an Express application. This application can then be used to start a server and specify server behavior.

The purpose of a server is to listen for requests, perform whatever action is required to satisfy the request, and then return a response. In order for our server to start responding, we have to tell the server where to *listen* for new requests by providing a port number argument to a method called app.listen(). The server will then listen on the specified [port](https://en.wikipedia.org/wiki/Port_(computer_networking)) and respond to any requests that come into it.

The second argument is a callback function that will be called once the server is running and ready to receive responses.

const PORT = 4001;

app.listen(PORT, () => {

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

});

In this example, our app.listen() call will start a server listening on port 4001, and once the server is started it will log 'Server is listening on port 4001'.

**Instructions**

**1.**

Import express using require syntax and assign it to an express variable.

Create an instance of an Express server and save it to a variable named app.

Start the server listening on the port defined by the PORT variable. When the server has started, log a message to the console that the server is listening for requests.

To actually start your server listening, run the command node app.js to run your server in [Node](https://nodejs.org/en/). At this point, it won’t do much, but if you’ve completed the steps above, it will log your message to show that the server started successfully.

When you want to check that you have written your starting server code correctly, use the ‘Check Work’ button.

// Import the express library here

const express = require('express');

// Instantiate the app here

const app = express();

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

// Invoke the app's `.listen()` method below:

app.listen(PORT, () => {

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

});

**Writing Your First Route**

Once the Express server is listening, it can respond to any and all requests. But how does it know what to do with these requests? To tell our server how to deal with any given request, we register a series of *routes*. Routes define the control flow for requests based on the request’s *path* and HTTP verb.

For example, if your server receives a GET request at ‘/monsters’, we will use a route to define the appropriate functionality for that HTTP verb (GET) and path (/monsters).

The path is the part of a request URL after the [hostname](https://en.wikipedia.org/wiki/Hostname) and port number, so in a request to localhost:4001/monsters, the path is /monsters (in this example, the hostname is ‘localhost’, the port number is ‘4001’).

The HTTP verb is always included in the request, and it is one of a [finite number of options](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods) used to specify expected functionality. GET requests are used for retrieving resources from a server, and we will discuss additional request types in later exercises.

Express uses app.get() to register routes to match GET requests. Express routes (including app.get()) usually take two arguments, a path (usually a string), and a callback function to handle the request and send a response.

const moods = [{ mood: 'excited about express!'}, { mood: 'route-tastic!' }];

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

// Here we would send back the moods array in response

});

The route above will match any GET request to '/moods' and call the callback function, passing in two objects as the first two arguments. These objects represent the request sent to the server and the response that the Express server should eventually send to the client.

If no routes are matched on a client request, the Express server will handle sending a 404 Not Found response to the client.

**Instructions**

**1.**

Now that your server starting code should be working properly, you can start up the Express Yourself machine. Start your server from the terminal window with node app.js. Once it logs that it is running, you can refresh the browser window currently displaying Not Found.

Inside **app.js**, create a route handler to handle a GET request to '/expressions'. For now, give it a req, res, next callback. For now, log the req object inside the callback. Verify that the route works and logs the request by starting your server and clicking the Refresh Expressions button which will send a GET /expressions request.

We will complete this route in the next exercise and finish the first round of functionality to the Express Yourself machine.

You may notice that there’s a line with the command app.use(express.static('public'));. This is used to make sure that once the server is started, you can reload the browser and see the Express Yourself machine.

const express = require('express');

const app = express();

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

// Use static server to serve the Express Yourself Website

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

// Open a call to `app.get()` below:

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

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Sending A Response**

HTTP follows a one request-one response cycle. Each client expects exactly one response per request, and each server should only send a single response back to the client per request. The client is like a customer at a restaurant ordering a large bowl of soup: the request is sent through the wait staff, the kitchen prepares the soup, and after is it prepared, the wait staff returns it to the customer. In the restaurant, it would be unfortunate if the soup never arrived back to the customer, but it would be equally problematic if the customer was given four large bowls of soup and was asked to consume them all at the exact same time. That’s impossible with only two hands!

Express servers send responses using the .send() method on the response object. .send() will take any input and include it in the response body.

const monsters = [{ type: 'werewolf' }, { type: 'hydra' }, { type: 'chupacabra' }];

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

res.send(monsters);

});

In this example, a GET /monsters request will match the route, Express will call the callback function, and the res.send() method will send back an array of spooky monsters.

In addition to .send(), .json() can be used to explicitly send JSON-formatted responses. .json() sends any JavaScript object passed into it.

**Instructions**

**1.**

Send the expressions array from your app.get handler. Now that you have a complete route, you can test it out by reloading the browser window and clicking the ‘Refresh Expressions’ button on the machine.

If you make changes to **app.js**, you will need to restart your server to see the changes in effect. You can do this by pressing Ctrl + C in the terminal window to stop the old server, and you can start it again with node app.js.

Hint

You can use res.send() or res.json() to send the expressions array.

const express = require('express');

const app = express();

// Serves Express Yourself website

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

const { getElementById, seedElements } = require('./utils');

const expressions = [];

seedElements(expressions, 'expressions');

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

// Use static server to serve the Express Yourself Website

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

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

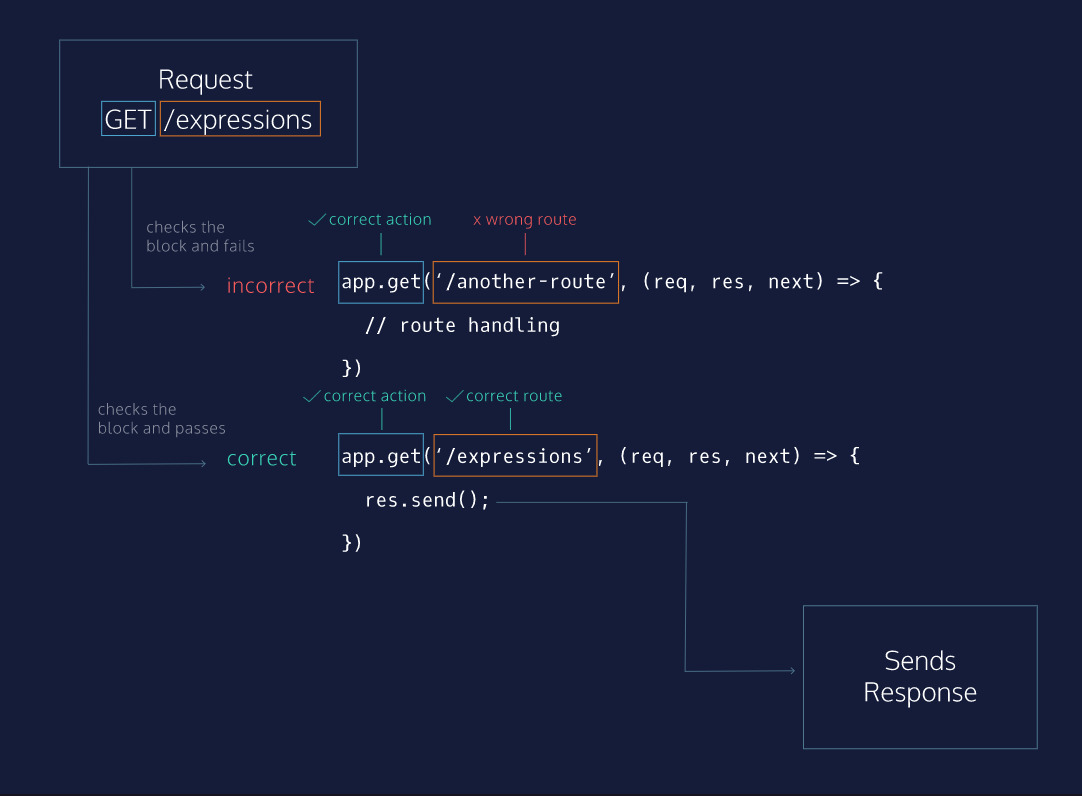
  res.send(expressions);

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});



**Getting A Single Expression**

Routes become much more powerful when they can be used dynamically. Express servers provide this functionality with named *route parameters*. Parameters are route path segments that begin with : in their Express route definitions. They act as [wildcards](https://expressjs.com/en/guide/routing.html#route-parameters), matching any text at that path segment. For example /monsters/:id will match both/monsters/1 and /monsters/45.

Express parses any parameters, extracts their actual values, and attaches them as an object to the request object: req.params. This object’s keys are any parameter names in the route, and each key’s value is the actual value of that field per request.

const monsters = { hydra: { height: 3, age: 4 }, dragon: { height: 200, age: 350 } };

// GET /monsters/hydra

app.get('/monsters/:name', (req, res, next) => {

console.log(req.params) // { name: 'hydra' };

res.send(monsters[req.params.name]);

});

In this code snippet, a .get() route is defined to match /monsters/:name path. When a GET request arrives for /monsters/hydra, the callback is called. Inside the callback, req.params is an object with the key name and the value hydra, which was present in the actual request path. The appropriate monster is retrieved by its name from the monsters object and sent back to the client.

**Instructions**

**1.**

Create a GET /expressions/:id get route that you will use to send back a single expression. You can use req.params object and the pre-written helper function getElementById(id, array) to find the correct expression before sending it back.

For instance, to find ID 560 from expressions, you would call getElementById(560, expressions);. This function returns the element object if it exists and undefined if it does not.

Don’t forget to restart your server when you make changes to **app.js**. To test the Express Yourself machine, use the box in the upper-left corner to send a GET request for a specified ID.

Checkpoint 2 Passed

Hint

The correct ID of the expression can be found using req.params.id.

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  res.send(foundExpression);

});

**Setting Status Codes**

Express allows us to set the [status code](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) on responses before they are sent. Response codes provide information to clients about how their requests were handled. Until now, we have been allowing the Express server to set status codes for us. For example, any res.send() has by default sent a 200 OK status code.

The res object has a .status() method to allow us to set the status code, and other methods like .send() can be chained from it.

const monsterStoreInventory = { fenrirs: 4, banshees: 1, jerseyDevils: 4, krakens: 3 };

app.get('/monsters-inventory/:name', (req, res, next) => {

const monsterInventory = monsterStoreInventory[req.params.name];

if (monsterInventory) {

res.send(monsterInventory);

} else {

res.status(404).send('Monster not found');

}

});

In this example, we’ve implemented a route to retrieve inventory levels from a Monster Store. Inventory levels are kept in the monsterStoreInventory variable. When a request arrives for /monsters-inventory/mothMen, the route matches and so the callback is invoked. req.params.name will be equal to 'mothMen' and so our program accesses monsterStoreInventory['mothMen']. Since there are no mothMen in our inventory,res.status() sets a 404 status code on the response, and .send() sends the response.

**Instructions**

**1.**

Let’s make sure that our GET /expressions/:id route handles invalid requests properly, for instance if we request an expression ID that does not exist.

Complete your route so that it sends back the correct expression object if it exists and sends back a 404 response if it does not.

Hint

Remember that getElementById returns an object for a valid ID and undefined for an invalid ID.

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send('undefined');

  }

});

# Matching Longer Paths

Parameters are extremely helpful in making server routes dynamic and able to respond to different inputs. Route parameters will match anything in their specific part of the path, so a route matching /monsters/:name would match all the following request paths:

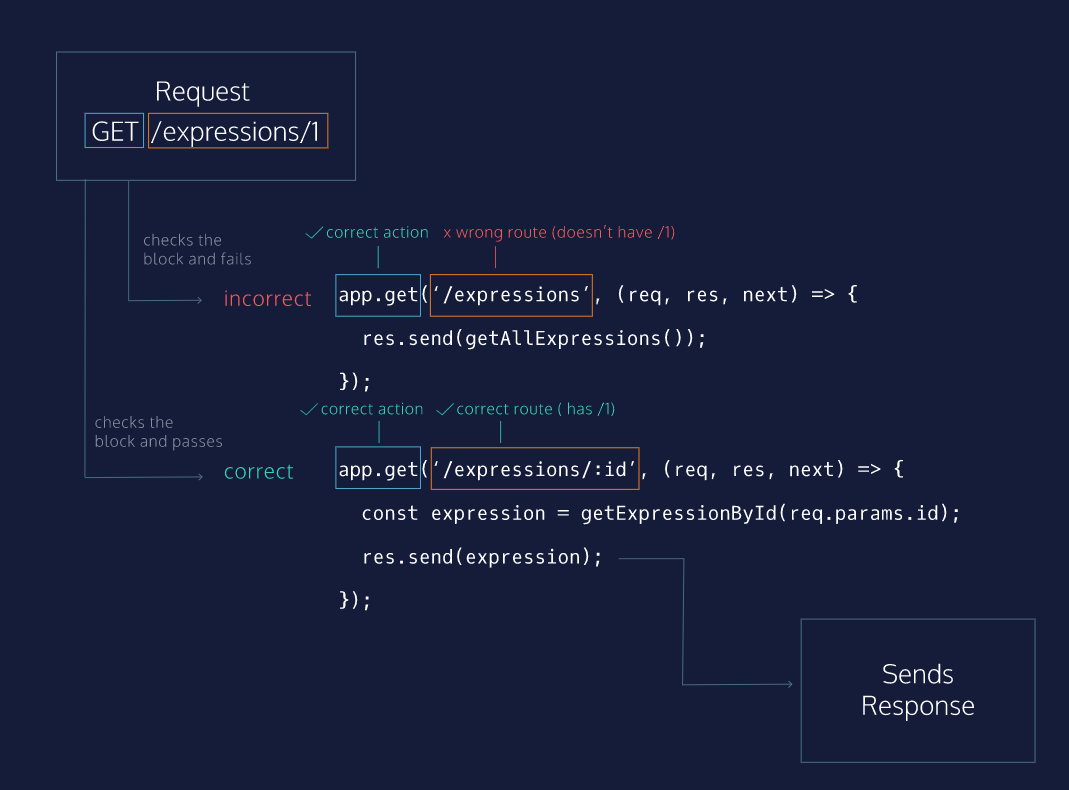
/monsters/hydra

/monsters/jörmungandr

/monsters/manticore

/monsters/123

In order for a request to match a route path, it must match the entire path, as shown in the diagram to the right. The request arrives for /expressions/1. It first tries to match the /expressions route, but because it has additional path segments after /expressions, it does not match this route and moves on to the next. It matches /expressions/:id because :id will match any value at that level of the path segment. The route matches, so the Express server calls the callback function, which in turn handles the request and sends a response.



**Other HTTP Methods**

[HTTP Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) defines a number of different method verbs with many use cases. So far, we have been using the GET request which is probably the most common of all. Every time your browser loads an image, it is making a GET request for that file!

This course will cover three other important HTTP methods: PUT, POST, and DELETE. Express provides methods for each one: app.put(), app.post(), and app.delete().

PUT requests are used for updating existing resources. In our Express Yourself machine, a PUT request will be used to update the name or emoji of an expression already saved in our database. For this reason, we will need to include a unique identifier as a route parameter to determine which specific resource to update.

**Instructions**

**1.**

For now, open a PUT /expressions/:id route with an empty (req, res, next) callback function. We will fully implement its functionality in the next exercise.

app.put('/expressions/:id',(req, res, next)=>{})

# Using Queries

You may have noticed in the previous exercise that our PUT route had no information about how to update the specified expression, just the id of which expression to update. It turns out that there was more information in the request in the form of a query string. [Query strings](https://en.wikipedia.org/wiki/Query_string) appear at the end of the path in URLs, and they are indicated with a ? character. For instance, in /monsters/1?name=chimera&age=1, the query string is name=chimera&age=1 and the path is /monsters/1/

Query strings do not count as part of the route path. Instead, the Express server parses them into a JavaScript object and attaches it to the request body as the value of req.query. The key: value relationship is indicated by the = character in a query string, and key-value pairs are separated by &. In the above example route, the req.query object would be { name: 'chimera', age: '1' }.

const monsters = { '1': { name: 'cerberus', age: '4' } };

// PUT /monsters/1?name=chimera&age=1

app.put('/monsters/:id', (req, res, next) => {

const monsterUpdates = req.query;

monsters[req.params.id] = monsterUpdates;

res.send(monsters[req.params.id]);

});

Here, we have a route for updating monsters by ID. When a PUT /monsters/1?name=chimera&age=1 request arrives, our callback function is called and, we create a monsterUpdates variable to store req.query. Since req.params.id is '1', we replace monsters['1']‘s value with monsterUpdates . Finally, Express sends back the new monsters['1'].

When updating, many servers will send back the updated resource after the updates are applied so that the client has the exact same version of the resource as the server and database.

**Instructions**

**1.**

Use req.query to update the proper element in the expressions array.

We’ve imported a helper function from **/utils.js** to help with this task.

You can use the updateElement() helper function in your PUT /expressions/:id route.

It takes three arguments:

* id (the ID number of the element)
* queryArguments (the new, updated expression object from req.query)
* elementList (the array which contains the element to update)

updateElement() updates that specific element in the elementList array (you’ll pass in the expressions array), and then returns the updated element.

Be sure to check that an expression with the id you provide exists in the expressions array (getIndexById() can help)!

To test your functionality with the Express Yourself machine, make sure your server is running, get all expressions, and then use the UPDATE tab to select an individual expression, select updates, and send the PUT request.

Hint

Remember:

* getIndexById will return -1 if the expressions array doesn’t contain an element with that id.
* updateElement will throw an error if you pass in an id that doesn’t exist in the array

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

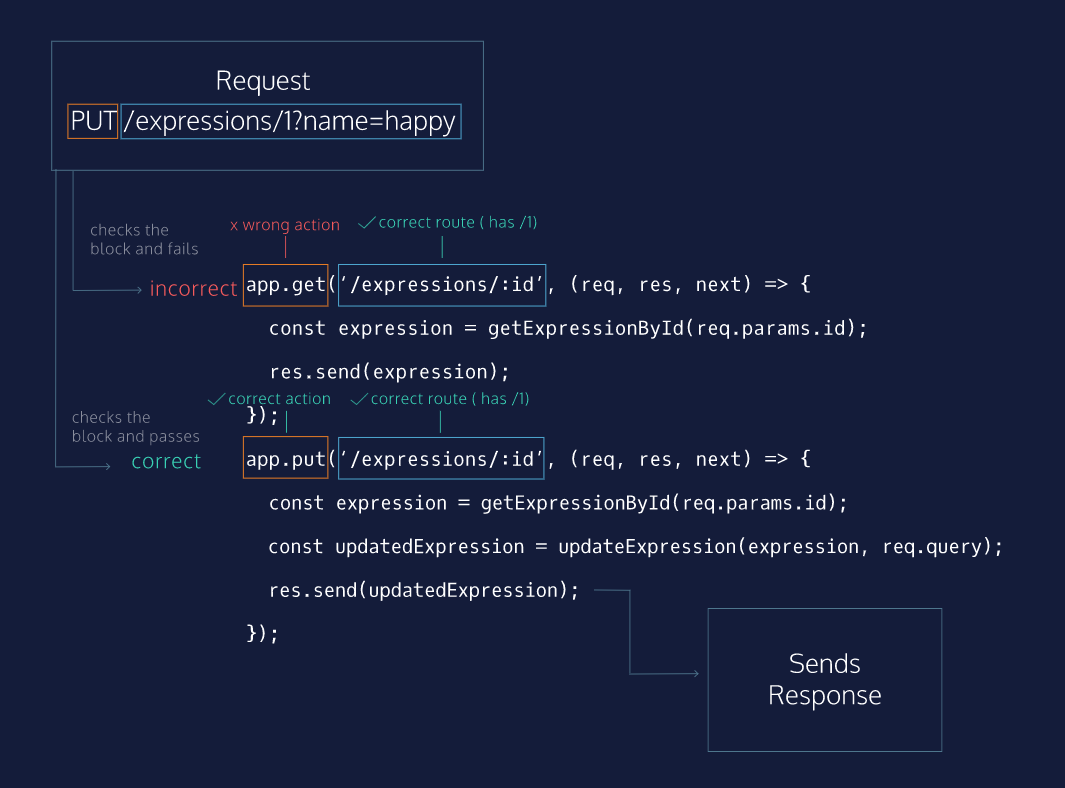
    res.status(404).send();

  }

});

# Matching By HTTP Verb

Express matches routes using both path and HTTP method verb. In the diagram to the right, we see a request with a PUT verb and /expressions (remember that the query is not part of the route path). The path for the first route matches, but the method verb is wrong, so the Express server will continue to the next registered route. This route matches both method and path, and so its callback is called, the necessary updating logic is executed, and the response is sent.



# Creating An Expression

POST is the HTTP method verb used for creating new resources. Because POST routes create new data, their paths do not end with a route parameter, but instead end with the type of resource to be created.

For example, to create a new monster, a client would make a POST request to /monsters. The client does not know the id of the monster until it is created and sent back by the server, therefore POST /monsters/:id doesn’t make sense because a client couldn’t know the unique id of a monster before it exists.

Express uses .post() as its method for POST requests. POST requests can use many ways of sending data to create new resources, including query strings.

The HTTP status code for a newly-created resource is 201 Created.

**Instructions**

**1.**

Create a POST /expressions route. It should send create and add a new expression to the expressions array if it is a valid new expression (meaning it has an emoji and name key). It should send back the new element with a 201 status code if it is valid, and it should send a 400 status code if the object is not valid.

You can use the createElement(elementType, objectToCreate) helper function to create a valid expression. The first argument is the type of element, so it should be 'expressions' in this case. The second argument should be the query object with an emoji and a name property. This function will return false if the objectToCreate does not contain all necessary key-value pairs, and it will return the newly-created element if object to create is valid. It does not add the created element to any arrays, you will need to do so yourself.

Don’t forget to restart your server and test as you implement the functionality. To test your route, use the POST tab in the upper left corner. Select a name and emoji and send the request to see if your route works as intended.

Hint

You can use the .push() method of the expressions array to add a new element after it has been created, for example:

const newElement = createElement('emoji', {name: 'example', emoji: ':)'});

if (newElement) {

elements.push(newElement);

}

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

# Deleting Old Expressions

DELETE is the HTTP method verb used to delete resources. Because DELETE routes delete currently existing data, their paths should usually end with a route parameter to indicate which resource to delete.

Express uses .delete() as its method for DELETE requests.

Servers often send a 204 No Content status code if deletion occurs without error.

**Instructions**

**1.**

Create a DELETE /expressions/:id route. It should send back a 404 response for a request with an invalid id, and it should delete the proper element from the expressions array and send a 204 status with a valid id.

To test your functionality, use the DELETE tab in the upper left. Select the ID to delete and send the request.

Checkpoint 2 Passed

Hint

You can use getIndexById to find the index of the element to delete. getIndexById will return -1 for a non-existent ID, and the proper index if it exists. Then you can use the [splice](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) method to remove the element.

app.delete('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

# DRYing Code With Functions

Beyond labeling, good code will leverage the strength of its programming language to avoid performing the same tasks

Take a look at the following code:

const addFive = number => {

const fiveAdded = number + 5;

console.log(`Your number plus 5 is ${fiveAdded}`);

}

const addTen = number => {

const tenAdded = number + 10;

console.log(`Your number plus 10 is ${tenAdded}`);

}

const addTwenty = number => {

const twentyAdded = number + 20;

console.log(`Your number plus 20 is ${twentyAdded}`);

}

The code above defines three different functions that accomplish the radically different tasks of: adding five to a number and logging the sum, adding ten to a number and logging the sum, and adding twenty to a number and logging the sum. While these three function definitions are not exact duplicates of each other, a well-designed application will be flexible enough to join similar functionality in a single element.

const addNumber = (number, addend) => {

const numAdded = number + addend;

console.log(`Your number plus ${addend} is ${numAdded}`);

}

As you can see, by adding an argument to the earlier functions we can simplify our application code which will ultimately save time should we realize that we also want an addFifty() function and an addHundred() function. Code that performs the same task in multiple places is repetitive, and the quality coder’s credo is “Don’t Repeat Yourself” (DRY). If a program performs similar tasks without refactoring into a function, it is said to “violate DRY”. “Violating DRY” is a programmer’s way of complaining: “This script is saying the same thing over and over! We can do the same thing with less code!” Let’s try to not repeat ourselves in this codebase by repurposing some of the more glaringly repeated code into functions we can call instead.

**Instructions**

**1.**

We have provided a front-end for testing out your routes throughout this lesson. To get it to display in each exercise, start your server (node app.js) and then refresh the browser to the right. A tool should appear that allows you to set request verbs, paths, and body information, and then make requests using that information. Use this tool to ensure your server is working as expected throughout this lesson, checking your server logs and examining the returned responses as you make changes.

Checkpoint 2 Passed

**2.**

Currently, each route logs a message with the HTTP method and a message that the request was received (i.e. 'GET Request Received'). Write a function logRequest that takes a single string parameter verb and logs a message formatted in the same fashion.

Checkpoint 3 Passed

**3.**

Replace the console.log calls that open each route and replace each with a call to logRequest. Pass in the method name for each route.

const express = require('express');

const app = express();

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

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

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Add your logging function here:

const logRequest = verb => {

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

}

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

  logRequest('GET');

  res.send(jellybeanBag);

  console.log('Response Sent');

});

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

  logRequest('POST');

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const body = JSON.parse(queryData);

    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) => {

  logRequest('GET');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

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

  }

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

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

  logRequest('POST');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

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

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(queryData).number) || 0;

    jellybeanBag[beanName].number += numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

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

  logRequest('POST');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

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

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(queryData).number) || 0;

    if (jellybeanBag[beanName].number < numberOfBeans) {

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

    }

    jellybeanBag[beanName].number -= numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

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

  logRequest('DELETE');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

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

  }

  jellybeanBag[beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

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

  logRequest('PUT');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

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

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const newName = JSON.parse(queryData).name;

    jellybeanBag[newName] = jellybeanBag[beanName];

    jellybeanBag[beanName] = null;

    res.send(jellybeanBag[newName]);

    console.log('Response Sent');

  });

});

app.listen(PORT, () => {

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

});

# Control Flow With next()

We’ve experienced writing middleware that performs its function and hands off the request and response objects to the next function in the stack, but why exactly do we have to write next() at the end of every middleware? If it always needs to be at the end of every function we write, it seems like an unnecessary piece of boilerplate. You might be surprised to learn that we aren’t going to introduce a way to automatically hand off the request and response objects without having to repeatedly write next(). Rather, we’re going to explore why it is useful to have next() as a separate function call. The biggest reason being we don’t always want to pass control to the next middleware in the stack.

For example, when designing a system with confidential information, we want to be able to selectively show that information to authorized users. In order to do that, we would create middleware that tests a user’s permissions. If the user has the permission necessary, we would continue through the middleware stack by calling next(). If it fails, we would want to let the user know that they’re not allowed to see the information they’re trying to access.

**Instructions**

Notice how our middleware correctly calls next. If your if block is entered (meaning the bean does not exist), the function returns to break from the middleware. We could also achieve the same result by putting all the code after the if block in an else.

next is called at the end of the middleware callback function. This placement ensures that if a bean does not exist, the proper error status is sent, but if it does exist, we attach it to the request object and proceed to the next matching route/middleware to complete the request-response cycle.