EXPRESS

What is Express?

Express is a powerful but flexible Javascript framework for creating web servers and APIs. It can be used for everything from simple static file servers to JSON APIs to full production servers.

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();
```

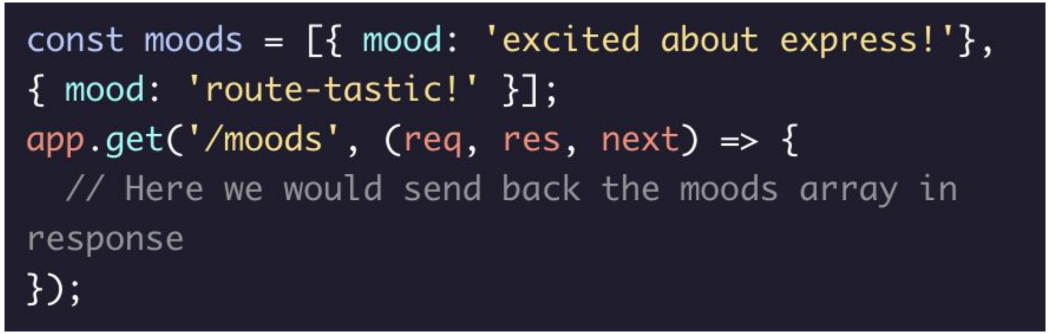
The purpose of a server is to listen for requests, perform whatever action is required to satisfy the request, and then return a response. In order for our server to start responding, we have to tell the server where to *listen* for new requests by providing a port number argument to a method called `app.listen()`. The server will then listen on the specified port 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}`);
});
```

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.

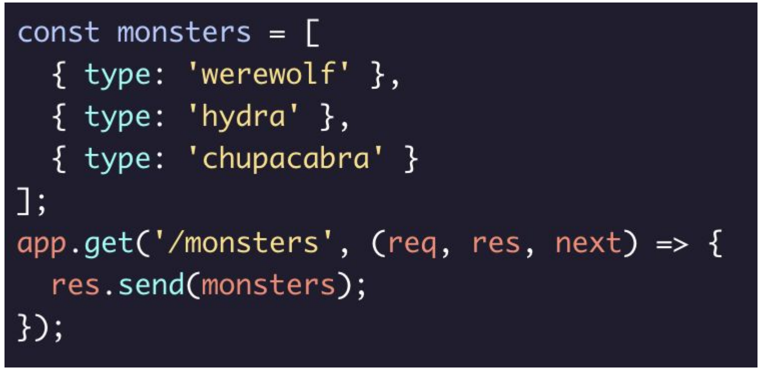


```
const moods = [{ mood: 'excited about express!'},
{ mood: 'route-tastic!' }];
app.get('/moods', (req, res, next) => {
  // Here we would send back the moods array in
response
});
```

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

Sending a response

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

Matching route paths

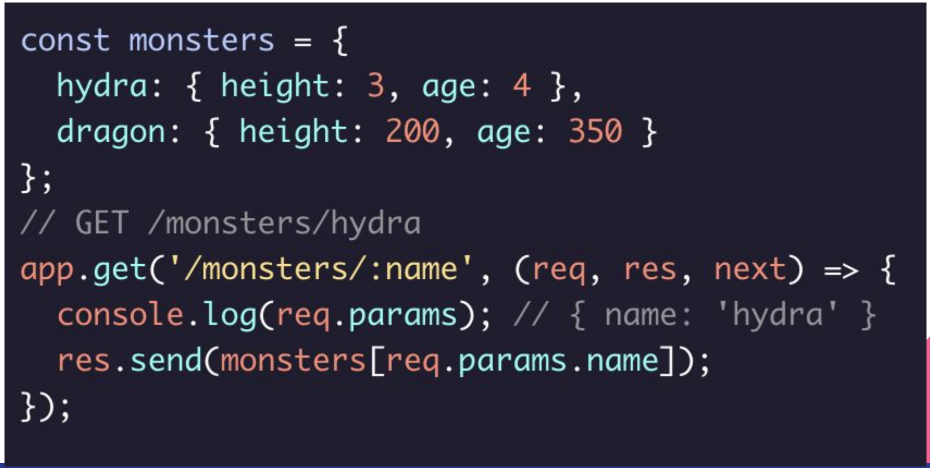
Express tries to match requests by route, meaning that if we send a request to `<server address>:<port number>/api-endpoint`, the Express server will search through any registered routes in order and try to match `/api-endpoint`.

Express searches through routes in the order that they are registered in your code. The first one that is matched will be used, and its callback will be called.

Getting a single element

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, 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.

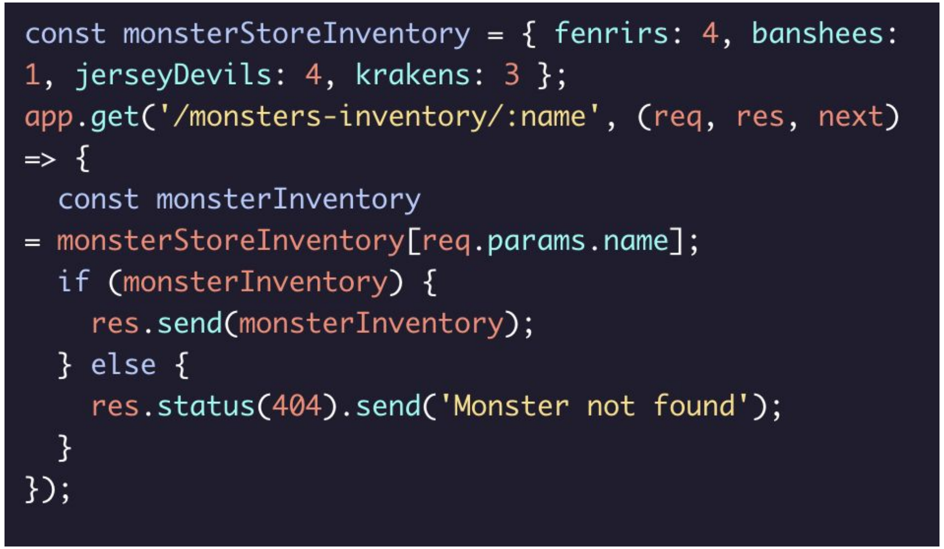


```
const monsters = {
  hydra: { height: 3, age: 4 },
  dragon: { height: 200, age: 350 }
};
// GET /monsters/hydra
app.get('/monsters/:name', (req, res, next) => {
  console.log(req.params); // { name: 'hydra' }
  res.send(monsters[req.params.name]);
});
```

Setting status codes

Express allows us to set the status code 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.

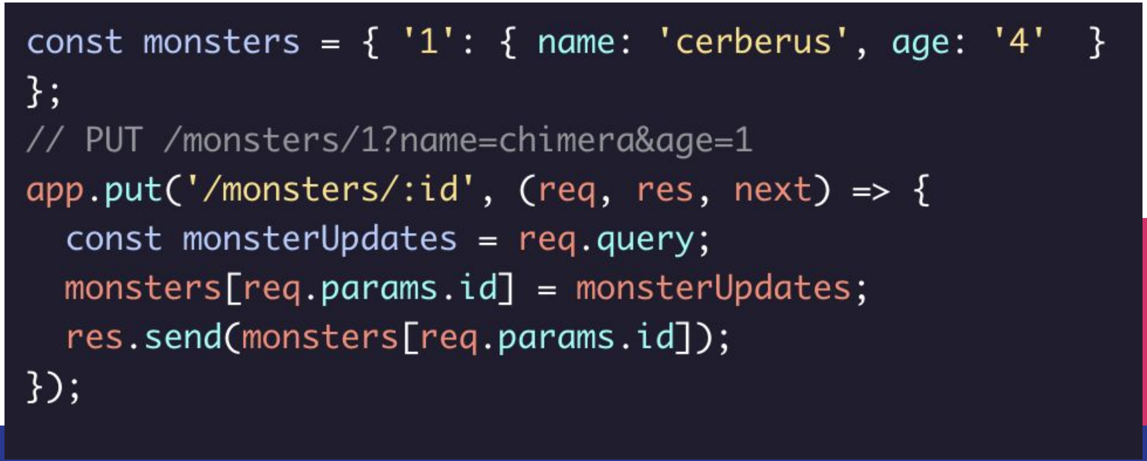


```
const monsterStoreInventory = { fenrirs: 4, banshees:
1, jerseyDevils: 4, krakens: 3 };
app.get('/monsters-inventory/:name', (req, res, next)
=> {
  const monsterInventory
= monsterStoreInventory[req.params.name];
  if (monsterInventory) {
    res.send(monsterInventory);
  } else {
    res.status(404).send('Monster not found');
  }
});
```

Query Strings

Query strings 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]);
});
```

Activity