

# Interactive Web Programming

1st semester of 2021

Murilo Camargos  
(**[murilo.filho@fgv.br](mailto:murilo.filho@fgv.br)**)

Heavily based on [Victoria Kirst](#) slides

# Today's schedule

## Today:

- Saving data
  - POST body
  - Body-parser
- Databases
  - MongoDB
  - System overview
  - mongo and mongod
  - Mongodb

## Announcements:

- The last [HW6](#) is released and due **June 23~25**

# Last time: `async / await`

What if we could get:

- Synchronous-*looking* code
- That actually ran asynchronously?

**// THIS CODE DOESN'T WORK**

```
const response = fetch('albums.json');  
const json = response.json();  
console.log(json);
```

# async / await

What if we could get the best of both worlds?

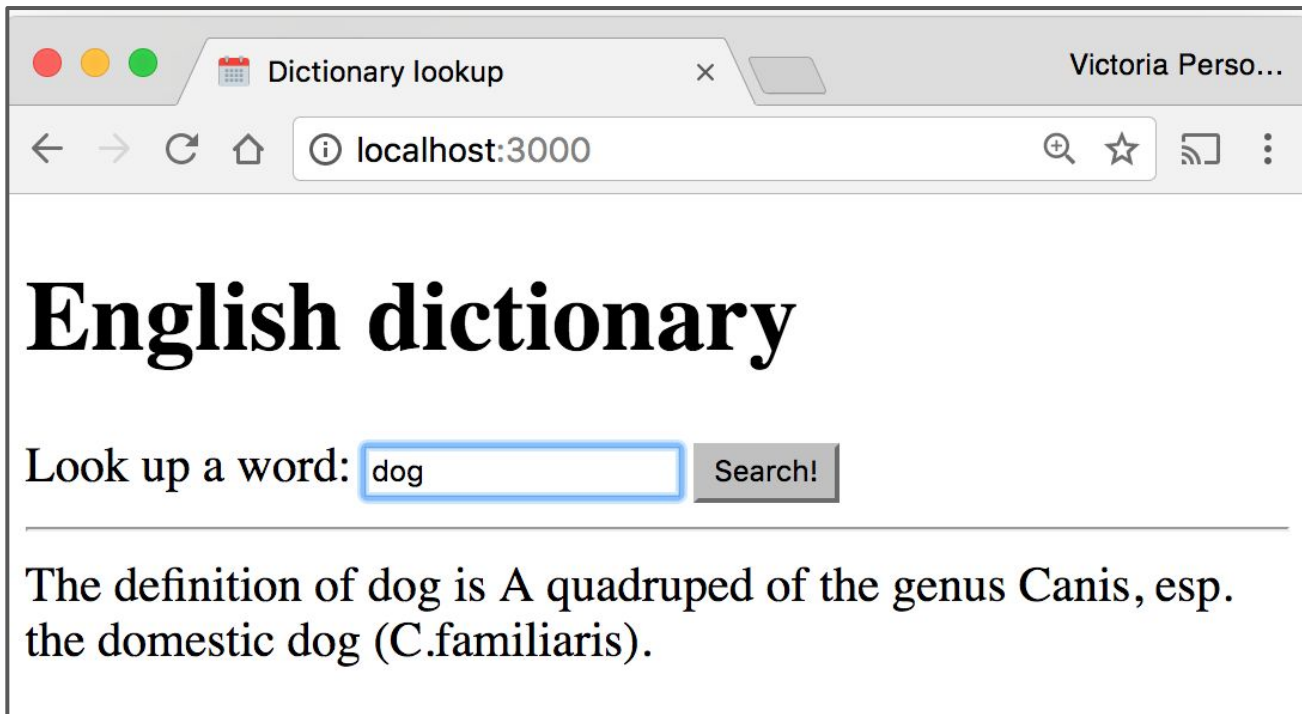
- Synchronous-*looking* code
- That actually ran asynchronously

**// But this code does work:**

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();
```

# Example: Dictionary

Given a `dictionary.json` file of word/value pairs, a dictionary app that lets you look up the definition of the word:



# Dictionary lookup

```
// Load a JSON file containing english words.
const englishDictionary = require('./dictionary.json');

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

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

  const key = word.toLowerCase();
  const definition = englishDictionary[key];

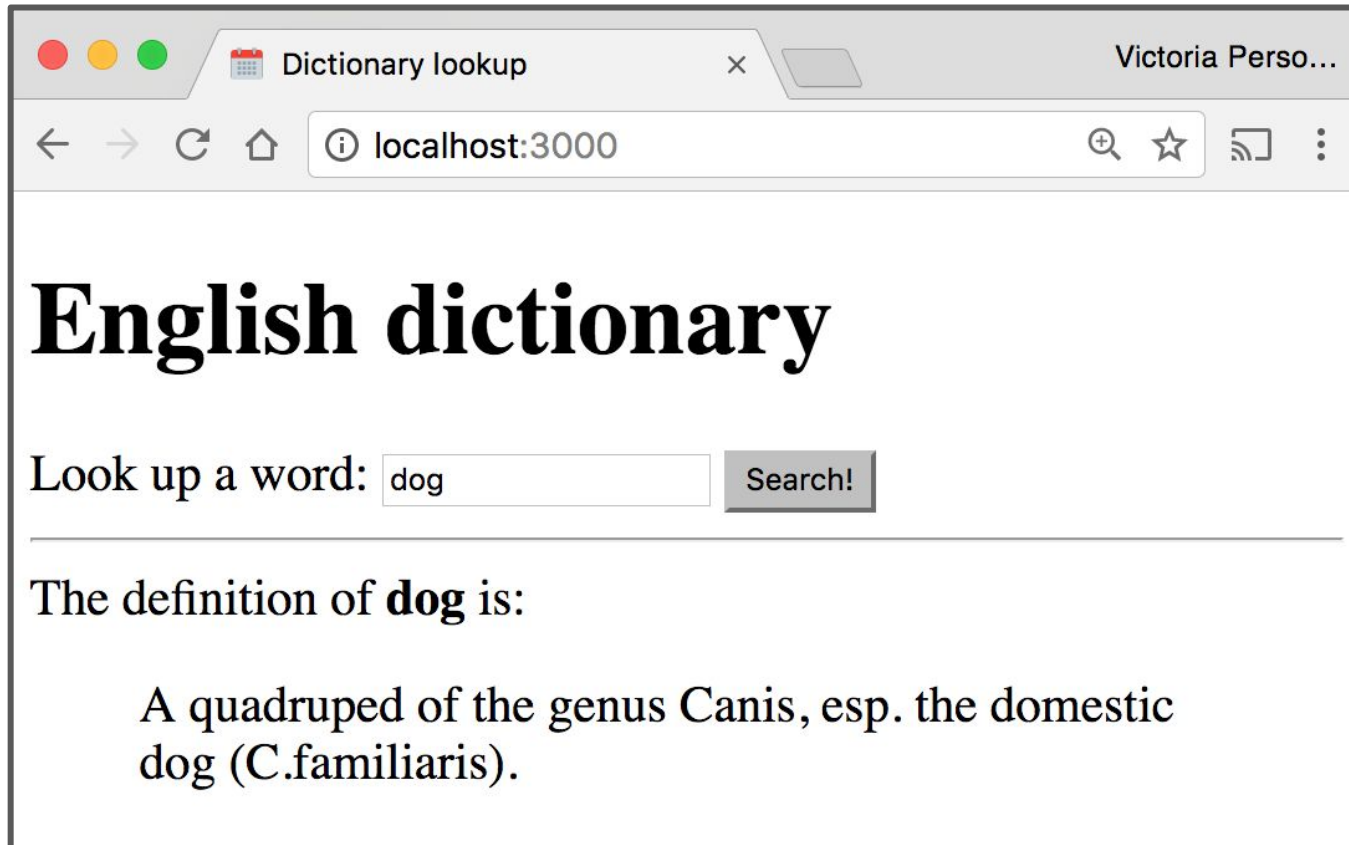
  res.send(`The definition of ${word} is ${definition}`);
}
app.get('/print/:word', onPrintWord);
```

# Dictionary fetch

```
async function onSearch(event) {  
  event.preventDefault();  
  const input = document.querySelector('#word-input');  
  const word = input.value.trim();  
  const result = await fetch('/print/' + word);  
  const text = await result.text();  
  
  const results = document.querySelector('#results');  
  results.innerHTML = text;  
}  
  
const form = document.querySelector('#search');  
form.addEventListener('submit', onSearch);
```

# Example: Dictionary

It'd be nice to have some flexibility on the display of the definition:





# JSON response

If we want to return a JSON response, we should use `res.json(object)` instead:

```
app.get('/', function (req, res) {  
  const response = {  
    greeting: 'Hello World!',  
    awesome: true  
  }  
  res.json(response);  
});
```

The parameter we pass to [res.json\(\)](#) should be a JavaScript object.

# Example: 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);
```

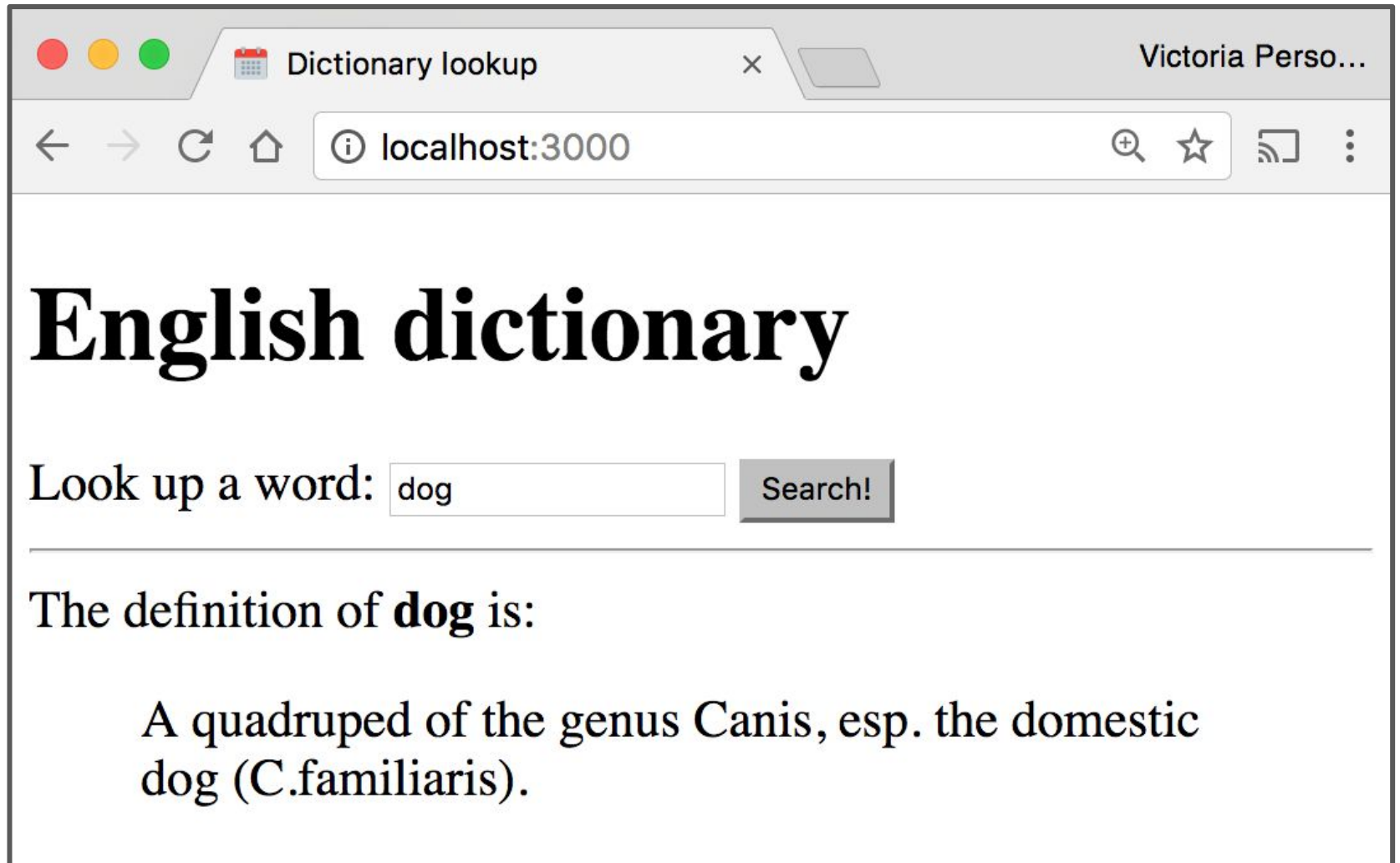
# Example: Dictionary fetch

```
async function onSearch(event) {
  event.preventDefault();
  const input = document.querySelector('#word-input');
  const word = input.value.trim();

  const results = document.querySelector('#results');
  results.classList.add('hidden');
  const result = await fetch('/lookup/' + word);
  const json = await result.json();

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

# Result



The image shows a browser window with the title "Dictionary lookup" and the user name "Victoria Perso...". The address bar shows "localhost:3000". The main content area displays the title "English dictionary" in a large, bold, serif font. Below the title is a search form with the text "Look up a word:" followed by an input field containing the word "dog" and a "Search!" button. A horizontal line separates the search form from the definition. The definition text reads: "The definition of **dog** is: A quadruped of the genus *Canis*, esp. the domestic dog (*C.familiaris*)." The word "dog" in the definition is bolded.

Dictionary lookup

Victoria Perso...

localhost:3000

# English dictionary

Look up a word:  Search!

---

The definition of **dog** is:

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

# Errata from last class

**Can I use await outside async functions?**

# Errata from last class

**Can I use await outside async functions?**

**The answer is NO!**

You can only use await inside async functions or in the top level bodies of modules.

# Errata from last class

Can I use await outside async functions?

**The answer is NO!**

You can only use await inside async functions or in the top level bodies of modules.

You can also use it in the chrome REPL

```
> const search = await fetch('https://api.tvmaze.com/search/shows?q=Witcher')
< undefined
> search
< ▶ Response {type: "cors", url: "https://api.tvmaze.com/search/shows?q=Witcher",
  redirected: false, status: 200, ok: true, ...}
```

# Errata from last class

Can I use await outside async functions?

**The answer is NO!**

Work around:

```
(async () => {
  try {
    var text = await main();
    console.log(text);
  } catch (e) {
    // Deal with the fact the chain failed
  }
})();
```



Posting data

# Example: Dictionary

What if we want to  
modify the definitions  
of words as well?



The screenshot shows a web browser window titled "Dictionary lookup" with the URL "localhost:3000". The page content is as follows:

## 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*).

# MongoDB installation

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

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

# POST message body: `fetch()`

## Client-side:

You should specify a **message body** in your `fetch()` call:

```
const message = {
  name: 'Victoria',
  email: 'vrk@stanford.edu'
};
const serializedMessage = JSON.stringify(message);
fetch('/helloemail', { method: 'POST', body: serializedMessage })
  .then(onResponse)
  .then(onTextReady);
```

# Server-side

**Server-side:** Handling the message body in NodeJS/Express is a little messy ([GitHub](#)):

```
app.post('/helloemail', function (req, res) {
  let data = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk) {
    data += chunk;
  });

  req.on('end', function() {
    const body = JSON.parse(data);
    const name = body.name;
    const email = body.email;
    res.send('POST: Name: ' + name + ', email: ' + email);
  });
});
```

# body-parser

We can use the [body-parser library](#) to help:

```
const bodyParser = require('body-parser');
```

This is not a NodeJS API library, so we need to install it:

```
$ npm install body-parser
```

# body-parser

We can use the [body-parser library](#) to help:

```
const bodyParser = require('body-parser');  
const jsonParser = bodyParser.json();
```

This creates a JSON parser stored in `jsonParser`, which we can then pass to routes whose message bodies we want parsed as JSON.

# POST message body

Now instead of this code:

```
app.post('/helloemail', function (req, res) {
  let data = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk) {
    data += chunk;
  });

  req.on('end', function() {
    const body = JSON.parse(data);
    const name = body.name;
    const email = body.email;
    res.send('POST: Name: ' + name + ', email: ' + email);
  });
});
```



# POST message body

We can write this code:

```
app.post('/helloparsed', jsonParser, function (req, res) {  
  const body = req.body;  
  const name = body.name;  
  const email = body.email;  
  res.send('POST: Name: ' + name + ', email: ' + email);  
});
```

[GitHub](#)

# POST message body

We can access the message body through `req.body`:

```
app.post('/helloparsed', jsonParser, function (req, res) {  
  const body = req.body;  
  const name = body.name;  
  const email = body.email;  
  res.send('POST: Name: ' + name + ', email: ' + email);  
});
```

[GitHub](#)

# POST message body

We can access the message body through `req.body`:

```
app.post('/helloparsed', jsonParser, function (req, res) {  
  const body = req.body;  
  const name = body.name;  
  const email = body.email;  
  res.send('POST: Name: ' + name + ', email: ' + email);  
});
```

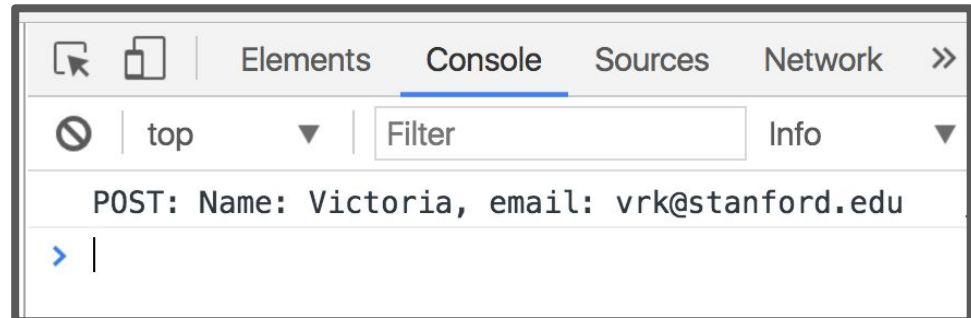
[GitHub](#)

Note that we also had to add the `jsonParser` as a parameter when defining this route.

# POST message body

Finally, we need to add JSON content-type headers on the `fetch()`-side ([GitHub](#)):

```
const message = {
  name: 'Victoria',
  email: 'vrk@stanford.edu'
};
const fetchOptions = {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(message)
};
fetch('/helloparsed', fetchOptions)
  .then(onResponse)
  .then(onTextReady);
```



# POST message body

From Express v4.16.0 onwards, we can simplify this by using the built-in JSON middleware:

```
const router = express.Router();  
router.use(express.json());
```

# POST message body

From Express v4.16.0 onwards, we can simplify this by using the built-in JSON middleware:

```
const router = express.Router();  
router.use(express.json());
```

Now, every endpoint you create within this route will use this middleware to transform the request body into an object:

```
router.post('/show', (req, res) => {  
  const tvmaze_id = req.body.tvmaze_id;
```

# Example: Dictionary

We will modify the dictionary example to POST the contents of the form.



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:

# fs-extra

We'll use the `fs-extra` library to write our change back to the `dictionary.json` file.

- [fs](#): NodeJS API library
  - Uses callbacks
- [fs-extra](#): npm library
  - Uses callbacks OR promises
  - `fs.writeFileSync(fileName, object)`



# Example: server-side

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

# Example: fetch()

```
async function onSet(event) {
  event.preventDefault();
  const setWordInput = results.querySelector('#set-word-input');
  const setDefInput = results.querySelector('#set-def-input');
  const word = setWordInput.value;
  const def = setDefInput.value;

  const message = {
    definition: def
  };
  const fetchOptions = {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(message)
  };
  await fetch('/set/' + word, fetchOptions);
}
```

Query parameters

# Query parameters

The Spotify Search API was formed using query parameters:

Example: Spotify Search API

[https://api.spotify.com/v1/search?type=album  
&q=beyonce](https://api.spotify.com/v1/search?type=album&q=beyonce)

- There were two query parameters sent to the Spotify search endpoint:
  - type, whose value is album
  - q, whose value is beyonce

# Query parameters

**Q: How do we read query parameters in our server?**

**A: We can access query parameters via `req.query`:**

```
app.get('/hello', function (req, res) {  
  const queryParams = req.query;  
  const name = queryParams.name;  
  res.send('GET: Hello, ' + name);  
});
```



# Recap

You can deliver parameterized information to the server in the following ways:

1. Route parameters
2. GET request with query parameters  
(**DISCOURAGED**: POST with query parameters)
3. POST request with message body

**Q: When do you use route parameters vs query parameters vs message body?**

# GET vs POST

- Use [GET](#) requests for retrieving data, not writing data
  - Use [POST](#) requests for writing data, not retrieving data
- You can also use more specific HTTP methods:
- PATCH: Updates the specified resource
  - DELETE: Deletes the specified resource

There's nothing technically preventing you from breaking these rules, but you should use the HTTP methods for their intended purpose.

# Route params vs Query params

Generally follow these rules:

- Use **route parameters** for required parameters for the request
- Use **query parameters** for:
  - Optional parameters
  - Parameters whose values can have spaces

These are conventions and are not technically enforced, nor are they followed by every REST API.



# Example: Spotify API

The Spotify API mostly followed these conventions:

<https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9>

- The Album ID is required and it is a route parameter.

<https://api.spotify.com/v1/search?type=album&q=the%20weeknd&limit=10>

- q is required but might have spaces, so it is a query parameter
- limit is optional and is a query parameter
- type is required but is a query parameter (breaks convention)

Notice both searches are GET requests, too

package.json

# Installing dependencies

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

```
$ npm install express
```

```
$ npm install body-parser
```

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, you should pass in the `--save` parameter:

```
$ npm install --save express  
$ npm install --save body-parser
```

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

```
"dependencies": {  
  "body-parser": "^1.17.1",  
  "express": "^4.15.2"  
},
```



# 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
```

# Databases and DBMS

# 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.

# Why use a database/DBMS

Why use a DBMS instead of saving to a JSON file?

- **fast**: can search/filter a database quickly compared to a file
- **scalable**: can handle very large data sizes
- **reliable**: mechanisms in place for secure transactions, backups, etc.
- **built-in features**: can search, filter data, combine data from multiple sources
- **abstract**: provides layer of abstraction between stored data and app(s)
  - Can change **where** and **how** data is stored without needing to change the code that connects to the database.

# Why use a database/DBMS

Why use a DBMS instead of saving to a JSON file?

- Also: Some services like Heroku will not permanently save files, so using `fs` or `fs-extra` **will not work**

# Disclaimer

Databases and DBMS is a huge topic in CS with multiple courses dedicated to it:

- Introduction to Databases
- Database System Principles
- Database System Implementation
- etc....

**In this class, we will cover only the very basics:**

- How one particular DBMS works (MongoDB)
- How to use MongoDB with NodeJS
- (later) Basic DB design

MongoDB



# 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

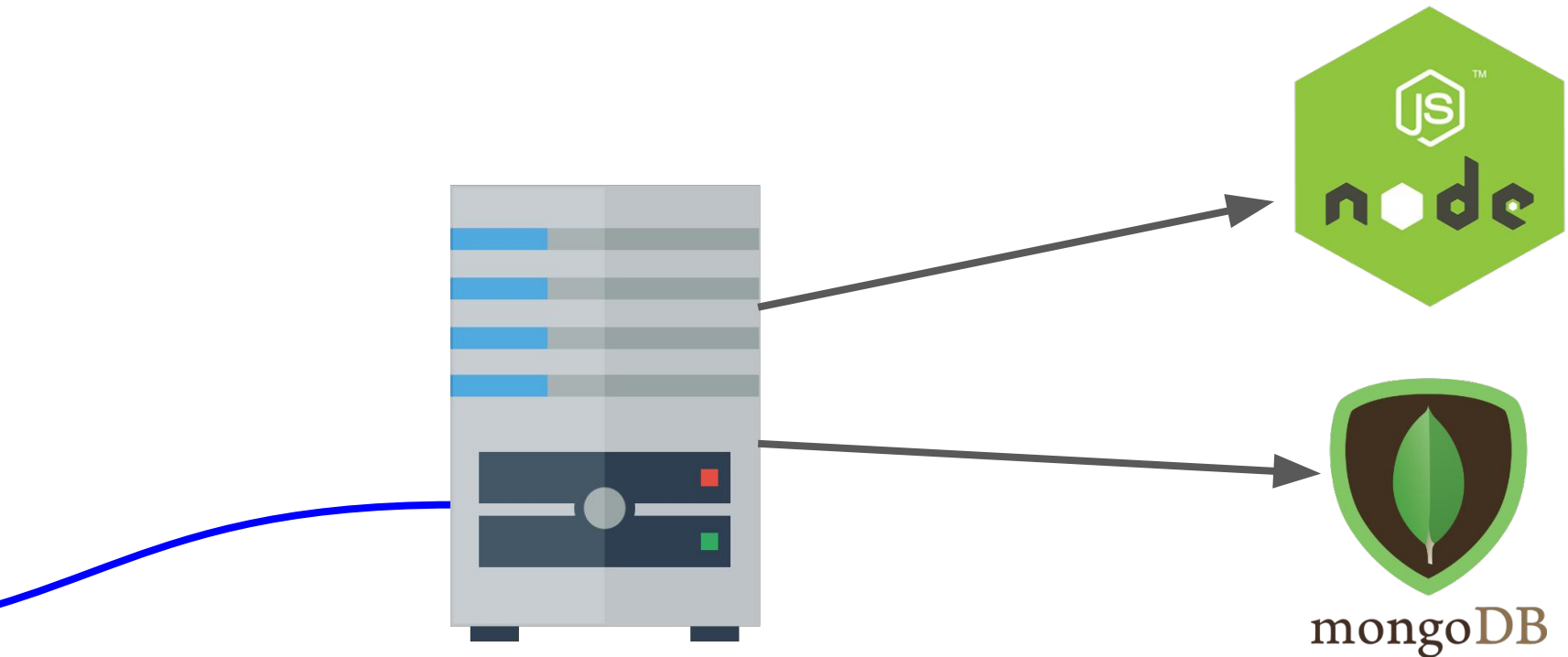
[Relational databases](#) have fixed schemas;

[document-oriented databases](#) have

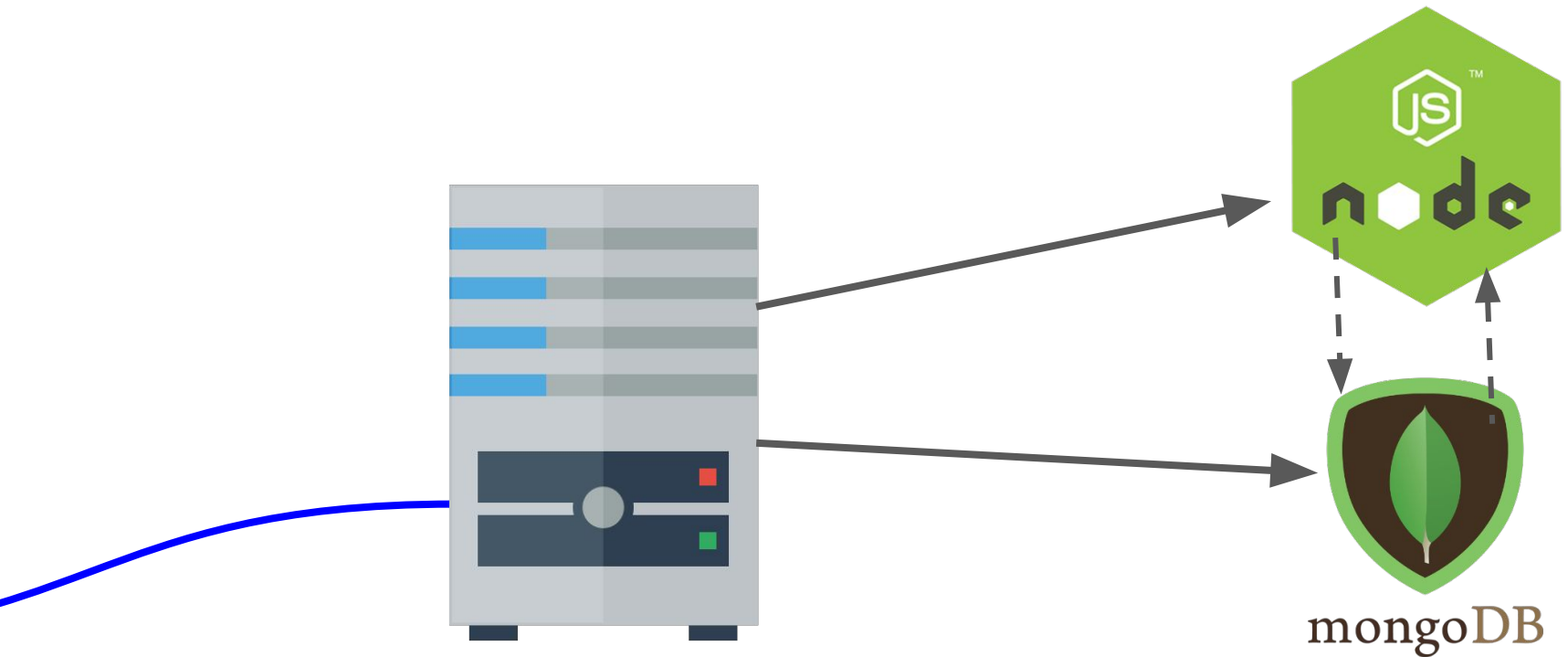
flexible schemas

## Document-oriented DB:

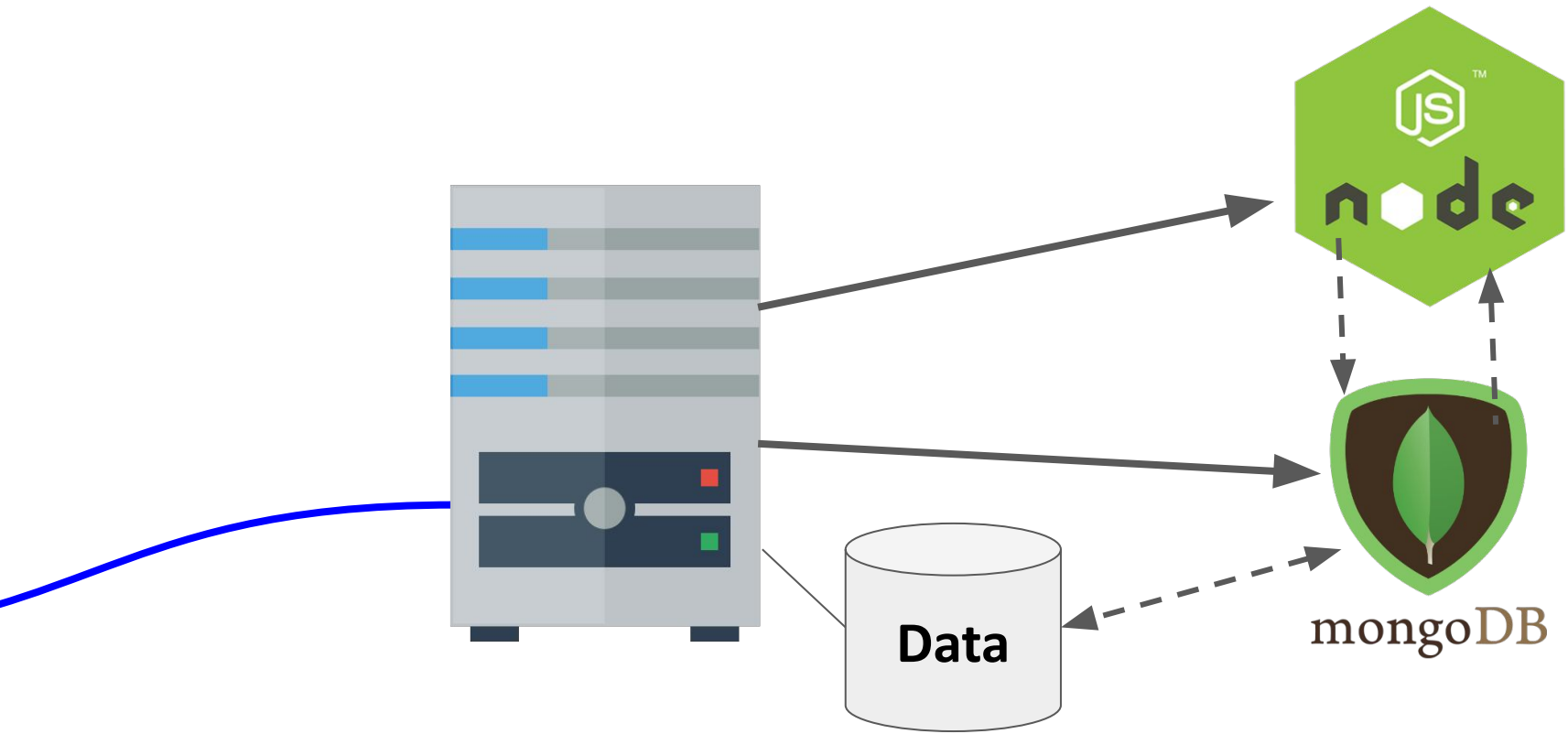
```
{
  name: "Lori",
  employer: "Self",
  occupation: "Entrepreneur"
}
{
  name: "Malia",
  school: "Harvard"
}
```



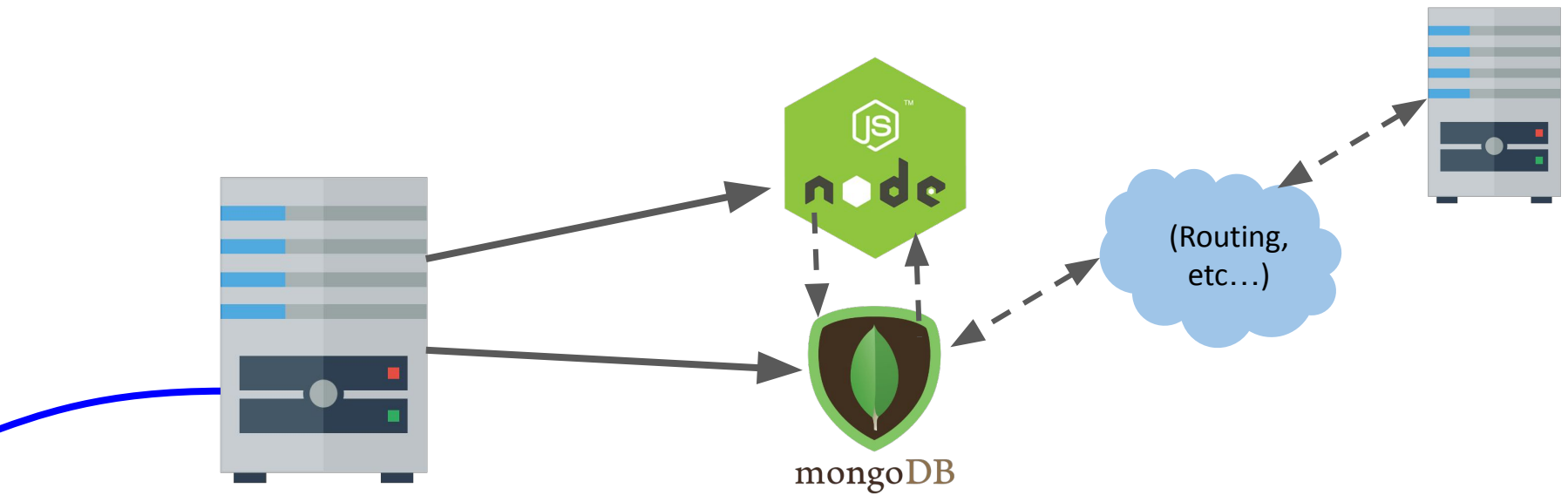
MongoDB is another **software program** running on the computer, alongside our NodeJS server program. It is also known as the **MongoDB server**.



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.

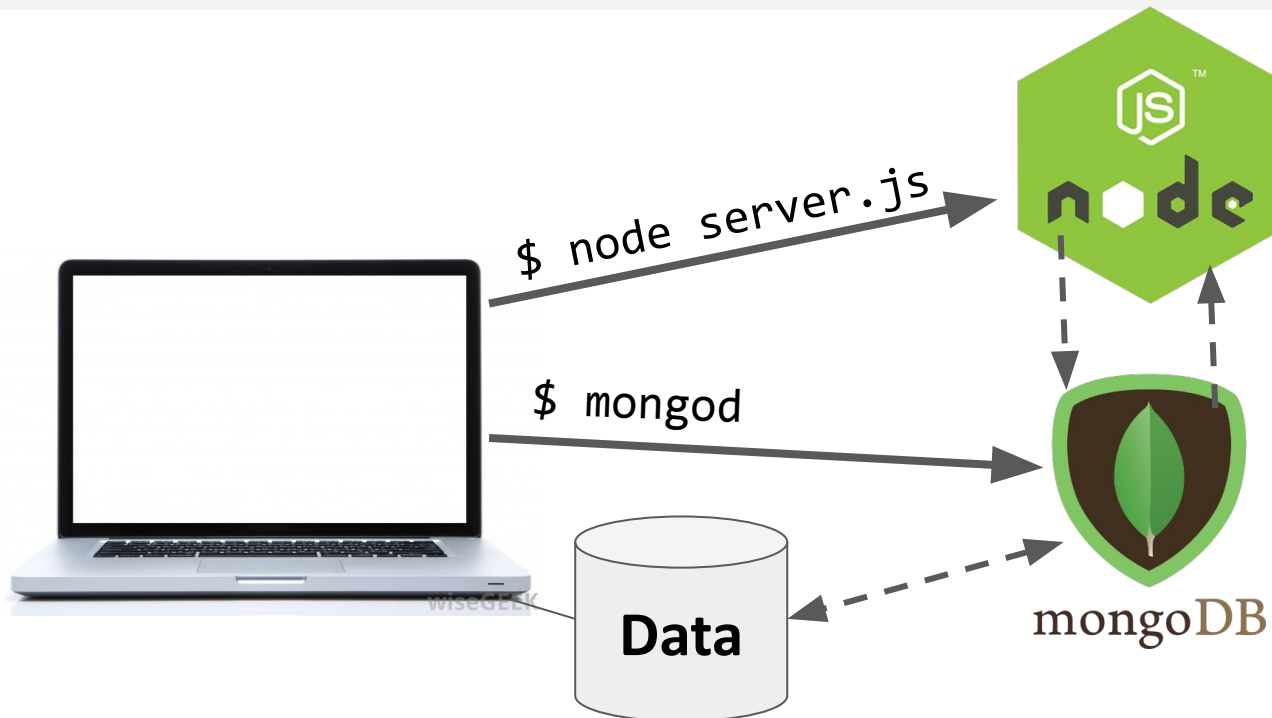


The database the MongoDB Server manages might be local to the server computer...



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

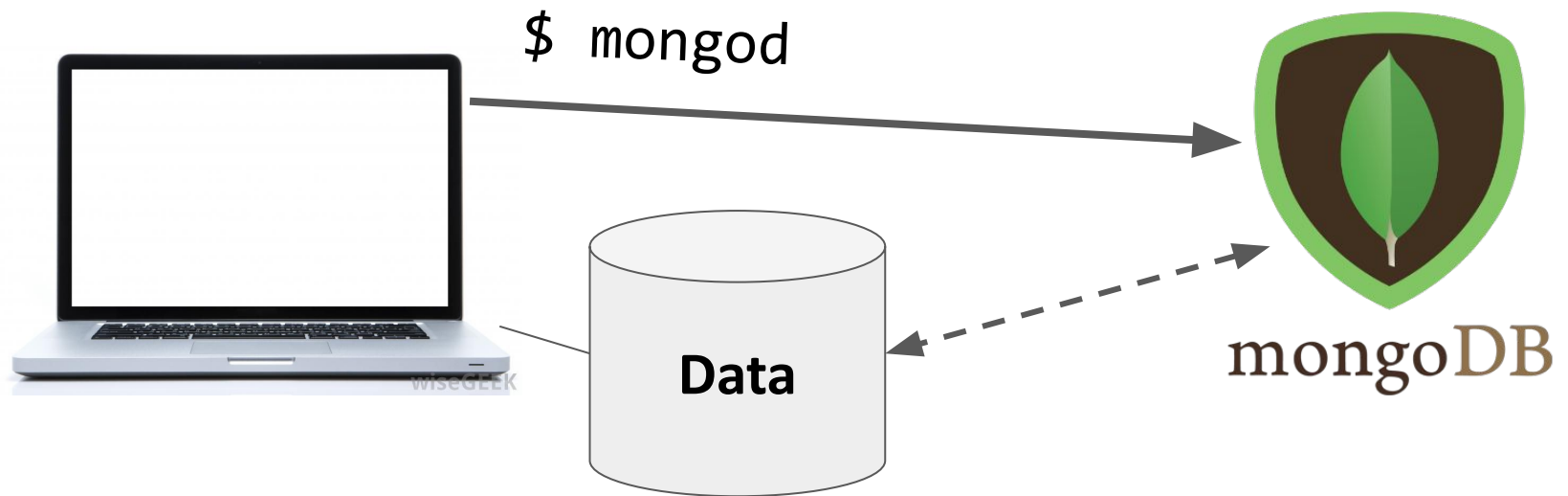
# System overview



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**

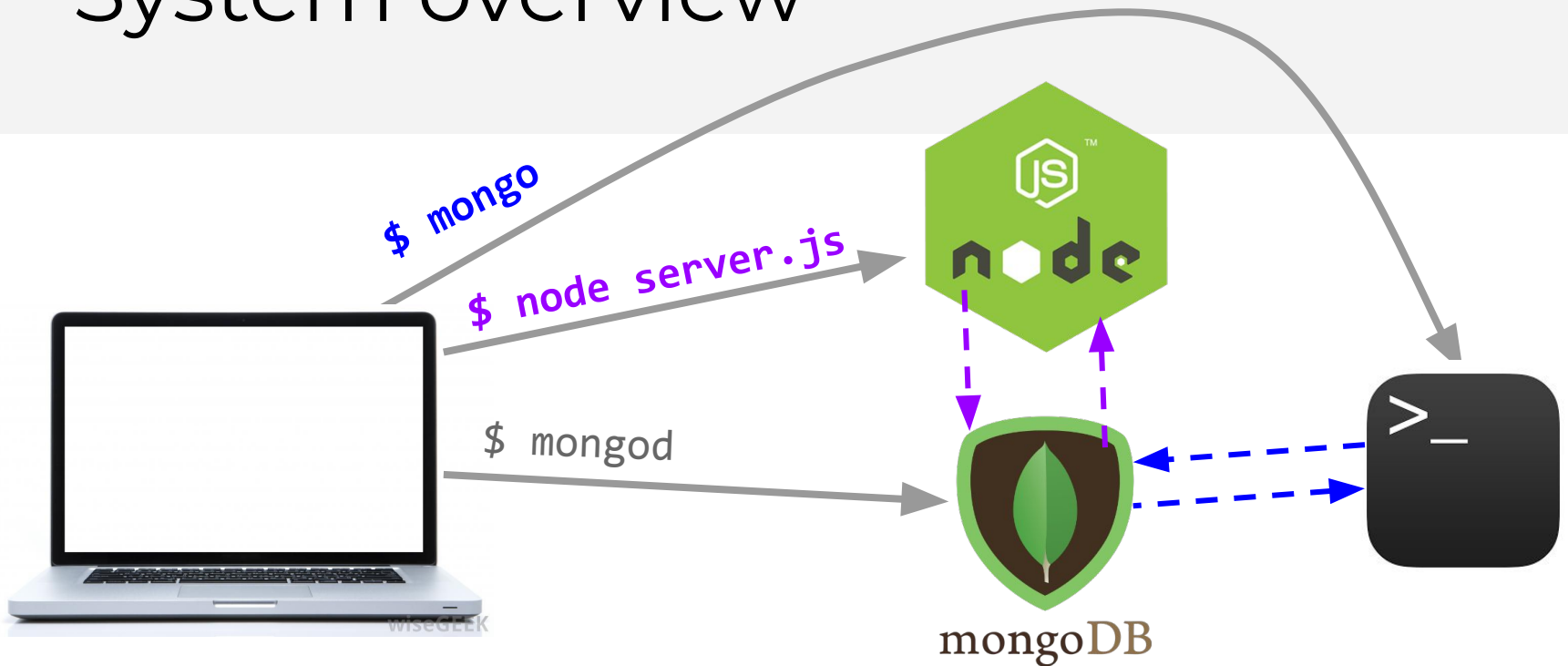
# System overview



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

## Documents:

```
{ "_id" :  
  ObjectId("5922b8a186ebd7  
3e42b1b53c"), "style" :  
  "july4", "message" :  
  "Dear Chip,\n\nHappy 4th  
of July!\n\n♥Dale" }
```

```
{ "_id" :  
  ObjectId("5922acf09e7640  
3b3a7549ec"), "style" :  
  "graduation", "message"  
: "Hi Pooh,\n\n🎉  
Congrats!!! 🎉\n\n<3  
Piglet" }
```

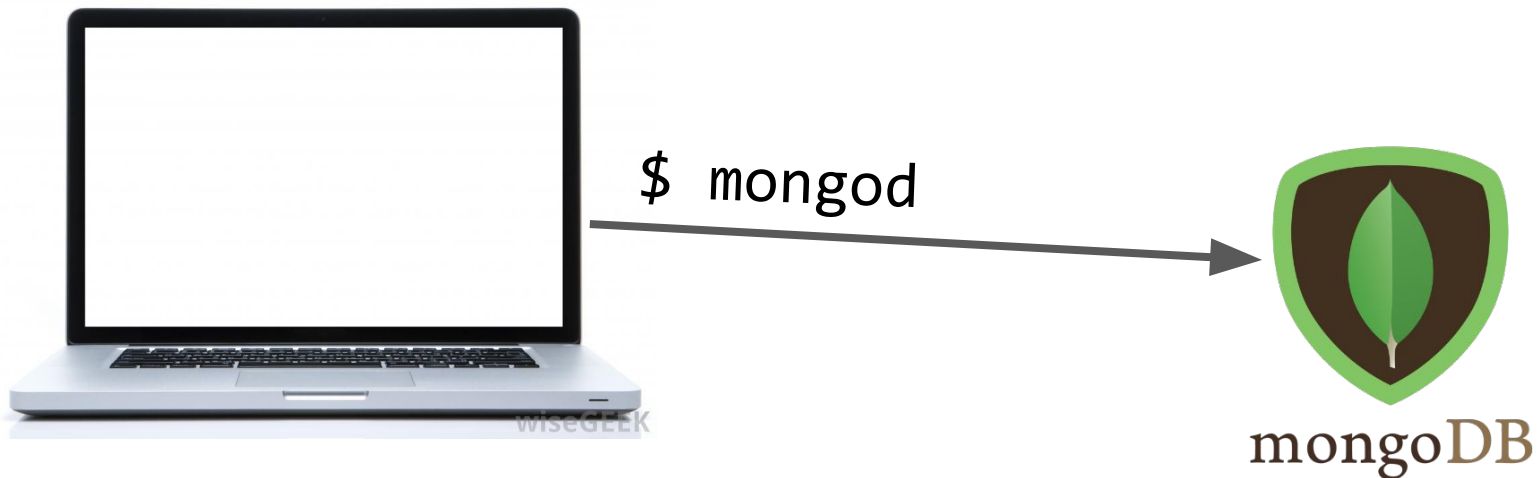
```
{ "_id" :  
  ObjectId("5922b90d86ebd7  
3e42b1b53d"), "style" :  
  "fathersday", "message"  
: "HFD" }
```

**Database:**  
ecards-db

**Collection:**  
card

The document keys are  
called **fields**

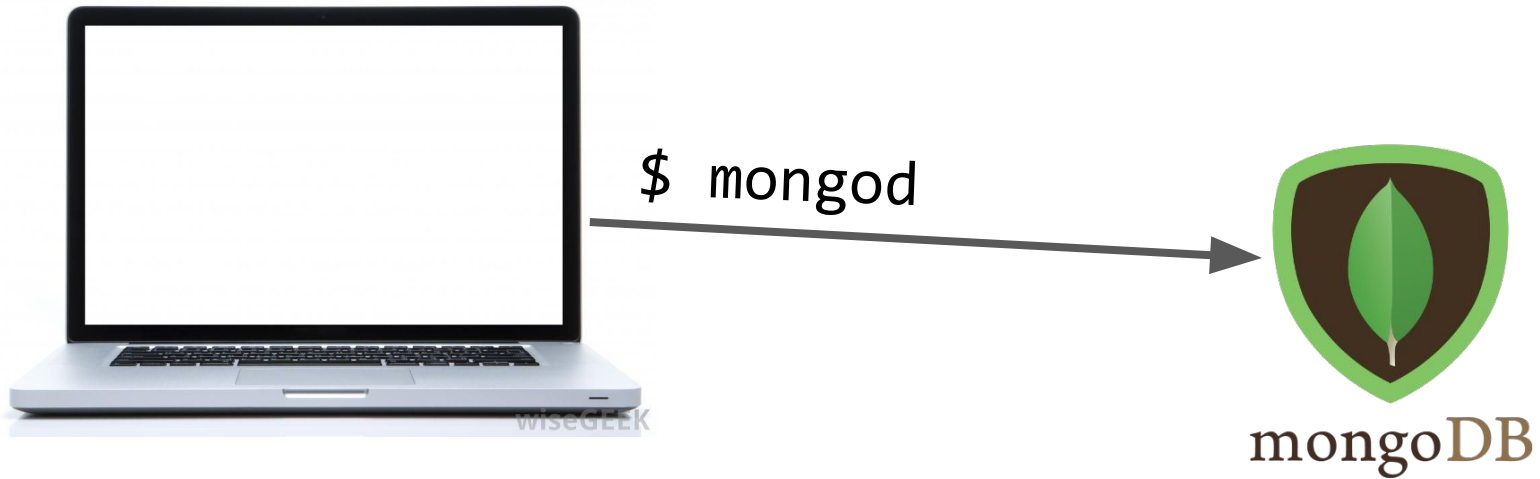
# mongod: Database process



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

More next time!