



# Intro to Node.js



# Today's Agenda

- Introduction to Node.js!!



# Modules

1. Web page structure and appearance with HTML5 and CSS.
2. Client-side interactivity with JS DOM and events.
3. Using web services (API's) as a client with JS.
4. ***Writing JSON-based web services with a server-side language.***
5. Storing and retrieving information in a database with MySQL and server-side programs.



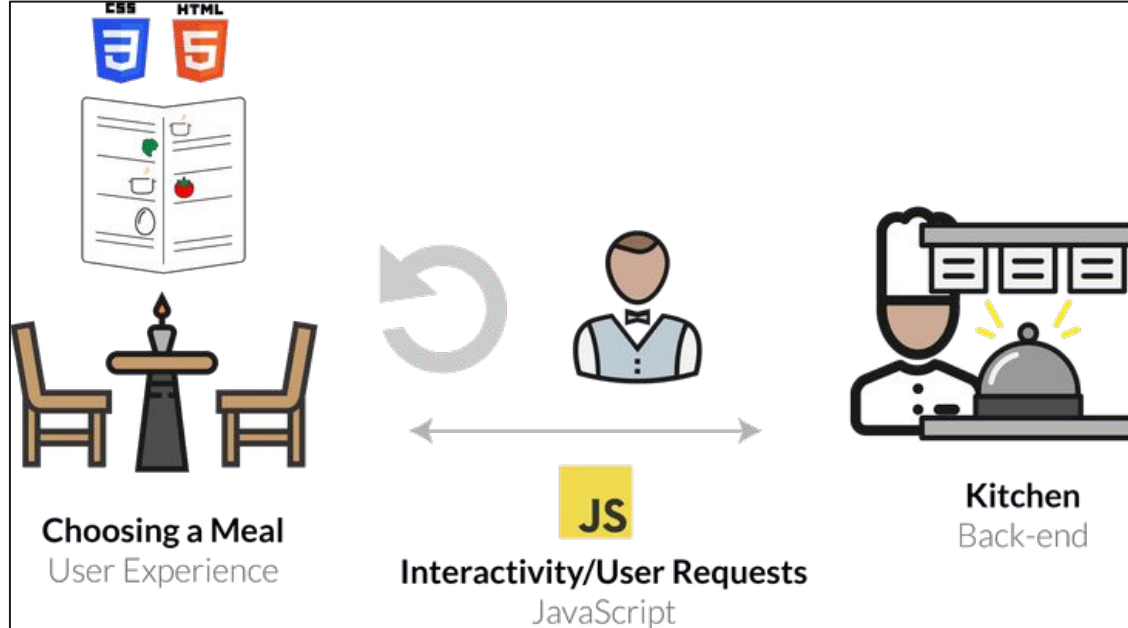
## Review: Web Service

Web service: software functionality that can be invoked through the internet using common protocols

It's like a remote function(s) you can call. Done by contacting a program on a web server

- Web services can be written in a variety of languages
- Many web services accept parameters and produce results
- Clients contact the server through the browser using XML over HTTP and/or AJAX  
Fetch code
- The service's output might be HTML but could be text, XML, JSON, or other content

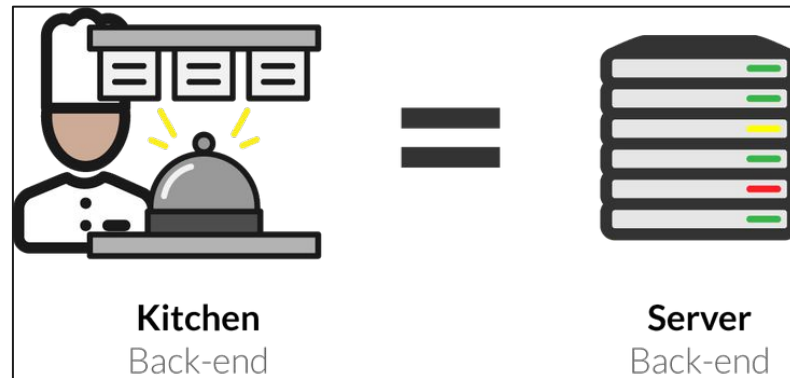
# So How Does a Web Service Respond to Requests



# Why Do We Need a Server to Handle Web Service Requests?

Servers are dedicated computers for processing data efficiently and delegating requests sent from many clients (often at once).

These tasks are not possible (or appropriate) in the client's browser.

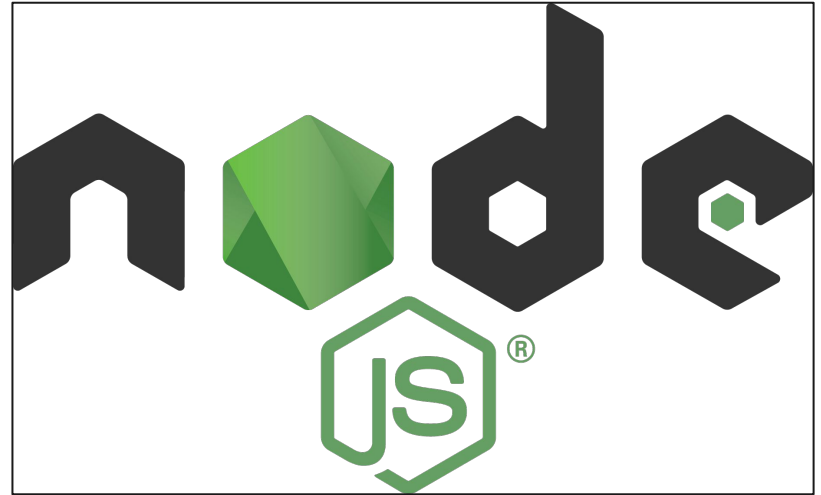




# Creating our own API

# Our (New) Server-Side Language: JS (Node)

- Open-source with an active developer community
- Flourishing package ecosystem
- Designed for efficient, asynchronous server-side programming
- 





# Languages for Server-Side Programming



Server-side programs are written using programming languages/frameworks such as [PHP](#), [Java/JSP](#), [Ruby on Rails](#), [ASP.NET](#), [Python](#), [Perl](#), [JS \(Node.js\)](#), and many, many more

Web servers contain software to run those programs and send back their output.



# What is Client-Side JS?

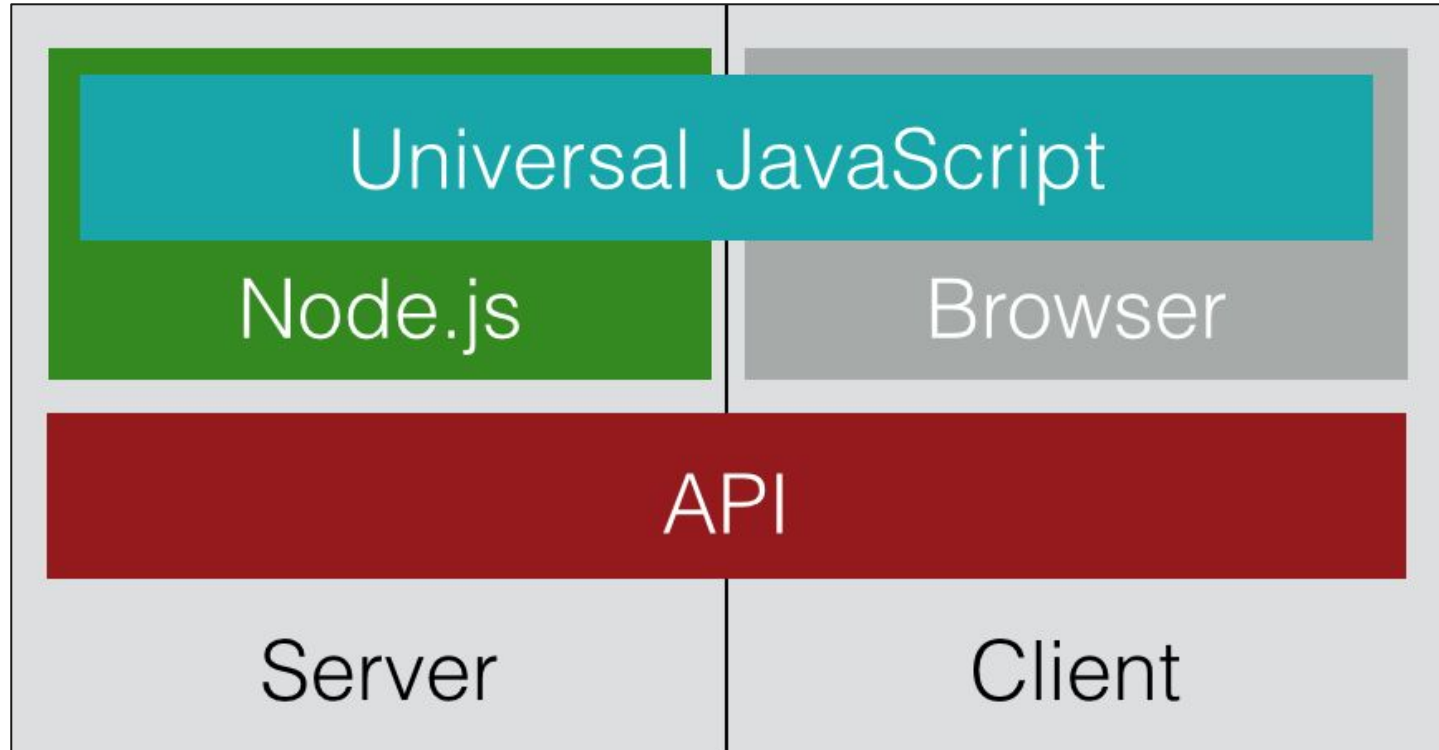
So far, we have used JS on the browser (client) to add interactivity to our web pages

"Under the hood", your browser requests the JS (and other files) from a URL resource, loads the text file of the JS, and interprets it realtime in order to define how the web page behaves.

In Chrome, it does this using the V8 JavaScript engine, which is an open-source JS interpreter made by Google. Other browsers have different JS engines (e.g. Firefox uses SpiderMonkey).

Besides the standard JS language features, you also have access to the DOM when running JS on the browser - this includes the `window` and `document`

# Client vs. Server-side JS





# Node.js: Server-side JS

Node.js uses the same open-source V8 JavaScript engine as Chrome  
Node.js is a runtime environment for running JS programs using the same core language features, but outside of the browser.

**When using Node, you do not have access to the browser objects/functions (e.g. document, window, addEventListener).**

Instead, you have access to functionality for managing HTTP requests, file i/o, and database interaction.

This functionality is key to building REST APIs!



# Getting started with Node.js

When you have Node installed (this week's section), you can run it immediately in the command line.

1. Start an interactive REPL with `node` (no arguments). This REPL is much like the Chrome browser's JS console tab.
2. Execute a JS program in the current directory with `node file.js`

# Starting a Node.js Project



There are a few steps to starting a Node.js application, but luckily most projects will follow the same structure.

When using Node.js, you will mostly be using the command line

1. Start a new project directory (e.g. `node-practice`)
2. Inside the directory, run `npm init` to initialize a `package.json` configuration file (you can keep pressing Enter to use defaults)
3. Install any modules with `npm install <package-name>`
4. Write your Node.js file! (e.g. `app.js`)
5. Include any front-end files in a `public` directory within the project.

Along the way, a tool called `npm` will help install and manage packages that are useful in your Node app.



# Starting a Node.js Project

Run `npm init` to create `package.json`



# Starting a Node.js Project

Run `npm install express` to install the Express.js package





# Starting app.js

Build your basic app:

```
"use strict";

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

app.get('/posts', function (req, res) {
  res.type("text").send("Hello World");
});

app.listen(8080);
```



# Node.js Modules

When you run a `.js` file using Node.js, you have access to default functions in JS (e.g. `console.log`)

In order to get functionality like file i/o or handling network requests, you need to import that functionality from modules - this is similar to the `import` keyword you have used in Java or Python.

In Node.js, you do this by using the `require()` function, passing the string name of the module you want to import.

For example, the module we'll use to respond to HTTP requests in Node.js is called `express`. You can import it like this:

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



## Quick reminder on `const` Keyword

Using `const` to declare a variable inside of JS just means that you can never change what that variable references. We've used this to represent "program constants" indicated by ALL\_UPPERCASE naming conventions

For example, the following code would not work:

```
const specialNumber = 1;  
specialNumber = 2; // TypeError: Assignment to constant variable.
```

When we store modules in Node programs, it is conventional to use `const` instead of `let` to avoid accidentally overwriting the module.

Unlike the program constants we define with `const` (e.g. `BASE_URL`), we use camelCase naming instead of ALL\_CAPS.



## app.listen()

To start the localhost server to run your Express app, you need to specify a port to listen to. The express app object has a function `app.listen` which takes a port number and optional callback function

At the bottom of your `app.js`, add the following code - (`process.env.PORT` is needed to use the default port when hosted on an actual server)

```
// Allows us to change the port easily by setting an environment
// variable, so your app works with our grading software
const PORT = process.env.PORT || 8000;
app.listen(PORT);
```



# Basic Routing in Express

Routes are used to define endpoints in your web service

Express supports different HTTP requests - we will learn GET and POST

Express will try to match routes in the order they are defined in your code

# Adding Routes in Express.js



```
app.get(path, (req, res) => {  
  ...  
});
```

- `app.get` allows us to create a GET endpoint. It takes two arguments: The endpoint URL path, and a callback function for modifying/sending the response.
- `req` is the request object, and holds items like the request parameters.
- `res` is the response object, and has methods to send data to the client.
- `res.set(...)` sets header data, like "content-type". Always set either "text/plain" or "application/json" with your response.
- `res.send(response)` returns the response as HTML text to the client.
- `res.json(response)` Does the same, but with a JSON object.

When adding a route to the path, you will retrieve information from the request, and send back a response using `res` (e.g. setting the status code, content-type, etc.)

If the visited endpoint has no matching route in your Express app, the response will be a 404 (resource not found)



## Useful Request Properties/Methods

Name	Description
<a href="#"><u>req.params</u></a>	Endpoint “path” parameters from the request
<a href="#"><u>req.query</u></a>	Query parameters from the request

# Useful Response Properties/Methods



Name	Description
<a href="#"><u>res.write(data)</u></a>	Writes data in the response without ending the communication
<a href="#"><u>res.end()</u></a>	Ends the process
<a href="#"><u>res.send()</u></a>	Sends information back (default text with HTML content type)
<a href="#"><u>res.json()</u></a>	Sends information back as JSON content type
<a href="#"><u>res.set()</u></a>	Sets header information, such as "Content-type"
<a href="#"><u>res.type()</u></a>	A convenience function to set content type (use "text" for "text/plain", use "json" for "application/json")
<a href="#"><u>res.status()</u></a>	Sets the response status code
<a href="#"><u>res.sendStatus()</u></a>	Sets the response status code with the default status text



# Setting the Content Type

By default, the content type of a response is HTML - we will only be sending plain text or JSON responses though in our web services

To change the content type, you can use the `res.set` function, which is used to set response header information (e.g. content type).

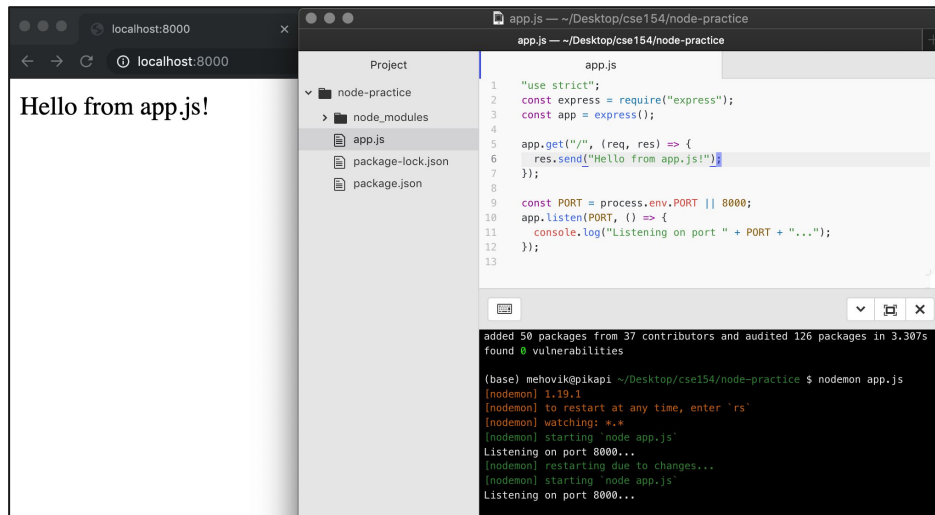
You can alternatively use `res.type("text")` and `res.type("json")` which are equivalent to setting `text/plain` and `application/json` Content-Type headers, respectively.

```
app.get('/hello', function (req, res) {  
  // res.set("Content-Type", "text/plain");  
  res.type("text"); // same as above  
  res.send('Hello World!');  
});
```

```
app.get('/hello', function (req, res) {  
  // res.set("Content-Type", "application/json");  
  res.json({ "msg" : "Hello world!" });  
  // can also do res.json({ "msg" : "Hello world!"});  
  // which also sets the content type to application/json  
});
```

# Adding our first “root” route

When you add a route in your Express app (here, we use `/` for the most basic "root" route), you can use `res.send` to send a response to the client (browser) and view using the port (here, 8000 on localhost)



The screenshot displays a web browser window on the left and a code editor on the right. The browser window shows the text "Hello from app.js!" on a white background. The code editor shows the file `app.js` with the following code:

```
1  "use strict";
2  const express = require("express");
3  const app = express();
4
5  app.get("/", (req, res) => {
6    res.send("Hello from app.js!");
7  });
8
9  const PORT = process.env.PORT || 8000;
10 app.listen(PORT, () => {
11   console.log("Listening on port " + PORT + "...");
12 });
13
```

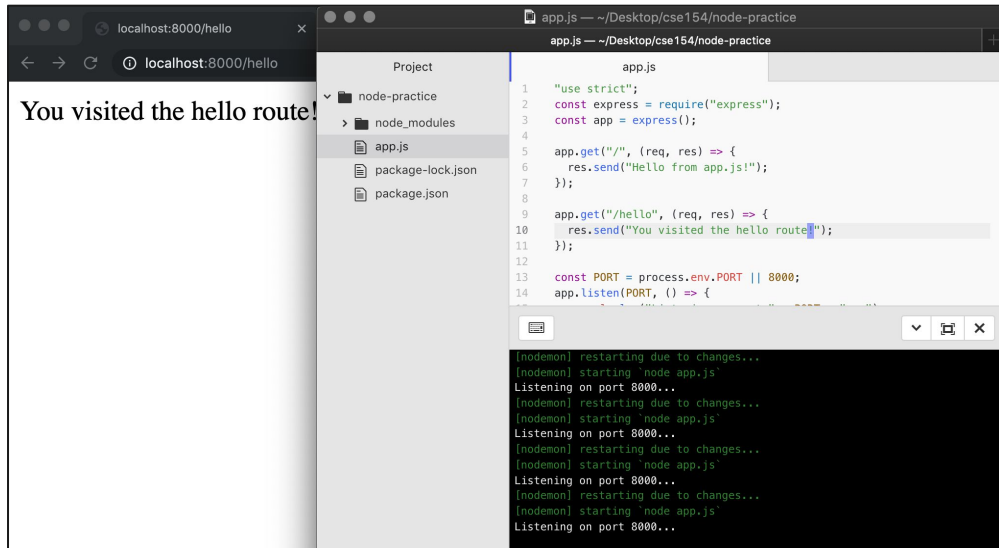
Below the code editor, a terminal window shows the output of running the application:

```
added 50 packages from 37 contributors and audited 126 packages in 3.307s
found 0 vulnerabilities

(base) mehovich@pikapi ~/Desktop/cse154/node-practice $ nodemon app.js
[nodemon] 1.19.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
Listening on port 8000...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
Listening on port 8000...
```

# Adding another route

You can add more routes in your file, and Express will use the first one matching the request path



The screenshot shows a web browser on the left and a code editor on the right. The browser's address bar shows 'localhost:8000/hello' and the page content says 'You visited the hello route!'. The code editor shows a file named 'app.js' with the following code:

```
1 "use strict";
2 const express = require("express");
3 const app = express();
4
5 app.get("/", (req, res) => {
6   res.send("Hello from app.js!");
7 });
8
9 app.get("/hello", (req, res) => {
10  res.send("You visited the hello route!");
11 });
12
13 const PORT = process.env.PORT || 8000;
14 app.listen(PORT, () => {
```

The terminal at the bottom of the code editor shows the following output:

```
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Listening on port 8000...
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Listening on port 8000...
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Listening on port 8000...
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Listening on port 8000...
```



# Request Parameters: Path Parameters

Act as wildcards in routes, letting a user pass in "variables" to an endpoint

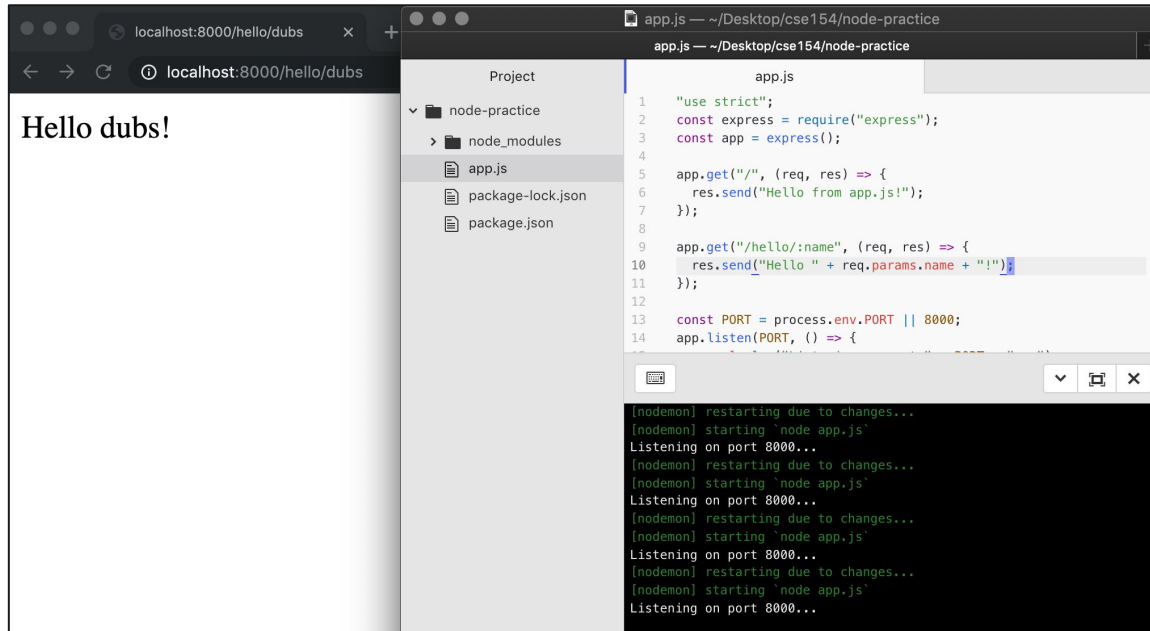
Define a route parameter with `:param`

```
Route path: /states/:state/cities/:city  
Request URL: http://localhost:8000/states/wa/cities/Seattle  
req.params: { "state": "wa", "city": "Seattle" }
```

These are attached to the request object and can be accessed with `req.params`

```
app.get("/states/:state/cities/:city", function (req, res) {  
  res.type("text");  
  res.send("You sent a request for " + req.params.city + ", " +  
    req.params.state);  
});
```

# Adding route parameters: Example



# Request Parameters: Path Parameters



You can also use query parameters in Express using the `req.query` object, though they are more useful for optional parameters.

```
Route path: /cityInfo  
Request URL: http://localhost:8000/cityInfo?state=wa&city=Seattle  
req.query: { "state": "wa", "city": "Seattle" }
```

```
app.get("/cityInfo", function (req, res) {  
  let state = req.query.state; // wa  
  let city = req.query.city;   // Seattle  
  // do something with variables in the response  
});
```

Unlike path parameters, these are not included in the path string (which are matched using Express routes) and we can't be certain that the accessed query key exists.

If the route requires the parameter but is missing, you should send an error to the client in the response.

# Setting Errors



The Response object has a `status` function which takes a status code as an argument. The 400 status code is what we'll use to send back an error indicating to the client that they made an invalid request.

A helpful message should always be sent with the error.

```
app.get("/cityInfo", function (req, res) {  
  let state = req.query.state;  
  let city = req.query.city;  
  if (!(state && city)) {  
    res.status(400).send("Error: Missing required city and state query  
parameters.");  
  } else {  
    res.send("You sent a request for " + city + ", " + state);  
  }  
});
```



# Summary of Building an Express App

1. Create a file (e.g. `app.js`)
2. Add required modules at the top (at minimum, `require('express')`)
3. Create an app instance: `const app = express();`
4. At the end of the file, listen to a port (e.g. 8000)
5. Add routes! `'/'` stands for the basic root route, which can be visited in your browser at `localhost:8000/` when your app is running.