CS193X: Web Programming Fundamentals

Spring 2017

Victoria Kirst
(vrk@stanford.edu)

CS193X schedule

Today

- MongoDB
- Servers and MongoDB

Friday

- Web application architecture
- Authentication

MongoDB installation

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

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

MongoDB

Database definitions

A **database** (**DB**) is an organized collection of data.

- In our dictionary example, we used a JSON file to store the dictionary information.
- By this definition, the JSON file can be considered a database.

A **database management system** (**DBMS**) is software that handles the storage, retrieval, and updating of data.

- Examples: MongoDB, MySQL, PostgreSQL, etc.
- Usually when people say "**database**", they mean data that is managed through a DBMS.

MongoDB

MongoDB: A popular open-source DBMS

- A *document-oriented* database as opposed to a *relational* database

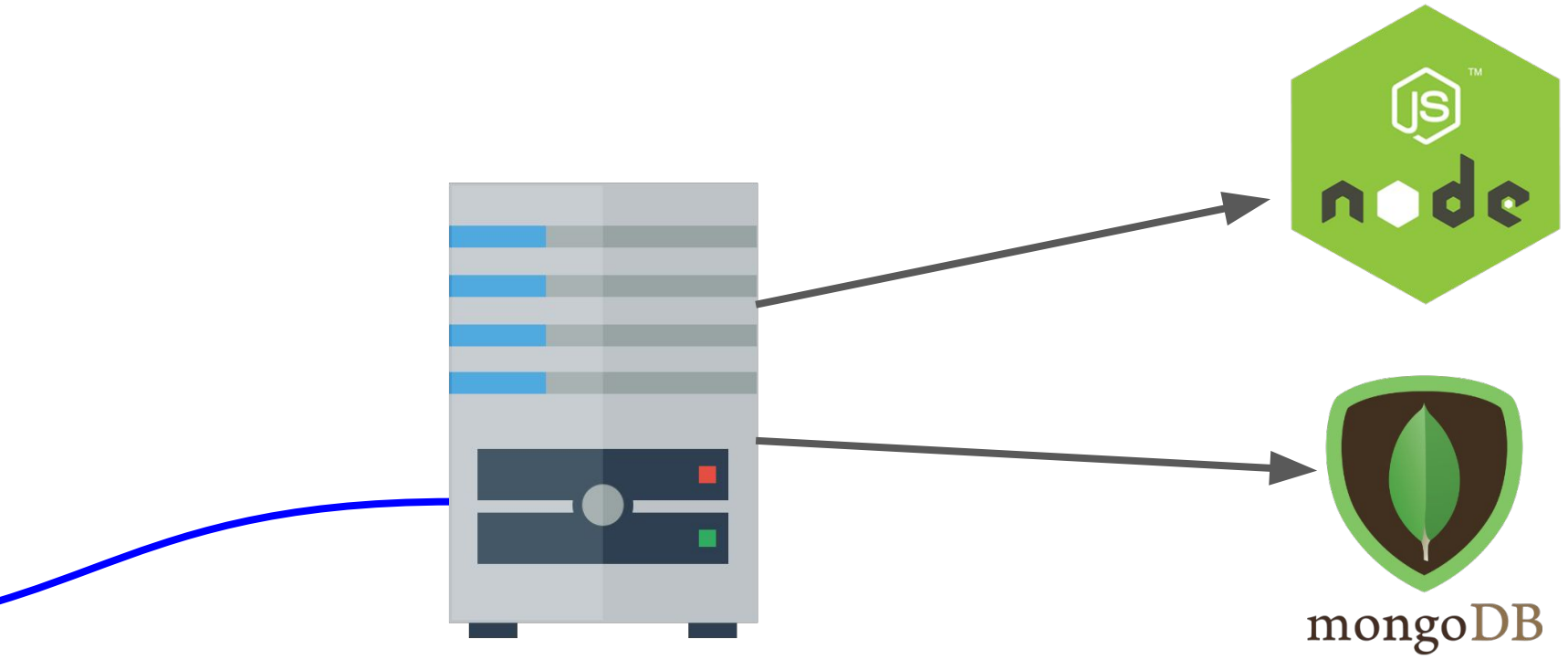
Relational database:

Name	School	Employer	Occupation
Lori	null	Self	Entrepreneur
Malia	Harvard	null	null

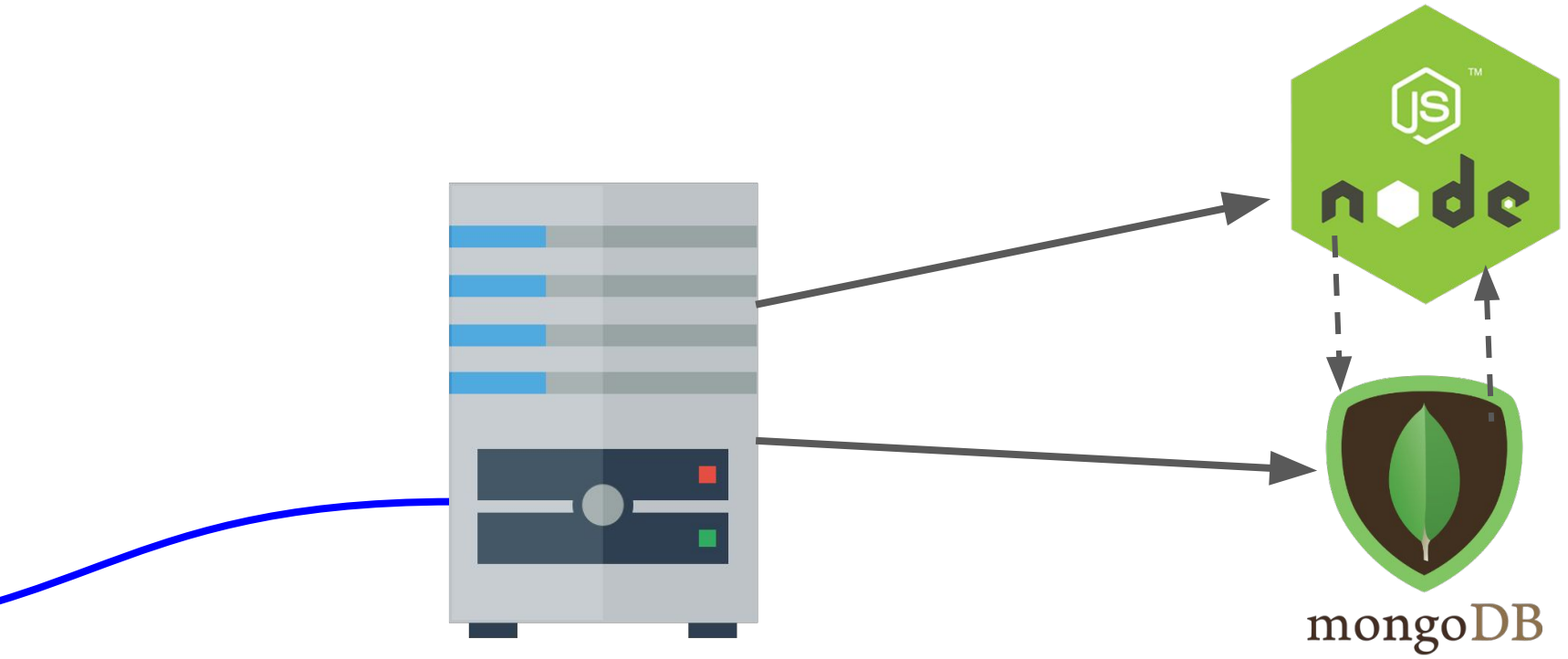
Relational databases have fixed schemas; document-oriented databases have flexible schemas

Document-oriented DB:

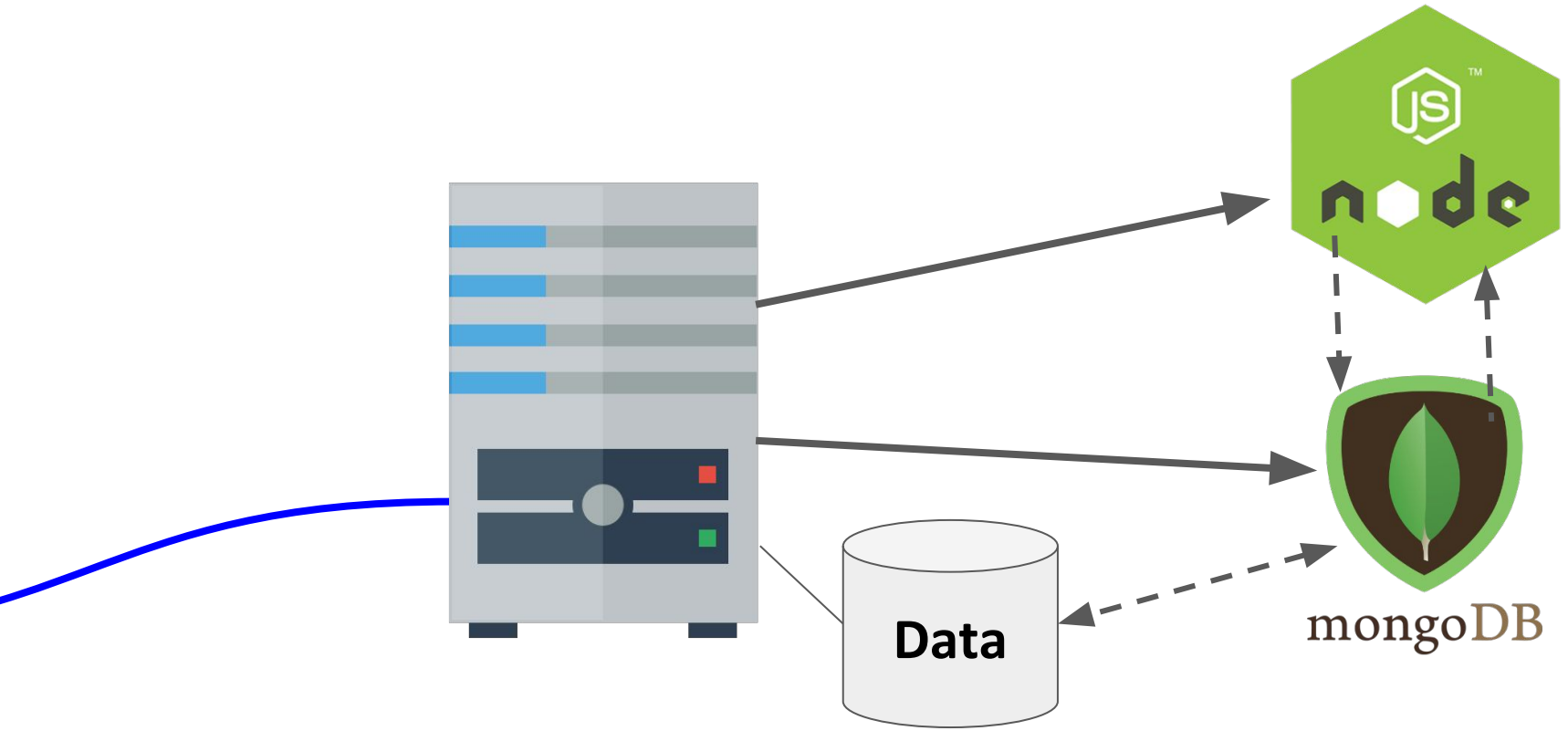
```
{
  name: "Lori",
  employer: "Self",
  occupation: "Entrepreneur"
}
{
  name: "Malia",
  school: "Harvard"
}
```



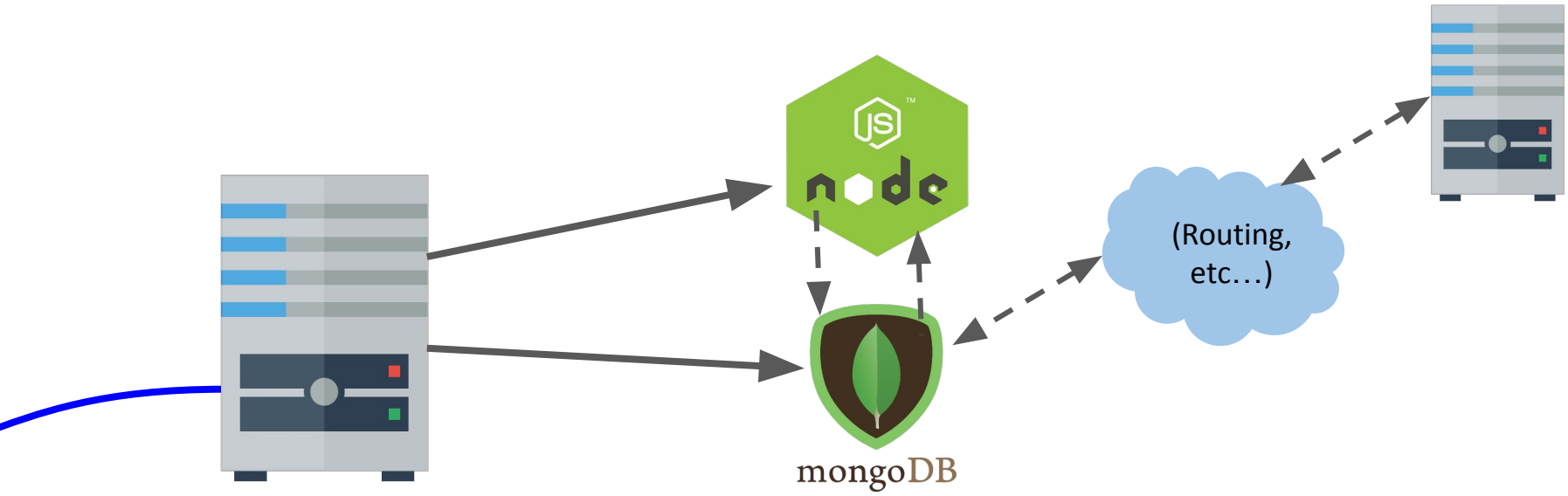
MongoDB is another **software program** running on the computer, alongside our NodeJS server program.
It is also known as the **MongoDB server**.



There are MongoDB libraries we can use in NodeJS to communicate with the MongoDB Server, which reads and writes data in the database it manages.

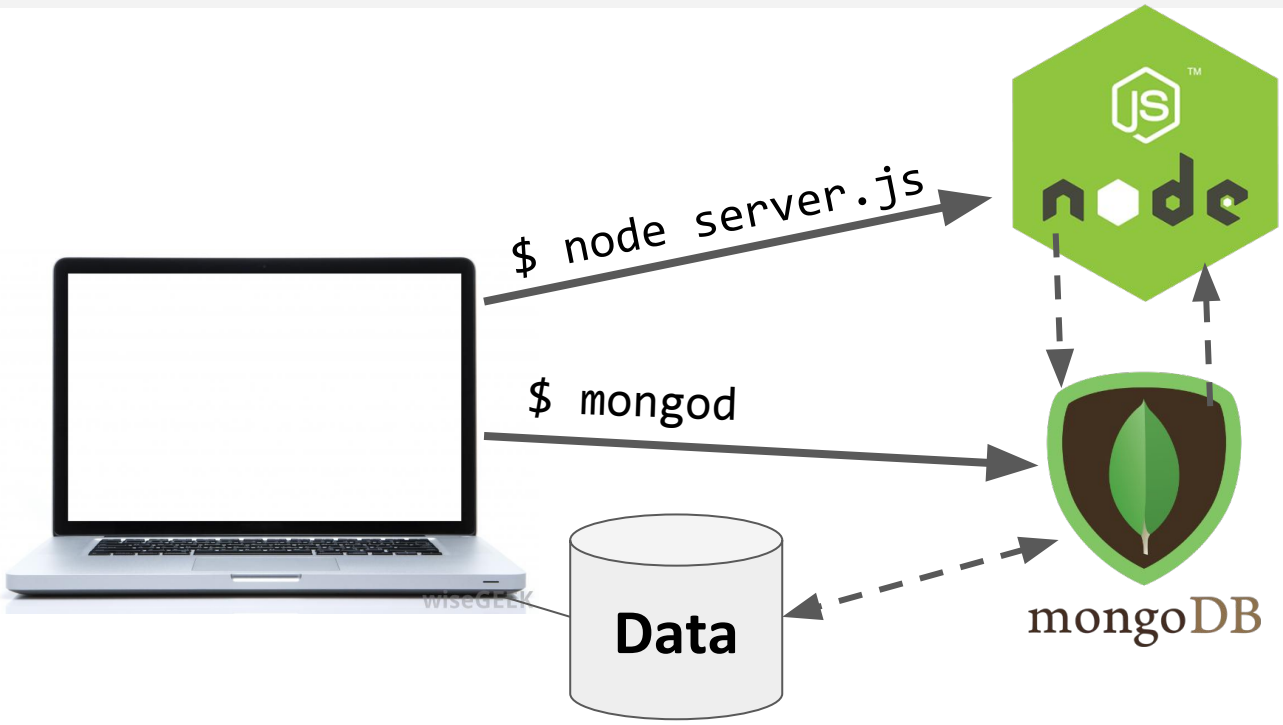


The database the MongoDB Server manages might be local to the server computer...



Or it could be stored on other server computer(s)
("cloud storage").

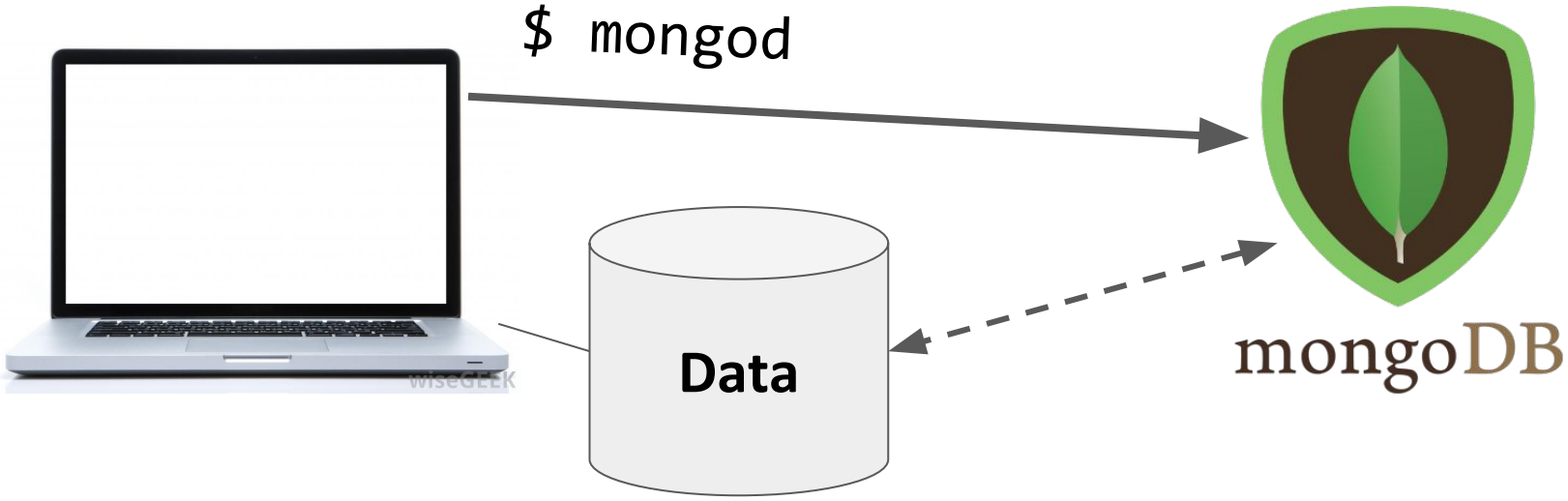
System overview



For development, we will have 2 processes running:

- node will run the main server program on port 3000
- **mongod will run the database server on a port 27017**

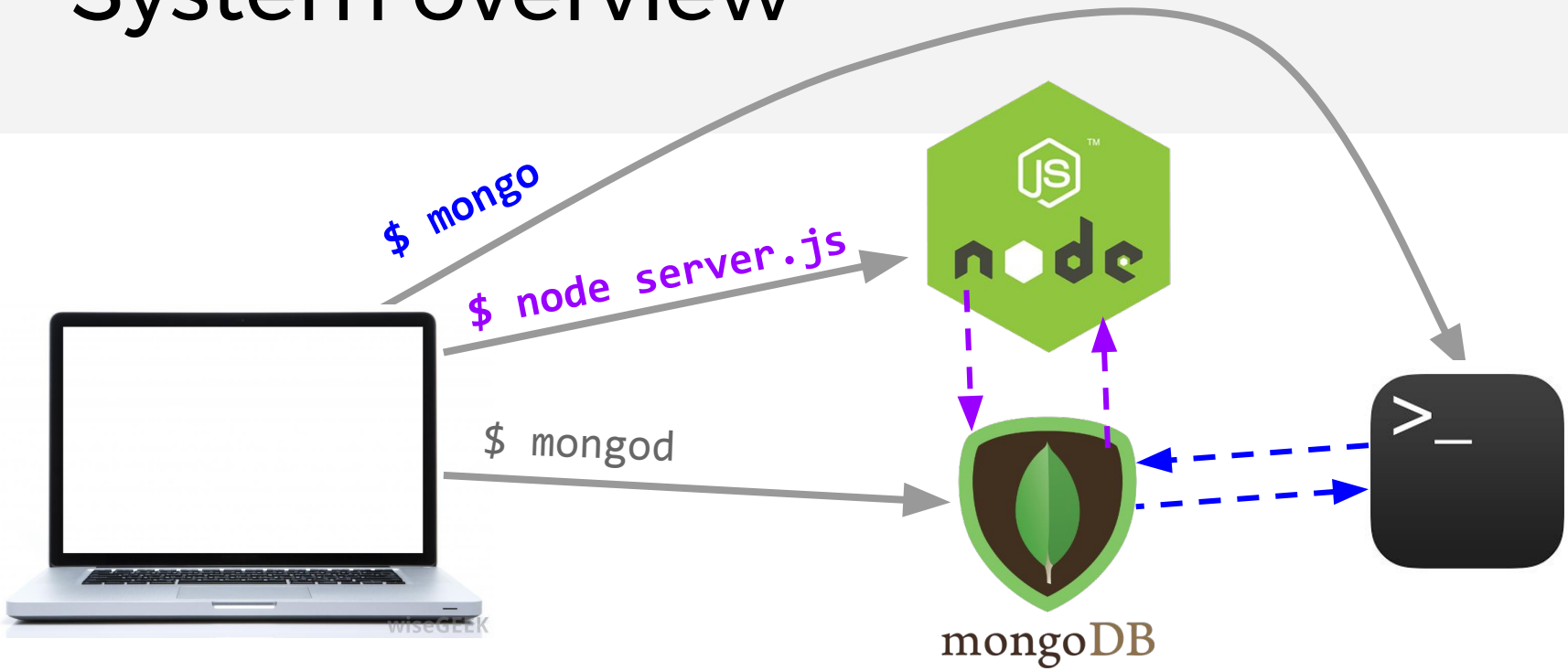
System overview



The mongod server will be bound to port 27017 by default

- The mongod process will be listening for messages to manipulate the database: insert, find, delete, etc.

System overview



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

- NodeJS libraries
- mongo command-line tool

MongoDB concepts

Database:

- A container of MongoDB **collections**

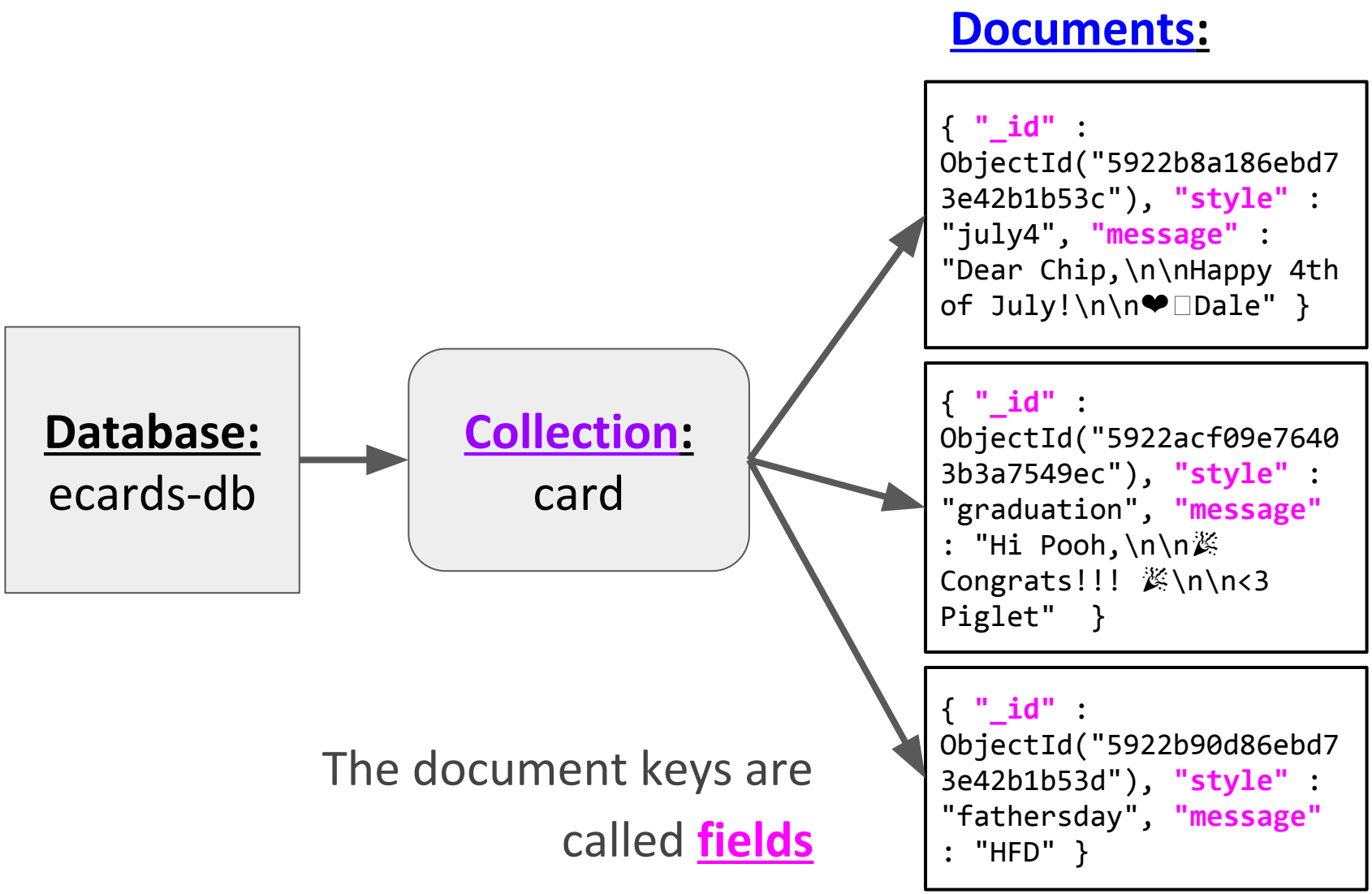
Collection:

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

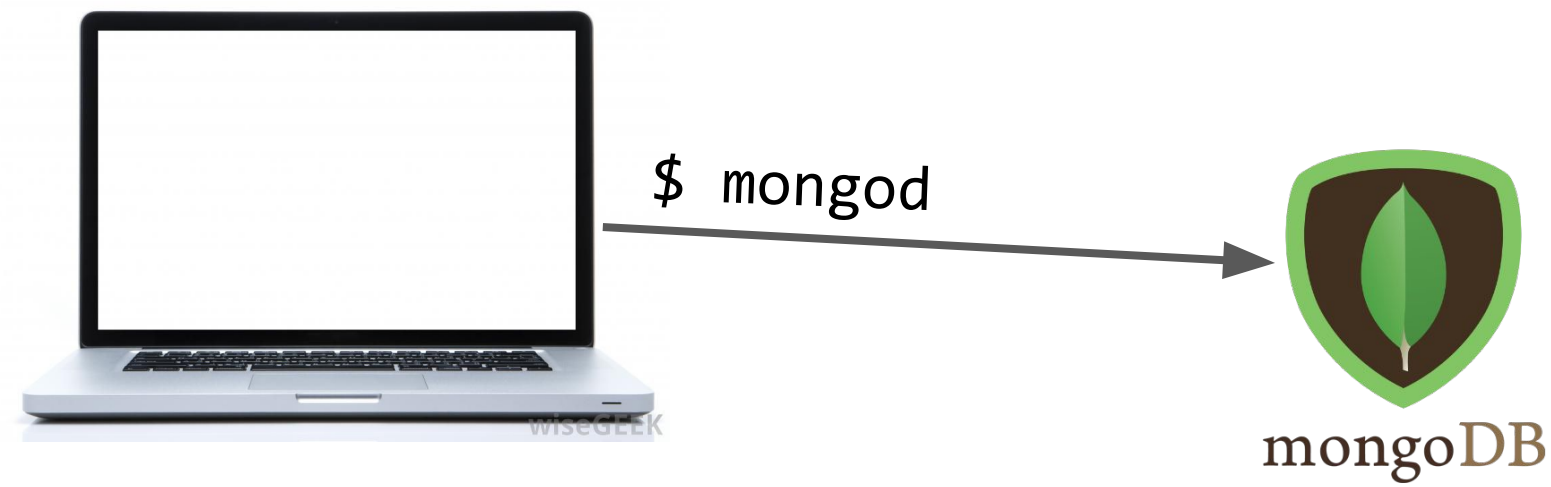
Document:

- 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



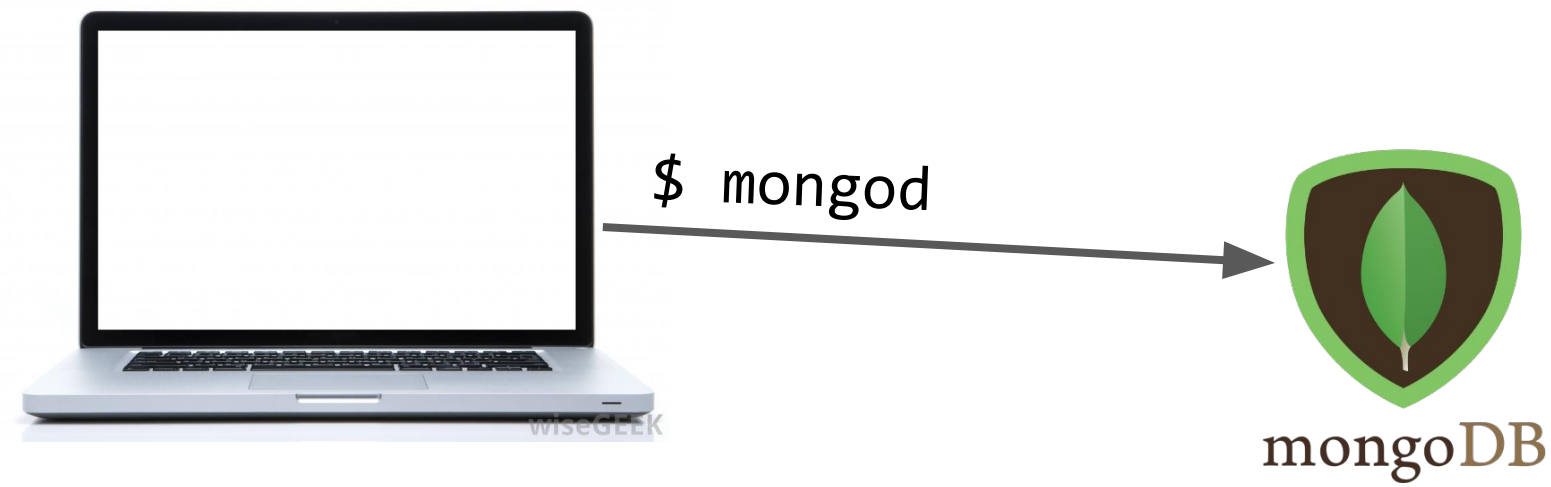
mongod: Database process



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

mongo shell

When should you use the `mongo` shell?

- Adding test data
- Deleting test data

NodeJS and MongoDB

NodeJS

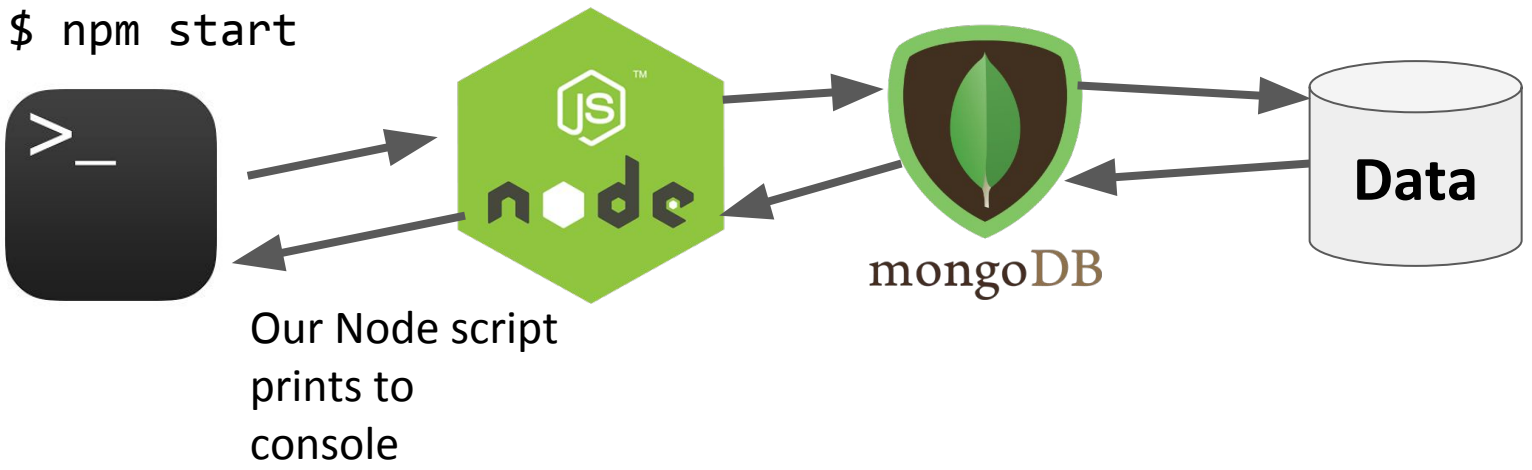
Recall: NodeJS can be used for writing scripts in JavaScript, completely unrelated to servers.

simple-script.js

```
function printPoem() {
  console.log('Roses are red,');
  console.log('Violets are blue,');
  console.log('Sugar is sweet,');
  console.log('And so are you.');
  console.log();
}

printPoem();
printPoem();
```

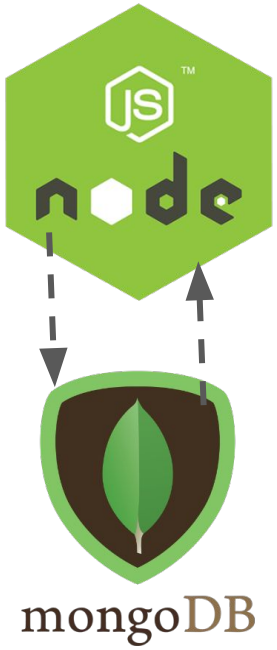
Mongo JS scripts

Before we start manipulating MongoDB from the server, let's just write some JavaScript files that will query MongoDb.



No web servers are involved yet!

NodeJS Driver



To read and write to the MongoDB database from Node we'll be using the **'mongodb'** library.

We will install via npm:

```
$ npm install --save mongodb
```

On the MongoDB website, this library is called the "MongoDB NodeJS Driver"

`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

Getting a Db object

You can get a reference to the database object by using the `MongoClient.connect(`***url***`, `***callback***`)` function:

- ***url*** is the connection string for the MongoDB server
- ***callback*** is the function invoked when connected
  - ***database*** parameter: the Db object

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

let db = null;
MongoClient.connect(MONGO_URL, function (err, database) {
  db = database;
});
```

Connection string

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

- The URL is to a MongoDB server, which is why it begins with mongodb:// and not http://
- The MongoDB server is running on our local machine, which is why we use `localhost`
- The end of the connection string specifies the database name we want to use.
  - If a database of that name doesn't already exist, it will be created the first time we write to it.

MongoDB Connection string format

Callbacks and Promises

Every asynchronous MongoDB method has two versions:

- Callback
- Promise

The callback version of `MongoClient.connect` is:

```
let db = null;
MongoClient.connect(MONGO_URL, function (err, database) {
  db = database;
});
```

Callbacks and Promises

Every asynchronous MongoDB method has two versions:

- Callback
- Promise

The Promise version is:

```
let db = null;
function onConnected(err, database) {
  db = database;
}
MongoClient.connect(MONGO_URL)
  .then(onConnected);
```

Callbacks and Promises

Every asynchronous MongoDB method has two versions:

- Callback
- Promise

The Promise + async/await version is:

```
let db = null;
async function main() {
  db = await MongoClient.connect(MONGO_URL);
}
main();
```

Using a collection

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

`const coll = db.collection(`***collectionName***`);`

- Obtains the collection object named ***collectionName*** and stores it in `coll`
- You do not have to create the collection before using it
  - It will be created the first time we write to it
- This function is **synchronous**

collection.insertOne (Callback)

`collection.insertOne(`***`doc, callback`***`);`

- Adds one item to the collection
- ***doc*** is a JavaScript object representing the key-value pairs to add to the collection
- The ***callback*** fires when it has finished inserting
  - The first parameter is an error object
  - The second parameter is a result object, where `result.`**`insertedId`** will contain the id of the object that was created

Callback version

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

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:

```
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

```
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

```
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

```
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:

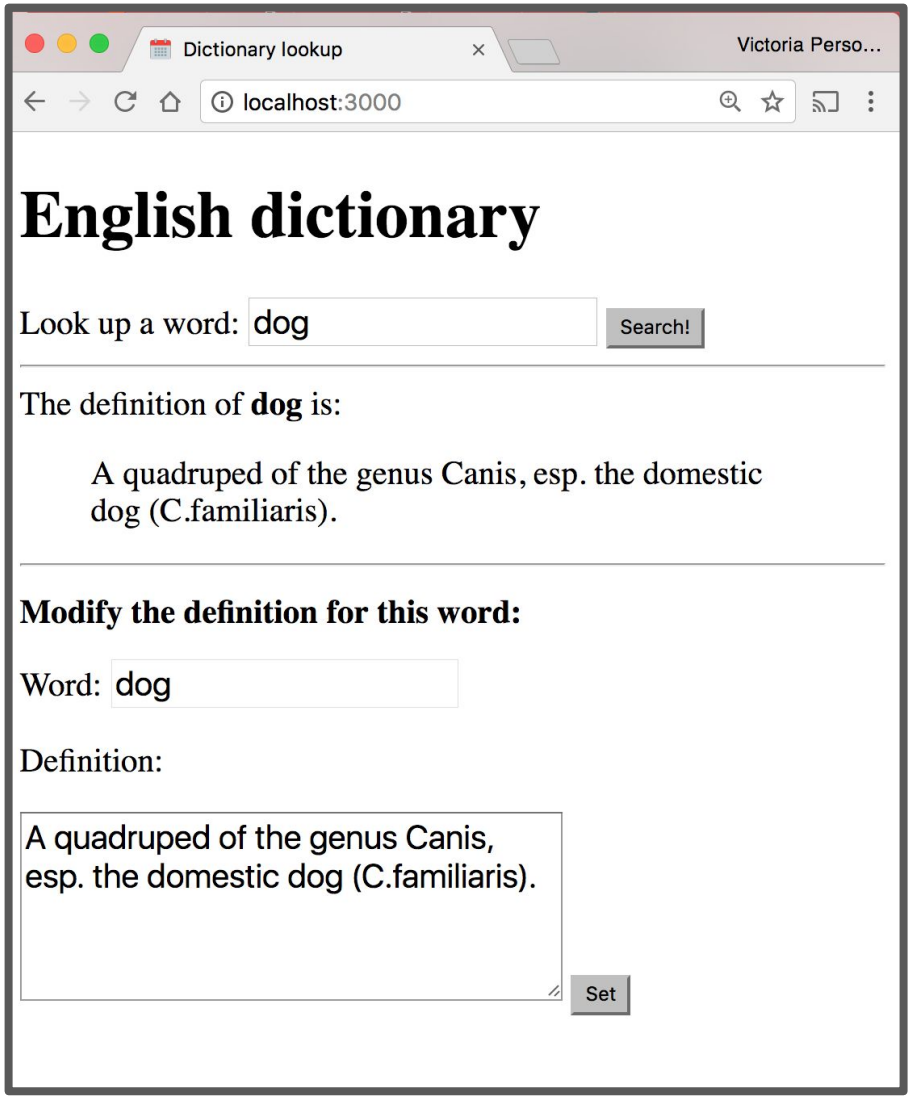
- Querying
  - Query selectors and projection operators
  - `db.collection('inventory').find({ qty: { $lt: 30 } });`
- Updating
  - Update operators

```
db.collection('words').updateOne(
  { word: searchWord },
  { $set: { definition: newDefinition }})
```

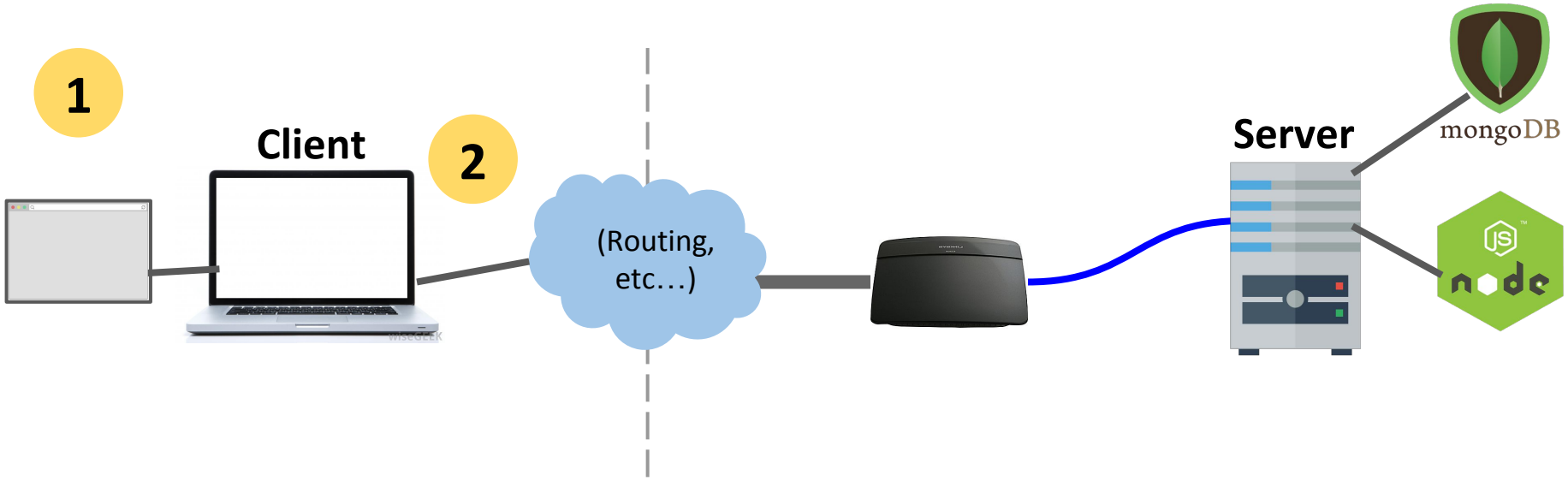
Using MongoDB in a server

Dictionary with MongoDB

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



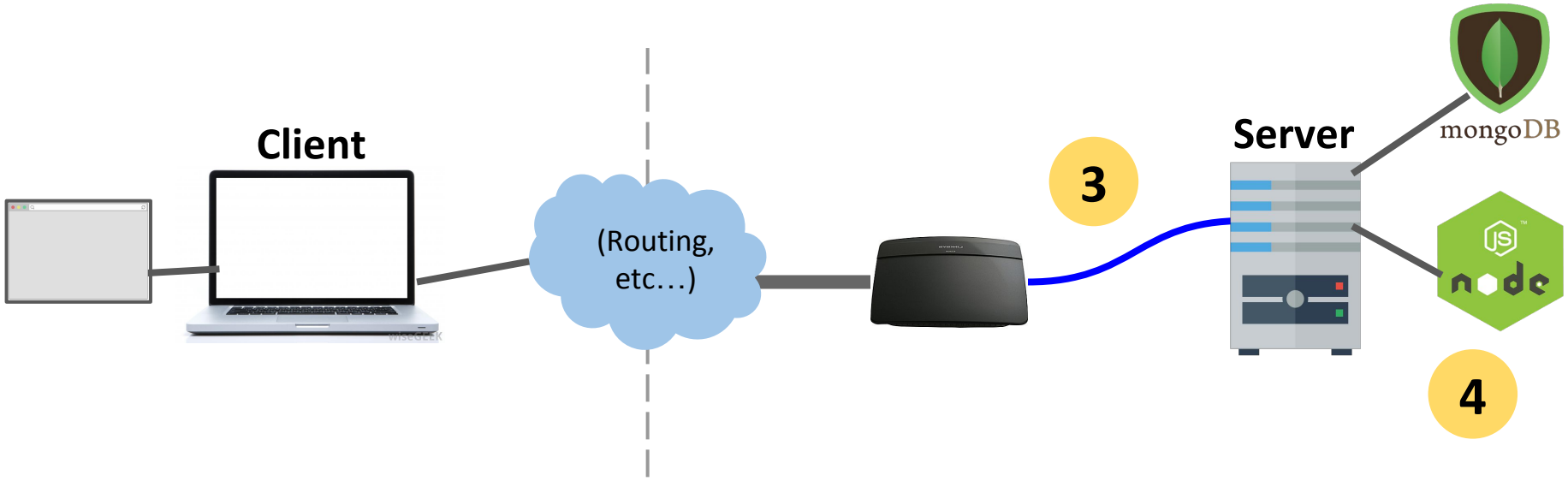
Review system



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

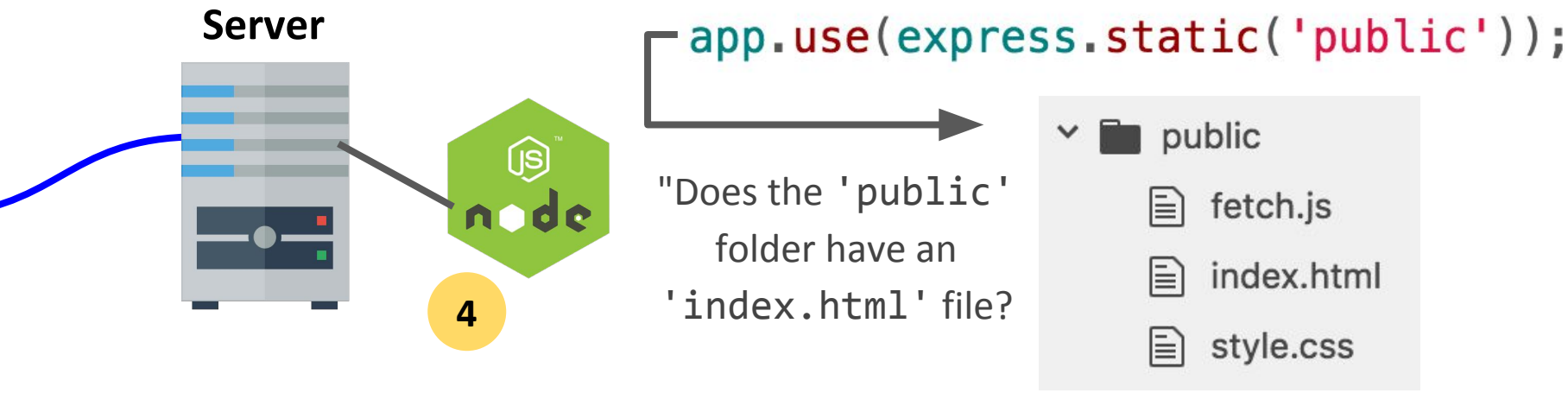
1. The user navigates to abc.com
2. The browser makes an HTTP GET request for abc.com

Review system



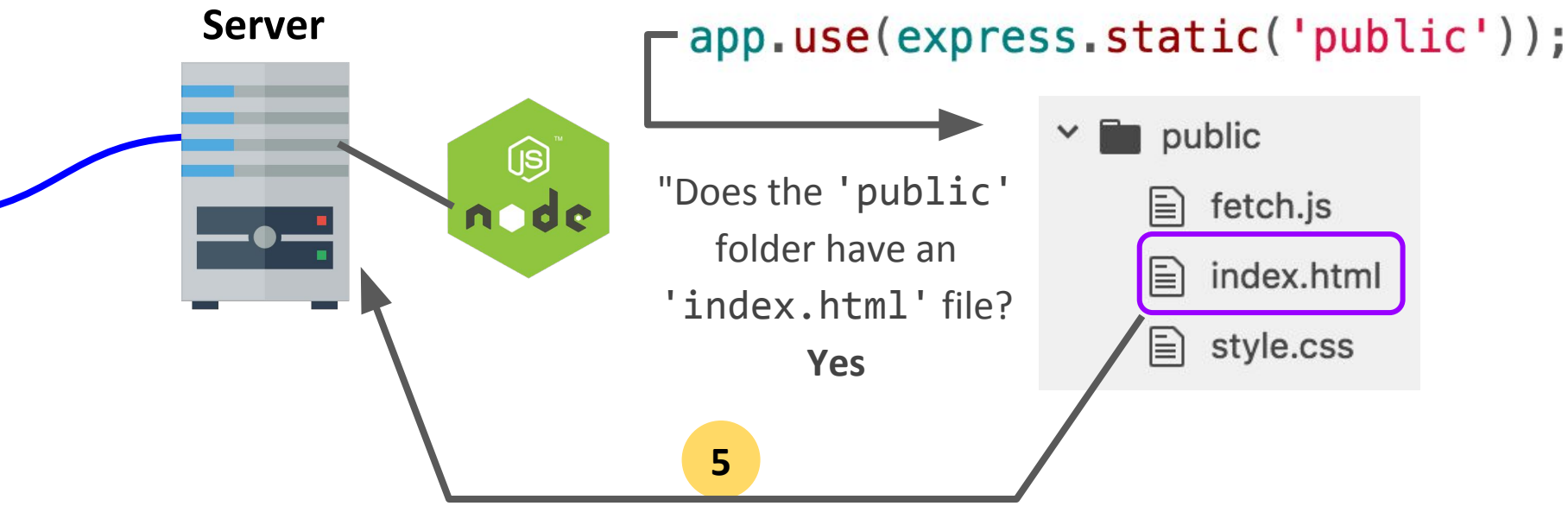
3. The server computer that is located at abc.com receives the HTTP GET request
4. The server computer gives the NodeJS server process the HTTP GET request message

Review system



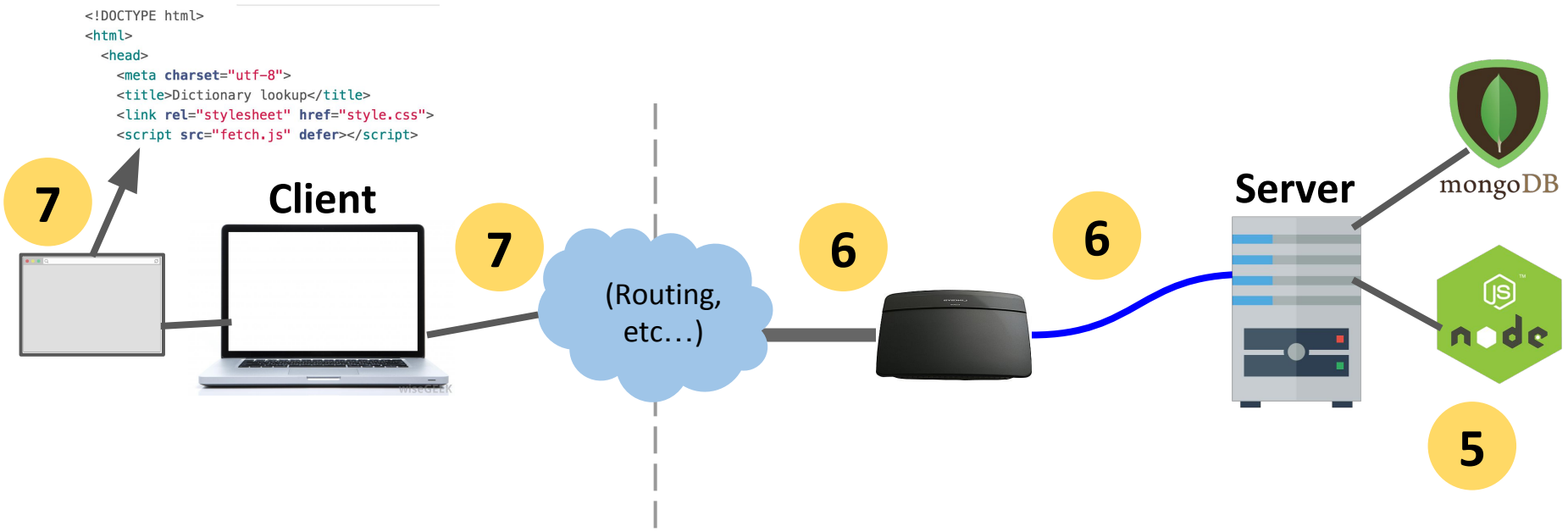
Our NodeJS server code has `app.use(express.static('public'));` so it will first look to see if an `index.html` file exists in the `public` directory.

Review system



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

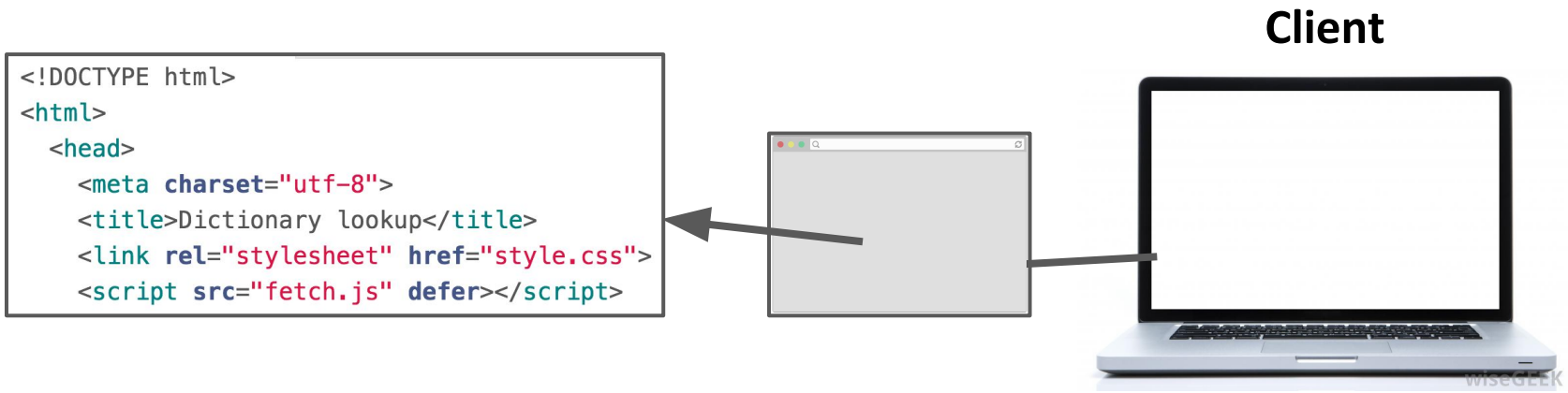
Review system



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

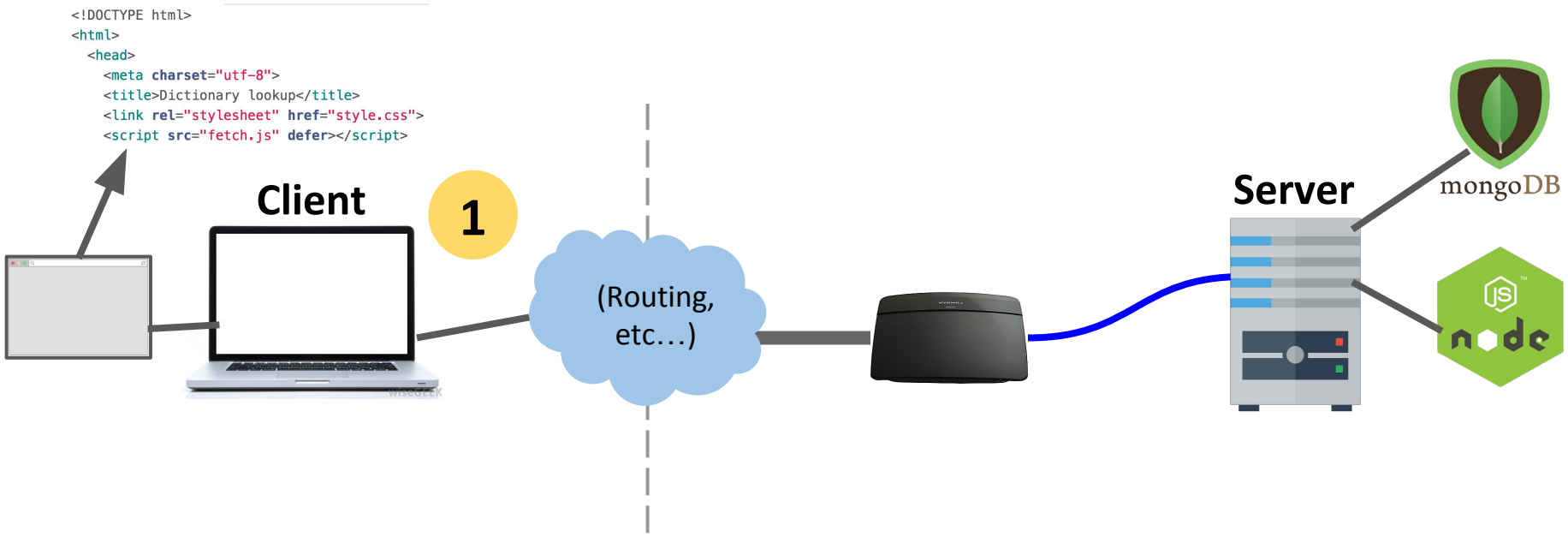
Review system

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



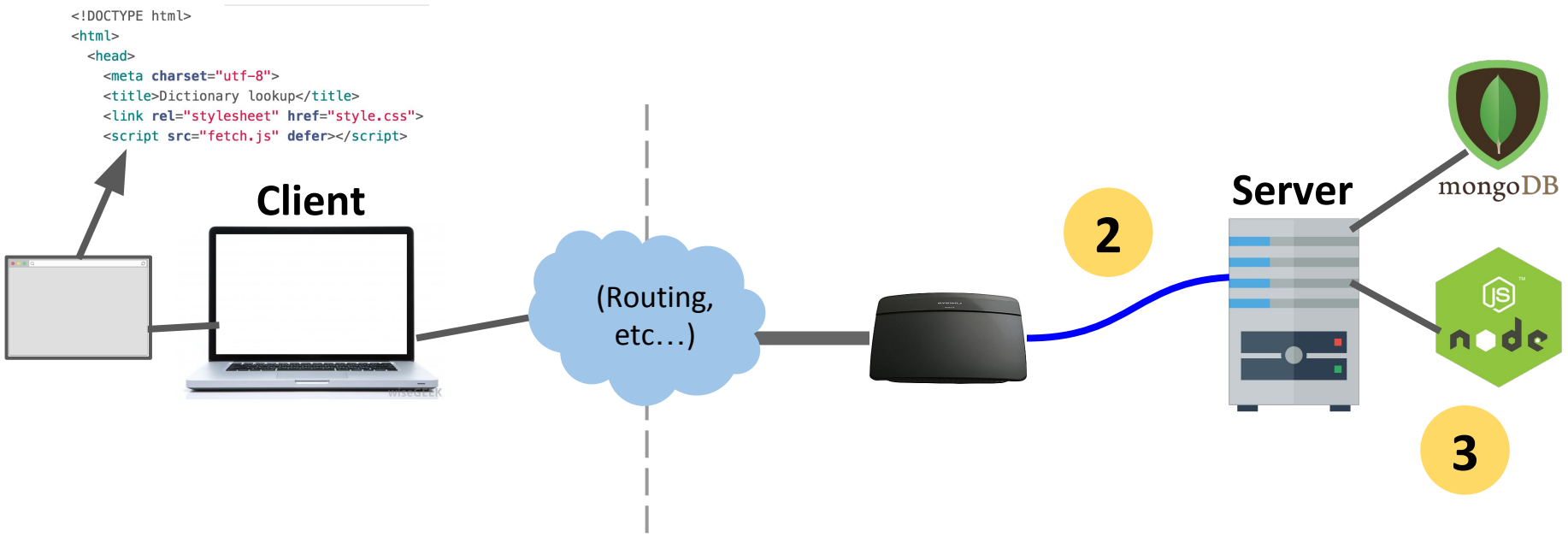
8. In rendering the HTML, the browser sees it needs style.css and fetch.js

Review system



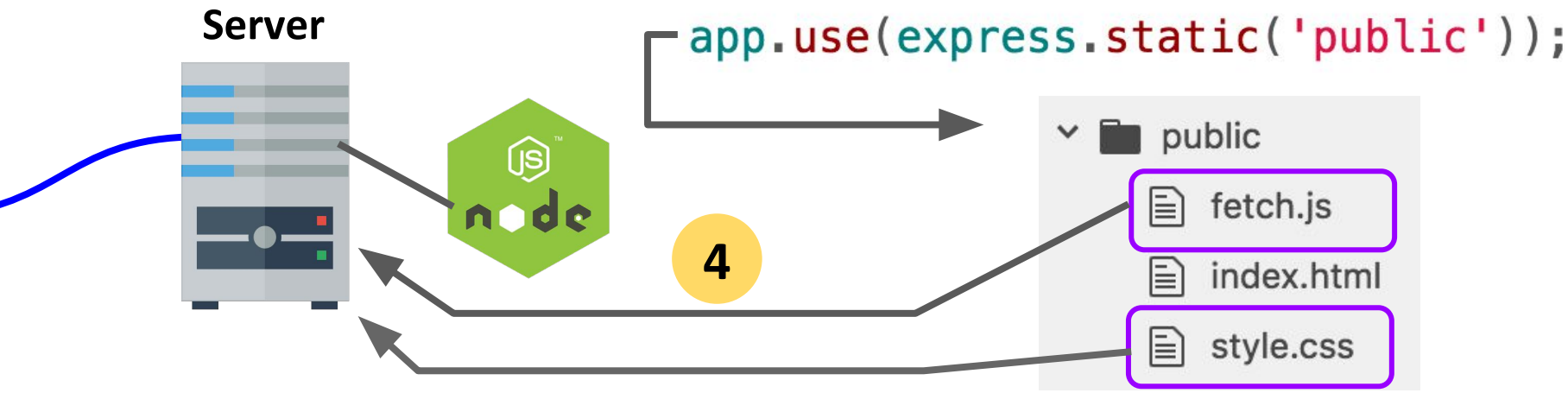
1. So the browser makes two more HTTP GET requests:
   - One for style.css
   - One for script.js

Review system



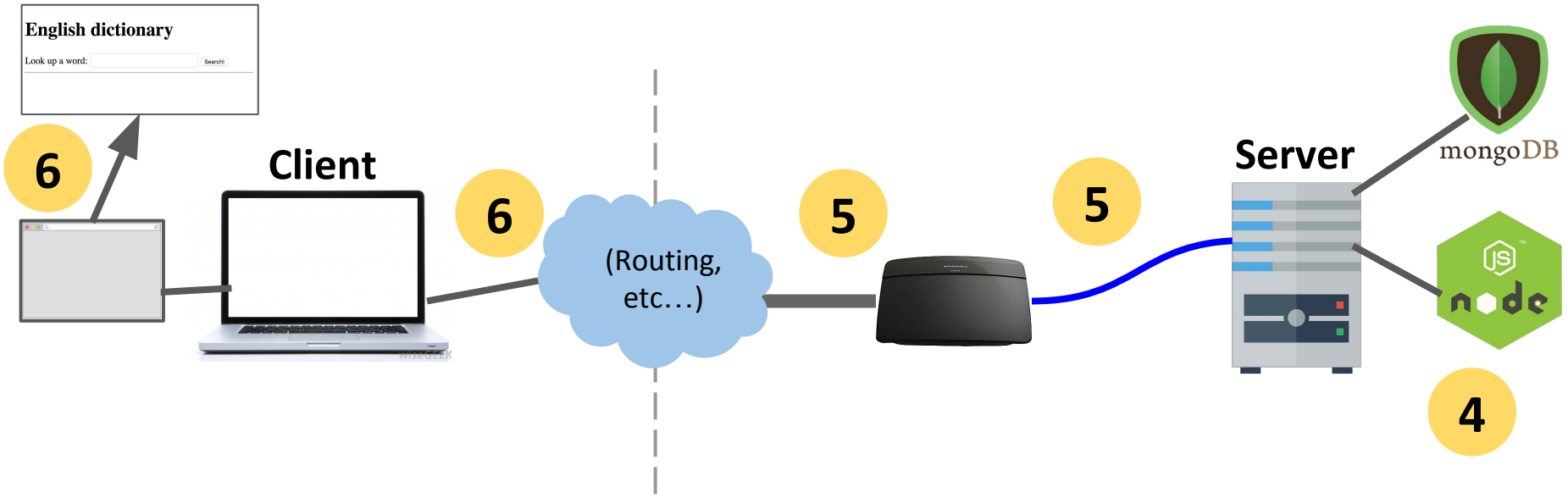
2. These GET requests get routed to the server computer
3. The server computer sends the GET requests to our NodeJS process

Review system



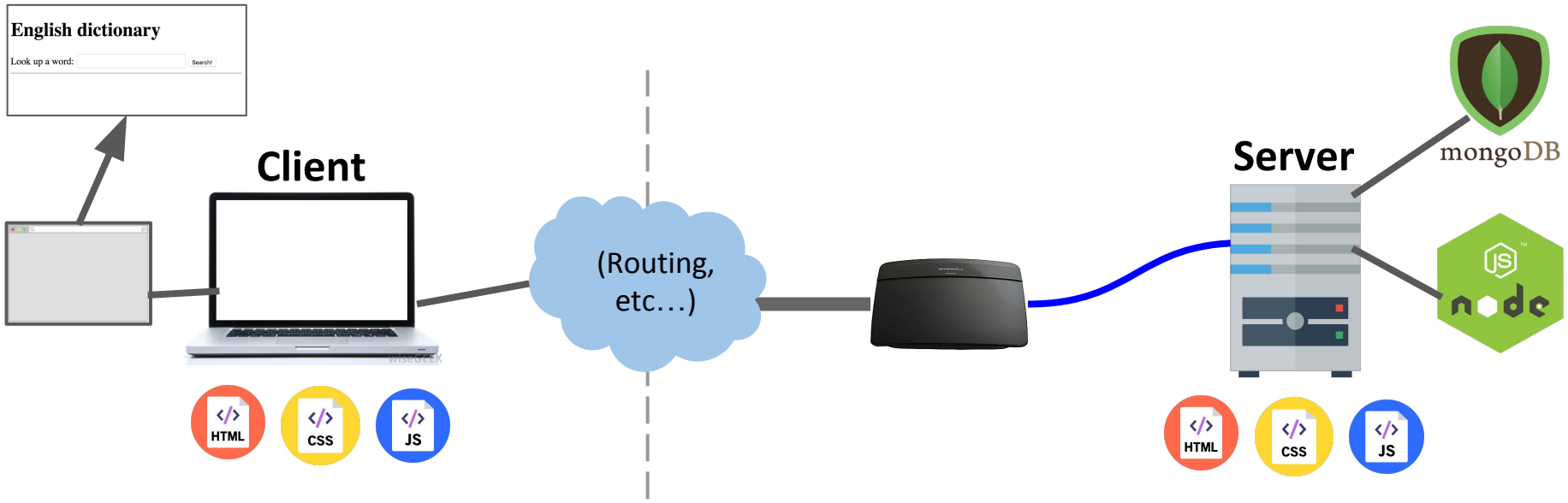
4. Our NodeJS server code finds fetch.js and style.css in the public directory, so it responds with those files

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

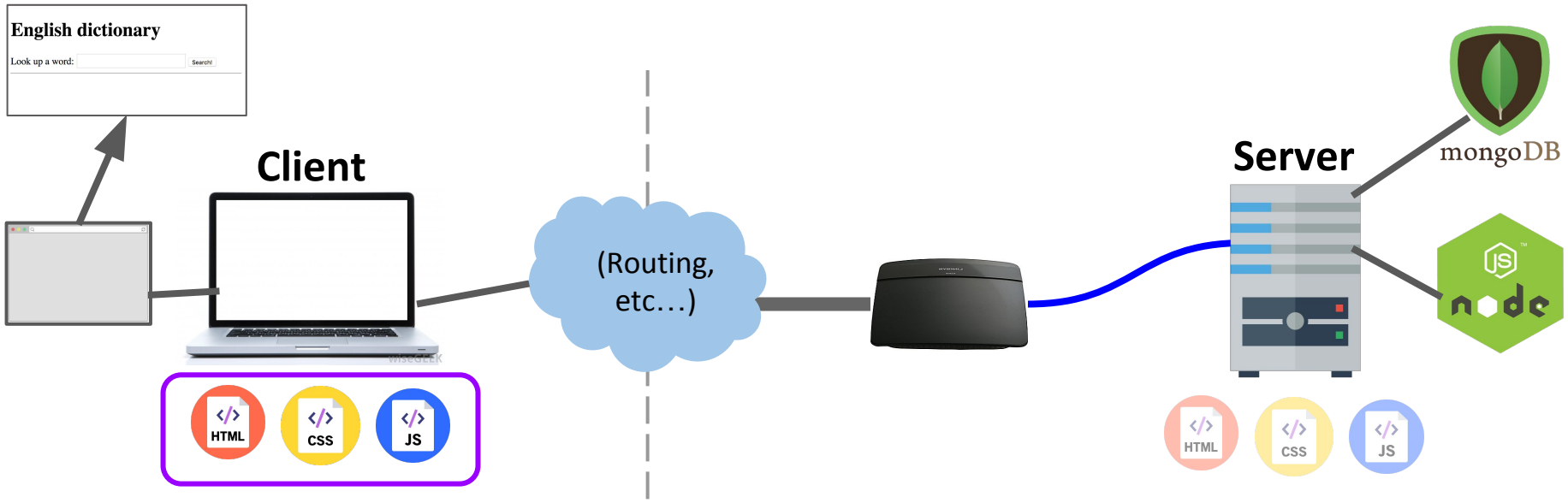
Review system



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

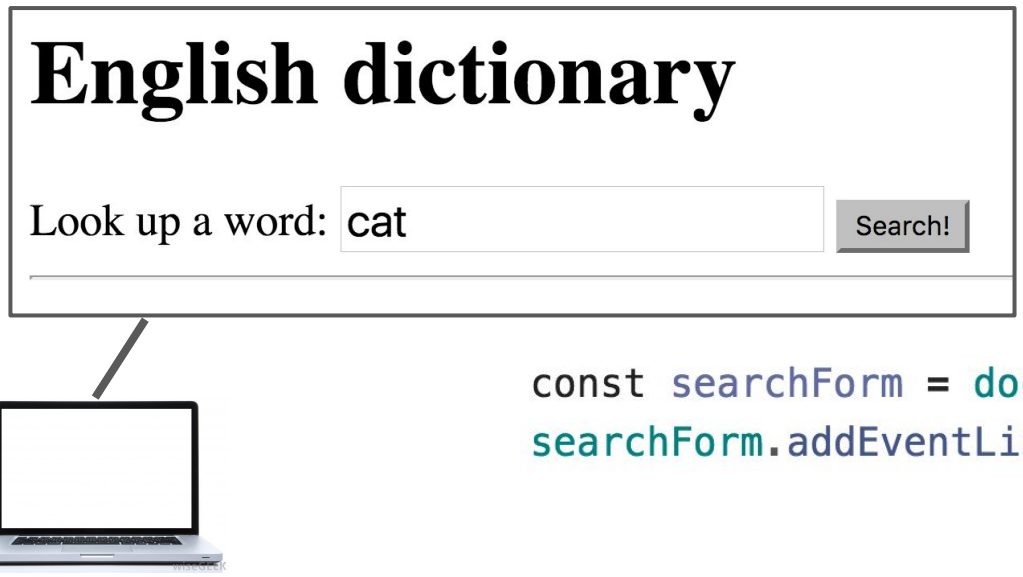


The server computer **provided** the files.

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



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

2

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

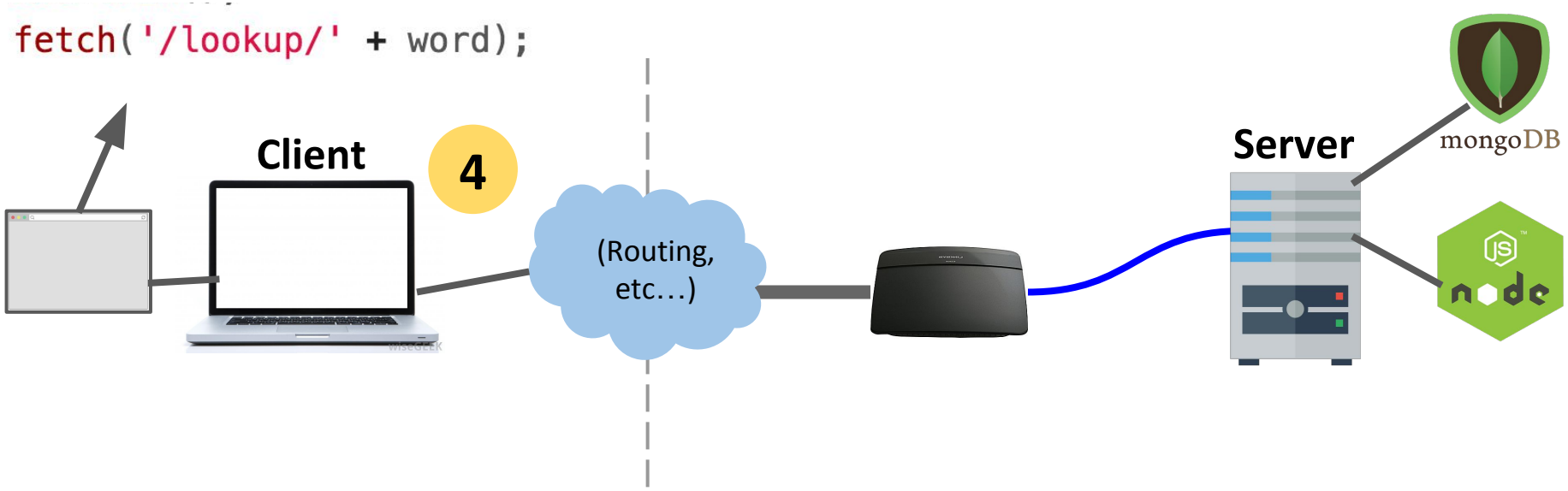
Client

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();
```

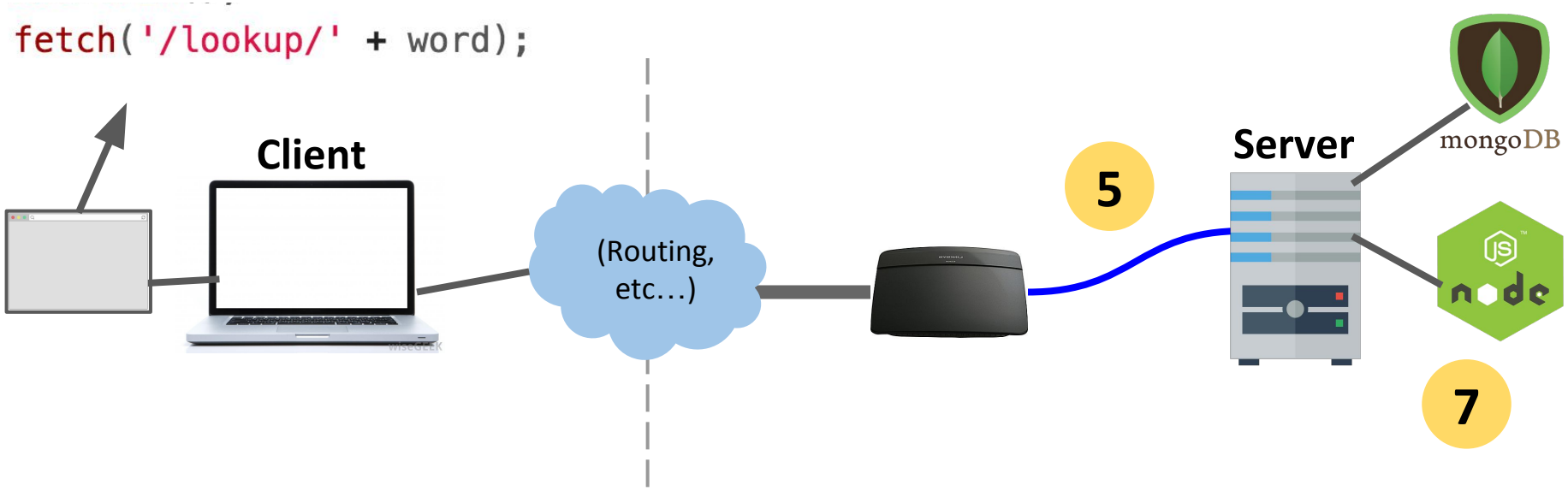
3. Our `onSearch` function includes a call to `fetch()`, which is going to trigger another HTTP GET request, this time for abc.com/lookup/cat.

Review system



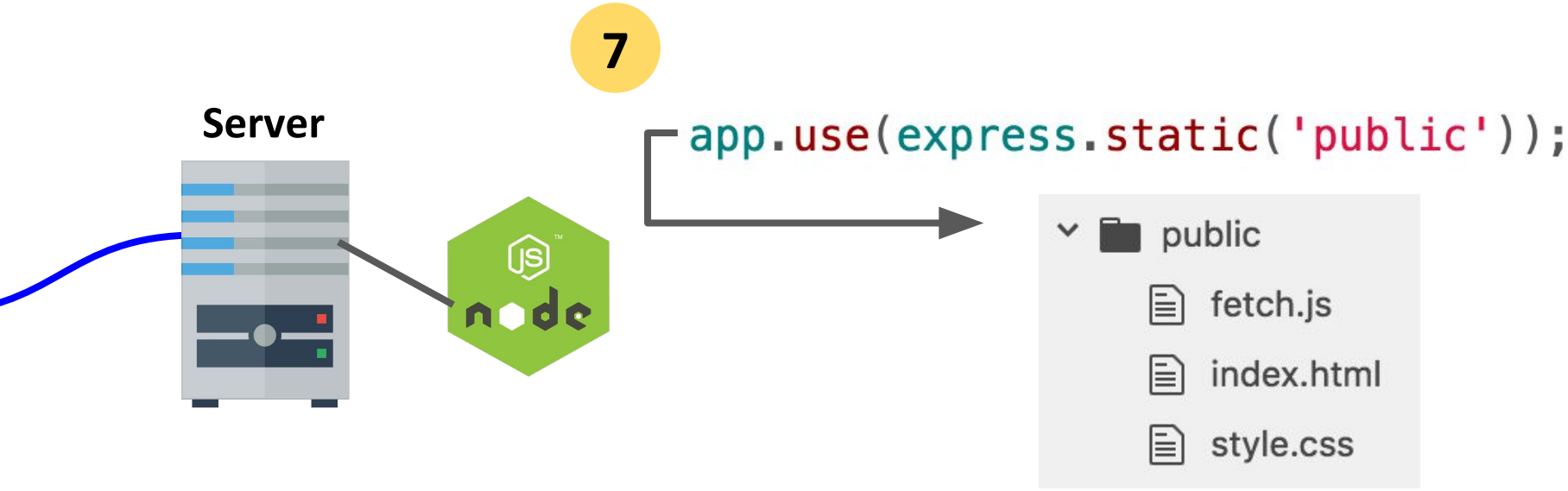
4. Because of the call to fetch(), the browser makes an HTTP GET request for abc.com/lookup/cat.

Review system



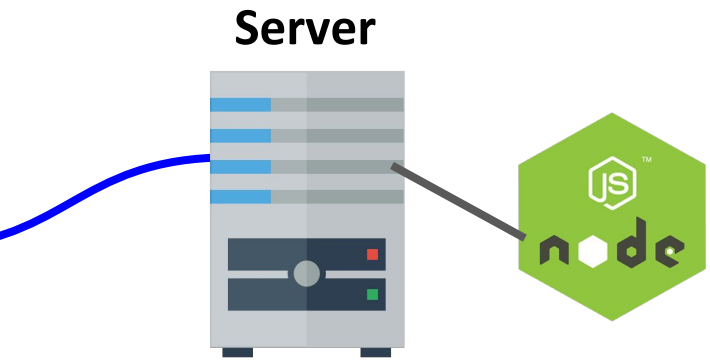
5. These GET requests get routed to the server computer
6. The server computer sends the GET requests to our NodeJS process

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

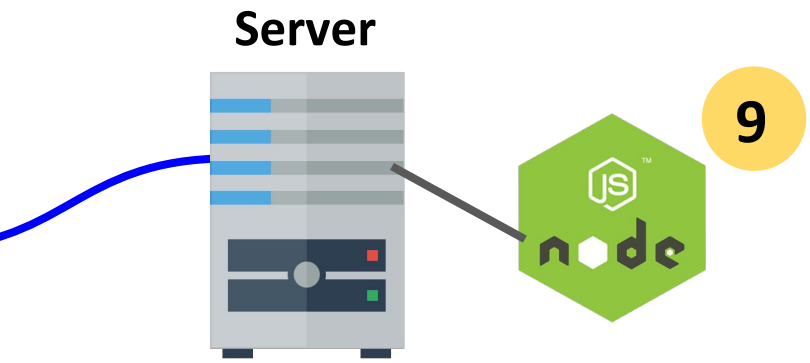


8

```
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

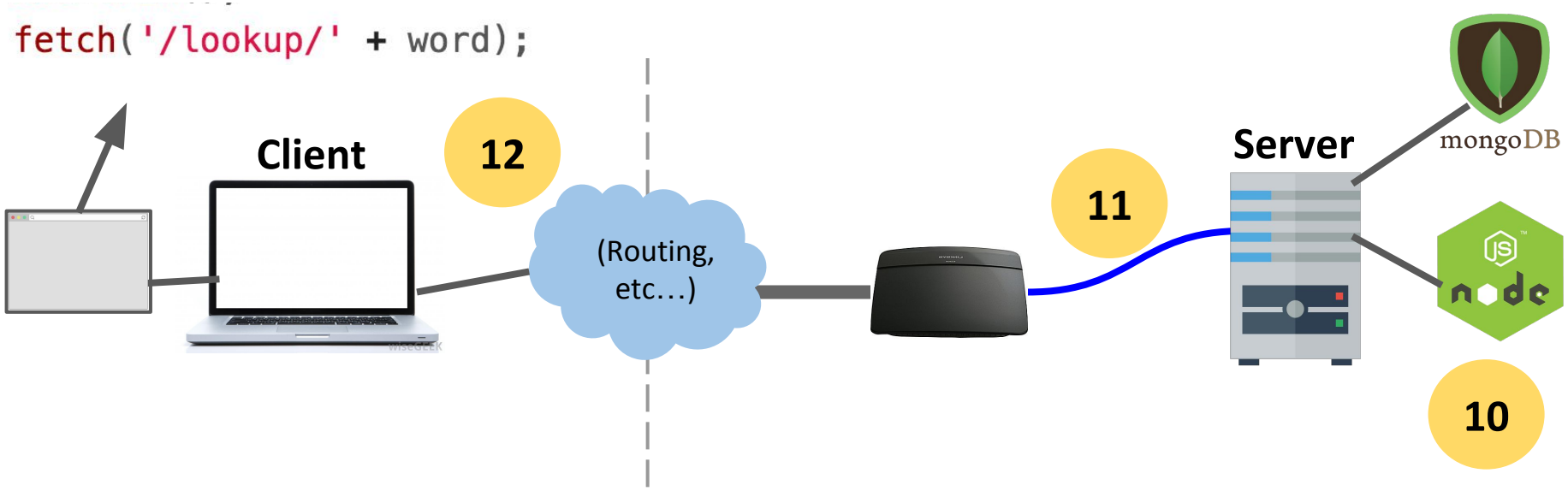



```
function onLookupWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word;

  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

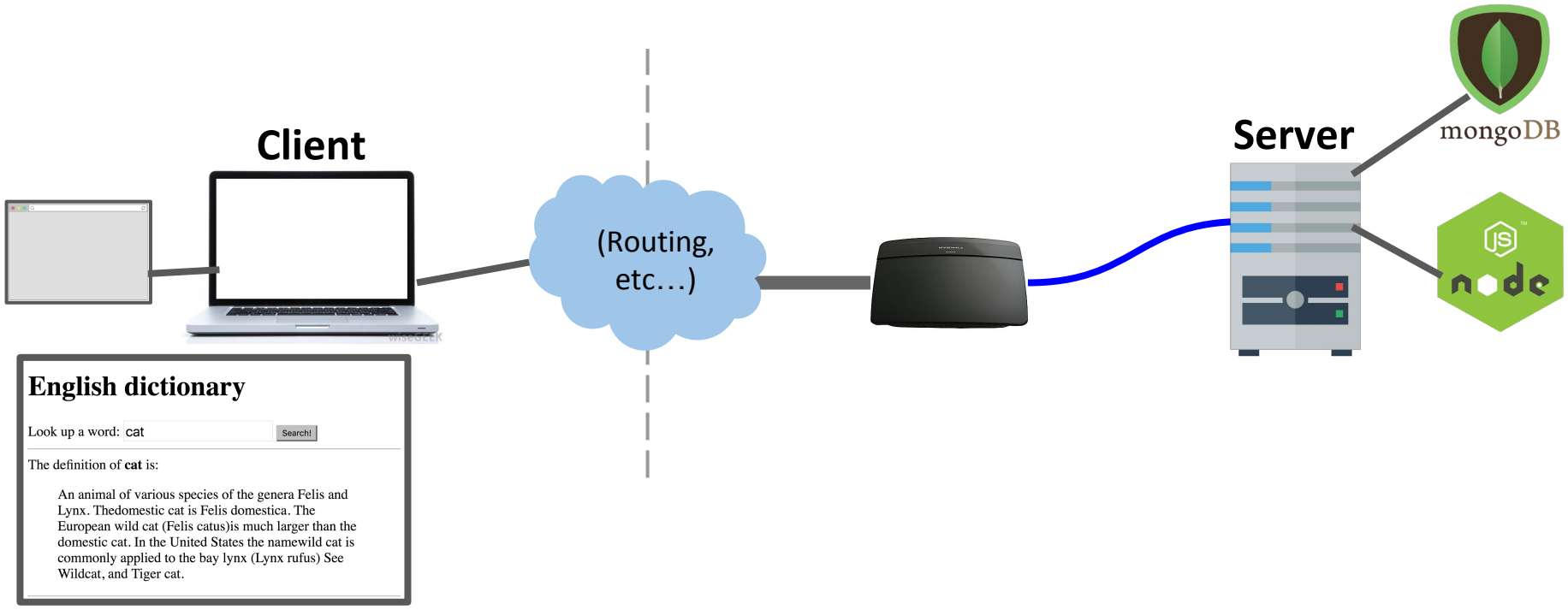
English dictionary

Look up a word: 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.

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

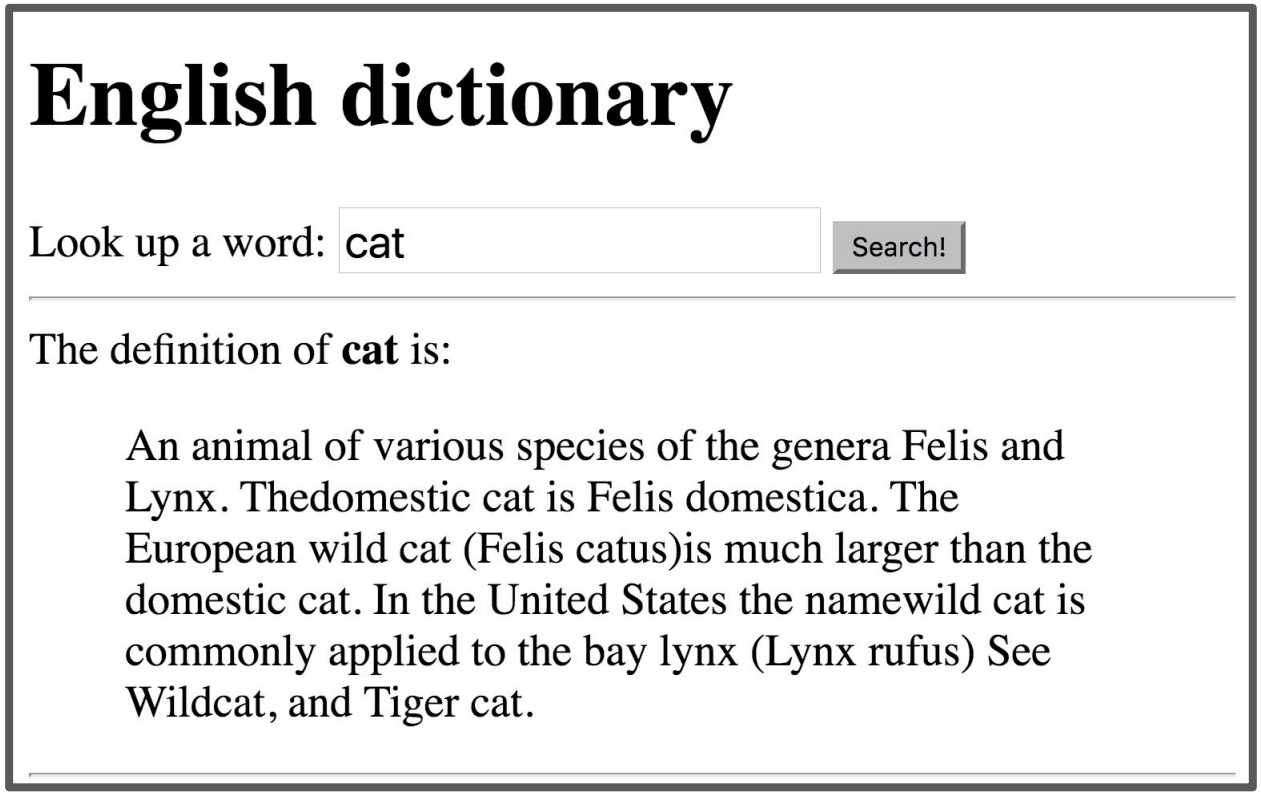
```
async function startServer() {
  db = await MongoClient.connect(MONGO_URL);
  collection = db.collection('words');
  await app.listen(3000);
  console.log('Listening on port 3000');
}
startServer();
```

Starting a server: After

```
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...



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);
```

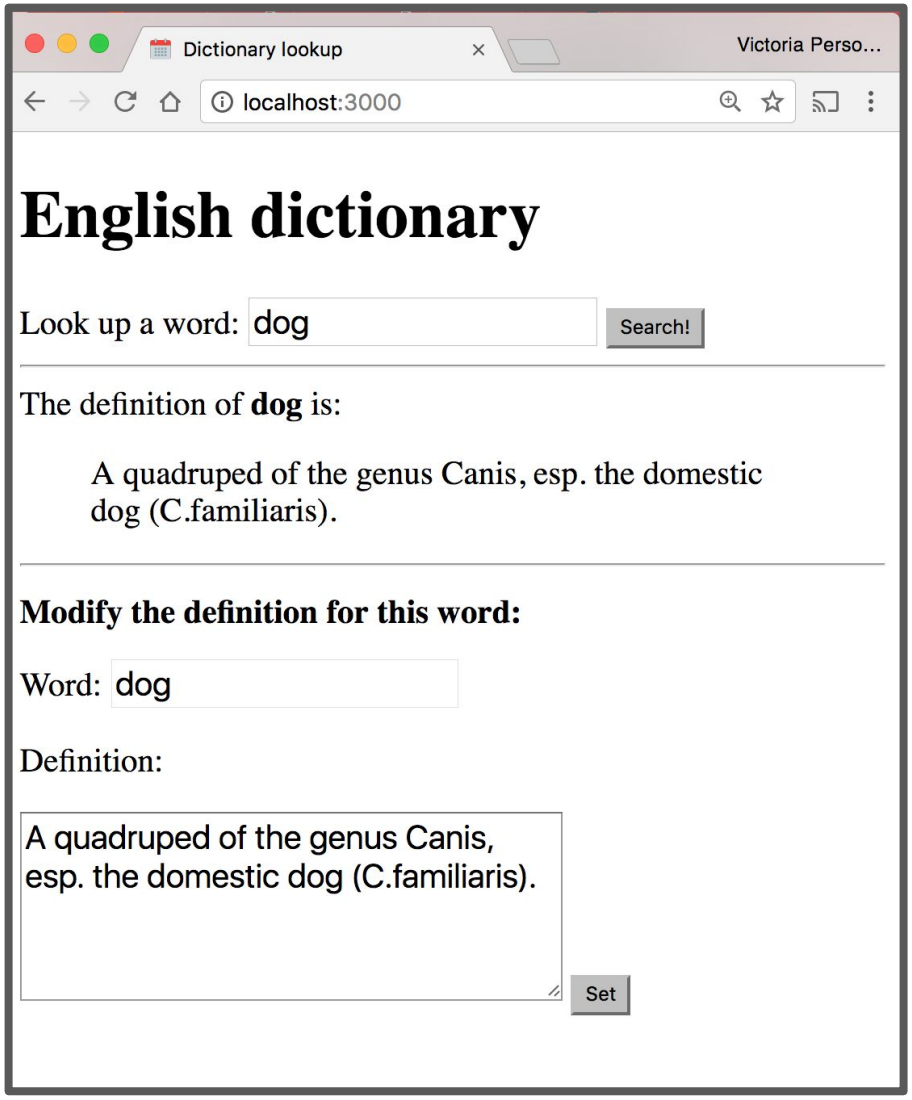
(Previous code: This **doesn't** use MongoDB)

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:



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

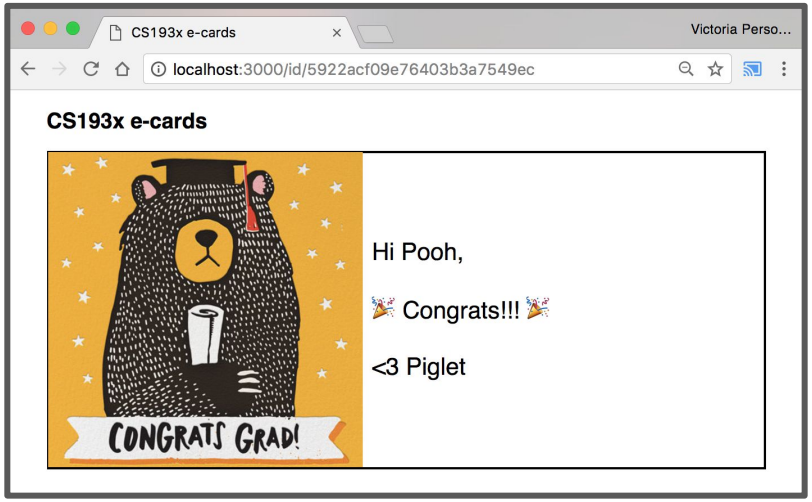
```
async function onSetWord(req, res) {
  const routeParams = req.params;
  const word = routeParams.word.toLowerCase();
  const definition = req.body.definition;

  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);
```

Overflow (if we have time)

Another example: E-cards

Example: E-cards

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



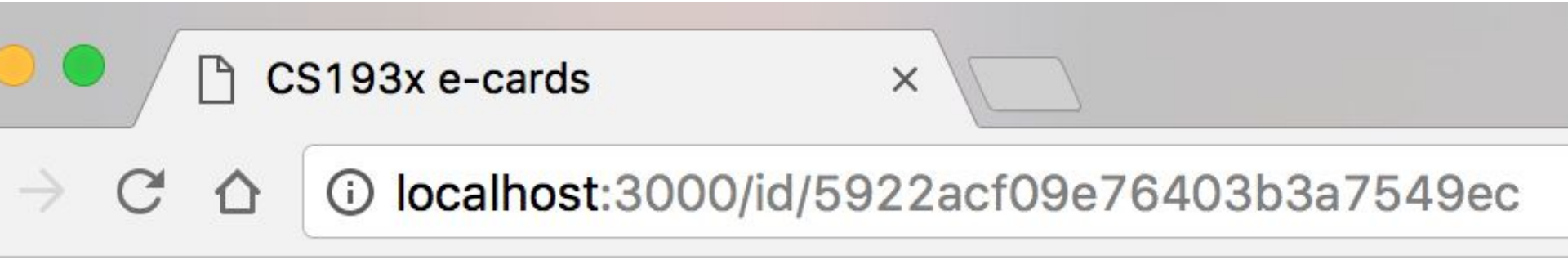
Setup

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

Setup

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

Example: E-card saving data

```
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);
```