Schedule

Today:

- `async/await`: A JavaScript language feature
  - **Not** Node-specific!
- Sending data to the server
- Returning JSON
- Saving data
  - POST body
  - `body-parser`
- Databases
  - MongoDB

Schedule

Next week:

- MongoDB with server
- One page website
  - Server rendering - Handlebars

Note

- In Wed (Oct 09):
  - Assignment 2 will be released
    - Due in 3 weeks (Oct 31)
  - Assignment 1: plagiarism-check results will be available

async/await

Two types of asynchrony

We have been working with two broad types of asynchronous events:

1. **Inherently asynchronous events**
- Example: `addEventListener('click')`. There is no such thing as a synchronous click event.

2. **Annoyingly asynchronous events**
- Example: **`fetch()`**. This function would be easier to use if it were synchronous, but for performance reasons it's asynchronous

Asynchronous `fetch()`

The usual asynchronous `fetch()` looks like this:

```
function onJsonReady(json) {
  console.log(json);
}

function onResponse(response) {
  return response.json();
}

fetch('albums.json')
    .then(onResponse)
    .then(onJsonReady);
```

Synchronous `fetch()`?

A hypothetical synchronous `fetch()` might look like this:

```
// THIS CODE DOESN'T WORK
const response = fetch('albums.json');
const json = response.json();
console.log(json);
```

This is a lot cleaner code-wise!!

However, a synchronous `fetch()` would freeze the browser as the resource was downloading, which would be terrible for performance.

async / await

What if we could get the best of both worlds?

- 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();
```

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();
```

???

`async` functions

A function marked `async` has the following qualities:

- It will behave more or less like a normal function if you don't put `await` expression in it.

- An `await` expression is of form:
  - `await` ***promise***

`async` functions

A function marked `async` has the following qualities:

- If there is an `await` expression, **the execution of the function will pause** until the `Promise` in the `await` expression is resolved.
  - Note: The browser is not blocked; it will continue executing JavaScript as the async function is paused.
- Then when the `Promise` is resolved, the execution of the function continues.
- The `await` expression evaluates to the resolved value of the `Promise`.

```
function onJsonReady(json) {
  console.log(json);
}
function onResponse(response) {
  return response.json();
}
fetch('albums.json')
    .then(onResponse)
    .then(onJsonReady);
```

The methods in purple return Promises.

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

```
function onJsonReady(json) {
  console.log(json);
}
function onResponse(response) {
  return response.json();
}
fetch('albums.json')
    .then(onResponse)
    .then(onJsonReady);
```

The variables in blue are the values that the `Promises` "resolve to".

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

async functions

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

async functions

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

async functions

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

Since we've reached an `await` statement, two things happen:

1. `fetch('albums.json');` runs
2. The execution of the `loadJson` function is paused here until `fetch('albums.json');` has completed.

async functions

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

At the point, the JavaScript engine will return from `loadJson()` and it will continue executing where it left off.

async functions

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

async functions

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

async functions

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

async functions

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

If there are other events, like if a button was clicked and we had a event handler for it, JavaScript will continue executing those events.

async functions

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

When the `fetch()` completes, the JavaScript engine will resume execution of `loadJson()`.

Recall: `fetch()` resolution

```
function onResponse(response) {
  return response.json();
}
fetch('albums.json')
    .then(onResponse)
```

Normally when `fetch()` finishes, it executes the `onResponse` callback, whose parameter will be response.

In `Promise`-speak:

- The return value of `fetch()` is a `Promise` that **resolves to** the `response` object.

async functions

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

The value of the `await` expression is the value that the `Promise` resolves to, in this case `response`.

async functions

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

async functions

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

Since we've reached an `await` statement, two things happen:

1. `response.json();` runs
2. The execution of the `loadJson` function is paused here until `response.json();` has completed.

async functions

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

If there are other events, like if a button was clicked and we had a event handler for it, JavaScript will continue executing those events.

async functions

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

async functions

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

When the `response.json()` completes, the JavaScript engine will resume execution of `loadJson()`.

Recall: `json()` resolution

```
function onJsonReady(jsObj) {
  console.log(jsObj);
}
function onResponse(response) {
  return response.json();
}
fetch('albums.json')
    .then(onResponse)
    .then(onJsonReady);
```

Normally when `json()` finishes, it executes the `onJsonReady` callback, whose parameter will be **`jsObj`**.

In `Promise`-speak:

- The return value of `json()` is a `Promise` that **resolves to** the **`jsObj`** object.

async functions

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

The value of the `await` expression is the value that the `Promise` resolves to, in this case `json`.

async functions

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

async functions

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

async functions

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

Note that the JS execution does *not* return back to the call site, since the JS execution already did that when we saw the first `await` expression.

Returning from async

Q: What happens if we return a value from an async function?

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

Returning from async

A: async functions must always return a `Promise`.

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

If you return a value that is **not** a `Promise` (such as `true`), then the JavaScript engine will automatically wrap the value in a `Promise` that resolves to the value you returned.

Returning from async

```
function loadJsonDone(value) {
  console.log('loadJson complete!');
  // Prints "value: true"
  console.log('value: ' + value);
}

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

More `async`

- Constructors cannot be marked `async`

- But you can pass `async` functions as parameters to:
  - `addEventListener` (Browser)
  - on (NodeJS)
  - `get/put/delete`/etc (ExpressJS)
  - Wherever you can pass a function as a parameter

`async` / `await` availability

Browsers:

- All major browsers support `async /await`

NodeJS:

- `async /await` available in v7.5+...

(FYI, underneath the covers `async/await` is implemented by generator functions, another functional programming construct)

Sending data to the server

Route parameters

When we used the Spotify API, we saw a few ways to send information to the server via our `fetch()` request.

Example: Spotify Album API

`https://api.spotify.com/v1/albums/`**`7aDBFWp72Pz4NZEtVBANi9`**

- The last part of the URL is a **parameter** representing the album id, `7aDBFWp72Pz4NZEtVBANi9`

A parameter defined in the URL of the request is often called a "**route parameter**."

Route parameters

Q: How do we read route parameters in our server?

A: We can use the **:*variableName*** syntax in the path to specify a route parameter (Express docs):

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

We can access the route parameters via **`req.params`**.

Route parameters

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

GitHub

Route parameters

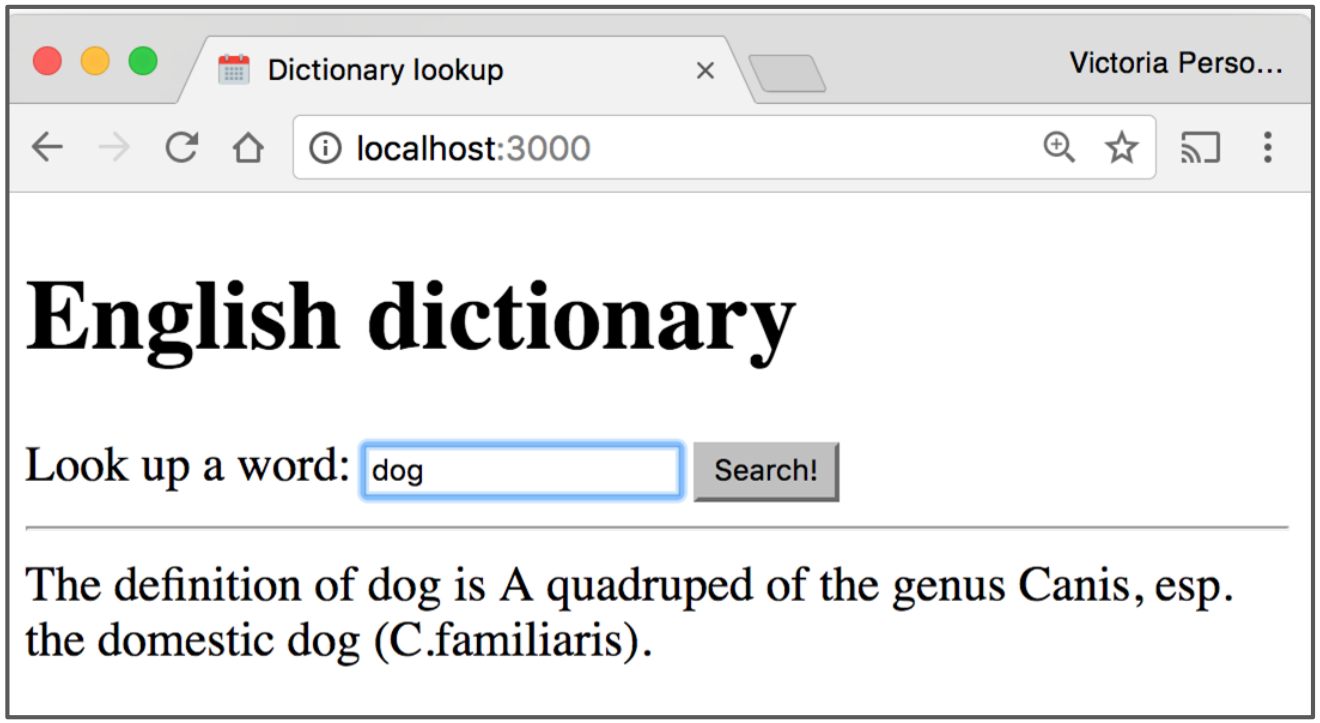
You can define multiple route parameters in a URL (docs):

```
app.get('/flights/:from-:to', function (req, res) {
  const routeParams = req.params;
  const from = routeParams.from;
  const to = routeParams.to;
  res.send('GET: Flights from ' + from + ' to ' + to);
});
```

GitHub

Example: Dictionary

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



Selected topic *:
Express Routes

* To understand code in some next slides

Recall: ExpressJS routes

We've been seeing ExpressJS routes that look like this, with an anonymous function parameter:

```
app.get('/', function(req, res) {
  // ...
});
```

ExpressJS routes

Of course, they can also be written like this, with a named function parameter:

```
function onGet(req, res) {
  // ...
}
app.get('/', onGet);
```

One more random thing: Template Literals

Template literals

Template literals allow you to embed expressions in JavaScript strings:

```
const port = process.env.PORT || 3000;
app.listen(port, function () {
  console.log('Server listening on port ' + port + '!');
});
```

```
const port = process.env.PORT || 3000;
app.listen(port, function () {
  console.log(`Server listening on port ${port}!`);
});
```

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

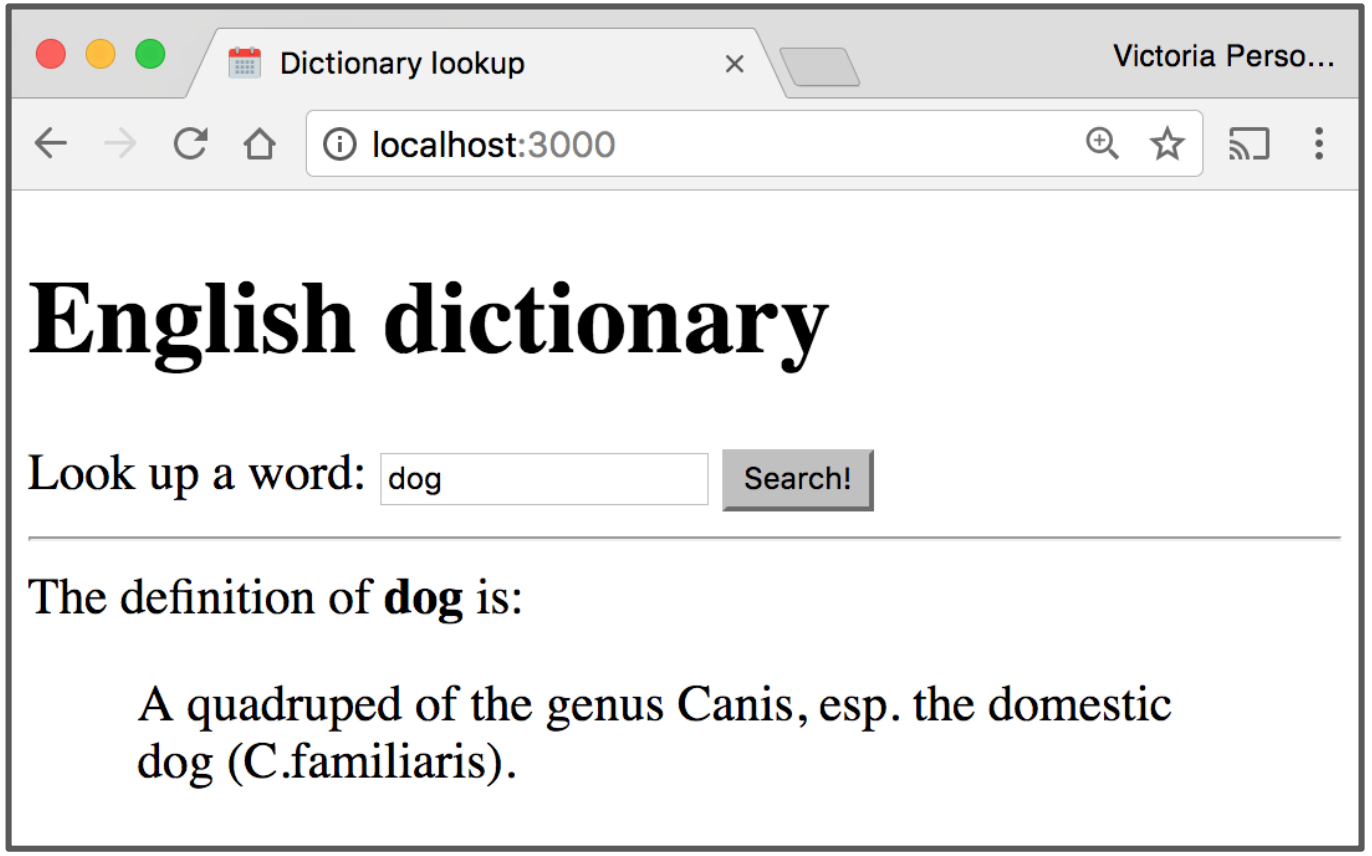
```
async function onSearch(event) {
  event.preventDefault();
  const input = document.querySelector('#word-input');
  const word = input.value.trim();
  const result = await fetch('/print/' + word);
  const text = await result.text();

  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:



Returning JSON from the server

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;
}
```

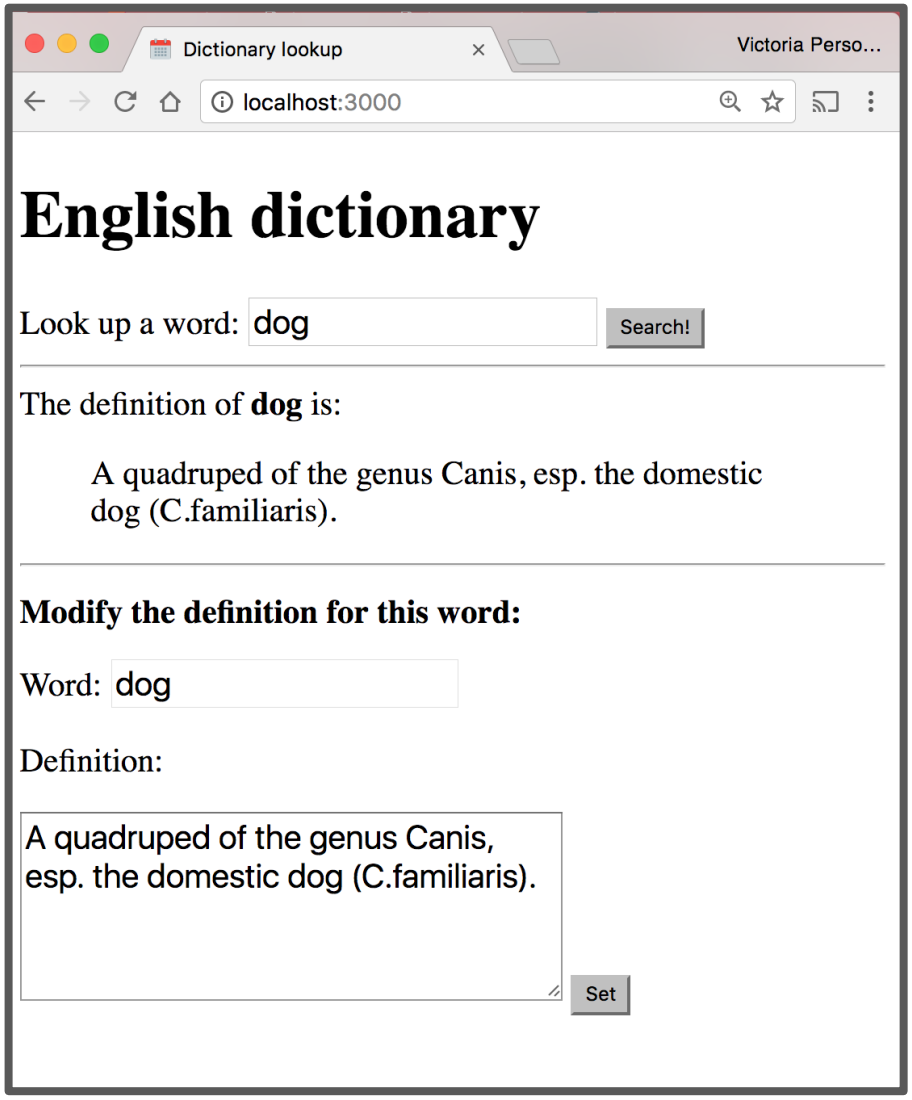
Result


Dictionary lookup
Victoria Perso...
localhost:3000
English dictionary
Look up a word:
dog
Search!
The definition of dog is:
A quadruped of the genus Canis, esp. the domestic dog (C.familiaris).

Saving data

Example: Dictionary

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



Posting data

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 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);
});
```

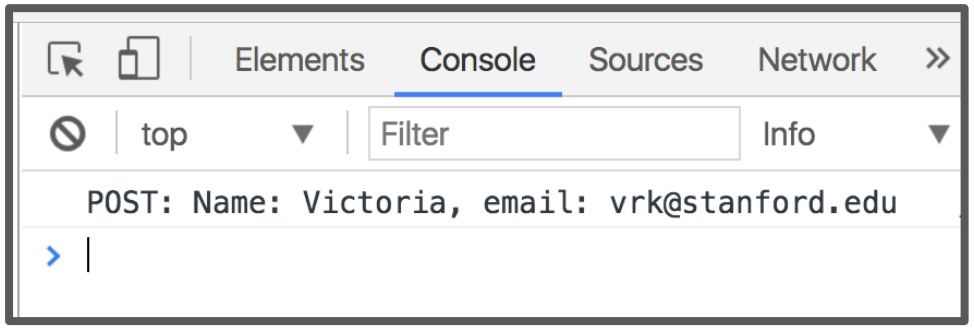
GitHub

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

POST message body

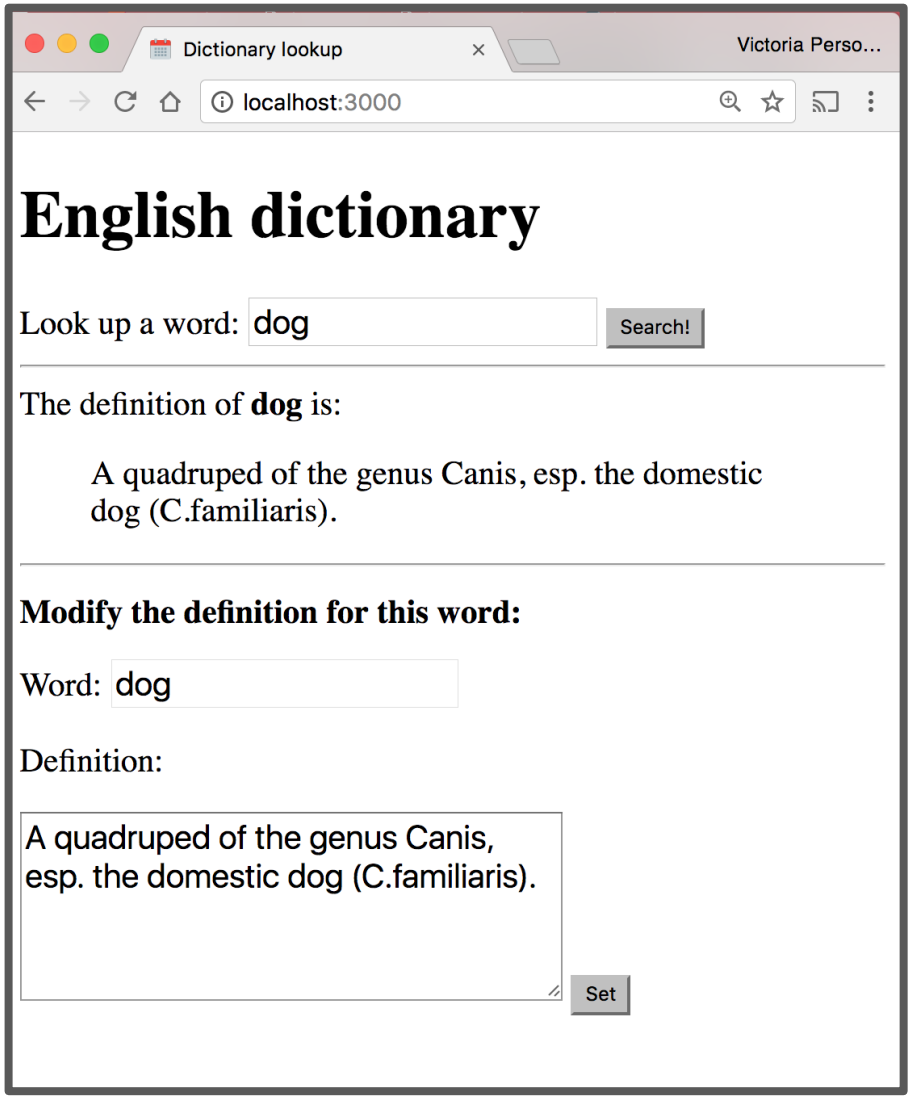
Finally, we need to add JSON content-type headers on the `fetch()`-side (GitHub):

```
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);
```



Example: Dictionary

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



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`

- 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);
});
```

GitHub

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)
1. 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

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:

- DBS: Database System
- DBA: Database System Implementation

In WPR, 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

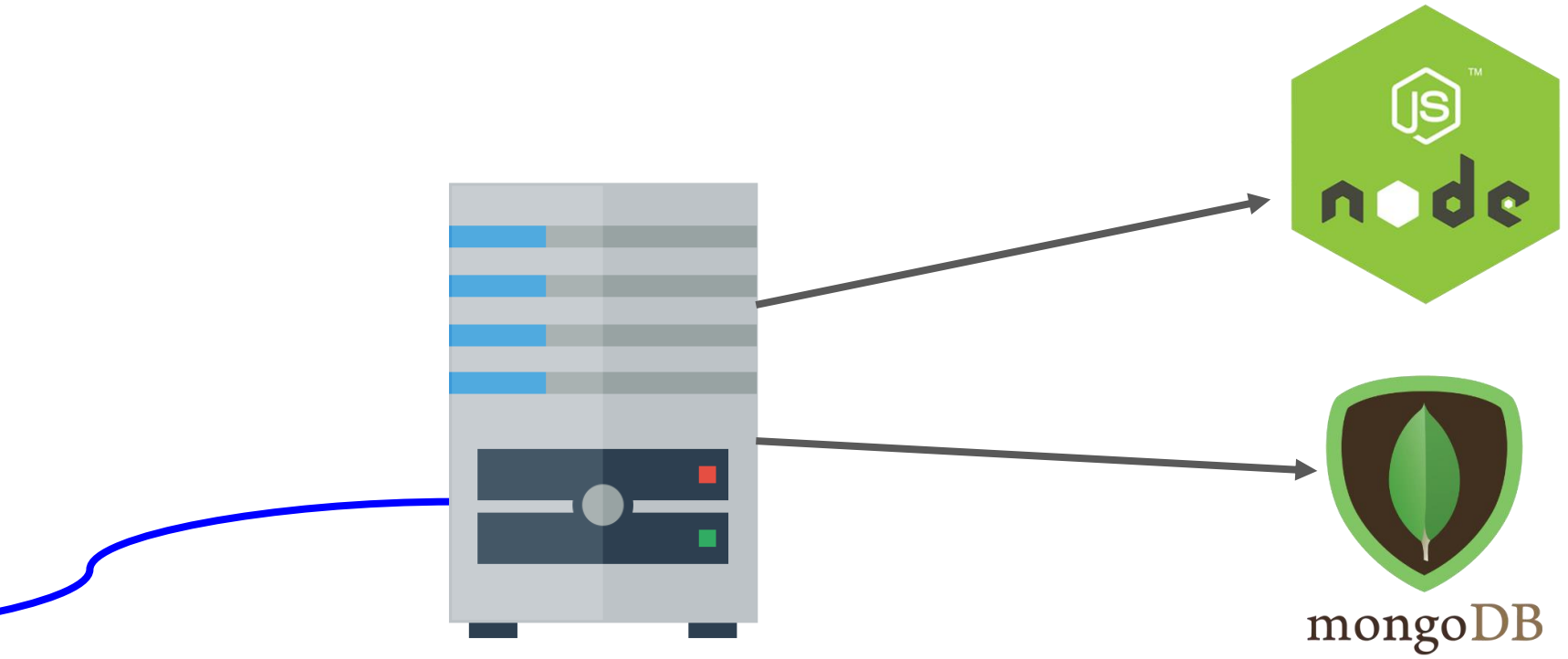
Relational database:

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

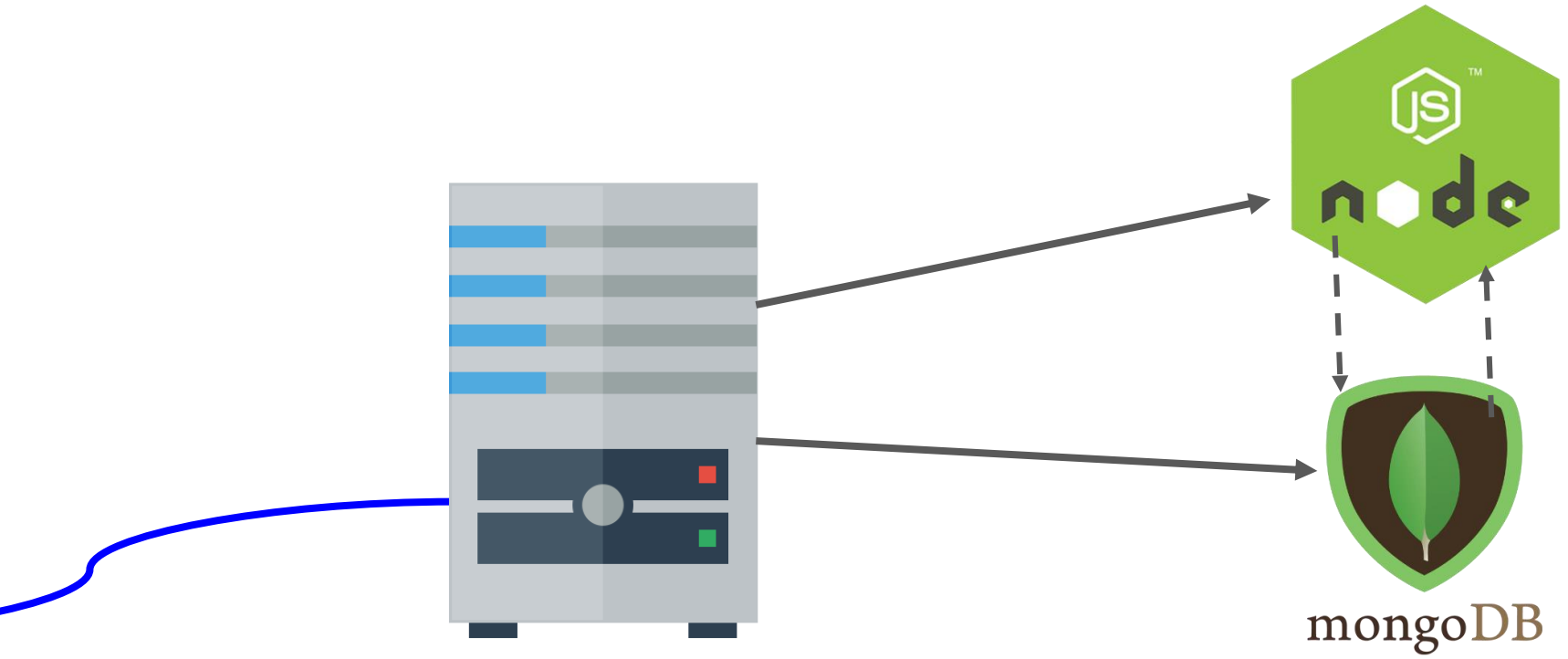
Relational databases have fixed schemas; document-oriented databases have flexible schemas

Document-oriented DB:

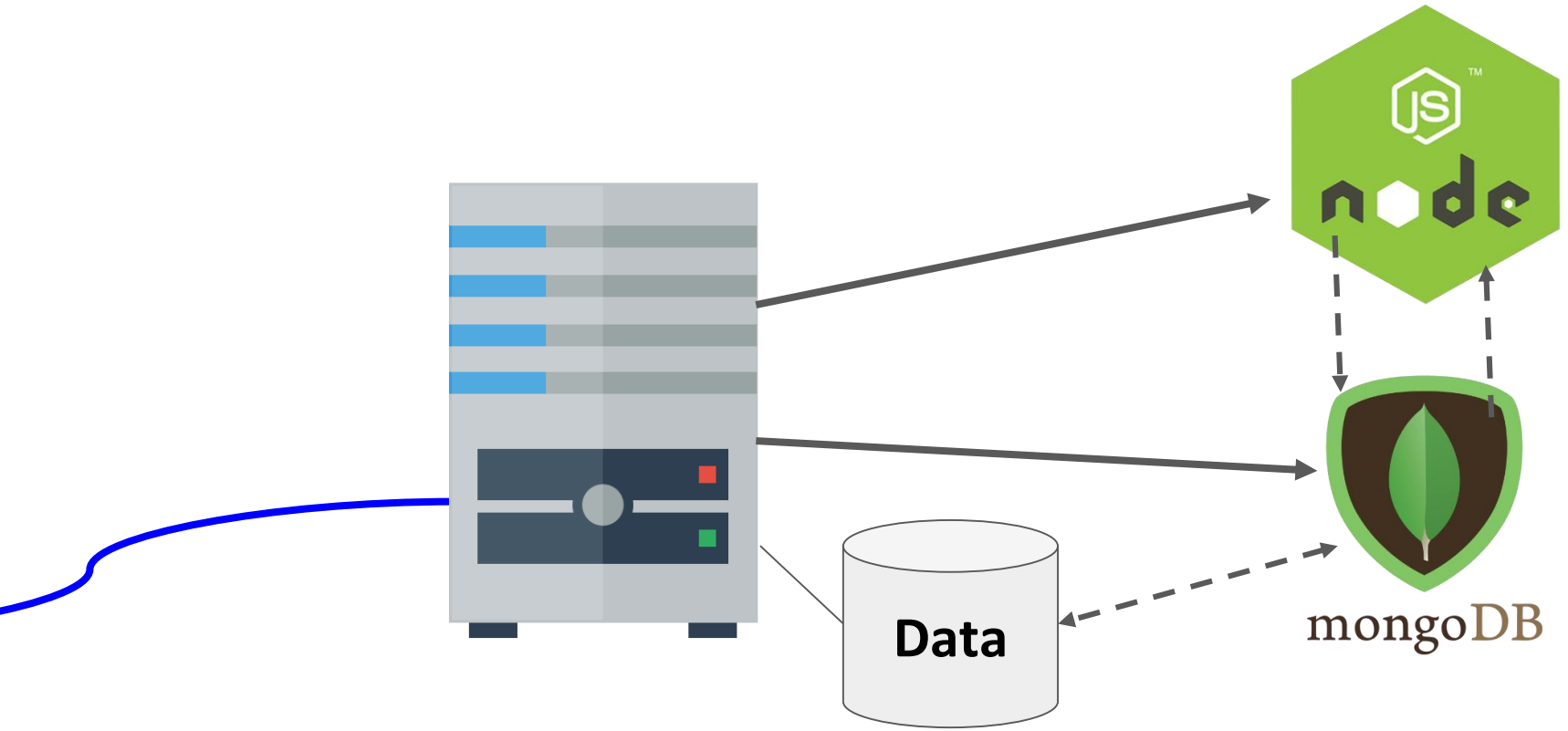
```
{
  name: "Lori",
  employer: "Self",
  occupation: "Entrepreneur"
}
{
  name: "Malia",
  school: "Harvard"
}
```



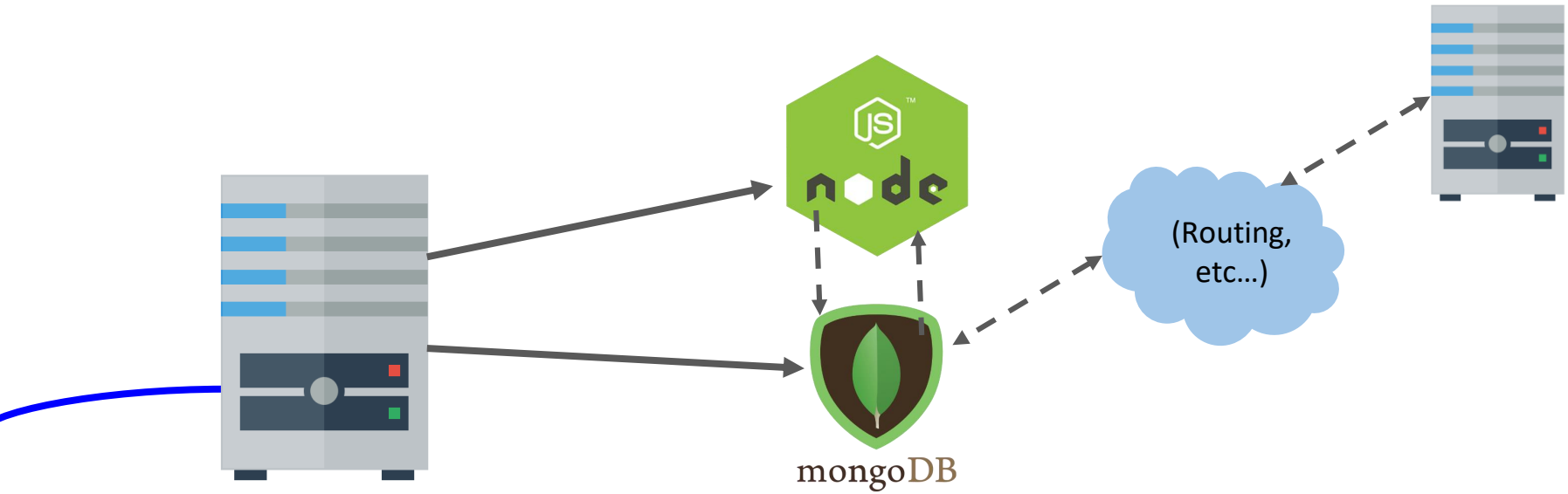
MongoDB is another **software program** running on the computer, alongside our NodeJS server program.
It is also known as the **MongoDB server**.



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.

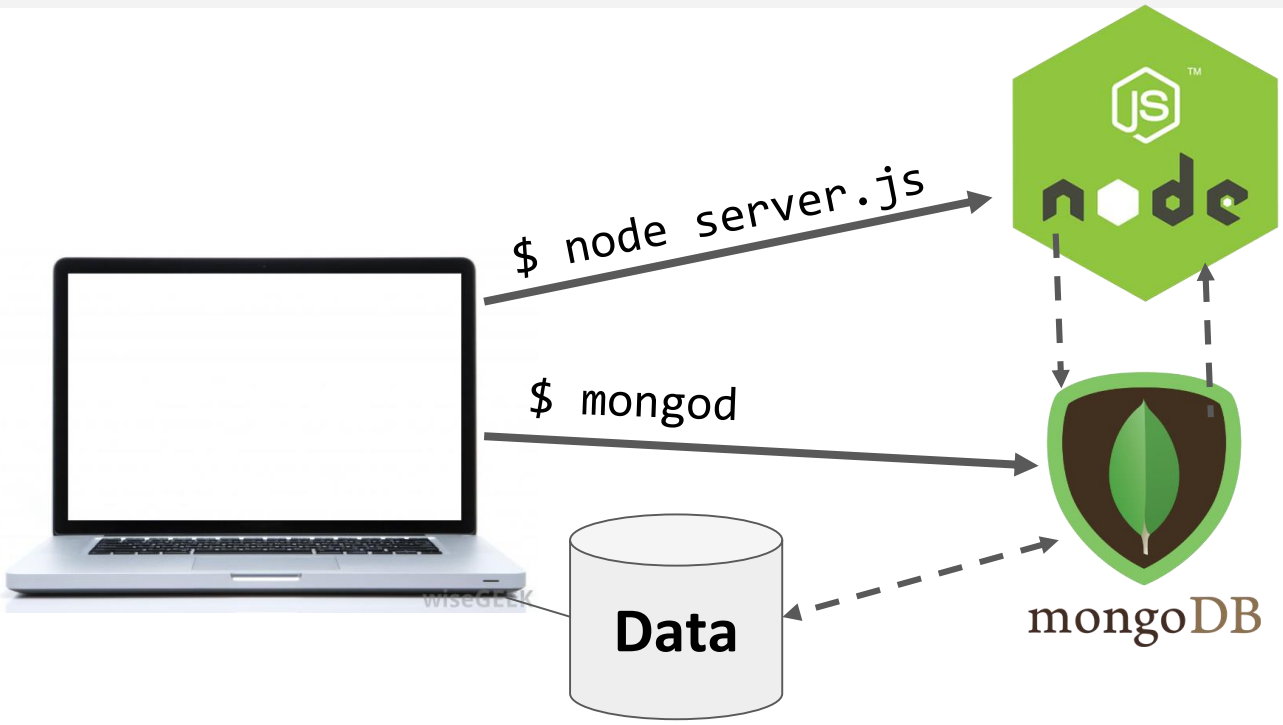


The database the MongoDB Server manages might be local to the server computer...



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

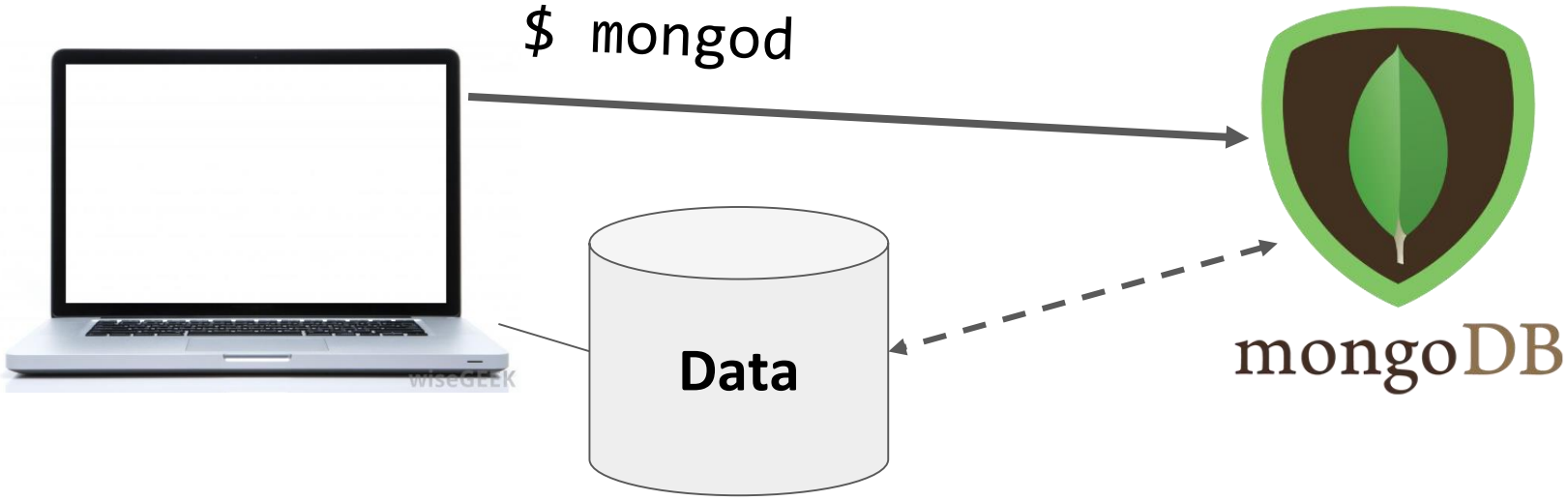
System overview



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**

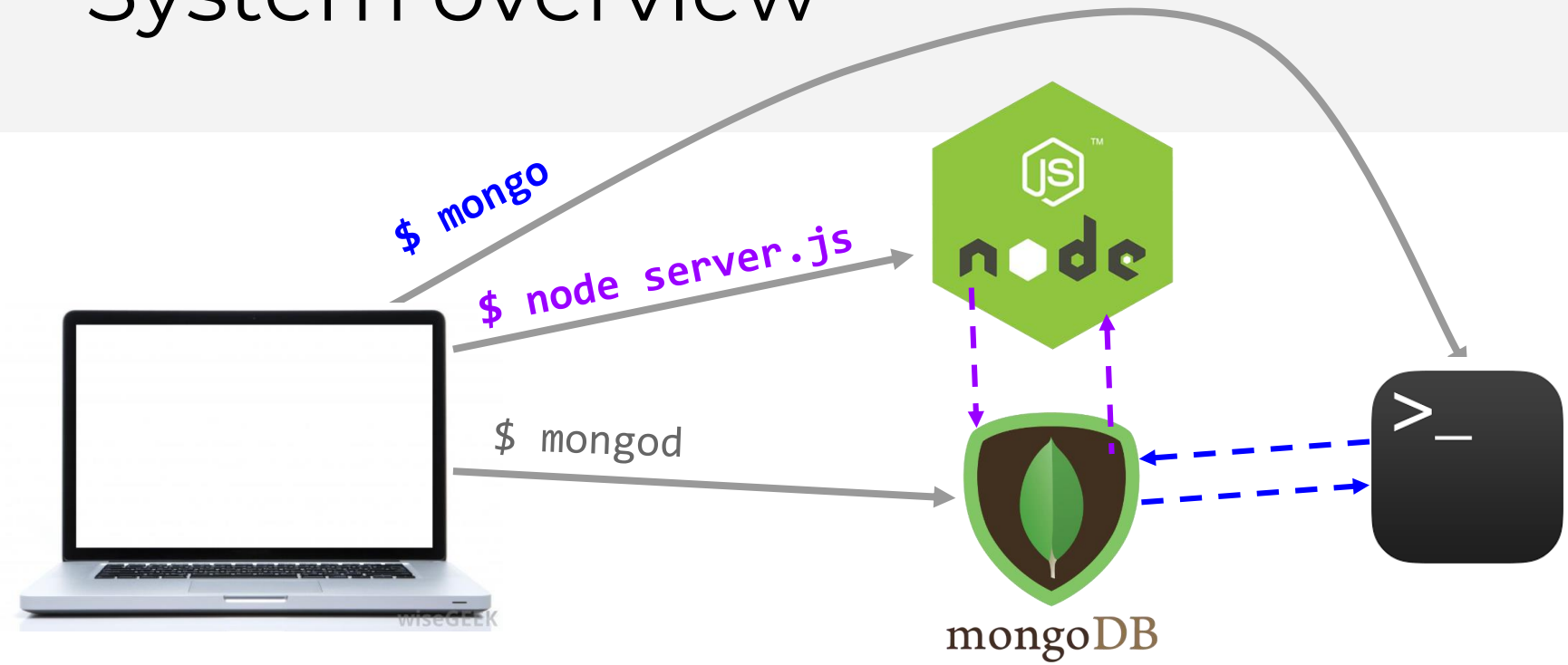
System overview



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**

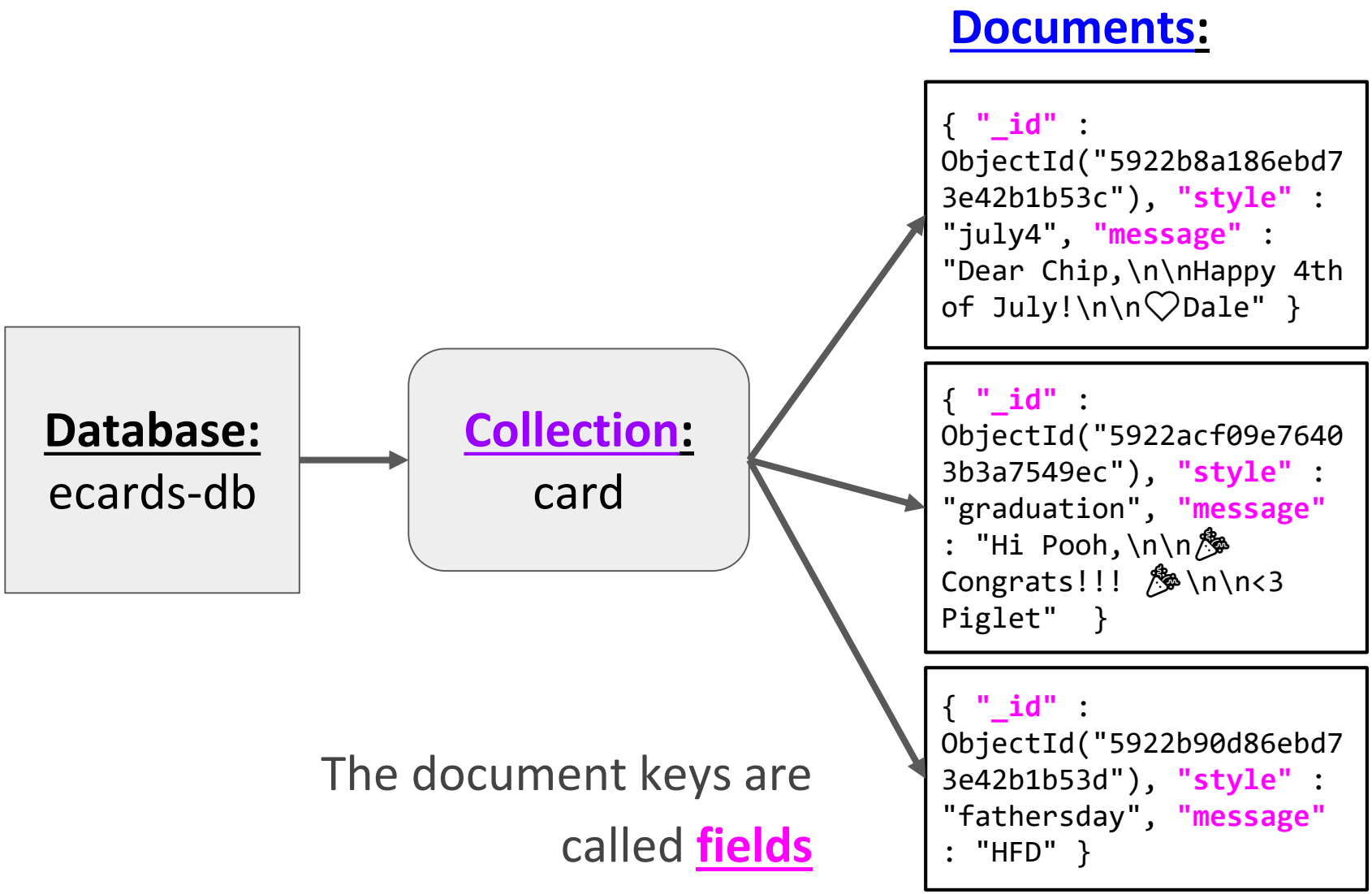
Collection:

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

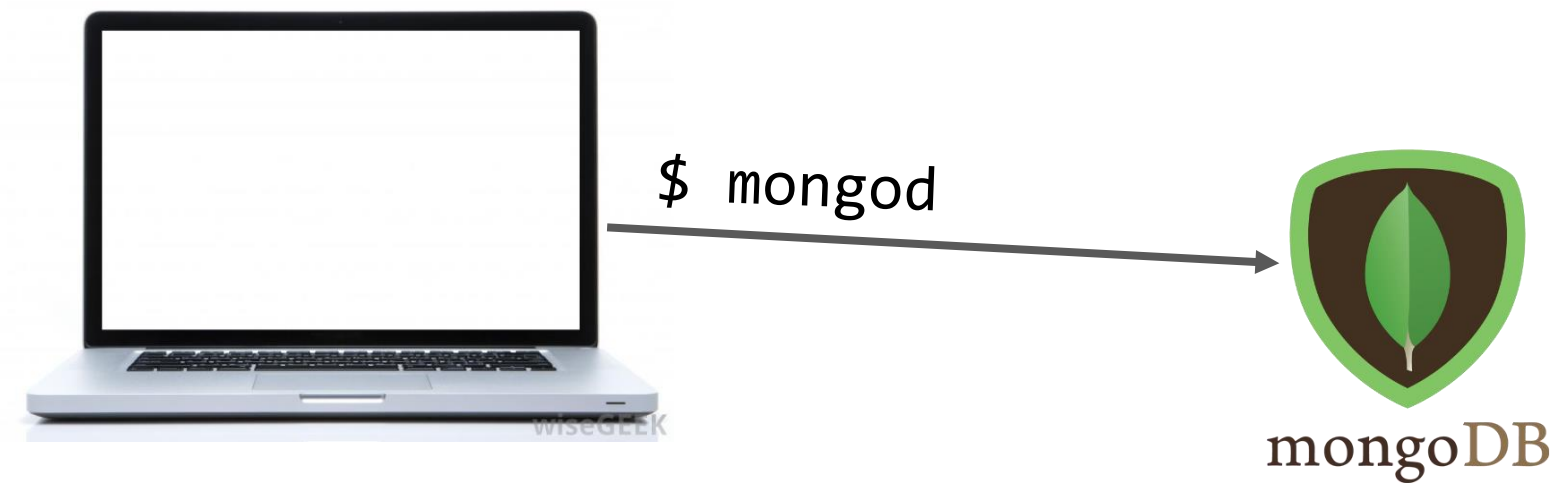
Document:

- 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



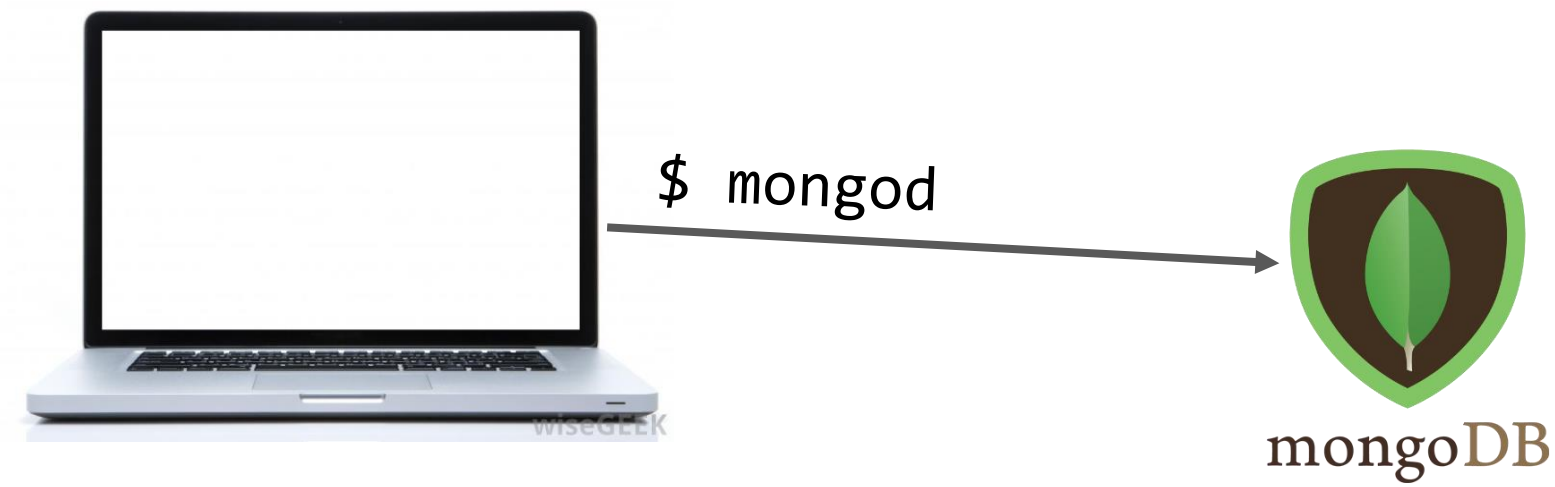
mongod: Database process



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!