CS193X schedule

We've made it to Week 8! I updated the Course Syllabus. Here's our new tentative schedule:

This week:
- Saving data; MongoDB
- Authentication
- Final project assigned Friday

Next week:
- NO CLASS MONDAY
- HW5 due Tuesday at 11:59pm
- Server-side polish
- Possibly a topic voted upon by the class
CS193X schedule

Next next week: Week 10
- Mon June 5: Last lecture!
  - A look back on all that we've learned in CS193x
  - An opinionated tour of frameworks, libraries, etc
- Wed June 7: No lecture

Finals week:
- June 12: Final project due @ 11:59pm
Today's schedule

Today:
- Saving data
  - POST body
  - body-parser
- Databases
  - MongoDB
  - System overview
  - mongo
  - mongod
  - mongodbc
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 screenshot]

**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).
// 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);
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:

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).
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

```javascript
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);
```
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

English dictionary

Look up a word:  

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:

```javascript
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: Handling the message body in NodeJS/Express is a little messy (GitHub):

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
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`:

```javascript
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);
});
```
POST message body

We can access the message body through req.body:

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

```javascript
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: Name: Victoria, email: vrk@stanford.edu
Example: Dictionary

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

English dictionary

Look up a word: dog

The definition of **dog** is:

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

Modify the definition for this word:

Word: dog

Definition:

A quadruped of the genus Canis, esp. the domestic dog (C.familiaris).
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.writeJson(fileName, object)`
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);
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`:

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

GET: Hello, Victoria
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%20wee
knd&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`
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
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:
- CS145: Introduction to Databases
- CS245: Database System Principles
- CS346: Database System Implementation

In CS193X, 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**: A popular open-source DBMS
- A *document-oriented* database as opposed to a *relational* database

### Relational database:

<table>
<thead>
<tr>
<th>Name</th>
<th>School</th>
<th>Employer</th>
<th>Occupation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lori</td>
<td>null</td>
<td>Self</td>
<td>Entrepreneur</td>
</tr>
<tr>
<td>Malia</td>
<td>Harvard</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

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

### Document-oriented DB:

```json
{
    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.
We will be using two ways of communicating to the MongoDB server:

- NodeJS libraries
- `mongo` command-line tool
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

Database: ecards-db

Collection: card

Documents:

1. 
   { 
   "_id": ObjectId("5922acf09e76403b3a7549ec"),  
   "style": "graduation",  
   "message": "Hi Pooh,

   🎓

   Congrats!!!

   <3

   Piglet"  }

2. 
   { 
   "_id": ObjectId("5922b8a186ebd73e42b1b53c"),  
   "style": "july4",  
   "message": "Dear Chip,

   Happy 4th

   of July!

   ❤ Dale"  }

3. 
   { 
   "_id": ObjectId("5922b90d86ebd73e42b1b53d"),  
   "style": "fathersday",  
   "message": "HFD"  }

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
More next time!