Schedule

Today:
- Querying REST APIs
  - Form submission
- Fetch API gotchas
  - CORS
  - Closures
- Single-threaded asynchrony
  - JS Event loop

Monday: NodeJS!
JSON
**JavaScript Object Notation**

**JSON**: Stands for **JavaScript Object Notation**
- Created by Douglas Crockford
- Defines a way of **serializing** JavaScript objects
  - **to serialize**: to turn an object into a string that can be deserialized
  - **to deserialize**: to turn a serialized string into an object
Fetch API and JSON

The Fetch API also has built-in support for JSON:

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

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

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

Return `response.json()` instead of `response.text()` and Fetch will essentially call `JSON.parse()` on the response string.
Querying REST APIs
**RESTful API**: URL-based API that has these properties:

- Requests are sent as an **HTTP request**:
  - **HTTP Methods**: GET, PUT, POST, DELETE, etc
- Requests are sent to **base URL**, also known as an "**API Endpoint**"
- Requests are sent with a specified **MIME/content type**, such as HTML, CSS, JavaScript, plaintext, JSON, etc.
Look at the URL for this Google slide deck:
https://docs.google.com/presentation/d/1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8

- **presentation**: Tells the server that we are requesting a doc of type "presentation"

- **d/1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8**: Tells the server to request a doc ("d") with the document id of "1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8"
RESTful API

Almost every website on the internet uses RESTful URLs / RESTful APIs to handle requests to its servers.

Notable alternatives to REST:
  - GraphQL,
    - Used by Facebook since 2012
    - Open-sourced by Facebook since 2015
    - Still early but some big clients: GitHub, Pinterest
  - Falcort
    - Netflix's REST alternative, introduced ~2015
    - Probably cool but never hear of anyone using it
    - Doesn't even have a Wikipedia page
Using REST APIs
3rd-Party APIs

Many websites expose REST APIs to outside developers. These are often called "3rd-party APIs" or "Developer APIs".

Examples:
- Spotify
- Giphy
- GitHub
- Hoards of Google APIs
- Facebook
- Instagram
- Twitter
- etc...

Try Googling "<product name> API" to see if one exists for a given company!
Example: Spotify

Spotify has a **REST API** that external developers (i.e. people who aren't Spotify employees) can query:

Our Web API endpoints give external applications access to Spotify catalog and user data.

Web API Base URL: https://api.spotify.com  
User Guide | Tutorial | Code Examples

<table>
<thead>
<tr>
<th>METHOD</th>
<th>ENDPOINT</th>
<th>USAGE</th>
<th>RETURNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/albums/{id}</td>
<td>Get an album</td>
<td>album</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/albums?ids={ids}</td>
<td>Get several albums</td>
<td>albums</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/albums/{id}/tracks</td>
<td>Get an album’s tracks</td>
<td>tracks*</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/artists/{id}</td>
<td>Get an artist</td>
<td>artist</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/artists?ids={ids}</td>
<td>Get several artists</td>
<td>artists</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/artists/{id}/albums</td>
<td>Get an artist’s albums</td>
<td>albums*</td>
</tr>
</tbody>
</table>
Example: Spotify

REST API structure (details):
- The **Base URL** is https://api.spotify.com
- The **HTTP method** is GET
- The **API endpoint** to query is:
  https://api.spotify.com/v1/albums/<spotify_id>
- It returns **JSON data** about the album that's requested

Web API Base URL: https://api.spotify.com

<table>
<thead>
<tr>
<th>METHOD</th>
<th>ENDPOINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/albums/{id}</td>
</tr>
</tbody>
</table>
Example: Spotify

If we had Spotify Album ID 7aDBFWp72Pz4NZEtVBANi9, how would we make a GET request for the album information?

REST API structure (details):
- The **Base URL** is https://api.spotify.com
- The **HTTP method** is GET
- The **API endpoint** to query is:
  https://api.spotify.com/v1/albums/<spotify_id>
- It returns **JSON data** about the album that's requested
GET request: Browse to URL

Loading a URL in a browser issues an HTTP GET request for that resource.

So if we just piece together this URL:

- **API Endpoint:**
  
  https://api.spotify.com/v1/albums/\(<\text{spotify\_id}\>\)

- **Album ID:** 7aDBFWp72Pz4NZEtVBANi9

- **Request URL:**
  
  https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9

If you click on the link, you see it returns a JSON object.
GET request: fetch()

Actually, the fetch() API also issues an HTTP GET request by default.

So if we do:

```javascript
fetch('https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9')
  .then(onResponse)
  .then(onTextReady);
```

...we can load the JSON data as a JavaScript object, as we did with our .json files!

(CodePen / demo)
Let's write a web page that asks the user to enter an artist's name, then displays the albums of that artist, as provided by the Spotify Search API. (live demo)
Spotify search API

Spotify Search URL:
https://api.spotify.com/v1/search?type=album&q=query
E.g.
https://api.spotify.com/v1/search?type=album&q=beyonce

Q: Hey, what's that at the end of the URL?
- ?type=album&q=beyonce
Query parameters

You can pass parameters to HTTP GET requests by adding `query parameters` to the URL:

```
?type=album&q=beyonce
```

- Defined as key-value pairs
  - `param=value`
- The first query parameter starts with a `?`
- Subsequent query parameters start with `&`
Reminder: HTML elements

Single-line text input:

```html
<input type="text" />
```

In JavaScript, you can read and set the input text via `inputElement.value`

Some other input types:
- `Select`
- `Textarea`
- `Checkbox`
Q: What if you want the form to submit after you click "enter"?
Form submit

1. Wrap your input elements in a `<form>`

```html
<form>
  <input type="text" id="artist-text" />
  <input type="submit" />
</form>
```

You should also use `<input  type="submit">` instead of `<button>` for the reason on the next slide...
2. Listen for the 'submit' event on the form element:

```javascript
const form = document.querySelector('form');
form.addEventListener('submit', this._onSubmit);
```

This is why you want to use `<input type="submit">` instead of `<button>` -- the 'submit' event will fire on click for but not `<button>`. 
Form submit

3. Prevent the default action before handling the event through `event.preventDefault()`:

```javascript
_onSubmit(event) {
  event.preventDefault();
  const textInput = document.querySelector('#artist-text');
  const query = encodeURIComponent(textInput.value);

  this.albumUrls = [];
  fetch(SPOTIFY_PATH + query)
    .then(this._onResponse)
    .then(this._onJsonReady);
}
```

The page will refresh on submit unless you explicitly prevent it.
Album example

Solution: [GitHub](https://github.com) / [Demo](#)
Other REST APIs
Giphy API

**Search Endpoint**

Search all Giphy GIFs for a word or phrase. Punctuation will be stripped and ignored. Use a plus or url encode for phrases. Example paul+rudd, ryan+gosling or american+psycho.

http://api.giphy.com/v1/gifs/search?q=funny+cat&api_key=dc6zaTOxFJmzC

Example search query.

**Path**

/v1/gifs/search

**Parameters**

- q – search query term or phrase
- limit – (optional) number of results to return, maximum 100. Default 25.
- offset – (optional) results offset, defaults to 0.
- rating – (optional) limit results to those rated (y,g, pg, pg-13 or r).
- lang – (optional) specify default country for regional content; format is 2-letter ISO 639-1 country code. See list of supported languages here
- fmt – (optional) return results in html or json format (useful for viewing responses as GIFs to debug/test)

https://github.com/Giphy/GiphyAPI#search-endpoint
Yelp API

/businesses/search

This endpoint returns up to 1000 businesses based on the provided search criteria. It has some basic information about the business. To get detailed information and reviews, please use the business id returned here and refer to /businesses/[id] and /businesses/[id]/reviews endpoints.

Note: at this time, the API does not return businesses without any reviews.

**Request**

GET https://api.yelp.com/v3/businesses/search

**Parameters**

These parameters should be in the query string.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>string</td>
<td>Optional. Search term (e.g. &quot;food&quot;, &quot;restaurants&quot;). If term isn't included we search everything. The term keyword also accepts business names such as &quot;Starbucks&quot;.</td>
</tr>
<tr>
<td>location</td>
<td>string</td>
<td>Required if either latitude or longitude is not provided. Specifies the combination of &quot;address, neighborhood, city, state or zip, optional country&quot; to be used when searching for businesses.</td>
</tr>
<tr>
<td>latitude</td>
<td>decimal</td>
<td>Required if location is not provided. Latitude of the location you want to search nearby.</td>
</tr>
<tr>
<td>longitude</td>
<td>decimal</td>
<td>Required if location is not provided. Longitude of the location you want to search nearby.</td>
</tr>
</tbody>
</table>

Fetch gotchas
CORS error

If you try to fetch() this JSON file:
http://web.stanford.edu/class/cs193x/lectures/18/albums.json

You get this error:

Q: Why do we get this error, when the JSON file is served over HTTP?

- Browser policies for what resources a web page can load

- **Cross-origin**: between two different domains
  
  - If abc.com/users requests something from abc.com/search, it's still a **same-origin** request (not cross-origin) because it's the same domain
  
  - But if abc.com/foo requests something from xyz.com/foo, it's a **cross-origin** request.
CORS summarized

- You can make **same-origin** requests by default for any request type

- You can make **cross-origin** requests by default for:
  - Images loaded via `<img>`
  - CSS files loaded via `<link>`
  - JavaScript files loaded via `<script>`
  - Etc

- You **cannot** make cross-origin requests by default for:
  - Resources loaded via `fetch()` or XHR
CORS configuration

However, a web server can be configured to override these default rules:

- If you want to allow other domains to make `fetch()` requests to your servers, you can configure your server to allow them (e.g. on apache)
  - All 3rd party APIs do this, otherwise you couldn't access them

- If you don't want other domains to certain resources such as images, you can disallow them
In this class

In CS193X, we will either be:
- Making same-origin requests
- Making requests on APIs that have allowed cross-origin access

So you don't need to do anything with CORS for CS193X.

Still, CORS is good to know about:
- Helps you understand error messages
- You may have to deal with this in the future (common scenario: file:// trying to access an HTTP resource: HTTP resource must allow CORS for this to be allowed)
Fetch and closures

What if instead of code like this in a class: (CodePen)

```javascript
loadAlbums() {
  fetch(JSON_PATH)
    .then(this._onResponse)
    .then(this._onJsonReady);
}

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

_onResponse(response) {
  return response.json();
}
```
Fetch and closures

We wrote code that looked like this, where onResponse and onJsonReady were inner functions (CodePen):

```javascript
loadAlbums() {
    function onJsonReady(json) {
        this.albumInfo = json.albums;
        this._renderAlbums();
    }
    function onResponse(response) {
        return response.json();
    }
    fetch(JSON_PATH)
        .then(onResponse)
        .then(onJsonReady);
}
```
Fetch and closures

Even if we bind `loadAlbums`:

```javascript
class App {
  constructor() {
    this.loadAlbums = this.loadAlbums.bind(this);
  }
}
```

We get this error ([CodePen](https://codepen.io/john-doe/pen/XYZ123)):

```
Uncaught (in promise) TypeError: Cannot set property 'albumInfo' of undefined
```
Closures and this

Every function has its own "this" argument, meaning closures (inner functions) also have their own "this" arguments...

```javascript
loadAlbums() {
    function onJsonReady(json) {
        this.albumInfo = json.albums;
        this._renderAlbums();
    }
    function onResponse(response) {
        return response.json();
    }
    fetch(JSON_PATH)
        .then(onResponse)
        .then(onJsonReady);
}
```
Closures and this

So even if you've bound the this value for loadAlbums, it doesn't bind the this value for the closures.

```javascript
loadAlbums() {
    function onJsonReady(json) {
        this.albumInfo = json.albums;
        this._renderAlbums();
    }

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

    fetch(JSON_PATH)
        .then(onResponse)
        .then(onJsonReady);
}
```
Solution 1: Bind explicitly

You can bind the closures to the `this` value of `loadAlbums` when it is called: (CodePen)

```javascript
loadAlbums() {
    function onResponse(response) {
        return response.json();
    }
    function onJsonReady(json) {
        this.albumInfo = json.albums;
        this._renderAlbums();
    }
    fetch(JSON_PATH)
        .then(onResponse.bind(this))
        .then(onJsonReady.bind(this));
}
```
Solution 2: Bind with =>

Functions defined with the arrow syntax are auto-bound to the "this" of their enclosing context (CodePen):

```javascript
loadAlbums() {
  const onJsonReady = (json) => {
    this.albumInfo = json.albums;
    this._renderAlbums();
  };
  const onResponse = (response) => {
    return response.json();
  };
  fetch(JSON_PATH)
    .then(onResponse)
    .then(onJsonReady);
}
```
Solution 2: Bind with =>

We can also use the concise syntax:

```javascript
loadAlbums() {
    const onJsonReady = (json) => {
        this.albumInfo = json.albums;
        this._renderAlbums();
    };
    fetch(JSON_PATH)
        .then(response => response.json())
        .then(onJsonReady);
}
```
Single-threaded asynchrony
Recall: Discography page

We wrote a web page that lists the Mariah Carey albums stored in [albums.json](http://albums.json) and lets us sort the albums:

([CodePen](http://CodePen) / [demo](http://demo))
Error?!

If we click on one of the buttons before the albums load, we get an error:

```
Uncaught TypeError: Cannot read property 'sort' of undefined
  at App._sortAlbums (pen.js:34)
  at SortButton._onClick (pen.js:74)
```

Why?!
When we first load the page, the following things happen immediately:

1. Script creates App
2. App creates SortButtons
3. App requests albums.json through fetch()
When the `fetch()` finishes, the app creates an Album object for each album that was fetched:

4. `fetch()` finishes

5. App creates Album objects
Before `fetch()` returns

However, before the `fetch()` completes, a user might click the sort button:

- Script.js
- App
- `fetch()` is still pending
- Albums.json
- "App, you should sort your album data"
- SortButton
- Click event!
The `albumInfo` field is filled out after the `fetch()` from `loadAlbums()` returns.
But if the button is clicked before `fetch()` returns, `albumInfo` is not defined yet and we get an error.

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

```
Uncaught TypeError: Cannot read property 'sort' of undefined
  at App._sortAlbums (pen.js:34)
  at SortButton._onClick (pen.js:74)
```
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.)

[Diagram with arrows indicating a "Click event!" and "Fetch finishes!" sequence]
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
```javascript
_showButtons() {
    const buttonContainer = document.querySelector('#button-container');
    const ascButton = new SortButton(
        buttonContainer, 'By year, descending', this._sortAlbums, SORT_YEAR_ASC);
    const descButton = new SortButton(
        buttonContainer, 'By year, ascending', this._sortAlbums, SORT_YEAR_DESC);
    const alphaElement = document.querySelector('#alpha');
    const alphaButton = new SortButton(
        buttonContainer, 'By title, alphabetical', this._sortAlbums, SORT_ALPHA_TITLE);
}

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

Don't show buttons until JSON is loaded ready: [CodePen](https://codepen.io)
Solutions

Or you can make the button event handler work independent of the fetch call
- Initialize `albumInfo` to an empty array in the constructor
- Sorting nothing does nothing, which is fine

[CodePen](https://codepen.io)
Single-threaded asynchrony
Is it possible for the `_onJsonReady` function to fire *in the middle* of `sortAlbums`?
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.
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

```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);
  }
}
```

Q: If in JavaScript, only one thing happens at a time, does that mean only one image loads at a time?
Empirically, that doesn't seem to be the case:

**Mariah Carey's albums**

- By year, descending
- By year, ascending
- By title, alphabetical
Network tab

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)
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');
```

(globally 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');
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

```
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');
```
Call stack + setTimeout

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

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

Call Stack

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');
Call stack + setTimeout

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

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

Call Stack

```
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');
```
```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');
```

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 + 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()
```
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

`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
What "enqueues" onTimerDone? How does it get fired?
Tasks, Micro-tasks, and the Event Loop
Tasks and the Event Loop

The JavaScript runtime can do only one thing at a time...

Call Stack

- 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.
Demo

Philip Roberts wrote a nice visualizer for the JS event loop:

- `setTimeout`
- `With click`