Schedule

Today:

- Server-side rendering with Handlebars
- Node modules
- Routes & Middleware
- Single page application

- Introduction to ReactJS

Recall: Server-side rendering with Handlebars

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**

(Short take on these)

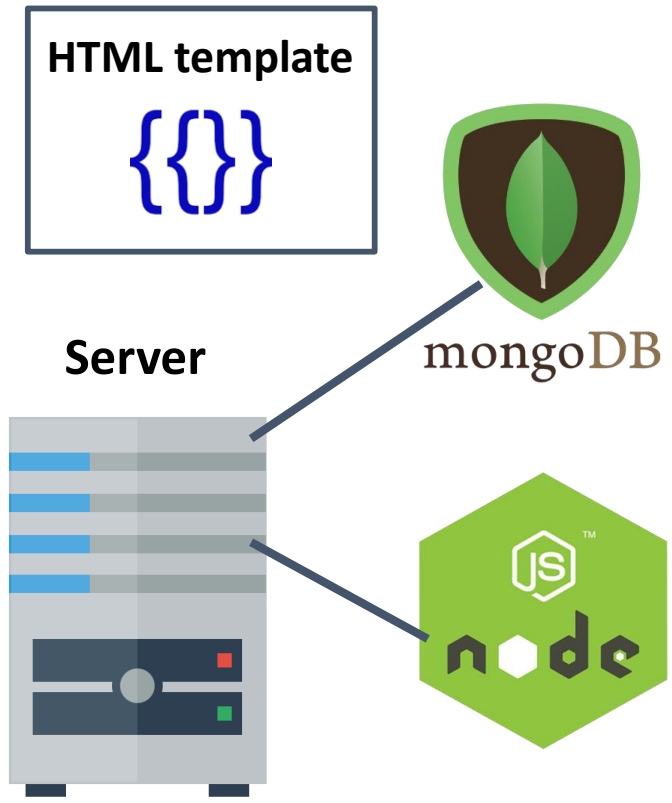
1. **Server-side rendering:**
   - We will show you how to do this
2. **Single-page application:**
   - We will show you how to do this
3. Combination of 1 and 2 ("**Isomorphic**" / "**Universal**")
   - This is probably the most common technique
   - We will talk about this but won't show you how to code it
4. **Progressive Loading**
   - This is probably the future (but it's complex)
   - We will talk about this but won't show you how to code it

Server-side rendering

Server-side rendering

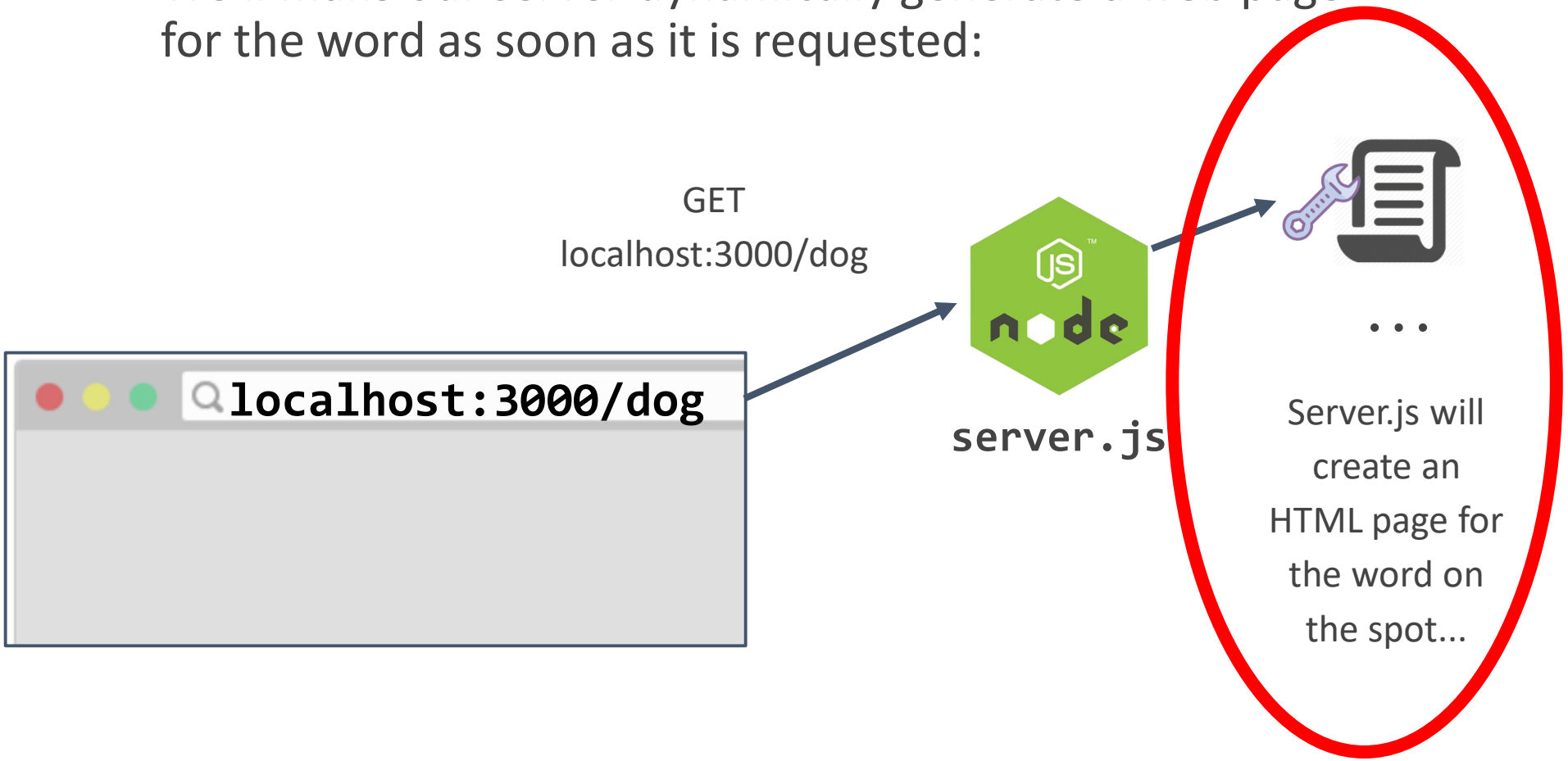
Multi-page web app:

- The server generates a different web page
- Usually involves filling out and returning an HTML template in response to a request
  - This is done by a **templating engine**: Pug (Jade), EJS, Handlebars, etc



Dynamically generated pages

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



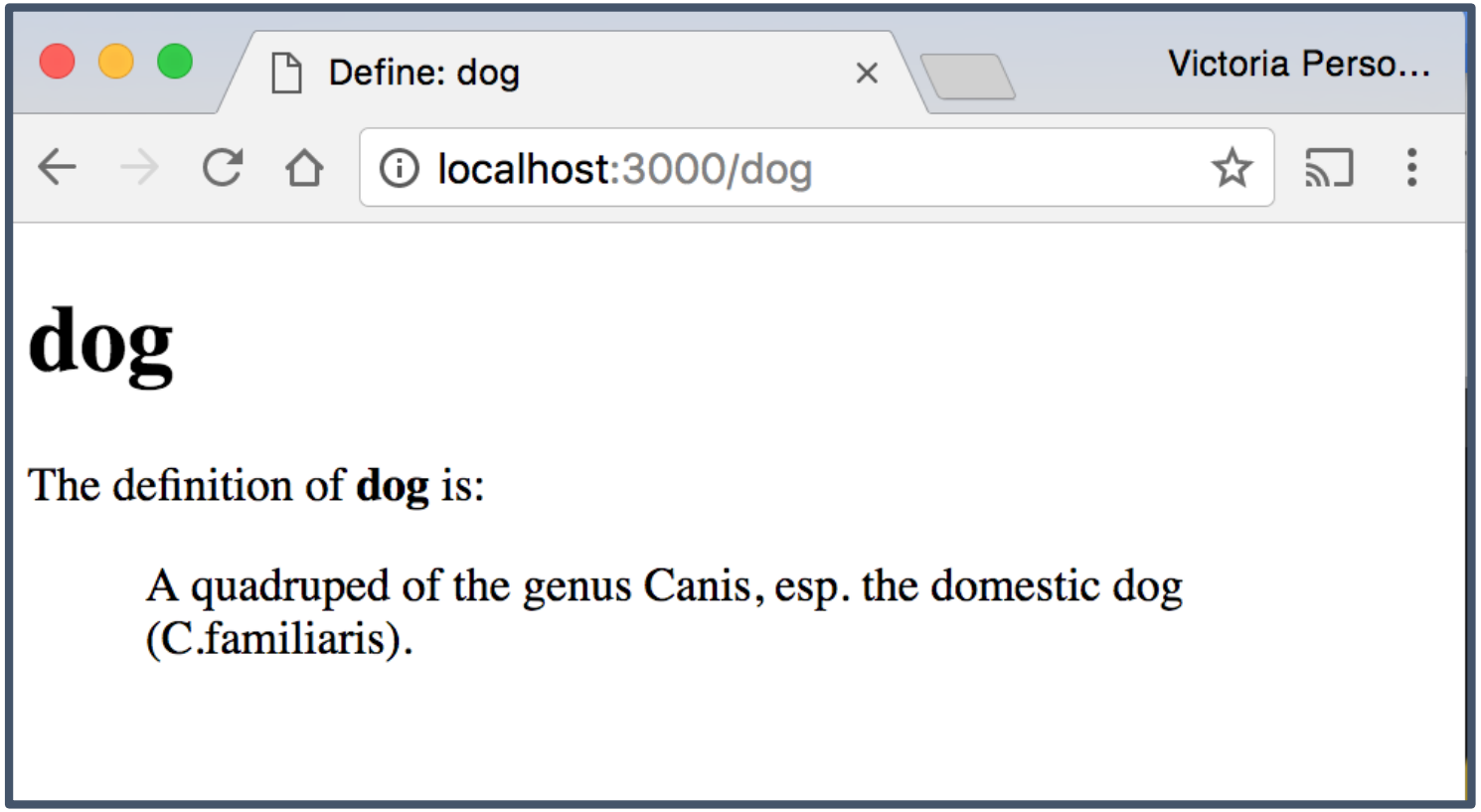
A solution: HTML Strings

We can make our HTML response:

```
  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:



HTML Strings

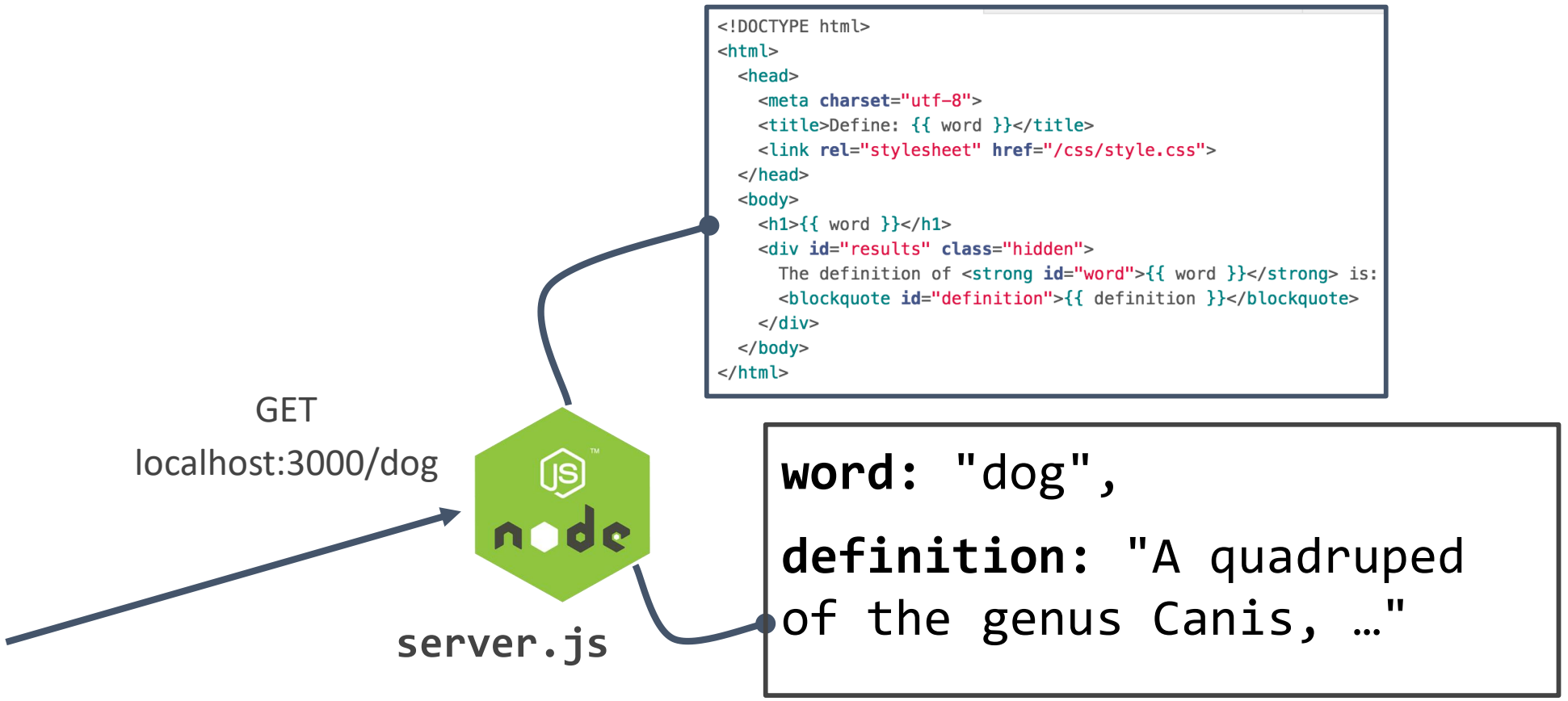
This works, but now we have a big HTML string in our server code:

```
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

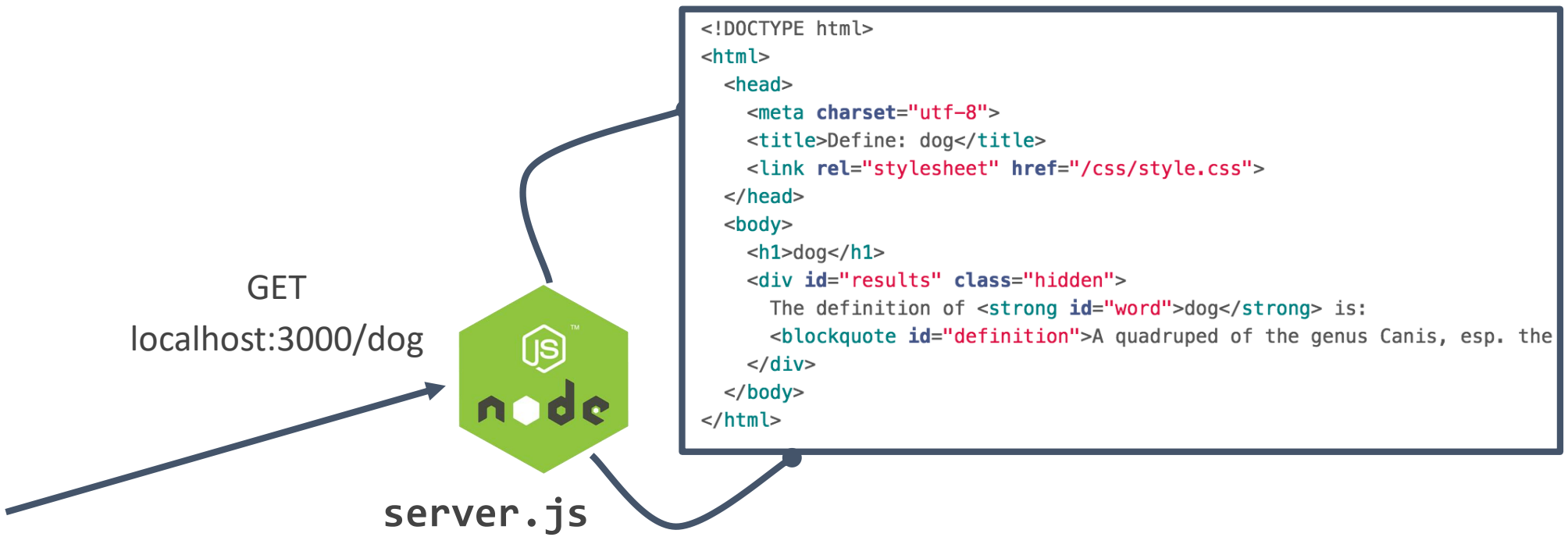
We want our NodeJS code to be able to take an HTML template, fill in its placeholder values, and return the completed page:




```
<!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>
```

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

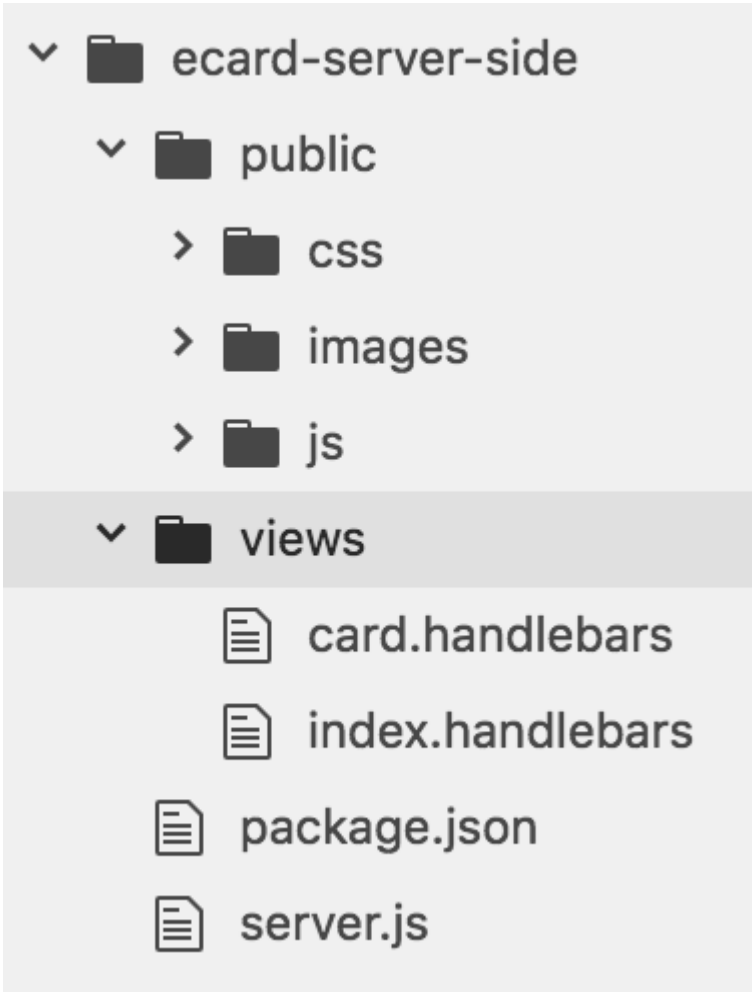
```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

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');
```

E-cards: Server-side rendering



We change our E-cards example to have 2 Handlebars templates:

- card.handlebars
- Index.handlebars

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

Note that there are no longer any HTML files in our `public/` folder.

index.handlebars

This is the same contents of index.html with no placeholders, since there were no placeholders needed:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>CS193x e-cards</title>
    <link rel="stylesheet" href="/css/card-style.css">
    <link rel="stylesheet" href="/css/creator-style.css">
    <script src="/js/creator-view.js" defer></script>
    <script src="/js/main.js" defer></script>
  </head>
  <body>
    <section class="main" id="creator-view">
      <h1>CS193x e-cards</h1>
      <h2>Preview</h2>
      <section id="card-view">
        <div id="card-image"></div>
        <div id="card-message"></div>
```

card.handlebars

But for the card-view, we want a different card style and message depending on the card:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>CS193x e-cards</title>
    <link rel="stylesheet" href="/css/card-style.css">
  </head>
  <body>
  <section class="main">
    <h1>CS193x e-cards</h1>
    <section id="card-view">
      <div id="card-image" class="{{ style }}"></div>
      <div id="card-message">{{ message }}</div>
    </section>
  </section>
  </body>
</html>
```

card.handlebars

But for the card-view, we want a different card style and message depending on the card:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>CS193x e-cards</title>
    <link rel="stylesheet" href="/css/card-style.css">
  </head>
  <body>
  <section class="main">
    <h1>CS193x e-cards</h1>
    <section id="card-view">
      <div id="card-image" class="{{ style }}"></div>
      <div id="card-message">{{ message }}</div>
    </section>
  </section>
  </body>
</html>
```

E-cards: Server-side rendering

Setting up NodeJS to use Handlebars and adding templates does nothing on its own. To use the templates, we need to call **`res.render`**:

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

`res.render`(***`viewName`***, ***`placeholderDefs`***)

- Returns the HTML stored in "`views/`***`viewName`***`.handlebars`" after replacing the placeholders, if they exist

E-cards: Server-side rendering

For retrieving the card, we have placeholders we need to fill in, so we define the placeholder values in the second parameter:

```
async function onGetCard(req, res) {
  const cardId = req.params.cardId;
  const collection = db.collection('card');
  const doc = await collection.findOne({ _id: ObjectID(cardId) });

  res.render('card', { message: doc.message, style: doc.style } );
}
app.get('/id/:cardId', onGetCard);
```

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

```
let i = 0;
function printCount() {
  i++;
  console.log(`count is now ${i}`);
}
module.exports.printCount = printCount;

```

scripts.js

```
const printUtil = require('./print-util.js');
printUtil.printCount();
printUtil.printCount();
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:

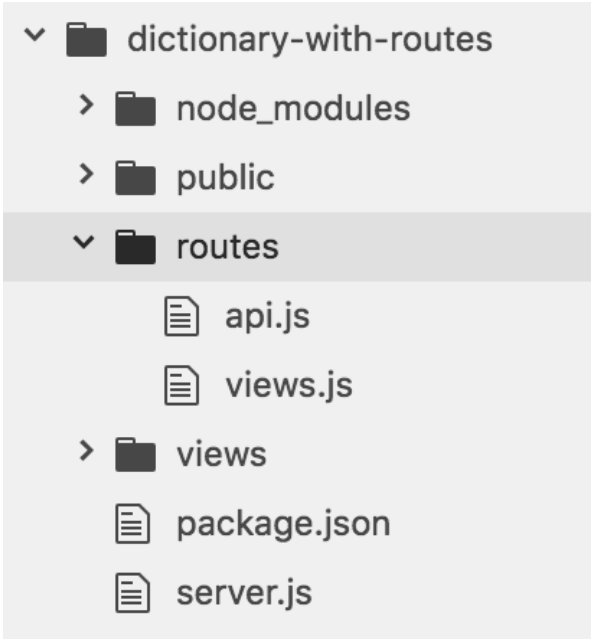
- **Starts the server**
- **Sets the template engine**
- **Serves the public/ directory**
- **Defines the JSON-returning routes**
- **Defines the HTML-returning routes**

→ 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

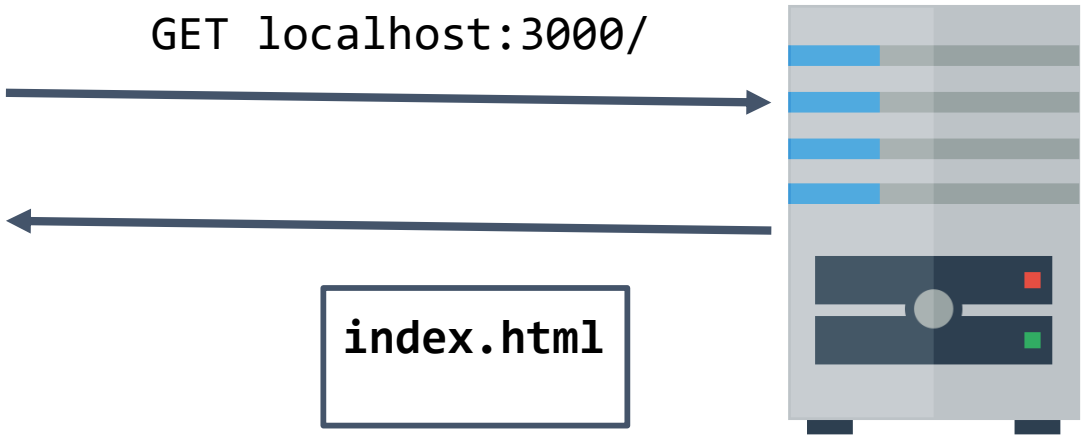
Express documentation:

- Router
- Writing / Using Middleware

Single-page web app

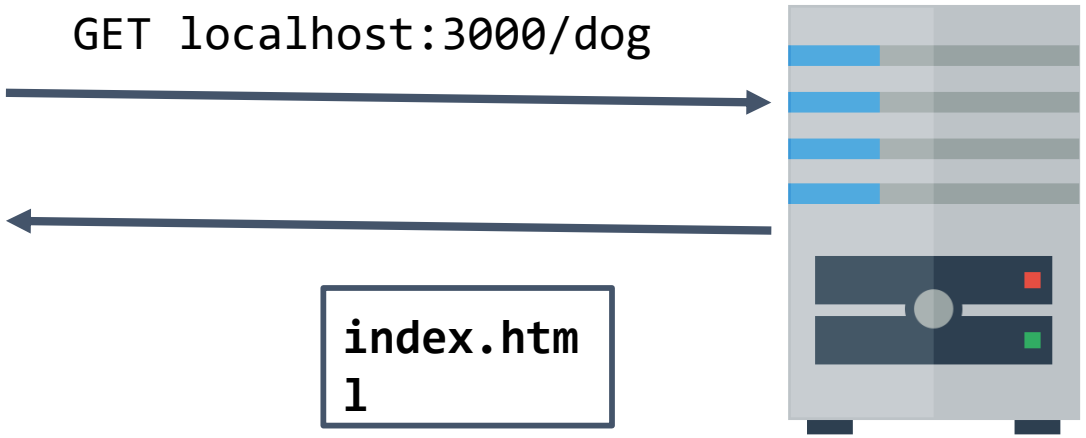
Single page web app

- 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

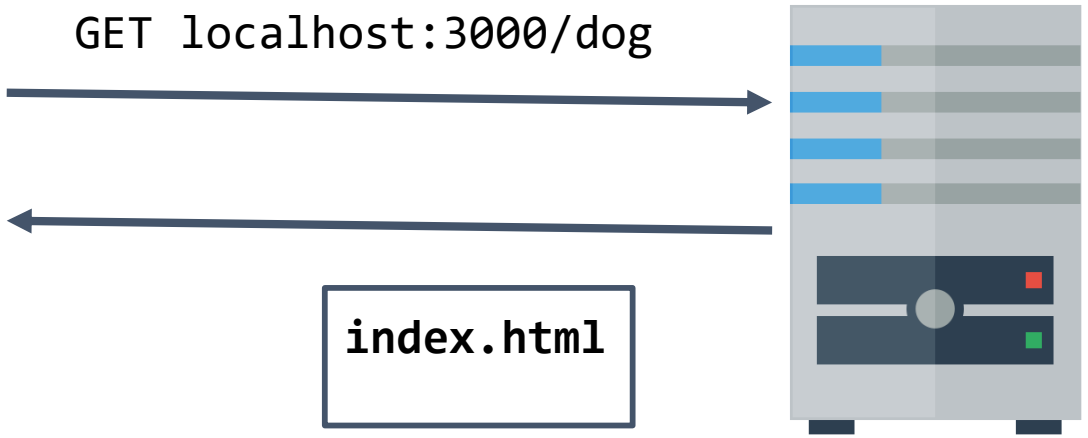
- The server always sends the **same one HTML file** for all requests to the web server.
- The server is configured so that requests to /**<word>** would still return e.g. index.html.
- The client JavaScript parses the URL to get the route parameters and initialize the app.



Single page web app

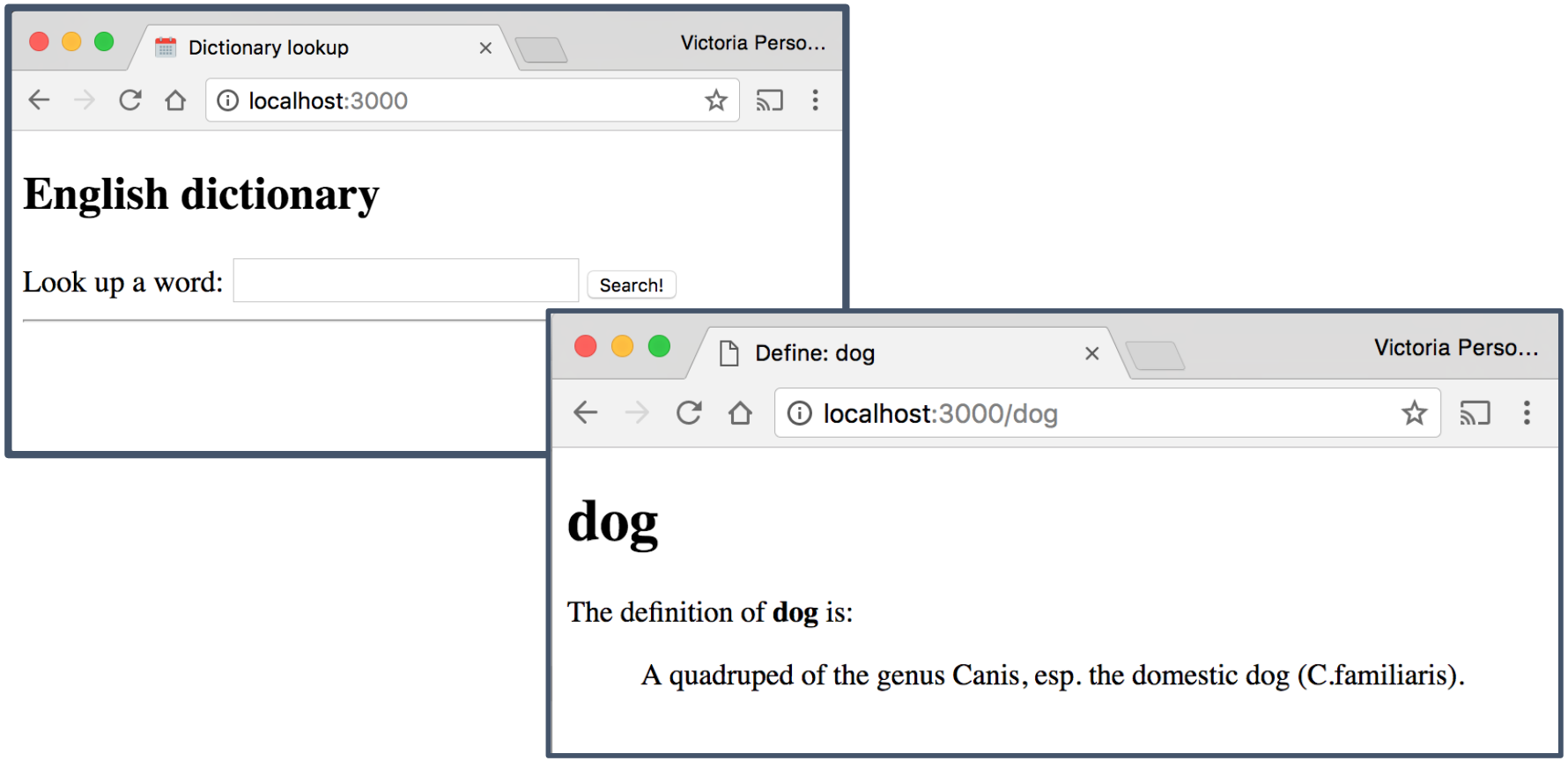
Another way to think of it:

- 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

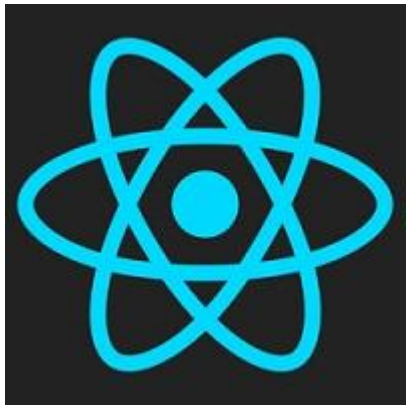
To display the word view, the client makes a `fetch()` requests for the definition.

```
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



Introduction to ReactJS

What is react

- React is a **JavaScript** library created by **Facebook**
- React is a **User Interface** (UI) library
- React is a tool for building **UI components**

```
import React from 'react';
import ReactDOM from 'react-dom';

class Test extends React.Component {
  render() {
    return <h1>Hello World!</h1>;
  }
}

ReactDOM.render(<Test />, document.getElementById('root'));
```

Why React?

Problems solved by react:

- DOM operations are quire expensive in terms of **performance**
- Page has data changes over time at high rates
  - Lots of people commenting on a post
  - Likes being generated…

→ require DOM to:

  - updates **very fast**,
  - reflect in other parts of UI if they use the same data

How React works?

React creates a VIRTUAL DOM in memory.

- Instead of manipulating the browser's DOM directly,

- React creates a virtual DOM in memory

→ does all the necessary manipulating

→ making the changes in the browser DOM.

React only changes what needs to be changed!

- React finds out what changes have been made, and changes **only** what needs to be changed.

- You will learn the various aspects of how React does this later.

Any others?

So why React?

– Opionated

Setting react

- Install create-react-app by running this command in your terminal:

```
C:\Users\Your Name>npm install -g create-react-app
```

- Then you are able to create a React application, let's create one called *myfirstreact*.

```
C:\Users\Your Name>npx create-react-app myfirstreact
```

Running react

- Move to the *myfirstreact* directory & run application

```
C:\Users\Your Name>cd myfirstreact
```

```
C:\Users\Your Name\myfirstreact>npm start
```