CS193X: Web Programming Fundamentals

Spring 2017

Victoria Kirst (vrk@stanford.edu)
Schedule

Today:
- Servers, generally
- NodeJS
- npm
- Express
- `fetch()` to localhost

If we have time:
- Single-threaded asynchrony
  - JS Event loop
Lecture code

All lecture code is in this git repository: https://github.com/yayinternet/lecture19

You will need to run the commands we show in lecture to run the server code!

Node installation instructions: http://web.stanford.edu/class/cs193x/install-node/
Servers
Server-side programming

The type of web programming we have been doing so far in 193x is called "client-side" programming:
- The code we write gets run in a browser on the user's (client's) machine

Today we will begin to learn about server-side programming:
- The code we write gets run on a server.
- Servers are computers run programs to generate web pages and other web resources.
Let's take another look at how clients and servers work...
CLIENT: You type a URL in the address bar and hit "enter"
Browser sends an HTTP GET request saying "Please GET me the index.html file at http://cs193x.stanford.edu"

Let's take a deeper look at this process...
Browser C++ code creates an array of bytes that is formatted in using **HTTP request message format**.

Browser asks operating system, "Hey, can you send this HTTP Get request message to http://cs193x.stanford.edu"?
Operating system sends a **DNS** query to look up the **IP address** of "http://cs193x.stanford.edu"

DNS server replies with the **IP address**, e.g. 171.67.215.200

- **DNS**: Domain Name System: Translate domain names to **IP address** of the computer associated with that address.
- **IP address**: Numerical unique identifier for every computer connected to the internet.
Operating system opens a **TCP** connection with the computer at 171.67.215.200

After the **TCP** connection is established, the OS can send the HTTP message to 171.67.215.200 *through* the **TCP** connection.

- **TCP**: Transmission Control Protocol, defines the data format for sending information over the wire. (Can be used for HTTP, FTP, etc)
SERVER: There is a computer that is connected to the internet at IP address 171.67.215.200.
On this computer is a web server program:

- The web server program is listening for incoming messages that are sent to it.
- The web server program can respond to messages that are sent to it.

**Node**: The platform we will use to create a web server program that will receive and respond to HTTP requests.

- Also known as "NodeJS"; these terms are synonyms
Aside: "Server"

The definition of server is overloaded:

- Sometimes "server" means the machine/computer that runs the server software.
- Sometimes "server" means the software running on the machine/computer.

You have to use context to know which is being meant.
Aside: Sockets

Q: What does it mean for a program to be "listening" for messages?

When the server first runs, it executes code to create a socket that allows it to receive incoming messages from the OS.

A socket is one end of a communication channel. You can send and receive data on sockets.

However, NodeJS will abstract this away so we don't have to think about sockets.
A TCP connection is established between the client and the server, so now the client and server can send messages directly to teach each other.
Now the operating system is receiving TCP packets from the wire, and the operating system begins sending the contents of the request to the server program.

OS to server program: "Hey, here's a message that was sent to me."
The server software parses the HTTP request and then decides what message it wants to send in response. It formats this message in HTTP, then asks the OS to send this response message over TCP back to the sender.
This HTTP response is then sent back to the client's OS, which notifies the browser of the HTTP response, and then the browser displays the web page.
Summary

When you navigate to a URL:
- Browser creates an HTTP GET request
- Operating system sends the GET request to the server over TCP

When a server computer receives a message:
- The server's operating system sends the message to the server software (via a socket)
- The server software then parses the message
- The server software creates an HTTP response
- The server OS sends the HTTP response to the client over TCP
Learn more

For more on basic server design, sockets and TCP/IP:
- CS110: Principles of Computer Systems

For more on computer networks:
- CS144: Introduction to Computer Networking (Prereq: CS110)
NodeJS
NodeJS:
- A JavaScript runtime written in C++.
- Can interpret and execute JavaScript.
- Includes support for the NodeJS API.

NodeJS API:
- A set of JavaScript libraries that are useful for creating server programs.

V8 (from Chrome):
- The JavaScript interpreter ("engine") that NodeJS uses to interpret, compile, and execute JavaScript code
NodeJS

NodeJS:
- A JavaScript runtime written in C++.
- Can interpret and execute JavaScript.
- Includes support for the NodeJS API.

NodeJS API:
- A set of JavaScript libraries that are useful for creating server programs.

V8 (from Chrome):
- The JavaScript interpreter ("engine") that NodeJS uses to interpret, compile, and execute JavaScript code.
First: Chrome

Chrome:
- A browser written in C++.
- Can interpret and execute JavaScript code.
- Includes support for the DOM APIs.

DOM APIs:
- JavaScript libraries to interact with a web page

V8:
- The JavaScript interpreter ("engine") that Chrome uses to interpret, compile, and execute JavaScript code
Chrome, V8, DOM

- Parser
- Execution Engine
- Garbage Collector

JavaScript runtime
(Call stack, memory, etc.)

DOM API Implementation
const name = 'V8';
console.log('V8');

"Please execute console.log()"
NodeJS, V8, NodeJS APIs

- V8
- JavaScript runtime
  - (Call stack, memory, etc.)
- NodeJS API Implementation
- Parser
- Execution Engine
- Garbage Collector
const x = 15;
    x++;
http.createServer();
What if you tried to call `document.querySelector('div');` in the NodeJS runtime?
document.querySelector('div');

ReferenceError: document is not defined
What if you tried to call `console.log('nodejs');` in the NodeJS runtime?
console.log('nodejs');

(NodeJS API implemented their own console.log)
NodeJS:
- A JavaScript runtime written in C++.
- Can interpret and execute JavaScript.
- Includes support for the NodeJS API.

NodeJS API:
- A set of JavaScript libraries that are useful for creating server programs.

V8 (from Chrome):
- The JavaScript interpreter ("engine") that NodeJS uses to interpret, compile, and execute JavaScript code.
Installation

**NOTE:** The following slides assume you have already installed NodeJS.

NodeJS installation instructions:

- [http://web.stanford.edu/class/cs193x/install-node/](http://web.stanford.edu/class/cs193x/install-node/)
Running node without a filename runs a REPL loop
  - Similar to the JavaScript console in Chrome, or when you run "python"

```sh
$ node
> let x = 5;
undefined
> x++
5
> x
6
```
NodeJS can be used for writing scripts in JavaScript, completely unrelated to servers.

```
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();
```
node command

The node command can be used to execute a JS file:

$ node fileName

$ node simple-script.js
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
Node for servers

Here is a very basic server written for NodeJS:

```javascript
const http = require('http');

const server = http.createServer();

server.on('request', function(req, res) {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World\n');
});

server.on('listening', function() {
    console.log('Server running!!');
});

server.listen(3000);
```

**WARNING:** We will **not** actually be writing servers like this!!!

We will be using ExpressJS to help, but we haven't gotten there yet.
require()

```javascript
const http = require('http');
const server = http.createServer();
```

The NodeJS `require()` statement loads a module, similar to `import` in Java or `include` in C++.

- We can `require()` modules included with NodeJS, or modules we've written ourselves.
- In this example, `http` is referring to the [HTTP NodeJS module](https://nodejs.org/api/http.html).
require()

const http = require('http');
const server = http.createServer();

The http variable returned by require('http') can be used to make calls to the HTTP API:
- http.createServer() creates a Server object
Emitter.on

```javascript
server.on('request', function(req, res) {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World\n');
});

server.on('listening', function() {
    console.log('Server running!');
});
```

The **on()** function is the NodeJS equivalent of `addEventListener`. 
Emitter.on

```javascript
server.on('request', function(req, res) {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World\n');
});
```

The `request` event is emitted each time there is a new HTTP request for the NodeJS program to process.
Emitter.on

```javascript
server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});
```

The `req` parameter gives information about the incoming request, and the `res` parameter is the response parameter that we write to via method calls.

- `statusCode`: Sets the HTTP status code.
- `setHeader()`: Sets the HTTP headers.
- `end()`: Writes the message to the response body then signals to the server that the message is complete.
listen() and listening

```javascript
server.on('listening', function() {
    console.log('Server running!');
});

server.listen(3000);
```

The `listen()` function will start accepting connections on the given **port number**.
- The `listening` event will be emitted when the server has been bound to a port.

**Q: What's a port? What is binding?**
**Ports and binding**

**port**: In the context of networking, a "logical" (as opposed to a physical) connection place
- A number from 0 to 65535 (16-bit unsigned integer)

When you start running a server process, you tell the operating system what port number to associate with it. This is called **binding**.
Operating system opens a **TCP** connection on port **80** of the computer at 171.67.215.200.

A TCP connection requires an IP address **and** a port number.
- If no port number is specified, 80 is the default for HTTP requests.

The server process running on port **80** is responding to requests.
Ports defaults

There are many well-known ports, i.e. the ports that will be used by default for particular protocols:

21: File Transfer Protocol (FTP)
22: Secure Shell (SSH)
23: Telnet remote login service
25: Simple Mail Transfer Protocol (SMTP)
53: Domain Name System (DNS) service
80: Hypertext Transfer Protocol (HTTP) used in the World Wide Web
110: Post Office Protocol (POP3)
119: Network News Transfer Protocol (NNTP)
123: Network Time Protocol (NTP)
143: Internet Message Access Protocol (IMAP)
161: Simple Network Management Protocol (SNMP)
194: Internet Relay Chat (IRC)
443: HTTP Secure (HTTPS)
Development server

```javascript
server.on('listening', function() {
    console.log('Server running!');
});

server.listen(3000);
```

For our development server, we can choose whatever port number we want. In this example, we've chosen 3000.
Running the server

When we run `node server.js` in the terminal, we see the following:

```
vrk:node-server $ node server.js
Server running!
```

The process does not end after we run the command, as it is now waiting for HTTP requests on port 3000.

Q: How do we send an HTTP request on port 3000?
We can send an HTTP GET request running on one of the ports on the local computer using the URL:

http://localhost:\textit{portNumber}, e.g. http://localhost:3000

\textbf{localhost} is a hostname that means "this computer."
Server response

Here is the result of the request to our HTTP server:

Hello World
Node for servers

const http = require('http');

const server = http.createServer();

server.on('request', function(req, res) {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World
');
});

server.on('listening', function() {
    console.log('Server running!');
});

server.listen(3000);
Node for servers

The NodeJS server APIs are actually pretty low-level:

- You build the request manually
- You write the response manually
- There's a lot of tedious processing code

```javascript
var http = require('http');

http.createServer(function(request, response) {
  var headers = request.headers;
  var method = request.method;
  var url = request.url;
  var body = [];

  request.on('error', function(err) {
    console.error(err);
  }).on('data', function(chunk) {
    body.push(chunk);
  }).on('end', function() {
    body = Buffer.concat(body).toString();
    // BEGINNING OF NEW STUFF

    response.on('error', function(err) {
      console.error(err);
    });

    response.statusCode = 200;
    response.setHeader('Content-Type', 'application/json');
    // Note: the 2 lines above could be replaced with this next one:
    // response.writeHead(200, {'Content-Type': 'application/json'});

    var responseBody = {
      headers: headers,
      method: method,
      url: url,
      body: body
    };

    // END BEGINNING OF NEW STUFF
    response.end(JSON.stringify(responseBody));
}).listen(3000);
```
ExpressJS

We're going to use a library called ExpressJS on top of NodeJS:

```javascript
const express = require('express');
const app = express();

app.get('/', function (req, res) {
    res.send('Hello World!');
})

app.listen(3000, function () {
    console.log('Example app listening on port 3000!');
})
```
Express routing
ExpressJS

However, Express is not part of the NodeJS APIs. If we try to use it like this, we'll get an error:

```javascript
const express = require('express');
const app = express();
```

```
module.js:327
    throw err;
^{

Error: Cannot find module 'express'
    at Function.Module._resolveFilename (internal/module.js:553:15)
    at Function.Module._load (internal/module.js:506:25)
    at Function.executeUserScript (chrome://extensions/content/user_script.js:92:76)
    at Module._extensions..js (internal/module.js:735:10)...
```

We need to install Express via npm.
When you install NodeJS, you also install npm:

- **npm**: Node Package Manager*:
  Command-line tool that lets you install packages (libraries and tools) written in JavaScript and compatible with NodeJS

- Can find packages through the online repository: https://www.npmjs.com/

*though the creators of "npm" say it's not an acronym (as a joke -_-)
npm install and uninstall

**npm install** *package-name*
- This downloads the *package-name* library into a *node_modules* folder.
- Now the *package-name* library can be included in your NodeJS JavaScript files.

**npm uninstall** *package-name*
- This removes the *package-name* library from the *node_modules* folder, deleting the folder if necessary.
Express example

$ npm install express
$ node server.js
Example app listening on port 3000!

Hello World
Express routes

You can specify **routes in Express**:

```javascript
app.get('/', function (req, res) {
    res.send('Main page!');
});

app.get('/hello', function (req, res) {
    res.send('GET hello!');
});

app.post('/hello', function (req, res) {
    res.send('POST hello!');
});
```
Express routes

```javascript
app.get('/hello', function (req, res) {
  res.send('GET hello!');
});

app.method(path, handler)
```

- Specifies how the server should handle HTTP method requests made to URL/path
- This example is saying:
  - When there's a GET request to http://localhost:3000/hello, respond with the text "GET hello!"
app.get('/hello', function (req, res) {
  res.send('GET hello!');
});

Express has its own Request and Response objects:
- req is a Request object
- res is a Response object
- res.send() sends an HTTP response with the given content
  - Sends content type "text/html" by default
Querying our server

Here are three ways to send HTTP requests to our server:

1. Navigate to http://localhost:3000/<path> in our browser
   a. Can only do GET requests

2. Call `fetch()` in web page
   a. We've done GET requests so far, but can send any type of HTTP request

3. `curl` command-line tool
   a. Debug tool we haven't seen yet
curl

curl: Command-line tool to send and receive data from a server ([Manual](#))

curl --request METHOD url

e.g.

$ curl --request PUT http://localhost:3000/hello
`fetch()` to localhost

If we try fetching to localhost from file://

```javascript
fetch('http://localhost:3000')
  .then(onResponse)
  .then(onTextReady);
```

We get this CORS error:

```html
Fetch API cannot load http://localhost:3000/. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
```

```javascript
Uncaught (in promise) TypeError: Failed to fetch
```
Server static data

We can instead serve our HTML/CSS/JS **statically** from the same server:

```javascript
const express = require('express');
const app = express();

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

app.get('/', function (req, res) {
    res.send('Main page!');
});
```
GET query params in Express

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

Query parameters are saved in `req.query`. 
GET query params in Express

Hello, Victoria
fetch() with POST

```javascript
app.post('/hello', function (req, res) {
    res.send('POST hello!');
});
```

On the server-side, you define your handler in `app.post()` to handle POST requests.
fetch() with POST

```javascript
function onTextReady(text) {
    console.log(text);
}

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

fetch('/hello', { method: 'POST' })
    .then(onResponse)
    .then(onTextReady);
```
fetch() with POST

```javascript
function onTextReady(text) {
    console.log(text);
}

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

fetch('hello', { method: 'POST' })
    .then(onResponse)
    .then(onTextReady);
```
Query params with POST

You can send query parameters via POST as well:

```javascript
function onTextReady(text) {
    console.log(text);
}

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

fetch('/hello?name=Victoria', { method: 'POST' })
    .then(onResponse)
    .then(onTextReady);

(WARNING: We will **not** be making POST requests like this!
We will be sending data in the body of the request instead of via query params.)
Query params with POST

These parameters are accessed the same way:

```javascript
app.post('/hello', function (req, res) {
    const queryParams = req.query;
    console.log(queryParams);
    const name = req.query.name;
    res.send('POST Hello, ' + name);
});
```

(WARNING: We will not be making POST requests like this! We will be sending data in the body of the request instead of via query params.)
Overflow (will cover if time)
Single-threaded asynchrony
Recall: Discography page

We wrote a web page that lists the Mariah Carey albums stored in albums.json and lets us sort the albums: (CodePen / demo)
Asynchronous events

We have written our code in a way that assumes `fetch()` will complete before clicking, but on a slow connection, that's not a safe assumption.
General problem

The problem stated generically:

- There are 2+ events that can occur at unpredictable times, and the two events are dependent on each other in some way

(Some people call this a "race condition", though other people reserve the term for multiple threads only.)
Solutions

You can either "force" loading to occur before button click, for example:

- Disable buttons until the JSON loads
- OR: Don't show buttons until the JSON loads
- OR: Don't show the UI at all until the JSON completes
Single-threaded asynchrony
loadAlbums() {
    fetch(JSON_PATH)
      .then(this._onResponse)
      .then(this._onJsonReady);
}

_onJsonReady(json) {
    this.albumInfo = json.albums;
    this._renderAlbums();
}

_onResponse(response) { 
    return response.json();
}

_sortAlbums(sortFunction) {
    this.albumInfo.sort(sortFunction);
    this._renderAlbums();
}
The browser is fetching albums.json...
User clicks a button, so the event handler is running...
Is it possible that while the click handler is still running (still on the call stack), the fetch() callback also fires?

```javascript
_sortAlbums(sortFunction) {
    this.albumInfo.sort(sortFunction);
    this._renderAlbums();
}

_onJsonReady(json) {
    this.albumInfo = json.albums;
    this._renderAlbums();
}
```
The answer is **No**, because JavaScript is **single-threaded**.

```javascript
_sortAlbums(sortFunction) {
    this.albumInfo.sort(sortFunction);
    this._renderAlbums();
}
_onJsonReady(json) {
    this.albumInfo = json.albums;
    this._renderAlbums();
}
```
Single-threaded?

Some hand-wavy definitions:

- **Single-threaded**:
  - When your computer processes one command at a time
  - There is one call stack

- **Multi-threaded**
  - When your computer processes multiple commands simultaneously
  - There is one call stack **per thread**

**thread**: a linear sequence of instructions; an executable container for instructions
Single-threaded JS

- We create a new Album for each album in the JSON file
- For each album, we create a new DOM Image

Q: If in JavaScript, only one thing happens at a time, does that mean only one image loads at a time?

```javascript
(renderAlbums) {
    const albumContainer = document.querySelector('#album-container');
    albumContainer.innerHTML = '';
    for (const info of this.albumInfo) {
        const album = new Album(albumContainer, info.url);
    }
}
```

```javascript
class Album {
    constructor(albumContainer, imageUrl) {
        // Same as document.createElement('img');
        const image = new Image();
        image.src = imageUrl;
        albumContainer.appendChild(image);
    }
}
```
Empirically, that doesn't seem to be the case:

Mariah Carey's albums

By year, descending  By year, ascending  By title, alphabetical
If we look at Chrome's Network tab, we see there are several images being loaded simultaneously:

Q: If JavaScript is single-threaded, i.e. if only one thing happens at a time, how can images be loaded in parallel?
JavaScript event loop
Note: see talk!

(For a perfectly great talk on this, see Philip Roberts' talk: https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=1s)

And for a perfectly great deep dive on this, see Jake Archibald's blog post: https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/

These slides are inspired by these resources!)
setTimeout

To help us understand the event loop better, let's learn about a new command, `setTimeout`:

```javascript
setTimeout(function, delay);
```

- `function` will fire after `delay` milliseconds
- [CodePen example](https://codepen.io)
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
Call stack + setTimeout

```javascript
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

(global function)
Call stack + setTimeout

```javascript
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');

console.log('Point A');
(global function)
```
Call stack + setTimeout

```javascript
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

(global function)
Call stack + setTimeout

```javascript
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');

setTimeout(...);
(global function)
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');

console.log('Point B');
(global function)
Call stack + setTimeout

```javascript
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

(global function)
Call stack + setTimeout

```javascript
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');

console.log('Point C');
onTimerDone();
Call stack + setTimeout

```javascript
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

onTimerDone()
Call stack + setTimeout

```javascript
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');

querySelector('h1');

onTimerDone();
```
Call stack + setTimeout

```javascript
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```
Call stack + setTimeout

```javascript
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
What "enqueue" on `onTimerDone`?
How does it get fired?

```
function onTimerDone() {
    console.log('Point C');
    const h1 = document.querySelector('h1');
    h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

`setTimeout(...);` (global function)
Tasks, Micro-tasks, and the Event Loop
The JavaScript runtime can do only one thing at a time...

- `setTimeout()`
- (global function)
Tasks and the Event Loop

But the JS runtime runs within a browser, which can do multiple things at a time.
Here's a picture of the major pieces involved in executing JavaScript code in the browser.
JS execution

- **Call stack**: JavaScript runtime call stack. Executes the JavaScript commands, functions.

- **Browser internal implementation**: The C++ code that executes in response to native JavaScript commands, e.g. `setTimeout`, `element.classList.add('style')`, etc.
**JS execution**

- **Call stack**: JavaScript runtime call stack. Executes the JavaScript commands, functions.

- **Browser internal implementation**: The C++ code that executes in response to native JavaScript commands, e.g., `setTimeout`, `element.classList.add('style')`, etc.

The browser itself is multi-threaded and multi-process!
- **Task Queue**: When the browser internal implementation notices a callback from something like setTimeout or addEventListener is should be fired, it creates a Task and enqueues it in the Task Queue.
- **Micro-task Queue**: Promises are special tasks that execute with higher priority than normal tasks, so they have their own special queue. (see details here)
**JS execution**

**Event loop:** Processes the task queues.
- When the call stack is empty, the event loop pulls the next task from the task queues and puts it on the call stack.
- The Micro-task queue has higher priority than the Task Queue.
Philip Roberts wrote a nice visualizer for the JS event loop:

- `setTimeout`
- `With click`