Asynchronous

- Matching human perception and expectations
- Asynchronous execution
- Fetch API
- Worker thread

Responsive* User Interfaces

- A responsive UI *feels* like it responds in a timely manner
  - Examples:

- This is accomplished primarily in **two ways**:
  1. Designing for human perception and expectations
  2. Using asynchronous execution

Responsiveness is not just system performance

* not related to **responsive layouts (**the layout term for adapting to different window sizes and/or devices)

Human Perception of Time

- Knowing *the duration of perceptual and cognitive processes* can inform the design of interactive systems that feel responsive
- Can examine results of *Mental Chronometry* studies
  - Minimal time to detect a gap of silence in sound: 4 ms
  - Minimal time to be affected by a visual stimulus: 10 ms
  - Time that vision is suppressed during a saccade: 100 ms
  - Maximum interval between cause-effect events: 140 ms
  - Time to comprehend a printed word: 150 ms
  - Visual-motor reaction time to an observed event: 1 s
  - Time to prepare for conscious cognition task: 10 s
  - Duration of unbroken attention to a single task: 6 s to 30 s

(times approximate)

Continuous Latency

Minimal time to be affected by a visual stimulus: 10 ms

→ continuous input latency should be less than 10ms

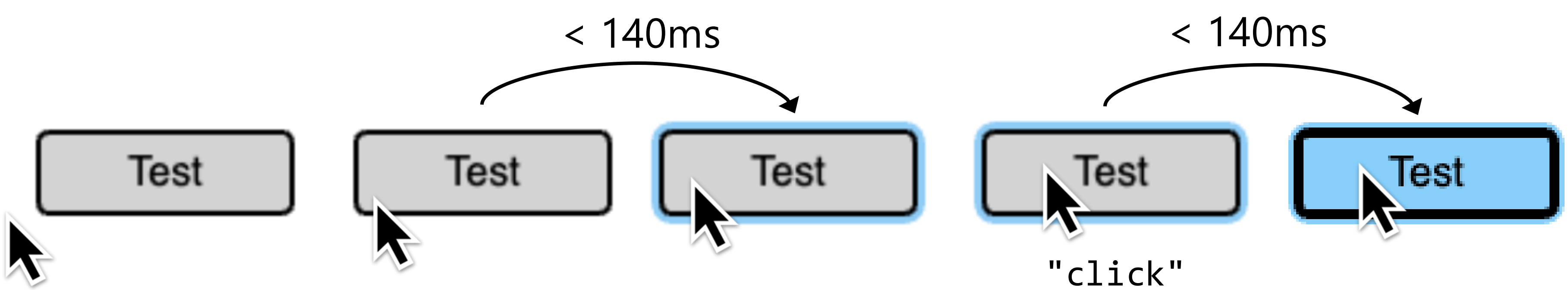
e.g. dragging a shape, how far behind the cursor is the shape

Input Feedback

Maximum interval between cause-effect events: 140 ms

→ input feedback should appear in less 140ms

e.g. time between pressing a button until the feedback changes



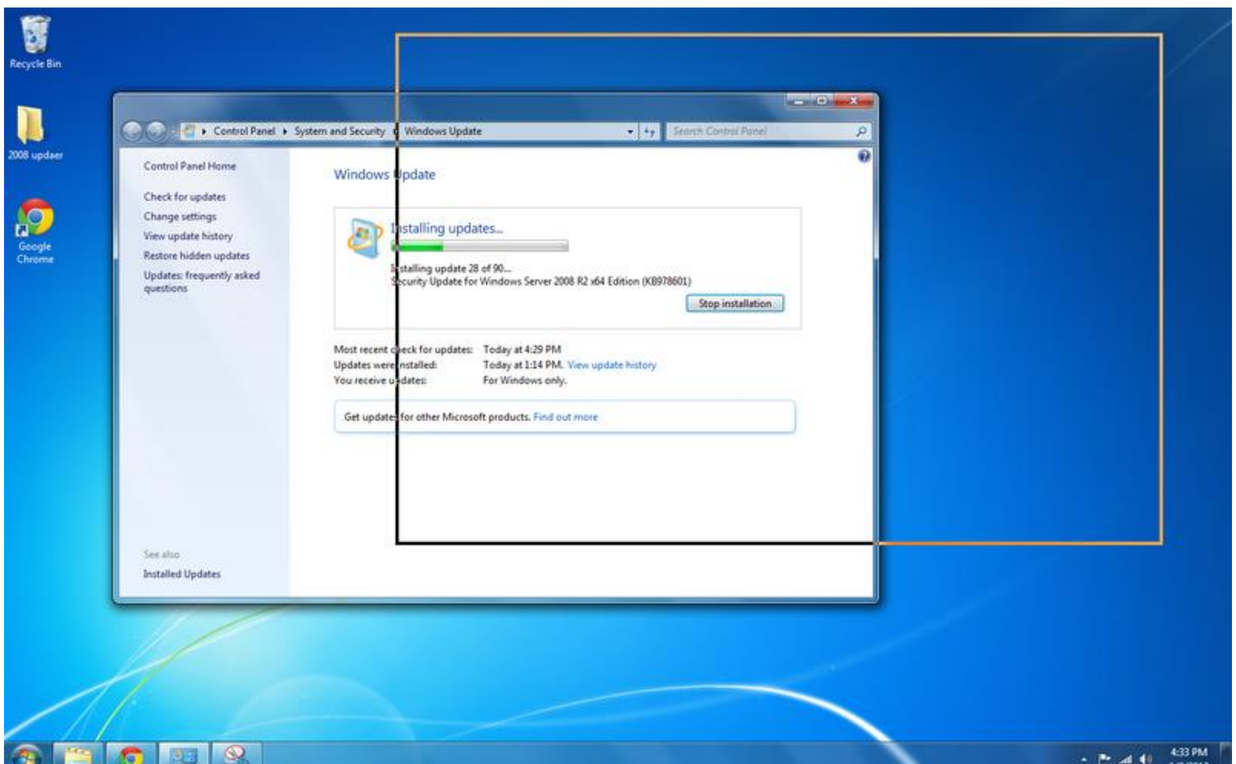
User Perception of Latency & Latency Improvements in Direct and Indirect Touch

- https://youtu.be/1dKlMZrM_sw

Graceful Degradation of Feedback

Simplify feedback for high-computation continuous input tasks

- *Examples*
  - window manager updates window rendering after drag
  - graphics editor only draws object outlines during manipulation
  - CAD package reduces render quality when panning or zooming



Busy Indicators

Visual-motor reaction time to an observed event: 1 s

→ Display *busy indicators* for operations that take 1s to about 3-4s

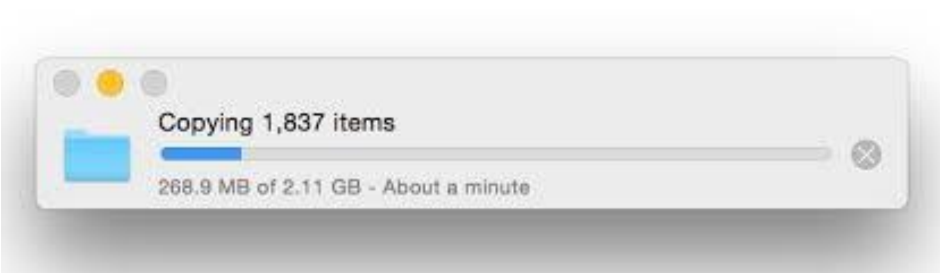
- Busy indicator design
  - Use visually cohesive cyclic animations
    (not repeating “progress” indicators)

Progress Bars

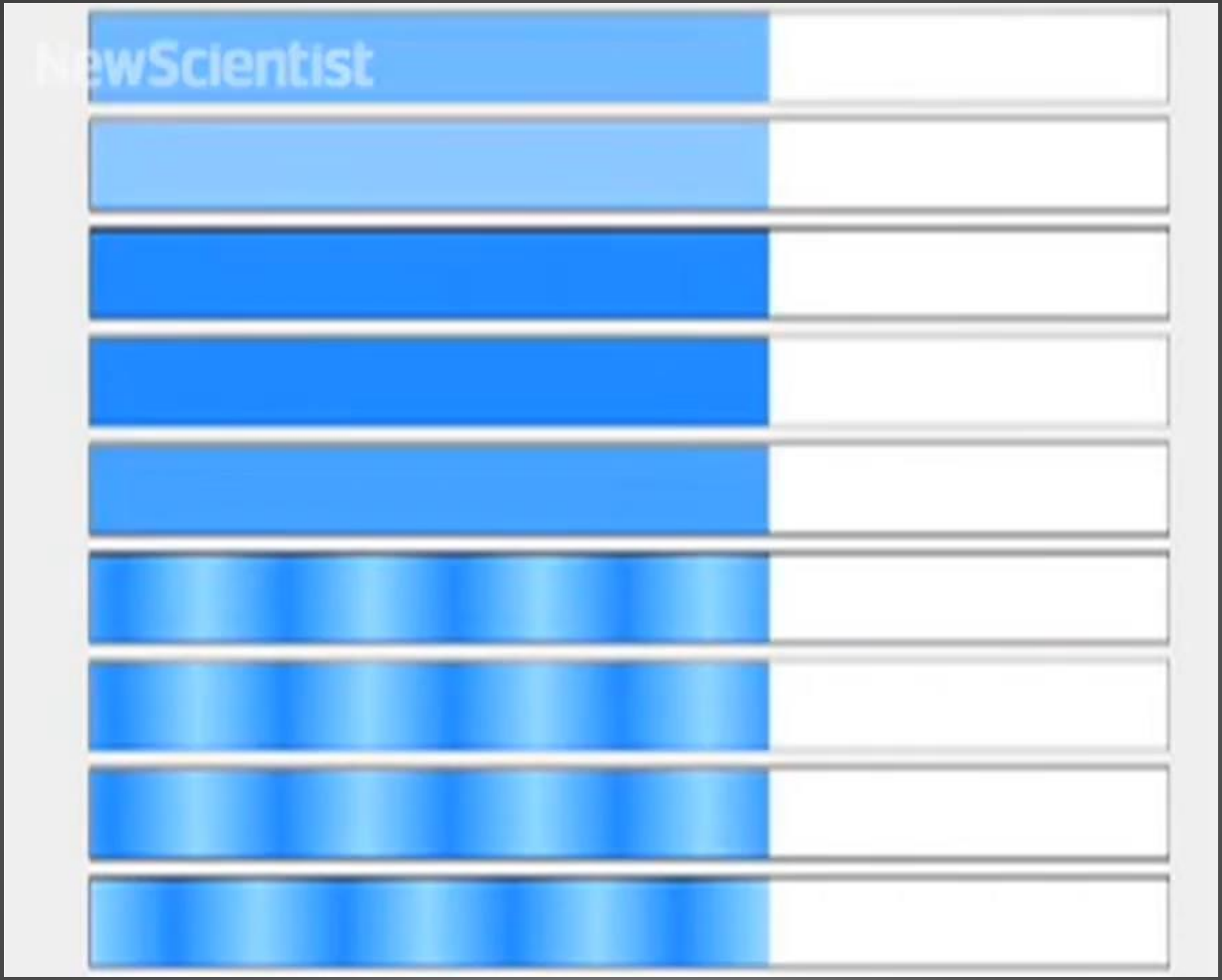
Visual-motor reaction time to an observed event: 1 s

→ Display *progress bars for* operations more than 3-4s

- Progress bar design
  - Show work remaining, not work completed
  - Use human precision, not computer precision
    (Bad: “243.5 seconds remaining”, Good: “about 4 minutes”)
  - Show smooth progress, not erratic bursts
  - Show total progress when multiple steps, not only step progress
  - Display finished state (e.g. 100%) very briefly at the end



(McInerney and Li, 2002)



Harrison et al. Faster Progress Bars (2010)

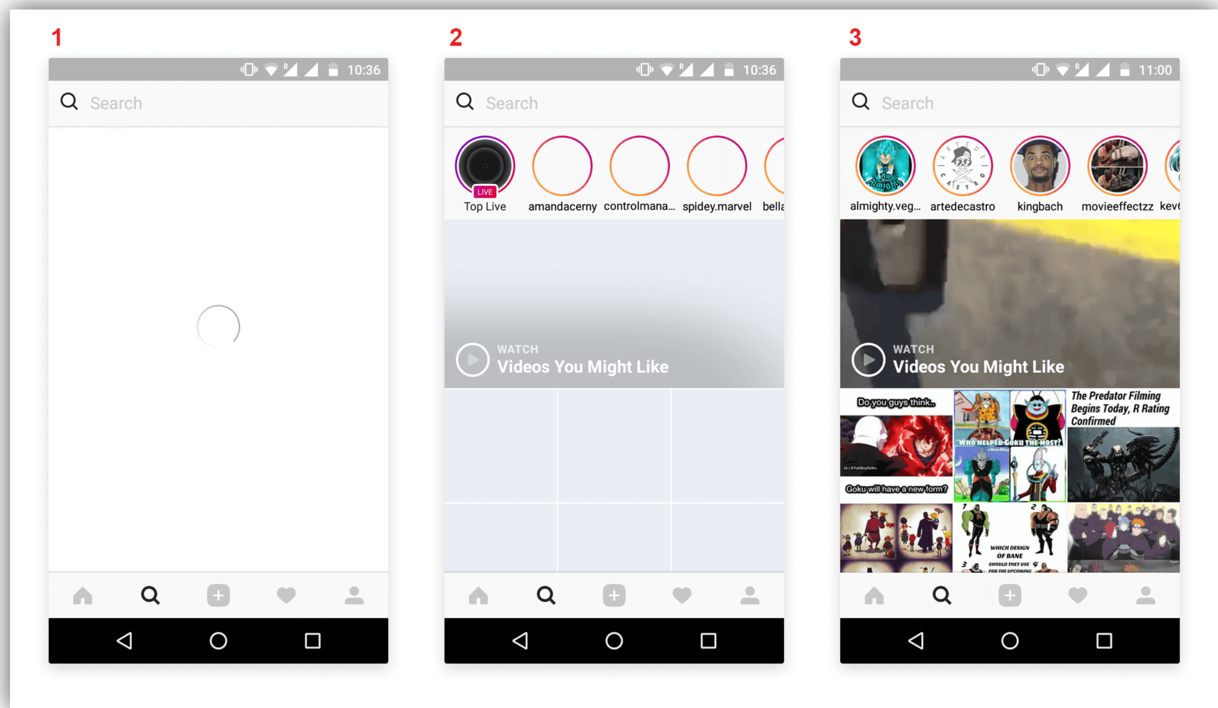
- https://www.newscientist.com/article/dn18754-visual-tricks-can-make-downloads-seem-quicker/

Progressive Loading

Visual-motor reaction time to an observed event: 1 s

→ Use *skeleton placeholders* when loading takes more than 1s

- Advantages:
  - User adjusts to a layout they'll eventually see
  - Loading process seems faster because there is an initial results



Instagram minimal version skeleton screen
https://blog.iamsuleiman.com/stop-using-loading-spinner-theres-something-better/

Progressive Loading

- Provide user with *some data* while loading rest of data
- *Examples*
  - word processor shows first page as soon as document opens
  - search function displays some items as soon as it finds them
  - webpage displays low resolution images, then higher resolution

Responsiveness by Predicting Next Operation

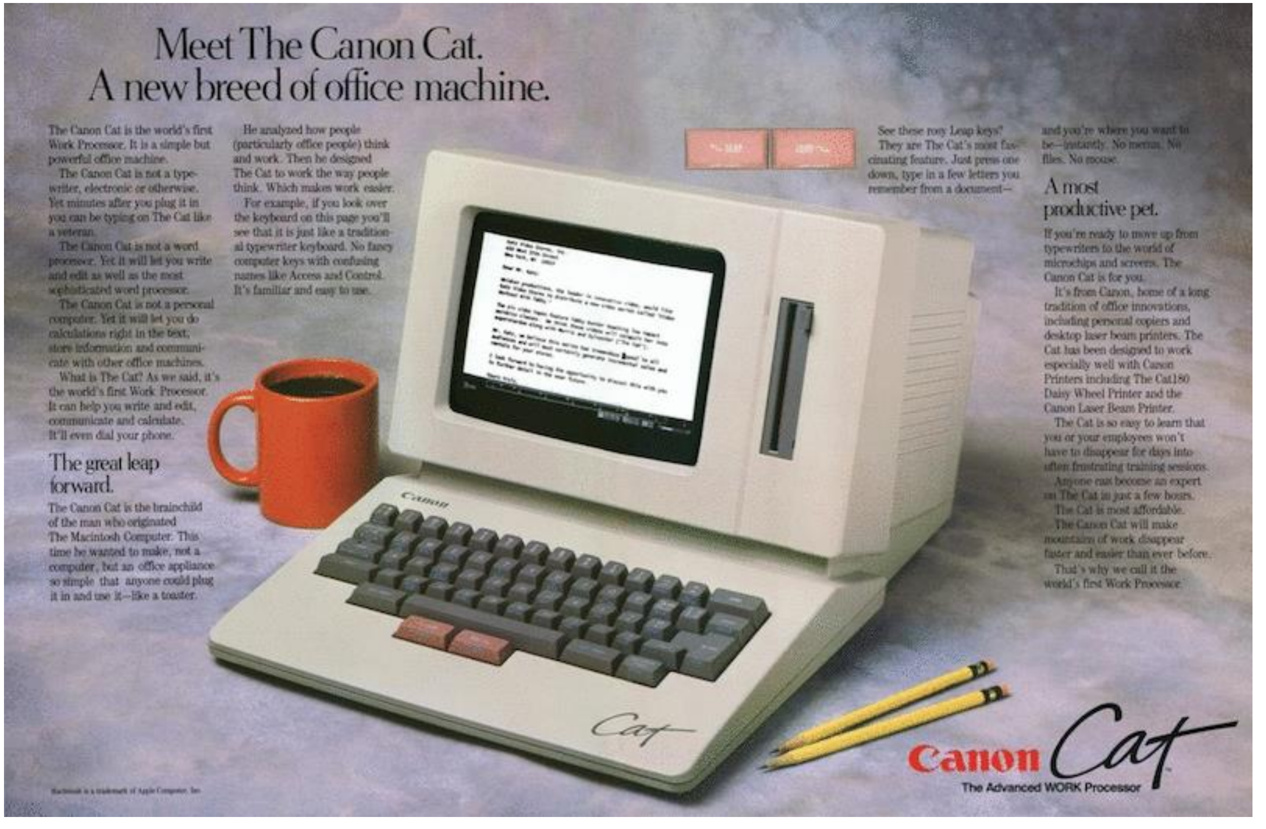
- Use periods of low load to pre-compute responses to high probability requests. Speeds up subsequent responses.
- *Examples*
  - text search function looks for next occurrence of the target word while user looks at the current
  - web browser “prerenders” pages that are likely to be visited next

https://developer.chrome.com/docs/web-platform/prerender-pages

Progressive Loading

Time to prepare for conscious cognition task: 10 s

→ Display image of document on last save,
while real one loads in less than 10s

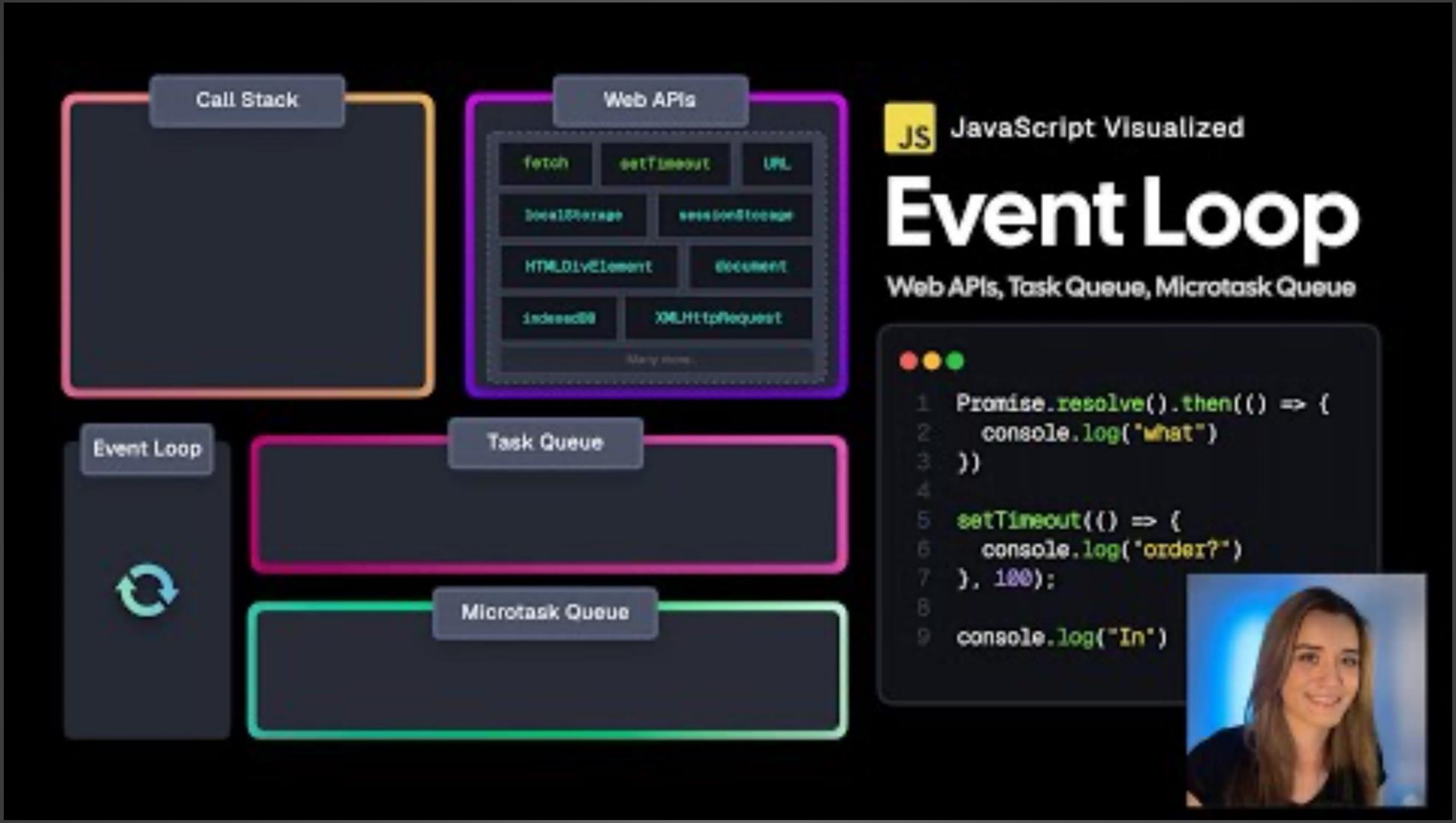


Asynchronous Execution

- Execute tasks independently from the main program flow
  - "Do more than one thing at once"
- Two main types of Asynchronous Execution
  1. **Asynchronous Programming**
  2. **Threading**

Asynchronous Programming

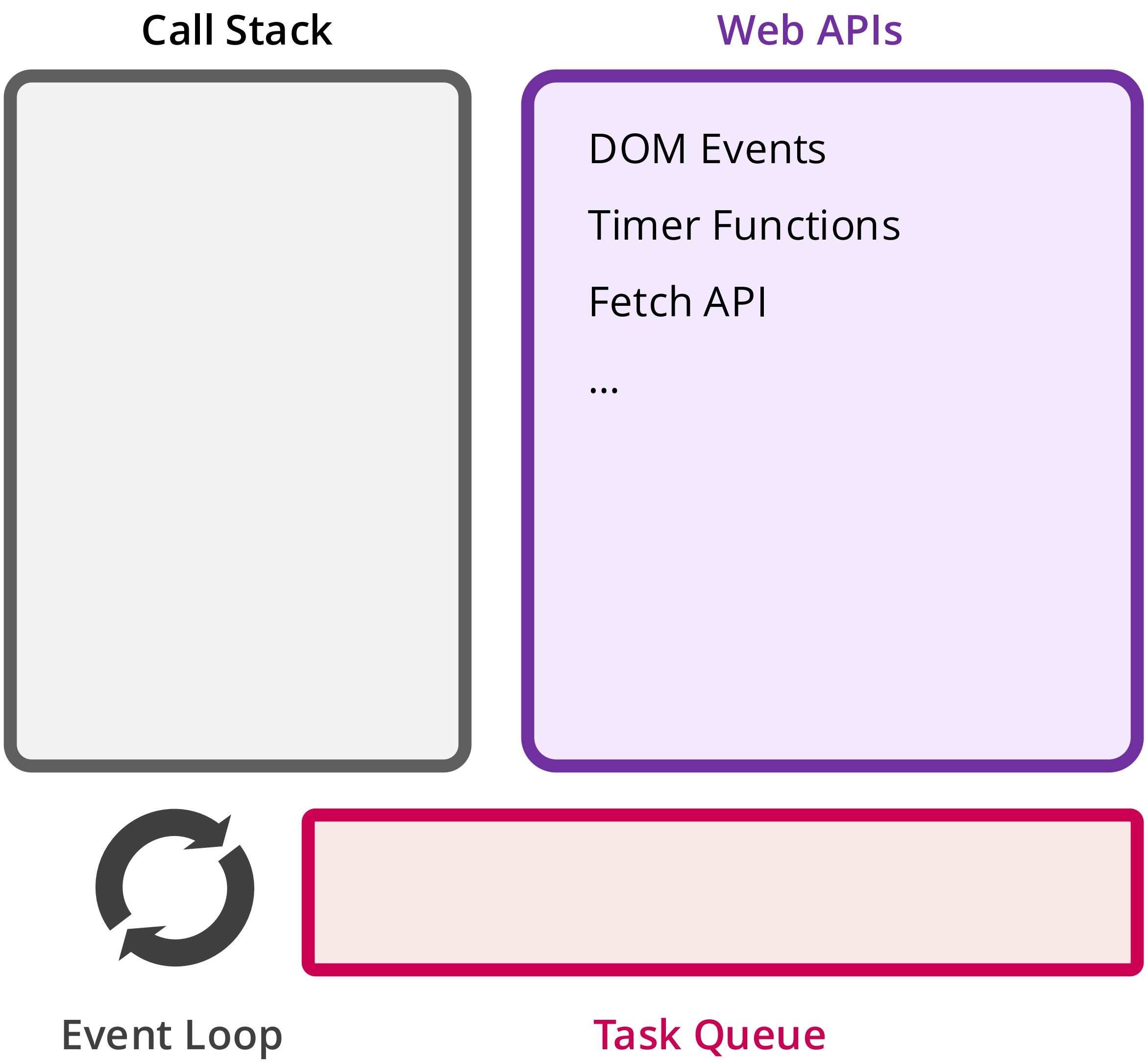
- A paradigm that allows for execution of tasks in a non-blocking manner in a **single thread**
  - NOT concurrent execution
- Related concepts in JavaScript and other Languages
  - Event driven programming
  - Promise/Future
  - Coroutines
  - Non-blocking I/O



JavaScript Visualized - Event Loop, Web APIs, (Micro)task Queue, **Lydia Hallie**

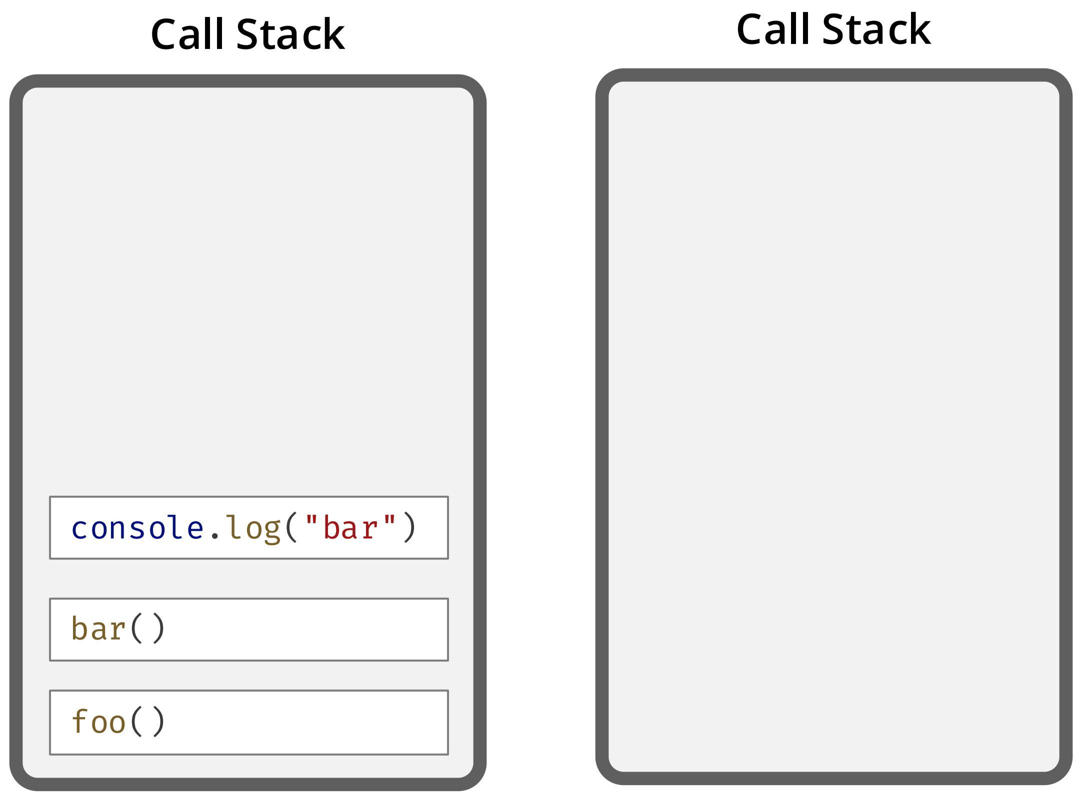
- https://www.youtube.com/watch?v=eiC58R16hb8

(Simplified) JavaScript Runtime Environment



Call Stack

```
1   console.log("start");
2
3   function bar() {
4     console.log("bar");
5   }
6
7   function foo() {
8     bar();
9     console.log("foo");
10  }
11
12  foo();
13
14  console.log("end");
15
```

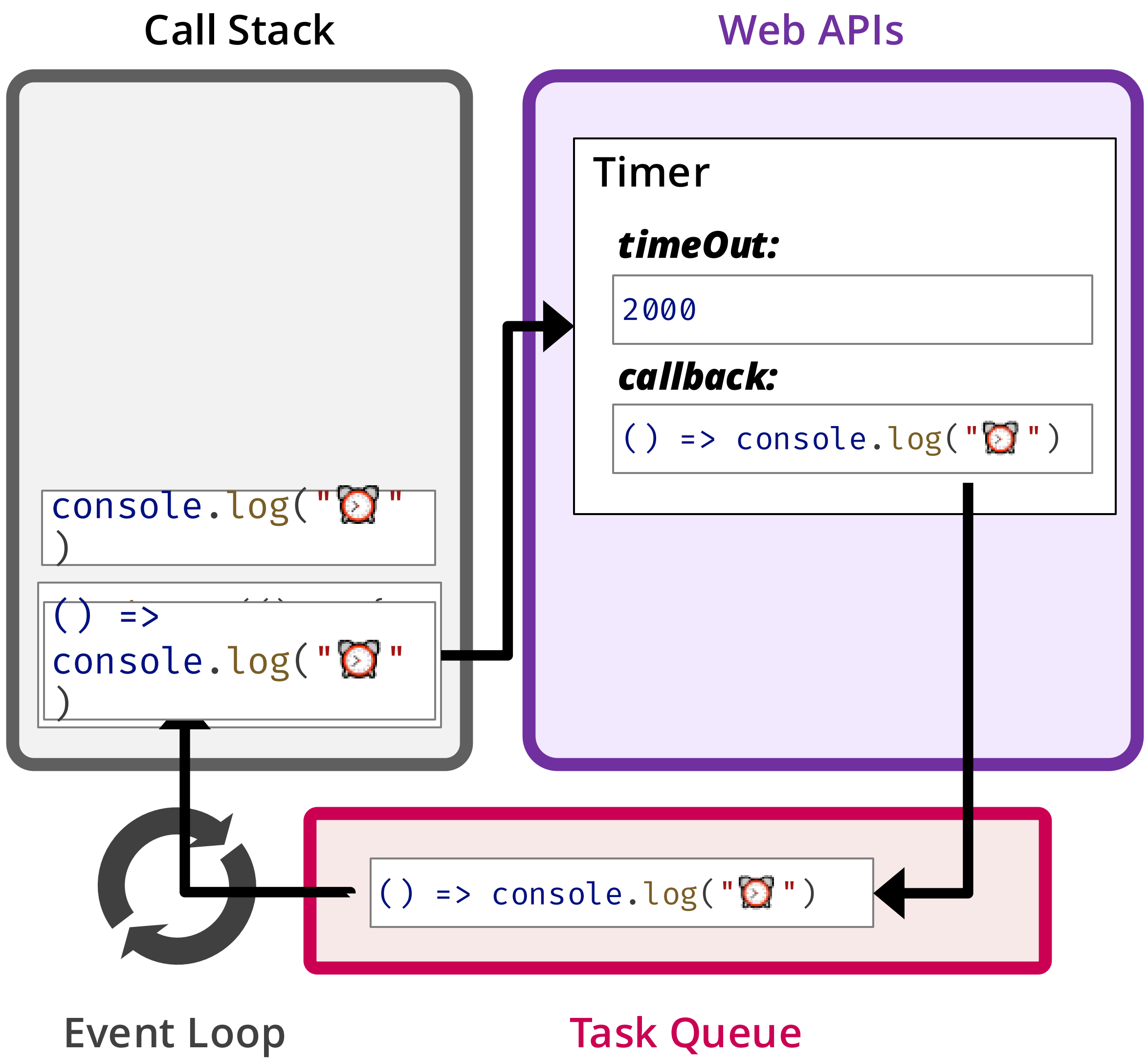


at line 12

at line 15

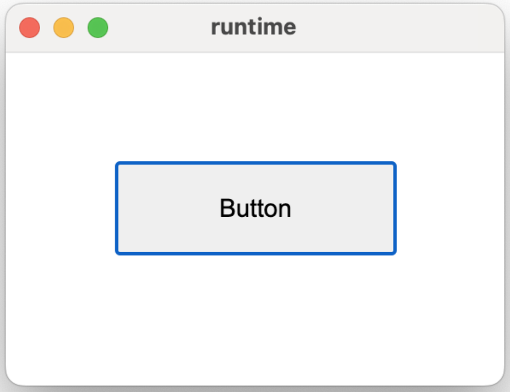
Web APIs, Task Queue, Event Loop

```
console.log("start");

setTimeout(() => {
  console.log("⏰");
}, 2000);

function bar() {
  console.log("bar");
}

function foo() {
  bar();
  console.log("foo");
}

foo();

console.log("end");

```



runtime

- Walkthrough for runtime environment
- Demos
  1. What if timer is 0ms?
  2. Uncomment long() in main
  3. Uncomment long() in button callback



Callbacks

- **Input events** are **asynchronous methods**
  - We handle them as callbacks bound to a DOM element

```
button.addEventListener("click", () => {
  console.log("💥 button");
  // do something


});
```

```
asyncOperation1(function(result1) {
  asyncOperation2(result1, function(result2) {
    asyncOperation3(result2, function(result3) {
      // More nested callbacks ...
    });
  });
});
};
```

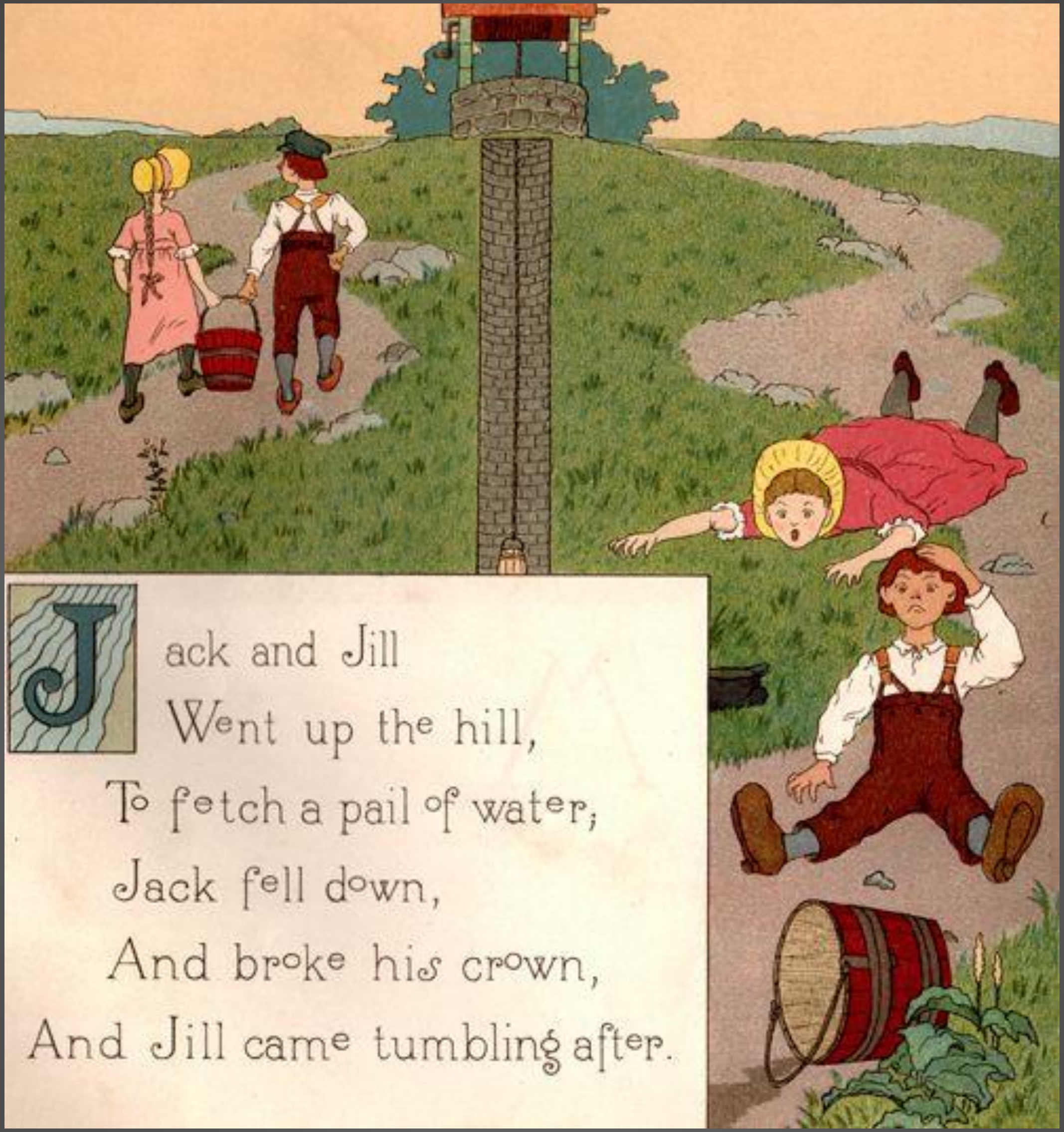
Callback Hell

- https://medium.com/@raihan_tazdid/callback-hell-in-javascript-all-you-need-to-know-296f7f5d3c1

Fetch API

- An interface for fetching resources across the network
- `fetch()` function
  - starts the process of fetching a resource from the network
- Returns a "Promise" object with three states:
  - *Pending*, when fetch process is happening
  - *Resolved*, when the process was successful and there's a valid response
  - *Rejected*, when the process failed and there's an error

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

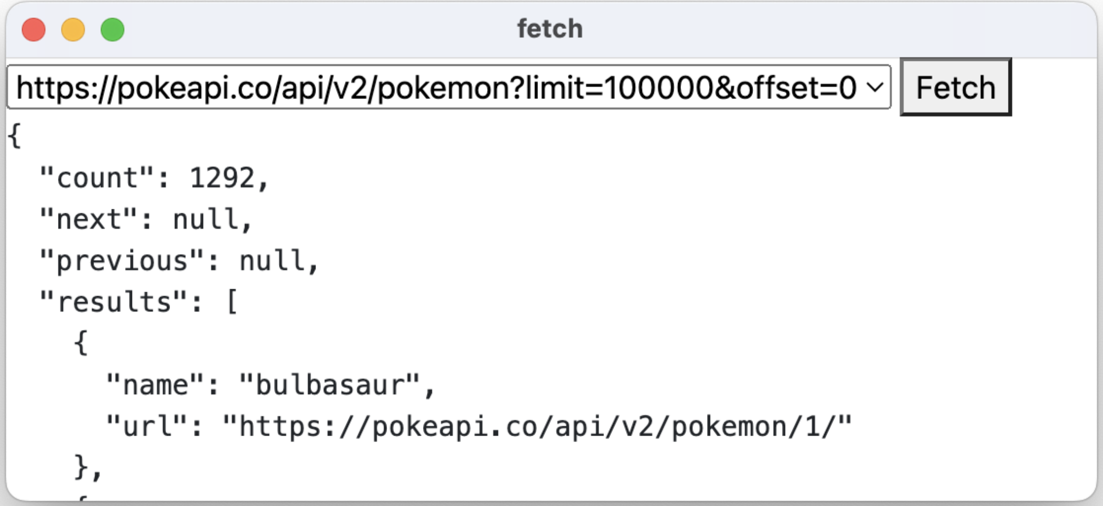


Jack and Jill Nursery Rhyme Analogy of Promises

- Inspired by https://blog.greenroots.info/javascript-promises-explain-like-i-am-five

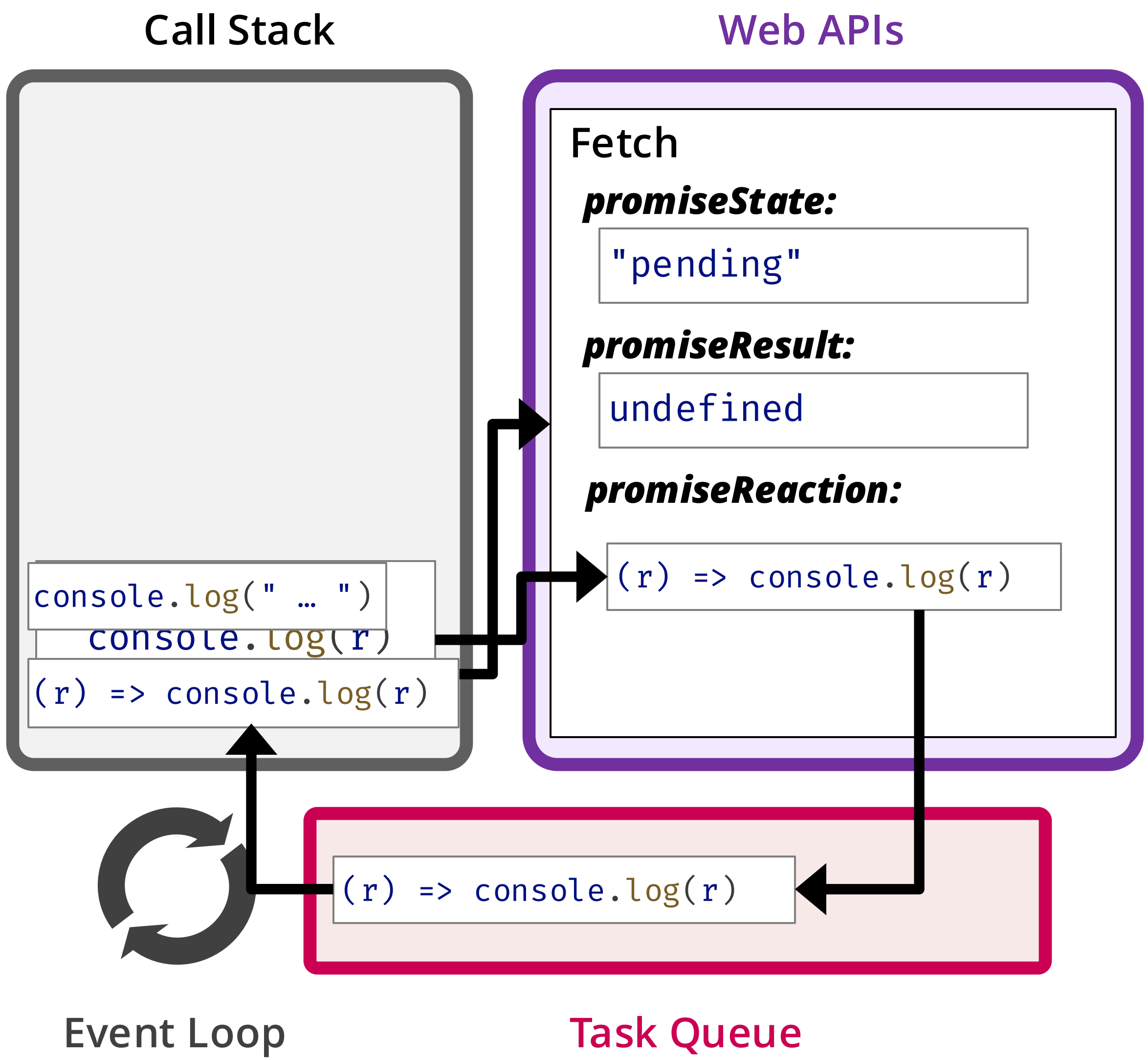
fetch

- Demos:
  - async function
  - Network throttling to simulate slow connection
- doFetch1() using Promises
  - fetch() with chained .then(...)
  - error handling
- doFetch2() with async/await



JavaScript Runtime with Fetch API

```
fetch(url)
  .then((r) => {
    console.log(r);
  })
  .catch((e) => {
    console.error(e);
  });
```



Fetch Progress

- Surprisingly complex to get progress during fetch
  - Use `ReadableStream`, read in chunks, …
- Link below shows an approach

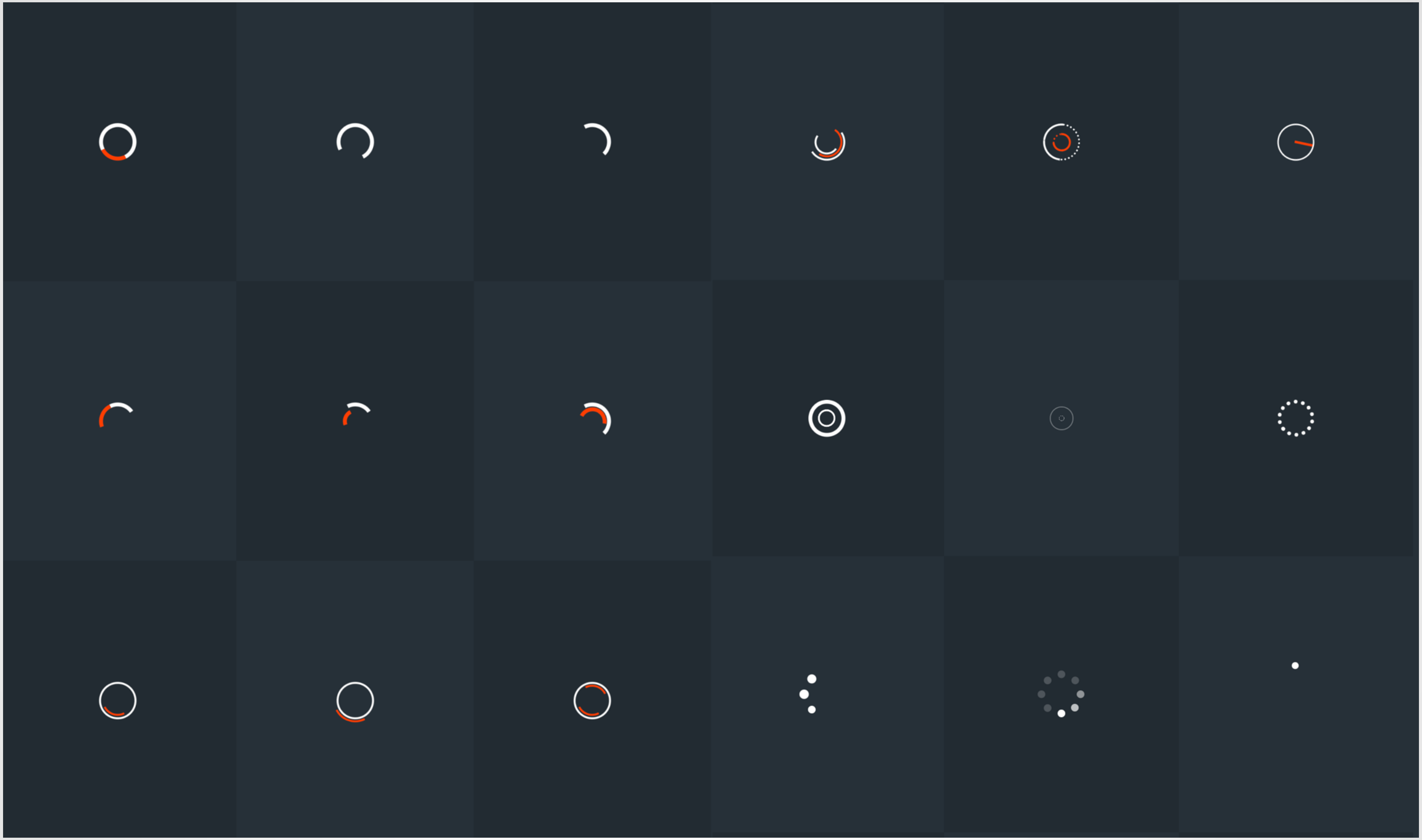
https://dev.to/tqbit/how-to-monitor-the-progress-of-a-javascript-fetch-request-and-cancel-it-on-demand-107f

fetch (with loader)

- A simple CSS class for a loader animation
- HTML

```
<div class="loader"></div>
```

- CSS rule
  - Uses CSS variables
  - Rounded corners to create circle
  - CSS animation



CSS Loaders & Spinners

- https://github.com/vineethtrv/css-loader

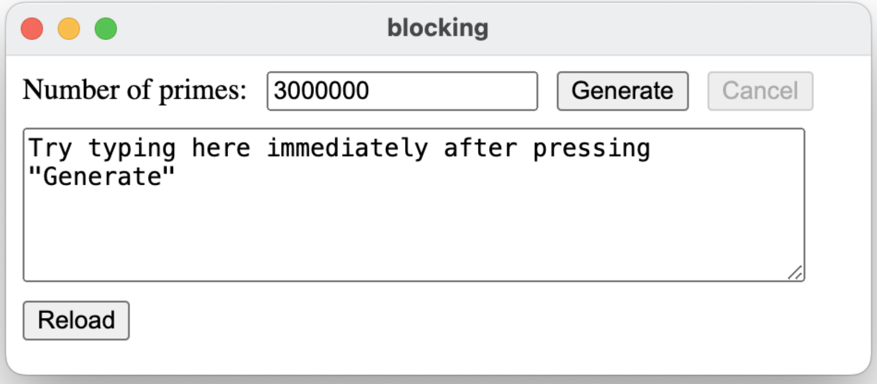
Handling (non-Web API) Long Tasks

- Goals
  - keep UI responsive
  - provide progress feedback
  - (ideally) allow long task to be paused or canceled

blocking

- Shows what happens when long tasks NOT handled asynchronously
  - DO NOT DO THIS!
- Demo
  - prime number generation code (intentionally inefficient)
  - dispatch blocked (try typing in textarea)
  - Cancel button unusable
  - Note even DOM update is blocked

```
output.textContent = "Starting ...";
```



Threading

- Manage multiple concurrent threads with shared resources, but executing different instructions
- Threads are a way to divide computation, reduce blocking
- Concurrency risks: e.g. two threads update a variable
- Browsers support **worker threads**
  - dedicated workers
  - shared workers
  - service workers

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing_workers

worker

- Uses web worker
  - create a dedicated worker
  - `generate.ts` has code for thread to execute
- `Worker.postMessage( ... )` to send message
- `Worker.addEventListener( ... )` to receive messages
- Messages from worker to main
  - main to thread: start
    `["generate", 100000]`
  - thread to main: progress
    `["progress", 0.5]`
  - thread to main: done
    `["done", 100000]`
- HTML progress bar