CS193X:
Web Programming Fundamentals

Spring 2017

Victoria Kirst
(vrk@stanford.edu)

CS193X schedule

Today

- Servers and MongoDB
- Web application architecture
  - Server-side rendering with Handlebars
  - Single-page web app

- Final Project Assigned
  - Proposal due **Friday 6/2**
- HW5 due **Tuesday 5/30**

MongoDB installation

This lecture assumes you have **installed MongoDB**:

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

MongoDB and NodeJS

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 objects

The `mongodb` Node library provides objects to manipulate the database, collections, and documents:

- [Db](#): Database; can get collections using this object

- [Collection](#): Can get/insert/delete documents from this collection via calls like `insertOne`, `find`, etc.

- Documents are not special classes; they are just JavaScript objects

collection.insertOne (Promise)

```
const result = await collection.insertOne(doc);
```

- Adds one item to the collection

- *doc* is a JavaScript object representing the key-value pairs to add to the collection

- Returns a `Promise` that resolves to a `result` object when the insertion has completed

  - `result.`**`insertedId`** will contain the id of the object that was created

Promise version

```
async function insertWordAsync(word, definition) {
  const doc = {
    word: word,
    definition: definition
  };
  const result = await collection.insertOne(doc);
  console.log(`Document id: ${result.insertedId}`);
}
```

We will be using the Promise + async/await versions of all the MongoDB asynchronous functions, as it will help us avoid callback hell

collection.findOne

`const doc = await collection.findOne(`*`query`*`);`

- Finds the first item in the collection that matches the query

- *query* is a JS object representing which fields to match on

- Returns a `Promise` that resolves to a `document` object when `findOne` has completed

  - `doc` will be the JS object, so you can access a field via `doc`*`.fieldName`*, e.g. `doc._id`

  - If nothing is found, `doc` will be `null`

collection.findOne

```
async function printWord(word) {
  const query = {
    word: word
  };
  const response = await collection.findOne(query);
  console.log(
    `Word: ${response.word},
     definition: ${response.definition}`);
}
```

collection.find()

`const cursor = await collection.find(`*`query`*`);`

- Returns a [Cursor](#) to pointing to the first entry of a set of documents matching the query

- You can use `hasNext` and `next` to iterate through the list:

```javascript
async function printAllWordsCursor() {
  const cursor = await collection.find();
  while (await cursor.hasNext()) {
    const result = await cursor.next();
    console.log(`Word: ${result.word}, definition: ${result.definition}`);
  }
}
```

(This is an example of something that is **a lot** easier to do with `async/await`)

collection.find().toArray()

```
const cursor = await collection.find(query);
const list = await cursor.toArray();
```

- [Cursor](#) also has a `toArray()` function that converts the results to an array

```javascript
async function printAllWords() {
  const results = await collection.find().toArray();
  for (const result of results) {
    console.log(`Word: ${result.word}, definition: ${result.definition}`);
  }
}
```

collection.update

`await collection.update(`***query, newEntry***`);`

- Replaces the item matching ***query*** with ***newEntry***

    - (Note: This is the simplest version of update. There are more complex versions of update that we will address later.)

collection.update

```javascript
async function updateWord(word, definition) {
  const query = {
    word: word
  };
  const newEntry = {
    word: word,
    definition: definition
  };
  const response = await collection.update(query, newEntry);
}
```

"Upsert" with collection.update

MongoDB also supports "upsert", which is
- Update the entry if it already exists
- Insert the entry if it doesn't already exist

```
const params = { upsert: true };
await collection.update(query, newEntry, params);
```

"Upsert" with collection.update

```
async function upsertWord(word, definition) {
  const query = {
    word: word
  };
  const newEntry = {
    word: word,
    definition: definition
  };
  const params = {
    upsert: true
  }
  const response = await collection.update(query, newEntry, params);
}
```

collection.deleteOne/Many

```
const result = await collection.deleteOne(query);
```

- Deletes the first the item matching *query*
- `result.deletedCount` gives the number of docs deleted

```
const result = await collection.deleteMany(query);
```

- Deletes all items matching *query*
- `result.deletedCount` gives the number of docs deleted
- Use `collection.deleteMany()` to delete everything

collection.deleteOne

```javascript
async function deleteWord(word) {
  const query = {
    word: word
  };
  const response = await collection.deleteOne(query);
  console.log(`Number deleted: ${response.deletedCount}`);
}
```

collection.deleteMany

```
async function deleteAllWords() {
  const response = await collection.deleteMany();
  console.log(`Number deleted: ${response.deletedCount}`);
}
```

Advanced queries

MongoDB has a very powerful querying syntax that we did not cover in these examples.

For more complex queries, check out:

- <u>Querying</u>
    - <u>Query selectors and projection operators</u>
    - ```
      db.collection('inventory').find({ qty: { $lt: 30 } });
      ```

- <u>Updating</u>
    - <u>Update operators</u>
      ```
      db.collection('words').updateOne(
        { word: searchWord },
        { $set: { definition: newDefinition }})
      ```

Looking up documents by MongoDB id (`ObjectID`)

MongoDB: _id

MongoDB **creates a unique id** for each doc it creates.

- This is stored in a **_id** field and consists of a string stored in an <u>ObjectID</u> object.

```
> db.people.insertOne({ name: 'Mary' })
```

```
> db.people.findOne({name: "Mary"})
{ "_id" : ObjectId("59287fde798a736fafe91b8e"), "name" : "Mary" }
```

Recall the id is returned after we've inserted a new document:

```
const result = await collection.insertOne(doc);
```

- result.**insertedId** will contain the _id of the object that was created

Find by Mongo ID: `ObjectID`

Let's say you have a route that creates documents:

```javascript
async function onSaveCard(req, res) {
  const style = req.body.style;
  const message = req.body.message;

  const doc = {
    style: style,
    message: message
  };
  const collection = db.collection('card');
  const response = await collection.insertOne(doc);

  res.json({ cardId: response.insertedId });
}
app.post('/save', jsonParser, onSaveCard);
```

Find by Mongo ID: `ObjectID`

Let's say you have a route that creates documents:

```javascript
async function onSaveCard(req, res) {
  const style = req.body.style;
  const message = req.body.message;

  const doc = {
    style: style,
    message: message
  };
  const collection = db.collectio
  const response = await collectio

  res.json({ cardId: response.insertedId });
}
app.post('/save', jsonParser, onSaveCard);
```

You are returning the MongoDB-generated unique ID for the document, for querying later.

Find by Mongo ID: `ObjectID`

If you want to find a document by its MongoDB generated `_id` field, **you must wrap the string id in an <u>ObjectID</u>**.

- You need to import the `ObjectID` class first.

```javascript
const ObjectID = require('mongodb').ObjectID;

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

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

Find by Mongo ID: `ObjectID`

If you want to find a document by its MongoDB generated `_id` field, **you must wrap the string id in an <u>ObjectID</u>**.

- You need to import the `ObjectID` class first.

```javascript
const ObjectID = require('mongodb').ObjectID;

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

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

Using MongoDB in a server

Dictionary with MongoDB

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

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

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

Starting a server: After

```javascript
const DATABASE_NAME = 'eng-dict';
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();
```

Example: Dictionary

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

English dictionary

Look up a word: cat   Search!

The definition of **cat** is:

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.

JSON Dictionary lookup

```javascript
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

```javascript
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:

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;

  // Write the entry back to the JSON file.
  await fse.writeJson('./dictionary.json', englishDictionary);
  res.json({ success: true });
}
app.post('/set/:word', jsonParser, onSetWord);
```

(Previous code: This **doesn't** use MongoDB)

MongoDB Dictionary write

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

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

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

Using MongoDB in a server

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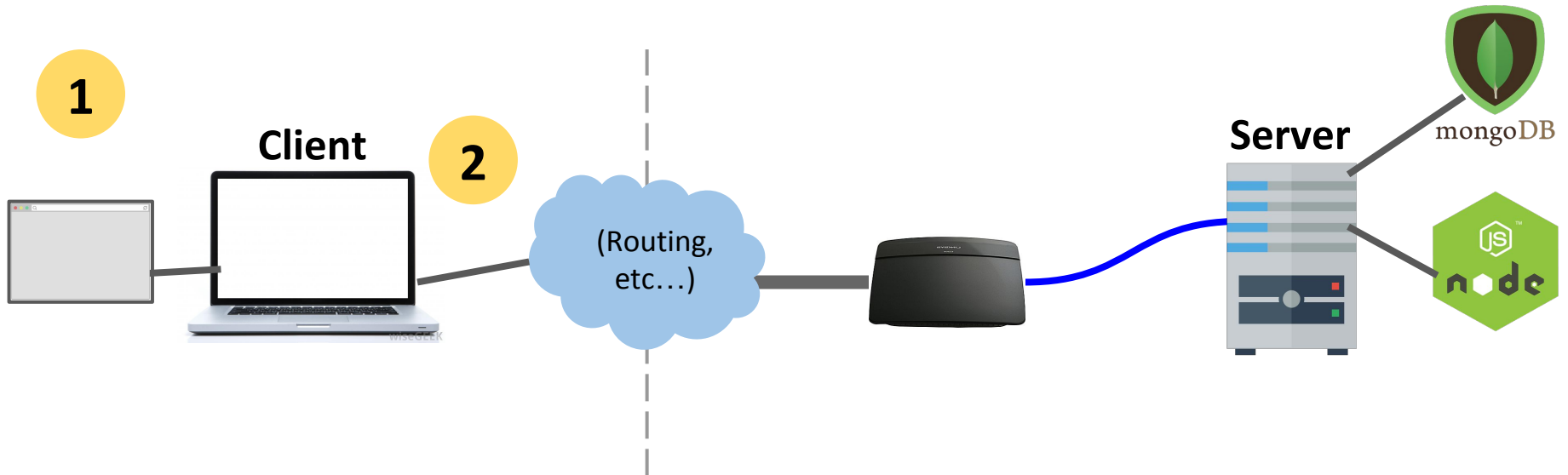
If we deployed our dictionary web app to abc.com:

1. The user navigates to abc.com

2. The browser makes an HTTP GET request for abc.com

Review system
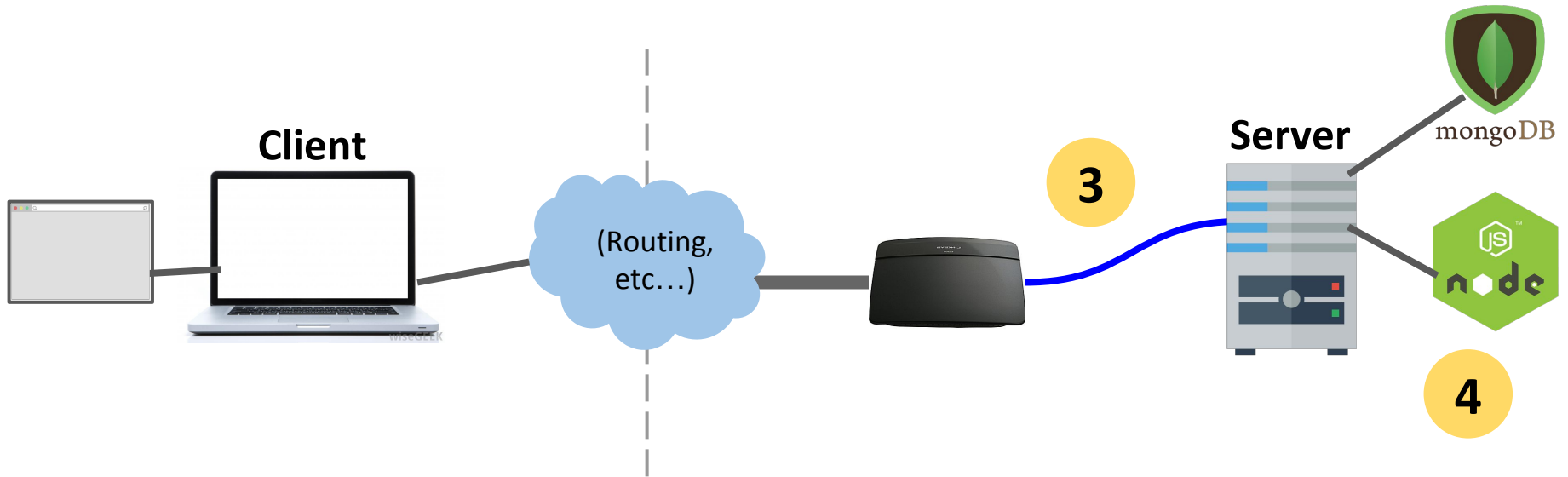
3. The server computer that is located at abc.com receives the HTTP GET request

4. The server computer gives the NodeJS server process the HTTP GET request message

Review system

Server

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

4

"Does the 'public' folder have an 'index.html' file?

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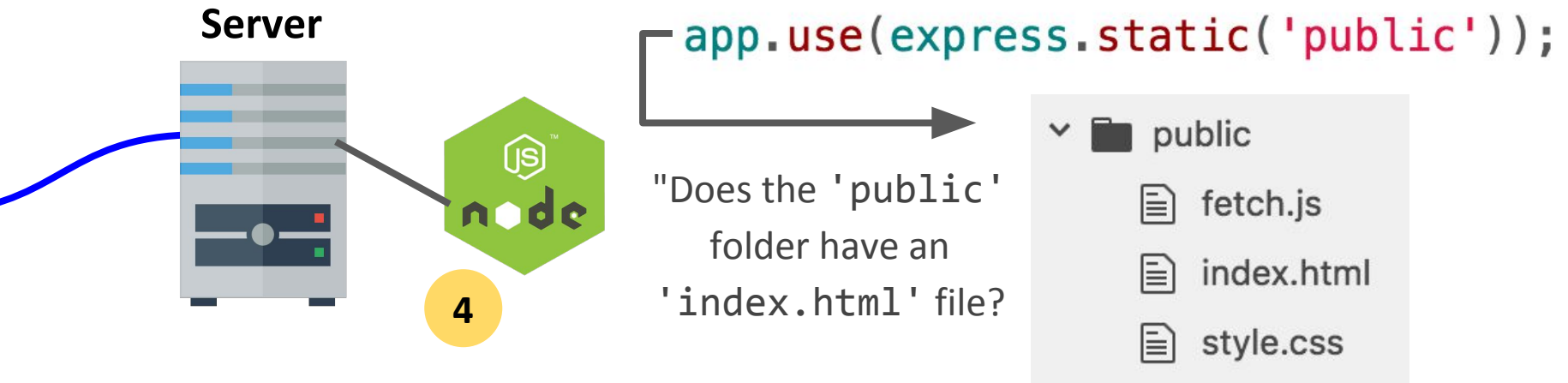
Our NodeJS server code has `app.use(express.static('public'));` so it will first look to see if an `index.html` file exists in the `public` directory.

Review system

Server

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

"Does the `public` folder have an `index.html` file?
Yes

- ∨ 📁 public
  - 📄 fetch.js
  - 📄 index.html
  - 📄 style.css

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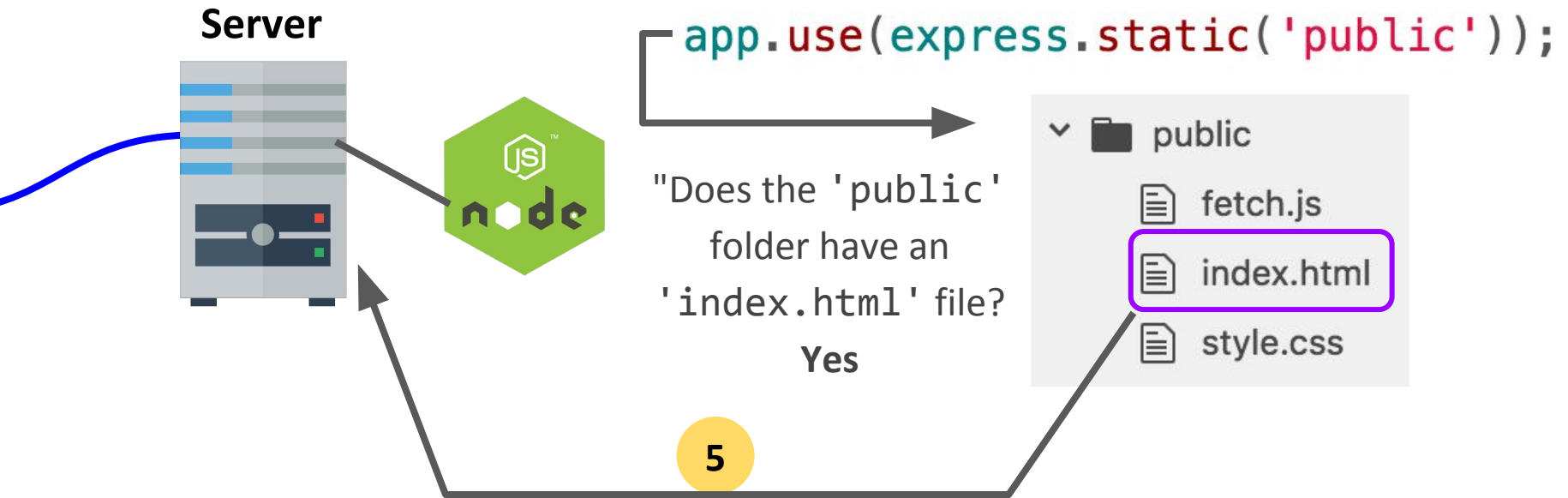
5. Since there is an index.html file, our NodeJS server will respond with the index.html file

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>
```

Client

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Dictionary lookup</title>
    <link rel="stylesheet" href="style.css">
    <script src="fetch.js" defer></script>
```
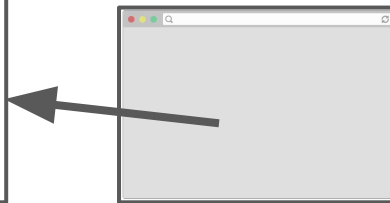
8. In rendering the HTML, the browser sees it needs style.css and fetch.js

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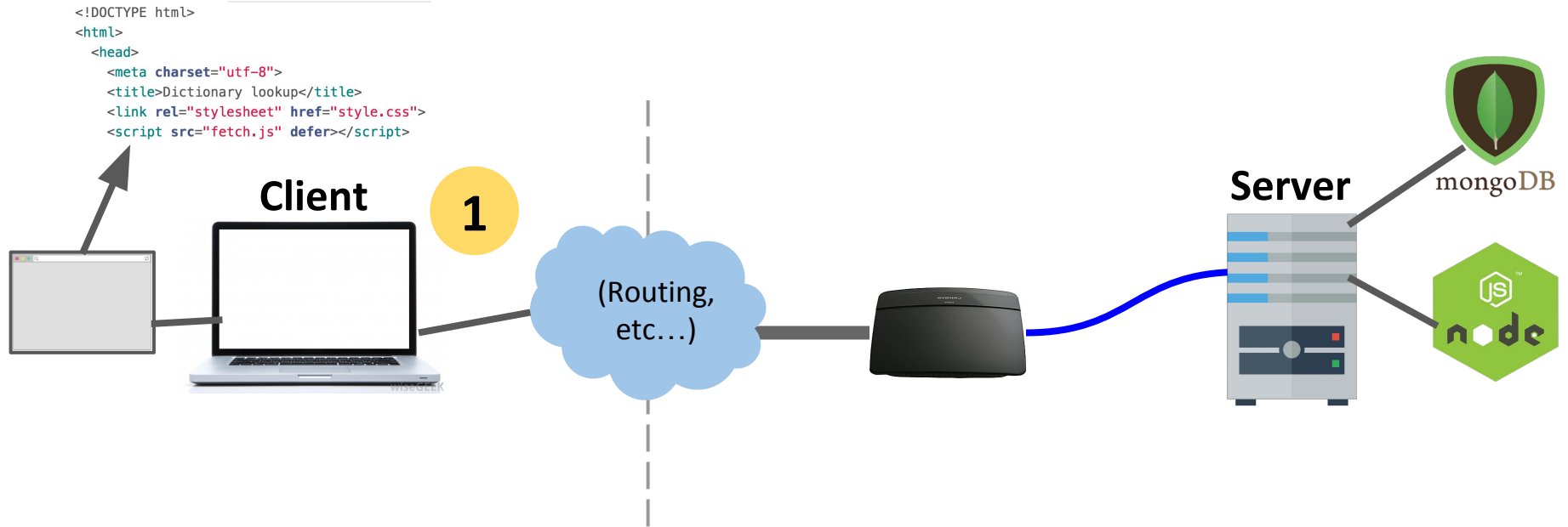
1. So the browser makes two more HTTP GET requests:
   - One for style.css
   - One for script.js

Review system

Client

(Routing, etc…)
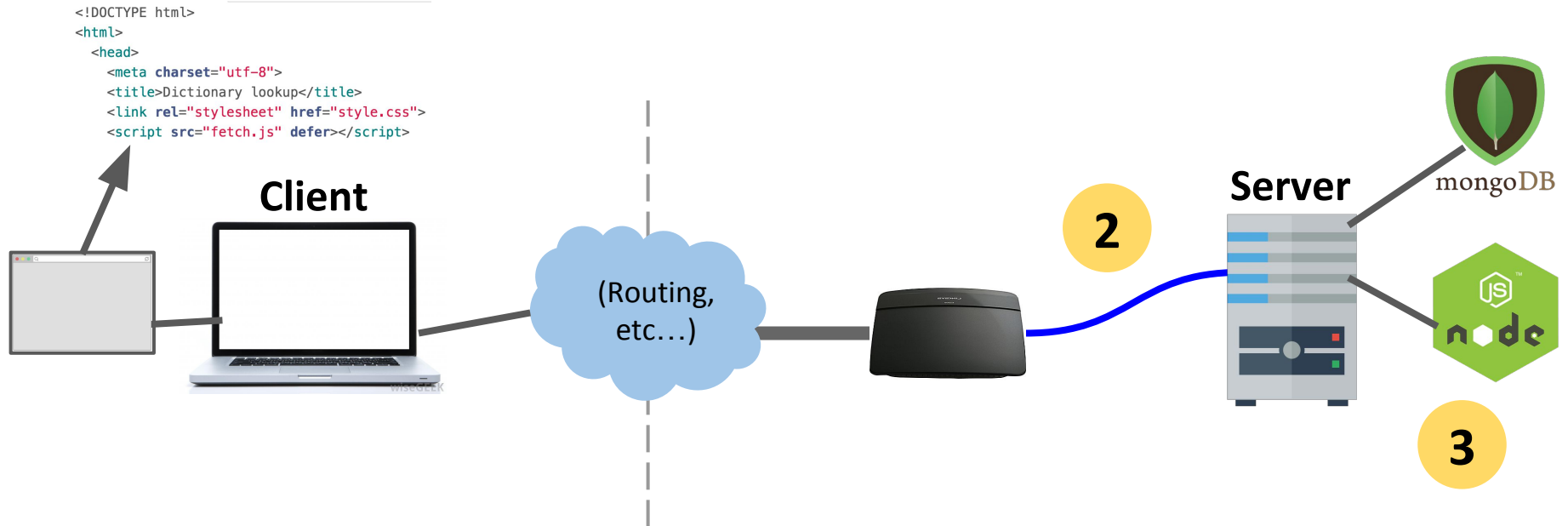
Server

2. These GET requests get routed to the server computer

3. The server computer sends the GET requests to our NodeJS process

Review system

Server

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

```
v 📁 public
    📄 fetch.js
    📄 index.html
    📄 style.css
```

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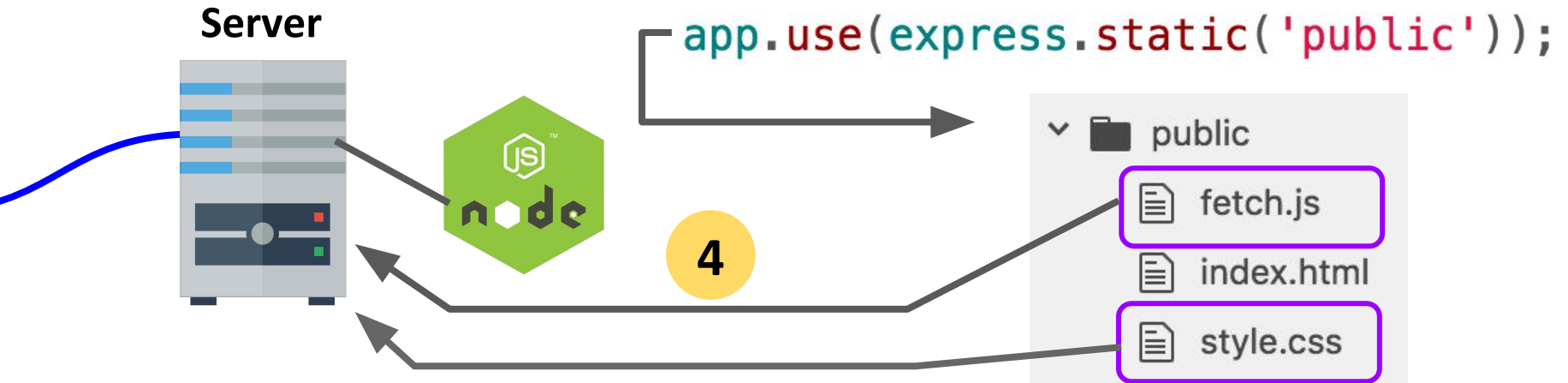
4. Our NodeJS server code finds fetch.js and style.css in the public directory, so it responds with those files

Review system
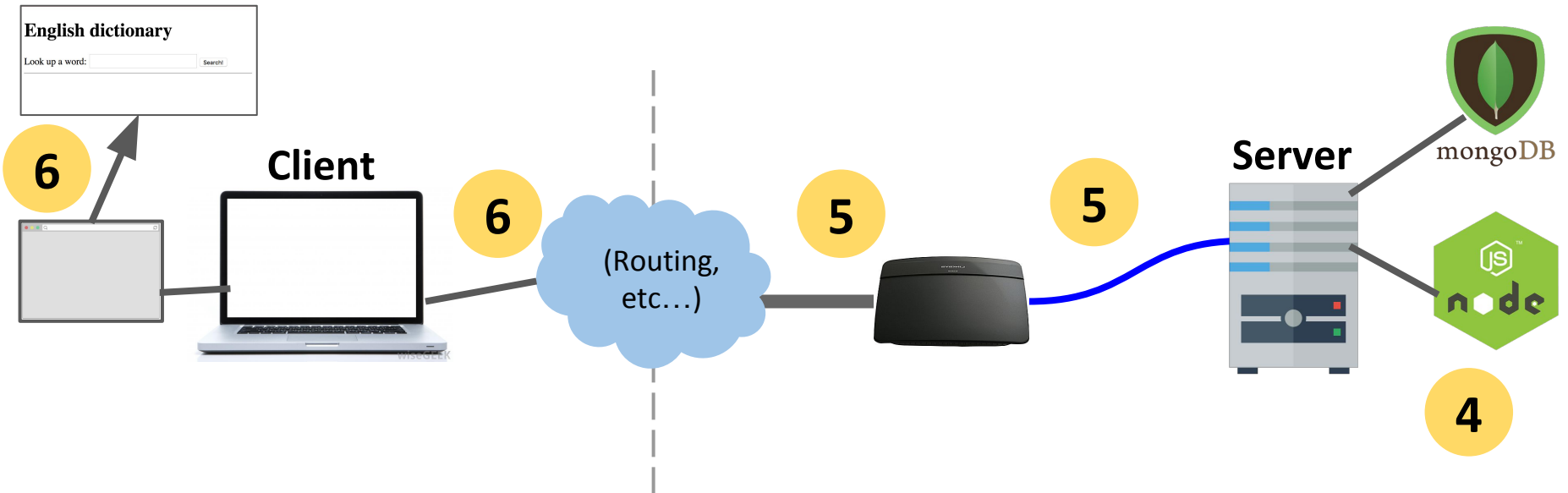
4. Our Node server program replies with the style.css and fetch.js files

5. The server computer sends these files back to the client

6. The browser receives the files and continues rendering index.html

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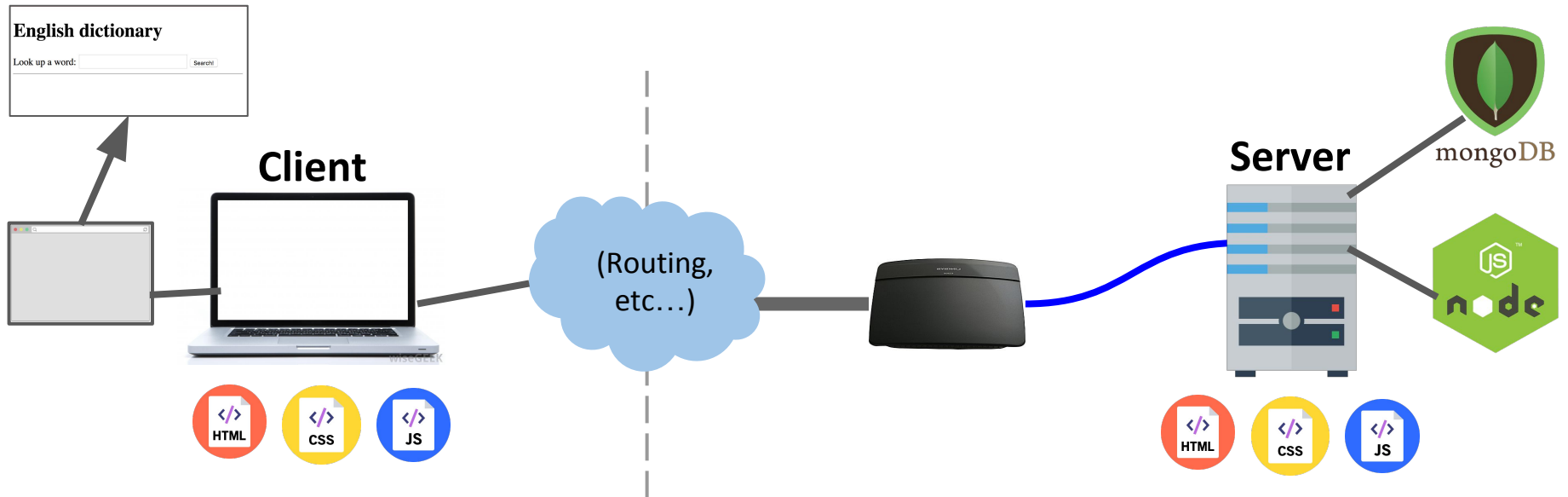
In this picture, there are **two copies** of index.html, style.css, and fetch.js:

- 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: `cat` [Search!]

1

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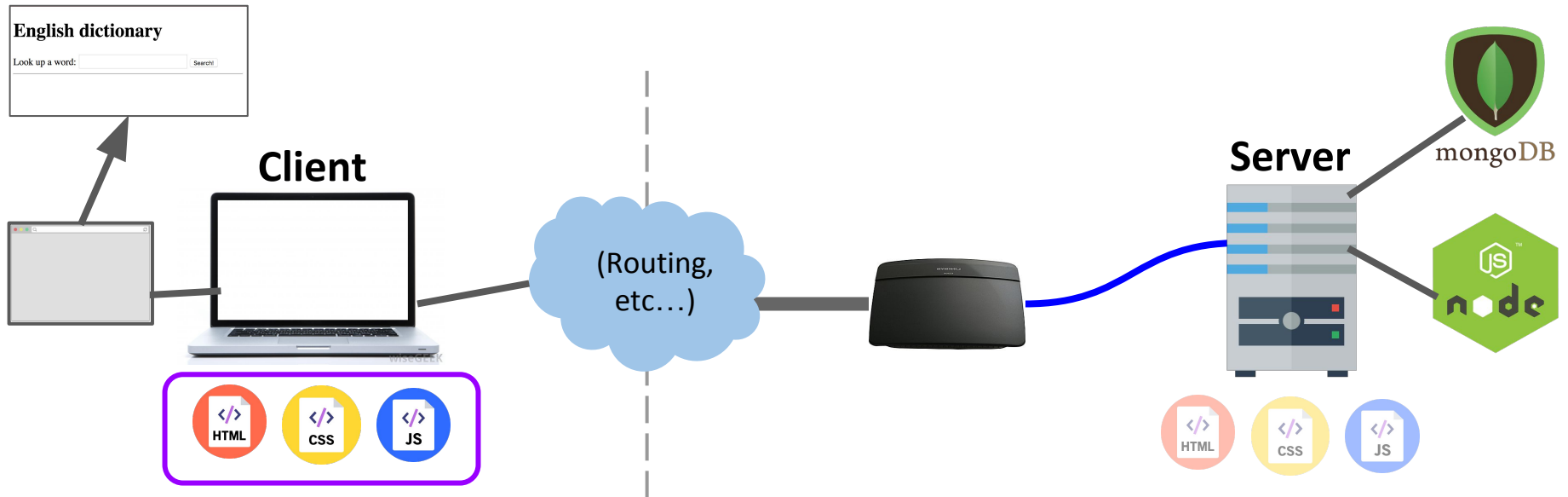const searchForm = document.querySelector('#search');
searchForm.addEventListener('submit', onSearch);
```

Client

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

English dictionary

Look up a word: cat   Search!

Client

2

```
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


```
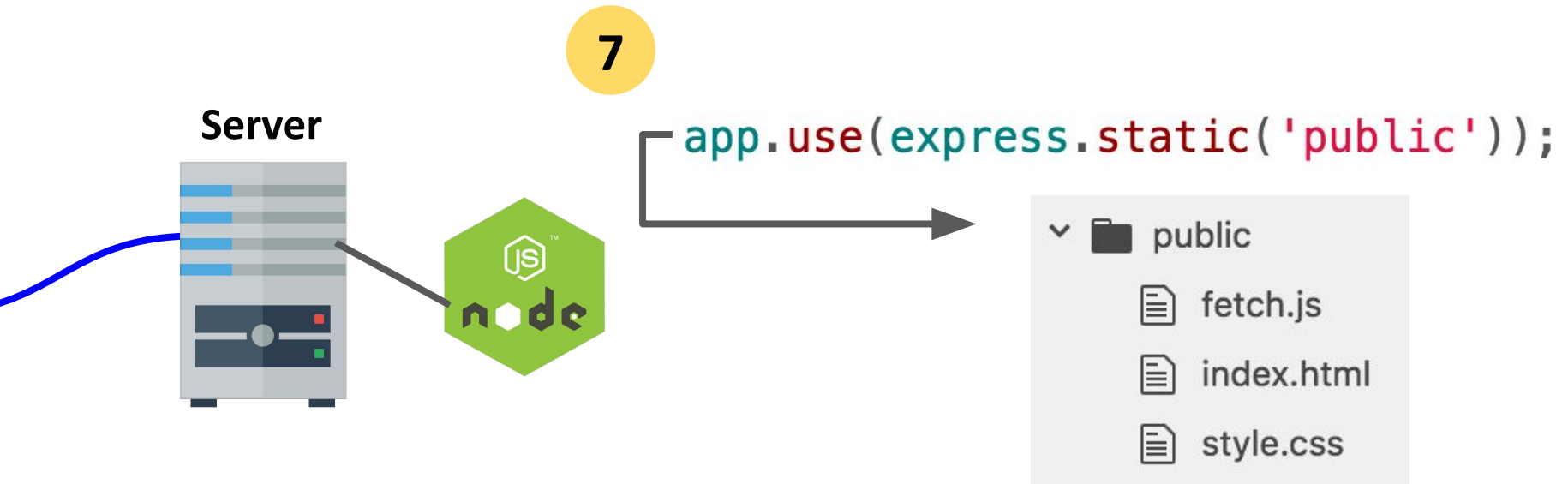fetch('/lookup/' + word);
```

Client

4

(Routing, etc...)

Server

mongoDB
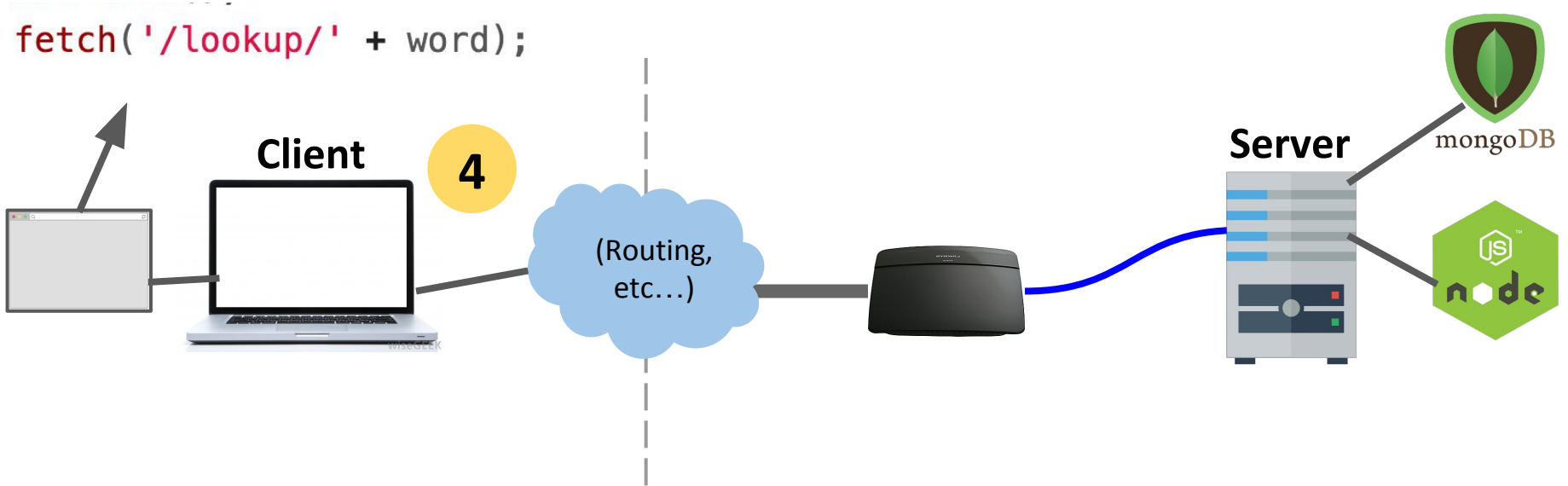
node

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

Review system


```
fetch('/lookup/' + word);
```

Client

(Routing, etc…)

5

Server

mongoDB

node

7
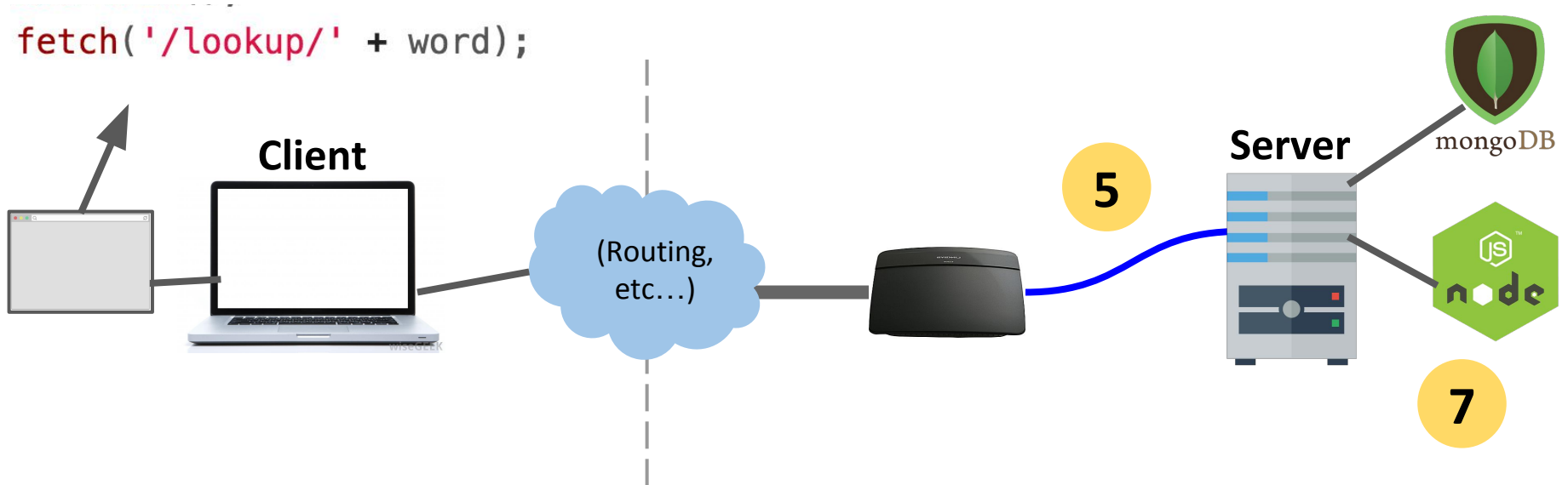
5. These GET requests get routed to the server computer

6. The server computer sends the GET requests to our NodeJS process

Review system

Server

⑦

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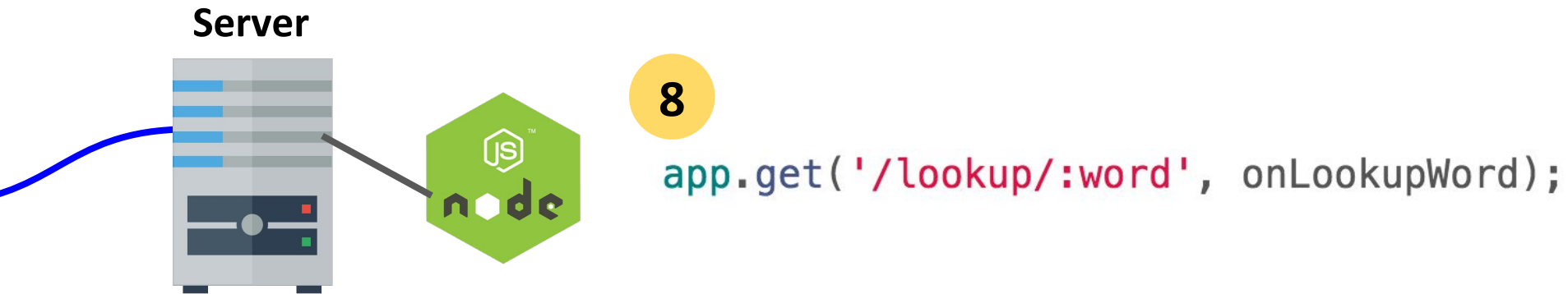app.use(express.static('public'));
```

```
∨ ▪ public
    📄 fetch.js
    📄 index.html
    📄 style.css
```

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

Review system

Server

8

```
app.get('/lookup/:word', onLookupWord);
```

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
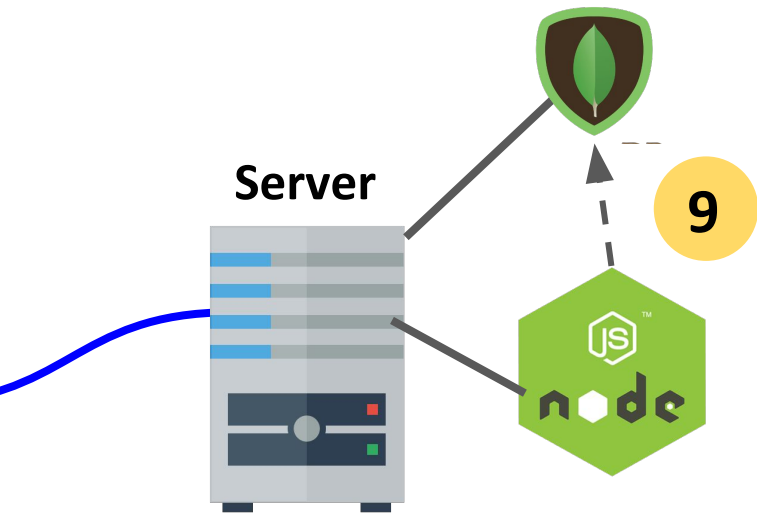

```
async function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  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);
```

9. We get the definition from the MongoDB database.

Review system


```
fetch('/lookup/' + word);
```

Client

12

(Routing, etc…)

11

Server

mongoDB

node

10
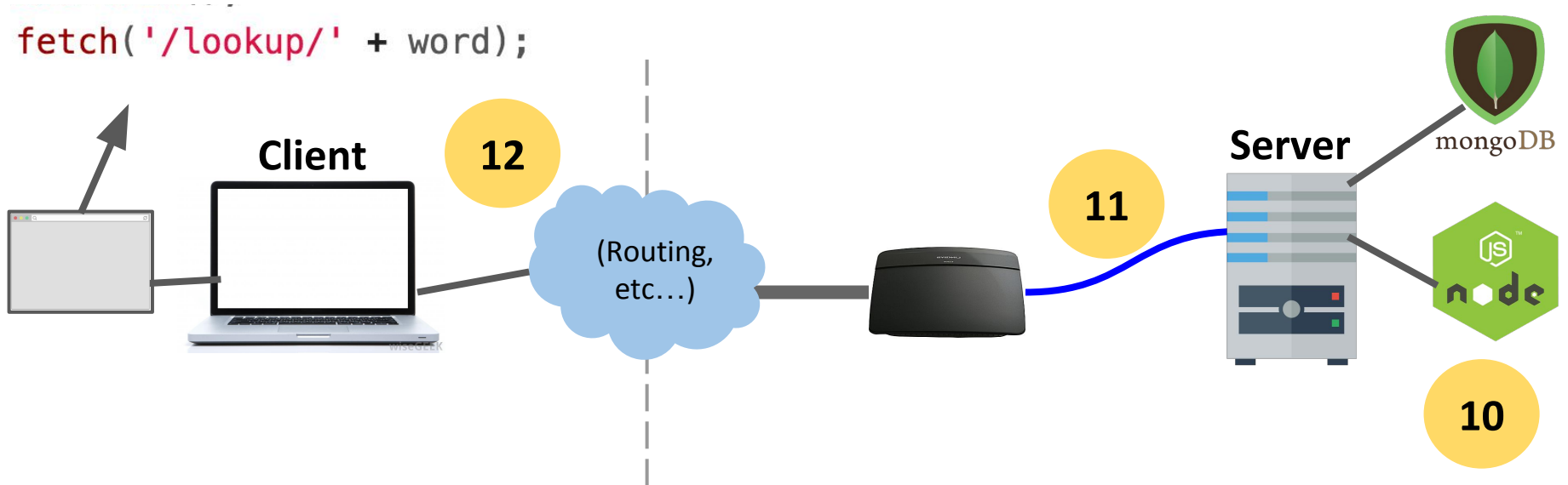
10. Our Node server program replies with JSON

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: cat   Search!

The definition of **cat** is:

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.

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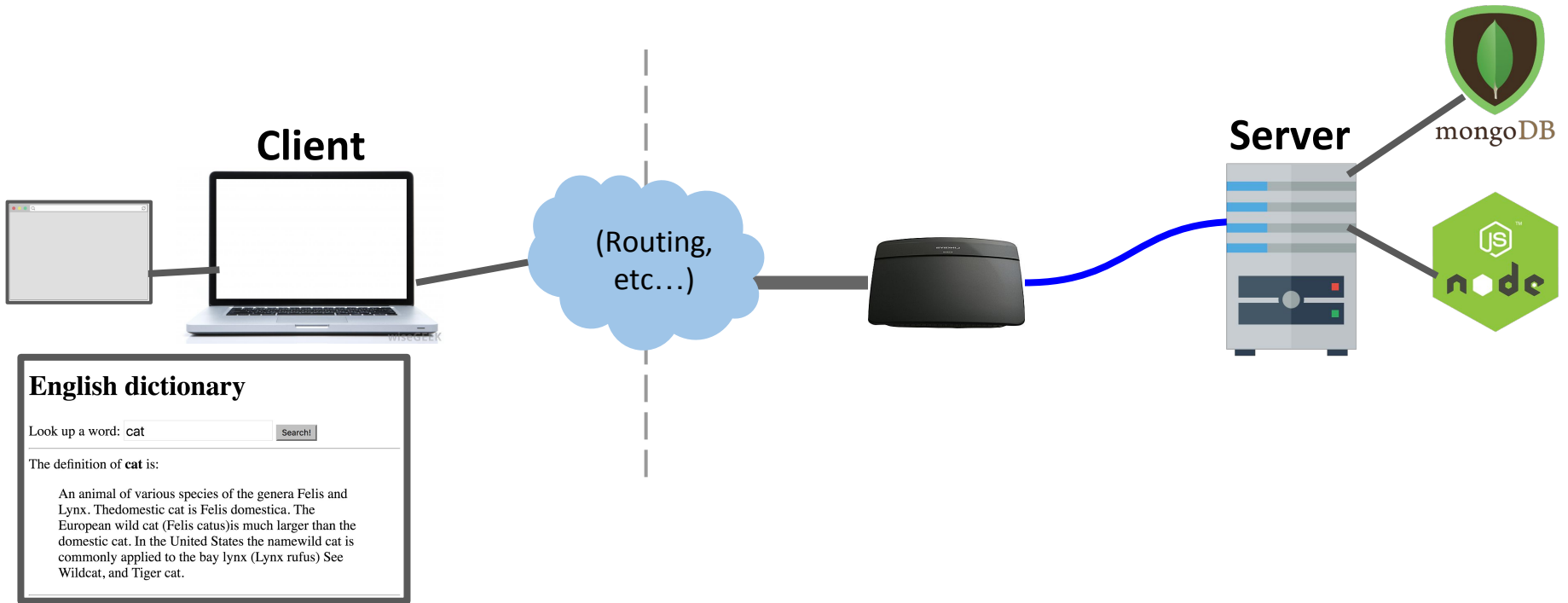
The server **generated** the JSON with the word and definition.
The client **consumed** the JSON with the word and definition.

Another example: E-cards

Example: E-cards

We'll be creating an e-card app, whose data is saved in a MongoDB database:

Example: E-cards

When the user loads to an index page, we want to present them with an E-Card Maker UI

Example: E-cards

When the user has created an e-card, we want it accessible via URL of this form:

/id/<ecard_id>

Step 1: Saving data

We'll need to save 3 pieces of data:

- Card style

- Card message

- A unique id for each card

Choose a style:

| Happy Graduation ⇕ |

Write a message:

Hi Pooh,

🎉 Congrats!!! 🎉

<3 Piglet

Create card

Example: E-card saving data

```javascript
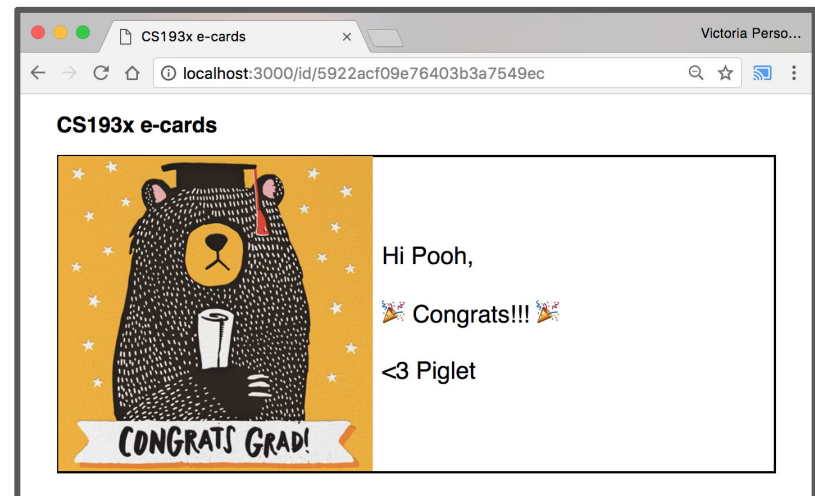async function onSaveCard(req, res) {
  const style = req.body.style;
  const message = req.body.message;
  const doc = {
    style: style,
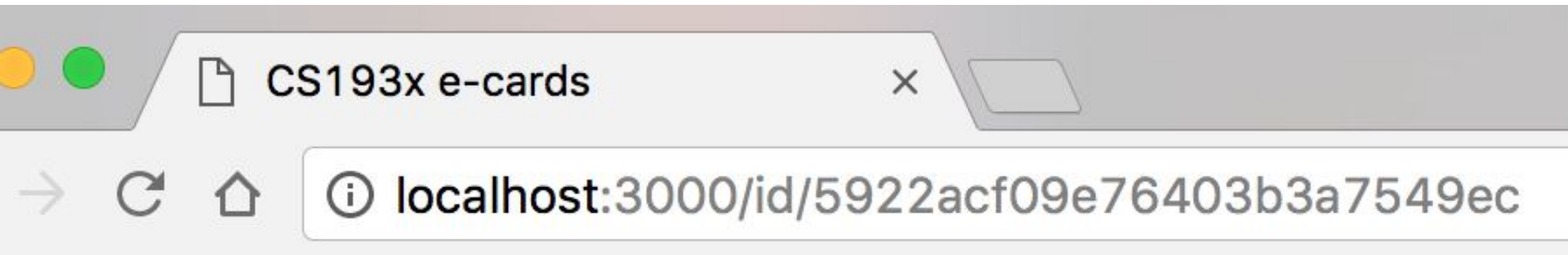    message: message
  };

  const collection = db.collection('card');
  const response = await collection.insertOne(doc);

  res.json({ cardId: response.insertedId });
}
app.post('/save', jsonParser, onSaveCard);
```

Step 2: Loading a card

When the user has created an e-card, we want it accessible via URL of this form:

`/id/<ecard_id>`

Web app architecture

How do we implement an app so that to show a different view depending on the URL path?

Create-a-card:
localhost:3000/
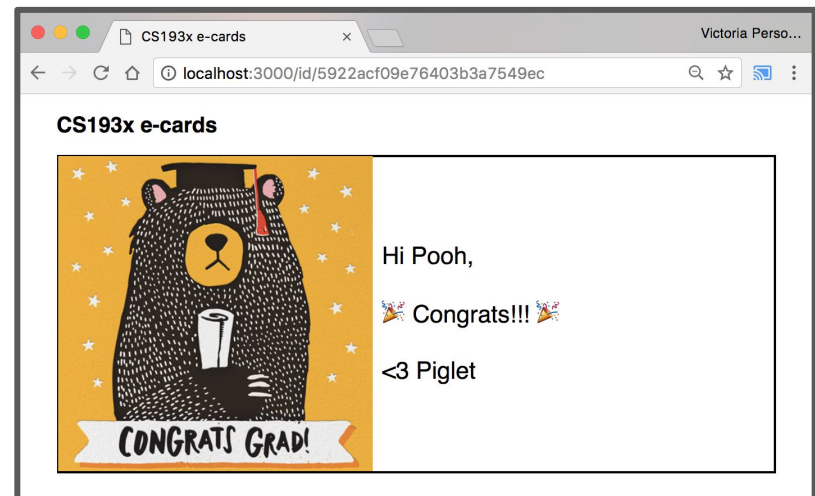
View card:
localhost:3000/id/**<card_id>**

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:**
    - **CS193X:** We will show you how to do this
2. **Single-page application:**
    - **CS193X:** We will show you how to do this
3. Combination of 1 and 2 ("**Isomorphic**" / "**Universal**")
    - This is probably the most common technique
    - **CS193X:** 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)
    - **CS193X:** 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**: Jade, Handlebars, etc

HTML template

{{}}

Server

mongoDB

node

Handlebars: Template engine

We will be showing the [Handlebars](#) template engine:

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

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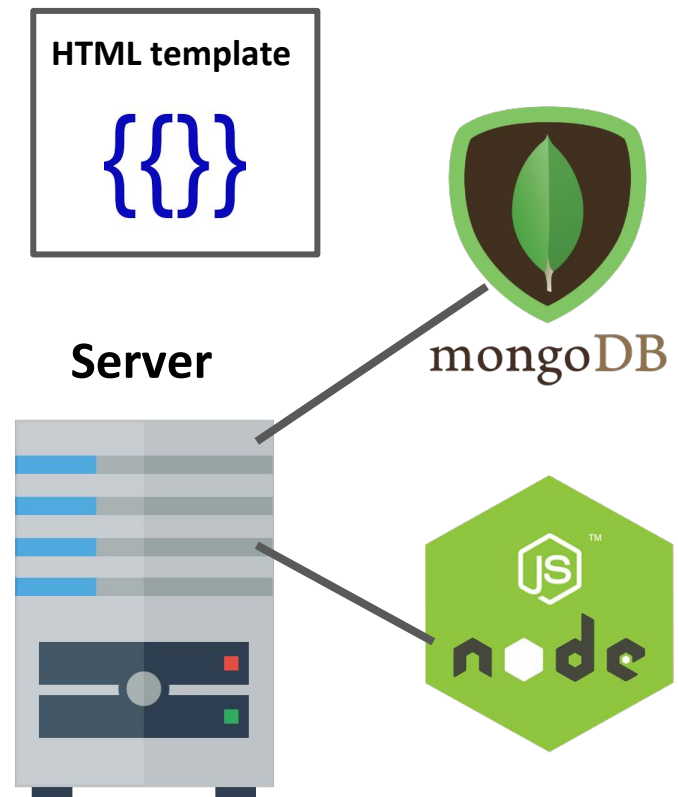<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

Handlebars and NodeJS

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

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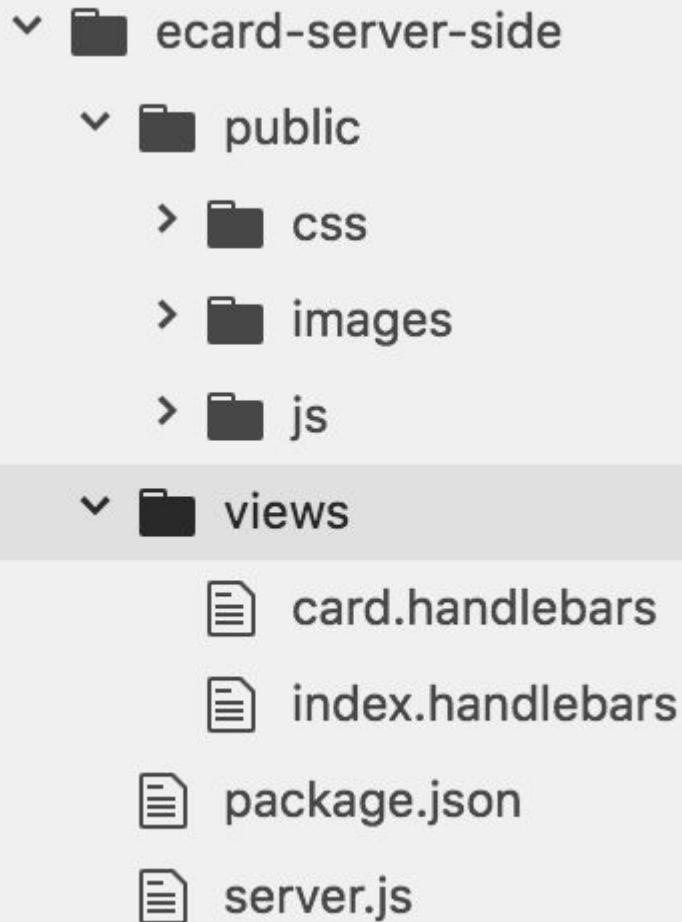const exphbs  = require('express-handlebars');
...

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

E-cards: Server-side rendering

ecard-server-side
- public
  - css
  - images
  - js
- views
  - card.handlebars
  - index.handlebars
- package.json
- server.js

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:

```html
<!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:

```html
<!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:

```html
<!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:

```javascript
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);
```

# 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 `/id/`**<card_id>** would still return e.g. index.html.

- The client JavaScript parses the URL to get the route parameters and initialize the app.

GET localhost:3000/

index.html

# 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 `/id/`**<card_id>** would still return e.g. index.html.

- The client JavaScript parses the URL to get the route parameters and initialize the app.

```
GET id/2338242394
```
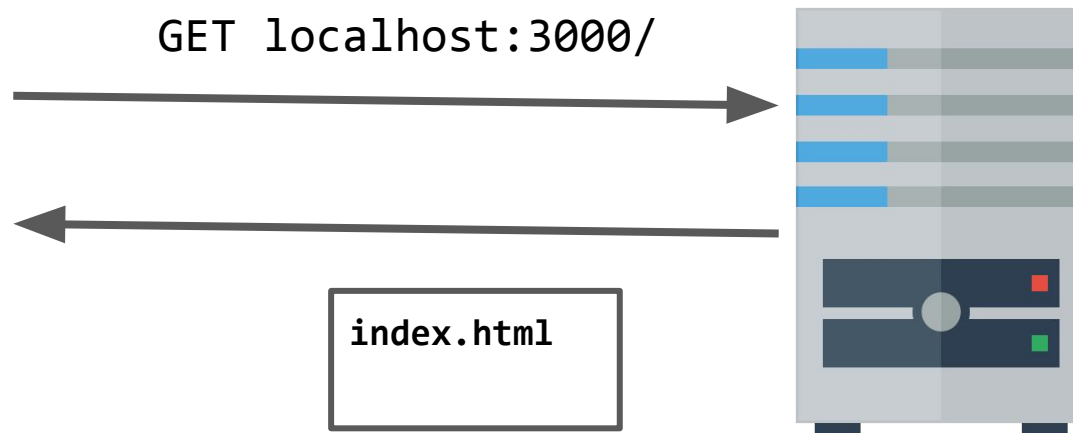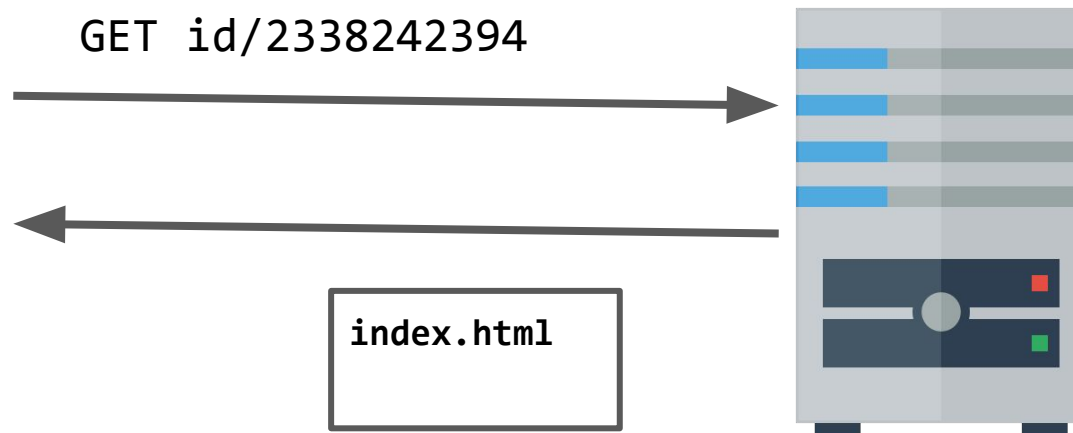
`index.html`

Single page web app

Another way to think of it:

- You configure several JSON routes for your server
- You embed all your view into index.html
- You use JavaScript to switch between the views

`GET id/2338242394`

`index.html`