

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.');
```

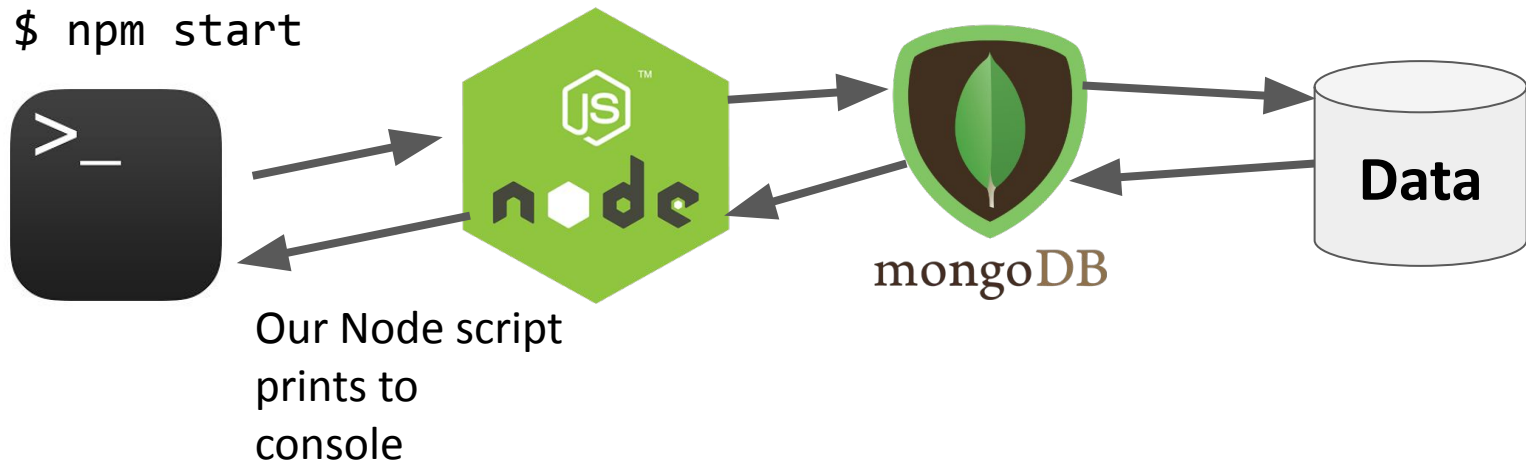


```
    console.log();  
}  
  
printPoem();  
printPoem();
```

Mongo JS scripts

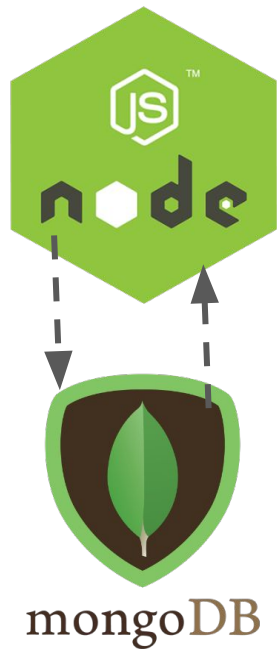
Before we start manipulating MongoDB from the server, let's just write some JavaScript files that will query MongoDB.

```
$ npm start
```



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](#) 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 } })`

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 [ObjectID](#) 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:

```
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', bodyParser, onSaveCard);
```

Find by Mongo ID: ObjectId

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

```
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('cards');  
  const response = await collection.insertOne(doc);  
  
  res.json({ cardId: response.insertedId });  
}  
app.post('/save', bodyParser.json, 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 [ObjectId](#)**.

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

```
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 [ObjectId](#)**.

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

```
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 with Express

[Github](#)

Dictionary with MongoDB

Let's work on an English dictionary example. The database will contain english words and their definitions.



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:

Starting the server:

```
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 based on the word we wrote in the form

English dictionary

Look up a word:

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.

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 be able to modify definitions in the dictionary , using the form textarea:



The screenshot shows a web browser window titled "Dictionary lookup" with the address bar displaying "localhost:3000". The page content includes a search bar with the text "dog" and a "Search!" button. Below the search bar, the definition of "dog" is displayed: "A quadruped of the genus Canis, esp. the domestic dog (C.familiaris)." There is a section titled "Modify the definition for this word:" which contains a "Word:" field with "dog" and a "Definition:" field with a large text area containing the same definition as above. A "Set" button is located at the bottom right of the definition text area.

Dictionary lookup

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:

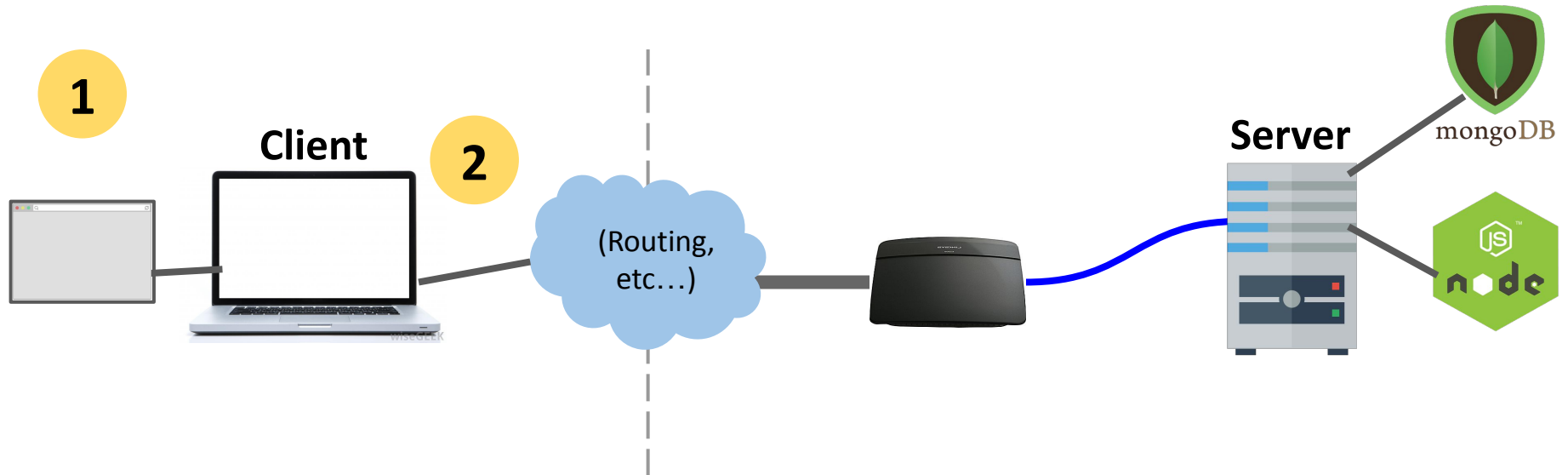
Definition:

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);
```


Full step by step review

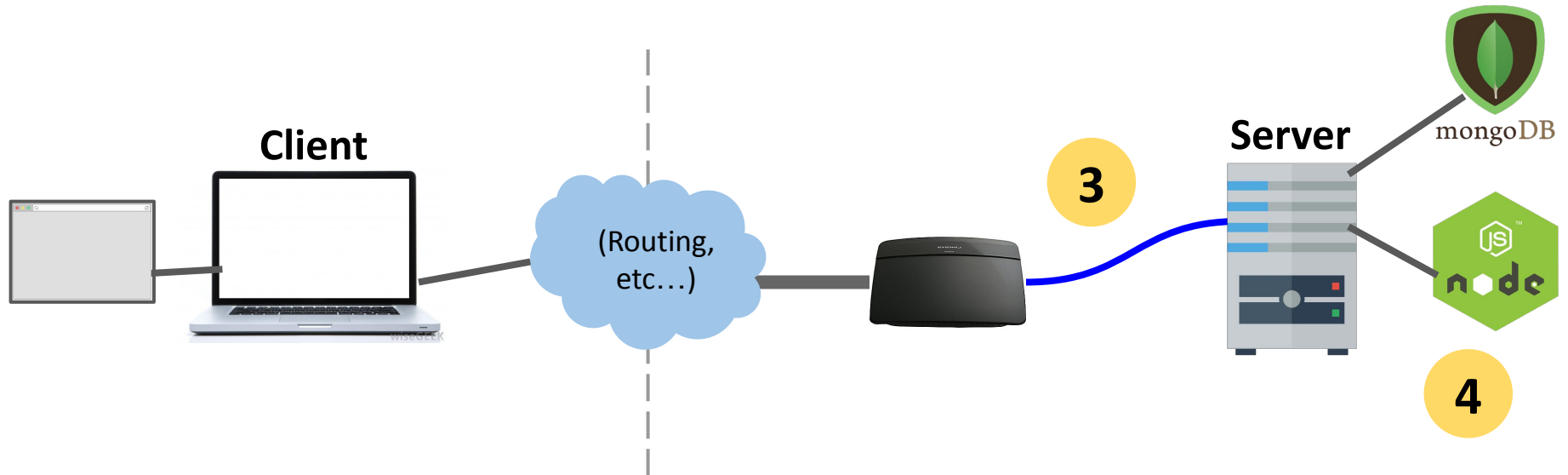
Review system



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

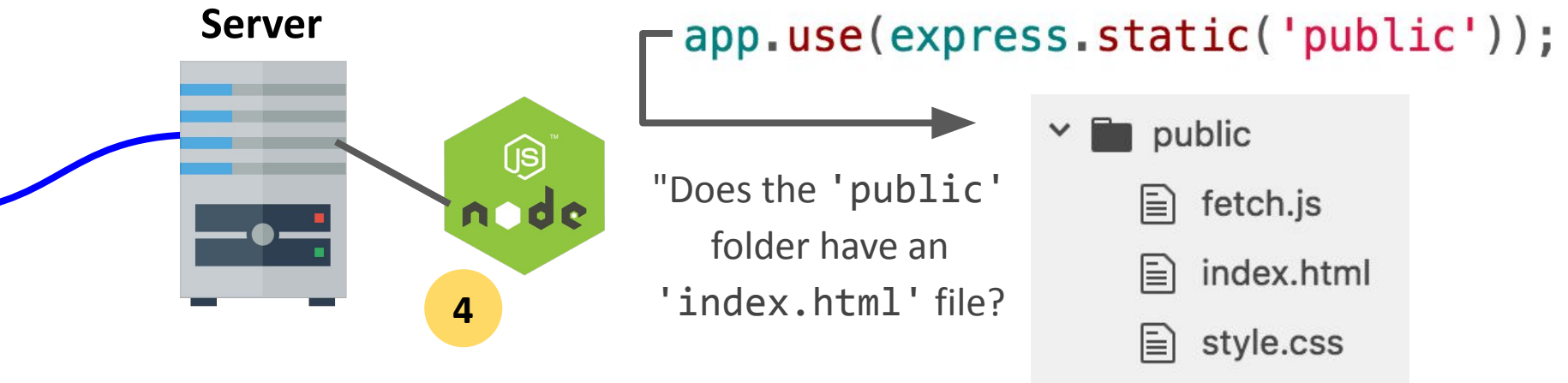
1. The user navigates to abc.com
2. The browser makes an HTTP GET request for abc.com

Review system



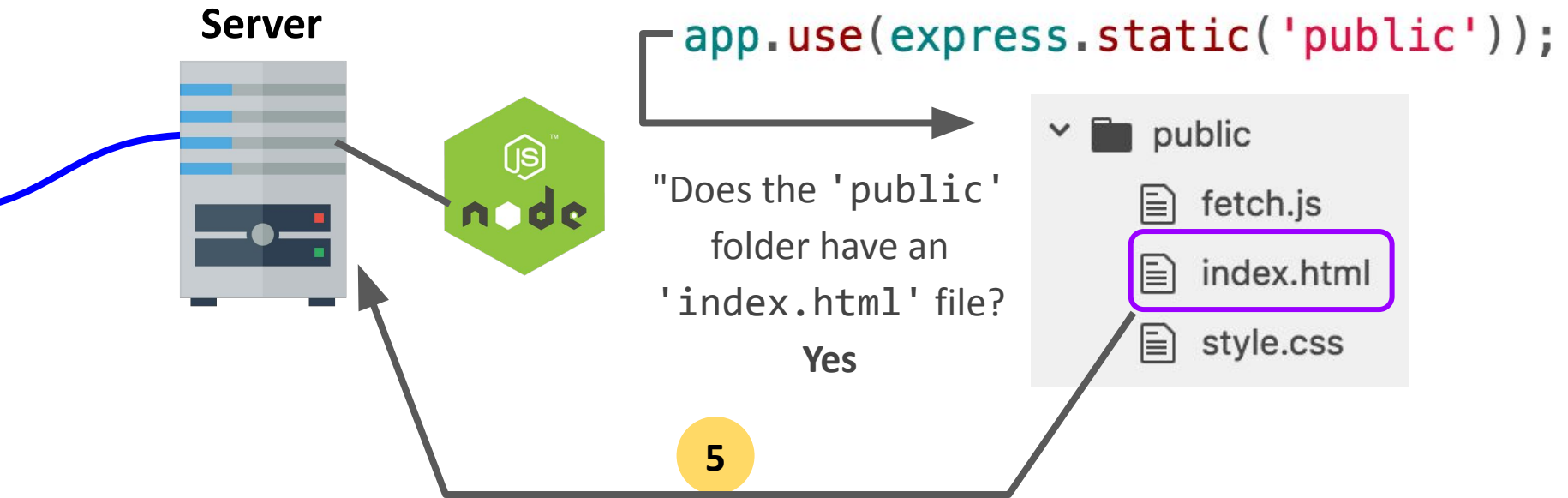
3. The server computer that is located at abc.com receives the HTTP GET request
4. The server computer gives the NodeJS server process the HTTP GET request message

Review system



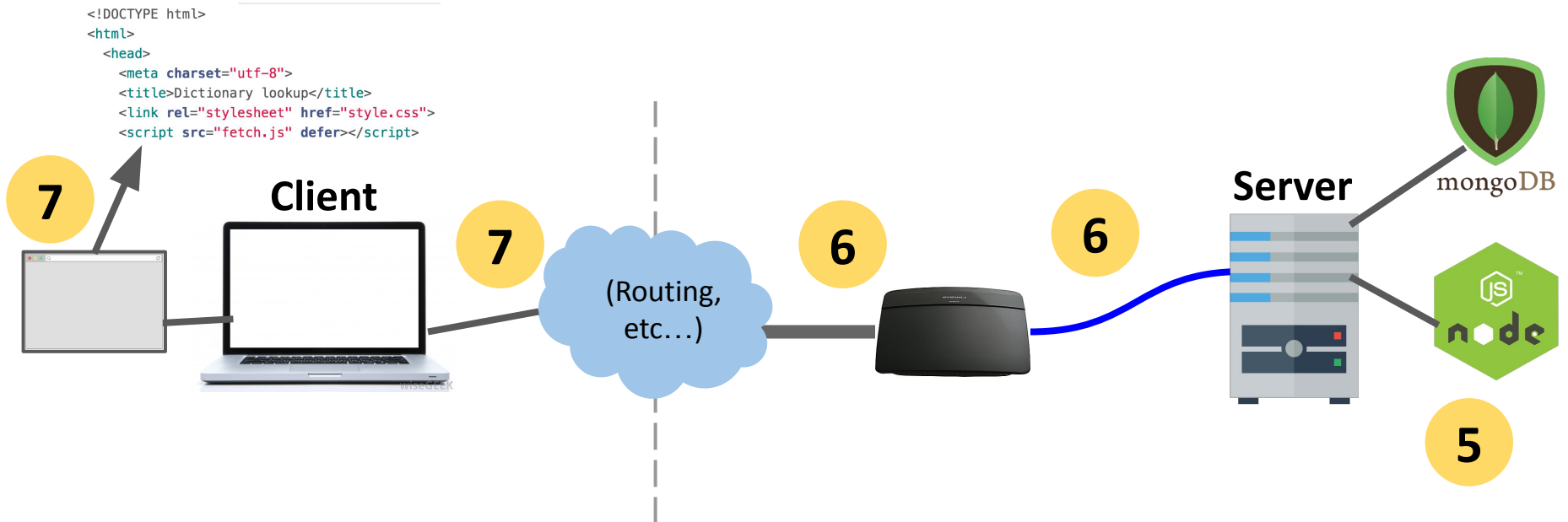
Our NodeJS server code has `app.use(express.static('public'));` so it will first look to see if an `index.html` file exists in the `public` directory.

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>
```

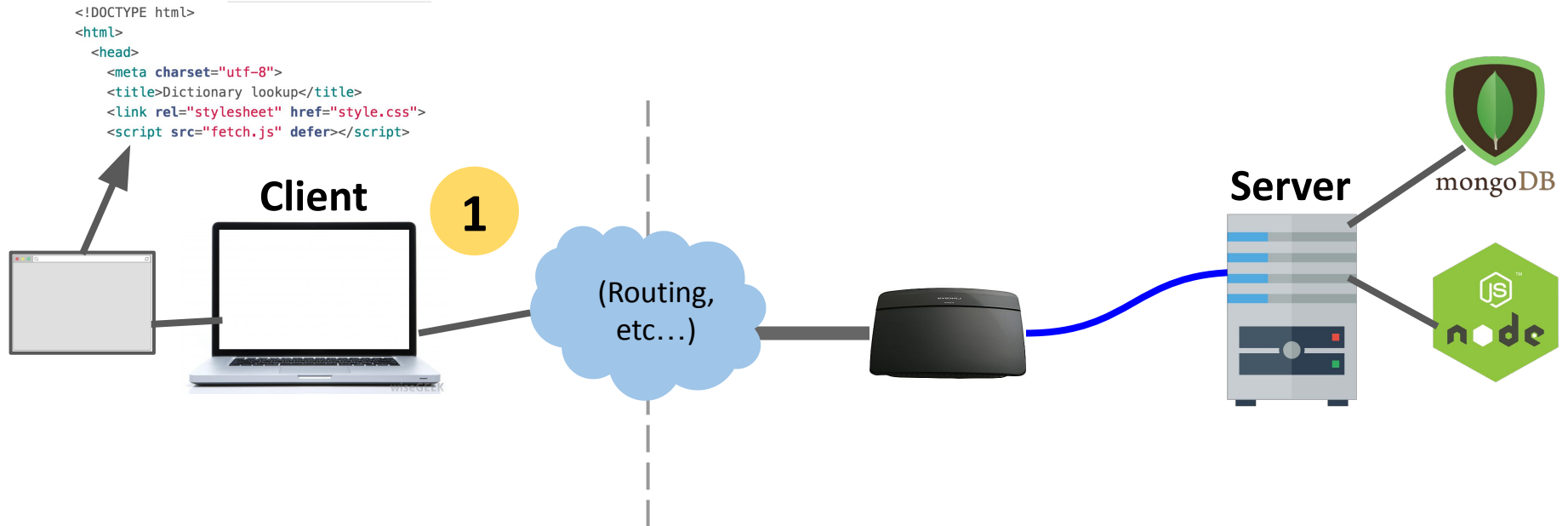
Client

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



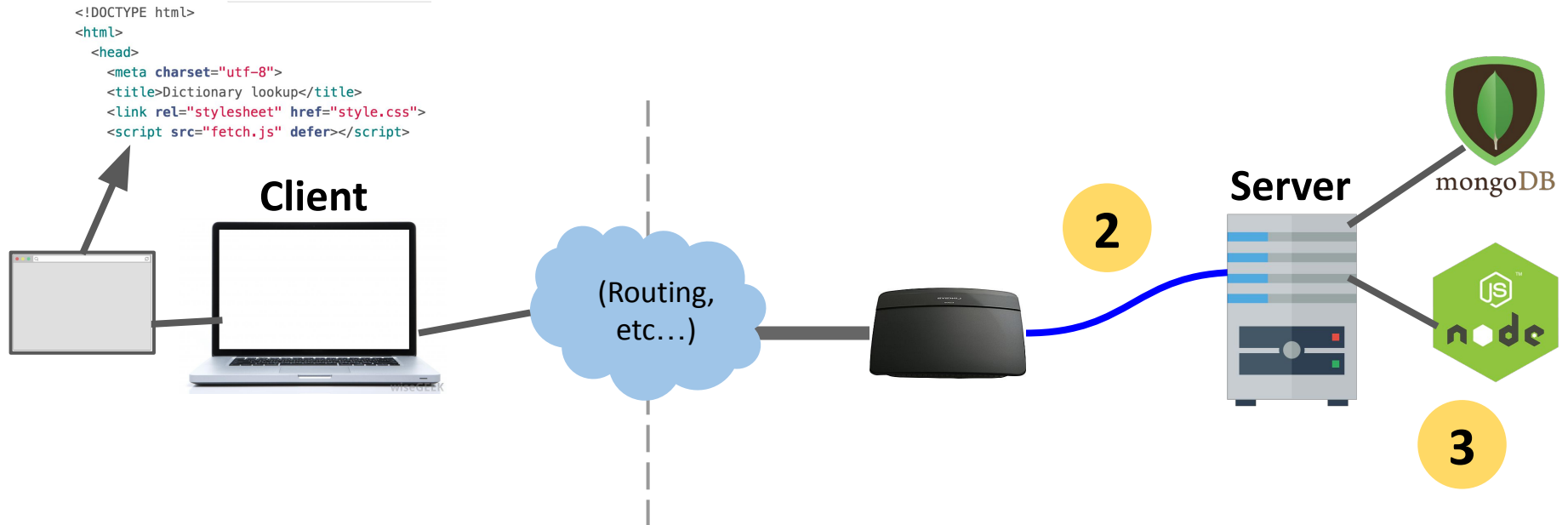
8. In rendering the HTML, the browser sees it needs style.css and fetch.js

Review system



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

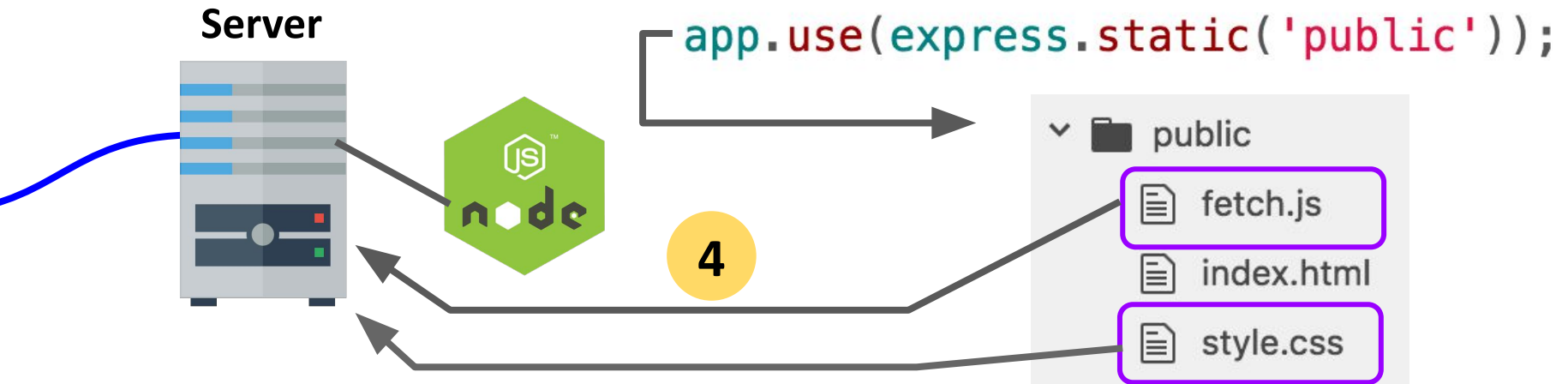
Review system



2. These GET requests get routed to the server computer

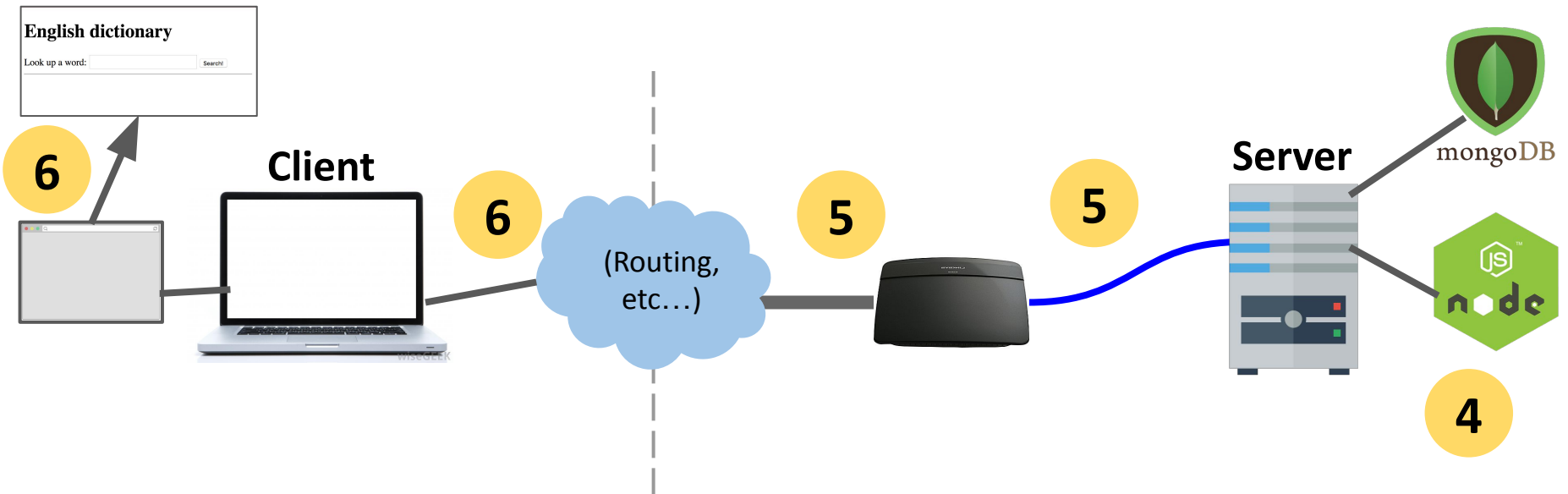
3. The server computer sends the GET requests to our NodeJS process

Review system



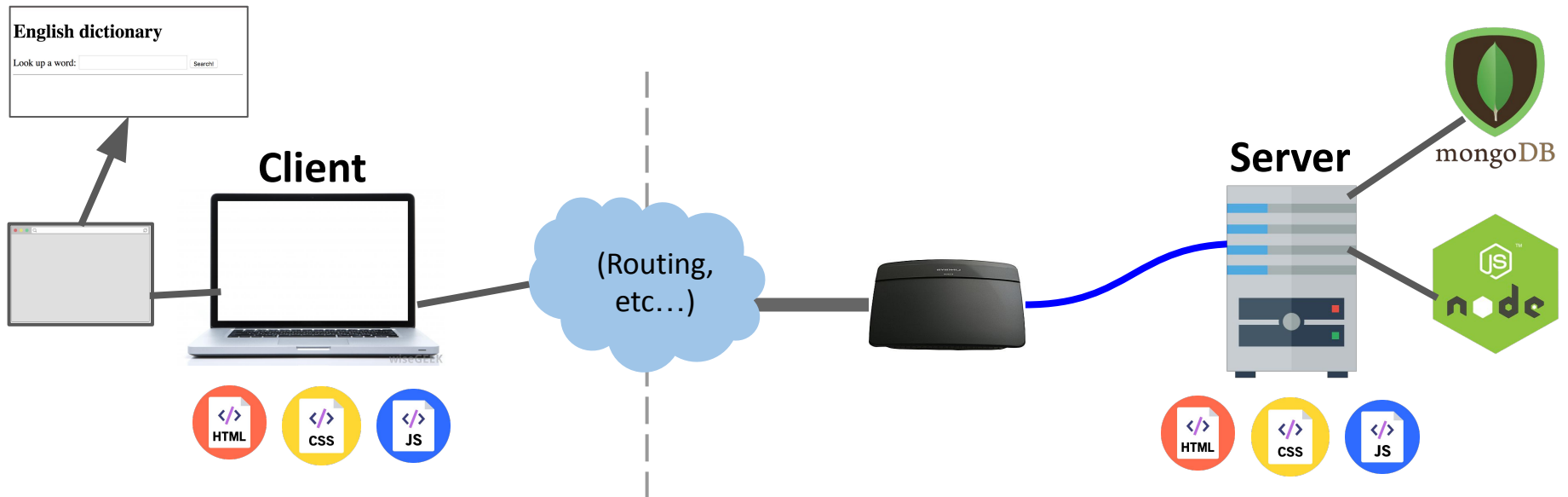
4. Our NodeJS server code finds fetch.js and style.css in the public directory, so it responds with those files

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`

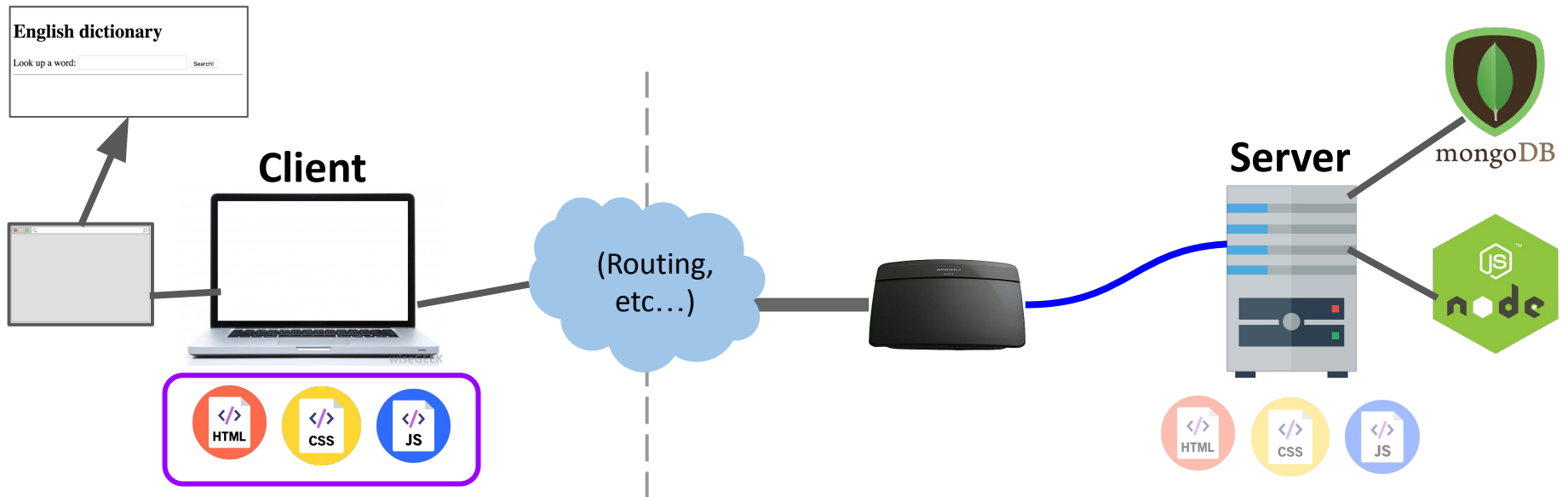
Review system



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:

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

2

English dictionary

Look up a word: cat

Search!



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



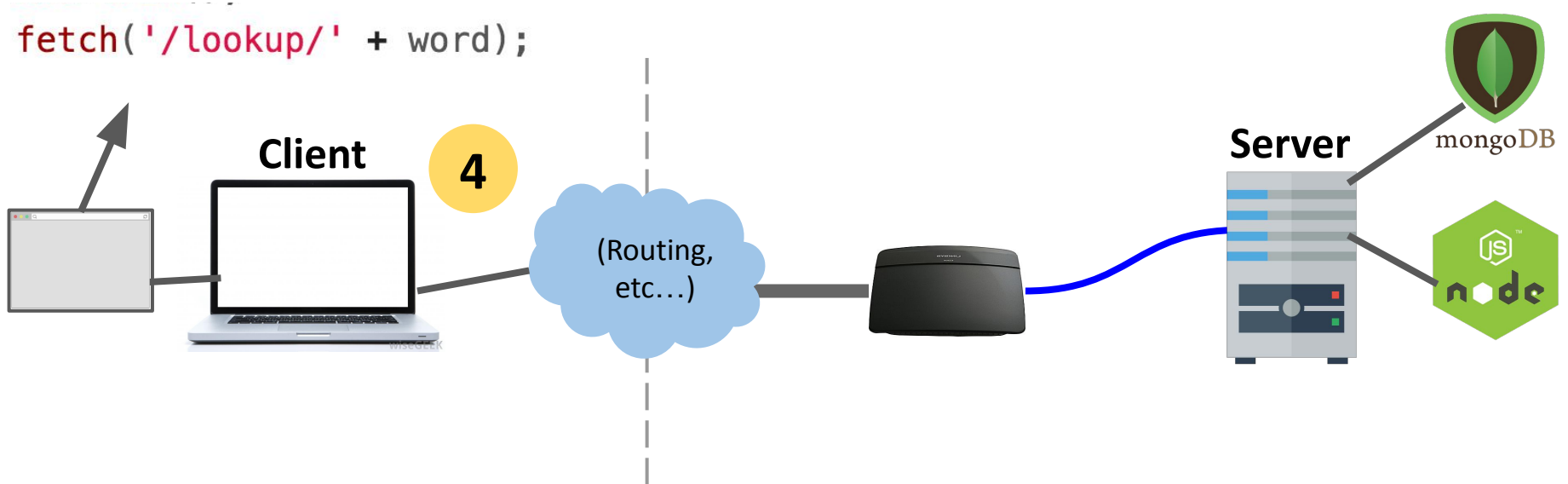
Client

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();  
}
```

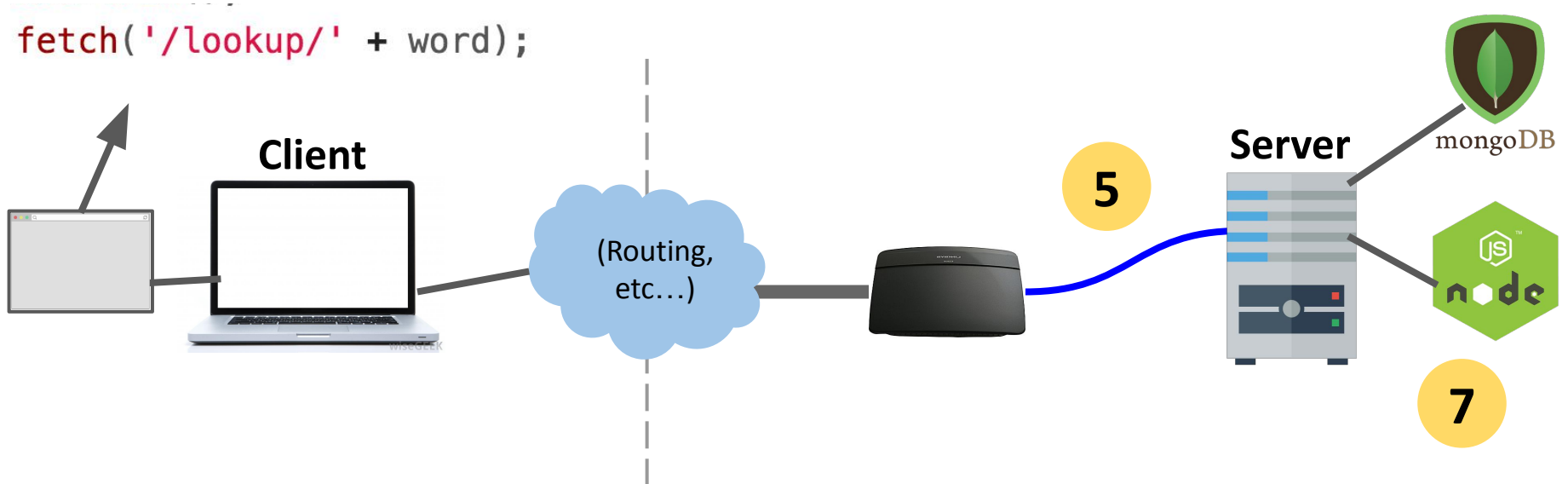
3. Our onSearch function includes a call to `fetch()`, which is going to trigger another HTTP GET request, this time for `abc.com/lookup/cat`.

Review system



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

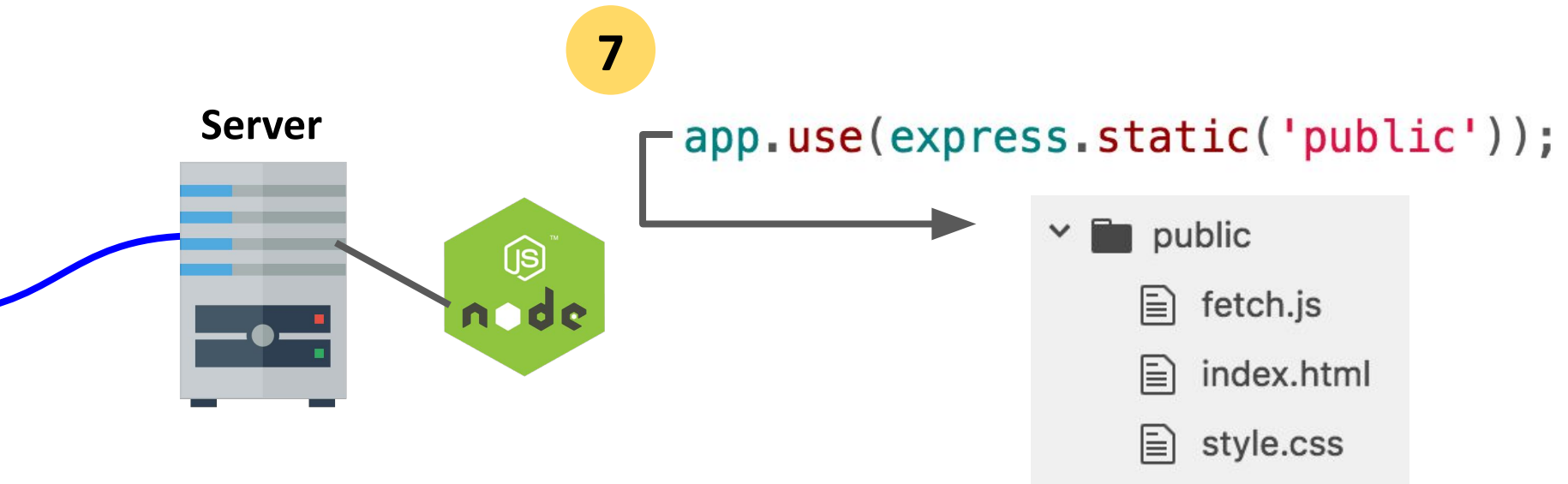
Review system



5. These GET requests get routed to the server computer

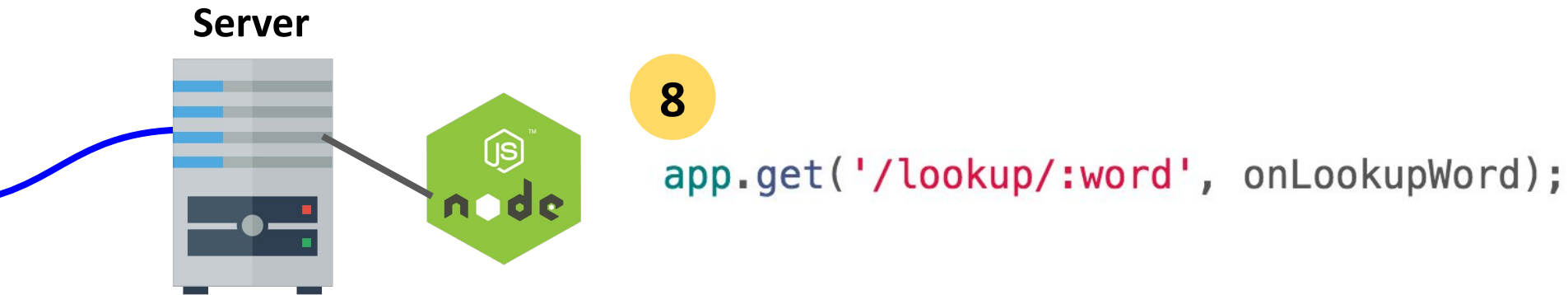
6. The server computer sends the GET requests to our NodeJS process

Review system



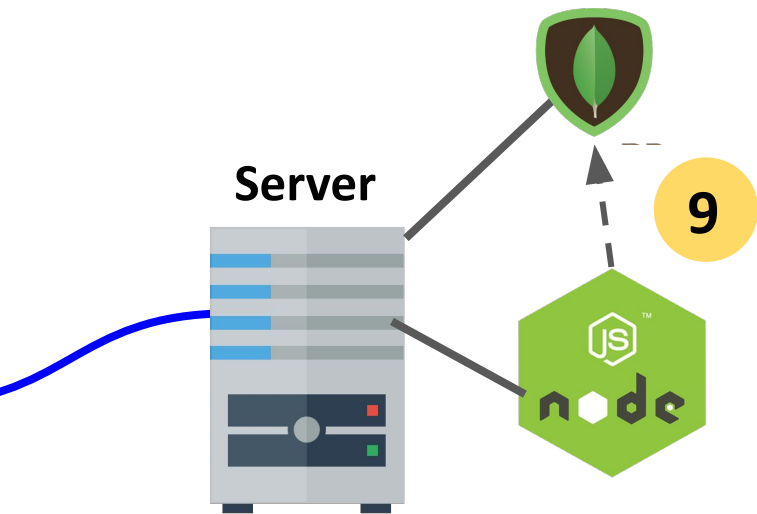
7. Our NodeJS server code first tries to see whether there's an "lookup/cat/index.html" in the public directory.

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

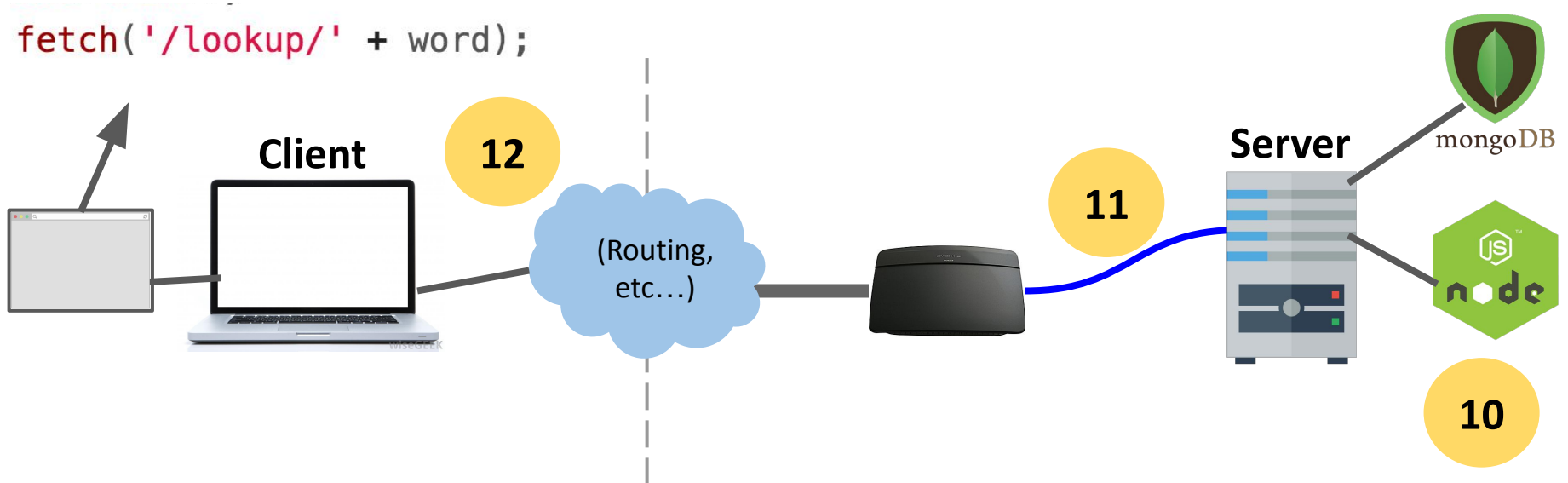
Review system



```
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



- 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

13

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

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

```
}
```



Client

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

Review system



Client

English dictionary

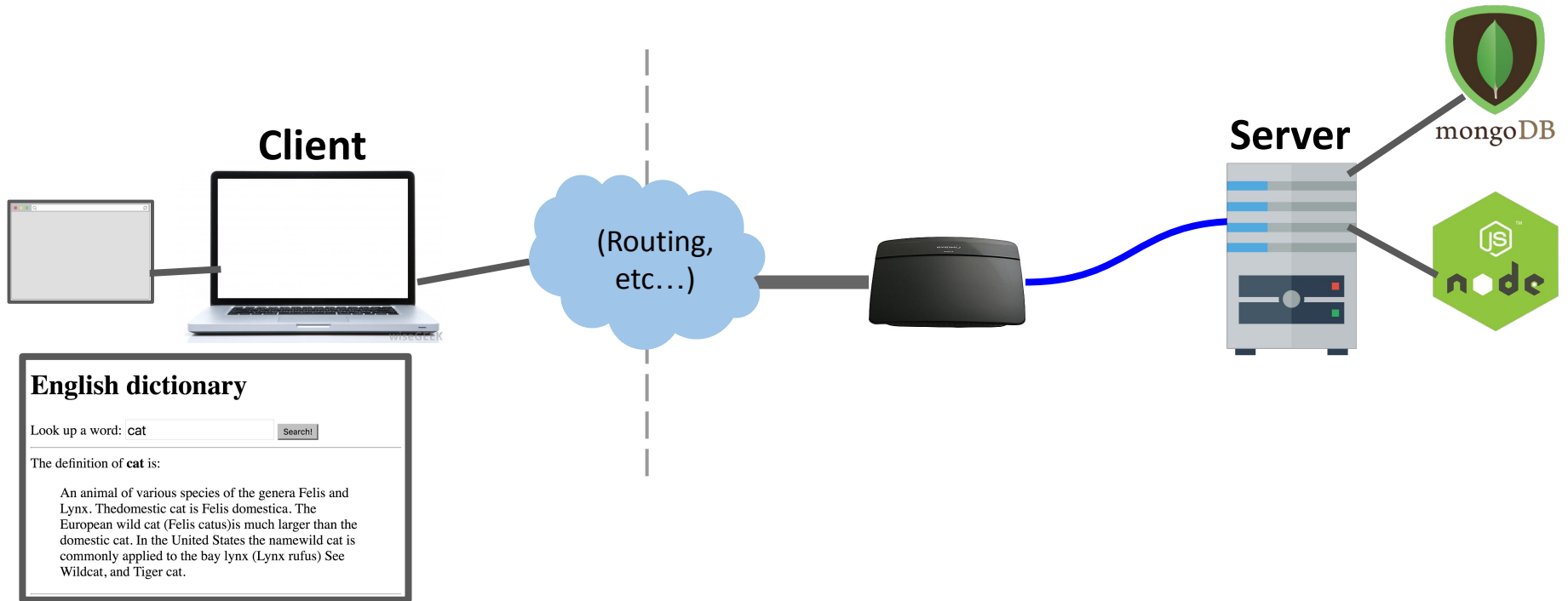
Look up a word:

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.

MIT License

Copyright (c) 2017 Victoria KIRST (vrk@stanford.edu)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.