CS193X: Web Programming Fundamentals
Spring 2017
Victoria Kirst (vrk@stanford.edu)
Schedule

Today:
- *async/await*: A JavaScript language feature
  - *Not* Node-specific!
- Sending data to the server
- Returning JSON
- package.json
- **HW5 released**
  - **Due May 30**, but please try to complete the setup steps by May 27
PSA: Reinstall Node!

Whoops, you should **install Node v7** instead of Node v6.

Please revisit the installation instructions:
- [http://web.stanford.edu/class/cs193x/install-node/](http://web.stanford.edu/class/cs193x/install-node/)

If you followed them earlier, please reinstall Node, this time selecting "Current" rather than "LTS."
The lecture code has been uploaded to this GitHub:
- [https://github.com/yayinternet/lecture21](https://github.com/yayinternet/lecture21)
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()
Synchronous `fetch()`?

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

```javascript
// 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-\emph{looking} code
- That actually ran asynchronously

// But this code does work:
```javascript
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-\textit{looking} code
- That actually ran asynchronously

\texttt{// But this code does work:}
\begin{verbatim}
async function loadJson() {
    const response = await fetch('albums.json');
    const json = await response.json();
    console.log(json);
}
loadJson();
\end{verbatim}
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
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);

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

The methods in purple return Promises.
function onJsonReady(json) {
    console.log(json);
}

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

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

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

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

```javascript
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 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 function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
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 an event handler for it, JavaScript will continue executing those events.
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

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

```javascript
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

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

```javascript
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.
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
Why async now?!
Because you'll use it on HW5!
Recall: ExpressJS routes

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

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

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

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

In HW5, the starter code defines an async function parameter:

```javascript
async function onGet(req, res) {
    // ...
}

app.get('/', onGet);
```

Which works about the same as a non-async function, except when you write an await inside of it.
You will need to use the provided *gsa-sheets* library, whose functions all return Promises:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRows()</td>
<td>Returns a Promise that resolves to the non-empty rows of the spreadsheet.</td>
</tr>
<tr>
<td>appendRow(row)</td>
<td>Adds the given row to the end of the spreadsheet. Returns a Promise that resolves when complete.</td>
</tr>
<tr>
<td>deleteRow(index)</td>
<td>Deletes the given row in the spreadsheet. Returns a Promise that resolves when complete.</td>
</tr>
</tbody>
</table>

(see more details [in the HW5 spec](#))
ExpressJS routes

You should use `await` expression with these method calls:

```javascript
async function onGet(req, res) {
    const result = await sheet.getRows();
    const rows = result.rows;
    console.log(rows);
}

// TODO(you): Finish onGet.
```

This will essentially let you work with the `gsa-sheets` methods as if they returned values instead of Promises.
async / await availability

**Browsers:**
- All major browsers support async/await, but it's pretty recent: Edge + Safari support came ~1 month ago

**NodeJS:**
- async/await available in v7.5+... which is why we need you to install v7 instead of v6

(FYI, underneath the covers async/await is implemented by generator functions, another functional programming construct)
One more random thing: Template Literals
Template literals allow you to embed expressions in JavaScript strings:

```javascript
const port = process.env.PORT || 3000;
app.listen(port, function () {
    console.log(`Server listening on port ${port}!`);
});
```
Sending data to the server
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):

```javascript
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);
});
Route parameters

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

```javascript
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);
});
```

GET: Flights from SFO to JFK
Example: Dictionary

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

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

```javascript
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);
```
Example: Dictionary fetch

```javascript
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;
}
```
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:

```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](https://github.com/expressjs/body-parser) 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);
    });
});
```
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
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);
});
```

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);
```
Example: Dictionary

We will modify the dictionary example to POST the contents of the form.
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);
});
```

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)
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.
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 kend&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"
}
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