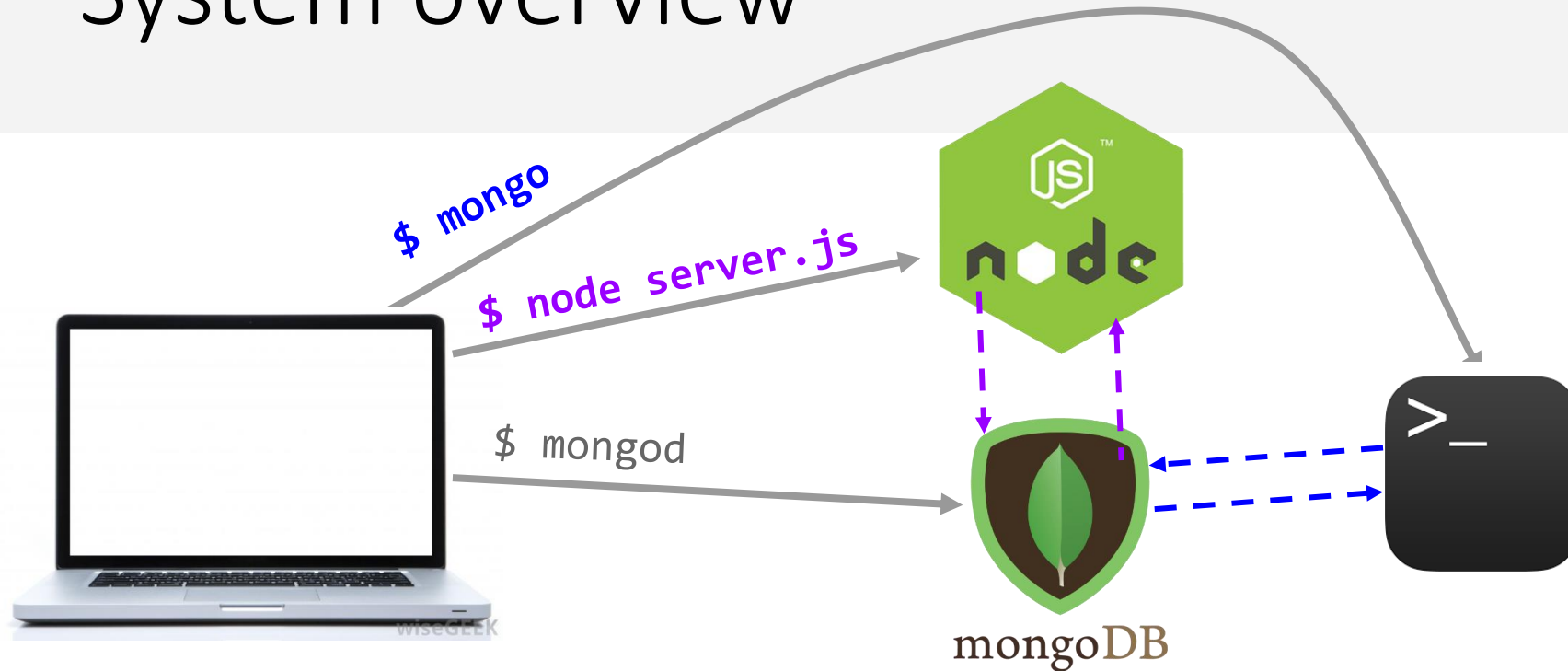


Schedule

Today:

- MongoDB: fetching data from MongoDB
- Selected topic: *package.json*
- Server-side rendering with Handlebars

System overview



We will be using two ways of communicating to the MongoDB server:

- NodeJS libraries
- mongo command-line tool

Recall: mongo command-line tool

MongoDB concepts

Database:

- A container of MongoDB **collections**

Collection:

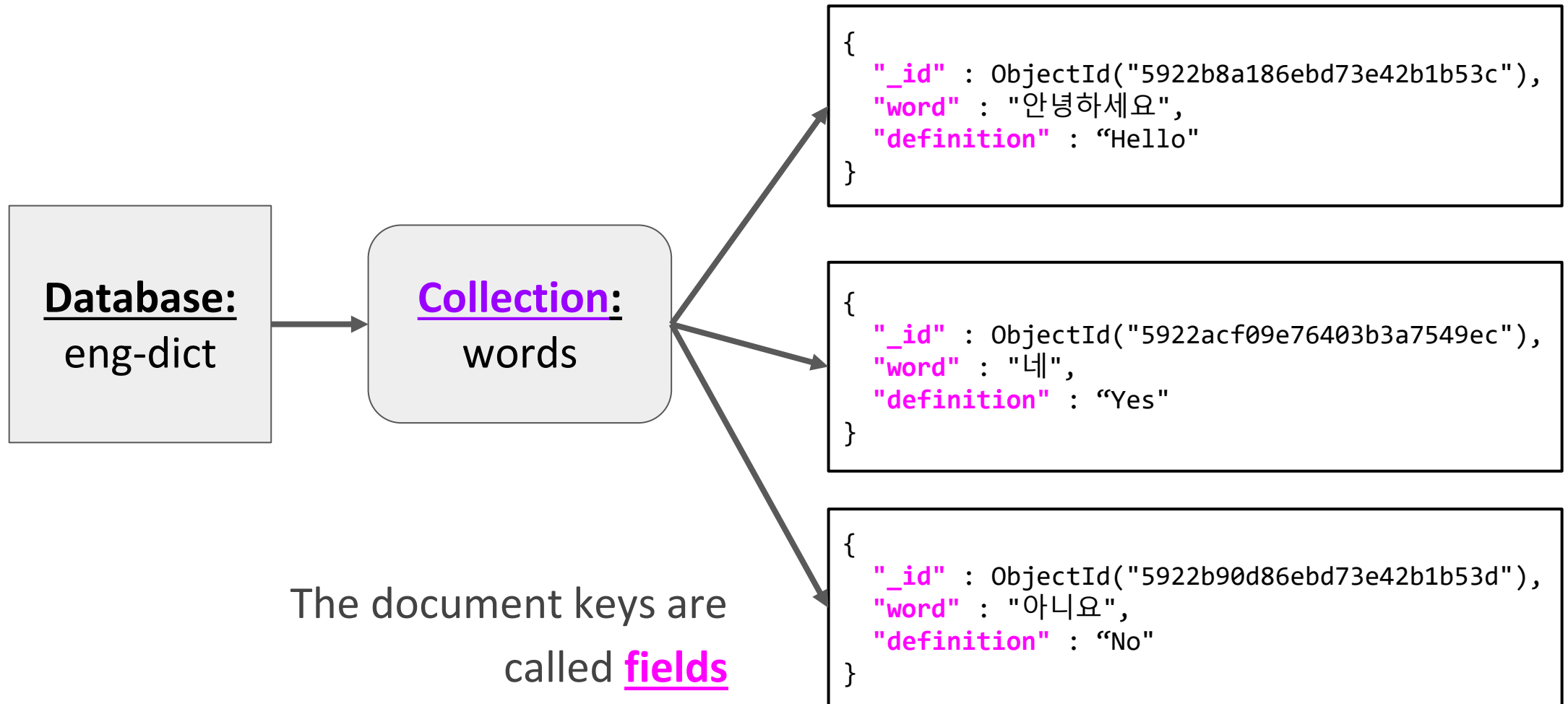
- A group of MongoDB **documents**.
- (**Table** in a relational database)

Document:

- A JSON-like object that represents one instance of a collection (**Row** in a relational database)
- Also used more generally to refer to any set of key-value pairs.

MongoDB Example

Documents:

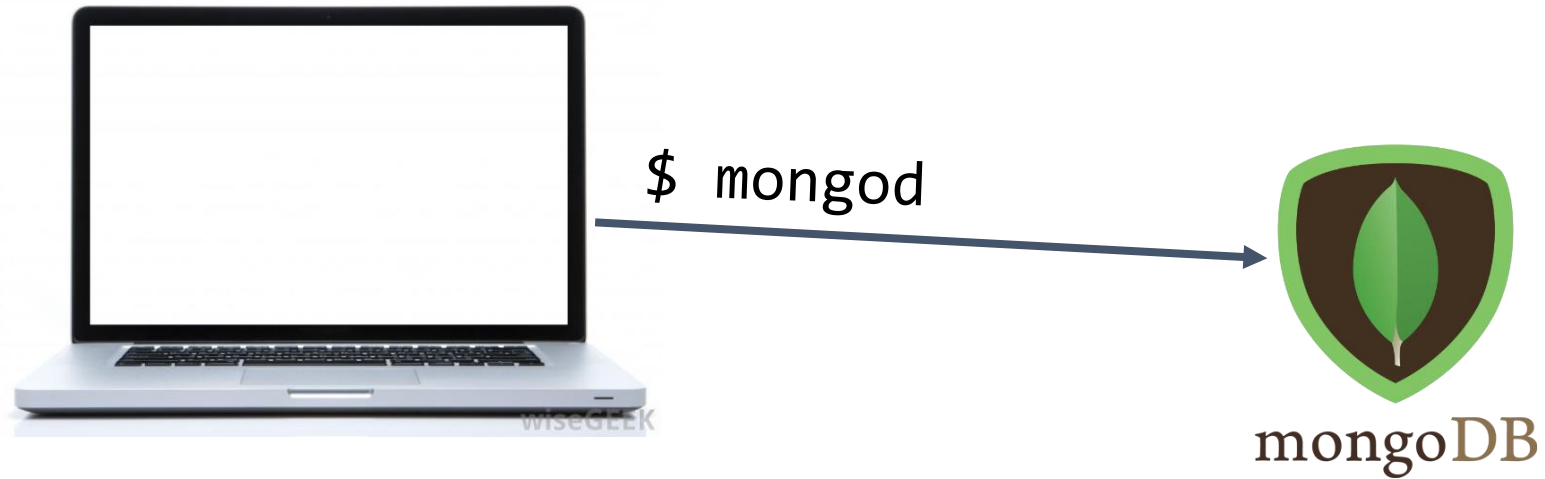


mongod: Database process



When you [install MongoDB](#), it will come with the `mongod` command-line program. This launches the MongoDB database management process and binds it to port 27017:
`$ mongod`

mongo: Command-line interface



You can connect to the MongoDB server through the **mongo** shell:
\$ mongo

mongo shell commands

- > show dbs
 - Displays the databases on the MongoDB server
- > use ***databaseName***
 - Switches current database to ***databaseName***
 - The ***databaseName*** does not have to exist already
 - It will be created the first time you write data to it
- > show collections
 - Displays the collections for the current database

mongo shell commands

> `db.collection`

- Variable referring to the *collection* collection

> `db.collection.find(query)`

- Prints the results of *collection* matching the query
- The *query* is a MongoDB Document (i.e. a JSON object)
 - To get everything in the *collection* use `db.collection.find()`
 - To get everything in the collection that matches `x=foo`,
`db.collection.find({x: 'foo'})`

mongo shell commands

> db.**collection**.findOne(*query*)

- Prints the first result of **collection** matching the query

> db.**collection**.insertOne(*document*)

- Adds **document** to the **collection**
- **document** can have any structure

> db.test.insertOne({ name: 'dan' })

> db.test.find()

{ "**_id**" : ObjectId("5922c0463fa5b27818795950"), "name" : "dan" }

MongoDB will automatically add a unique **_id** to every document in a collection.

mongo shell commands

> db.**collection**.deleteOne(*query*)

- Deletes the first result of **collection** matching the query

> db.**collection**.deleteMany(*query*)

- Delete multiple documents from **collection**.
- To delete all documents, db.**collection**.deleteMany({})

> db.**collection**.drop()

- Removes the collection from the database

Use-case: debugging

- Context: To show a list of all words in our dictionary, but empty list ???
- Completed code with server API, client fetch()
- Check **browser** console: no error
- Check **NodeJS** console: no error

WHAT TO DO?

Use-case: debugging

- Context: To show a list of all words in our dictionary, but empty list ???
- Completed code with server API, client fetch()
- Check browser console: no error
- Check NodeJS console: no error

WHAT TO DO?

Yes, Debugging (use a tool to run prog step-by-step & observe if any thing unexpected)

But, everything good

- **Check if any words in DB**
- Run the query in command-line if any results

mongo shell

When should you use the mongo shell?

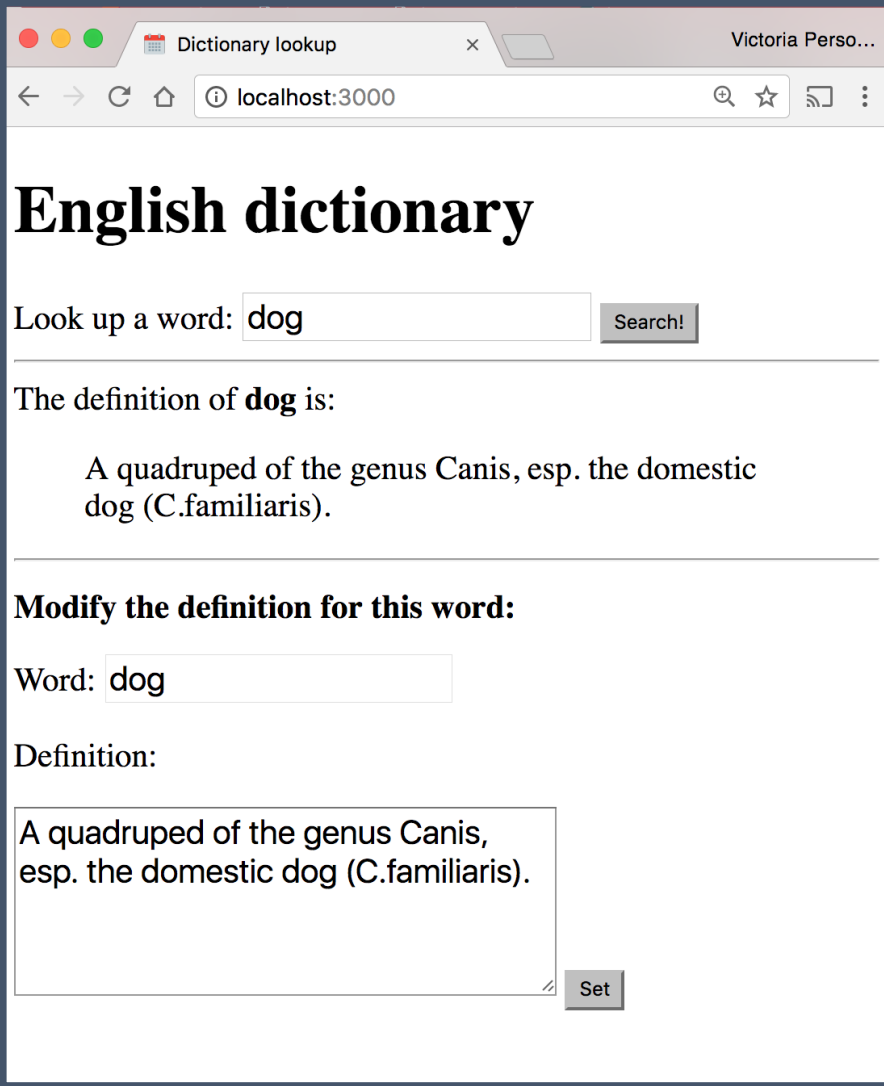
- Adding test data
- Deleting test data
- **Debugging:** to be sure if query gives expected results
- SSH

Continue with what we left off

Using MongoDB in a server

Dictionary with MongoDB

Let's change our Dictionary example to use a MongoDB backend instead of dictionary.json.



Dictionary lookup x Victoria Perso...

localhost:3000

English dictionary

Look up a word:

The definition of **dog** is:

A quadruped of the genus Canis, esp. the domestic dog (C.familiaris).

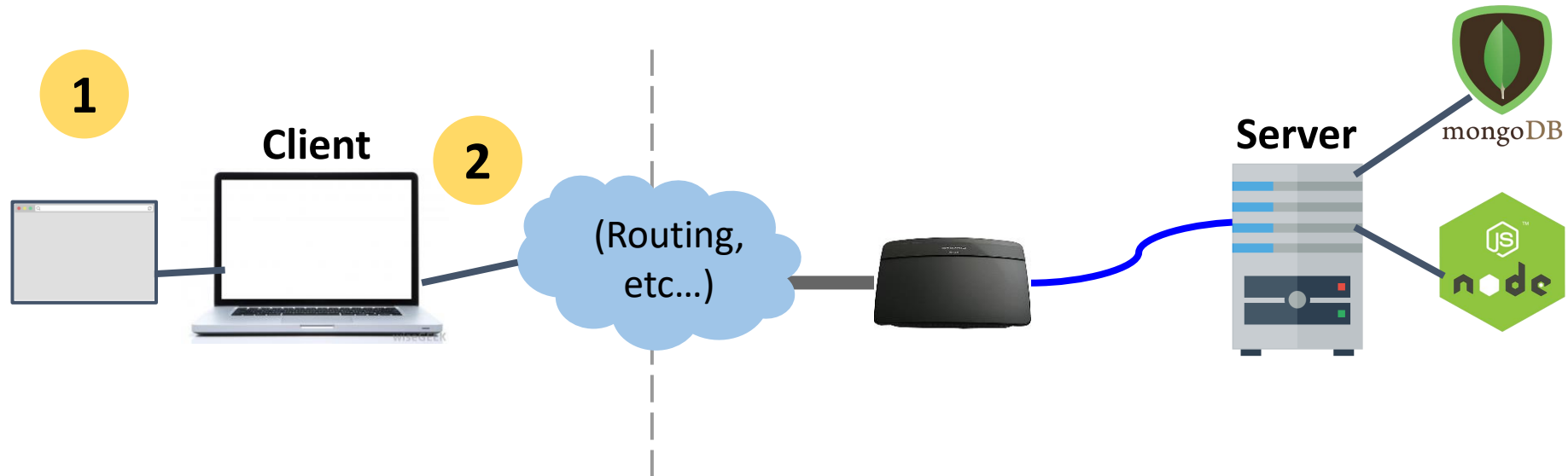
Modify the definition for this word:

Word:

Definition:

A quadruped of the genus Canis, esp. the domestic dog (C.familiaris).

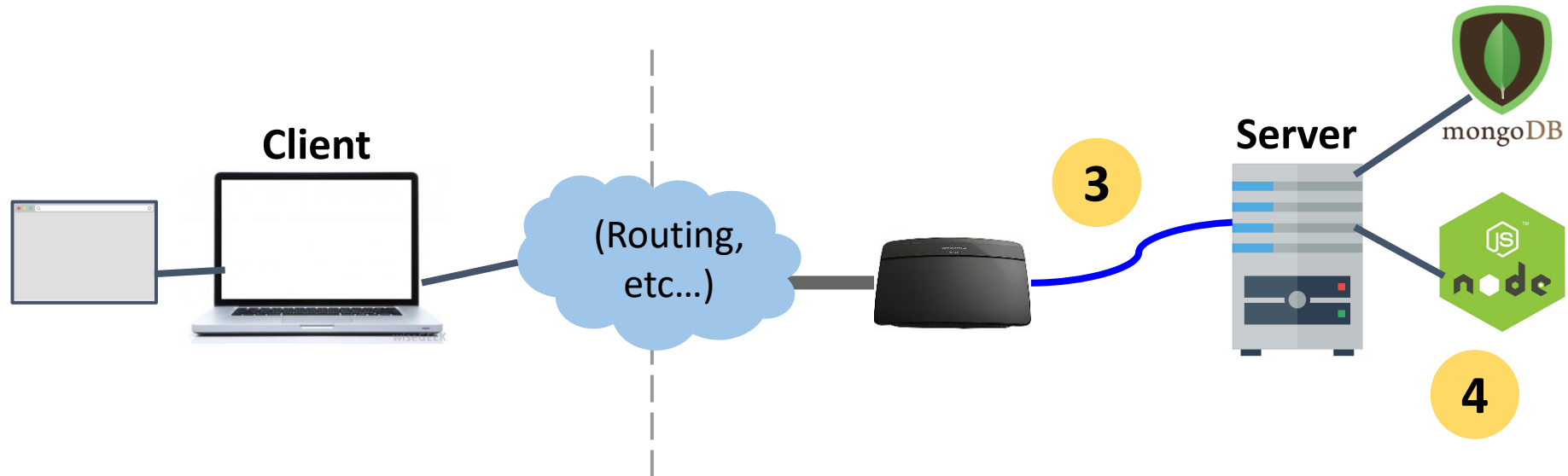
Review system



If we deployed our dictionary web app to abc.com:

1. The user navigates to abc.com
2. The browser makes an HTTP GET request for abc.com

Review system



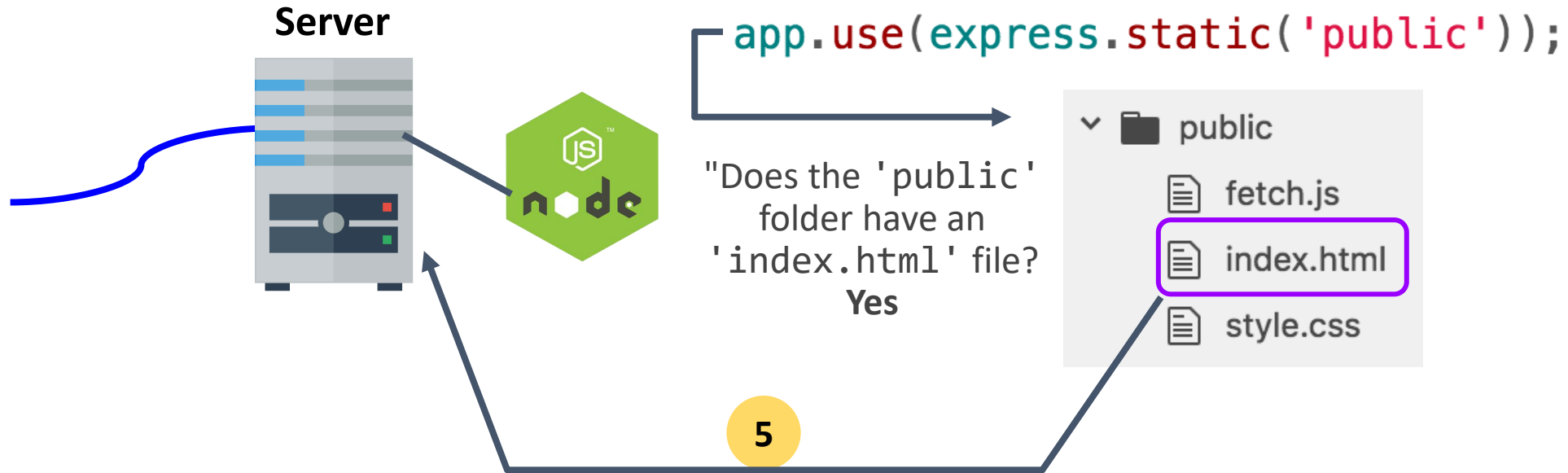
3. The server computer that is located at abc.com receives the HTTP GET request
4. The server computer gives the NodeJS server process the HTTP GET request message

Review system



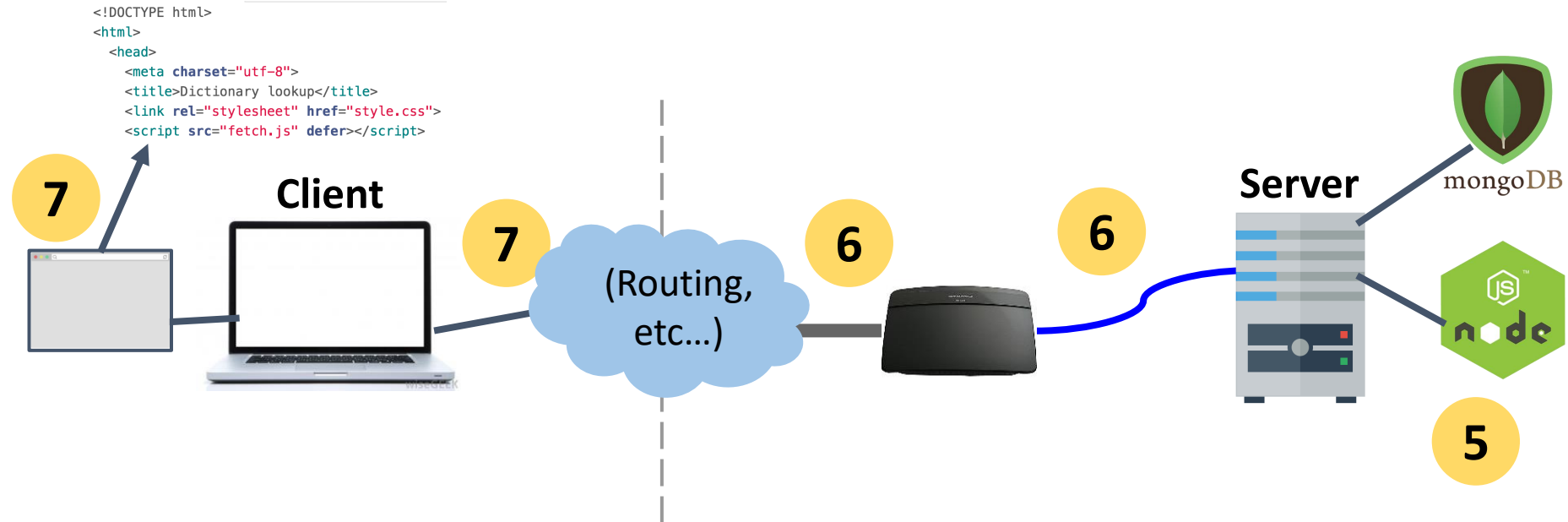
Our NodeJS server code has `app.use(express.static('public'))`; so it will first look to see if an `index.html` file exists in the `public` directory.

Review system



5. Since there is an index.html file, our NodeJS server will respond with the index.html file

Review system

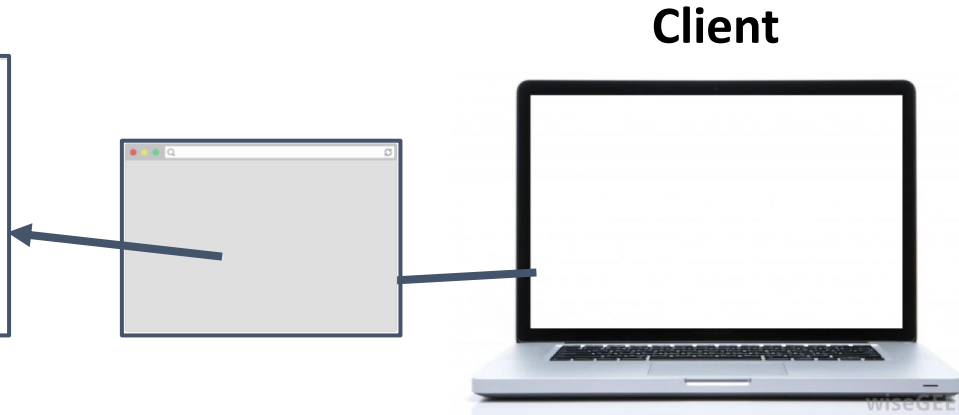


5. Our Node server program replies with the index.html file
6. The server computer sends back the index.html file
7. The browser receives the index.html file and begins to render it

Review system

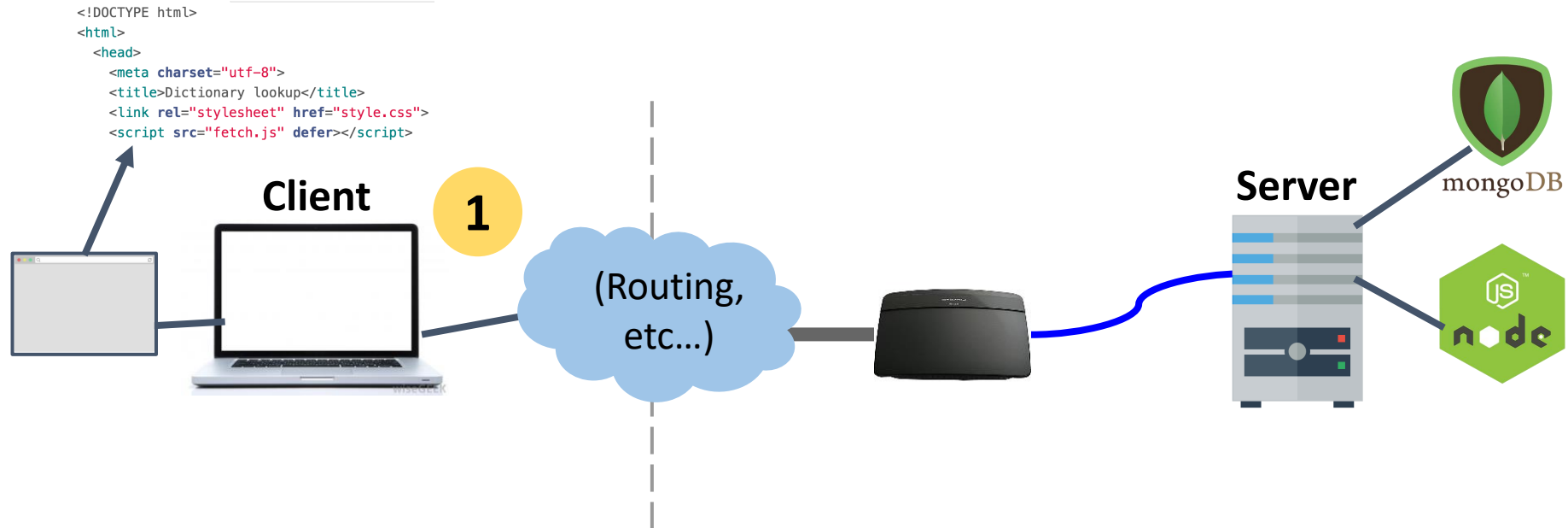
```
<link rel="stylesheet" href="style.css">  
<script src="fetch.js" defer></script>
```

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



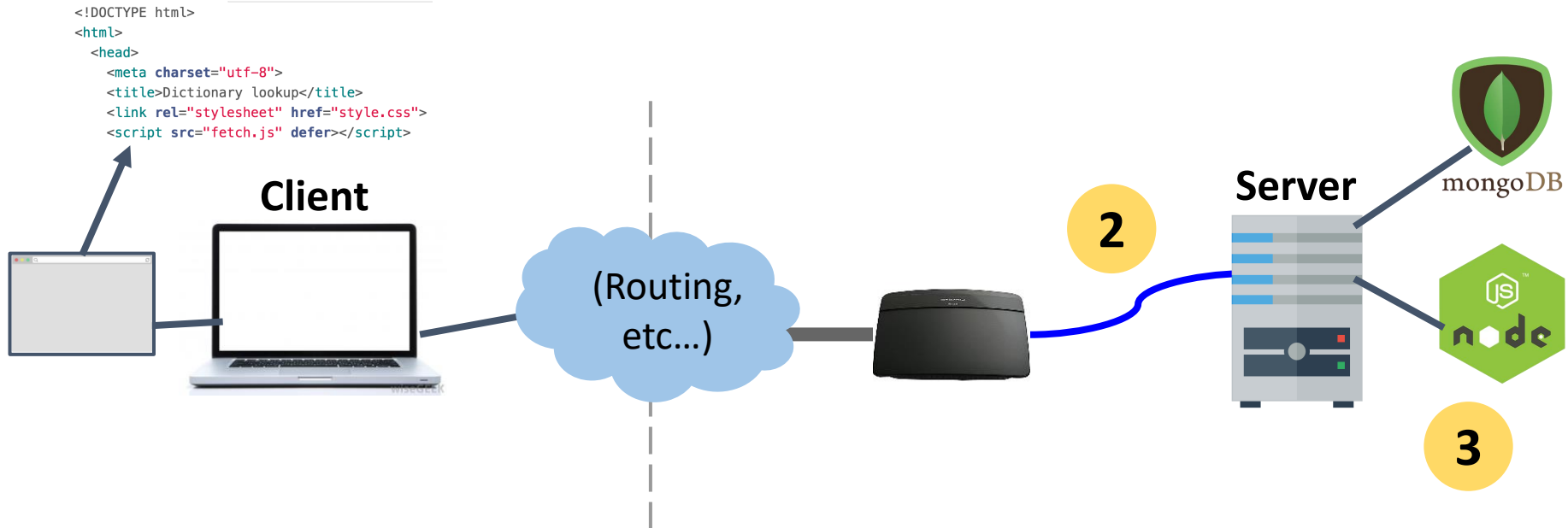
8. In rendering the HTML, the browser sees it needs style.css and fetch.js

Review system



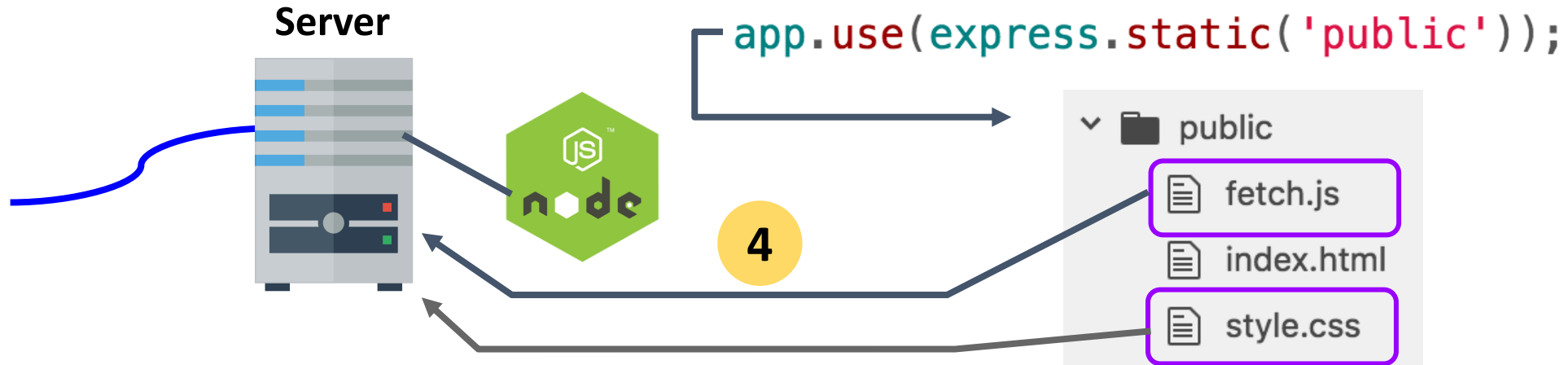
1. So the browser makes two more HTTP GET requests:
 - One for style.css
 - One for script.js

Review system



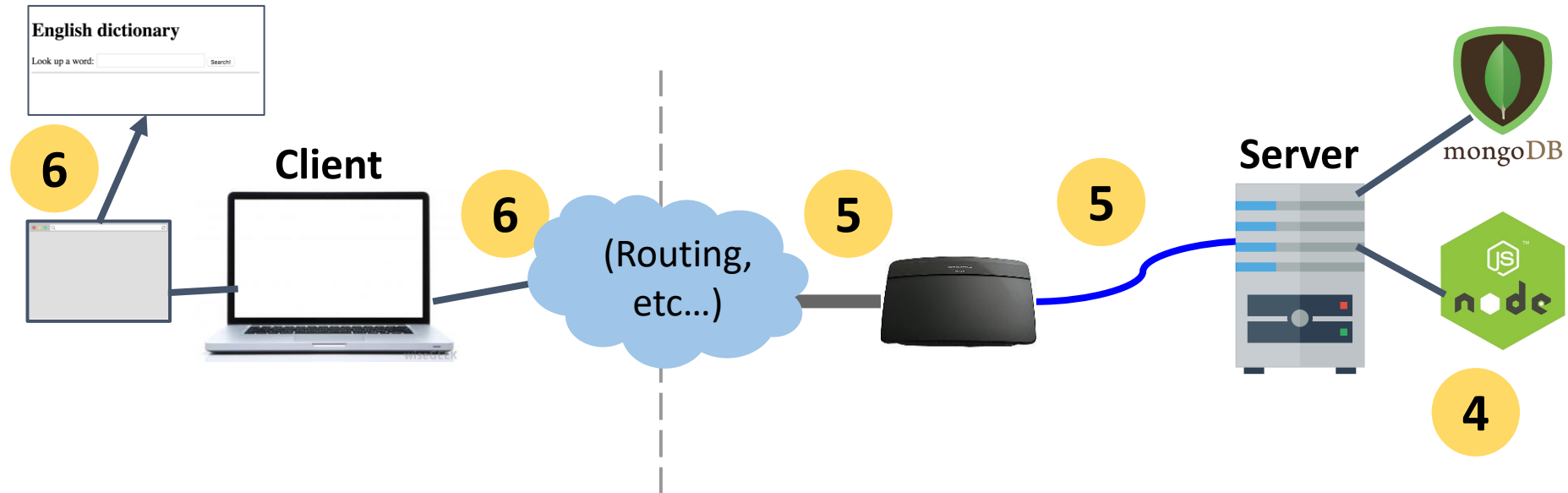
2. These GET requests get routed to the server computer
3. The server computer sends the GET requests to our NodeJS process

Review system



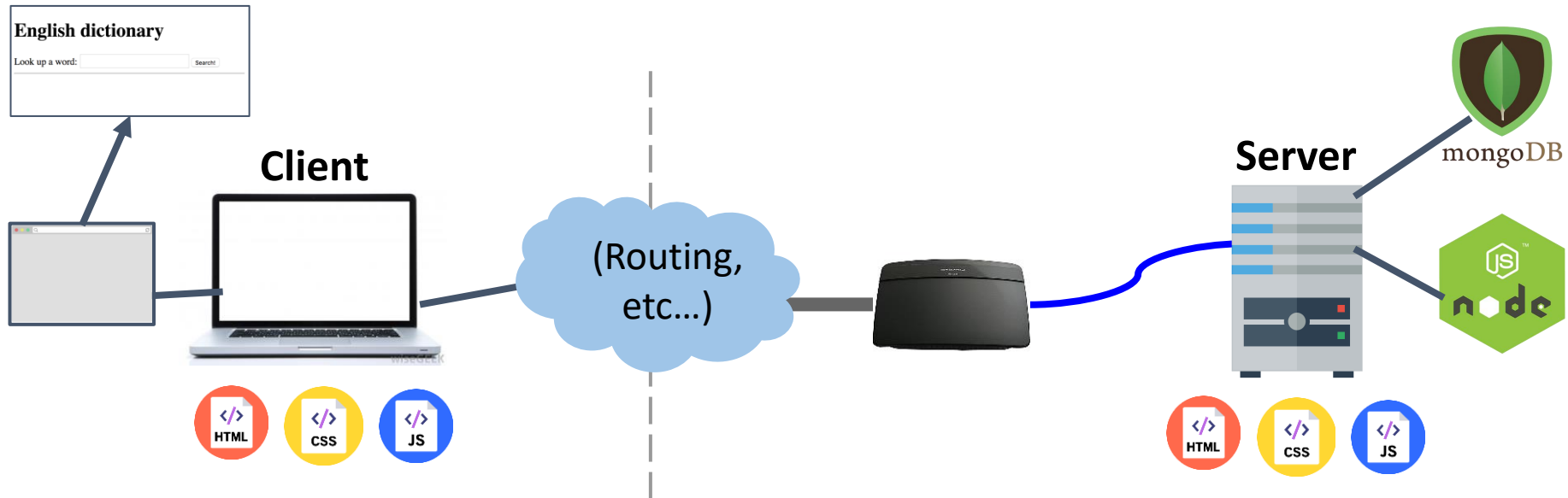
4. Our NodeJS server code finds `fetch.js` and `style.css` in the `public` directory, so it responds with those files

Review system



4. Our Node server program replies with the style.css and fetch.js files
5. The server computer sends these files back to the client
6. The browser receives the files and continues rendering index.html

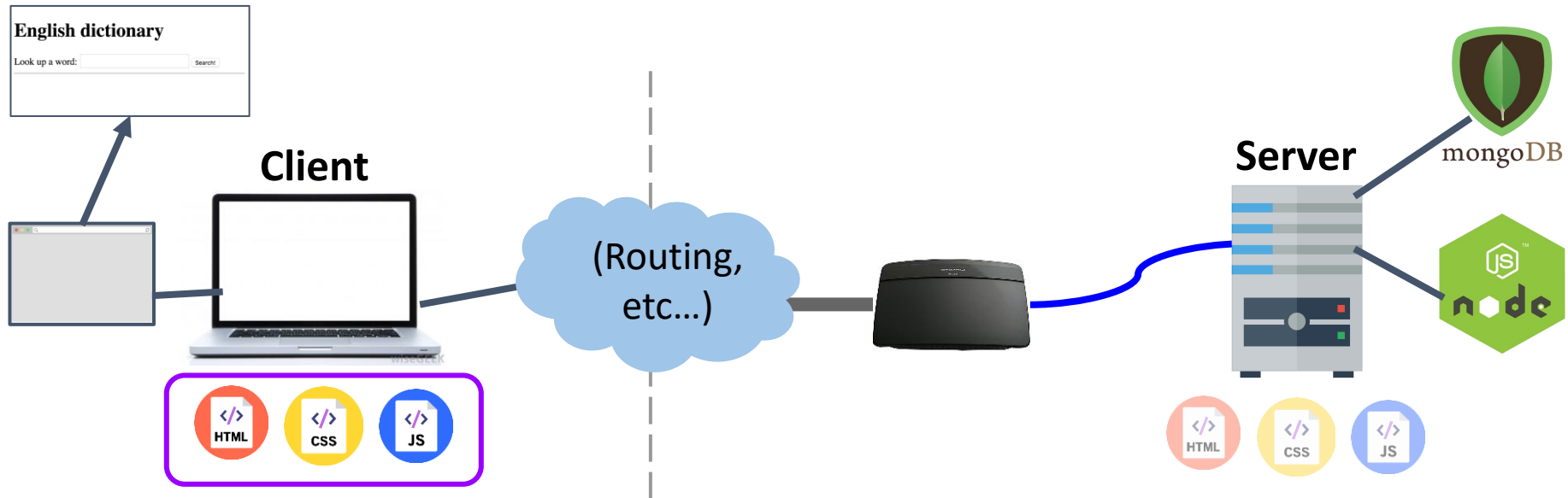
Review system



In this picture, there are **two copies** of index.html, style.css, and fetch.js:

- The server computer has these files stored in its file system
- The browser has just downloaded the files from the server

Review system



The server computer **provided** the files.

But the client computer is going to **execute** the files.

- So the code in `fetch.js` is going to be run on the client, not on the server.

Review system

English dictionary

Look up a word:

1



Client

```
const searchForm = document.querySelector('#search');  
searchForm.addEventListener('submit', onSearch);
```

1. The client has rendered the page and ran the JavaScript in `fetch.js` to attach the event listeners.
2. Then, when we enter a word and hit "Search"...

Review system

2

English dictionary

Look up a word:



Client

```
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();  
}
```

2. ...the onSearch function is executed on the client.

Review system



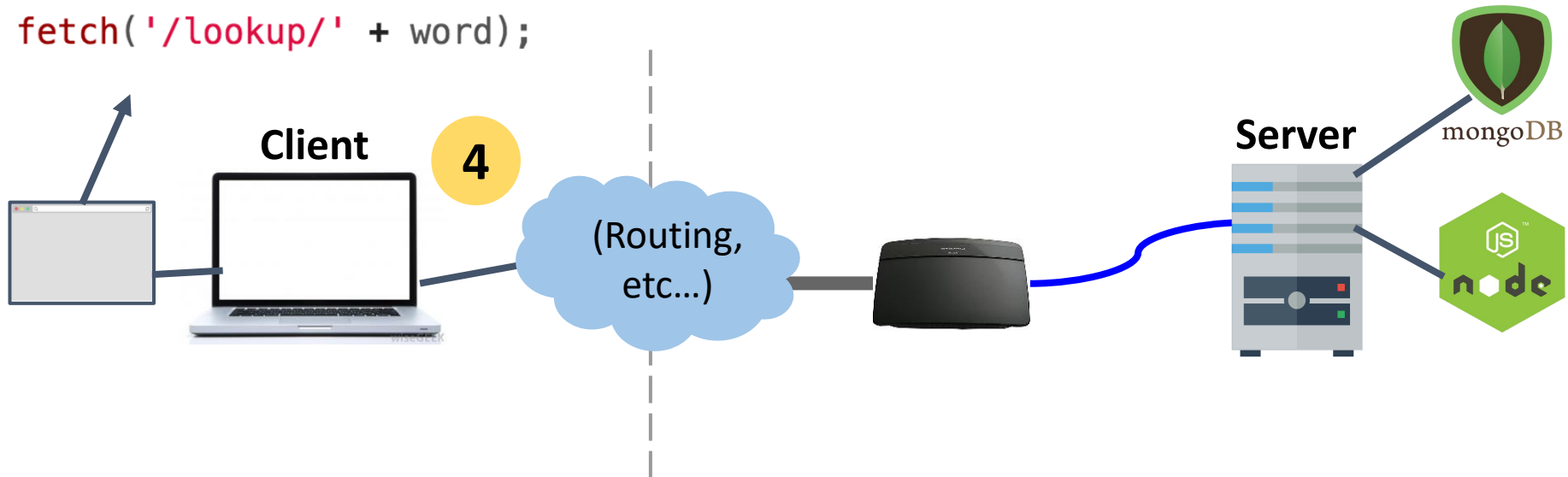
Client

3

```
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();  
}
```

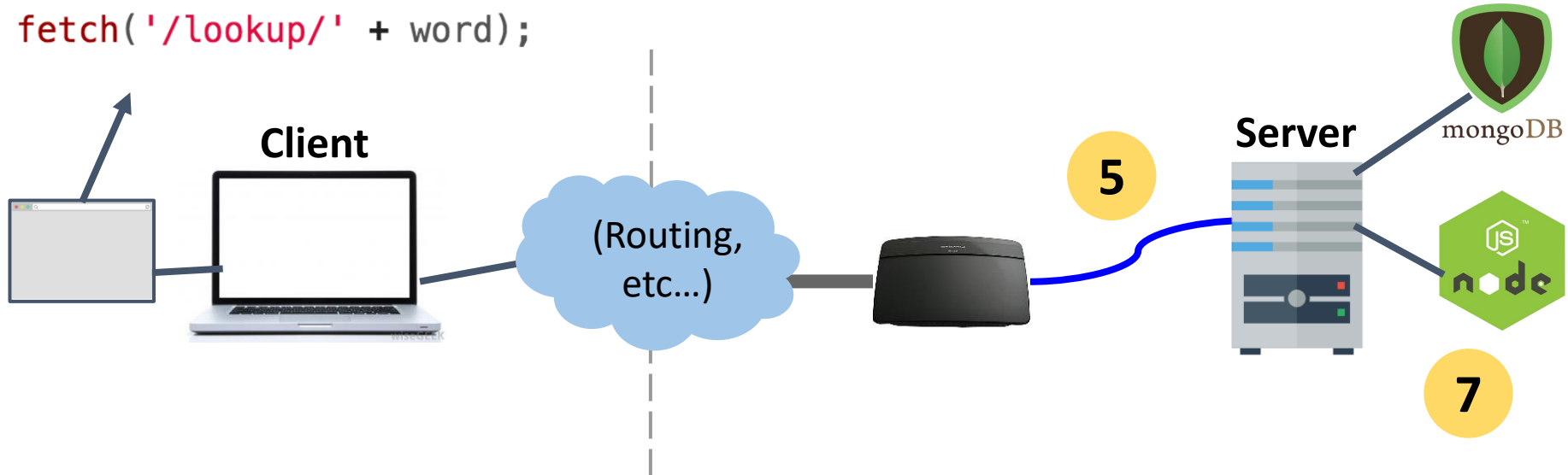
3. Our onSearch function includes a call to `fetch()`, which is going to trigger another HTTP GET request, this time for `abc.com/lookup/cat`.

Review system



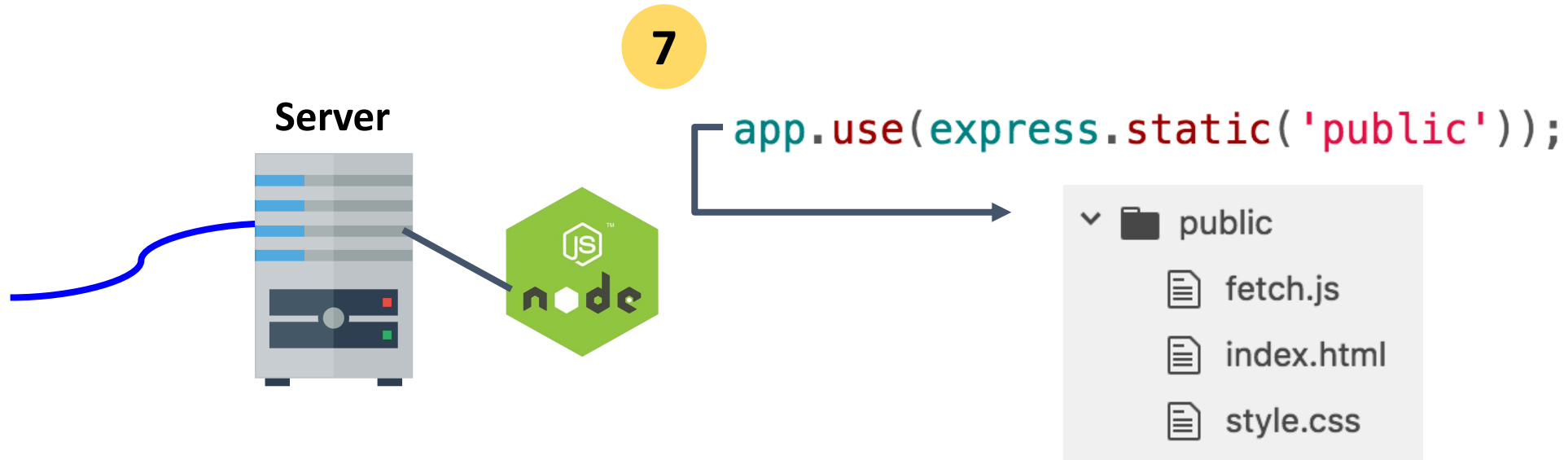
4. Because of the call to `fetch()`, the browser makes an HTTP GET request for `abc.com/lookup/cat`.

Review system



- 5. These GET requests get routed to the server computer
- 6. The server computer sends the GET requests to our NodeJS process

Review system



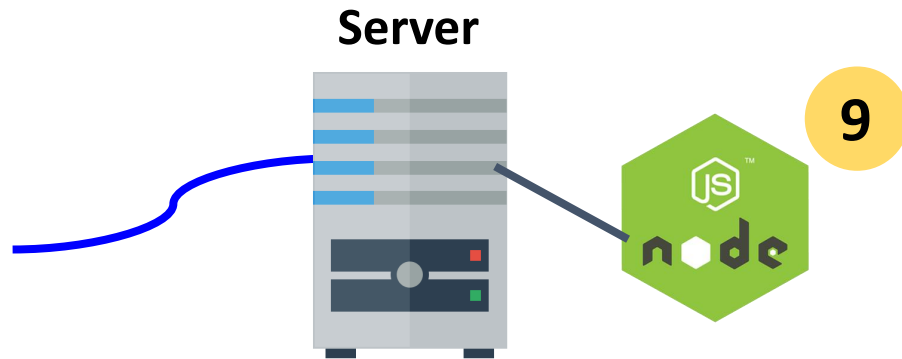
7. Our NodeJS server code first tries to see whether there's an "lookup/cat/index.html" in the `public` directory.

Review system



8. "public/lookup/cat/index.html" doesn't exist, so now it sees whether there's a route that matches GET "/lookup/cat":
- '/lookup/:word' matches, so onLookupWord is executed on the server

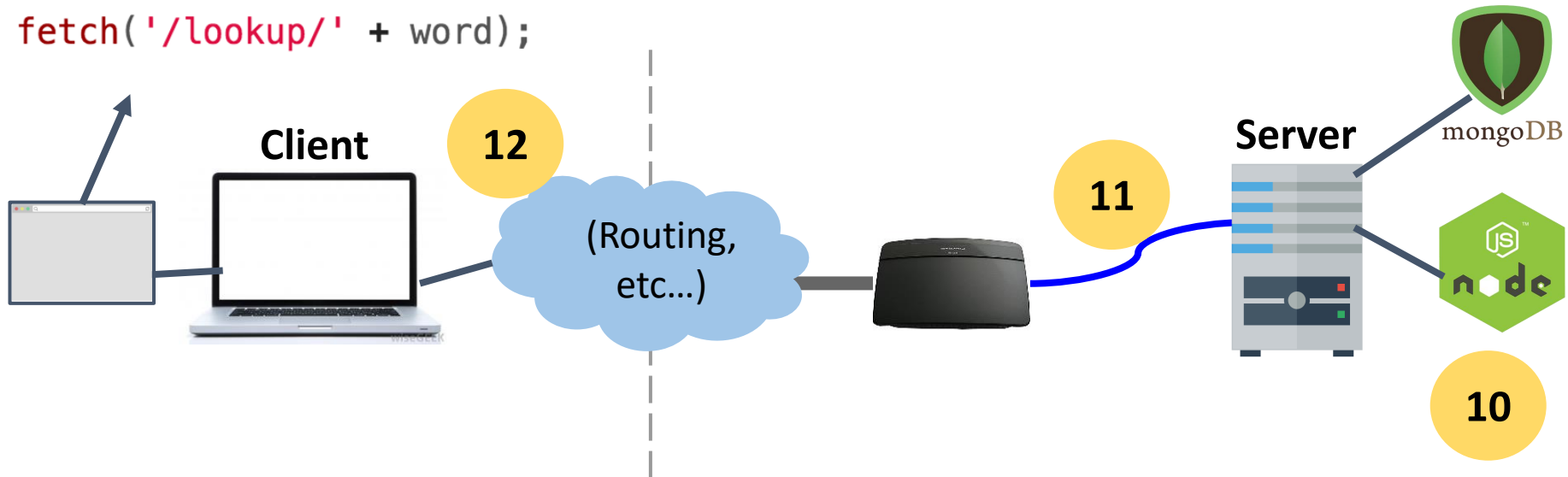
Review system



```
function onLookupWord(req, res) {  
  const routeParams = req.params;  
  const word = routeParams.word;  
  
  const key = word.toLowerCase();  
  const definition = englishDictionary[key];  
  
  res.json({  
    word: word,  
    definition: definition  
  });  
}
```

9. In the version we wrote before, we get the definition from the JSON dictionary file that's also located on the server.
- We'll change this to query MongoDB instead.

Review system



- 10. Our Node server program replies with JSON
- 11. The server computer sends JSON back to the client
- 12. The browser receives the JSON and continues executing the JavaScript

Review system



Client

13

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

```
wordDisplay.textContent = json.word;  
defDisplay.textContent = json.definition;  
results.classList.remove('hidden');
```

```
}
```

13. The onSearch function continues executing with the JSON results and updates the client page.

Review system



Client

English dictionary

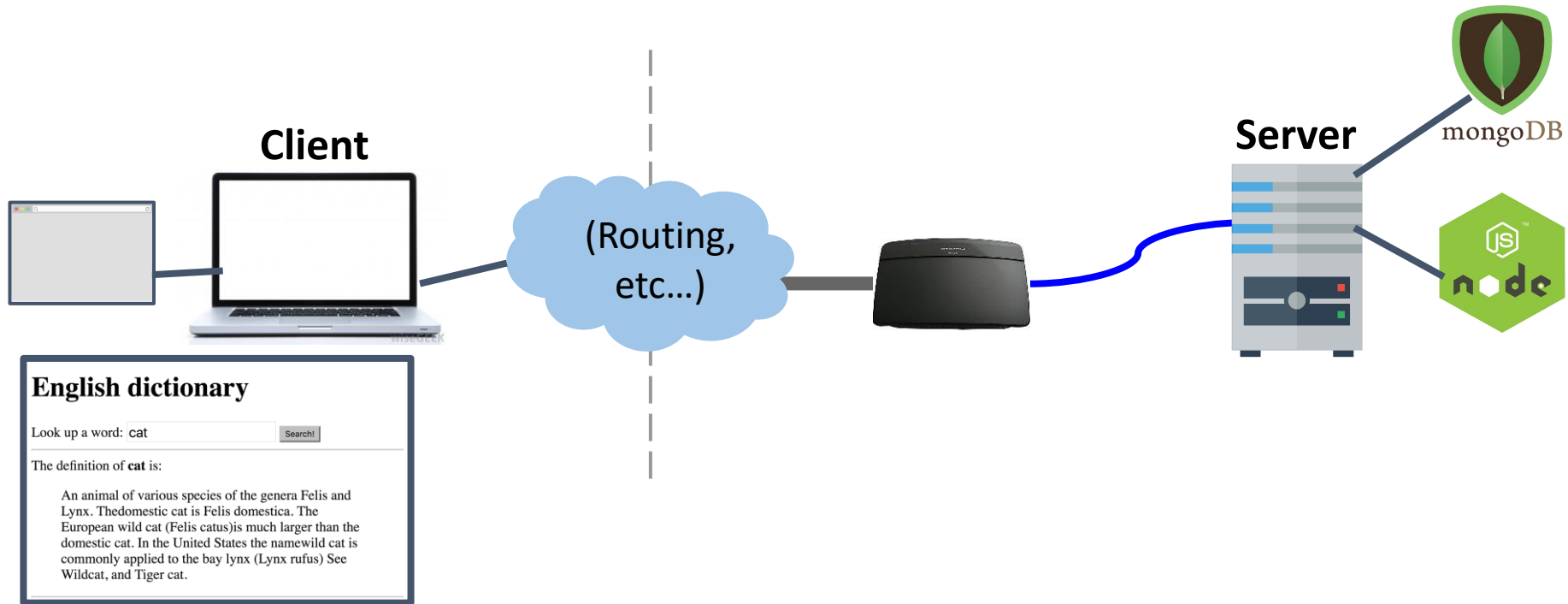
Look up a word:

Search!

The definition of **cat** is:

An animal of various species of the genera Felis and Lynx. The domestic cat is Felis domestica. The European wild cat (Felis catus) is much larger than the domestic cat. In the United States the name wild cat is commonly applied to the bay lynx (Lynx rufus) See Wildcat, and Tiger cat.

Review system



The server **generated** the JSON with the word and definition.
The client **consumed** the JSON with the word and definition.

Using MongoDB in a server

Starting a server: Before

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

(Previous code: This
doesn't use MongoDB)

Starting a server: After

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

Starting a server: After

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

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

let db = null;
let collection = null;

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

startServer();
```

Example: Dictionary

We want our server to load definitions from the dictionary...

English dictionary

Look up a word:

Search!

The definition of **cat** is:

An animal of various species of the genera Felis and Lynx. The domestic cat is Felis domestica. The European wild cat (Felis catus) is much larger than the domestic cat. In the United States the name wild cat is commonly applied to the bay lynx (Lynx rufus) See Wildcat, and Tiger cat.

JSON Dictionary lookup

```
function onLookupWord(req, res) {  
  const routeParams = req.params;  
  const word = routeParams.word;  
  
  const key = word.toLowerCase();  
  const definition = englishDictionary[key];  
  
  res.json({  
    word: word,  
    definition: definition  
  });  
}  
app.get('/lookup/:word', onLookupWord);
```

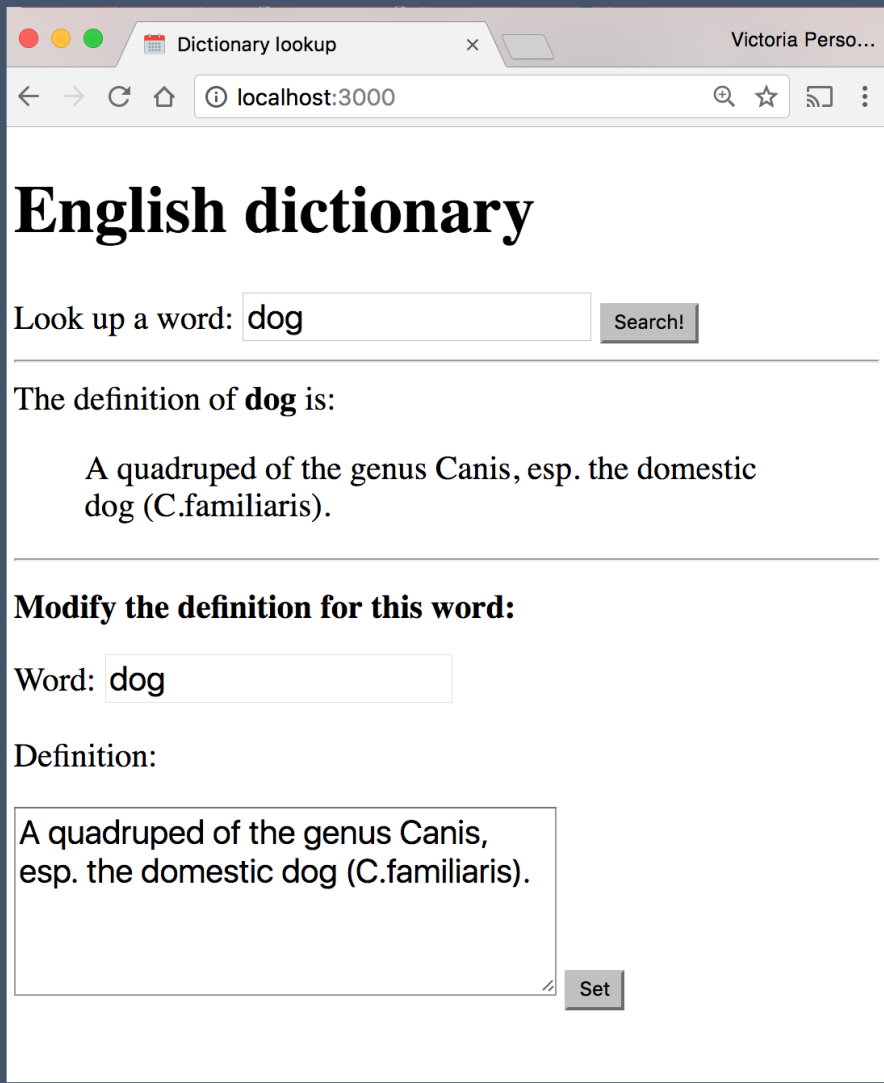
(Previous code: This
doesn't use MongoDB)

MongoDB Dictionary lookup

```
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);
```


Dictionary with MongoDB

And we want to modify definitions in the dictionary:



Dictionary lookup x Victoria Perso...

localhost:3000

English dictionary

Look up a word:

The definition of **dog** is:

A quadruped of the genus Canis, esp. the domestic dog (C.familiaris).

Modify the definition for this word:

Word:

Definition:

JSON Dictionary write

```
async function onSetWord(req, res) {  
  const routeParams = req.params;  
  const word = routeParams.word;  
  
  const definition = req.body.definition;  
  const key = word.toLowerCase();  
  englishDictionary[key] = definition;  
  
  res.json({ success: true });  
}  
app.post('/set/:word', onSetWord);
```

(Previous
code: This
doesn't use
MongoDB)

MongoDB Dictionary write

```
async function onSetWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;
  const definition = req.body.definition;
  const key = word.toLowerCase();

  const filter = { word: key };
  const update = { definition: definition };
  const options = { upsert: true };

  collection.updateOne()
  const response =
    await collection.updateOne(filter, { $set: update }, options);

  res.json({ success: true });
}
app.post('/set/:word', onSetWord);
```

package.json

Installing dependencies

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

```
$ npm install express  
$ npm install mongodb
```

These get written to the `node_modules` directory.

Uploading server code

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

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

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

Managing dependencies

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

npm provides a mechanism for this: [package.json](#)

package.json

You can put a file named [package.json](#) in the root directory of your NodeJS project to specify metadata about your project.

Create a [package.json](#) file using the following command:
`$ npm init`

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

Auto-generated package.json

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

[GitHub](#)

Saving deps to package.json

Now when you install packages:

```
$ npm install express  
$ npm install mongodb
```

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

```
"dependencies": {  
  "express": "^4.17.1",  
  "mongodb": "^3.6.2"  
}
```

Saving deps to package.json

If you remove the node_modules directory:

```
$ rm -rf node_modules
```

You can install your project dependencies again via:

```
$ npm install
```

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

npm scripts

Your package.json file also defines scripts:

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

You can run these scripts using `$ npm scriptName`

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

```
$ npm start
```

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)

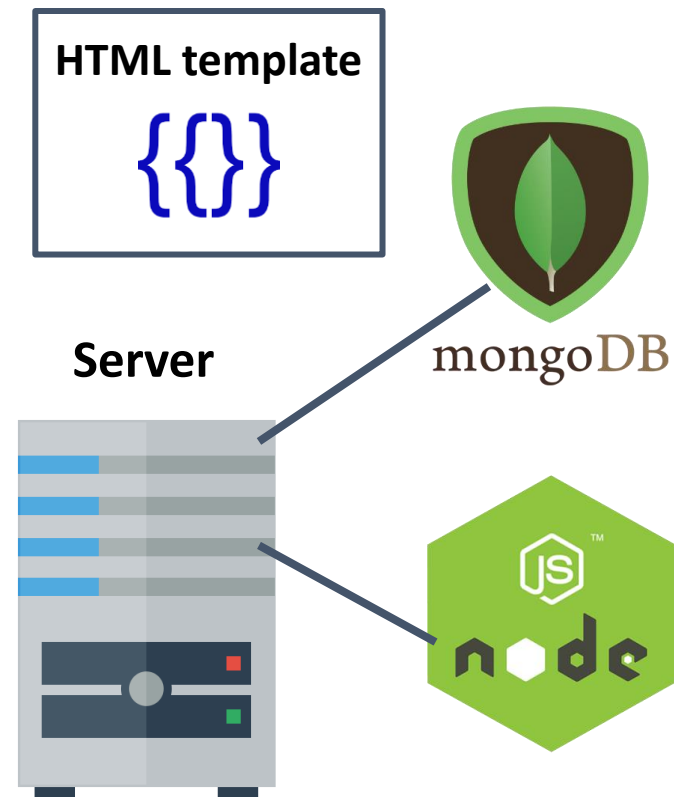
1. **Server-side rendering:**
 - We will show you how to do this
2. **Single-page application:**
 - We will show you how to do this
3. Combination of 1 and 2 ("**Isomorphic**" / "**Universal**")
 - This is probably the most common technique
 - We will talk about this but won't show you how to code it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

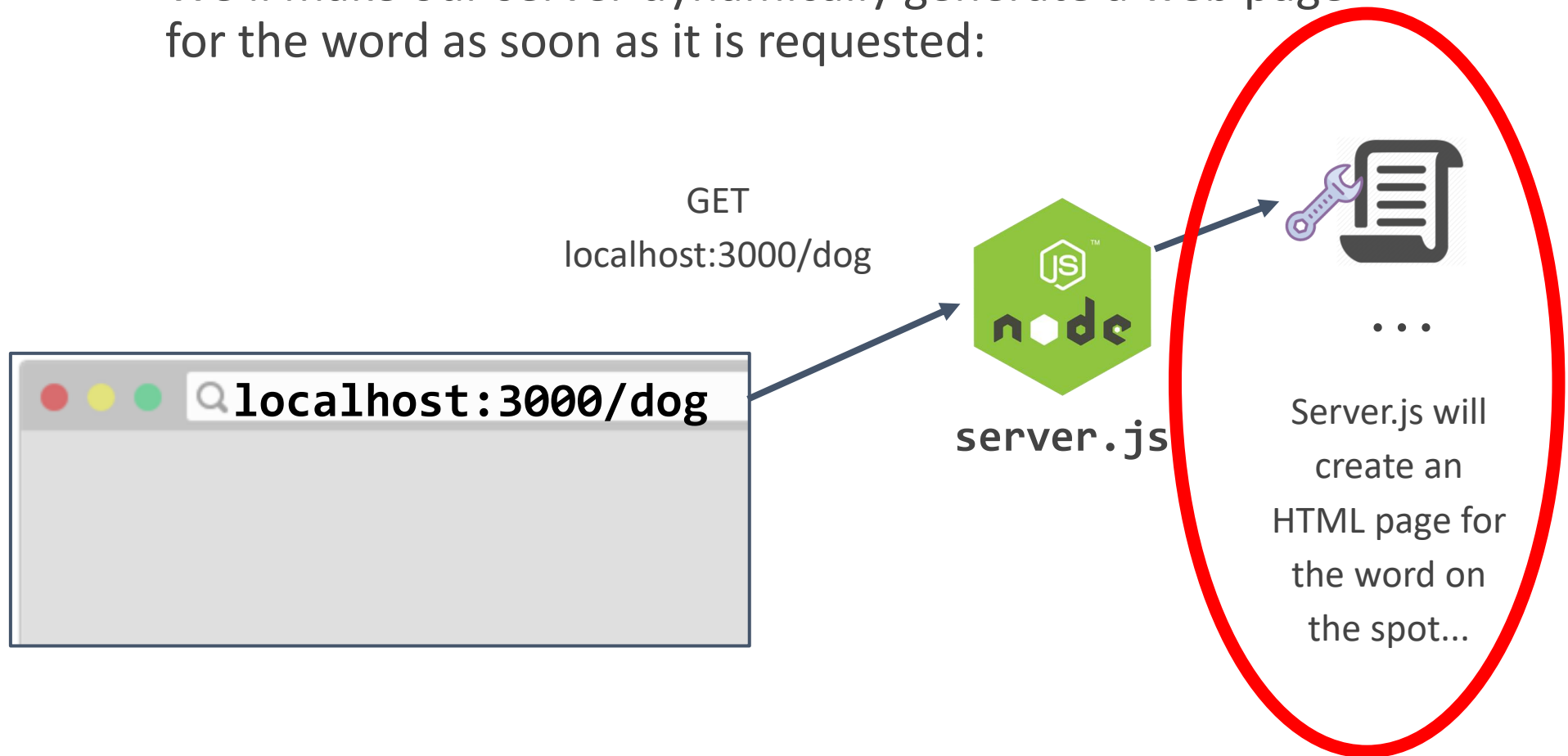
- The server generates a different web page
- Usually involves filling out and returning an HTML template in response to a request
 - This is done by a **templating engine**: Pug (Jade), EJS, Handlebars, etc



Dynamically generated pages

Example: Dictionary

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



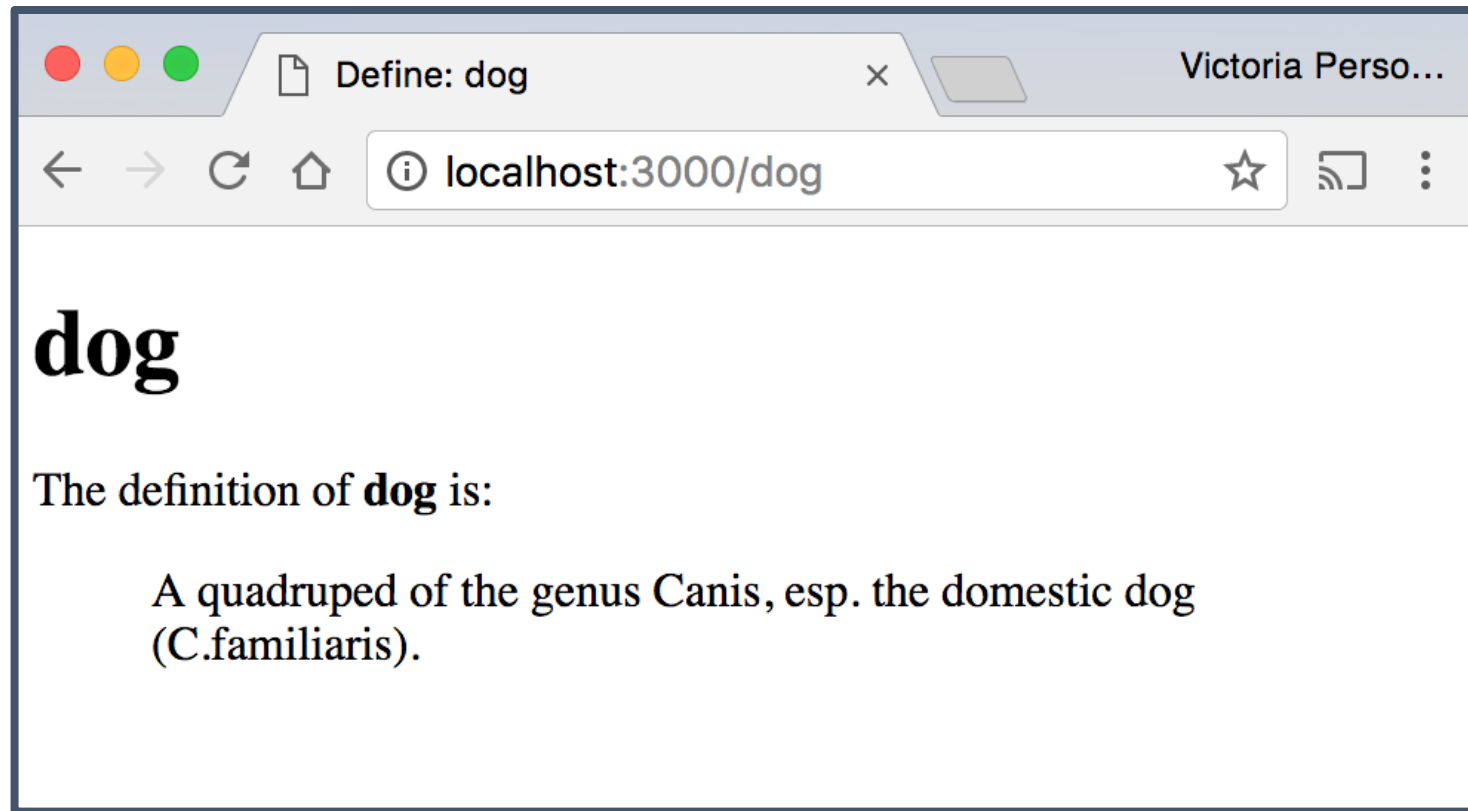
A solution: HTML Strings

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;

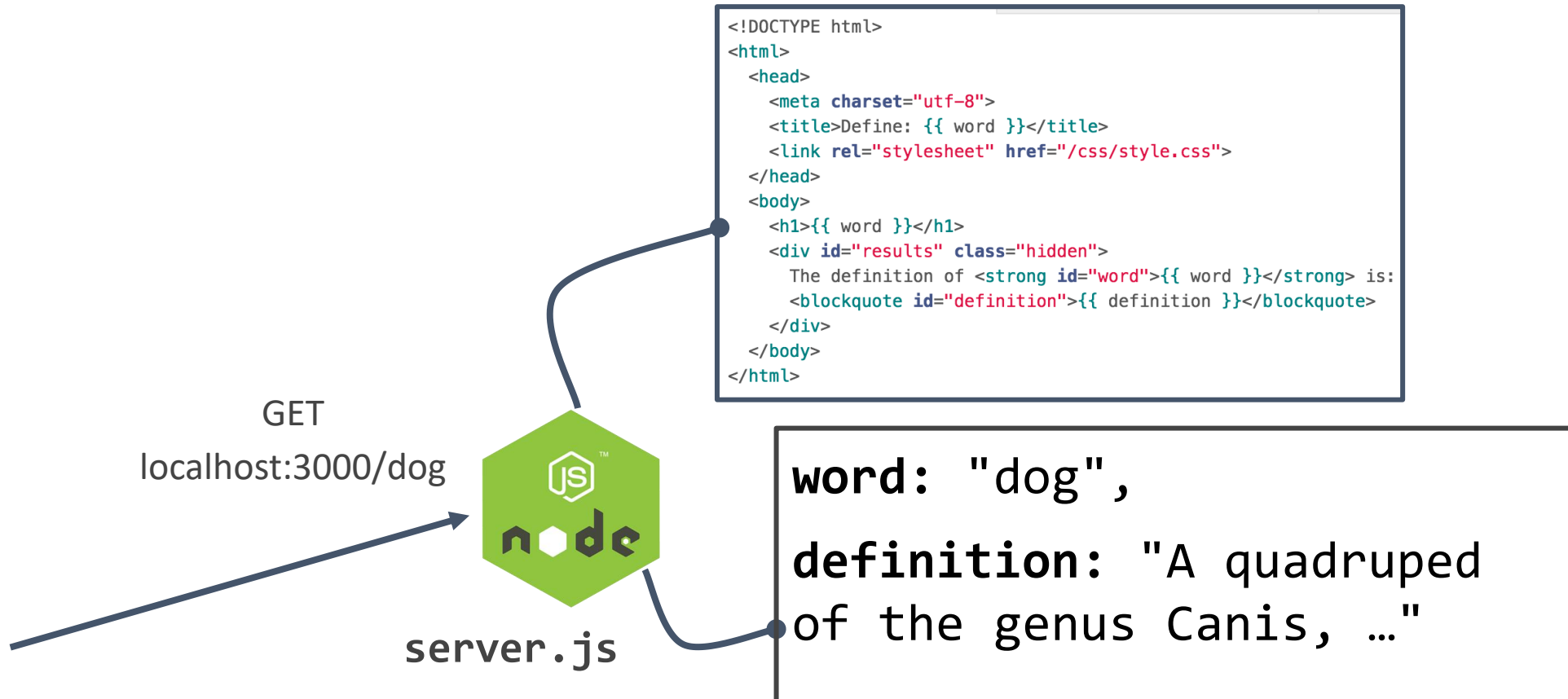
  const query = { word: word.toLowerCase() };
  const result = await collection.findOne(query);
  const definition = result ? result.definition : '';

  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

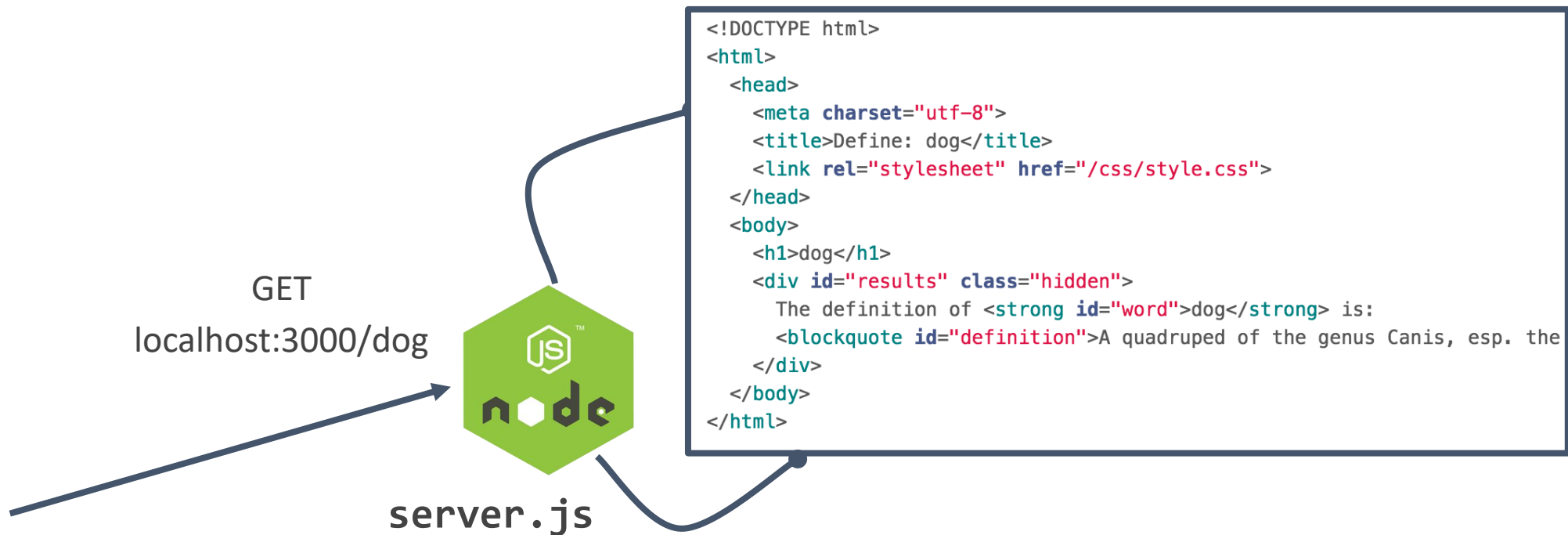
Goal: HTML Template

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



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 exphrs = require('express-handlebars');
```

```
...
```

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

```
const hbs = exphrs.create();
```

```
app.engine('handlebars', hbs.engine);
```

```
app.set('view engine', 'handlebars');
```

eng-dict: Server-side rendering

```
✓ eng-dict-server-side
  ✓ public
    > css
    ✓ js
  ✓ views
    🍷 index.handlebars
    🍷 word.handlebars
  {} package.json
  JS server.js
```

We change our eng-dict example to have 2 Handlebars templates:

- index.handlebars
- word.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 lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Dictionary</title>
  <link rel="stylesheet" href="/css/style.css">
  <script src="/js/index.js" defer></script>
</head>
<body>
  <h1>English dictionary</h1>
  <form>
    <label for="word">Look up a word:</label>
    <input type="text" id="word">
    <button type="submit">Search!</button>
  </form>
  <hr>
```

word.handlebars

But for the word-view, we want a different word and definition depending on the word:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <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>
```

eng-dict: 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

eng-dict: Server-side rendering

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

```
async function onGetWord(req, res) {  
  const wordId = req.params.id;  
  const collection = db.collection('words');  
  const doc = await collection.findOne({ _id: ObjectId(wordId) });  
  
  res.render('word', { word: doc.word, definition: doc.definition });  
}  
app.get('/words/:id', onGetWord);
```