Interactive Web Programming

1st semester of 2021

Murilo Camargos
(**murilo.filho@fgv.br**)

Heavily based on **Victoria Kirst** slides

Today's schedule

Today

- Web application architecture
  - Server-side rendering with Handlebars
- Modules, Middleware, Routes
- Web application architecture
  - Single-page web app
- Authentication

Next class (our last!)

- Important ideas we didn't cover (e.g. testing and accessibility)
- Libraries and frameworks

NodeJS and MongoDB

Errata from last class

Last class we created a connection like this

```
const { MongoClient } = require('mongodb');
const express = require('express');

const app = express();
const MONGO_URL = 'mongodb://localhost:27017/eng-dict'

async function startServer() {
    const conn = await MongoClient.connect(MONGO_URL);
    const collection = db.collection('words');
    await app.listen(3000);
    console.log('Example app listening on port 3000');
}
startServer();
```

Errata from last class

Current versions will accept this:

```
const { MongoClient } = require('mongodb');
const express = require('express');

const app = express();
const MONGO_URL = 'mongodb://localhost:27017/'

async function startServer() {
    const conn = await MongoClient.connect(MONGO_URL, {useUnifiedTopology: true});
    const db = conn.db('eng-dict');
    const collection = db.collection('words');
    await app.listen(3000);
    console.log('Example app listening on port 3000');
}
startServer();
```

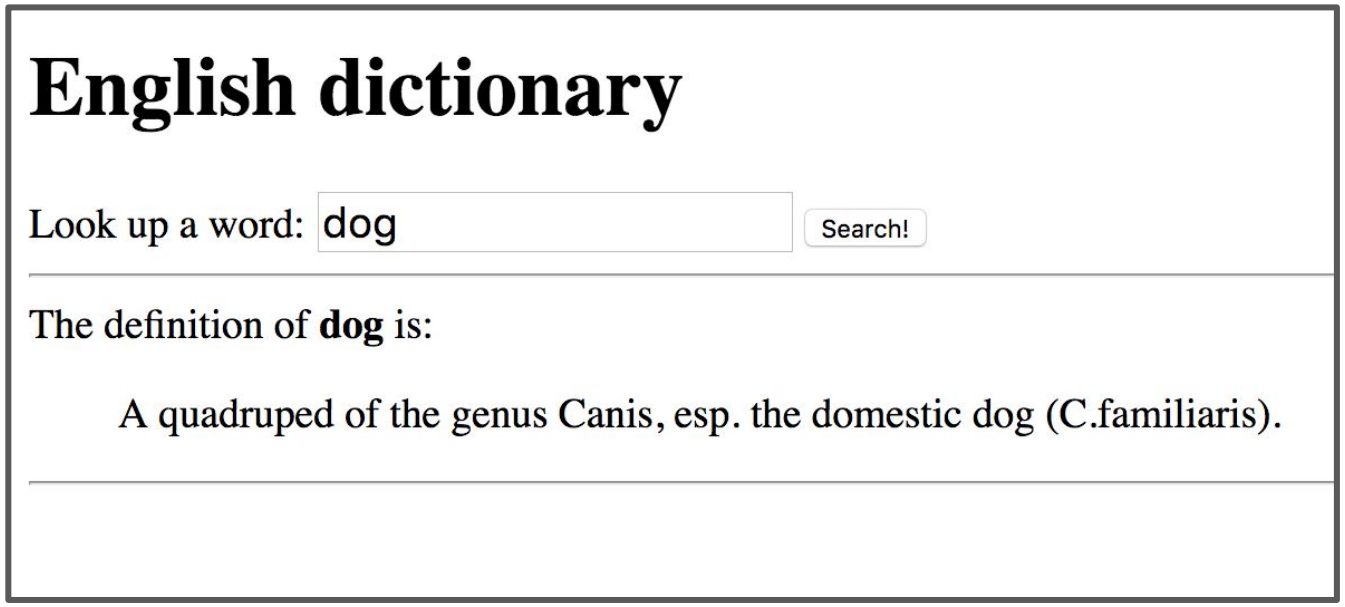
Errata from last class

Current versions will accept this:

```
const { MongoClient } = require('mongodb');
const express = require('express');

const app = express();
const MONGO_URL = 'mongodb://localhost:27017/'

async function startServer() {
    const conn = await MongoClient.connect(MONGO_URL, {useUnifiedTopology: true});
    const db = conn.db('eng-dict');
    const collection = db.collection('words');
    await app.listen(3000);
    console.log('Example app listening on port 3000');
}
startServer();
```

Dictionary example

Let's go back to our dictionary example that let us look up the definition of words (GitHub):

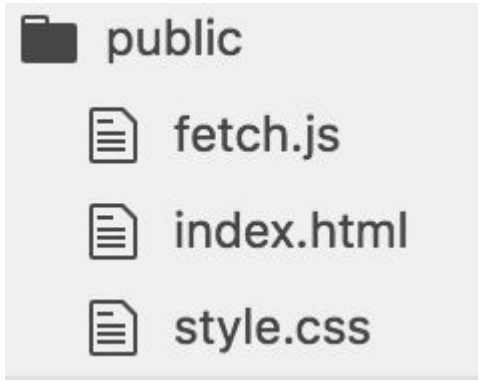


Current code: Static routes

We've defined our server to:

1. Statically serve index.html / style.css / fetch.js

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



Current code: Static routes

We've defined our server to:

1. Statically serve index.html / style.css / fetch.js

So when there's a request to localhost:3000/ ...

Current code: Static routes

We've defined our server to:

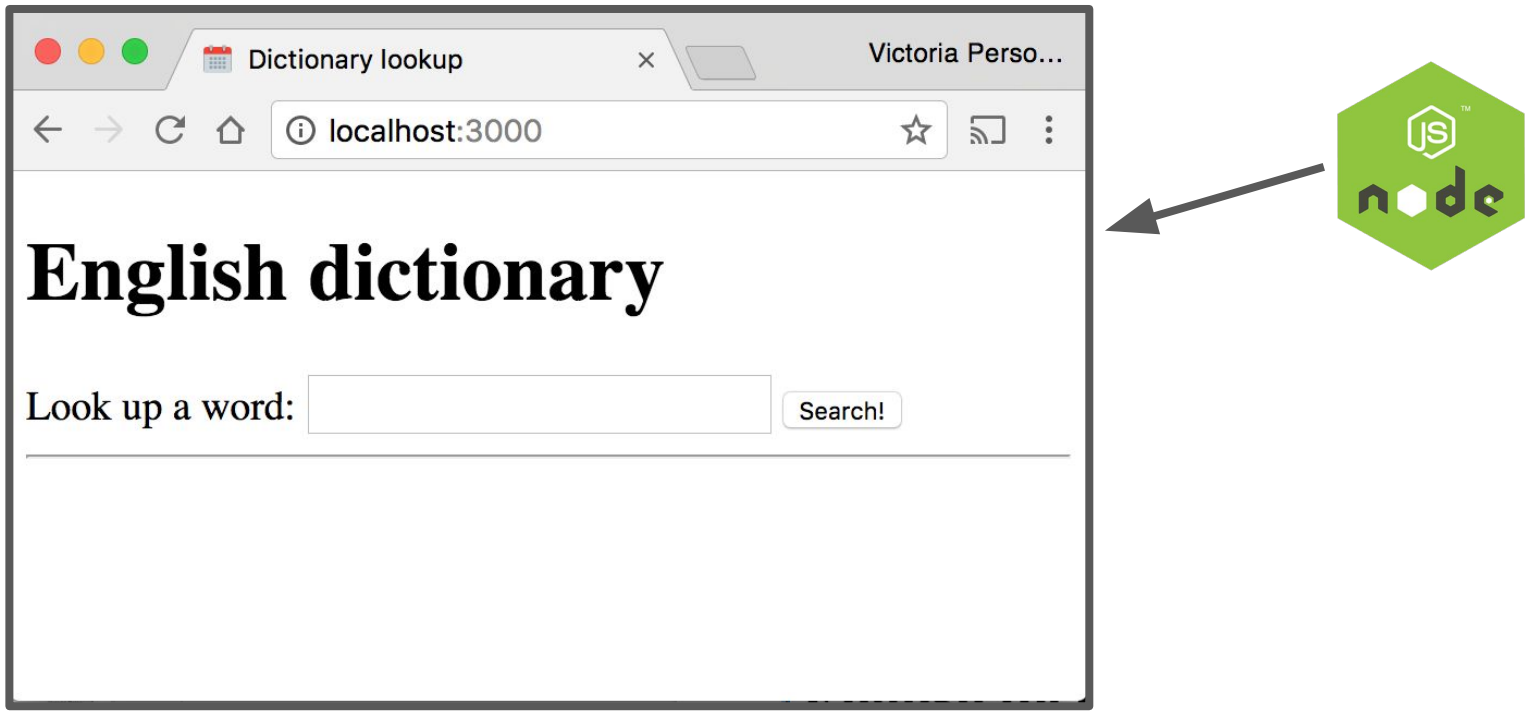
1. Statically serve index.html / style.css / fetch.js

- Our NodeJS server.js program checks the public directory to see if "index.html" exists, which is does, so it replies with that file.
- It similarly provides fetch.js and style.css, embedded in index.html.

Current code: Static routes

We've defined our server to:

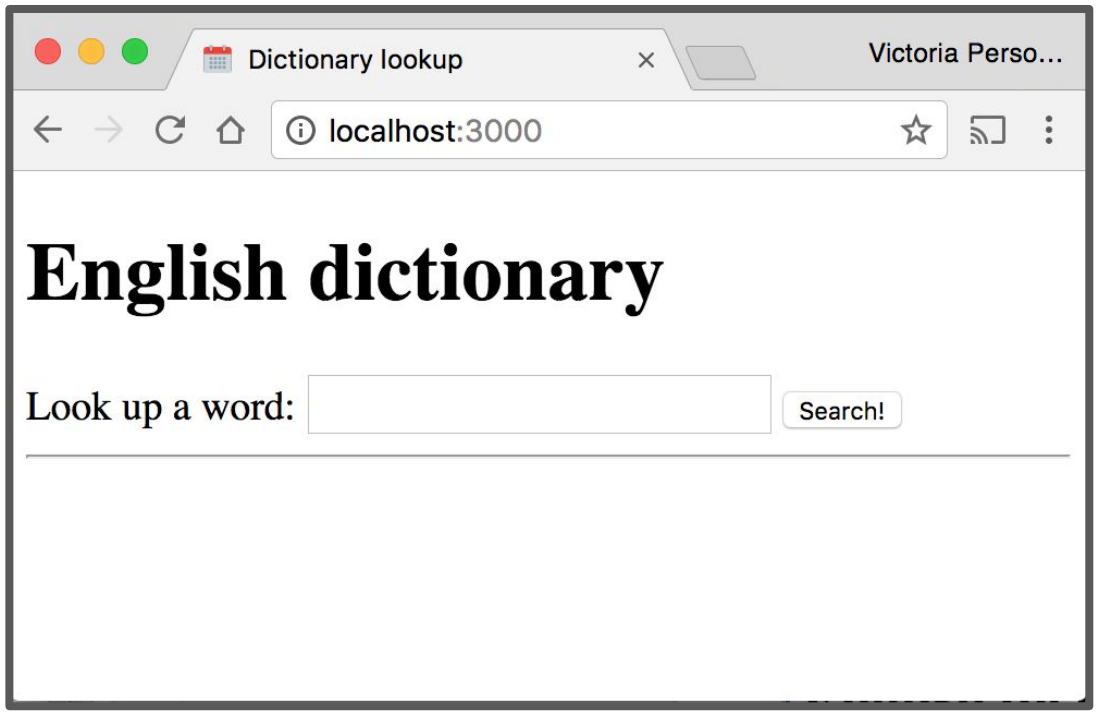
1. Statically serve index.html / style.css / fetch.js



Current code: JSON route

We've also defined our server to:

2. Return JSON in response to request to lookup/<word>



Current code: JSON route

We've also defined our server to:

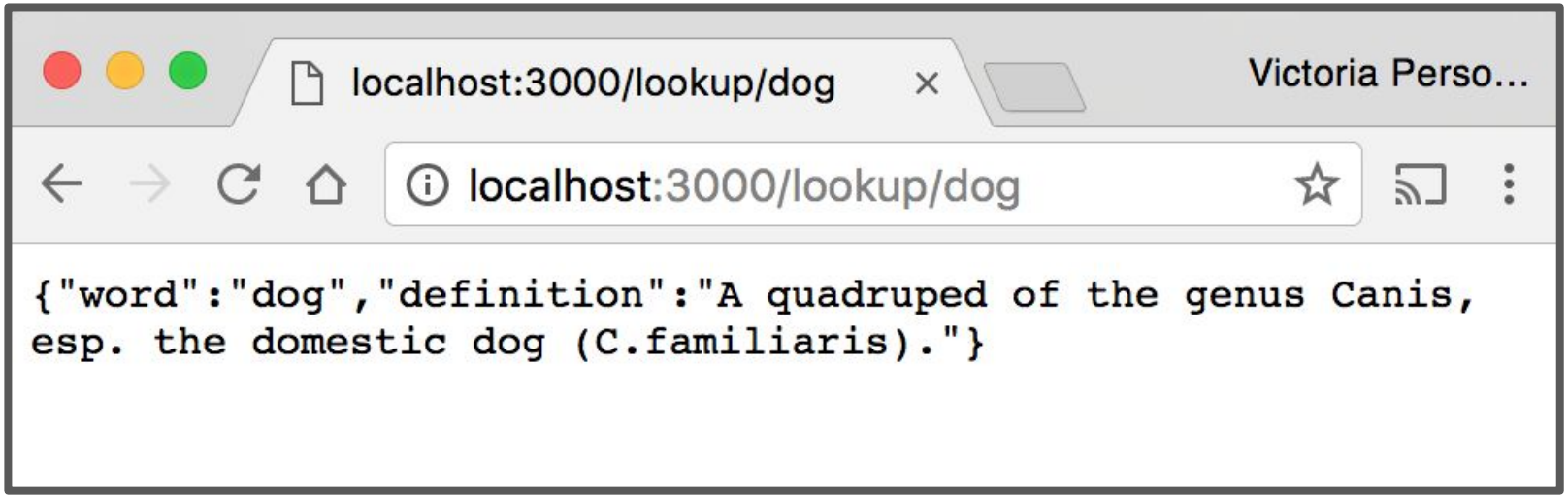
2. Return JSON in response to request to lookup/<word>

```
async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);

  const response = {
    word: word,
    definition: result ? result.definition : ''
  };
  res.json(response);
}
app.get('/lookup/:word', onLookupWord);
```

Current code: JSON route

We've also defined our server to:

2. Return JSON in response to request to lookup/<word>



If we navigate to http://localhost:3000/lookup/dog, we see the raw JSON response.

Current code: JSON route

We've also defined our server to:

2. Return JSON in response to request to lookup/<word>

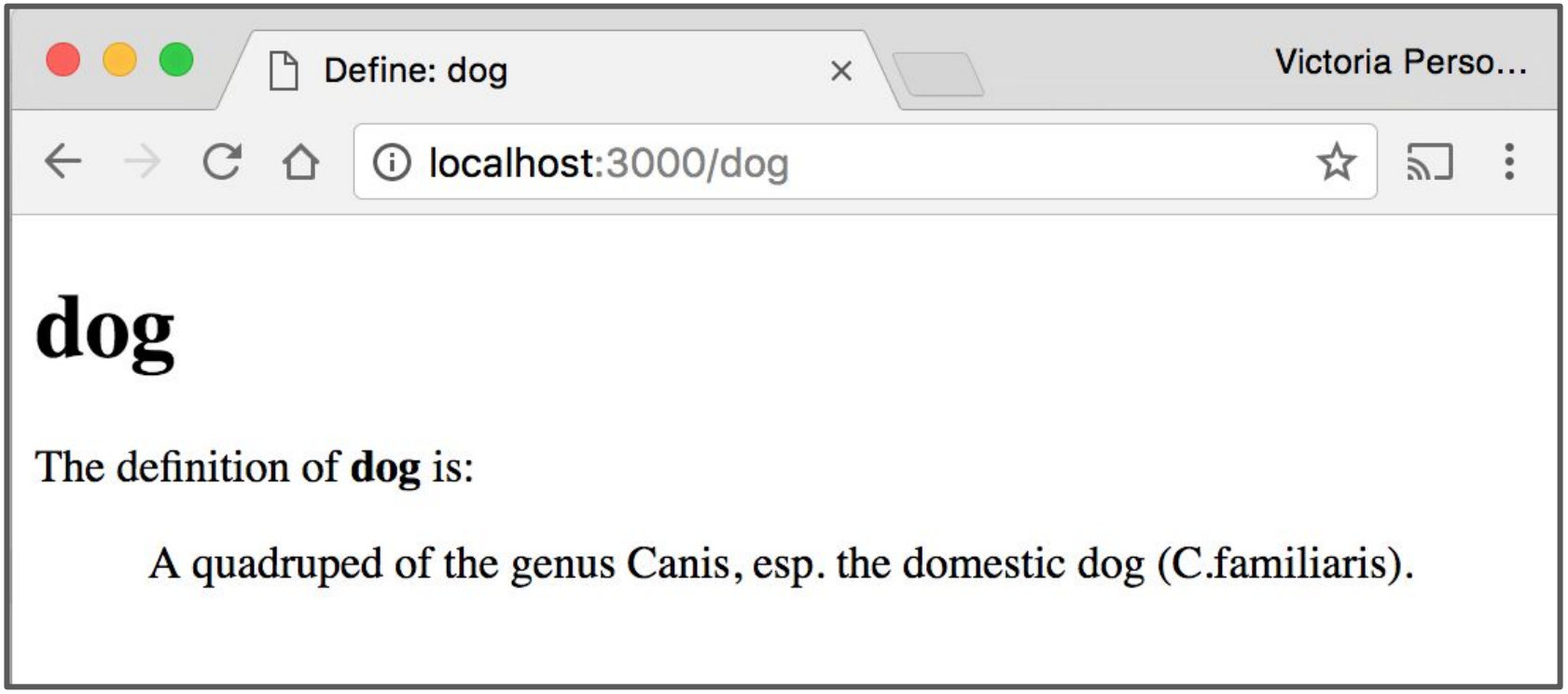
But how we actually query our JSON route is through `fetch()`.

```
async function onSearch(event) {
  event.preventDefault();
  const input = document.querySelector('#word-input');
  const word = input.value.trim();

  const result = await fetch('/lookup/' + word);
  const json = await result.json();

  const wordDisplay = results.querySelector('#word');
  const defDisplay = results.querySelector('#definition');
  wordDisplay.textContent = json.word;
  defDisplay.textContent = json.definition;
}
```

Dynamically generated web pages

Dictionary example

What if we wanted each definition to also have its own page: **http://localhost:3000/dog** should show a web page with the definition of "dog:"



Naive solution: static files

One solution would be to create 1000s of very similar web pages:

cat/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Define: cat</title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <h1>cat</h1>
    <div id="results" class="hidden">
      The definition of <strong id="word">cat</strong> is:
      <blockquote id="definition">An animal of various species of the genera Felis
and Lynx. Thedomestic cat is Felis domestica. The European wild cat (Felis
catus)is much larger than the domestic cat. In the United States the namewild cat
is commonly applied to the bay lynx (Lynx rufus) See Wildcat, and Tiger cat.
</blockquote>
    </div>
  </body>
</html>
```

Naive solution: static files

One solution would be to create 1000s of very similar web pages:

category/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Define: category</title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <h1>category</h1>
    <div id="results" class="hidden">
      The definition of <strong id="word">category</strong> is:
      <blockquote id="definition">One of the highest classes to which the objects
of knowledge orthought can be reduced, and by which they can be arranged in
asystem; an ultimate or undecomposable conception; a predicament.The categories or
predicaments -- the former a Greek word, the latterits literal translation in the
Latin language -- were intended byAristotle and his followers as an enumeration of
all things capableof being named; an enumeration by the summa genera i.e., the
mostextensive classes into which things could be distributed. J. S. Mill.
</blockquote>
```

Naive solution: static files

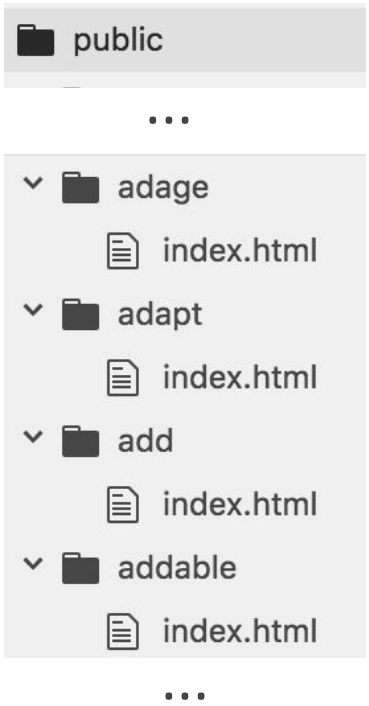
One solution would be to create 1000s of very similar web pages:

catastrophe/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Define: catastrophe</title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <h1>catastrophe</h1>
    <div id="results" class="hidden">
      The definition of <strong id="word">catastrophe</strong> is:
      <blockquote id="definition">A violent and widely extended change in the
surface of theearth, as, an elevation or subsidence of some part of it, effected
byinternal causes. Whewell.</blockquote>
    </div>
  </body>
</html>
```

Naive solution: static files

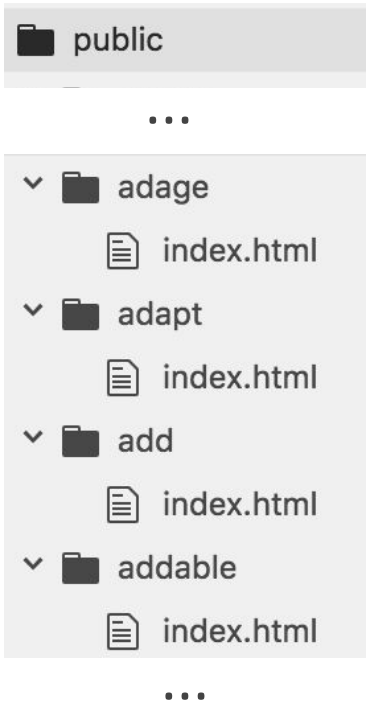
We could put each of these files under `public/` and have a unique HTML file for each word of the dictionary.



Naive solution: static files

However, that would be a pain:

- Very time-consuming to hand-code
- Even if we wrote a script to generate the 1000s of HTML files, it'd be annoying to have to rerun the script every time there's a new word, new definition, if there's a change in the HTML format, etc.



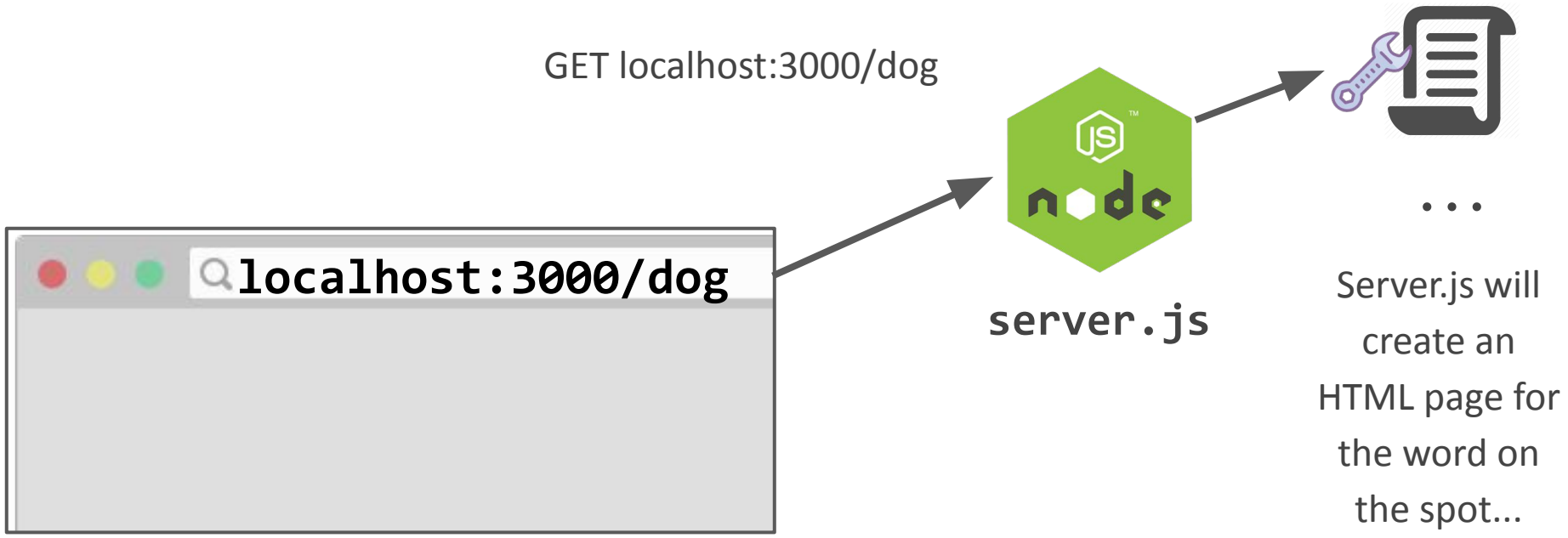
Dynamically generated pages

Instead, we'll make our server dynamically generate a web page for the word as soon as it is requested:

GET localhost:3000/dog

Dynamically generated pages

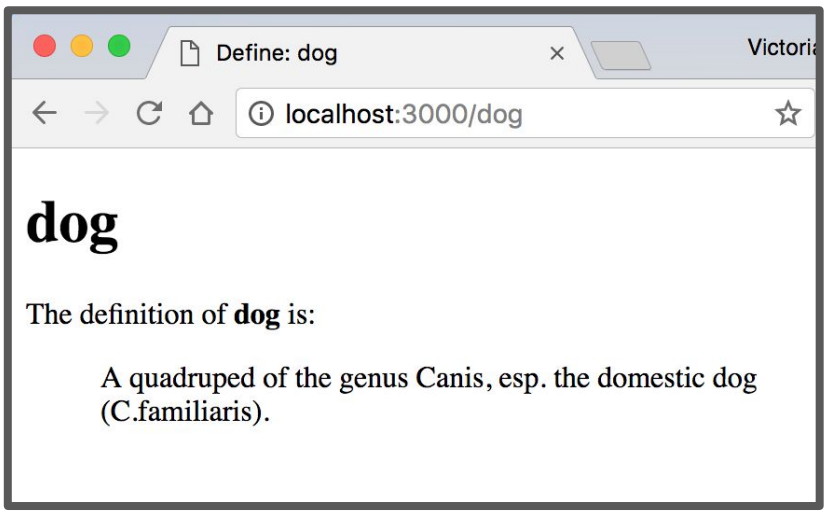
Instead, we'll make our server dynamically generate a web page for the word as soon as it is requested:

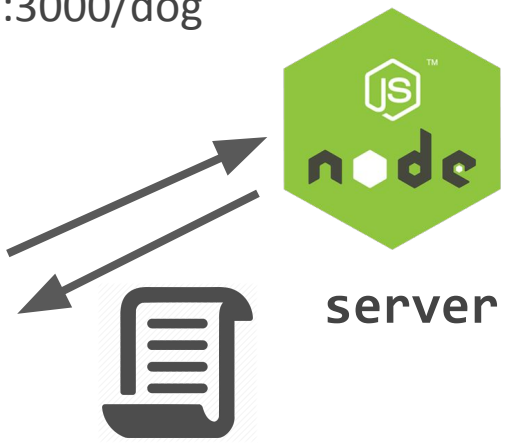


Dynamically generated pages

Instead, we'll make our server dynamically generate a web page for the word as soon as it is requested:

GET localhost:3000/dog

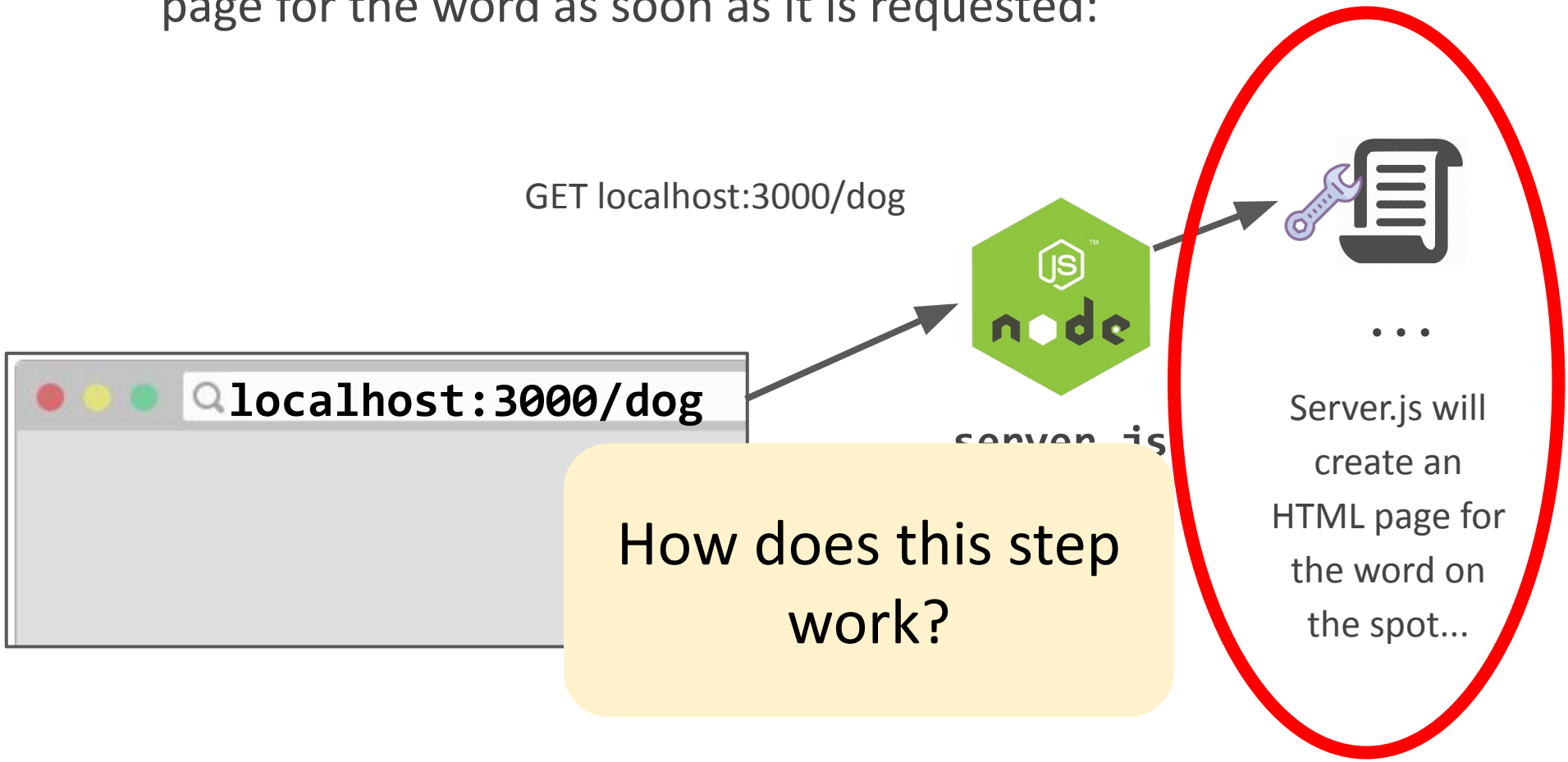




And we see the HTML displayed.

Dynamically generated pages

Instead, we'll make our server dynamically generate a web page for the word as soon as it is requested:



Recall: Web app architectures

Structuring a web app

There are roughly 4 strategies for architecting a web application:

1. **Server-side rendering:**
   Server sends a new HTML page for each unique path
2. **Single-page application:**
   Server sends the exact same web page for every unique path (and the page runs JS to change what it look like)
3. Combination of 1 and 2 ("**Isomorphic**" / "**Universal**")
4. **Progressive Loading**

Structuring a web app

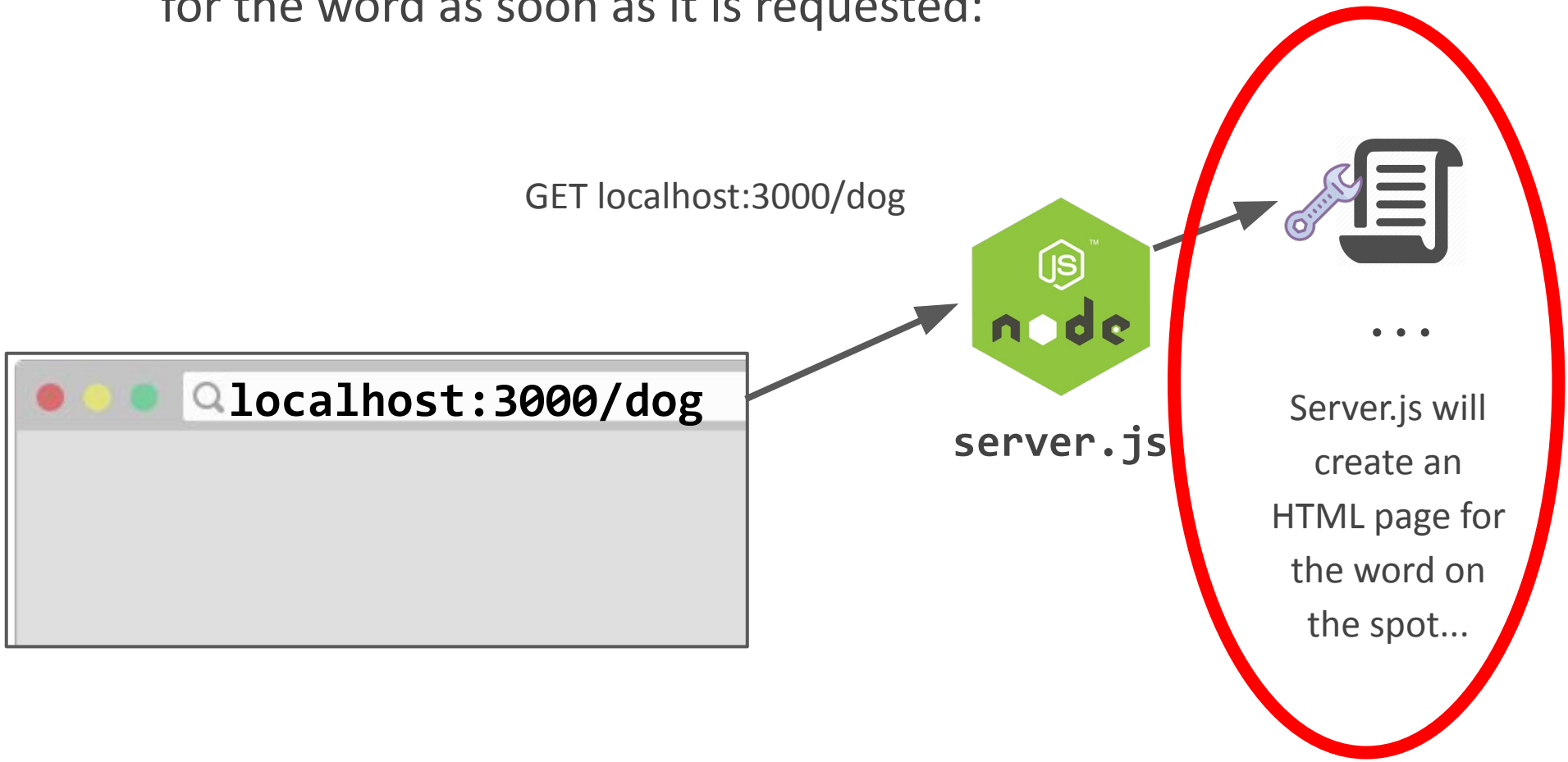
There are roughly 4 strategies for architecting a web application:

1. **Server-side rendering:**
   Server sends a new HTML page for each unique path

   → Let's start with this one.

Dynamically generated pages

We'll make our server dynamically generate a web page for the word as soon as it is requested:



HTML Strings

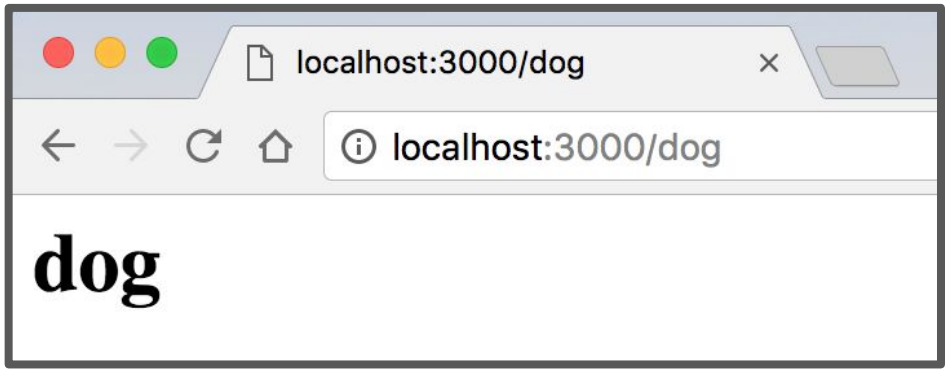
We start with defining a dummy route:

```
async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const result = `<h1>${word}</h1>`;
  res.send(result);
}
app.get('/:word', onViewWord);
```

HTML Strings

We start with defining a dummy route:

```
async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const result = `<h1>${word}</h1>`;
  res.send(result);
}
app.get('/:word', onViewWord);
```



This just echoes what we passed in as a query parameter.

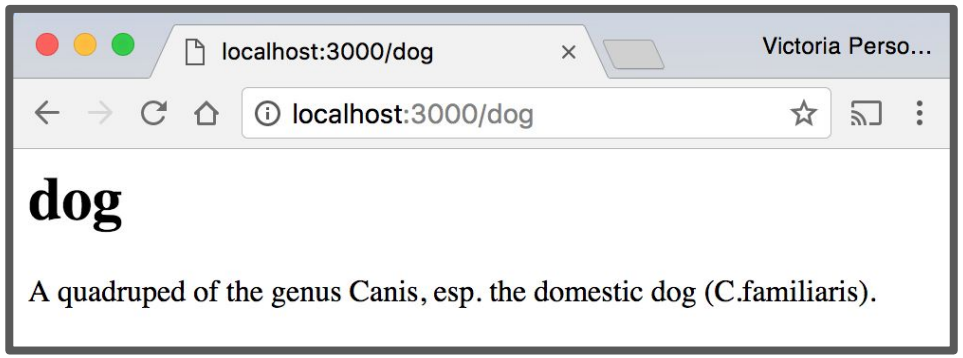
HTML Strings

Now we look up and show the definition too:

```
async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  let response = `<h1>${word}</h1>`;
  response += `<p>${definition}</p>`;
  res.end(response);
}
app.get('/:word', onViewWord);
```

HTML Strings

```
async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  let response = `<h1>${word}</h1>`;
  response += `<p>${definition}</p>`;
  res.end(response);
}
app.get('/:word', onViewWord);
```



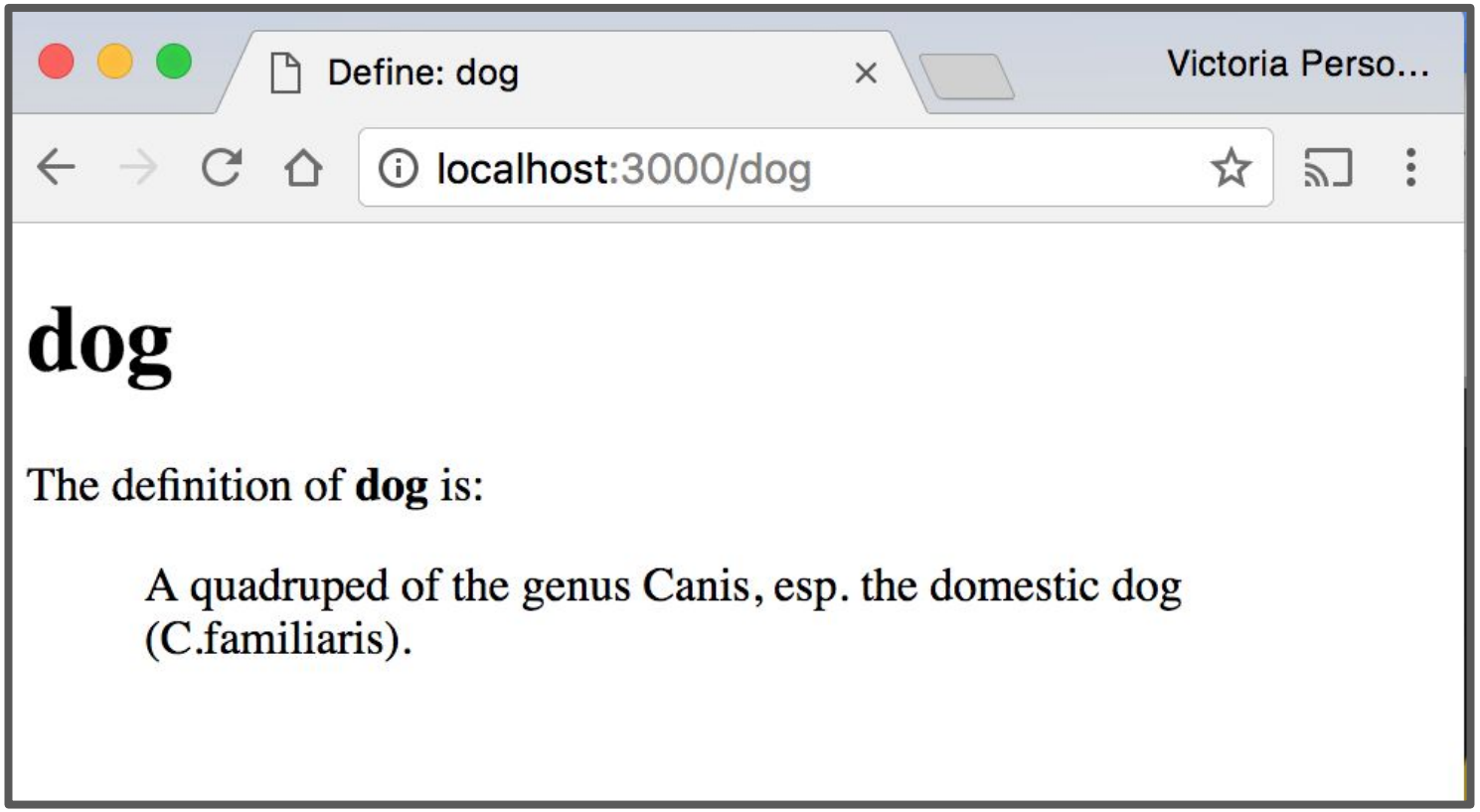
HTML Strings

We can make our HTML response a little fancier:

```
  const response =
    `<!DOCTYPE html>
    <html>
      <head>
        <meta charset="utf-8">
        <title>Define: ${word}</title>
        <link rel="stylesheet" href="/css/style.css">
      </head>
      <body>
        <h1>${word}</h1>
        <div id="results" class="hidden">
          The definition of <strong id="word">${word}</strong> is:
          <blockquote id="definition">${definition}</blockquote>
        </div>
      </body>
    </html>`;
  res.end(response);
}
```

HTML Strings

We can make our HTML response a little fancier:



HTML Strings

This works, but now we have a big HTML string in our server code:

```
async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  const response =
    `<!DOCTYPE html>
    <html>
      <head>
        <meta charset="utf-8">
        <title>Define: ${word}</title>
        <link rel="stylesheet" href="/css/style.css">
      </head>
      <body>
        <h1>${word}</h1>
        <div id="results" class="hidden">
          The definition of <strong id="word">${word}</strong> is:
          <blockquote id="definition">${definition}</blockquote>
        </div>
      </body>
    </html>`;
  res.end(response);
}
app.get('/:word', onViewWord);
```

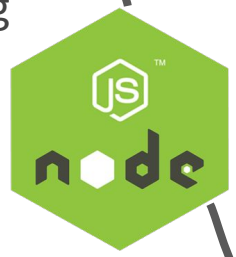
Template Engines

Goal: HTML Template

We want our NodeJS code to be able to take an HTML template, fill in its placeholder values, and return the completed page:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Define: {{ word }}</title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <h1>{{ word }}</h1>
    <div id="results" class="hidden">
      The definition of <strong id="word">{{ word }}</strong> is:
      <blockquote id="definition">{{ definition }}</blockquote>
    </div>
  </body>
</html>
```

GET localhost:3000/dog

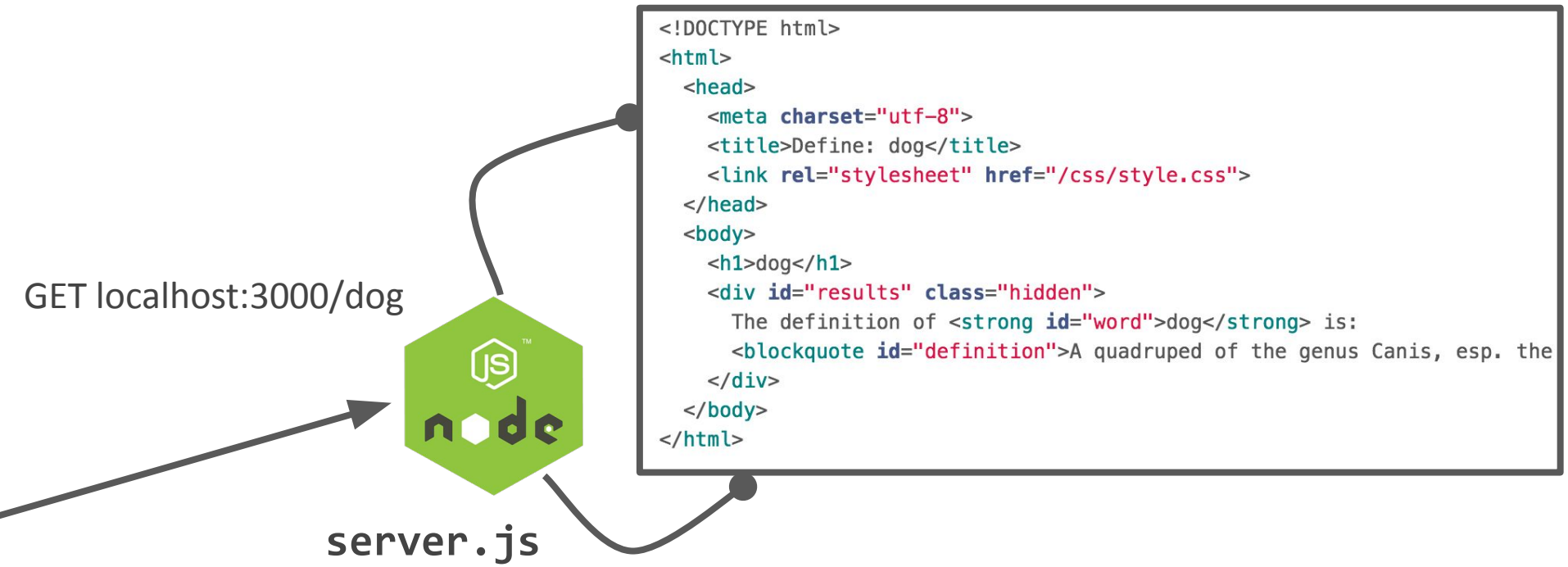


server.js

```
word: "dog",
definition: "A quadruped
of the genus Canis, ..."
```

Goal: HTML Template

We want our NodeJS code to be able to take an HTML template, fill in its placeholder values, and return the completed page:



Template Engine

Template Engine: Allows you to define templates in a text file, then fill out the contents of the template in JavaScript.

- Node will replace the variables in a template file with actual values, then it will send the result to the client as an HTML file.

Some popular template engines:

- **Handlebars**: We'll be using this one
- Pug
- EJS

Handlebars: Template engine

- Handlebars lets you write templates in HTML
- You can embed `{{ placeholders }}` within the HTML that can get filled in via JavaScript.
- Your templates are saved in `.handlebars` files

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

Handlebars and NodeJS

You can setup Handlebars and NodeJS using the `express-handlebars` NodeJS library:

```
const exphbs  = require('express-handlebars');
...

const app = express();
const hbs = exphbs.create();
app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');
```

Dictionary example

So instead of our large template string:

```
async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  const response =
    `<!DOCTYPE html>
    <html>
      <head>
        <meta charset="utf-8">
        <title>Define: ${word}</title>
        <link rel="stylesheet" href="/css/style.css">
      </head>
      <body>
        <h1>${word}</h1>
        <div id="results" class="hidden">
          The definition of <strong id="word">${word}</strong> is:
          <blockquote id="definition">${definition}</blockquote>
        </div>
      </body>
    </html>`;
  res.end(response);
}
app.get('/:word', onViewWord);
```

Handlebars template

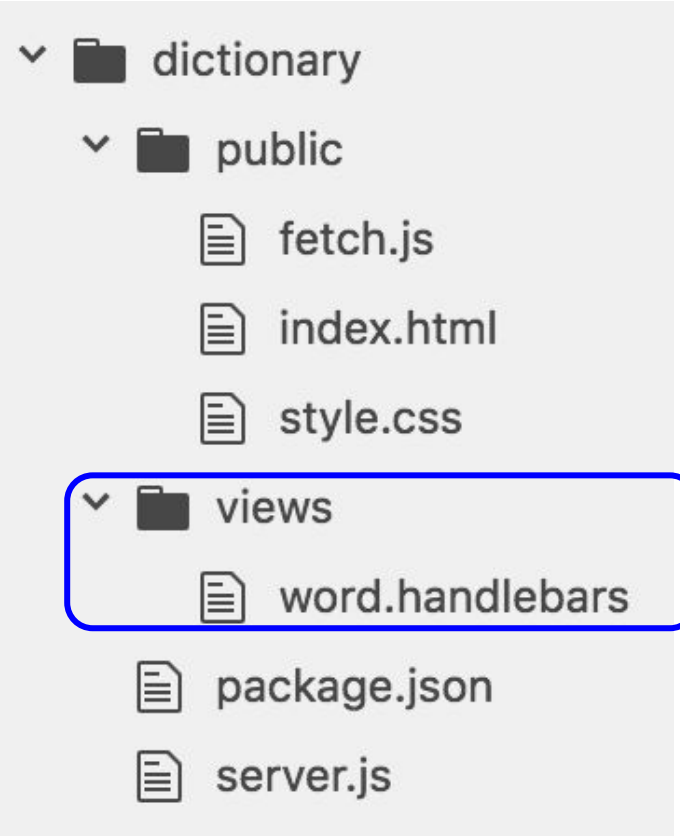
We can create a template in a file **`words.handlebars`**:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Define: {{ word }}</title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <h1>{{ word }}</h1>
    <div id="results" class="hidden">
      The definition of <strong id="word">{{ word }}</strong> is:
      <blockquote id="definition">{{ definition }}</blockquote>
    </div>
  </body>
</html>
```

`.handlebars` is the file extension for Handlebar templates.

Handlebars template

We save this in a directory called "views":



`views/` is the default directory in which Handlebars will look for templates.

Dictionary example

Now instead of returning a long string on the server side:

```
async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  const response =
    `<!DOCTYPE html>
    <html>
      <head>
        <meta charset="utf-8">
        <title>Define: ${word}</title>
        <link rel="stylesheet" href="/css/style.css">
      </head>
      <body>
        <h1>${word}</h1>
        <div id="results" class="hidden">
          The definition of <strong id="word">${word}</strong> is:
          <blockquote id="definition">${definition}</blockquote>
        </div>
      </body>
    </html>`;
  res.end(response);
}
app.get('/:word', onViewWord);
```

Set template engine

We configure our server to use Handlebars as the template engine:

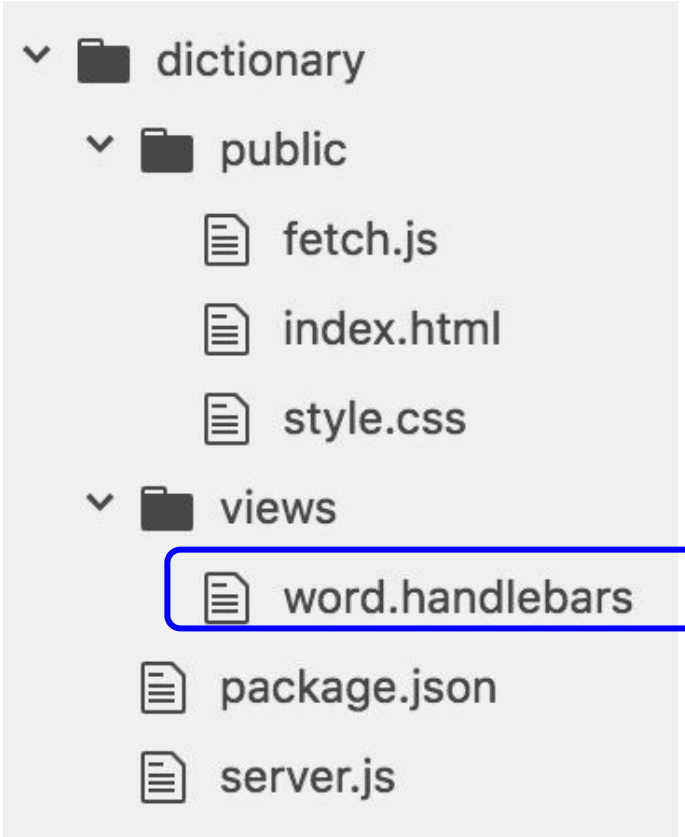
```
const exphbs  = require('express-handlebars');

const app = express();
const hbs = exphbs.create();
app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');
```

Call `res.render`

We call **`res.render`** to fill in the "word" Handlebars template.

```
async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  const placeholders = {
    word: word,
    definition: definition
  };
  res.render('word', placeholders);
}
app.get('/:word', onViewWord);
```

Call `res.render`




```
const placeholders = {
  word: word,
  definition: definition
};
res.render('word', placeholders);
```

The first parameter is the string name of the template to render, without the ".handlebars" extension.

Call `res.render`

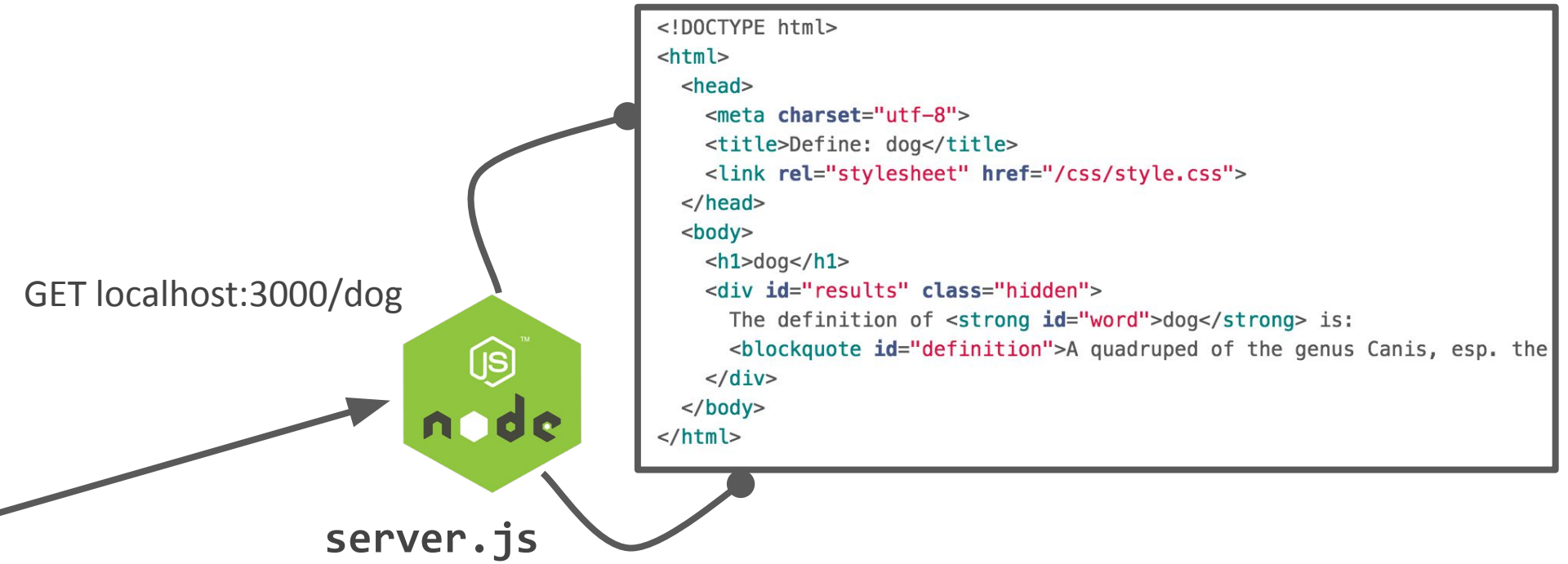
The second parameter contains the definitions of the variables to be filled out in the template.

```
const placeholders = {
  word: word,
  definition: definition
};
res.render('word', placeholders);
```

```
...
  <title>Define: {{ word }}</title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
  <h1>{{ word }}</h1>
  <div id="results" class="hidden">
    The definition of <strong id="word">{{ word }}</strong> is:
    <blockquote id="definition">{{ definition }}</blockquote>
...
```

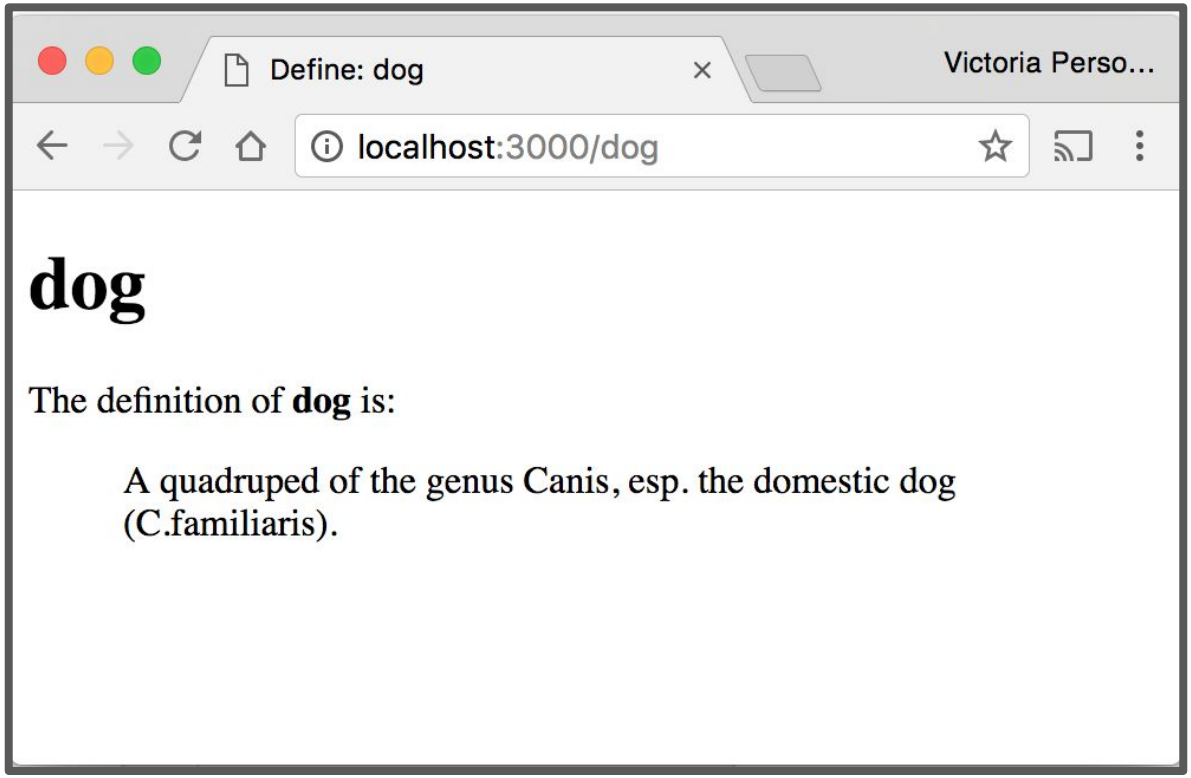
HTML Template

The Handlebars templating engine will return to the client the filled in template as HTML:

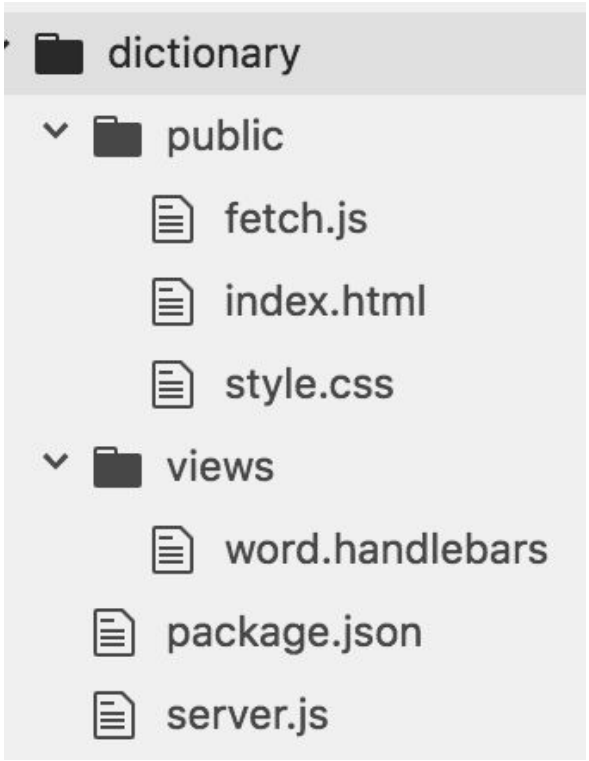


HTML Template

The Handlebars templating engine will return to the client the filled in template as HTML:



Note on templates



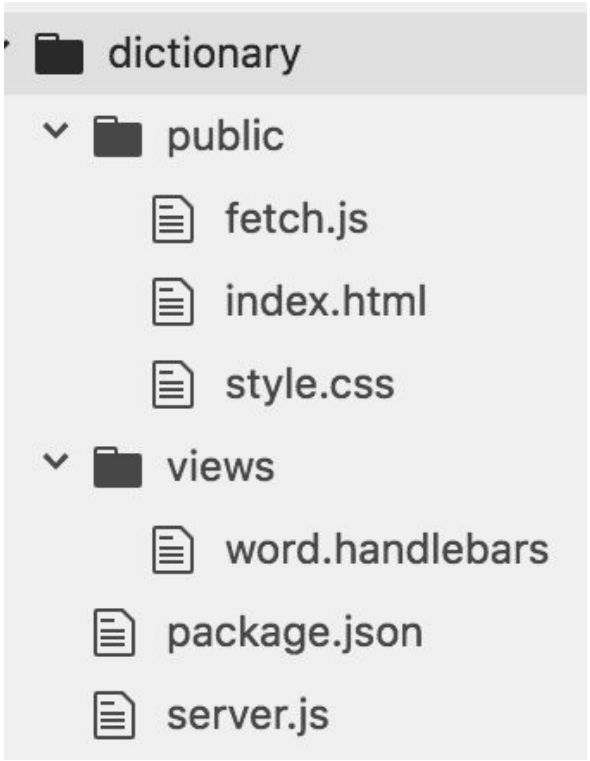
- The server is dynamically generating HTML files to return back to the client, but these HTML files **aren't** saved on the server; they are just passed directly to the client from memory.

Note on style

Right now, we are splitting our view code in two places:

- **index.html**: The file that gets loaded when you go to localhost:3000
- **word.handlebars**: The HTML template for word pages

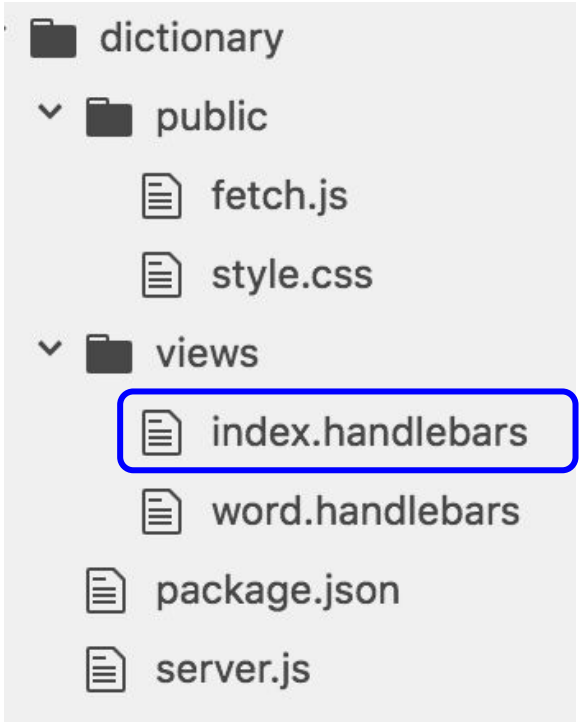
Style-wise, if you use a template engine, you usually serve *all* your view files via the template engine.



Note on style

We can make index.html into a Handlebars "template" that does not have any parameters:

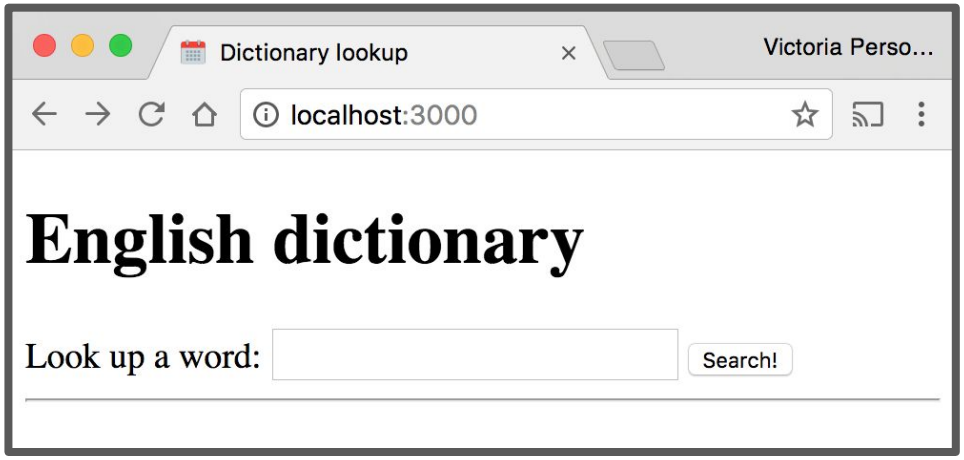
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Dictionary lookup</title>
    <link rel="stylesheet" href="style.css">
    <script src="fetch.js" defer></script>
  </head>
  <body>
    <h1>English dictionary</h1>

    <form id="search">
      Look up a word: <input type="text" id="word-input"/>
      <input type="submit" value="Search!">
    </form>

    <hr />

    <div id="results" class="hidden">
      The definition of <a href="" id="word"></a> is:
      <blockquote id="definition"></blockquote>
      <hr />
    </div>

  </body>
</html>
```



Note on style

Then in server.js, we will render the "index" template when there is a GET request to localhost:3000:

```
function onViewIndex(req, res) {
  res.render('index');
}
app.get('/', onViewIndex);
```



Completed example

Completed example code:

- dictionary-server-side
- See run instructions

Modules and Routes

Routes

So far, our server routes have all been defined in one file.

Right now, server.js:

- Starts the server
- Sets the template engine
- Serves the public/ directory
- Defines the JSON-returning routes
- Defines the HTML-returning routes

As our server grows, it'd be nice to split up server.js into separate files.

```
const express = require('express');
const MongoClient = require('mongodb').MongoClient;

const exphbs  = require('express-handlebars');

const app = express();
const hbs = exphbs.create();
app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

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

const DATABASE_NAME = 'eng-dict2';
const MONGO_URL = `mongodb://localhost:27017/${DATABASE_NAME}`;

let db = null;
let collection = null;

async function startServer() {
  // Set the db and collection variables before starting the server.
  db = await MongoClient.connect(MONGO_URL);
  collection = db.collection('words');
  // Now every route can safely use the db and collection objects.
  await app.listen(3000);
  console.log('Listening on port 3000');
}
startServer();

////////////////////////////////////////////////////////////////////////////////

// JSON-returning route

async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);

  const response = {
    word: word,
    definition: result ? result.definition : ''
  };
  res.json(response);
}
app.get('/lookup/:word', onLookupWord);

////////////////////////////////////////////////////////////////////////////////

// HTML-returning route

async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  const placeholders = {
    word: word,
    definition: definition
  };
  res.render('word', placeholders);
}
app.get('/:word', onViewWord);

function onViewIndex(req, res) {
  res.render('index');
}
app.get('/', onViewIndex);

```

NodeJS modules

NodeJS allows you to load external files, or "**modules**", via `require()`. We've already loaded three types of modules:

- Core NodeJS modules, e.g. `require('http')`
- External NodeJS modules downloaded via npm, e.g. `require('express')`
- A JSON file, e.g. `require('./dictionary.json')`

We will now see how to define **our own NodeJS modules** and include them in other JavaScript files via the `require()` statement.

NodeJS modules

A NodeJS module is just a JavaScript file.

- One module = One file
- There can only be one module per file

Let's say that you define the following JavaScript file:

silly-module.js

```
1  // This is a *poor* style and a bad example of a module;
2  // you should NOT write modules like this.
3
4  // This runs immediately, as soon as it is included.
5  console.log('hello');
6
```

NodeJS modules

You can include it in another JavaScript file by using the require statement:

scripts.js

```
1  require('./silly-module.js');
2
```

- Note that you **MUST** specify "./", "../", "/", etc.
- Otherwise NodeJS will look for it in the node_modules/ directory. See require() resolution rules

NodeJS modules

silly-module.js

```
1  // This is a *poor* style and a bad example of a module;
2  // you should NOT write modules like this.
3
4  // This runs immediately, as soon as it is included.
5  console.log('hello');
6
```

scripts.js

```
1  require('./silly-module.js');
```

```
$ node scripts.js
hello
```

The NodeJS file executes immediately when require()d.

Private variables

Everything declared in a module is **private to that module** by default.

Let's say that you define the following JavaScript file:

broken-module.js

```
1  // This is a private variable that is not shared.
2  let helloCounter = 0;
3
4  // This is a private function that is not shared.
5  function printHello() {
6    helloCounter++;
7    console.log('hello');
8  }
```

Private variables

If we include it and try to run `printHello` or access `helloCounter`, it will not work:

scripts.js

```
1  require('./broken-module.js');
2  printHello();
```

```
$ node scripts.js
/.../scripts.js:2
printHello();
^

ReferenceError: printHello is not defined
    at Object.<anonymous>
```

Private variables

If we include it and try to run `printHello` or access `helloCounter`, it will not work:

scripts.js

```
1  require('./broken-module.js');
2  console.log(helloCounter);
```

```
$ node scripts.js
scripts.js:2
console.log(helloCounter);
            ^

ReferenceError: helloCounter is not defined
    at Object.<anonymous>
```

`module.exports`

- `module` is a special object automatically defined in each NodeJS file, representing the current module.
- When you call `require('./fileName.js')`, the `require()` function will return the value of **`module.exports`** as defined in fileName.js
  - `module.exports` is initialized to an empty object.

broken-module.js

```
1  // This is a private variable that is not shared.
2  let helloCounter = 0;
3
4  // This is a private function that is not shared.
5  function printHello() {
6    helloCounter++;
7    console.log('hello');
8  }
```

scripts.js

```
1  const result = require('./broken-module.js');
2  console.log(result);
```

```
$ node scripts.js
{}
```

Prints an empty object because we didn't modify `module.exports` in broken-module.js.

string-module.js

```
1  // This is a pretty silly module as well.
2  module.exports = 'hello there';
3
```

scripts.js

```
1  const result = require('./string-module.js');
2  console.log(result);
3
```

```
$ node scripts.js
hello there
```

- Prints "hello there", because we set `module.exports` to "hello there" in string-module.js.
- The value of "result" is the value of "`module.exports`" in string-module.js.

function-module.js

```
1  function printHello() {
2    console.log('hello');
3  }
4  module.exports = printHello;
5
```

scripts.js

```
1  const result = require('./function-module.js');
2  console.log(result);
3  result();
```

```
$ node scripts.js
[Function: printHello]
hello
```

- We can export a function by setting it to `module.exports`

print-util.js

```
1  function printHello() {
2    console.log('hello');
3  }
4
5  function greet(name) {
6    console.log(`hello, ${name}`);
7  }
8
9  module.exports.printHello = printHello;
10 module.exports.greet = greet;
11
```

scripts.js

```
1  const printUtil = require('./print-util.js');
2  printUtil.printHello();
3  printUtil.greet('world');
4  printUtil.greet("it's me");
```

```
$ node scripts.js
hello
hello, world
hello, it's me
```

- We can export multiple functions by setting fields of the `module.exports` object

print-util.js

```
1  function printHello() {
2    console.log('hello');
3  }
4
5  function greet(name) {
6    console.log(`hello, ${name}`);
7  }
8
9  module.exports.printHello = printHello;
10 module.exports.greet = greet;
11
```

scripts.js

```
1  const printUtil = require('./print-util.js');
2  printUtil.printHello();
3  printUtil.greet('world');
4  printUtil.greet("it's me");
```

```
$ node scripts.js
hello
hello, world
hello, it's me
```

- We can export multiple functions by setting fields of the `module.exports` object

print-util.js

```
1  let i = 0;
2  function printCount() {
3    i++;
4    console.log(`count is now ${i}`);
5  }
6  module.exports.printCount = printCount;
7
```

scripts.js

```
1  const printUtil = require('./print-util.js');
2  printUtil.printCount();
3  printUtil.printCount();
4  printUtil.printCount();
```

```
$ node scripts.js
count is now 1
count is now 2
count is now 3
```

- You can create private variables and fields by not exporting them.

Simple module examples

Module example code is here:

- simple-modules
- Run instructions

NodeJS Module documentation:

- https://nodejs.org/api/modules.html

Back to Routes

Routes

So far, our server routes have all been defined in one file.

Right now, server.js:

- Starts the server
- Sets the template engine
- Serves the public/ directory
- Defines the JSON-returning routes
- Defines the HTML-returning routes

As our server grows, it'd be nice to split up server.js into separate files.

```
const express = require('express');
const MongoClient = require('mongodb').MongoClient;

const exphbs  = require('express-handlebars');

const app = express();
const hbs = exphbs.create();
app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

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

const DATABASE_NAME = 'eng-dict2';
const MONGO_URL = `mongodb://localhost:27017/${DATABASE_NAME}`;

let db = null;
let collection = null;

async function startServer() {
  // Set the db and collection variables before starting the server.
  db = await MongoClient.connect(MONGO_URL);
  collection = db.collection('words');
  // Now every route can safely use the db and collection objects.
  await app.listen(3000);
  console.log('Listening on port 3000');
}
startServer();

////////////////////////////////////////////////////////////////////////////////

// JSON-returning route

async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);

  const response = {
    word: word,
    definition: result ? result.definition : ''
  };
  res.json(response);
}
app.get('/lookup/:word', onLookupWord);

////////////////////////////////////////////////////////////////////////////////

// HTML-returning route

async function onViewWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  const placeholders = {
    word: word,
    definition: definition
  };
  res.render('word', placeholders);
}
app.get('/:word', onViewWord);

function onViewIndex(req, res) {
  res.render('index');
}
app.get('/', onViewIndex);

```

Goal: HTML vs JSON routes

Let's try to split server.js into 3 files.

Right now, server.js does the following:

- **Starts the server**
- **Sets the template engine**
- **Serves the public/ directory**
- **Defines the JSON-returning routes**
- **Defines the HTML-returning routes**

⟶ We'll continue to use **server.js** for the logic in blue

⟶ We'll try to move JSON routes to **api.js**

⟶ We'll try to move the HTML routes to **view.js**

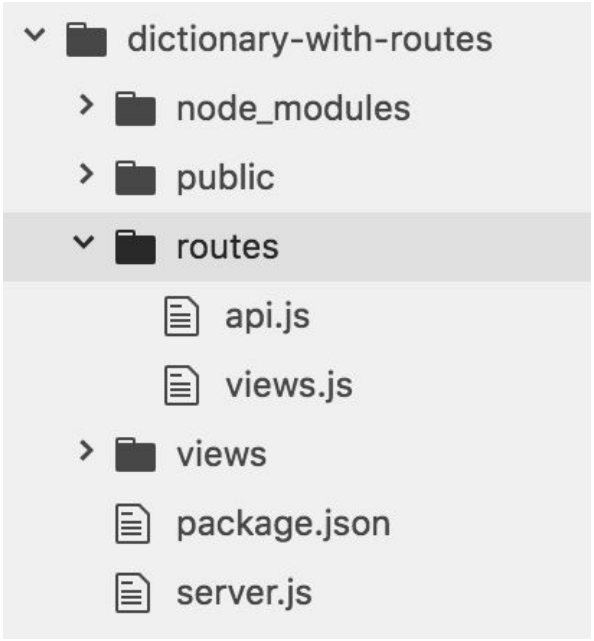
Goal: HTML vs JSON routes

→ We'll continue to use **server.js** for the logic in blue

→ We'll try to move JSON routes to **api.js**

→ We'll try to move the HTML routes to **view.js**

Desired directory structure:



Desired: server.js

```
const express = require('express');
const MongoClient = require('mongodb').MongoClient;
const exphbs  = require('express-handlebars');

const app = express();
const hbs = exphbs.create();
app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

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

const DATABASE_NAME = 'eng-dict2';
const MONGO_URL = `mongodb://localhost:27017/${DATABASE_NAME}`;

let db = null;
let collection = null;

async function startServer() {
  // Set the db and collection variables before starting the server.
  db = await MongoClient.connect(MONGO_URL);
  collection = db.collection('words');
  // Now every route can safely use the db and collection objects.
  await app.listen(3000);
  console.log('Listening on port 3000');
}
startServer();
```

We'd like to keep all set-up stuff in server.js...

Desired api.js (DOESN'T WORK)

And we'd like to be able to define the /lookup/:word route in a different file, something like the following:

```
async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);

  const response = {
    word: word,
    definition: result ? result.definition : ''
  };
  res.json(response);
}
app.get('/lookup/:word', onLookupWord);
```

Q: How do we define routes in a different file?

Router

Express lets you create `Router` objects, on which you can define modular routes:

api.js

```
1  const express = require('express');
2  const router = express.Router();
3
4  async function onLookupWord(req, res) {
5    ...
6  }
7  router.get('/lookup/:word', onLookupWord);
8
9  module.exports = router;
10
```

Router

```
1  const express = require('express');
2  const router = express.Router();
3
4  async function onLookupWord(req, res) {
5    ...
6  }
7  router.get('/lookup/:word', onLookupWord);
8
9  module.exports = router;
10
```

- Create a new `Router` by calling `express.Router()`
- Set routes the same way you'd set them on App
- Export the router via `module.exports`

Using the Router

Now we include the router by:

- Importing our router module via `require()`
- Calling `app.use(router)` on the imported router

```
const api = require('./routes/api.js');
const app = express();
app.use(api);
```

Now the app will also use the routes defined in routes/api.js!

However, **we have a bug** in our code...

MongoDB variables

We need to access the MongoDB collection in our route...

```
const express = require('express');
const router = express.Router();

async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);

  const response = {
    word: word,
    definition: result ? result.definition : ''
  };
  res.json(response);
}
router.get('/lookup/:word', onLookupWord);

module.exports = router;
```

MongoDB variables

...Which used to be defined as a global variable in server.js.

Q: What's the right way to access the database data?

```
let db = null;
let collection = null;

async function startServer() {
  // Set the db and collection variables before
  db = await MongoClient.connect(MONGO_URL);
  collection = db.collection('words');
  // Now every route can safely use the db and
  await app.listen(3000);
  console.log('Listening on port 3000');
}
startServer();
```

Middleware

In Express, you define middleware functions that get called certain requests, depending on how they are defined.

The app.METHOD routes we have been writing are actually middleware functions:

```
function onViewIndex(req, res) {
  res.render('index');
}
app.get('/', onViewIndex);
```

onViewIndex is a middleware function that gets called every time there is a GET request for the "/" path.

Middleware: `app.use()`

We can also define middleware functions using `app.use()`:

```
// Middleware function that prints a message for every request.
function printMessage(req, res, next) {
  console.log('request to server!');
  next();
}
app.use(printMessage);
```

Middleware functions receive 3 parameters:

- `req` and `res`, same as in other routes
- **`next`**: Function parameter. Calling this function invokes the next middleware function in the app.
  - If we resolve the request via `res.send`, `res.json`, etc, we don't have to call `next()`

Middleware: app.use()

We can write middleware that defines new fields on each request:

```
const db = await MongoClient.connect(MONGO_URL);
const collection = db.collection('words');

// Adds the "words" collection to every MongoDB request.
function setCollection(req, res, next) {
  req.collection = collection;
  next();
}
app.use(setCollection);
```

Middleware: app.use()

Now if we load this middleware on each request:

```
async function startServer() {
  const db = await MongoClient.connect(MONGO_URL);
  const collection = db.collection('words');

  // Adds the "words" collection to every MongoDB request.
  function setCollection(req, res, next) {
    req.collection = collection;
    next();
  }
  app.use(setCollection);
  app.use(api);

  await app.listen(3000);
  console.log('Listening on port 3000');
}
```

Middleware: app.use()

Now if we load this middleware on each request:

```
async function startServer() {
  const db = await MongoClient.connect(MONGO_URL);
  const collection = db.collection('words');

  // Adds the "words" collection to every MongoDB request.
  function setCollection(req, res, next) {
    req.collection = collection;
    next();
  }
  app.use(setCollection);
  app.use(api);

  await app.listen(3000);
  console.log('Listening on port 3000');
}
```

Note that we need to use the api router **AFTER** the middleware

Middleware: app.use()

Then we can access the collection via `req.collection`:

```
async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await req.collection.findOne(query);

  const response = {
    word: word,
    definition: result ? result.definition : ''
  };
  res.json(response);
}
router.get('/lookup/:word', onLookupWord);
```

Middleware: app.use()

Then we can access the collection via `req.collection`:

```
async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  const query = { word: word.toLowerCase() };
  const result = await req.collection.findOne(query);

  const response = {
    word: word,
    definition: result ? result.definition : ''
  };
  res.json(response);
}
router.get('/lookup/:word', onLookupWord);
```

Views router

We can similarly move the HTML-serving logic to views.js and `require()` the module in server.js:

```
const api = require('./routes/api.js');
const views = require('./routes/views.js');

app.use(setCollection);
app.use(api);
app.use(views);
```

Views router

```
const express = require('express');
const router = express.Router();

async function onViewWord(req, res) {
  ...
  res.render('word', placeholders);
}
router.get('/:word', onViewWord);

function onViewIndex(req, res) {
  res.render('index');
}
router.get('/', onViewIndex);

module.exports = router;
```

Routes and middleware

Simple middleware example code is here:

- simple-middleware
- Run instructions

Dictionary with routes example code here:

- dictionary-with-routes
- Run instructions

Express documentation:

- Router
- Writing / Using Middleware

Recall: Web app architectures

Structuring a web app

There are roughly 4 strategies for architecting a web application:

1. **Server-side rendering:**
   Server sends a new HTML page for each unique path
2. **Single-page application:**
   Server sends the exact same web page for every unique path (and the page runs JS to change what it look like)
3. Combination of 1 and 2 ("**Isomorphic**" / "**Universal**")
4. **Progressive Loading**

Structuring a web app

There are roughly 4 strategies for architecting a web application:

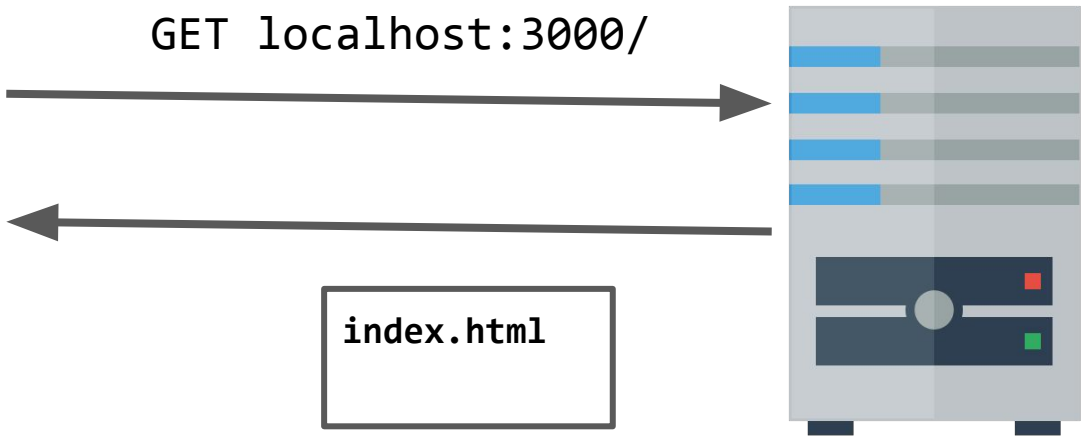
1. **Server-side rendering:**
   Server sends a new HTML page for each unique path
2. **Single-page application:**
   Server sends the exact same web page for every unique path (and the page runs JS to change what it look like)

   → Let's talk about this one now

Single-page web app

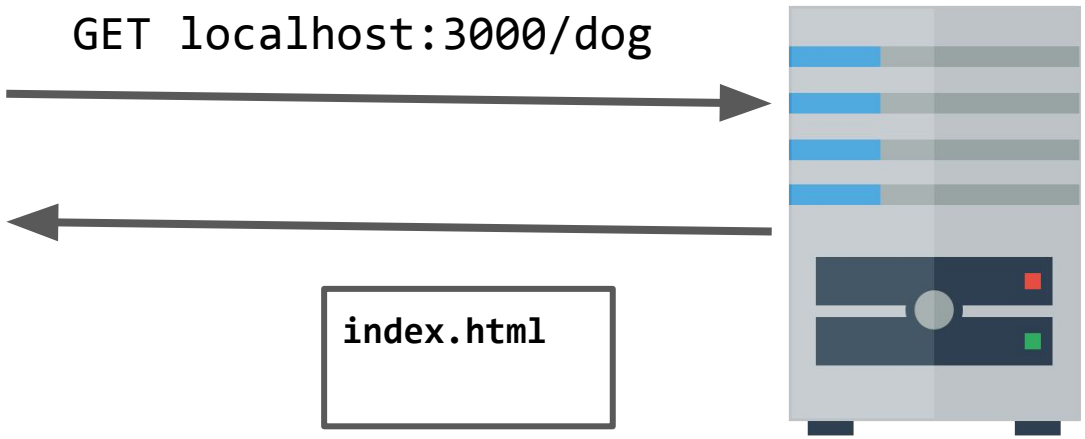
Single page web app

- The server always sends the **same one HTML file** for all requests to the web server.
- The server is configured so that requests to /**<word>** would still return e.g. index.html.
- The client JavaScript parses the URL to get the route parameters and initialize the app.



Single page web app

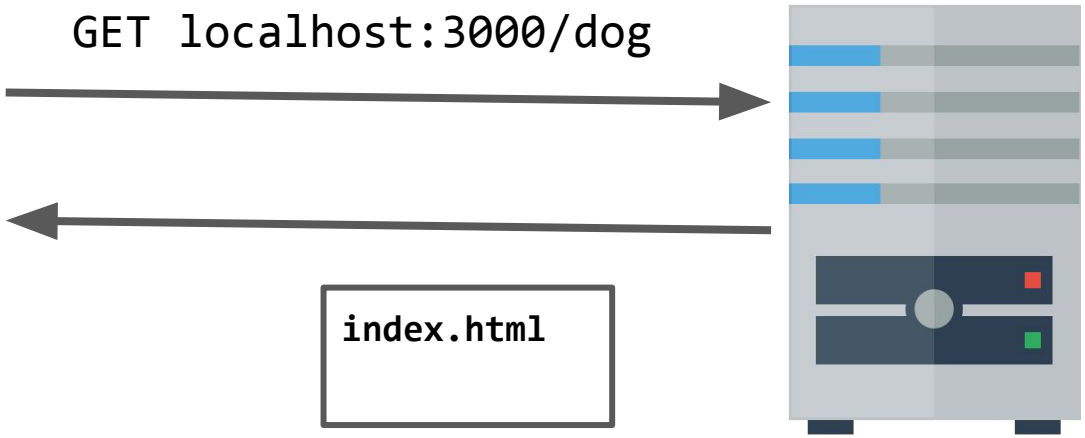
- The server always sends the **same one HTML file** for all requests to the web server.
- The server is configured so that requests to /**<word>** would still return e.g. index.html.
- The client JavaScript parses the URL to get the route parameters and initialize the app.



Single page web app

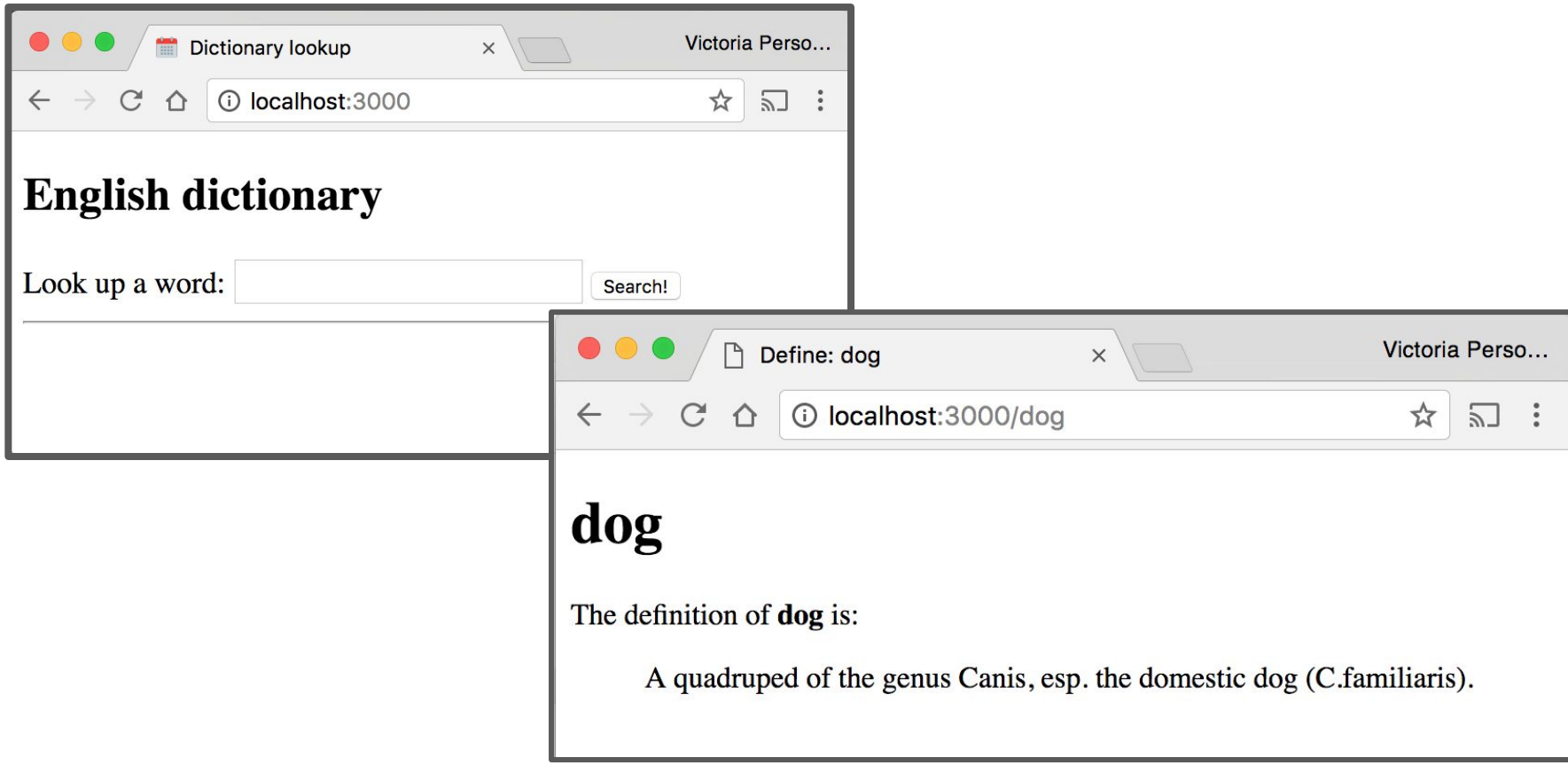
Another way to think of it:

- You embed **all your views** into index.html
- You use JavaScript to switch between the views
- You configure JSON routes for your server to handle sending and retrieving data



Dictionary example

Let's write our dictionary example as a single-page web app.



Recall: Handlebars

For our multi-page dictionary app, we had two handlebars files: index.handlebars and word.handlebars

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Dictionary lookup</title>
    <link rel="stylesheet" href="style.css">
    <script src="fetch.js" defer></script>
  </head>
  <body>
    <h1>English dictionary</h1>

    <form id="search">
      Look up a word: <input type="text" id="word-input"/>
      <input type="submit" value="Search!">
    </form>

    <hr />

    <div id="results" class="hidden">
      The definition of <a href="" id="word"></a> is:
      <blockquote id="definition"></blockquote>
      <hr />
    </div>

  </body>
</html>
```

index.handlebars

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Define: {{ word }}</title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <h1>{{ word }}</h1>
    <div id="results" class="hidden">
      The definition of <strong id="word">{{ word }}</strong> is:
      <blockquote id="definition">{{ definition }}</blockquote>
    </div>
  </body>
</html>
```

word.handlebars

SPA

In a single-page web app, the HTML for both the Search page and the Word page are in index.html:

```
<!-- View for the search page -->
<section id="main-view" class="hidden">
  <h1>English dictionary</h1>

  <form id="search">
    Look up a word: <input type="text" id="word-input"/>
    <input type="submit" value="Search!">
  </form>

  <hr />

  <div id="results" class="hidden">
    The definition of <a href="" id="word"></a> is:
    <blockquote id="definition"></blockquote>
    <hr />
  </div>
</section>

<!-- View for a single word -->
<section id="word-view" class="hidden">
  <h1></h1>
  The definition of <strong id="wv-word"></strong> is:
  <blockquote id="wv-def"></blockquote>
</section>
```

Server-side routing

For all requests that are not JSON requests, we return "index.html"

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

async function onAllOtherPaths(req, res) {
  res.sendFile(path.resolve(__dirname, 'public', 'index.html'));
}
app.get('*', onAllOtherPaths);
```

Client-side parameters

All views are hidden at first by the client.

```
<!-- View for the search page -->
<section id="main-view" class="hidden">
  ...
</section>

<!-- View for a single word -->
<section id="word-view" class="hidden">
  ...
</section>
```

Client-side parameters

When the page loads, the client looks at the URL to decide what page it should display.

```
const urlPathString = window.location.pathname;
const parts = urlPathString.split('/');
if (parts.length > 1 && parts[1].length > 0) {
  const word = parts[1];
  this._showWordView(word);
} else {
  this._showSearchView();
}
```

Client-side parameters

To display the word view, the client makes a `fetch()` requests for the definition.

```
class WordView {
  constructor(containerElement, word) {
    this.containerElement = containerElement;
    this._onSearch(word);
  }

  async _onSearch(word) {
    const result = await fetch('/lookup/' + word);
    const json = await result.json();
```

Completed example

Completed example code:

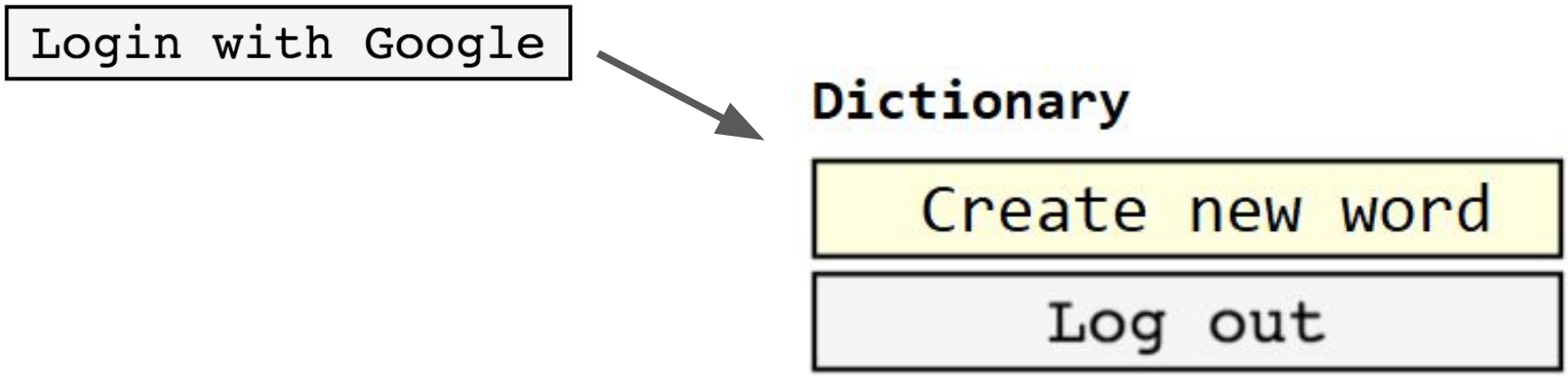
- dictionary-spa
- See run instructions

Authentication

Adding user login

What if you want to add user login to your web page?

- For example, what if we extended the dictionary app so that you had to log in before you could create a new word?



Authentication is hard

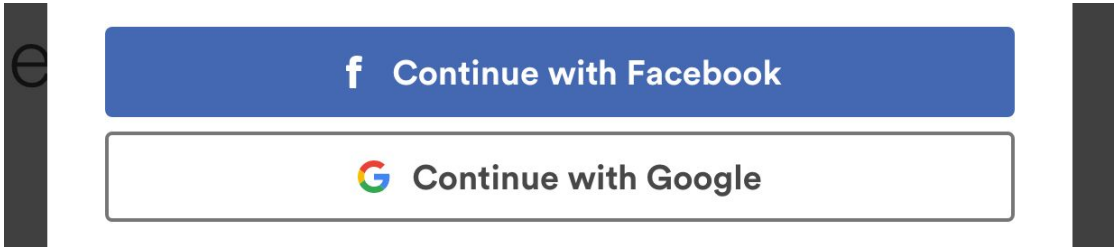
Trying to write your own login system is difficult:

- How are you going to save passwords securely?
- How do you help with forgotten passwords?
- How do you make sure users set a good password?
- Etc.

Luckily, **you don't have to build your own login**.

OAuth2

- OAuth2 is a standard for user authentication
- For users:
  - It allows a user to log into a website like AirBnB via some other service, like Gmail or Facebook
- For developers:
  - It lets you authenticate a user without having to implement log in
- Examples: "Log in with Facebook"



OAuth2 APIs

Companies like Google, Facebook, Twitter, and GitHub have OAuth2 APIs:

- Google Sign-in API
- Facebook Login API
- Twitter Login API
- GitHub Apps/Integrations

- OAuth2 is standardized, but the libraries that these companies provide are all different.
- You must read the documentation to understand how to connect via their API.

Using OAuth2

All OAuth2 libraries are going to be different, but they work like the following:

1. Get an API key
2. Whitelist the domains that can call your API key
3. Insert a `<script>` tag containing <company>'s API
4. In the **frontend** code:
    a. Use <company>'s API to create a login button
    b. When the user clicks the login button, you will get information like:
        i. Name, email, etc
        ii. Some sort of **Identity Token**

Aside: API keys

Generally you're not supposed to store API keys in your GitHub repo, even though we did in some lecture examples.

→ How are you supposed to store API keys?

API keys: Store in Env Vars

Generally you're not supposed to store API keys in your GitHub repo, even though we did in some lecture examples.

⟶ How are you supposed to store API keys?

⟶ Best practice: Use Environment Variables

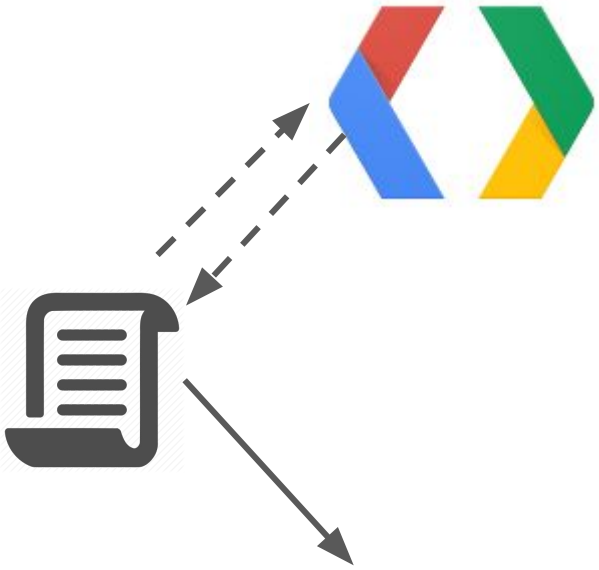
- Set the environment variable on your host, such as Heroku
- Can access the environment variable's value in NodeJS via `process.env.VAR_NAME`

Using OAuth2

You need to authenticate the identity of the client on the backend as well:

- In the **backend** code:
  - Use <company>'s libraries to verify the token from the client is a valid token

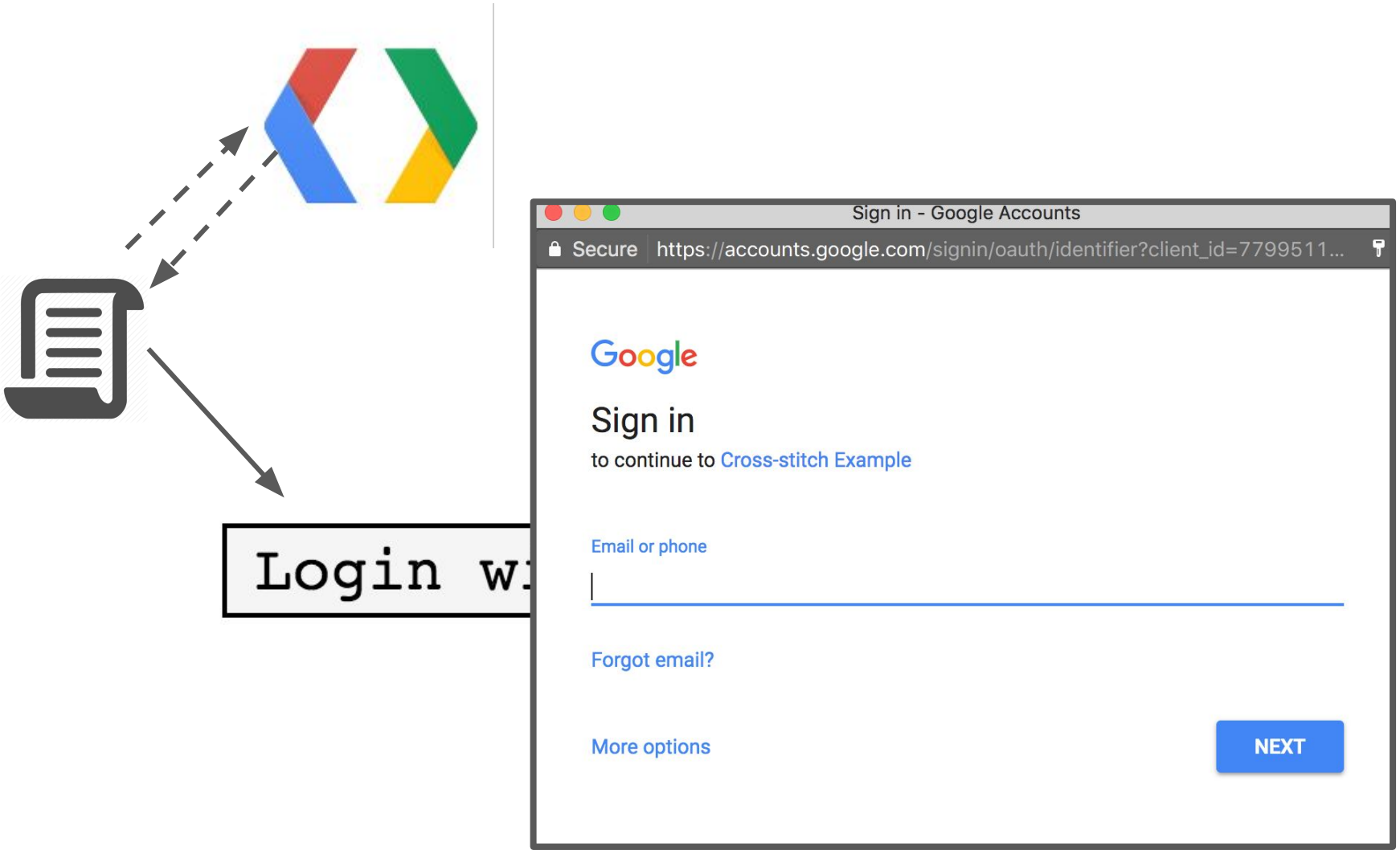
Using OAuth2: Frontend



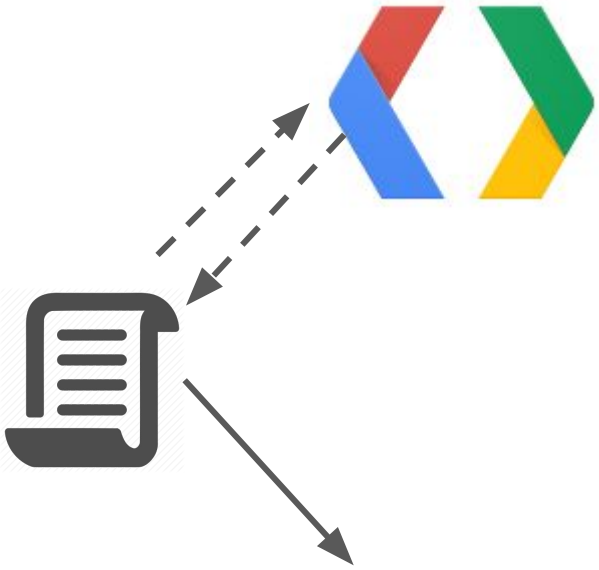
- Load the Google API by calling Google's library functions with the client id
- Add a button that, when clicked, prompts the user to log into Google

Login with Google

Using OAuth2: Frontend



Using OAuth2: Frontend



- When the user logs in, the login callback will fire with information about the user
  - Name, email, etc
  - Will also include an **IdentityToken**, which will expire after a certain amount of time

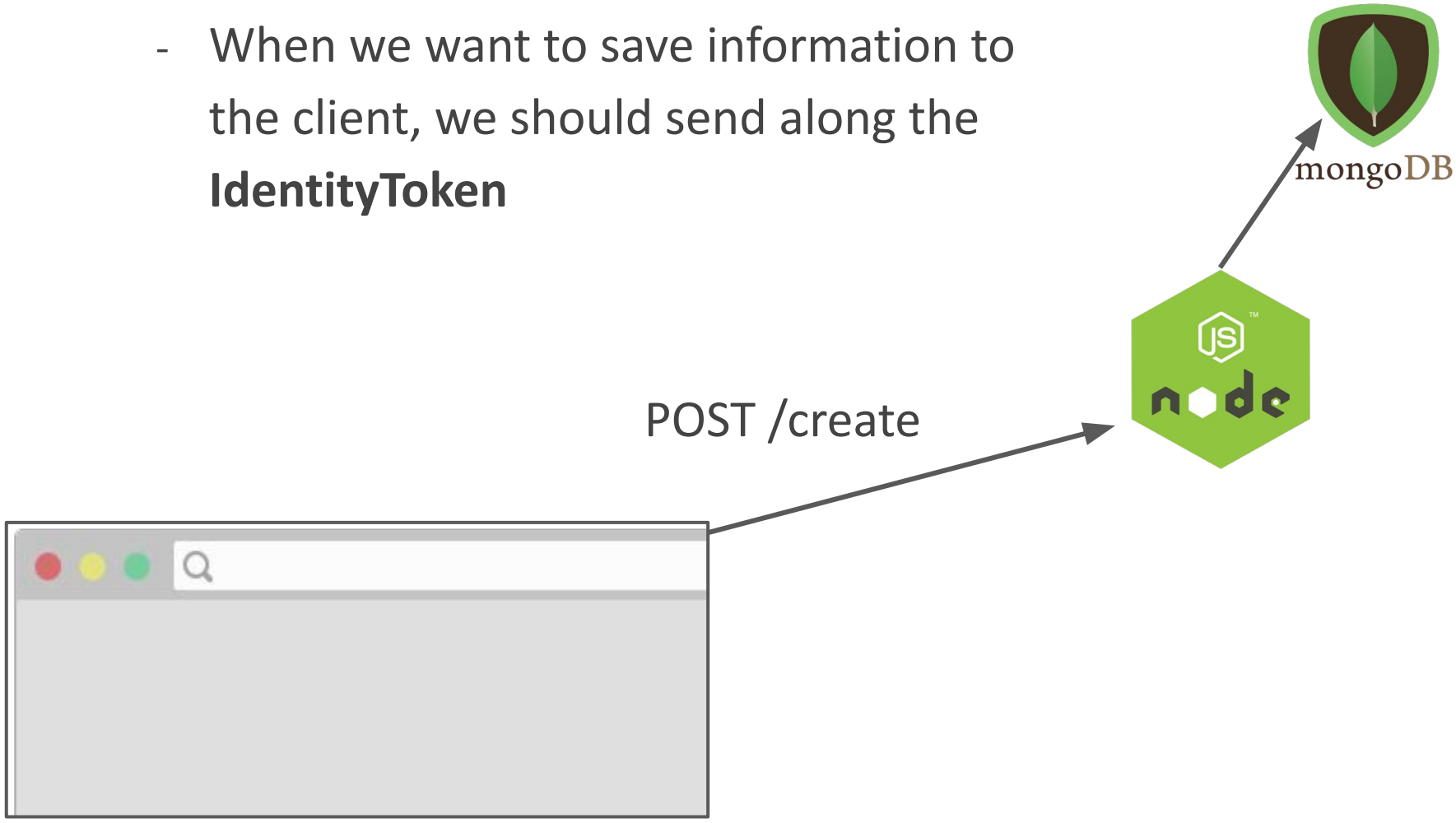
Dictionary

Create new word

Log out

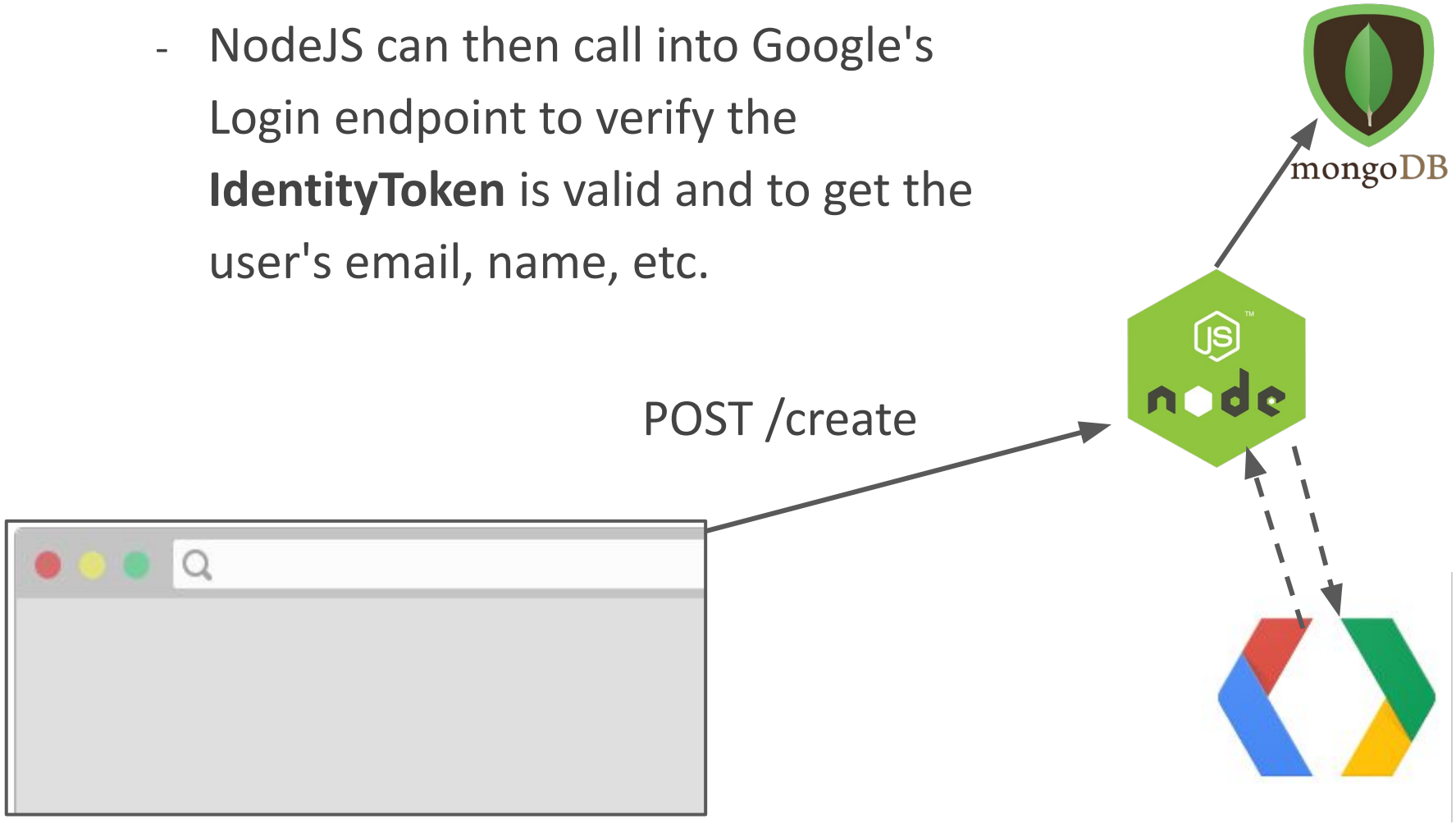
Using OAuth2: Backend

- When we want to save information to the client, we should send along the **IdentityToken**



Using OAuth2: Backend

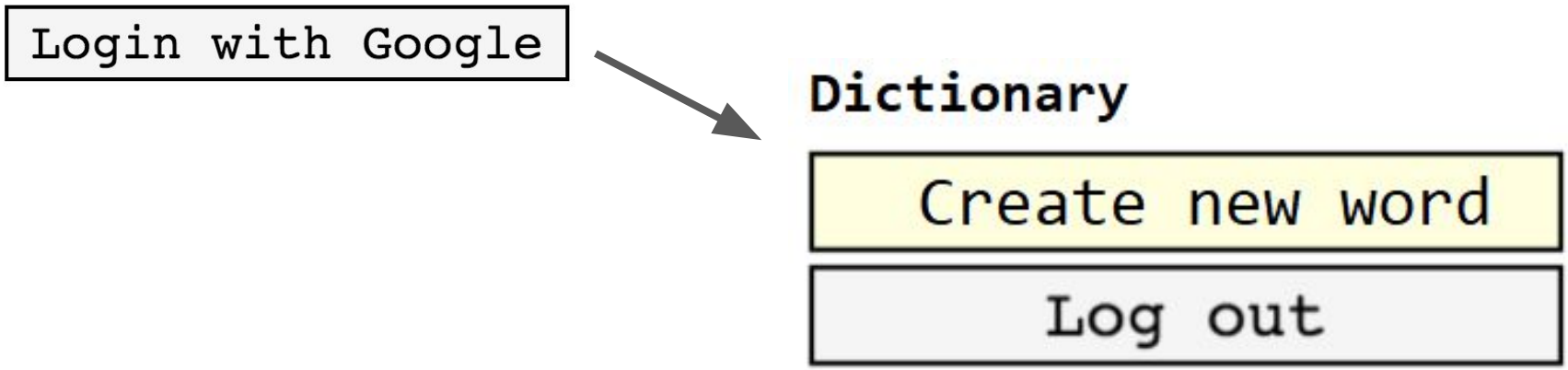
- NodeJS can then call into Google's Login endpoint to verify the **IdentityToken** is valid and to get the user's email, name, etc.



Adding user login

Adding user login to dictionary:

- Now we have **two collections:** Users and Words




```
const words = db.collection('words');
const users = db.collection('users');
```

Saving words

Every word now has an author associated with it:

```
async function onSetWord(req, res) {
  const idToken = req.body.idToken;
  const userInfo = await auth.validateToken(idToken);

  const userQuery = { email: userInfo.email };
  const userResponse = await req.users.findOne(userQuery);

  const routeParams = req.params;
  const word = routeParams.word.toLowerCase();
  const definition = req.body.definition;

  const query = { word: word };
  const newEntry = { word: word, definition: definition, authorId: ObjectID(userResponse._id) };
  const params = { upsert: true };
  const response = await req.collection.update(query, newEntry, params);

  res.json({ success: true, id: response._id });
}
router.post('/:word', onSetWord);
```

MongoDB database design

For more on MongoDB database design, MongoDB wrote a short, helpful blog series:

- 6 Rules of Thumb for MongoDB Schema Design:
  - Part 1: Basic modeling techniques
  - Part 2: Referencing
  - Part 3: Design recommendations

For *a lot* more on database design, take a database class!