

# Interactive Web Programming

1st semester of 2021

Murilo Camargos  
(**`murilo.filho@fgv.br`**)

Heavily based on [Victoria Kirst](#) slides

# Schedule

## Today:

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

## Next:

- D3 lib!

**HW4 is out!** Due May 11.

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:

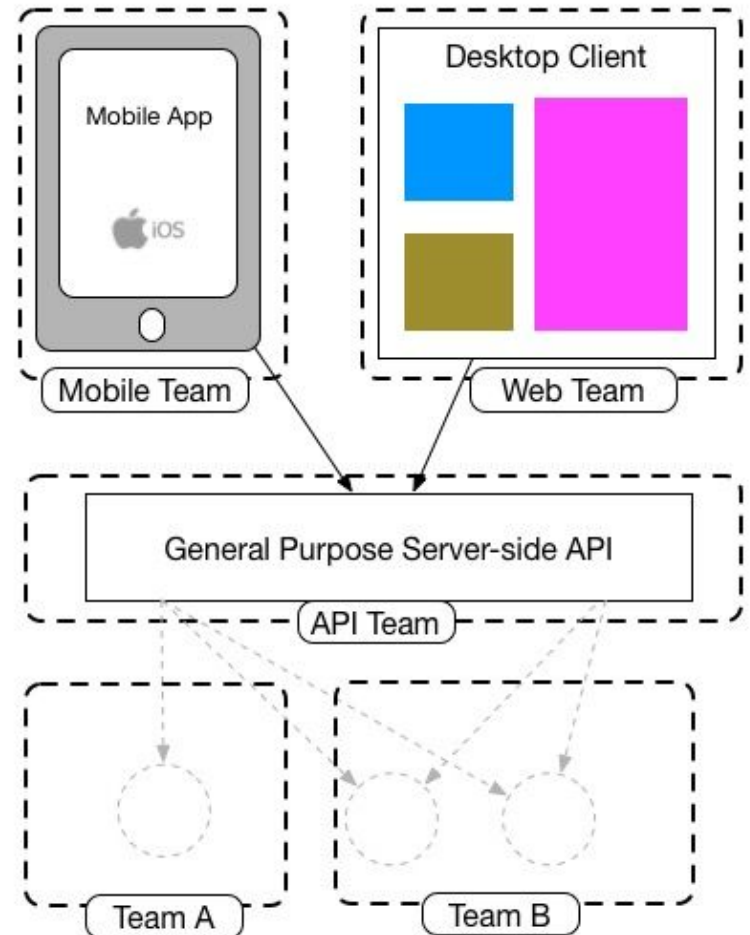
```
function onJsonReady(json) {  
    console.log(json);  
}  
  
function onResponse(response) {  
    return response.json();  
}  
  
fetch('images.json')  
    .then(onResponse)  
    .then(onJsonReady);
```

Return  
`response.json()`  
instead of  
`response.text()`  
and Fetch will  
essentially call  
`JSON.parse()` on the  
response string.

# Querying REST APIs

# Why APIs?

- Simple and standardized to access resources.
- It can be easily scalable.
- It offers more security through isolation.



# RESTful API

RESTful API: URL-based API that has these properties:

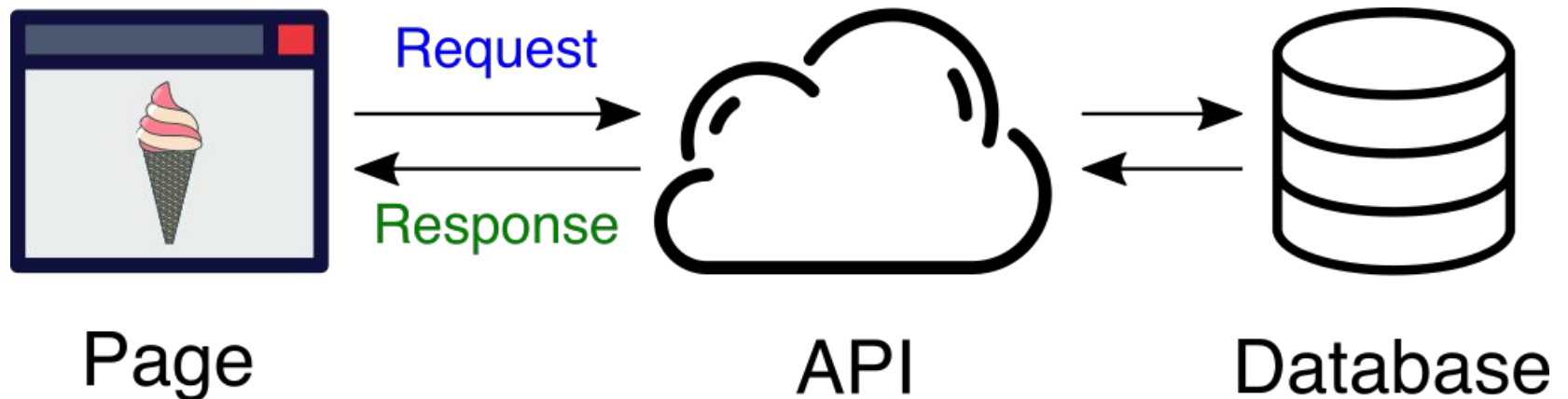
- Requests are sent as an **HTTP request**:
  - HTTP Methods: GET, PUT, POST, DELETE, etc
- Requests are sent to **base URL**, also known as an "**API Endpoint**"
- Simple and standardized
  - Always use HTTP protocol and methods
- Scalable and stateless
  - Don't have to synchronize data state across front and back-ends.
- Good performance with caching



# RESTful API example

Let's say we have an Ice Cream Shop website that allows the user to see the available flavors and the manager to change that information.

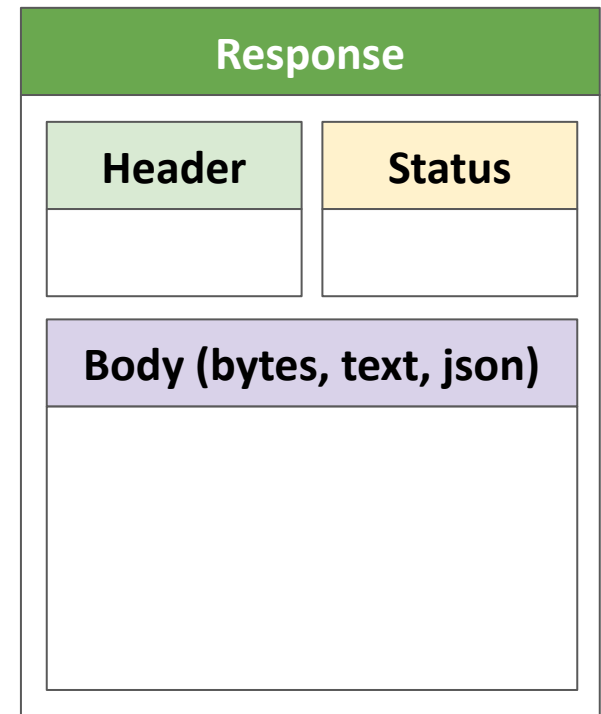
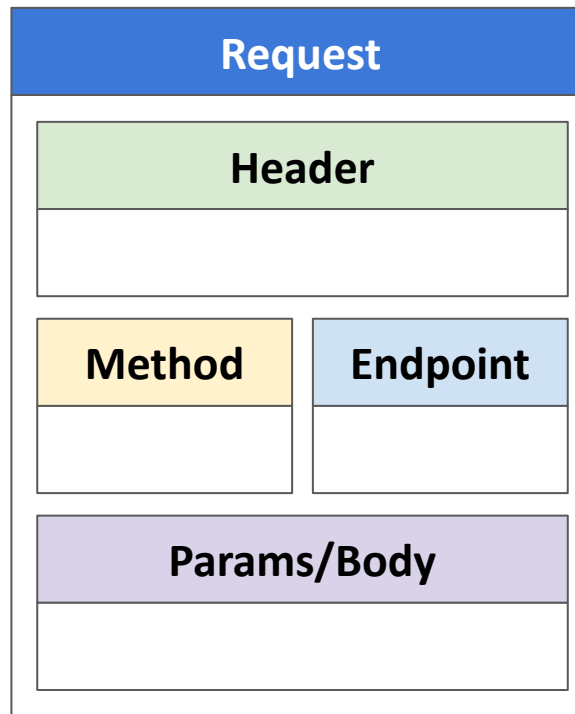
The API can be accessed at [\*\*https://www.icecream.com/api/\*\*](https://www.icecream.com/api/)



# RESTful API example

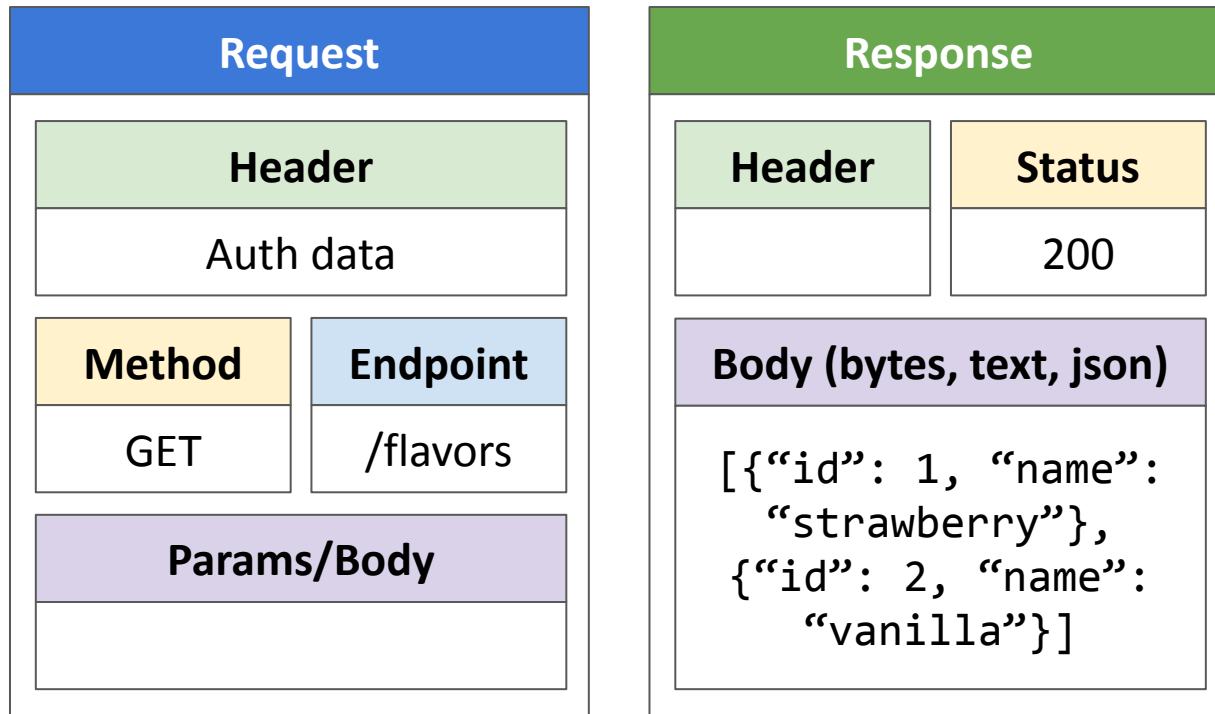
We would like to perform CRUD operations:

Operation	HTTP method
Create	POST
Read	GET
Update	PUT
Delete	DELETE



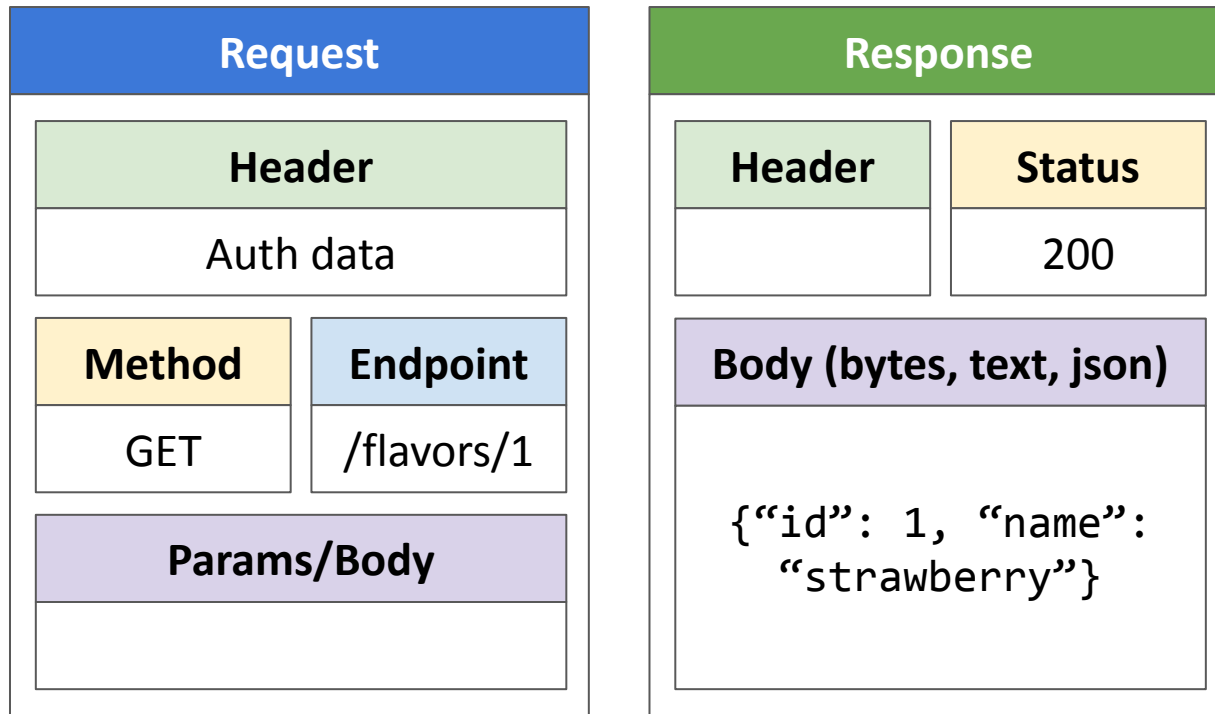
# RESTful API example

If we want to read the list of available flavors:



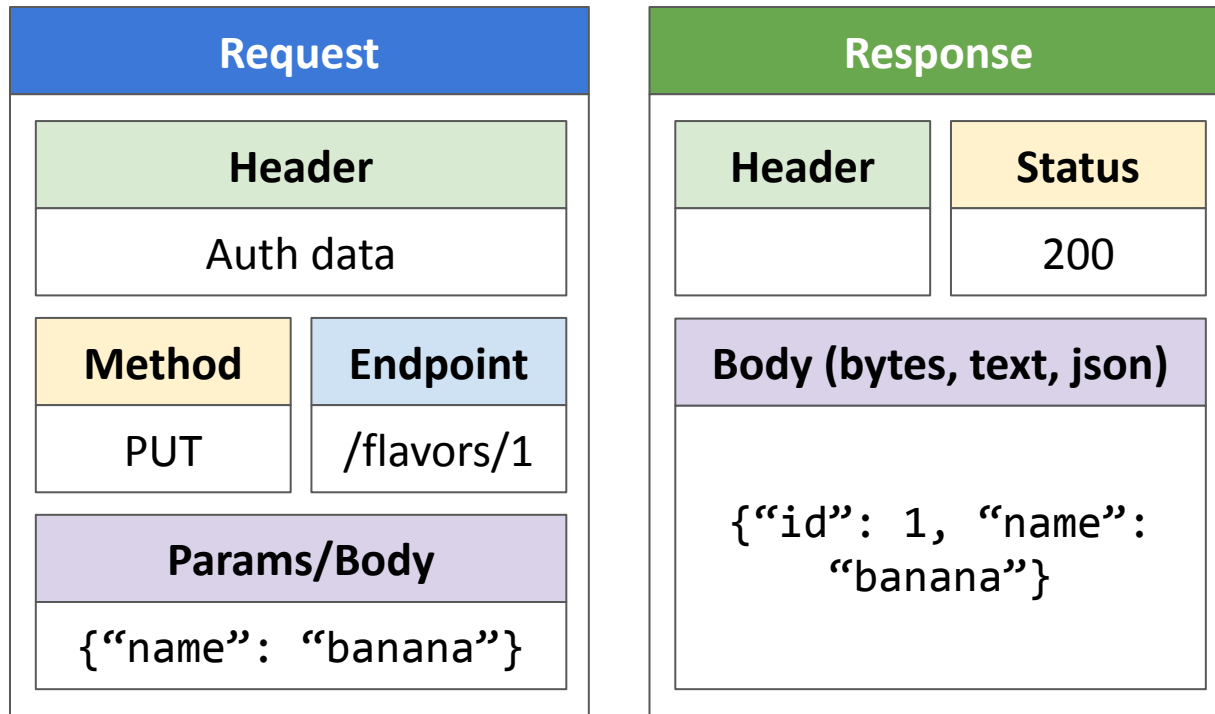
# RESTful API example

We can also get a specific flavor



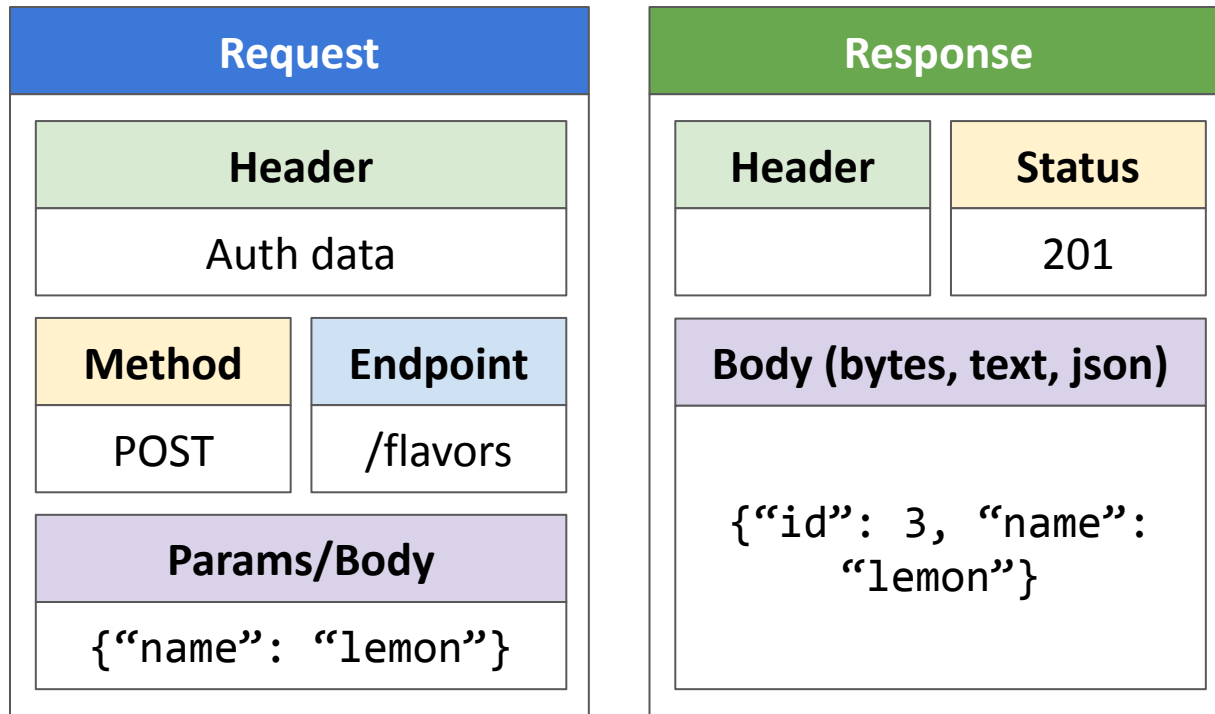
# RESTful API example

Let's say we want to change a flavor's name



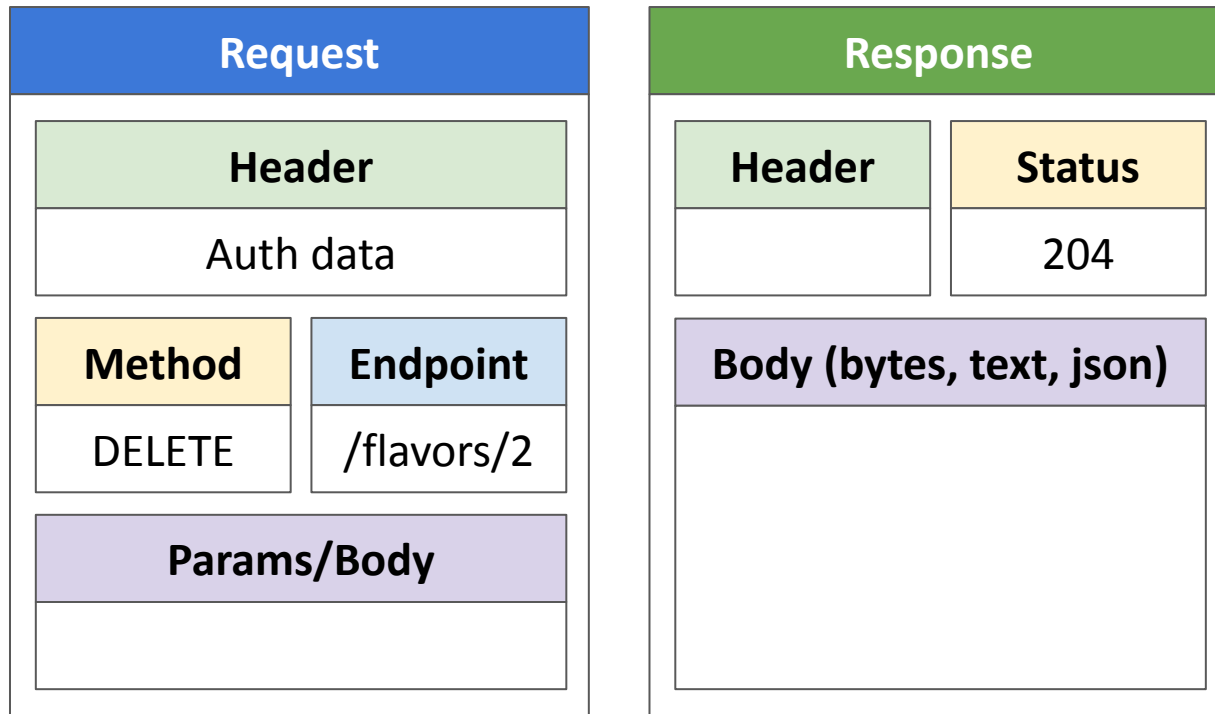
# RESTful API example

Let's say we want to create a new flavor



# RESTful API example

We can also delete a resource using DELETE HTTP method



# 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
- [Falcor](#)?
  - 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: TVMaze

TVMaze has a [REST API](#) that external developers (i.e. people who aren't TVMaze employees) can query:



If you want to add TV information to your website or app then you've come to the right place!

We provide a free, fast and clean REST API that's easy to use, returns JSON and conforms to the HATEOAS and HAL principles. The root url is <http://api.tvmaze.com> and the available endpoints are documented below. If you have any questions or suggestions regarding the API, please post them on [our forums](#).

In addition to the free public API, there's a user-level API available for all [Premium](#) members. The documentation for the user API can be viewed [here](#).

To stay up to date with the latest changes, you can follow the changelog thread [here](#).

## Table of Contents

### Search

- Show Search
- Show single search
- Show Lookup
- People search

### Schedule

- Web/Streaming Schedule
- Full Schedule

### Shows

- Show main information
- Show episode list
- Episode by number

- Episodes by date
- Show seasons
  - Season episodes
- Show cast
- Show crew
- Show AKA's
- Show images
- Show index

### Episodes

- Episode main information

### People

- Person main information

- Person cast credits
- Person crew credits

### Updates

- Show updates

### Embedding

### Images

### HTTPS

### Caching

### Rate Limiting

### Cors

### Licensing

### Enterprise API

# Example: TVMaze

## REST API structure ([details](#)):

- The **Base URL** is `https://api.tvmaze.com`
- The **HTTP method** is GET
- The **API endpoint** to query is:  
`https://api.tvmaze.com/search/shows?q=:query`
- It returns **JSON data** about the album that's requested

---

### Show search

Search through all the shows in our database by the show's name. A fuzzy algorithm is used (with a fuzziness value of 2), meaning that shows will be found even if your query contains small typos. Results are returned in order of relevancy (best matches on top) and contain each show's full information.

The most common usecase for this endpoint is when you're building a local mapping of show names to TVmaze ID's and want to make sure that you're mapping to exactly the right show, and not to a different show that happens to have the same name. By presenting each show's basic information in a UI, you can have the end-user pick a specific entry from that list, and have your application store the chosen show's ID or URL. Any subsequent requests for information on that show can then be directly made to that show's URL.

- URL: `/search/shows?q=:query`
- Example: <http://api.tvmaze.com/search/shows?q=girls>

# Example: TVMaze

If we had a TV Show name “The Witcher”, how would we make a GET request for the album information?

## REST API structure ([details](#)):

- The **Base URL** is `https://api.tvmaze.com`
- The **HTTP method** is GET
- The **API endpoint** to query is:  
`https://api.tvmaze.com/search/shows?q=:query`
- 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.tvmaze.com/search/shows?q=The Witcher`

- **Query:** *The Witcher*

- **Request URL:**

[https://api.tvmaze.com/search/shows?q=\*The Witcher\*](https://api.tvmaze.com/search/shows?q=<i>The Witcher</i>)

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:

```
fetch('https://api.tvmaze.com/search/shows?q=The  
Witcher')  
    .then(onResponse)  
    .then(onTextReady);
```

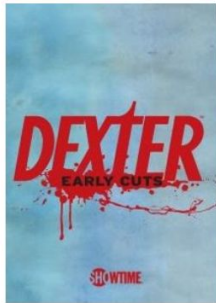
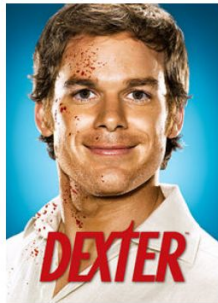
...we can load the JSON data as a JavaScript object, as we did with our .json files!

([CodePen](#))

# Shows example

Let's write a web page that asks the user to enter a search query, then displays the found tv shows

Enter a TV show name:





# TVMaze search API

TVMaze Search URL:

<https://api.tvmaze.com/search/shows?q=query>

E.g.

[https://api.tvmaze.com/search/shows?q=\*The Witcher\*](https://api.tvmaze.com/search/shows?q=The%20Witcher)

**Q: Hey, what's that at the end of the URL?**

- ?q=The Witcher

# Query parameters

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

**?**q=The Witcher**&**param2=...

- 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:

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

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

Some other input types:

- [Select](#)
- [Textarea](#)
- [Checkbox](#)

# Form submit

Q: What if you want the form to submit after you click "enter"?

# Form submit

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

```
<form>  
  <input type="text" id="tv-show-text" />  
  <input type="submit" />  
</form>
```

You should also use `<input type="submit">` instead of `<button>` for the reason on the next slide...

# Form submit

2. Listen for the 'submit' event on the form element:

```
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()`:

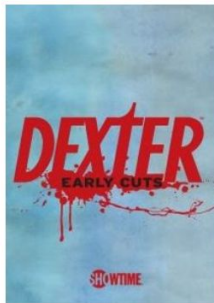
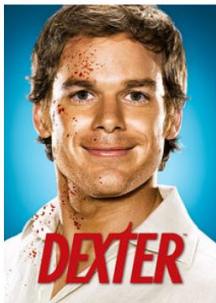
```
_onSubmit(event) {  
  event.preventDefault();  
  const textInput = document.querySelector('#tv-show-text');  
  const query = encodeURIComponent(textInput.value);  
  
  this.showUrls = [];  
  fetch(TVMAZE_PATH + query)  
    .then(this._onResponse)  
    .then(this._onJsonReady);  
}
```

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

# Show search example

Solution: [GitHub](#) / [Demo](#)

Enter a TV show name:





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=dc6zaT0xFJmzC
```

[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.

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

[https://www.yelp.com/developers/documentation/v3/business\\_search](https://www.yelp.com/developers/documentation/v3/business_search)

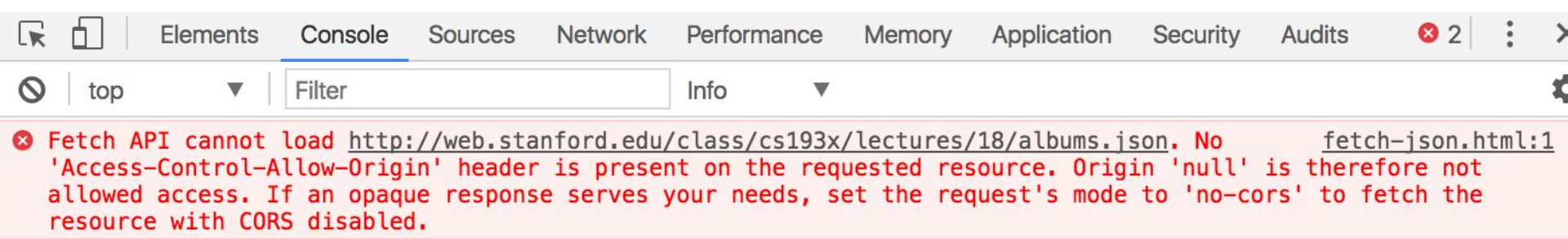
Fetch gotchas

# 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?**

# CORS

**CORS:** Cross-Origin Resource Sharing ([wiki](#))

- 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 IWP, 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 IWP.**

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](#))

```
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](#)):

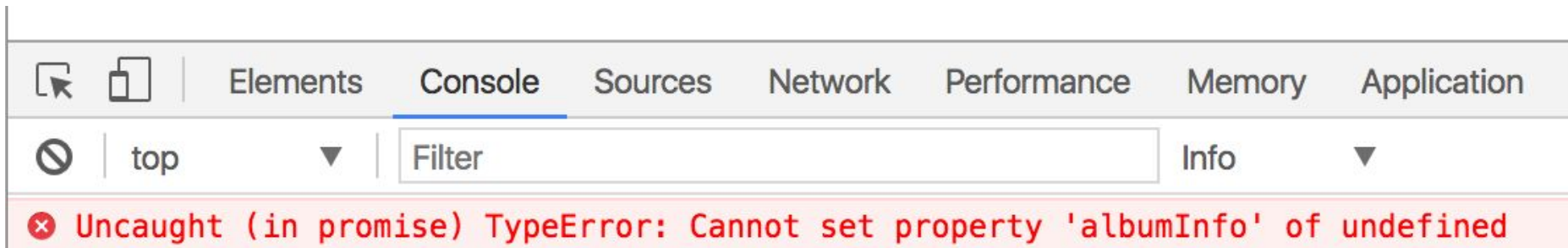
```
loadAlbums() {  
  function onJsonReady(json) {  
    this.albumInfo = json.albums;  
    this._renderAlbums();  
  }  
  function onResponse(response) {  
    return response.json();  
  }  
  fetch(JSON_PATH)  
    .then(onResponse)  
    .then(onJsonReady);  
}
```

# Fetch and closures

Even if we bind loadAlbums:

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

We get this error ([CodePen](#)):



# Closures and this

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

```
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.

```
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](#))

```
loadAlbums() {  
  function onJsonReady(json) {  
    this.albumInfo = json.albums;  
    this._renderAlbums();  
  }  
  function onResponse(response) {  
    return response.json();  
  }  
  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](#)):

```
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:

```
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

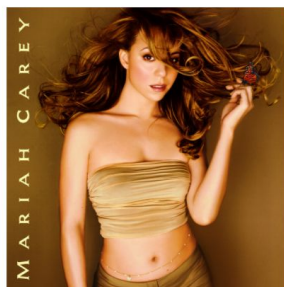
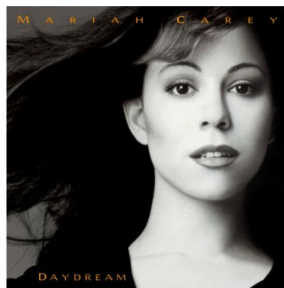
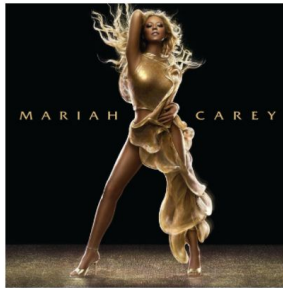
We wrote a web page that lists the Mariah Carey albums stored in [albums.json](#) and lets us sort the albums: ([CodePen](#) / [demo](#))

## Mariah Carey's albums

By year, descending

By year, ascending

By title, alphabetical



# Error?!

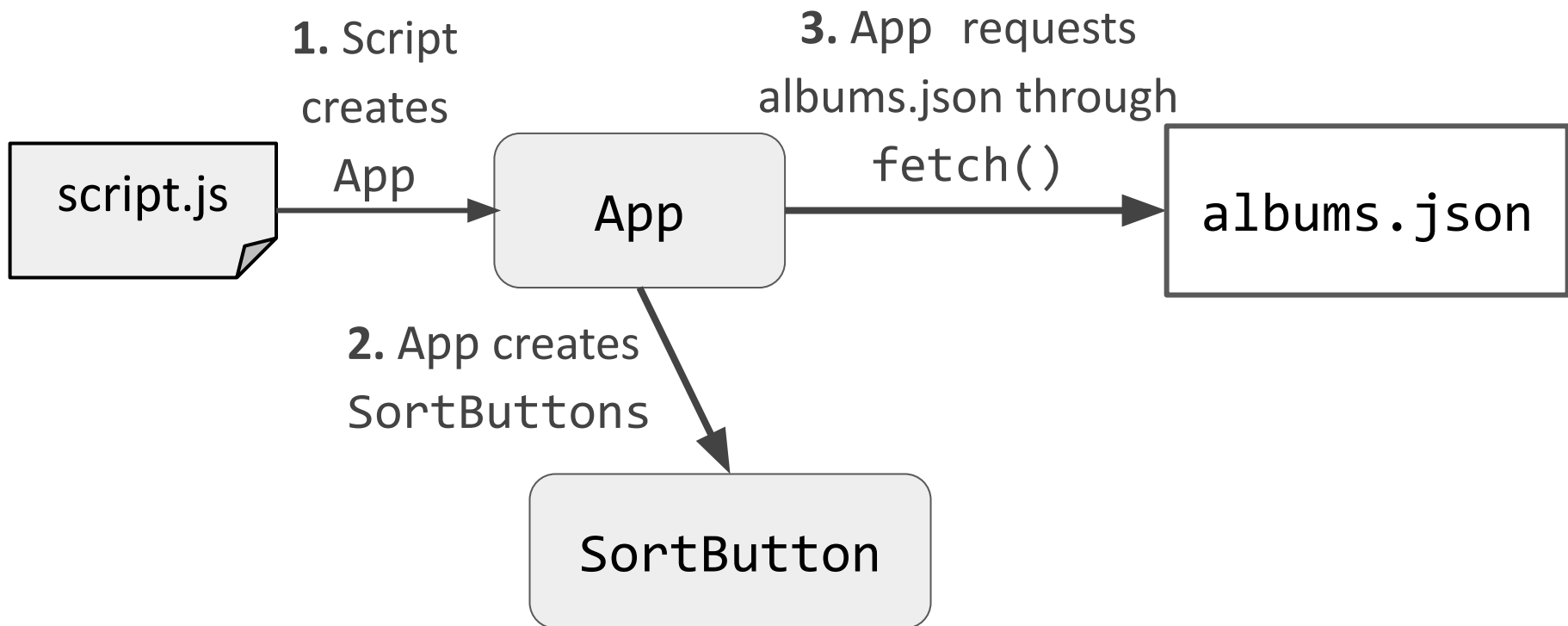
If we click on one of the buttons before the albums load, we get an error:

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

## Why?!

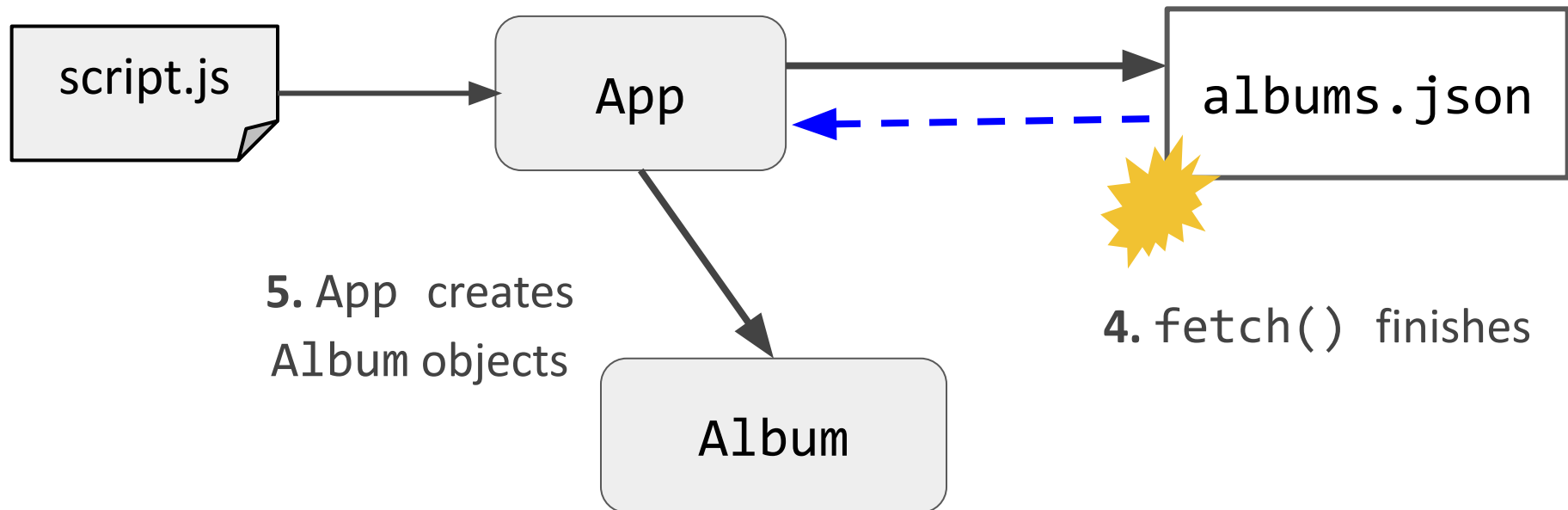
# On page load

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



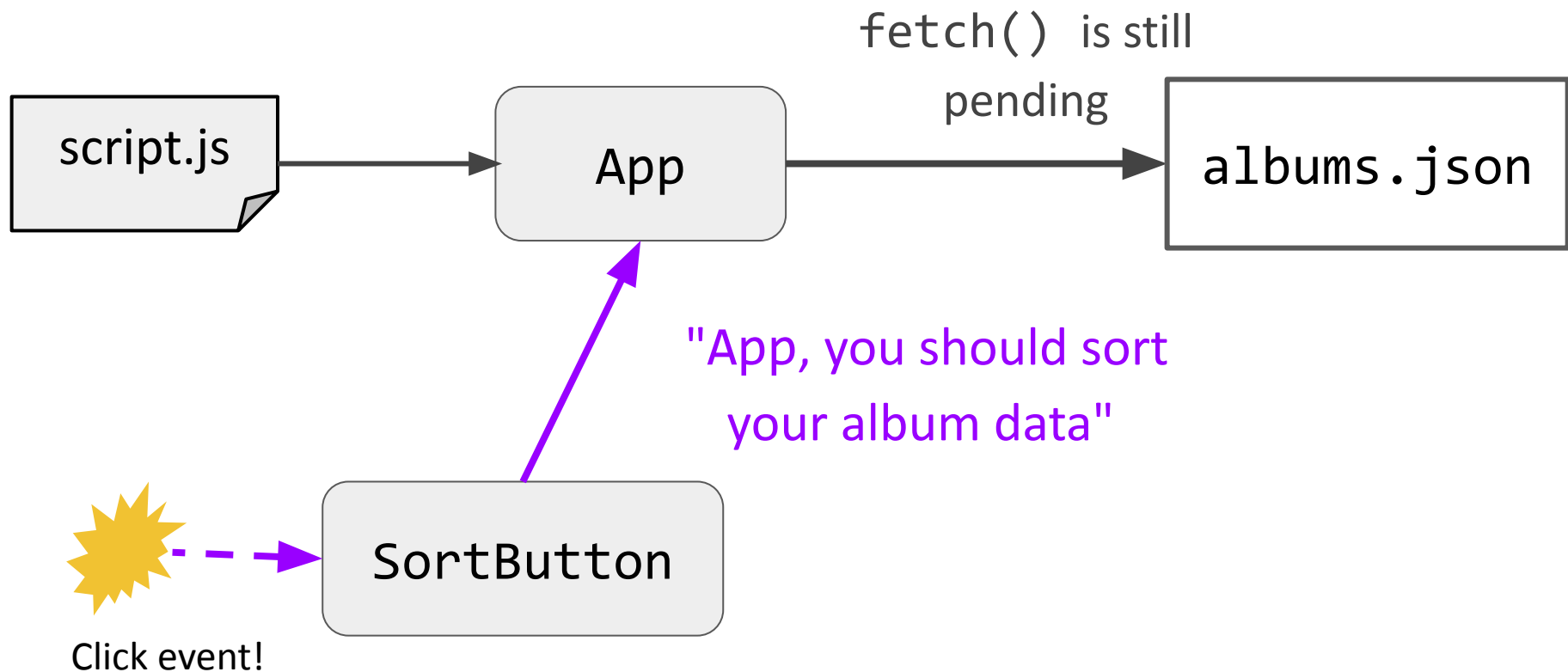
# On page load

When the `fetch()` finishes, the app creates an Album object for each album that was fetched:



# Before `fetch()` returns

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



The albumInfo field is filled out after the fetch() from loadAlbums() returns



```
loadAlbums() {  
  fetch(JSON_PATH)  
    .then(this._onResponse)  
    .then(this._onJsonReady);  
}  
  
_onJsonReady(json) {  
  this.albumInfo = json.albums;  
  this._renderAlbums();  
}  
  
_onResponse(response) {  
  return response.json();  
}
```

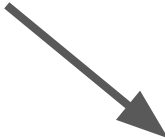
A solid blue arrow points from the `fetch()` call in the `loadAlbums()` function to the `this.albumInfo` assignment in the `_onJsonReady` method. A dashed blue arrow points from the text "The albumInfo field is filled out after the fetch() from loadAlbums() returns" to the same `this.albumInfo` assignment.

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



By year, descending

But if the button is  
clicked before  
fetch() returns,  
albumInfo is not  
defined yet and we  
get an error.

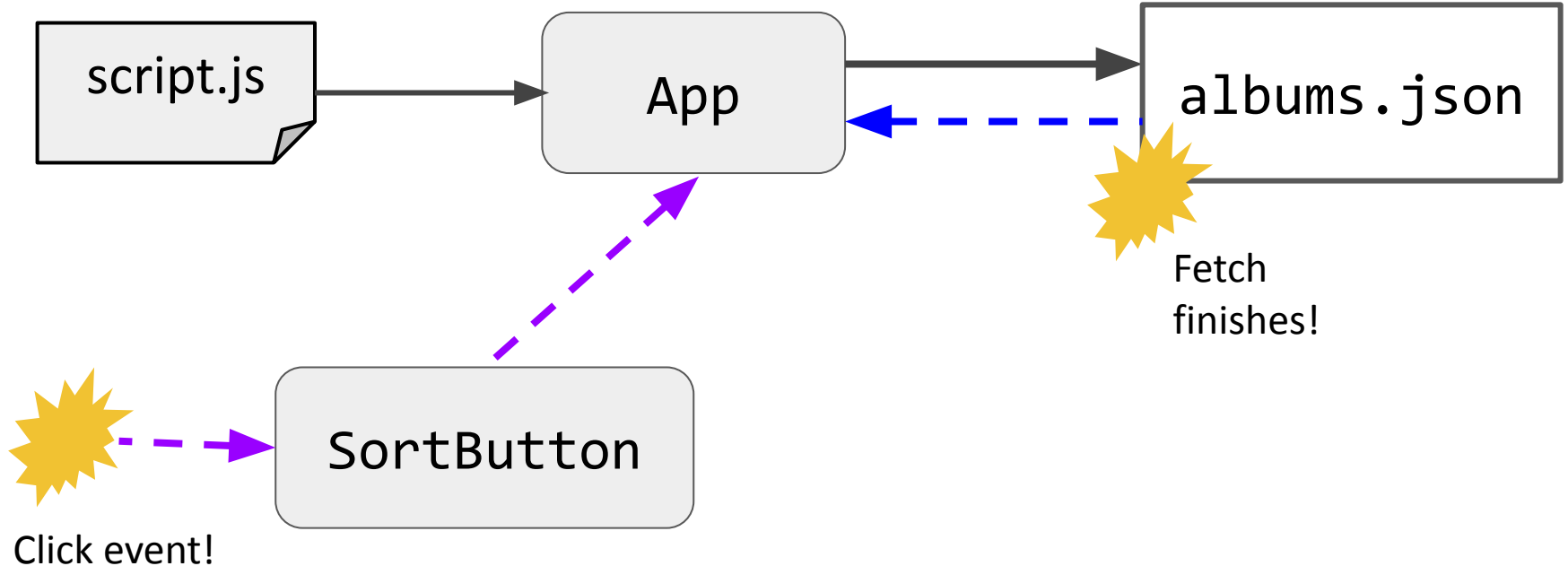


```
_sortAlbums(sortFunction) {  
  this.albumInfo.sort(sortFunction);  
  this._renderAlbums();  
}
```

✖ ▶ 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.

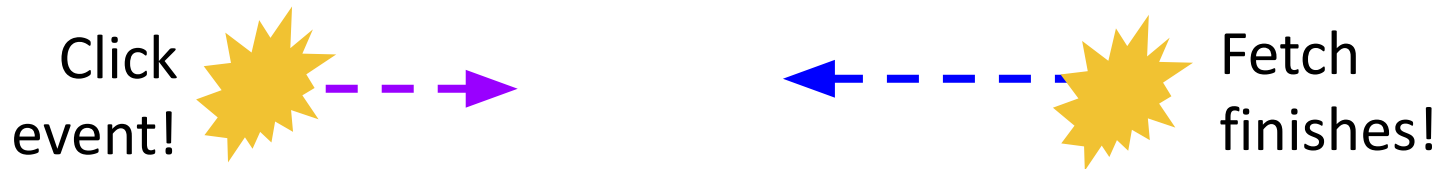


# General problem

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.)



# 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

```
_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](#)

# 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](#)

Single-threaded asynchrony

Is it possible for the  
\_onJsonReady function  
to fire \*in the middle\* of  
sortAlbums?




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



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

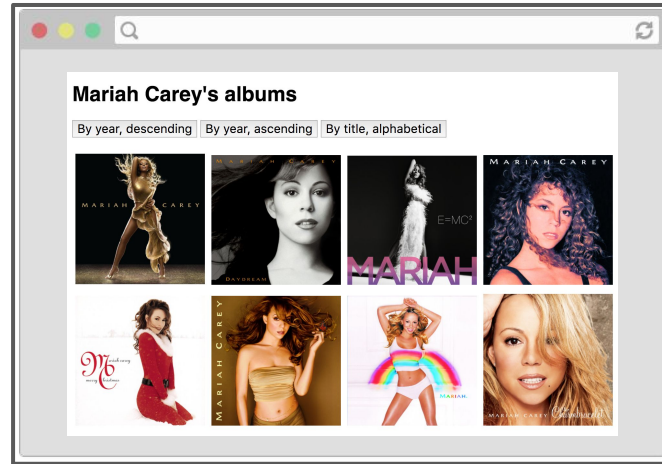
```
_onResponse(response) {  
  return response.json();  
}
```

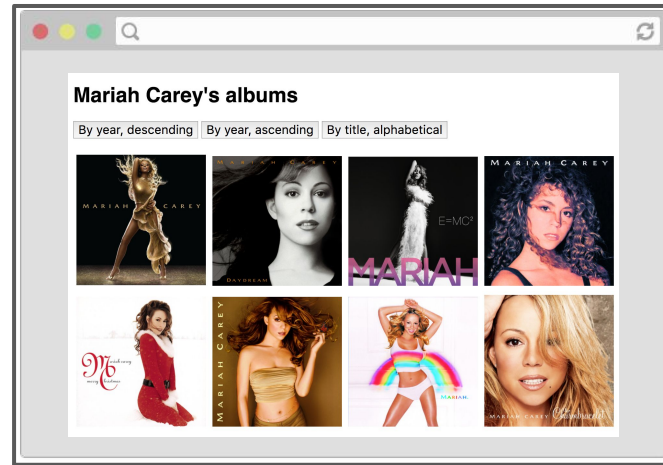


```
_sortAlbums(sortFunction) {  
  this.albumInfo.sort(sortFunction);  
  this._renderAlbums();  
}
```



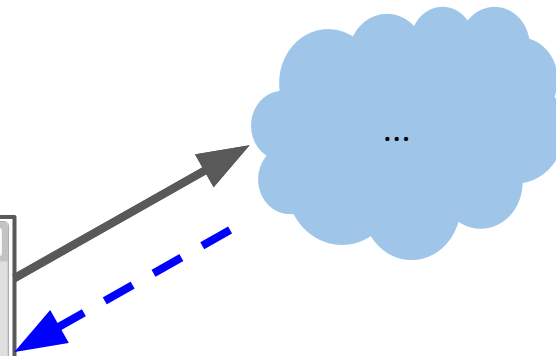
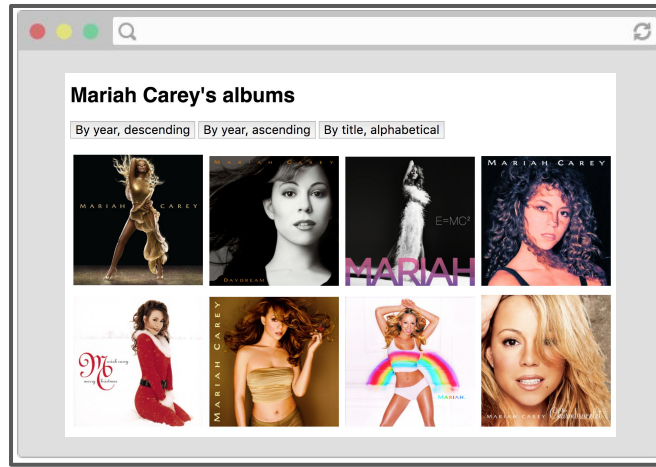
The browser is  
fetching  
albums.json...





By year, descending

User clicks a  
button, so the  
event handler is  
running



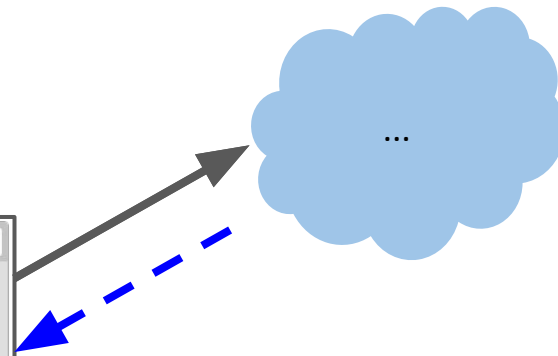
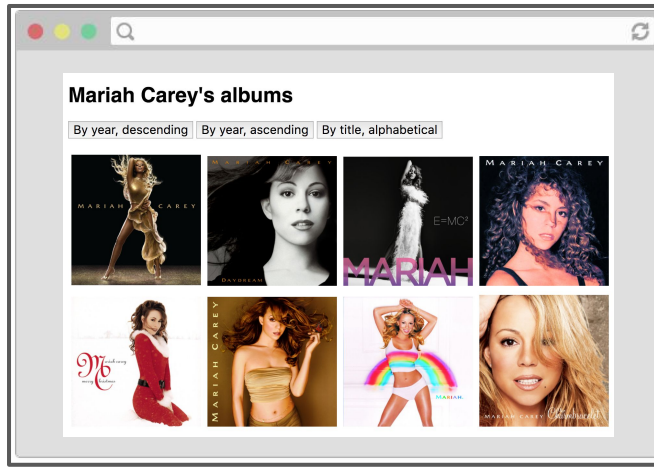
Is it possible that **while** the click handler is still running (still on the call stack), the `fetch()` callback also fires?



```
_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**.



```
_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

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

```
class Album {  
  constructor(albumContainer, imageUrl) {  
    // Same as document.createElement('img');  
    const image = new Image();  
    image.src = imageUrl;  
    albumContainer.append(image);  
  }  
}
```

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

# Image loading

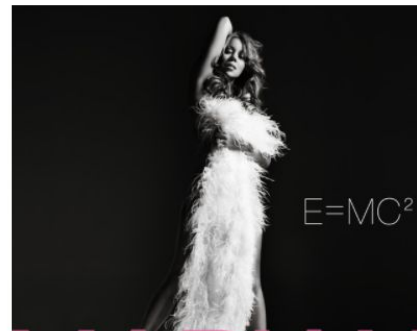
Empirically, that doesn't seem to be the case:

## 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:

Name	Status	Type	Initiator	Size	Time	Waterfall	
 0638f0ddf70003cb94b43aa5e4004d85...	200	jpeg	Other	4.0 KB	13.25 s		
 bca35d49f6033324d2518656531c9a89...	200	jpeg	Other	4.0 KB	13.25 s		
 82f13700dfa78fa877a8cdec725ad552c...	200	jpeg	Other	451 B	13.25 s		
 676275b41e19de3048fddfb72937ec0db...	200	jpeg	Other	2.7 KB	13.25 s		
 2424877af9fa273690b688462c5afbad6...	200	jpeg	Other	452 B	13.25 s		
 dca82bd9c1ccae90b09972027a408068...	200	jpeg	Other	453 B	557 ms		
 0638f0ddf70003cb94b43aa5e4004d85...	200	jpeg	Other	454 B	696 ms		
 bca35d49f6033324d2518656531c9a89...	200	jpeg	Other	451 B	790 ms		
 82f13700dfa78fa877a8cdec725ad552c...	200	jpeg	Other	451 B	Pending		
 676275b41e19de3048fddfb72937ec0db...	200	jpeg	Other	450 B	Pending		
 2424877af9fa273690b688462c5afbad6...	200	jpeg	Other	452 B	Pending		

**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>

And for a perfectly great deep dive on this, see Jake Archibald's blog post:

<https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/>

<https://www.youtube.com/watch?v=cCOL7MC4PI0>

These slides are inspired by these resources!)

# setTimeout


To help us understand the event loop better, let's learn about a new command, [setTimeout](#):

`setTimeout(function, delay);`

- *function* will fire after *delay* milliseconds
- [CodePen example](#)

# Call stack + setTimeout

## Call Stack



```
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

## Call Stack

```
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

## Call Stack

```
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

## Call Stack

```
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

## Call Stack

```
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)



# Call stack + setTimeout

## Call Stack

```
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

## Call Stack

```
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

## Call Stack

```
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

## Call Stack

```
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

## Call Stack

```
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');
```

onTimerDone()

# Call stack + setTimeout

## Call Stack

```
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

## Call Stack

```
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');
```

onTimerDone()

# Call stack + setTimeout

## Call Stack

```
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


## Call Stack

```
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');
```

onTimerDone()

# Call stack + setTimeout

## Call Stack



```
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');
```

onTimerDone()

# Call stack + setTimeout

## Call Stack

```
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

```
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 "enqueues" onTimerDone?  
How does it get fired?

## Call Stack

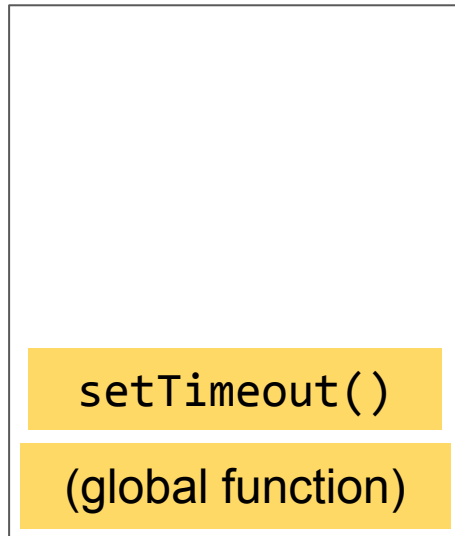
setTimeout(...);

(global function)

# Tasks, Micro-tasks, and the Event Loop

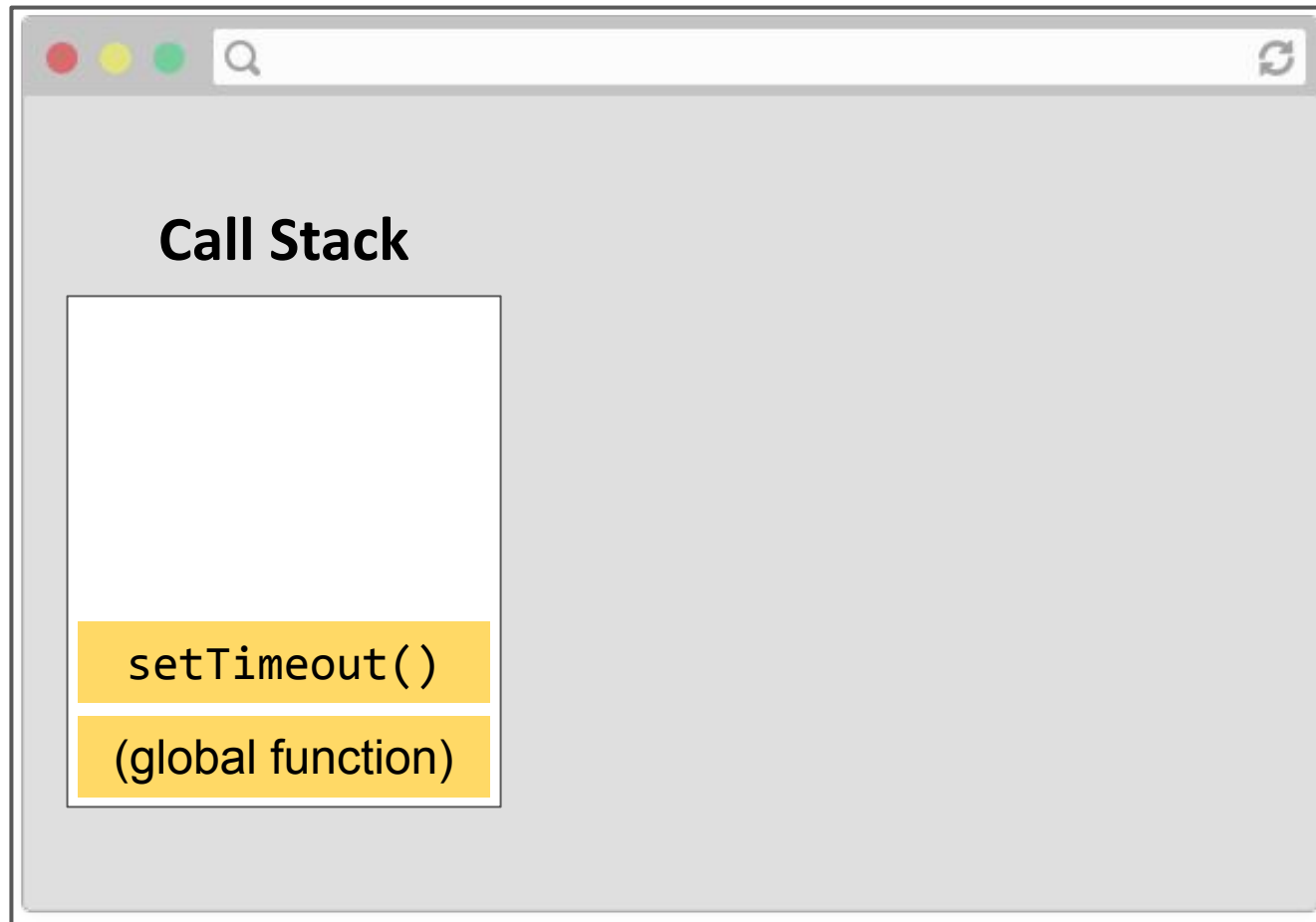
# Tasks and the Event Loop

## Call Stack

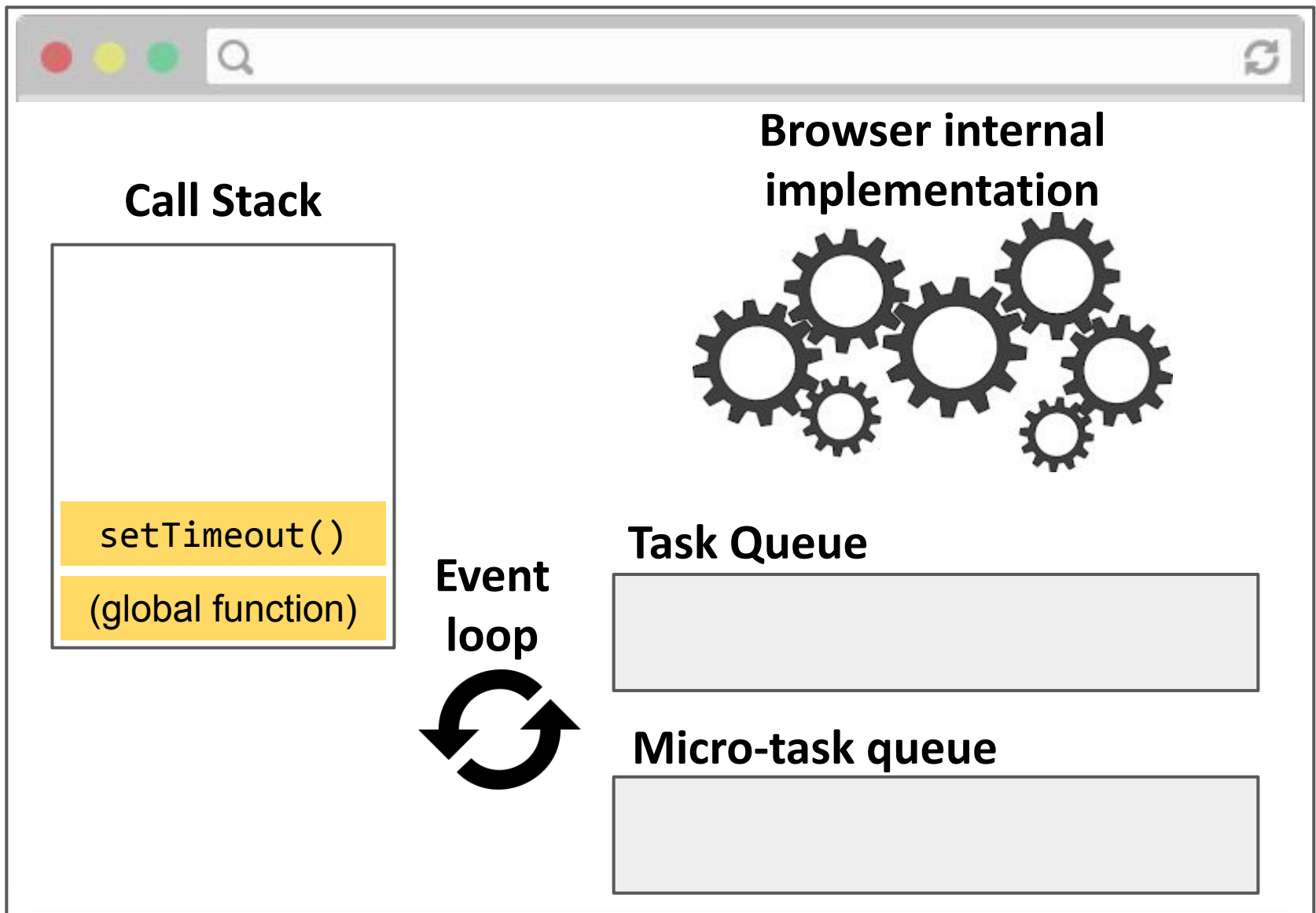


The JavaScript runtime can do only one thing at a time...

# Tasks and the Event Loop



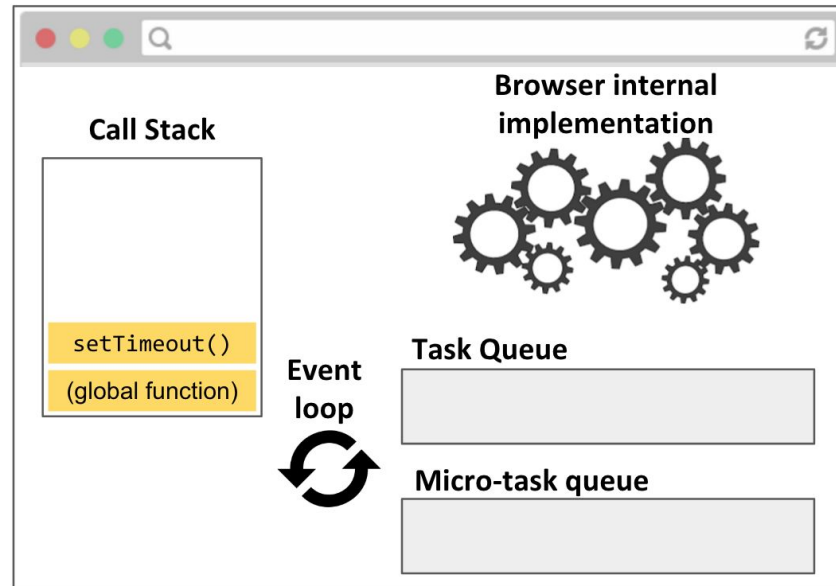
But the JS runtime runs within a browser, which can do multiple things at a time.



Here's a picture of the major pieces involved in executing JavaScript code in the browser.

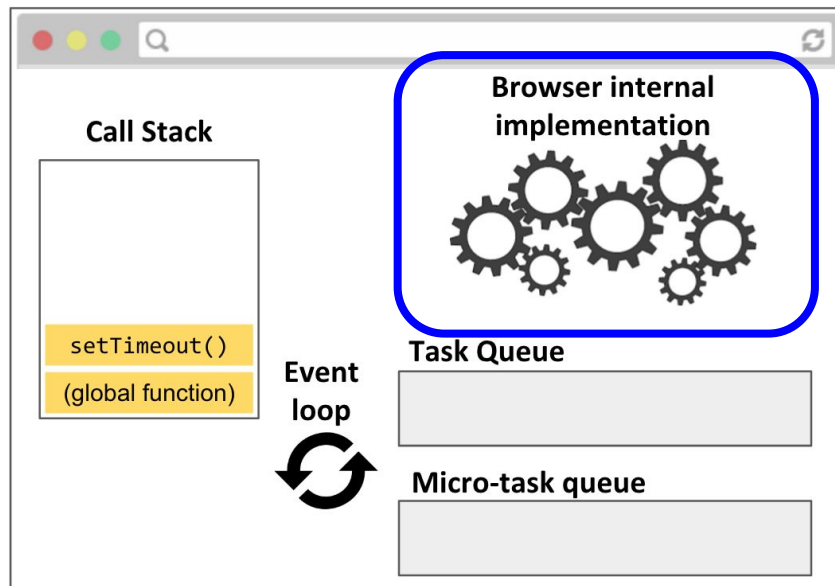


# JS execution



- **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.

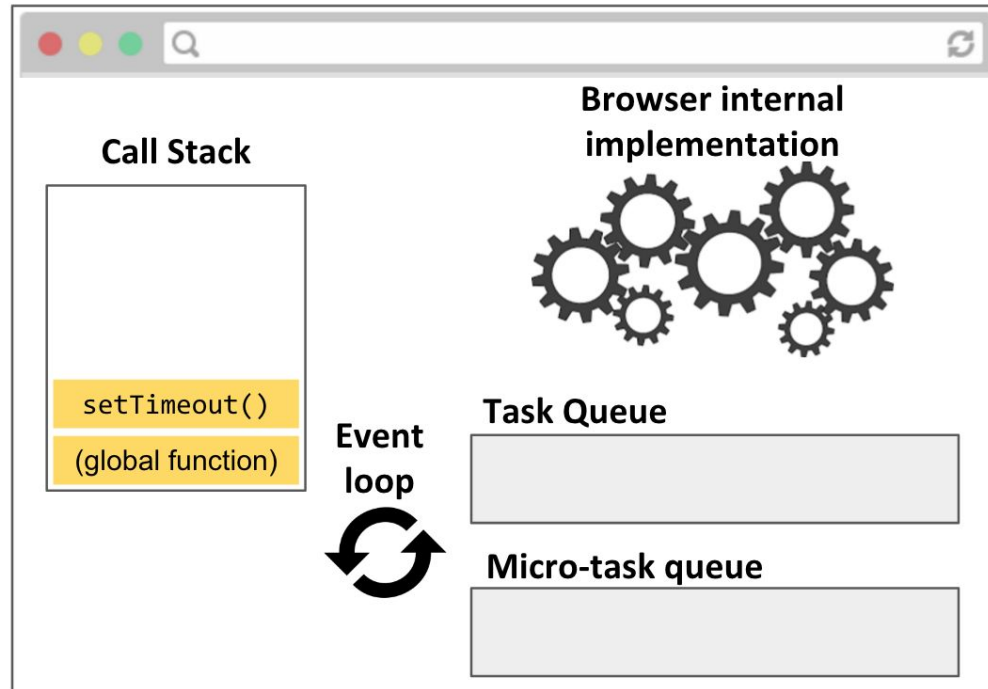
# JS execution



**The browser itself is multi-threaded and multi-process!**

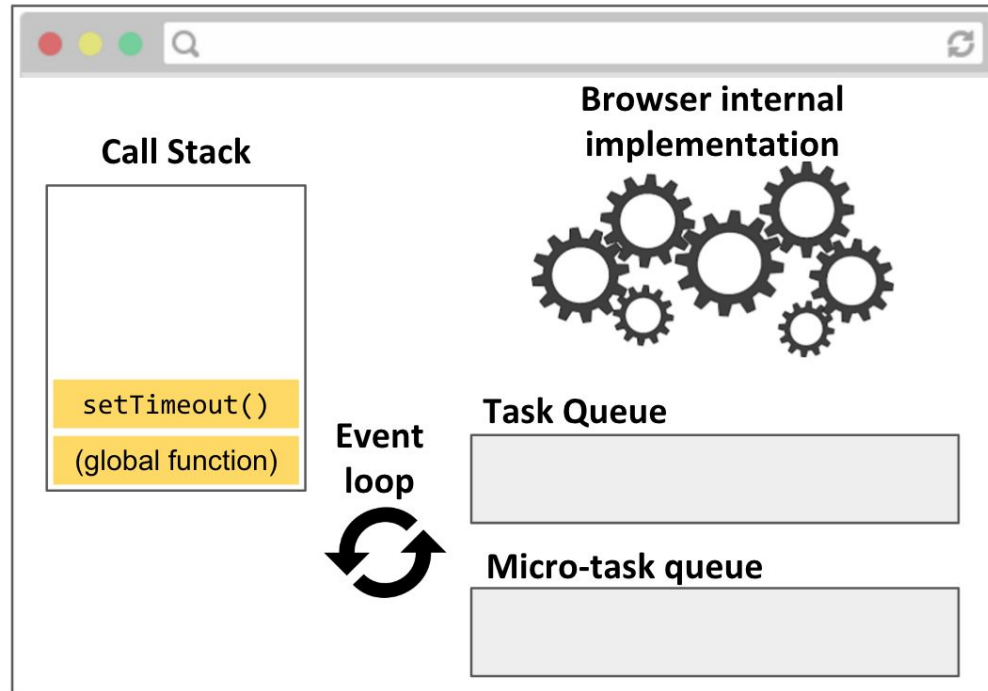
- **Call stack:** JavaScript runtime call stack. Executes the JavaScript commands, functions.
- **Browser internal implementation:** The C++ code that executes in response to native JavaScript commands, e.g. `setTimeout`, `element.classList.add('style')`, etc.

# JS execution



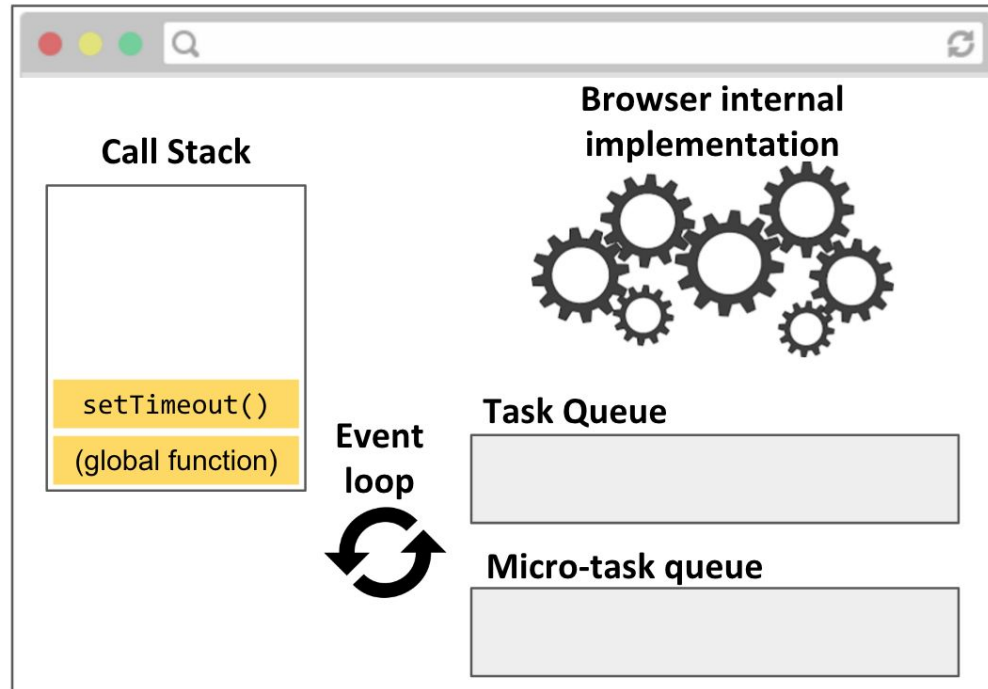
- **Task Queue:** When the browser internal implementation notices a callback from something like `setTimeout` or `addEventListener` is should be fired, it creates a Task and enqueues it in the Task Queue

# JS execution



- **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](#)