Schedule

Today:

- `async/await`: A JavaScript language feature
  - **Not** Node-specific!
- Databases
  - MongoDB
  - NodeJS & MongoDB

Note

- **Assignment 2 released in Wed (Oct 21)**
  - **Due Nov 11**

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

```javascript
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)

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 Advanced

In WPR, we will cover only the very basics:

- How one particular DBMS works (MongoDB)
- How to use MongoDB with NodeJS

MongoDB

MongoDB

MongoDB: A popular open-source DBMS

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

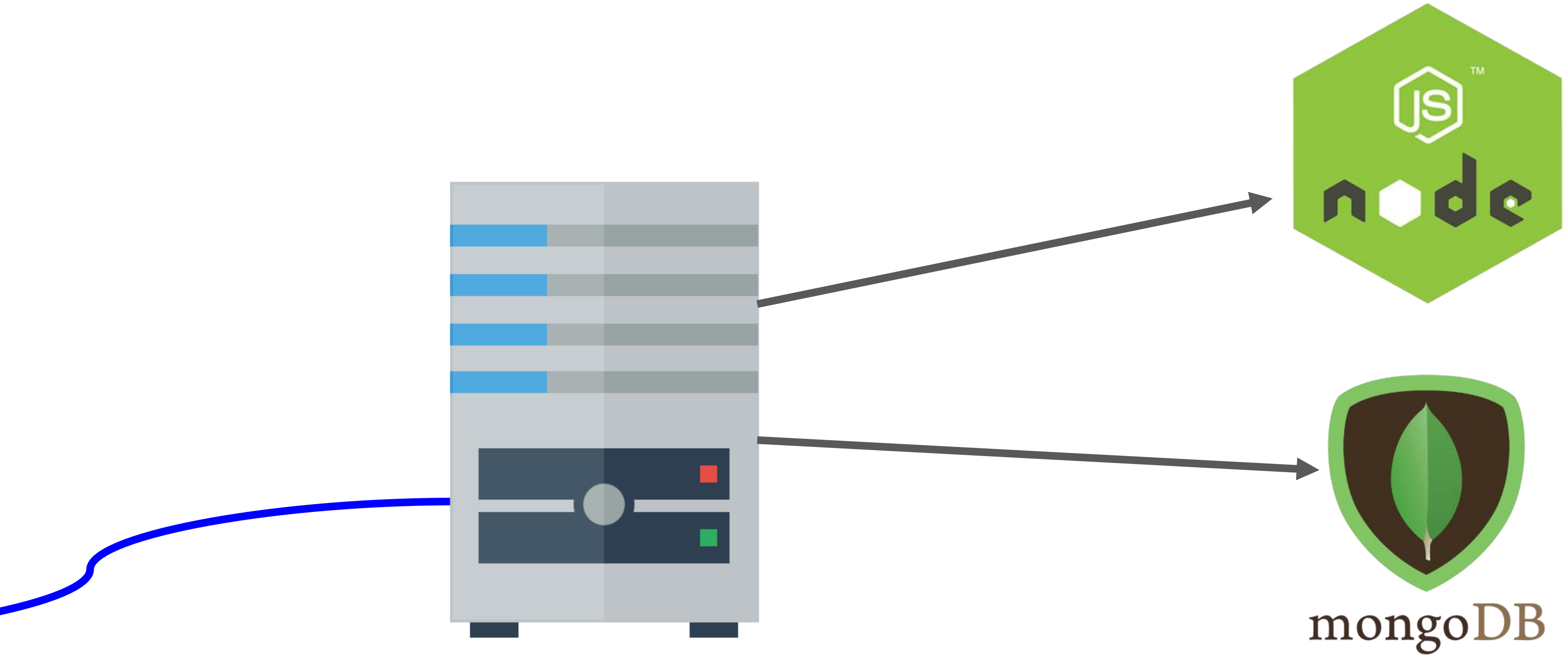
Relational database:

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

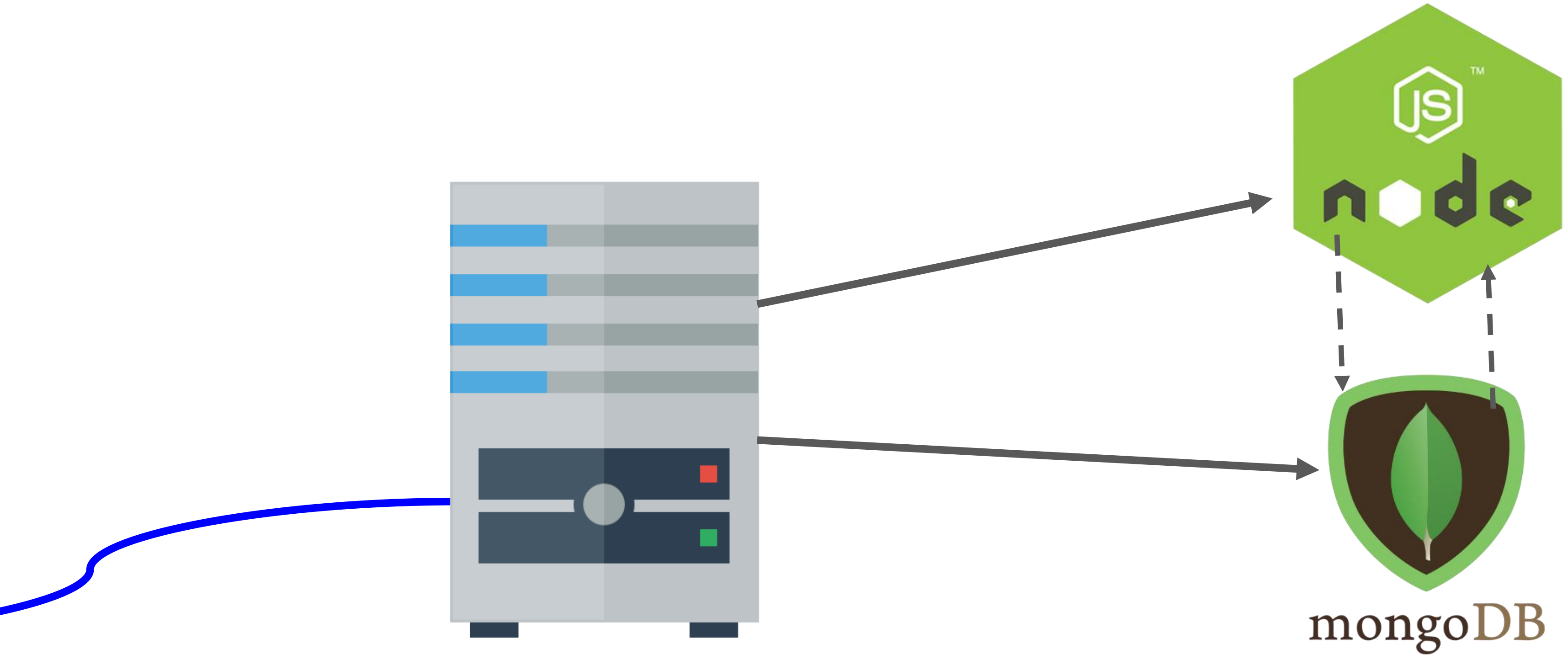
Relational databases have fixed schemas; document-oriented databases have flexible schemas

Document-oriented DB:

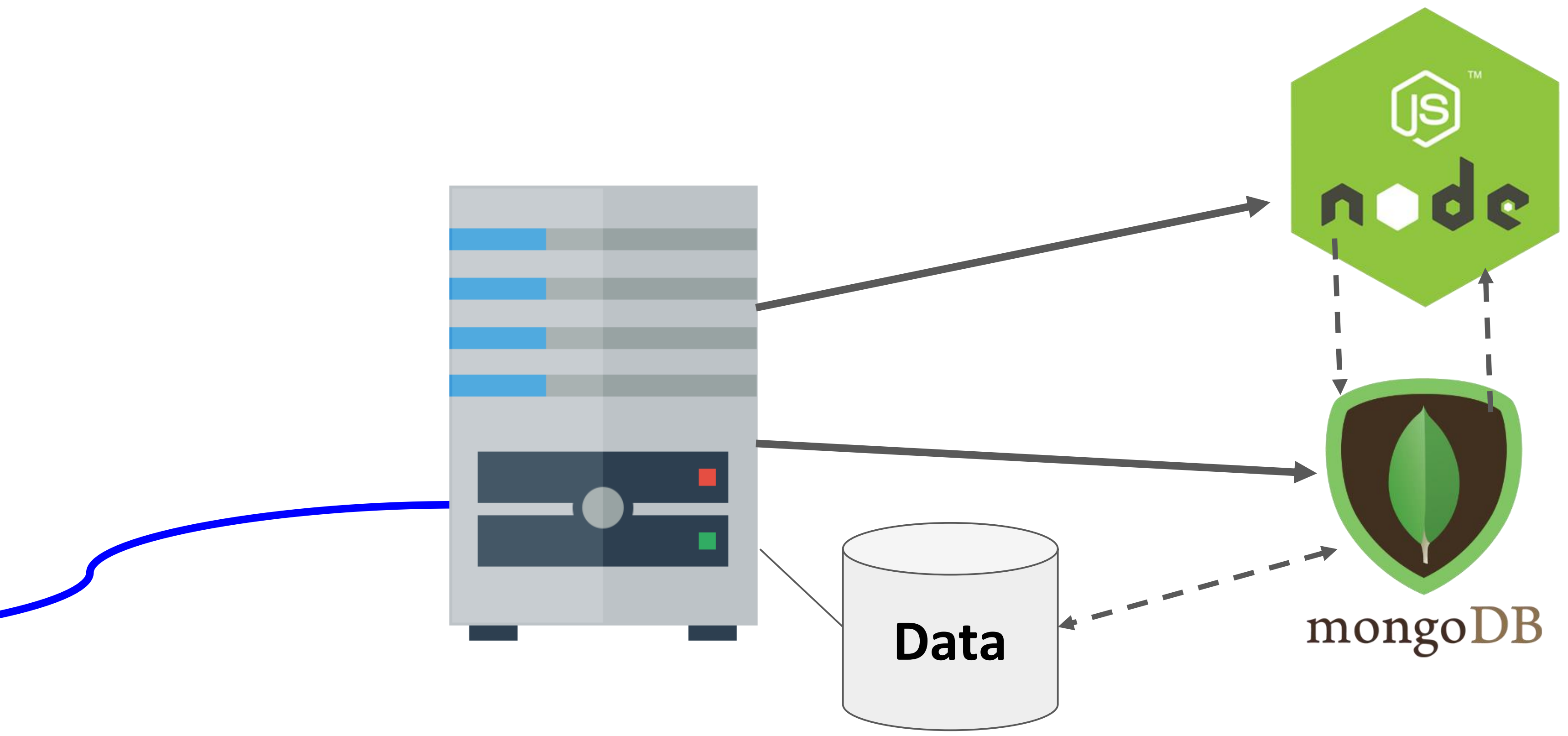
```
{
  name: "Lori",
  employer: "Self",
  occupation: "Entrepreneur"
}
{
  name: "Malia",
  school: "Harvard"
}
```



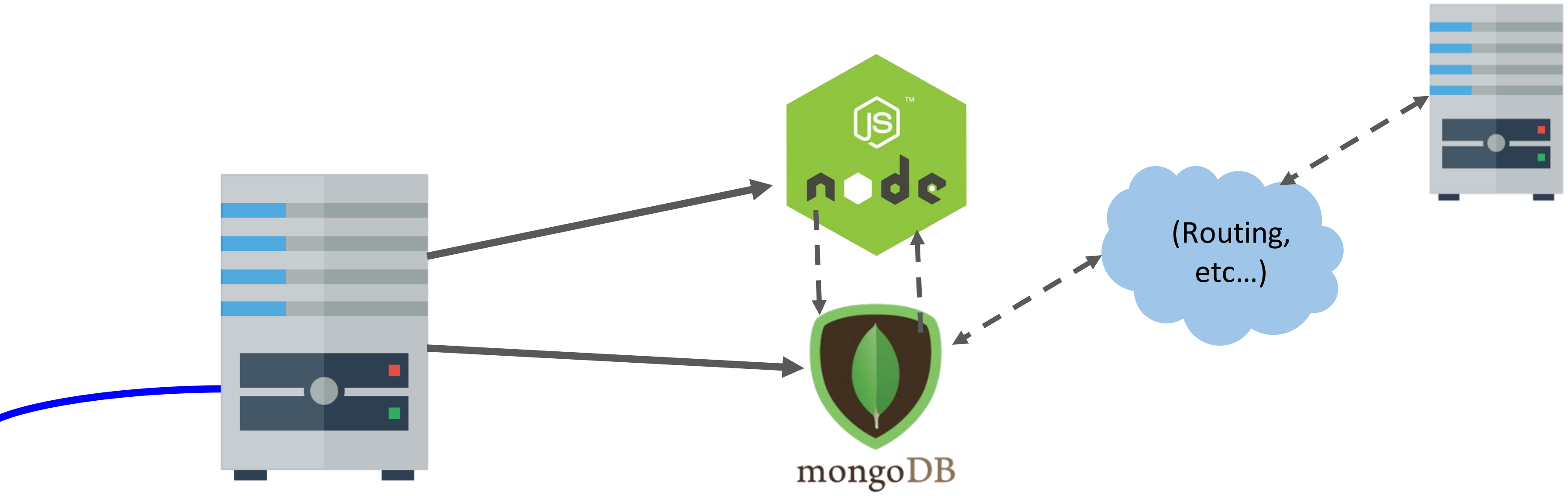
MongoDB is another **software program** running on the computer, alongside our NodeJS server program.
It is also known as the **MongoDB server**.



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.

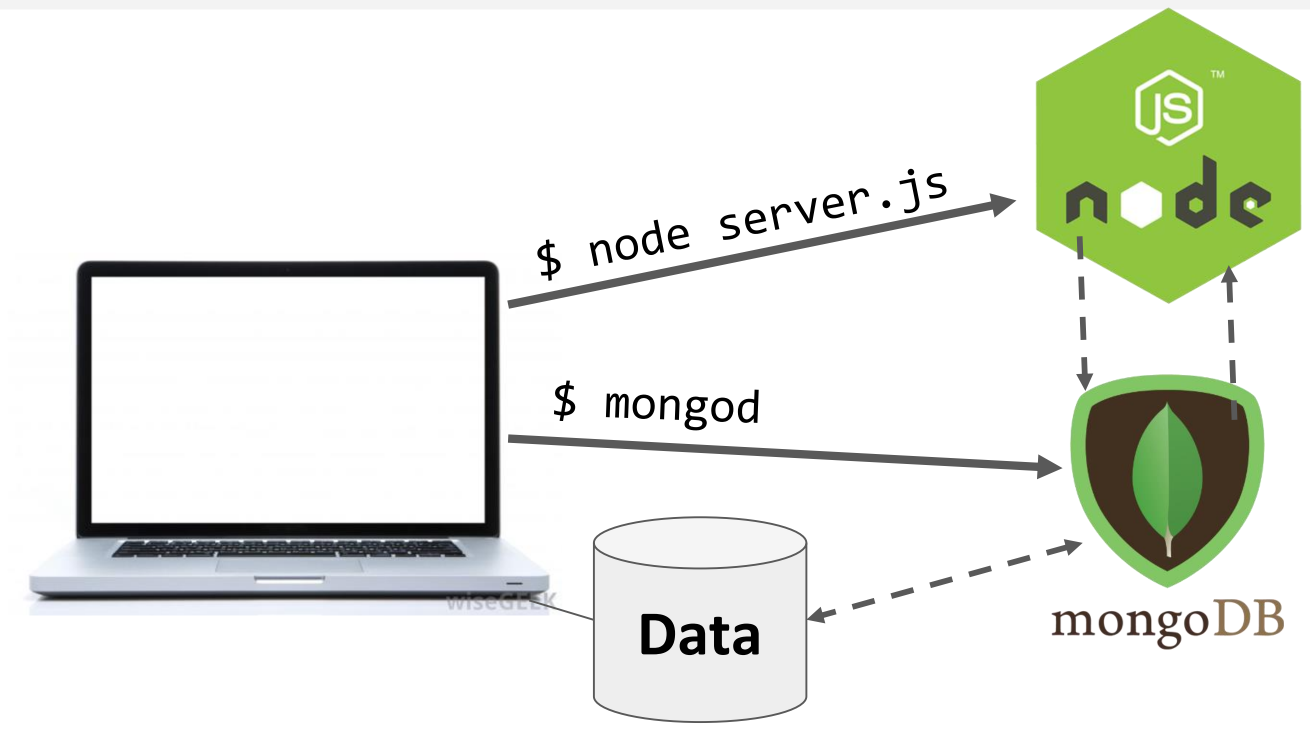


The database the MongoDB Server manages might be local to the server computer...



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

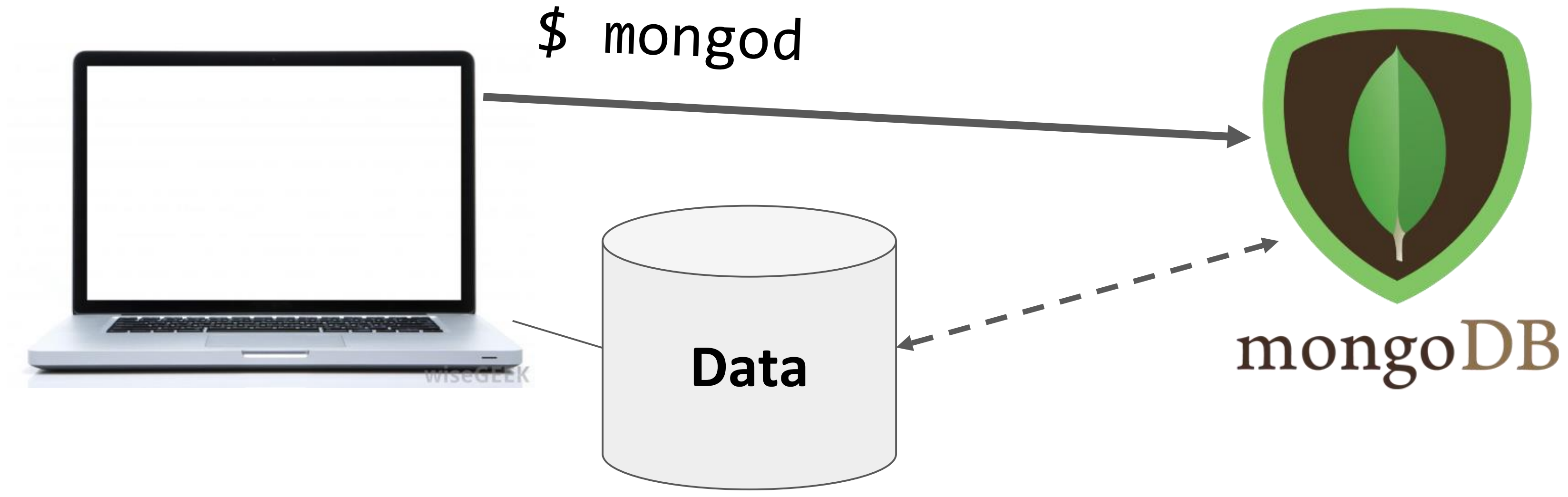
System overview



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**

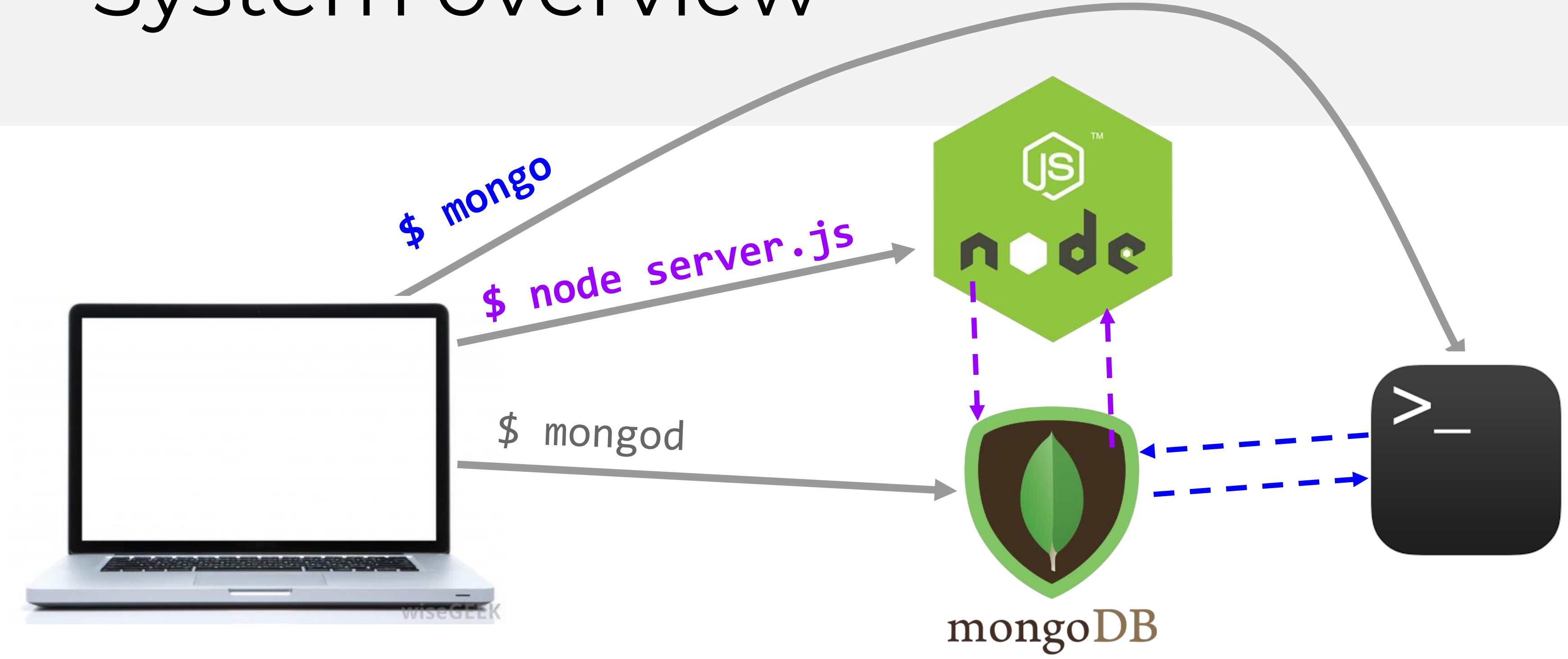
System overview



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**

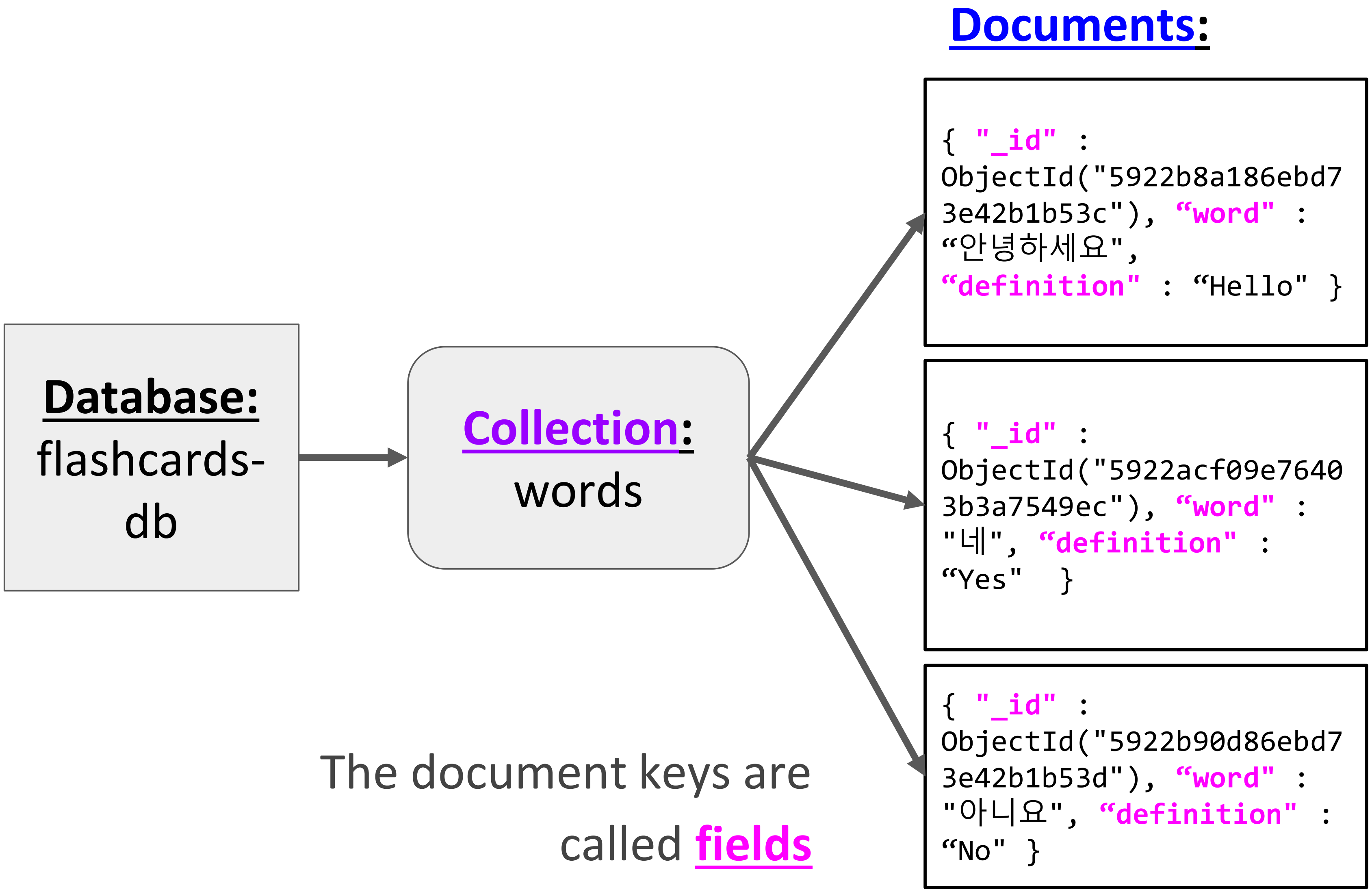
Collection:

- 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



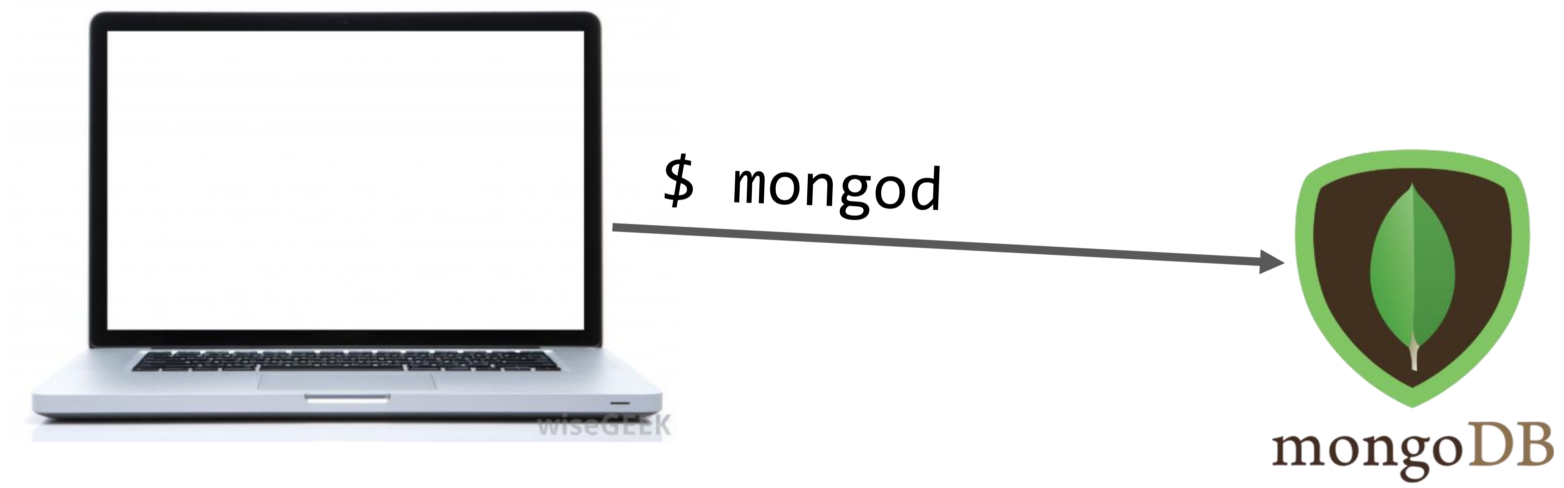
```
{ "_id" : ObjectId("5922b8a186ebd73e42b1b53c"), “word" : “안녕하세요", “definition" : “Hello" }
```

```
{ "_id" : ObjectId("5922acf09e76403b3a7549ec"), “word" : "네", “definition" : “Yes"   }
```

```
{ "_id" : ObjectId("5922b90d86ebd73e42b1b53d"), “word" : "아니요", “definition" : “No" }
```

The document keys are called **fields**

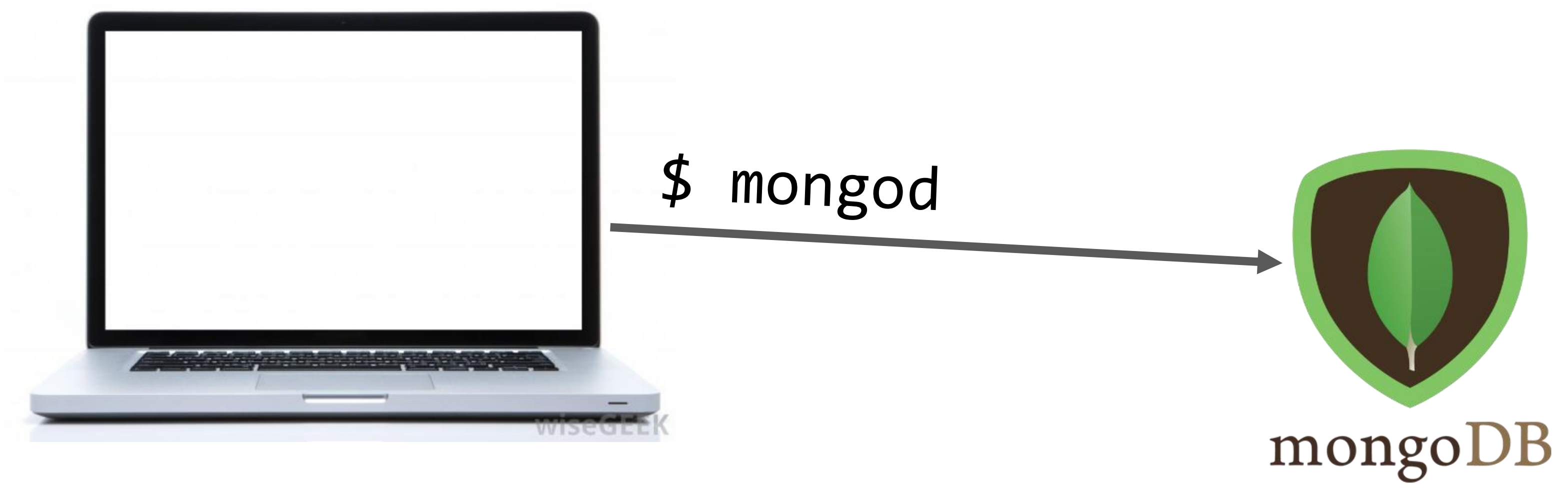
mongod: Database process



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
- Debugging

<br />

NodeJS and MongoDB

NodeJS

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

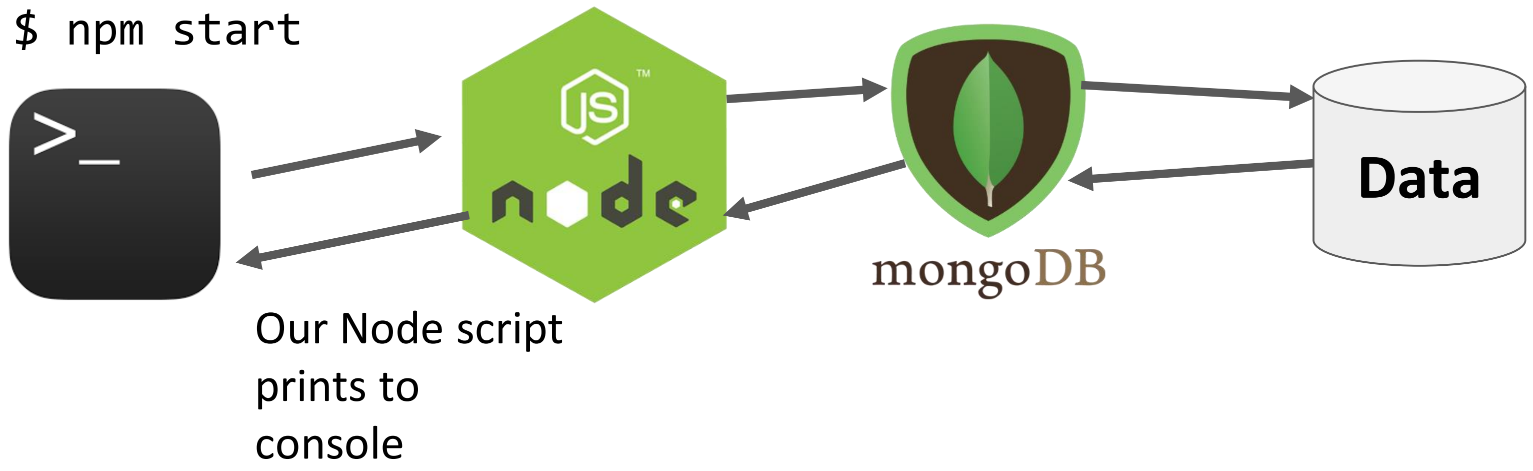
`simple-script.js`

```
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();
```

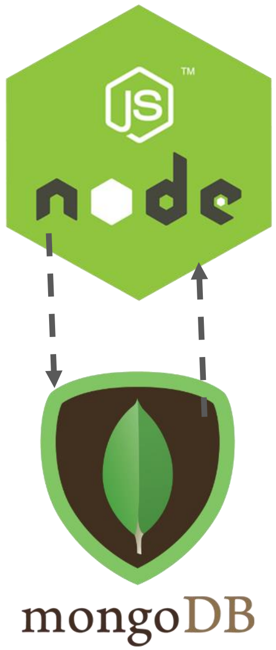
Mongo JS scripts

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, client ) {
  db = client.db();
});
```

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, client ) {
  db = client.db();
});
```

Callbacks and Promises

Every asynchronous MongoDB method has two versions:

- Callback
- Promise

The Promise version is:

```
let db = null;
function onConnected(err, client ) {
  db = client.db();
}
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).db();
}
main();
```

Using a collection

```
async function main() {
  db = await MongoClient.connect(MONGO_URL) .db();
  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:

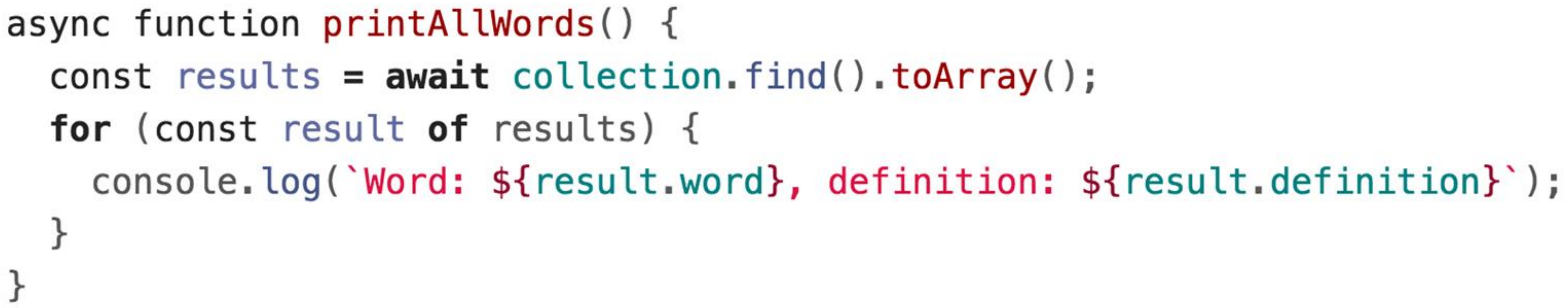
```
async function printAllWordsCursor() {
  const cursor = await collection.find();
  while (await cursor.hasNext()) {
    const result = await cursor.next();
    console.log(`Word: ${result.word}, definition: ${result.definition}`);
  }
}
```

(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 }})
```

Next week:

Server-side rendering - handlebars