CS193X: Web Programming Fundamentals

Spring 2017

Victoria Kirst
(vrk@stanford.edu)

Schedule

Today:

- npm
- Express
- `fetch()` to localhost
- package.json
- **HW4 due tonight, late cutoff Friday**
- **HW5 released sometime tomorrow**

Friday:

- Saving and retrieving data

Node installation; lecture code

NOTE: The following slides assume you have already installed NodeJS.

NodeJS installation instructions:

- http://web.stanford.edu/class/cs193x/install-node/

All **lecture code** is split between these gits repositories:

- https://github.com/yayinternet/lecture19
- https://github.com/yayinternet/lecture20

NodeJS

NodeJS

NodeJS:

- A JavaScript runtime written in C++.
- Can interpret and execute JavaScript.
- Includes support for the NodeJS API.

NodeJS API:

- A set of JavaScript libraries that are useful for creating server programs.

V8 (from Chrome):

- The JavaScript interpreter ("engine") that NodeJS uses to interpret, compile, and execute JavaScript code

node command

Running node without a filename runs a REPL loop

- Similar to the JavaScript console in Chrome, or when you run "python"

```
$ node
> let x = 5;
undefined
> x++
5
> x
6
```

node command

The node command can be used to execute a JS file (GitHub):

$ node ***fileName***

```
$ node simple-script.js
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
```

Node for servers

server.js (GitHub):

```
const http = require('http');

const server = http.createServer();

server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.on('listening', function() {
  console.log('Server running!');
});

server.listen(3000);
```

Node for servers

Include the HTTP NodeJS library

```
const http = require('http');

const server = http.createServer();
```

When the server gets a request, send back "Hello World" in plain text

```
server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});
```

When the server is started, print a log message

```
server.on('listening', function() {
  console.log('Server running!');
});
```

Start listening for messages!

```
server.listen(3000);
```

Node for servers

The NodeJS server APIs are actually pretty low-level:

- You build the request manually
- You write the response manually
- There's a lot of tedious processing code

```
var http = require('http');

http.createServer(function(request, response) {
  var headers = request.headers;
  var method = request.method;
  var url = request.url;
  var body = [];
  request.on('error', function(err) {
    console.error(err);
  }).on('data', function(chunk) {
    body.push(chunk);
  }).on('end', function() {
    body = Buffer.concat(body).toString();
    // BEGINNING OF NEW STUFF

    response.on('error', function(err) {
      console.error(err);
    });

    response.statusCode = 200;
    response.setHeader('Content-Type', 'application/json');
    // Note: the 2 lines above could be replaced with this next one:
    // response.writeHead(200, {'Content-Type': 'application/json'})

    var responseBody = {
      headers: headers,
      method: method,
      url: url,
      body: body
    };
```

ExpressJS

Node for servers

We're going to use a library called ExpressJS on top of NodeJS. Here's our server code **without** Express (GitHub):

```
const http = require('http');

const server = http.createServer();

server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.on('listening', function() {
  console.log('Server running!');
});

server.listen(3000);
```

ExpressJS

And **with** Express (GitHub):

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

app.get('/', function (req, res) {
  res.send('Hello World!');
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

ExpressJS

Express is not part of the NodeJS APIs.

If we try to use it, we'll get an error:

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

```
module.js:327
    throw err;
    ^

Error: Cannot find module 'express'
    at Function.Module._resolveFilename
```

We need to install Express via npm.

npm

When you install NodeJS, you also install `npm`:

- **npm**: Node Package Manager*:
  Command-line tool that lets you install **packages** (libraries and tools) written in JavaScript and compatible with NodeJS
- Can find packages through the online repository:
  https://www.npmjs.com/

*though the creators of "npm" say it's not an acronym (as a joke -_-)

npm install and uninstall

`npm install` ***package-name***

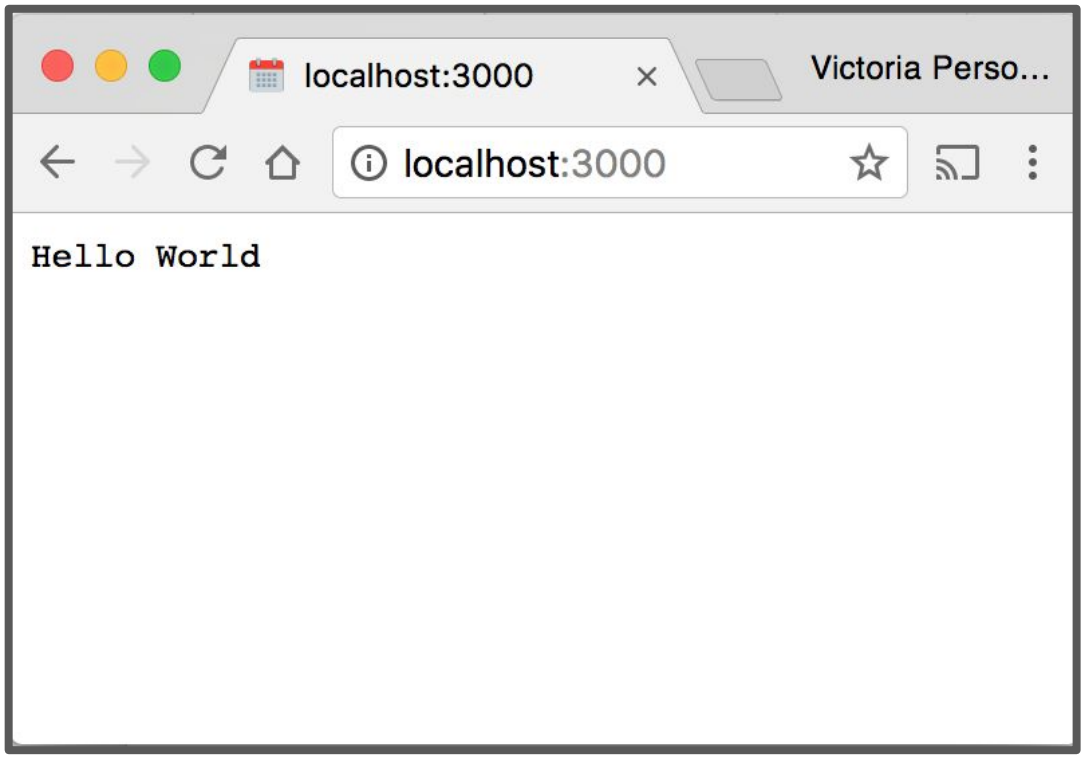
- This downloads the *package-name* library into a `node_modules` folder.
- Now the *package-name* library can be included in your NodeJS JavaScript files.

`npm uninstall` ***package-name***

- This removes the *package-name* library from the `node_modules` folder, deleting the folder if necessary

Express example

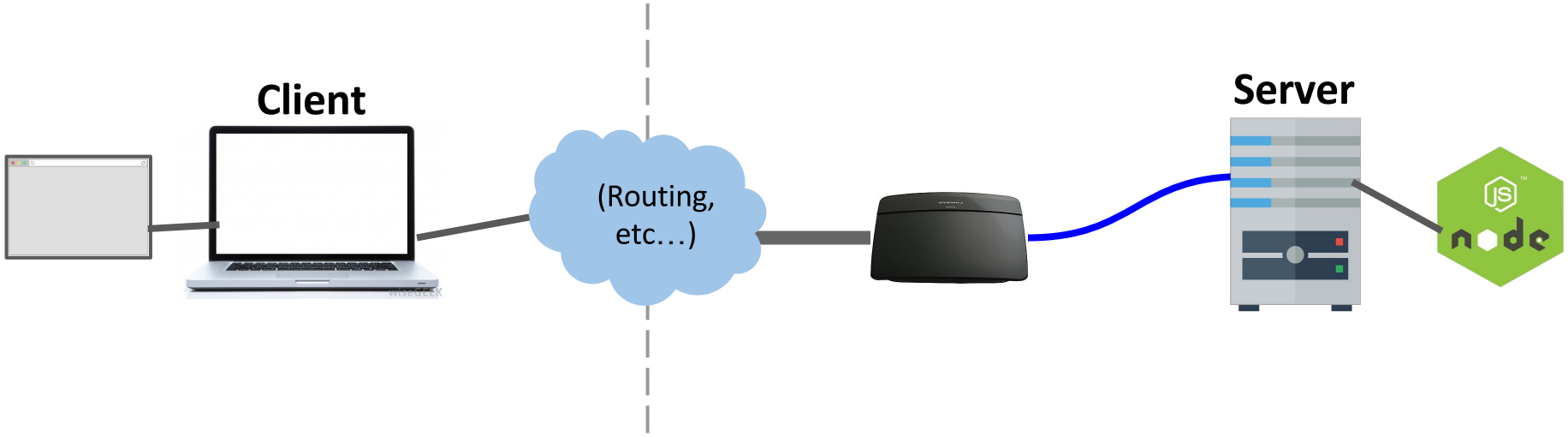
```
$ npm install express
$ node server.js
Example app listening on port 3000!
```



Understanding localhost

Local server?

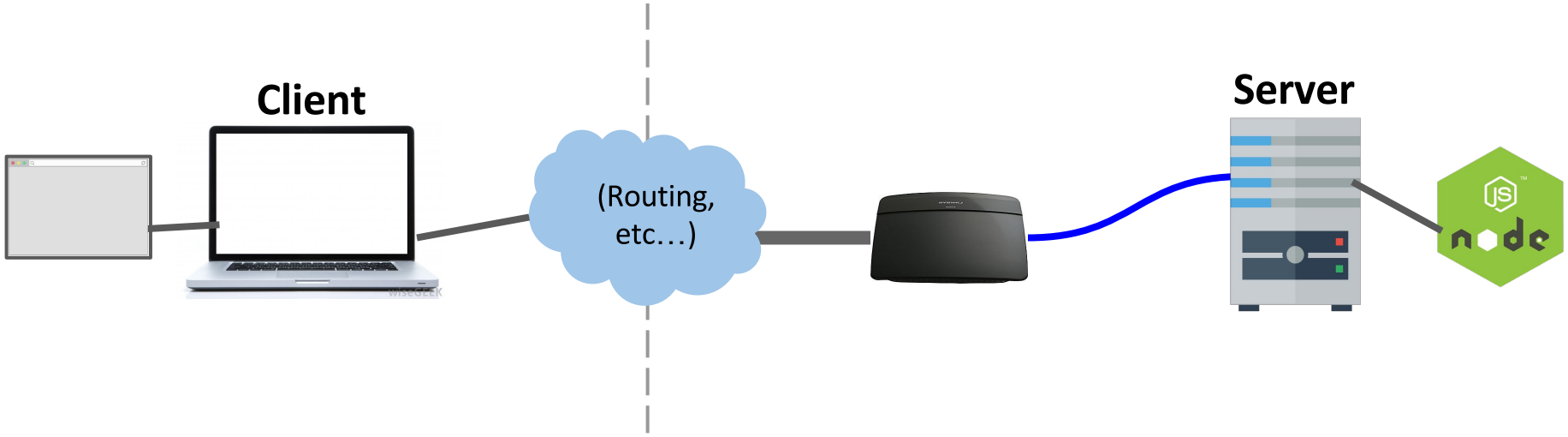
The client/server diagrams shown in previous lectures have always involve two separate machines:



But when we run our server locally, isn't there only one computer involved?

Local server?

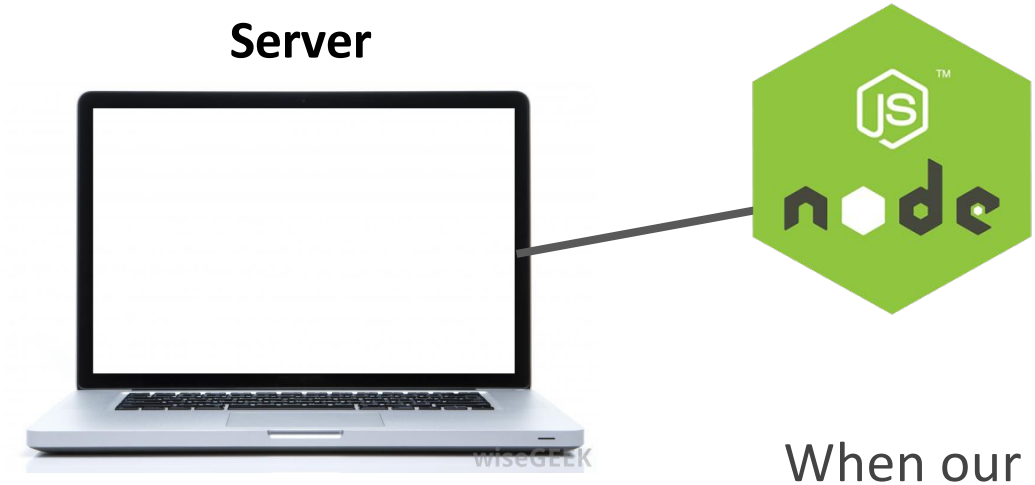
The client/server diagrams shown in previous lectures have always involve two separate machines:



But when we run our server locally, isn't there only one computer involved? **A: Yes, when we execute our Node server and access it via localhost, our laptop is both the client and the server machine.**

Running a server

When we run **`$ node server.js`** which runs `server.listen(3000)`, **our laptop becomes a server**:

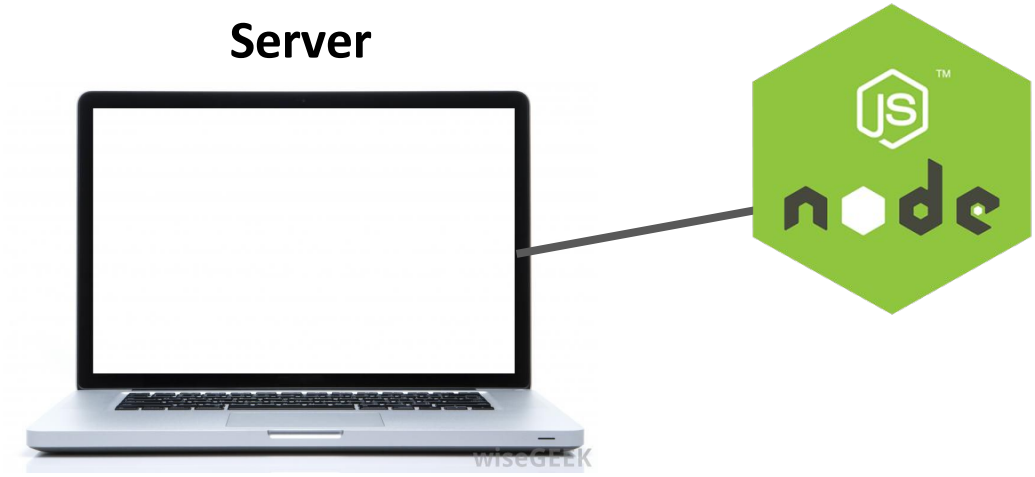



```
vrk:node-server $ node server.js
Server running!
```

When our laptop's operating system receives HTTP messages sent to port 3000, it will send those messages to our Node server.

Running a server

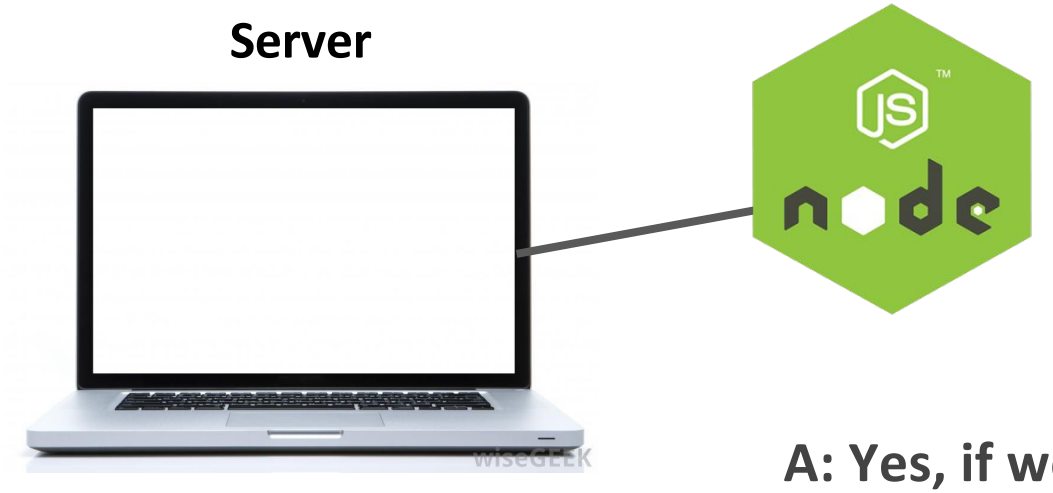
Q: If our laptop became a server as soon as we ran "node `server.js`", can other computers connect to our server?




```
vrk:node-server $ node server.js
Server running!
```

Running a server

Q: If our laptop became a server as soon as we ran "node `server.js`", can other computers connect to our server?




```
vrk:node-server $ node server.js
Server running!
```

A: Yes, if we configure our machine correctly. Instead of doing this ourselves, we'll use a tool called localtunnel to help us.

localtunnel

Localtunnel is a **command-line tool** that is also distributed via npm.

`npm install -g` ***package-name***

- We can install packages **globally** using the `-g` switch
- **Only used for command-line tools**

To install localtunnel, we run:

```
$ npm install -g localtunnel
```

We can now run the `lt` command via command-line.

Installing tools with npm

`$ npm install` ***package-name***

- Downloads the ***package-name*** library and puts the source code for the library in a `node_modules` directory.
- **Used for libraries** or command-line tools that will only work in the directory you've installed it.

`$ sudo npm install -g` ***package-name***

- Installs the command-line tools that are provided by ***package-name***.
- You will probably need superuser (sudo) privileges.
- **Only used for command-line tools**

localtunnel

If you start your server in one terminal window:

```
$ node server.js
```

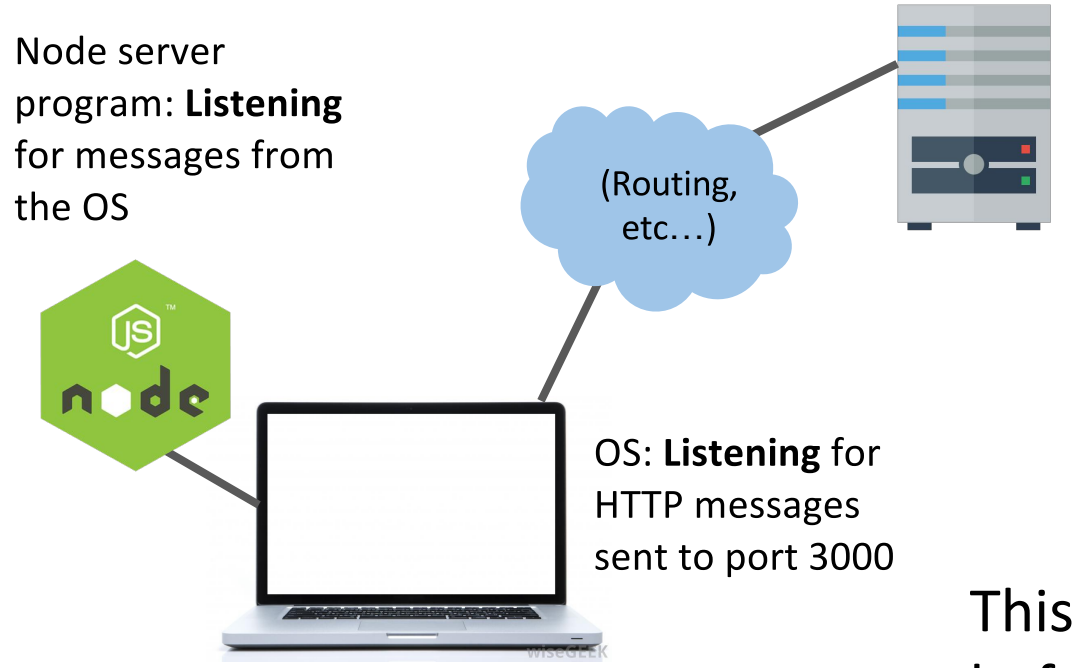
And you run localtunnel in a different window, using the port number your server is bound to (3000 in our case):

```
$ lt --port 3000
```

Localtunnel will reply with a URL that anyone can use to access your locally running server.

Note: This should only be done for development/demos and should *not* be how you deploy services!

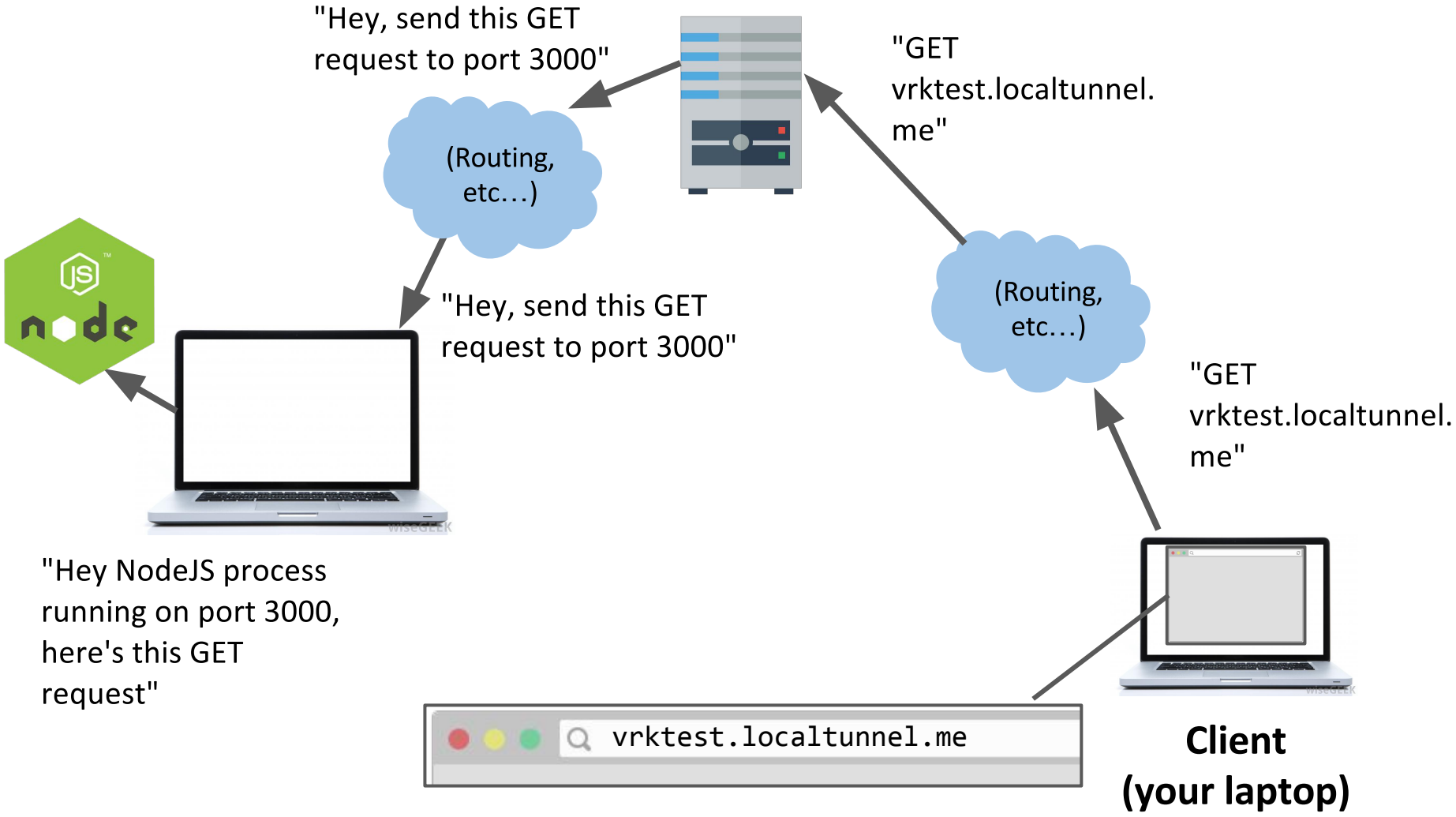
localtunnel setup



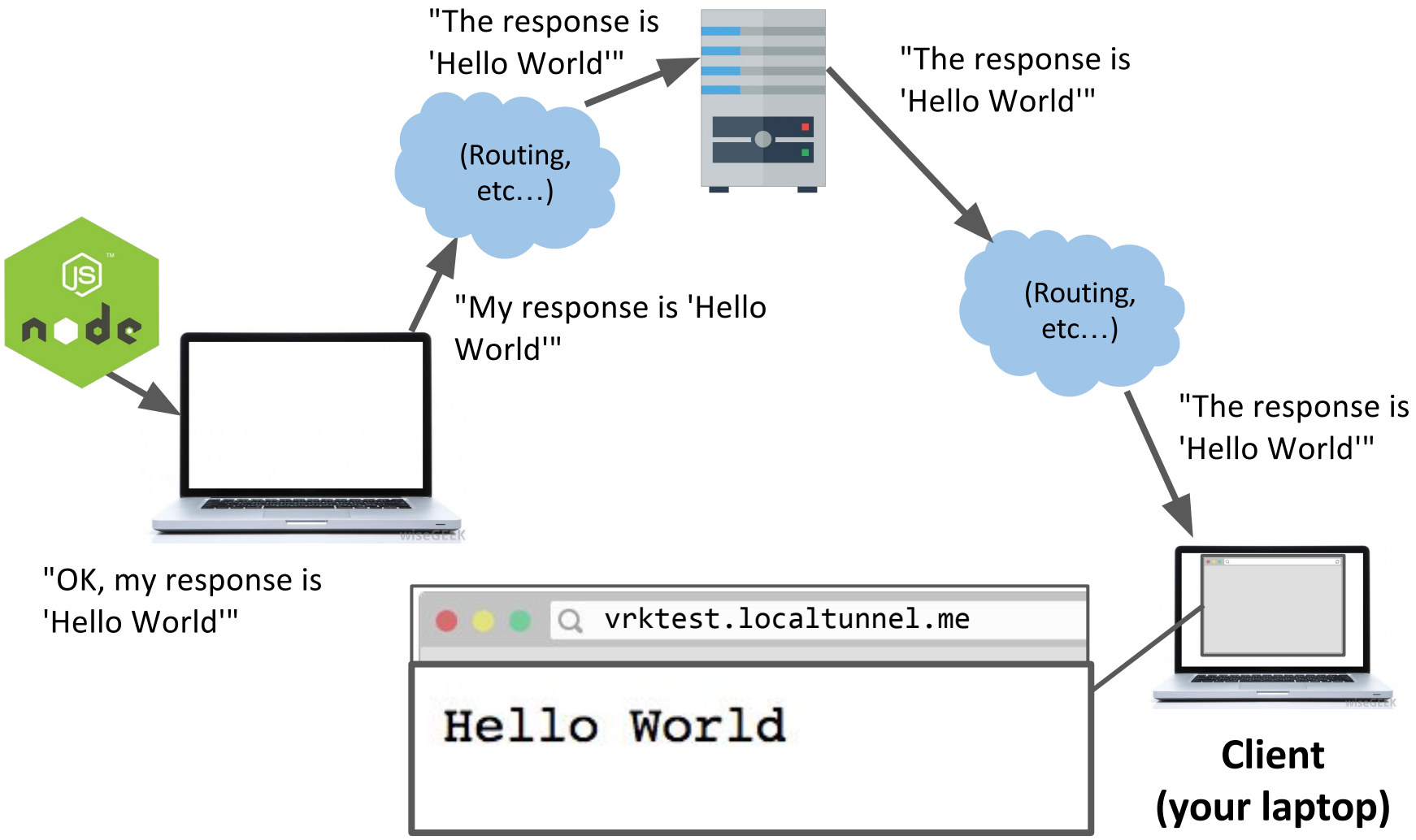

```
vrk:node-server $ node server.js
Server running!
```

This is the state of the world before anyone connects to vrktest.localtunnel.me...

localtunnel request



localtunnel response



localtunnel demo

Notice:

- If I kill my "node server.js" process, the URL no longer works (will timeout)

- If a bunch of people access the URL at the same time, my server handles each response one at a time

- If I am doing other stuff on my computer (surfing the web; using PhotoShop) it'll hinder the performance of my server

"Real" servers

Therefore, most "real" servers are setup like this:

- Instead of running on a random laptop, servers run on **dedicated machines** that only runs the server software
- Server computers are installed with a **different OS that** is optimized for running server software
- There are **backup server computers** running in case one dies (software crashes; power goes out; hardware fails; etc)
- There are **multiple computers** with identical server programs running if you have a lot of traffic
  - Since each computer can only receive requests one at a time, to receive more requests simultaneously, you need multiple machines

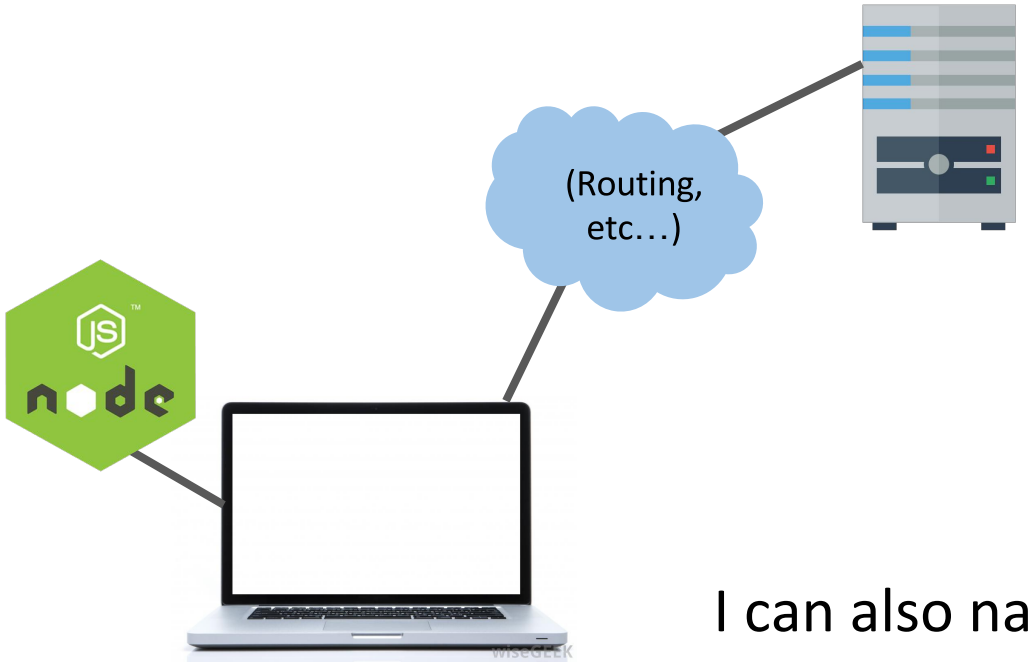
"Real" servers

Veeeeery general rule of thumb: A single-machine server should be able to handle ~1000 to 10,000 simultaneous requests

- It will **receive** each request one at a time
- It may **process** the requests in parallel

(In other words: Unless you are a) expecting >1000 simultaneous requests to your web server, and b) picky about exactly how those requests are served, you really don't need to be deploying your server via AWS, Google Cloud Platform, or other IaaS. It's not that AWS/GCP isn't worth learning; it's just not the first thing you need to learn.)

localtunnel on local machine

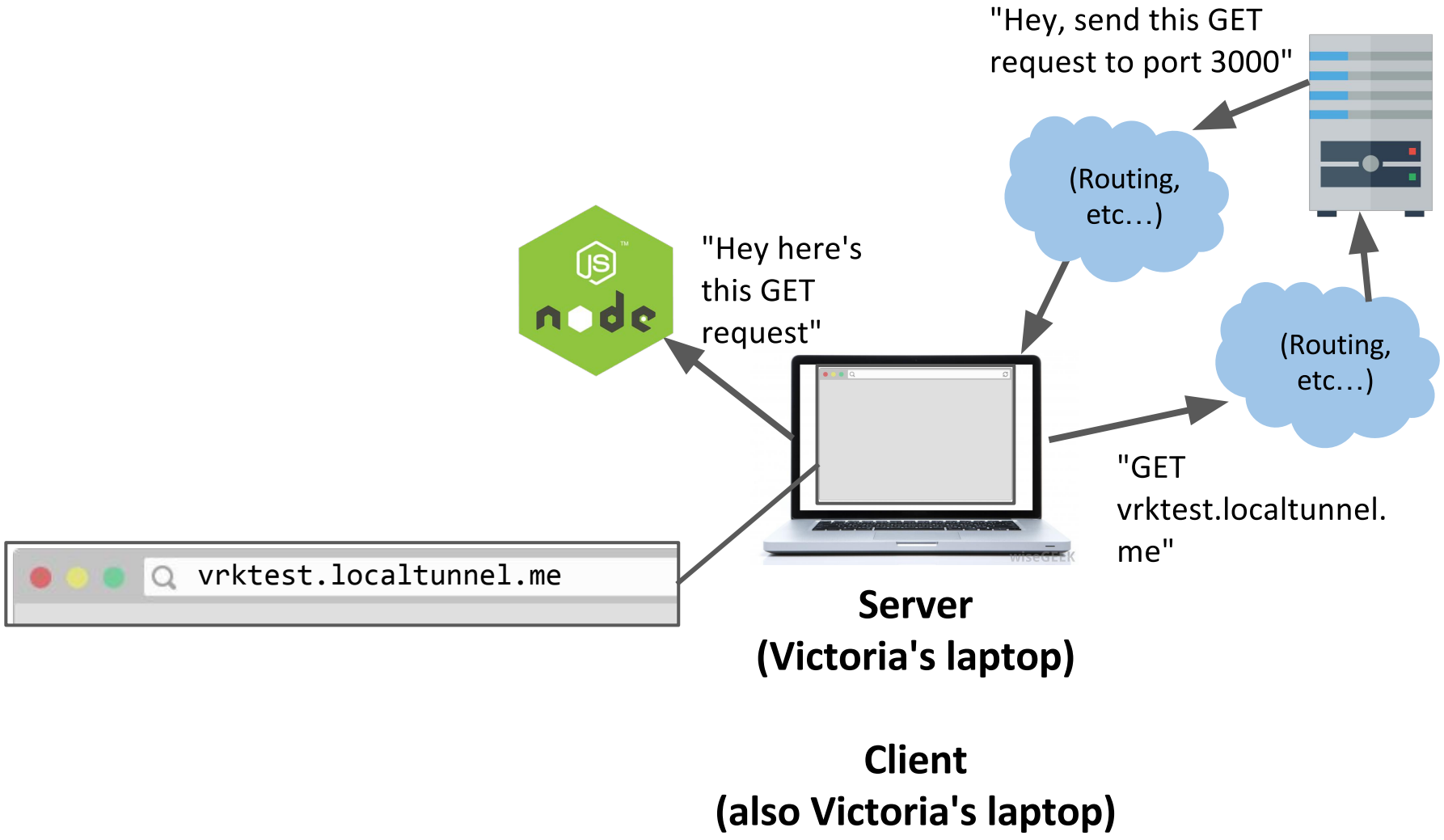


Server
(Victoria's laptop)

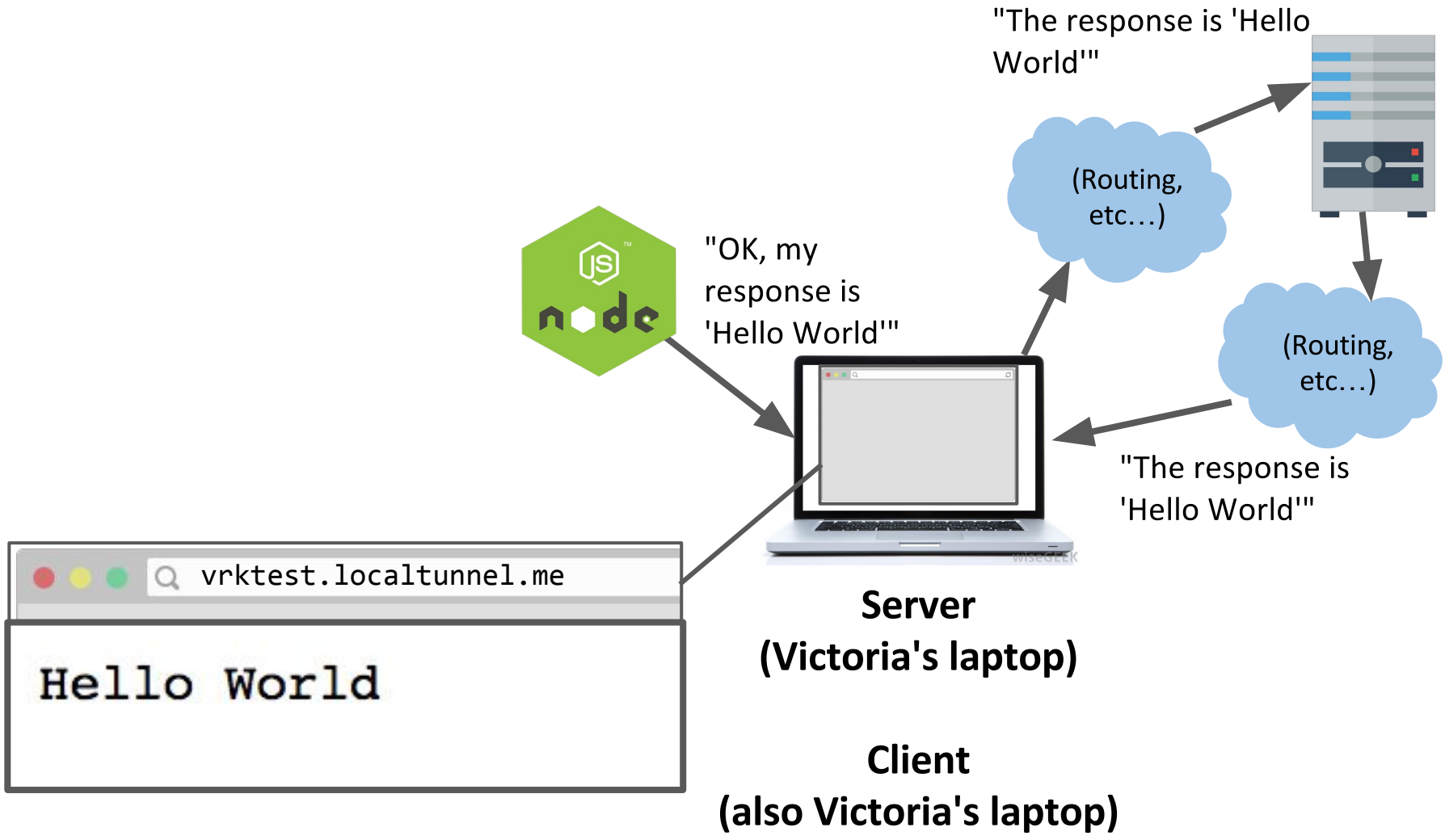
I can also navigate to:
`vrktest.localtunnel.me`
from my own laptop.

- In this scenario, my laptop is both the server and the client.

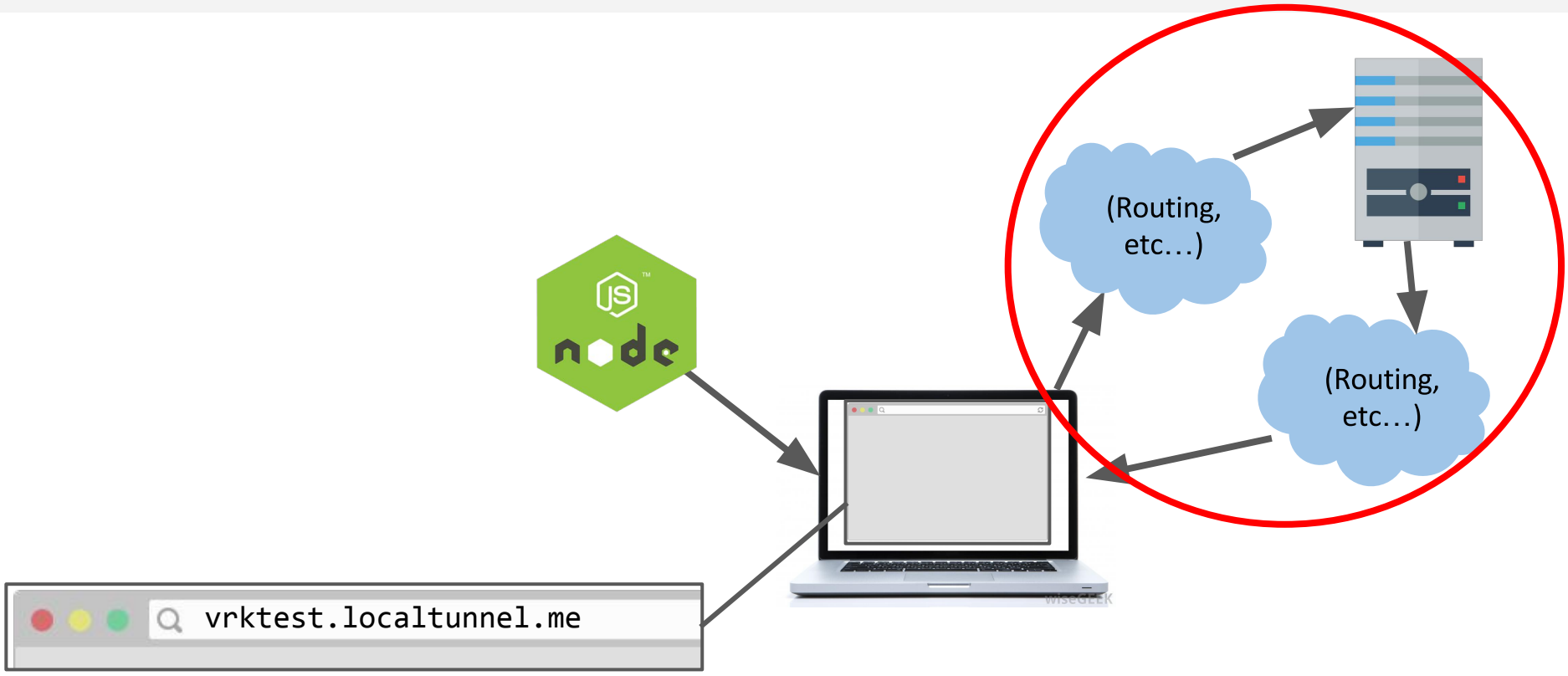
localtunnel request



localtunnel response

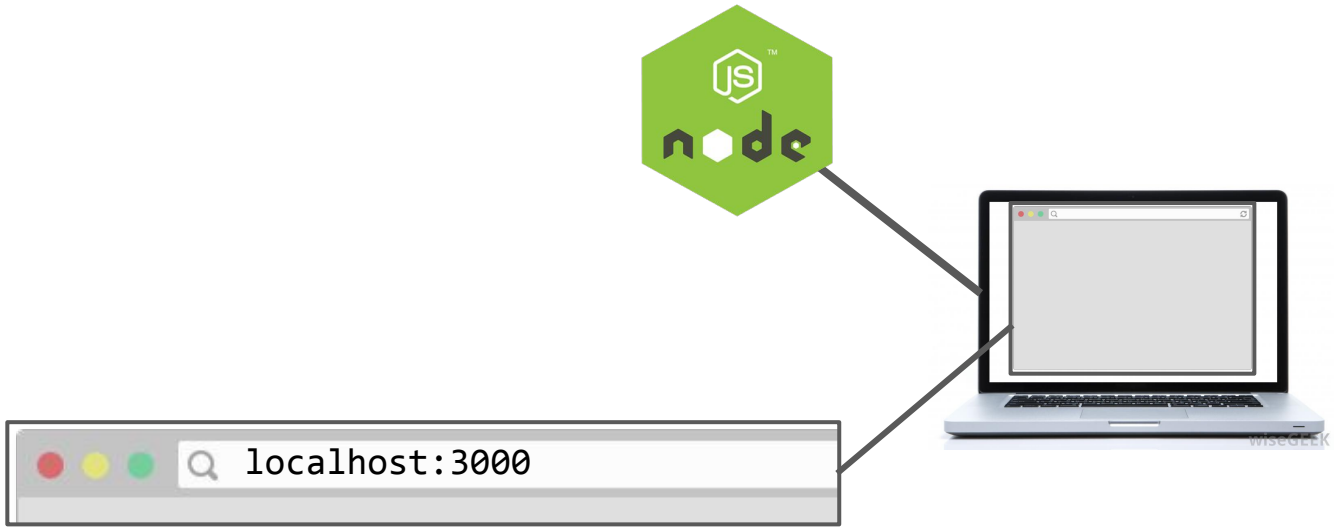


localtunnel is not needed



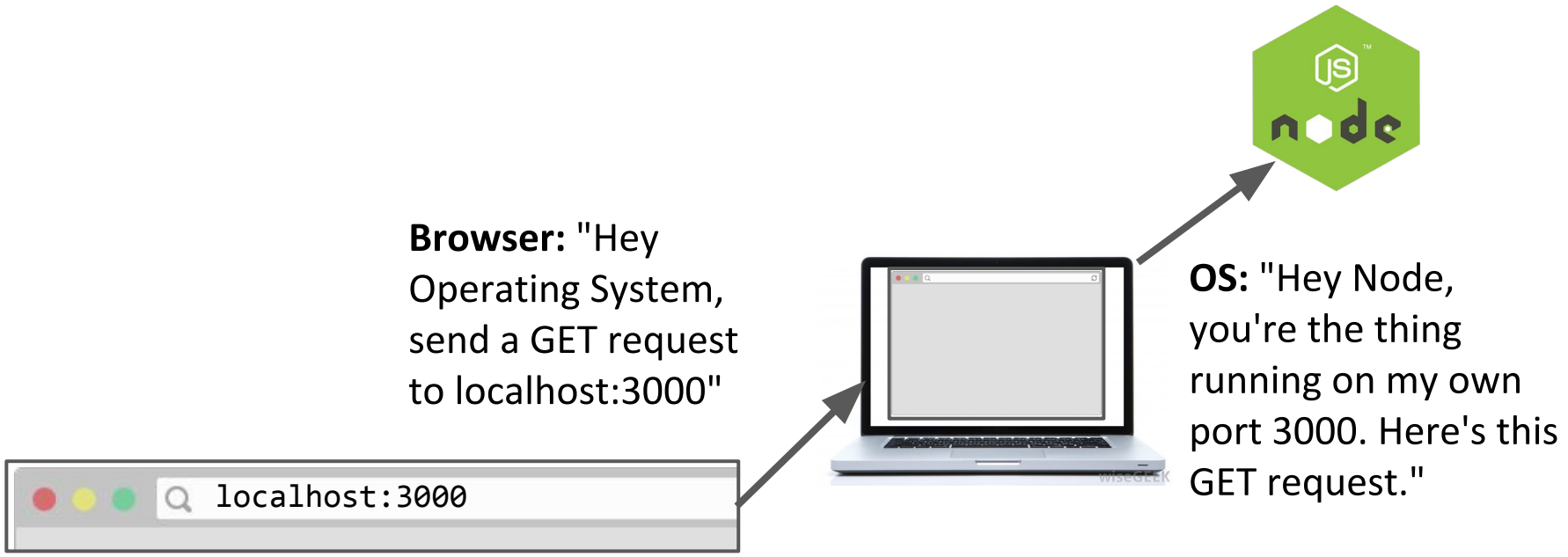
But since the client and the server are on the same machine… this routing/proxying step is unnecessary.

localhost

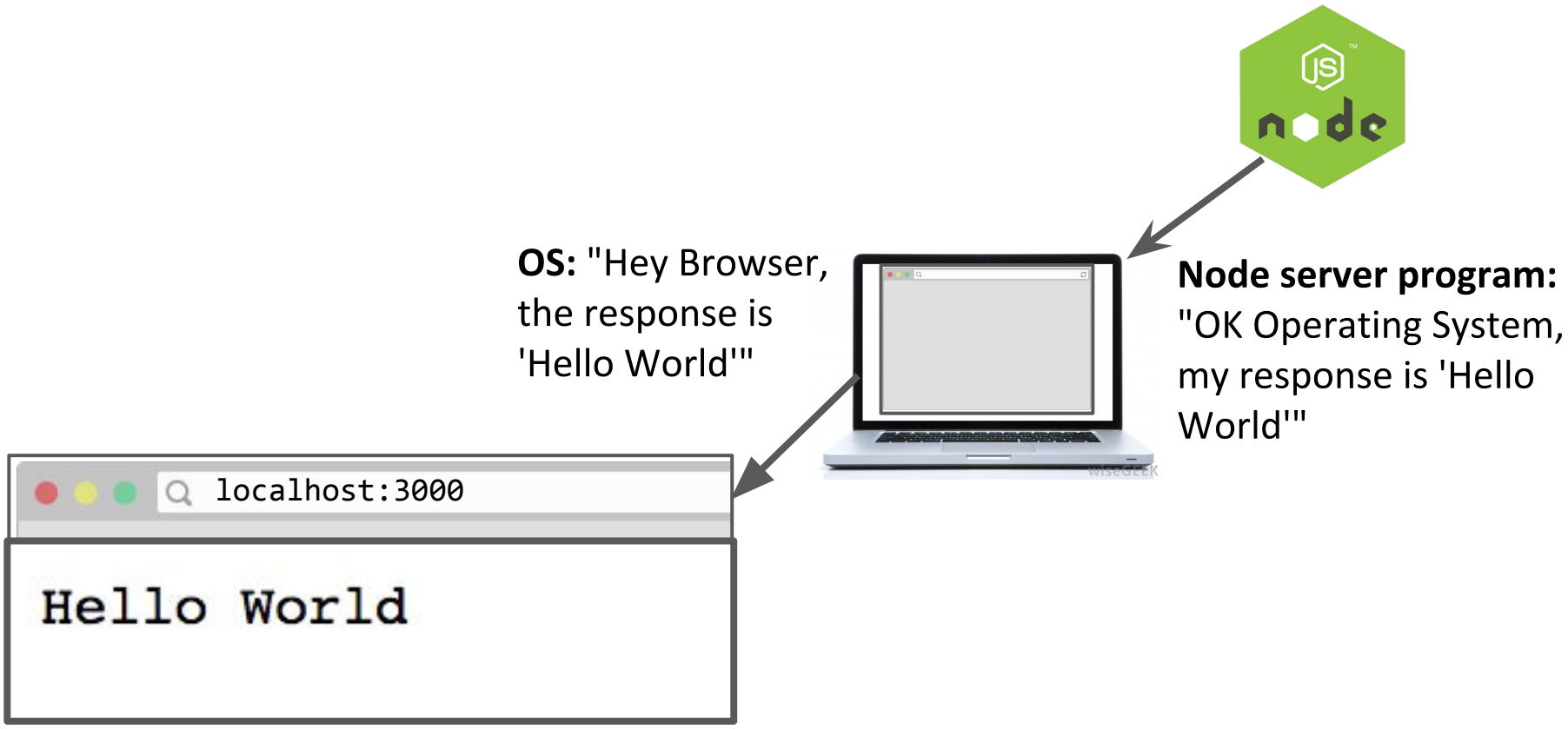


Instead you can query the process running on port 3000 using http://localhost:3000

Localhost request



Localhost response



Back to Express

ExpressJS

Here's our server written using NodeJS and Express (GitHub):

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

app.get('/', function (req, res) {
  res.send('Hello World!');
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

Let's examine what's going on more carefully...

ExpressJS

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

The `require()` lets us load the ExpressJS module.

The module actually contains a function that creates a new Express Application object.

ExpressJS

```
app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

The ExpressJS `listen()` is identical to the NodeJS `listen()` function:

- This binds the server process to the given **port number**.
- Now messages sent to the OS's port 3000 will be routed to this server process.
- The function parameter is a callback that will execute when it starts listening for HTTP messages (when the process has been bound to port 3000)

ExpressJS

```
app.get('/', function (req, res) {
  res.send('Hello World!');
})
```

app.***method***(***path***, ***handler***)

- Specifies how the server should handle HTTP ***method*** requests made to URL/***path***
- The function callback will fire every time there's a new response.
- This example is saying: When there's a GET request to http://localhost:3000/, respond with the text "Hello World!"

More routes

Here are some other routes in Express:

```
app.get('/', function (req, res) {
  res.send('Main page!');
});

app.get('/hello', function (req, res) {
  res.send('GET hello!');
});

app.post('/hello', function (req, res) {
  res.send('POST hello!');
});
```

Handler parameters

```
app.get('/', function (req, res) {
  res.send('Hello World!');
})
```

Express has its own `Request` and `Response` objects:

- `req` is a Request object
- `res` is a Response object
- `res.send()` sends an HTTP response with the given content
  - Sends content type "text/html" by default

Hello world server

Here's how we put it all together again:

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

app.get('/', function (req, res) {
  res.send('Hello World!');
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

Querying our server

HTTP requests

Our server is written to respond to HTTP requests (GitHub):

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

app.get('/', function (req, res) {
  res.send('Hello World!');
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

Q: How do we sent HTTP requests to our server?

Querying our server

Here are three ways to send HTTP requests to our server:

1. Navigate to http://localhost:3000/<path> in our browser
   a. **Caveat:** Can only do GET requests

2. Call `fetch()` in web page
   a. We've done GET requests so far, but can send any type of HTTP request

3. `curl` command-line tool
   a. Debug tool we haven't seen yet

curl

`curl`: Command-line tool to send and receive data from a server (Manual)

`curl --request` ***METHOD url***

e.g.

```
$ curl --request POST http://localhost:3000/hello
```

Querying with `fetch()`

We can try querying our server the same way we've queried the Spotify or Giphy servers, i.e. via the `fetch()` command:

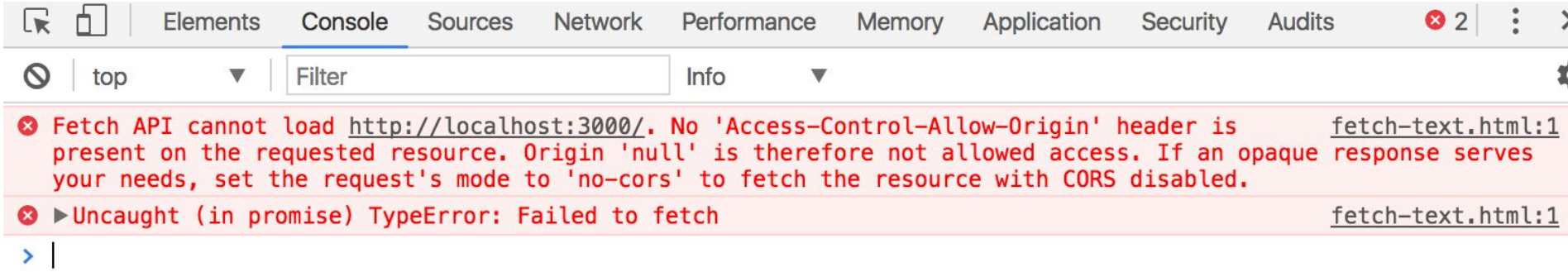
```
function onTextReady(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('http://localhost:3000/')
    .then(onResponse)
    .then(onTextReady);
```

fetch() to localhost

But if we try fetching to localhost from file://

```
fetch('http://localhost:3000')
    .then(onResponse)
    .then(onTextReady);
```

We get this CORS error:



Recall: CORS

CORS: **C**ross-**O**rigin **R**esource **S**haring (wiki)

- Browser policies for what resources a web page can load
- You **cannot** make cross-origin requests by default for:
  - Resources loaded via `fetch()` or XHR

The problem is that we are trying to `fetch()`
http://localhost:3000 from **file:///**

- Since the two resources have different origins, this is disallowed by default CORS policy

Cross-origin solutions

The problem is that we are trying to `fetch()`
http://localhost:3000 from **file:///**

Two ways to solve this:

1. Change the server running on localhost:3000 to allow cross-origin requests, i.e. to allow requests from different origins (such as file:///)

2. **Preferred solution:** Load the frontend code statically from the same server, so that the request is from the same origin

Solution 1: Enable CORS

```
app.get('/', function (req, res) {
  res.header("Access-Control-Allow-Origin", "*");
  res.send('Main page!');
});
```

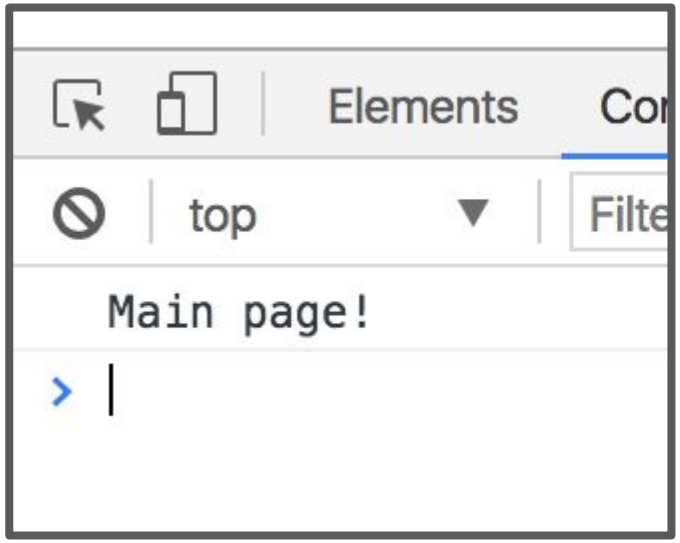
You can set an Access-Control-Allow-Origin HTTP header before sending your response.

- This is the server saying to the browser in its response: "Hey browser, I'm totally fine with websites of any origin requesting this file."

Solution 1: Enable CORS

Now the fetch will succeed (GitHub):

```
function onTextReady(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('http://localhost:3000/')
    .then(onResponse)
    .then(onTextReady);
```



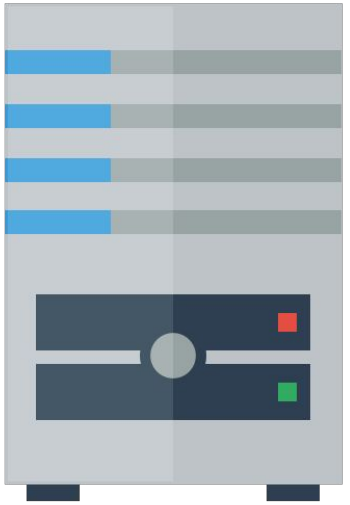
Cross-origin solutions

However, you wouldn't have to enable CORS at all if you were making requests from the same origin.

Preferred solution: Load the frontend code statically from the same server, so that the request is from the same origin.

Recall: Web services

Sometimes when you type a URL into your browser, the URL represents **an API endpoint**.



That is, the URL represents **a parameterized request**, and the web server dynamically generates a response to that request.

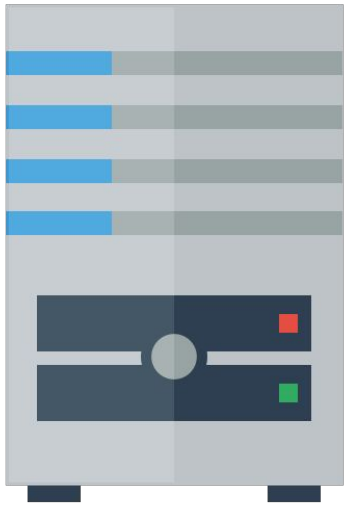
That's how our NodeJS server treats routes defined like this:

```
app.get('/hello', function (req, res) {
  res.send('GET hello!');
});
```

Recall: File servers

Other times when you type a URL in your browser, the URL is a **path to a file** on the hard drive of the server:

- The web server software grabs that file from the server's local file system, and sends back its contents to you



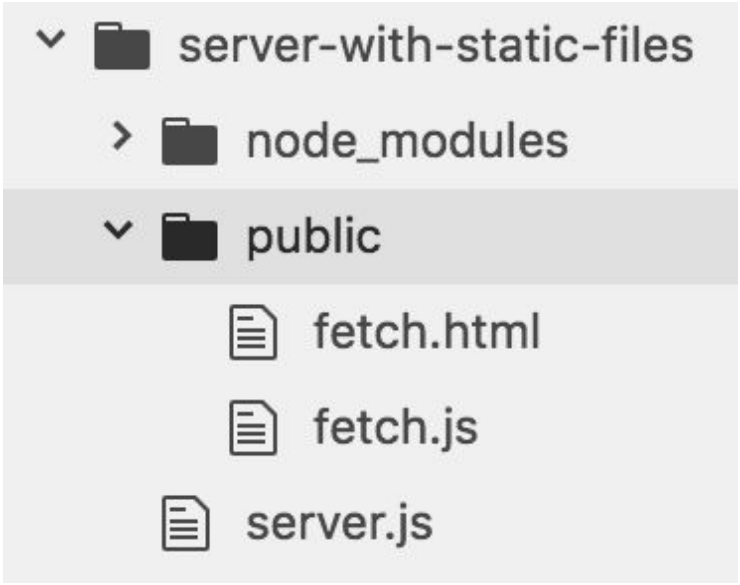
We can make our NodeJS server also sometimes serve files "statically," meaning instead of treating **all** URLs as API endpoints, some URLs will be treated as file paths.

Solution 2: Statically served files

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

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

app.get('/', function (req, res) {
  res.send('Main page!');
});
```

This line of code makes our server now start serving the files in the 'public' directory directly.

Server static data




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

Now Express will serve:

http://localhost:3000/fetch-text.html

http://localhost:3000/fetch-text.js

http://localhost:3000/style.css

Express looks up the files relative to the static directory, so the name of the static directory ("public" in this case) is not part of the URL (GitHub)

Different `fetch()` methods

`fetch()` with POST

```
app.post('/hello', function (req, res) {
  res.send('POST hello!');
});
```

On the server-side, we have defined a function in `app.post()` to handle POST requests to `/hello`.

Q: How do we make a POST request via `fetch()`?

Changing the `fetch()` method

```
fetch('/hello', { method: 'POST' })
    .then(onResponse)
    .then(onTextReady);
```

Q: How do we make a POST request via `fetch()`?

A: We can change the HTTP method via a second parameter to `fetch()`, which specifies an options object:

- `method:` specifies the HTTP request method, e.g. POST, PUT, PATCH, DELETE, etc.
  - GET is the default value.

fetch() with POST

```
function onTextReady(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('/hello', { method: 'POST' })
    .then(onResponse)
    .then(onTextReady);
```

GitHub

Sending data to the server

Route parameters

When we used the Spotify API, we saw a few ways to send information to the server via our `fetch()` request.

Example: Spotify Album API

`https://api.spotify.com/v1/albums/`**`7aDBFWp72Pz4NZEtVBANi9`**

- The last part of the URL is a **parameter** representing the album id, `7aDBFWp72Pz4NZEtVBANi9`

A parameter defined in the URL of the request is often called a "**route parameter**."

Route parameters

Q: How do we read route parameters in our server?

A: We can use the ***:variableName*** syntax in the path to specify a route parameter (Express docs):

```
app.get('/hello/:name', function (req, res) {
  const routeParams = req.params;
  const name = routeParams.name;
  res.send('GET: Hello, ' + name);
});
```

We can access the route parameters via **`req.params`**.

Route parameters

```
app.get('/hello/:name', function (req, res) {
  const routeParams = req.params;
  const name = routeParams.name;
  res.send('GET: Hello, ' + name);
});
```

GitHub

Route parameters

You can define multiple route parameters in a URL (docs):

```
app.get('/flights/:from-:to', function (req, res) {
  const routeParams = req.params;
  const from = routeParams.from;
  const to = routeParams.to;
  res.send('GET: Flights from ' + from + ' to ' + to);
});
```

GitHub

Query parameters

The Spotify Search API was formed a little differently:

Example: Spotify Search API

`https://api.spotify.com/v1/search?type=album&q=beyonce`

- There were two query parameters sent to the Spotify search endpoint:
  - `type`, whose value is `album`
  - `q`, whose value is `beyonce`

Query parameters

Q: How do we read query parameters in our server?

A: We can access query parameters via **`req.query`**:

```
app.get('/hello', function (req, res) {
  const queryParams = req.query;
  const name = queryParams.name;
  res.send('GET: Hello, ' + name);
});
```

GitHub

Query params with POST

You can send query parameters via POST as well:

```
function onTextReady(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('/hello?name=Victoria', { method: 'POST' })
    .then(onResponse)
    .then(onTextReady);
```

(WARNING: We will not be making POST requests like this!

We will be sending data in the body of the request instead of via query params.)

Query params with POST

These parameters are accessed the same way:

```
app.post('/hello', function (req, res) {
  const queryParams = req.query;
  const name = queryParams.name;
  res.send('POST: Hello, ' + name);
});
```

GitHub

(WARNING: We will not be making POST requests like this!

We will be sending data in the body of the request instead of via query params.)

POST message body

However, generally it is poor style to send data via query parameters in a POST request.

Instead, you should specify a message body in your `fetch()` call:

```
const message = {
  name: 'Victoria',
  email: 'vrk@stanford.edu'
};
const serializedMessage = JSON.stringify(message);
fetch('/helloemail', { method: 'POST', body: serializedMessage })
    .then(onResponse)
    .then(onTextReady);
```

POST message body

Handling the message body in NodeJS/Express is a little messy (GitHub):

```
app.post('/helloemail', function (req, res) {
  let data = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk) {
     data += chunk;
  });

  req.on('end', function() {
    const body = JSON.parse(data);
    const name = body.name;
    const email = body.email;
    res.send('POST: Name: ' + name + ', email: ' + email);
  });
});
```

body-parser

We can use the body-parser library to help:

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

This is not a NodeJS API library, so we need to install it:

```
$ npm install body-parser
```

body-parser

We can use the body-parser library to help:

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

const jsonParser = bodyParser.json();
```

This creates a JSON parser stored in `jsonParser`, which we can then pass to routes whose message bodies we want parsed as JSON.

POST message body

Now instead of this code:

```
app.post('/helloemail', function (req, res) {
  let data = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk) {
     data += chunk;
  });

  req.on('end', function() {
    const body = JSON.parse(data);
    const name = body.name;
    const email = body.email;
    res.send('POST: Name: ' + name + ', email: ' + email);
  });
});
```

POST message body

We can access the message body through `req.body`:

```
app.post('/helloparsed', jsonParser, function (req, res) {
  const body = req.body;
  const name = body.name;
  const email = body.email;
  res.send('POST: Name: ' + name + ', email: ' + email);
});
```

GitHub

POST message body

We can access the message body through `req.body`:

```
app.post('/helloparsed', jsonParser, function (req, res) {
  const body = req.body;
  const name = body.name;
  const email = body.email;
  res.send('POST: Name: ' + name + ', email: ' + email);
});
```

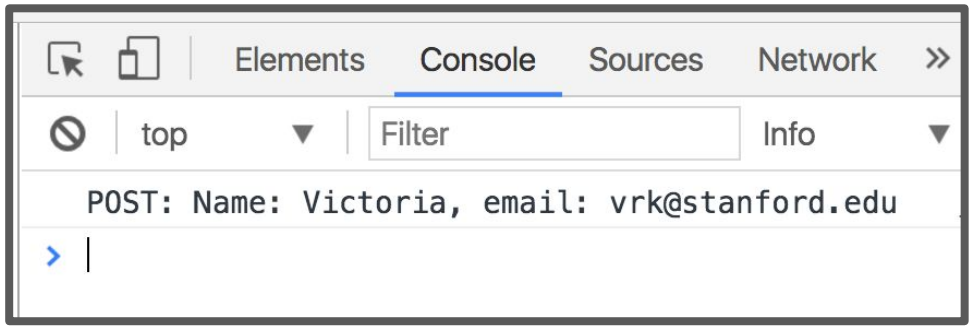
GitHub

Note that we also had to add the `jsonParser` as a parameter when defining this route.

POST message body

Finally, we need to add JSON content-type headers on the `fetch()`-side (GitHub):

```
const message = {
  name: 'Victoria',
  email: 'vrk@stanford.edu'
};
const fetchOptions = {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(message)
};
fetch('/helloparsed', fetchOptions)
  .then(onResponse)
  .then(onTextReady);
```



Recap

You can deliver parameterized information to the server in the following ways:

1. Route parameters
2. GET request with query parameters
   (DISCOURAGED: POST with query parameters)
3. POST request with message body

Q: When do you use route parameters vs query parameters vs message body?

GET vs POST

- Use GET requests for retrieving data, not writing data

- Use POST requests for writing data, not retrieving data
  You can also use more specific HTTP methods:
  - PATCH: Updates the specified resource
  - DELETE: Deletes the specified resource

There's nothing technically preventing you from breaking these rules, but you should use the HTTP methods for their intended purpose.

Route params vs Query params

Generally follow these rules:

- Use **route parameters** for required parameters for the request
- Use **query parameters** for:
  - Optional parameters
  - Parameters whose values can have spaces

These are conventions and are not technically enforced, nor are they followed by every REST API.

Example: Spotify API

The Spotify API mostly followed these conventions:

https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9

- The Album ID is required and it is a route parameter.

https://api.spotify.com/v1/search?type=album&q=the%20weeknd&limit=10

- q is required but might have spaces, so it is a query parameter
- limit is optional and is a query parameter
- type is required but is a query parameter (breaks convention)

Notice both searches are GET requests, too

package.json

Installing dependencies

In our examples, we had to install the express and body-parser npm packages.

```
$ npm install express
$ npm install body-parser
```

These get written to the `node_modules` directory.

Uploading server code

When you upload NodeJS code to a GitHub repository (or any code repository), you should **not** upload the `node_modules` directory:

- You shouldn't be modifying code in the `node_modules` directory, so there's no reason to have it under version control
- This will also increase your repo size significantly

Q: But if you don't upload the node_modules directory to your code repository, how will anyone know what libraries they need to install?

Managing dependencies

If we don't include the `node_modules` directory in our repository, we need to somehow tell other people what `npm` modules they need to install.

`npm` provides a mechanism for this: `package.json`

package.json

You can put a file named `package.json` in the root directory of your NodeJS project to specify metadata about your project.

Create a `package.json` file using the following command:

```
$ npm init
```

This will ask you a series of questions then generate a `package.json` file based on your answers.

Auto-generated package.json

```
{
  "name": "fetch-to-server",
  "version": "1.0.0",
  "description": "Example of fetching to a server",
  "main": "server.js",
  "dependencies": {
    "body-parser": "^1.17.1",
    "express": "^4.15.2"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "Victoria Kirst",
  "license": "ISC"
}
```

GitHub

Saving deps to package.json

Now when you install packages, you should pass in the --save parameter:

```
$ npm install --save express
$ npm install --save body-parser
```

This will also add an entry for this library in package.json.

```
"dependencies": {
  "body-parser": "^1.17.1",
  "express": "^4.15.2"
},
```

Saving deps to package.json

If you remove the node_modules directory:

```
$ rm -rf node_modules
```

You can install your project dependencies again via:

```
$ npm install
```

- This also allows people who have downloaded your code from GitHub to install all your dependencies with one command instead of having to install all dependencies individually.

npm scripts

Your package.json file also defines scripts:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node server.js"
},
```

You can run these scripts using `$ npm` ***scriptName***

E.g. the following command runs "node server.js"

```
$ npm start
```