The Javascript Execution Model

Kevin Charles, 2020

github.com/kevinchar93/js_exec_model_presentation

Note some information in this talk / script has been removed for it to be used outside my workplace

- So today i'm going to give a hopefully short talk on how javascript gets executed

* so in this talk i want to give a brief high level overview of JavaScript & the way it gets executed

* since it's quite different from the other c-style languages we use

* and hopefully sharing this kind of high level knowledge, should make it easier for others to get where our designs for [redacted] are coming from

Clarification - DOM

- Document Object Model
- API for an HTML page
- Synonymous for UI in web development

* so i want to start out making a clarification

* if you hear me saying the word DOM

* it stands for document object model, which is kind of like an api for an html page

* javascript can use it to update the webpage

* and it tends to be synonymous for UI in web development

* so if you have me say the word DOM this is what i mean

Javascript a Description

- Programming language
- Event driven
- Asynchronous
- Nonblocking
- Typically used in the browser


```
right another definition

javascript is an
    * event driven
    * asynchronous
    * non-blocking
    * programming language that you typically run in the browser
```

Javascript a Description

- Event driven - Lots of responding to events
- Asynchronous - keep running while waiting for other code
- Nonblocking - Long running code does not block

* what do i mean by all of this

    * by event driven - i mean a lot of javascript programming is about responding to events

    * by asynchronous - i mean a script can continue running while waiting for some other code to finish

    * and by non-blocking - i mean long running code does not block other code from running

* it's easier to show how all these features work with a couple of examples

setTimeout example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


* so here we have an html file

* it has a single h1 tag

    * this html file then imports a javascript file

    * the contents of which you can see on the right

    * for this to run

    * the browser parses the html

    * then the script file is then JIT compiled (into machine code) & executed from top to bottom

* right, now….. i'll explain what the javascript attached to this page actually does

setTimeout example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


* we define a function called demo

setTimeout example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


* in that function we start out by looking up the h1 tag using its id
* and we change its text to say "Hello World"

setTimeout example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


* next we log "output 1" to the console

setTimeout example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


* then we create a function and store it in a variable (without calling it)
    * this function is going to log "output 2" to the console

setTimeout example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


* next we call setTimeout , which will call a function we provide after a delay we can set

* here the delay is 5000 ms, and then it will call the function timeoutCallback

setTimeout example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


- We log "output 3" to the console

setTimeout example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

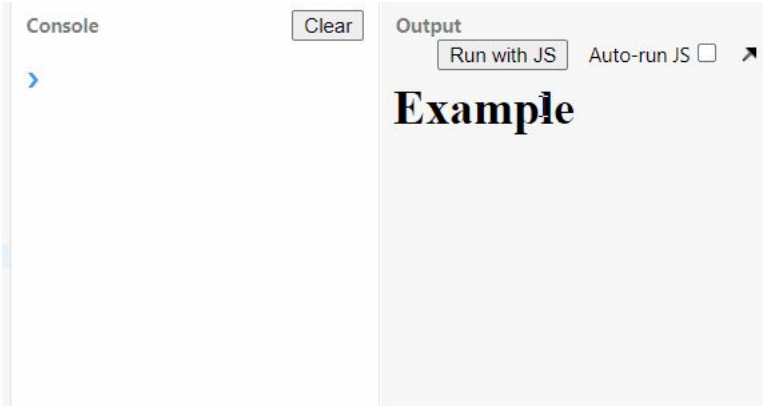
JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


* finally at the bottom we call the function demo, to actually run it

* let's see what the output of this script is

setTimeout example



>>> show a video of the script running

* so you can see the text is changed to hello world

* we log output 1, output 3, then after five seconds output 2

setTimeout example 2

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

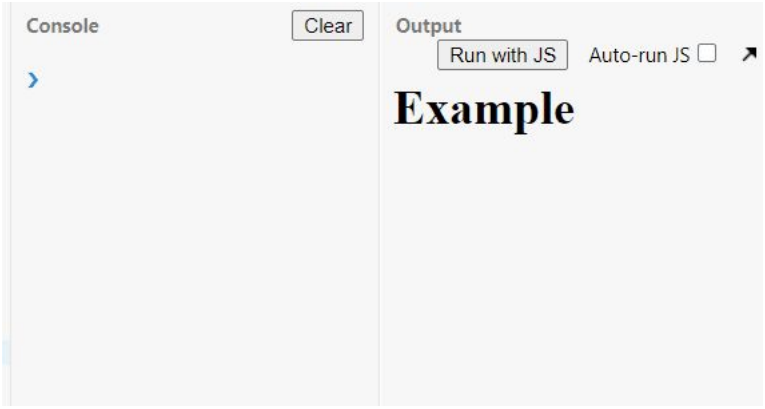
```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 0);
  console.log(" output 3 ");
}

demo();
```


* now let's make a slight modification to this script, and run it again

* i've changed the timeout to zero , so you would think the callback should be run immediately

* let's see what the output of this script is

setTimeout example 2




```
>>> show a video of the script running

* the output is exactly the same as before, it just happens faster
since there's no five second delay
```

A couple of questions

- Where is this running?
- How do we get pixels?
- setTimeout behaviour?

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <h1 id="example1">Example</h1>
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```


>>> go back to showing script

* so here's a couple of questions:
    * where does this code actually run?
    * how does the browser translate what we've done into pixels on screen?
    * and
    * why does settimeout behave like this?
* this is where the main thread & the event loop come in
* i'll come back to this example later on

Javascript a Hosted Language

- Sandbox environment provided by browser
- Environment provides APIs
- Environment is single threaded (main thread)

* you can view javascript as hosted language in a sandbox environment provided by the browser

    * the environment gives you apis to interact with the page & the browser (think things like location, clipboard access, desktop notifications)

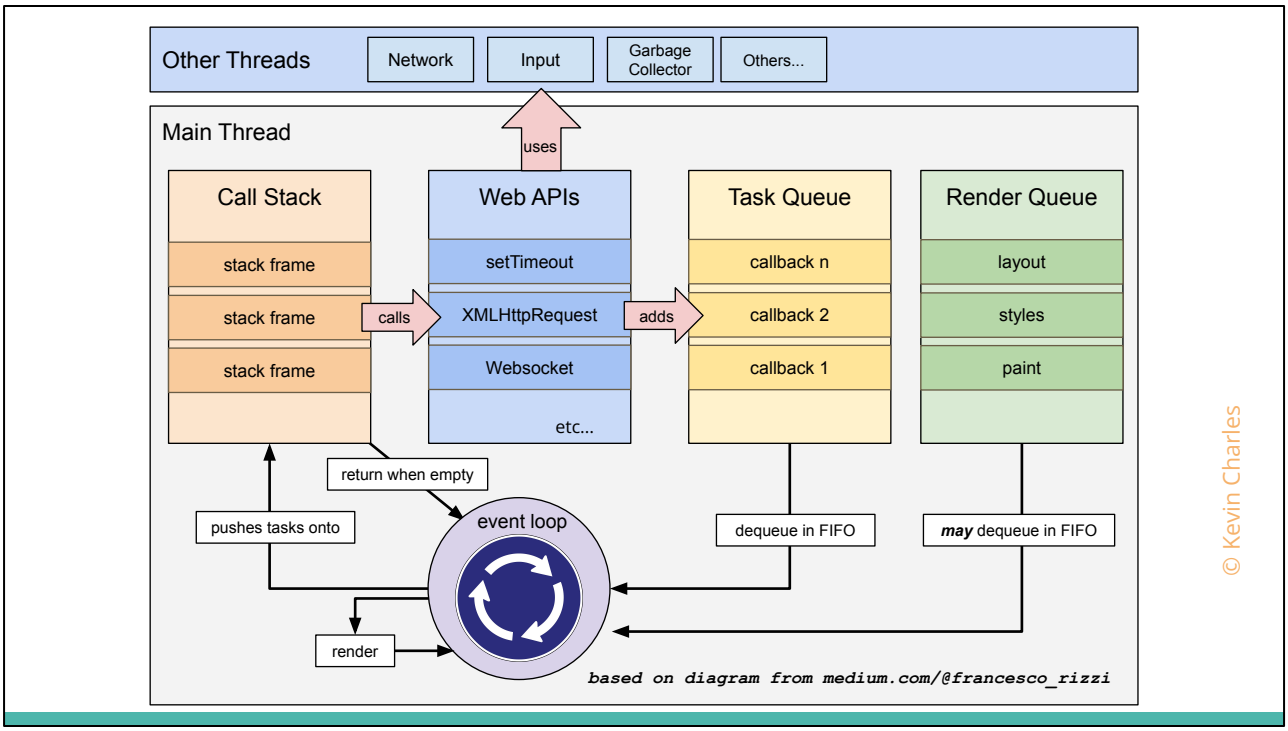
    * and this environment is single threaded (so it's thread is appropriately called main thread)

Main Thread & Event Loop

- Main thread responsible
  - Running JavaScript
  - Interpreting HTML/CSS
  - Page rendering
  - Possibly more
- Event loop managing this
  - Abstracted from javascript
  - Different from other languages - explicit thread control
    - UI frameworks do something similar

* this main thread is responsible for lots of things which include
    * running your javascript code
    * interpreting your html and css,
    * rendering the webpage
    * there may be more
* there is an event loop running in the main thread managing all of this
    * but this is abstracted away from your javascript code
* this is quite different from other languages where you would have explicit control over the creation and management of your threads
    * that said you typically see something similar in other languages when you are using a ui framework

* now that i've said all of this let's look at a visualisation of the event loop



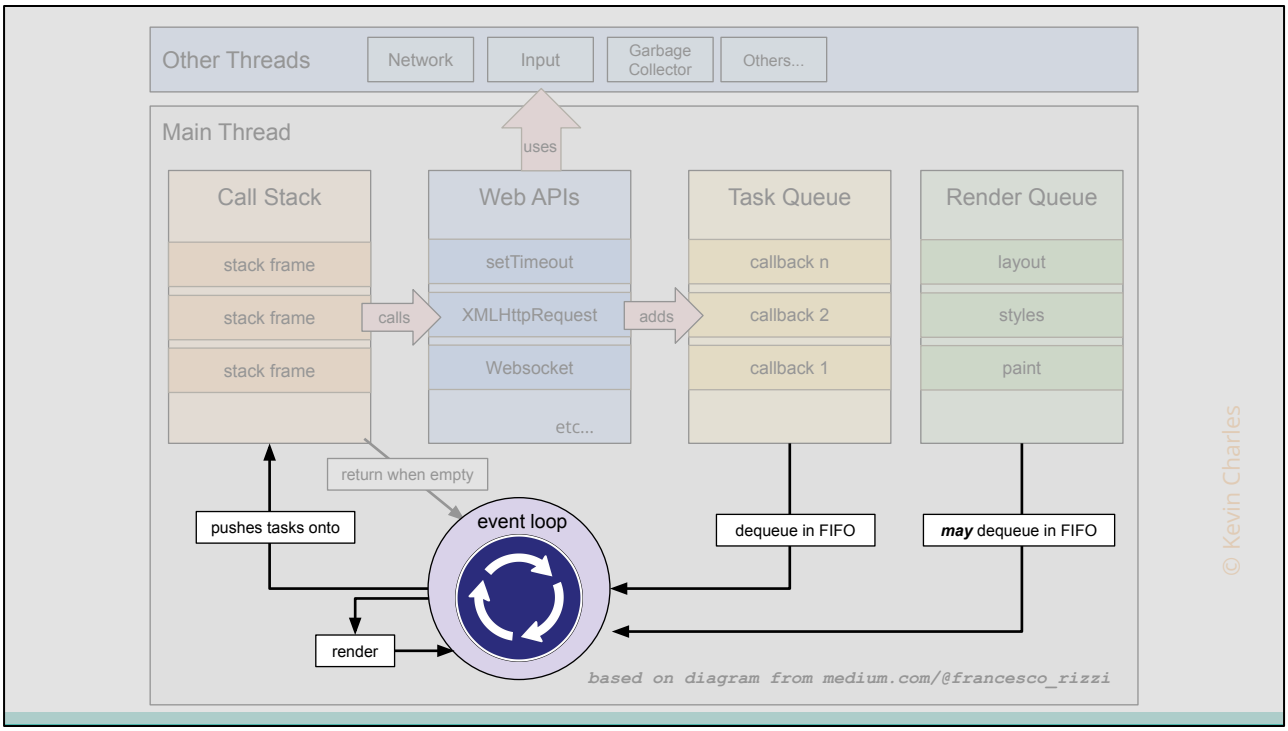
>>> based on diagram form medium.com/@francesco_rizzi

* this is a representation of what's going on in the main thread at a simplified level

* you could go into a lot more detail than this… … but this representation is good for what i'm trying to explain

* this diagram is kind of like a flowchart slash representation of how things in the main thread relate to one another

* there's quite a few parts here, so i'm going to step through this bit by bit



* to start off with we have the event loop which drives all this

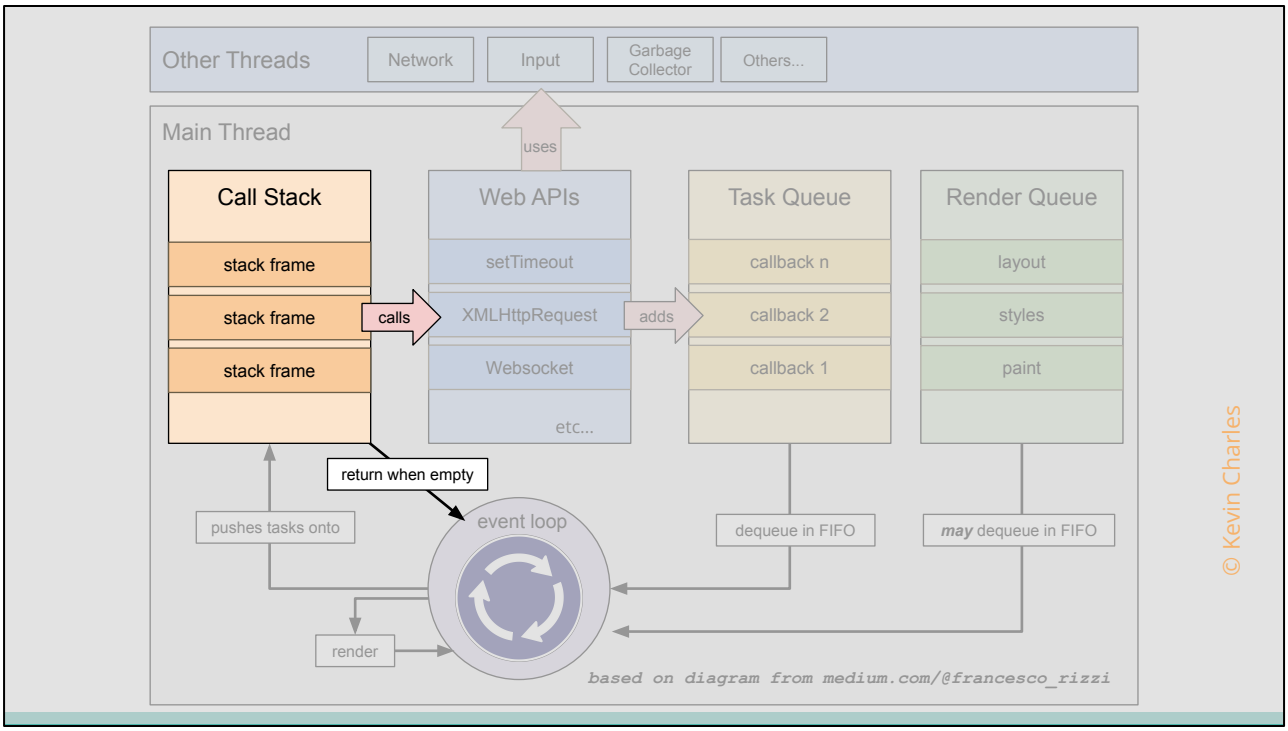
    * as the name suggests this is just a simple loop that can decide to do one of two things
        * run your javascript code
        * or
        * render the webpage

    * and as you can see on the diagram it reads from either the task queue or render queue

    * The precise algorithm for when it decides to render depends on number of factors (it basically will not render if it does not need to)

    * In fact you could have thousands of executions of your JavaScript code between rendering periods

    * When it does begin running JavaScript it takes a callback from the task queue and puts it onto the stack which executes it

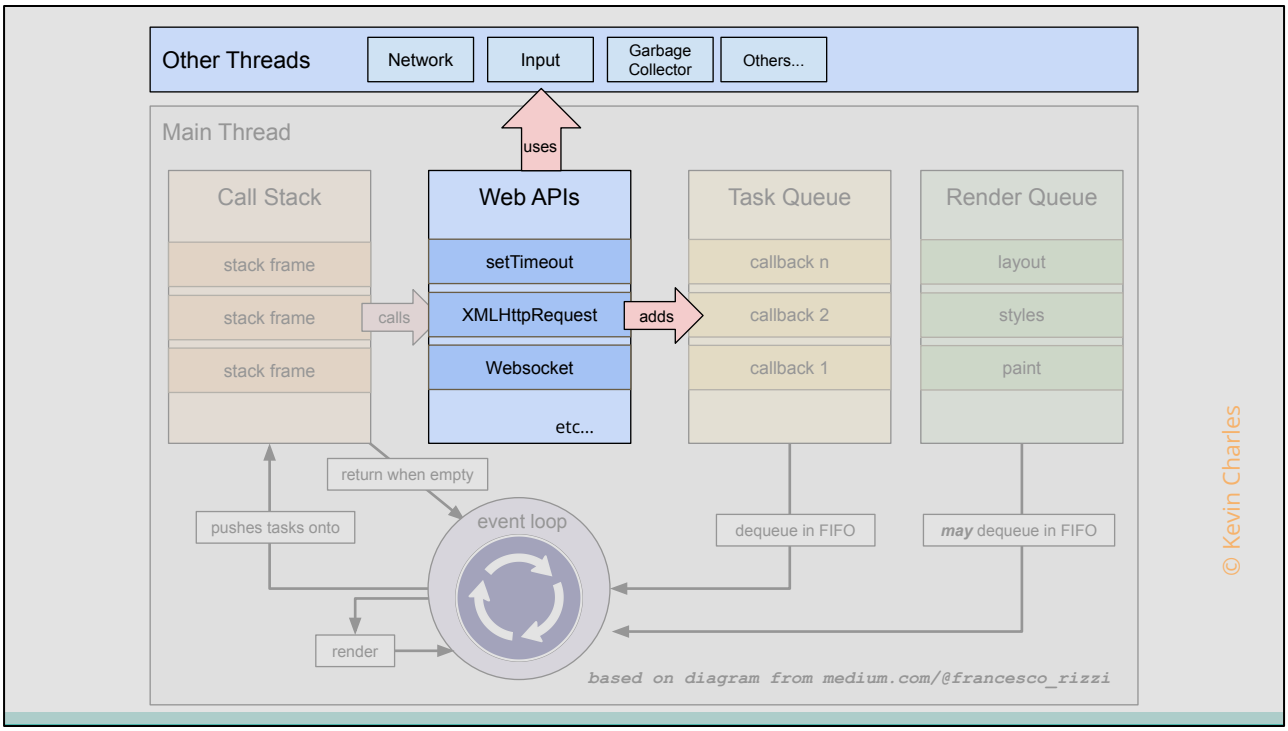


* next we have the call stack

    * This is where JavaScript code will be executed in a synchronous manner
    * and as your functions call one another the stack will grow and shrink to keep track of the running javascript program

    * javascript uses something called run to completion scheduling
    * which means any code in the call stack will keep running until the stack is empty
    * then control will be passed back to the event loop

    * so if you've ever had a webpage freeze on you this is how it happens
    * a long running bit of javascript code is probably preventing the event loop from rendering until the call stack is empty
    * and it’s not just rendering that blocked, you can’t respond to any new input until the current call stack is emptied



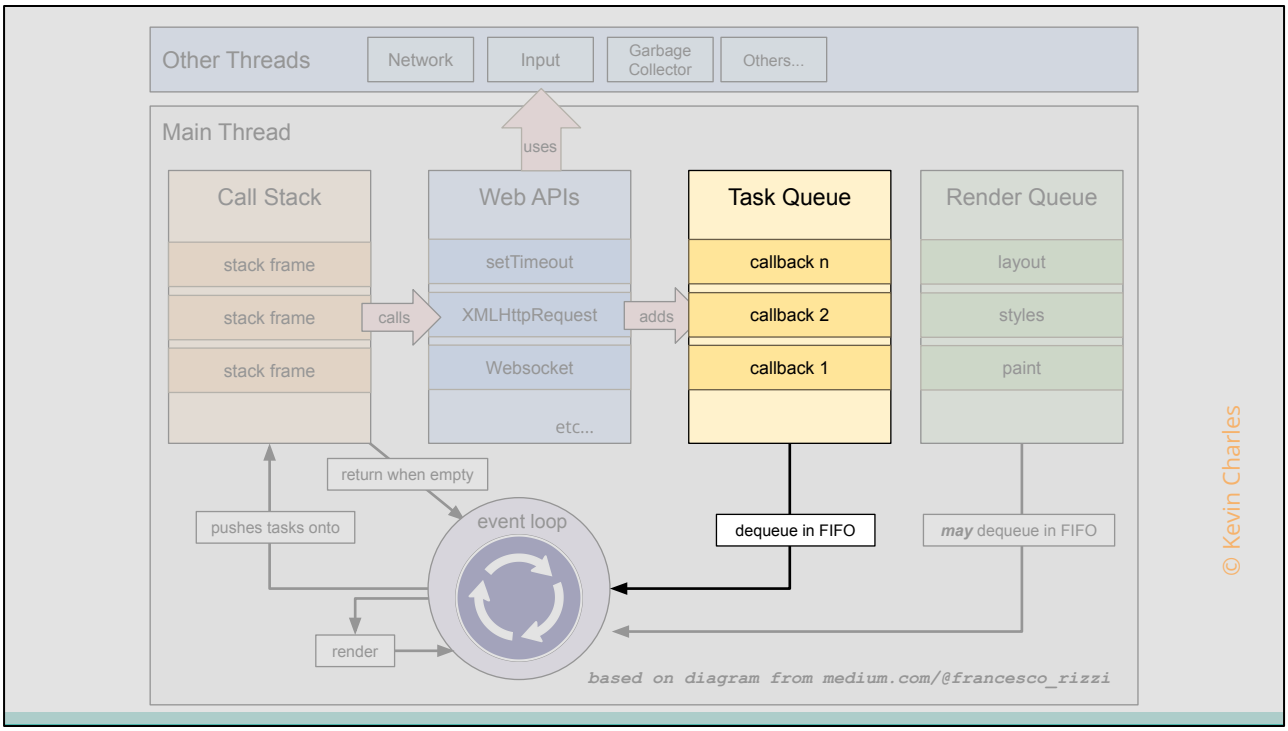
* the web apis are where things get interesting

    * pretty much almost every web API is asynchronous
    * so when your javascript code calls a web API you give it a callback and it returns immediately

    * behind-the-scenes another thread is used by that web api, to do whatever was requested
        * so as you can see at the top, there are threads that handle network, input and a whole load of other things

    * when that other thread is complete, it can't run a callback immediately - as this would interrupt whatever is in the call stack

    * so it queues a task for the event loop to run the callback it was provided when the event loop gets round to it

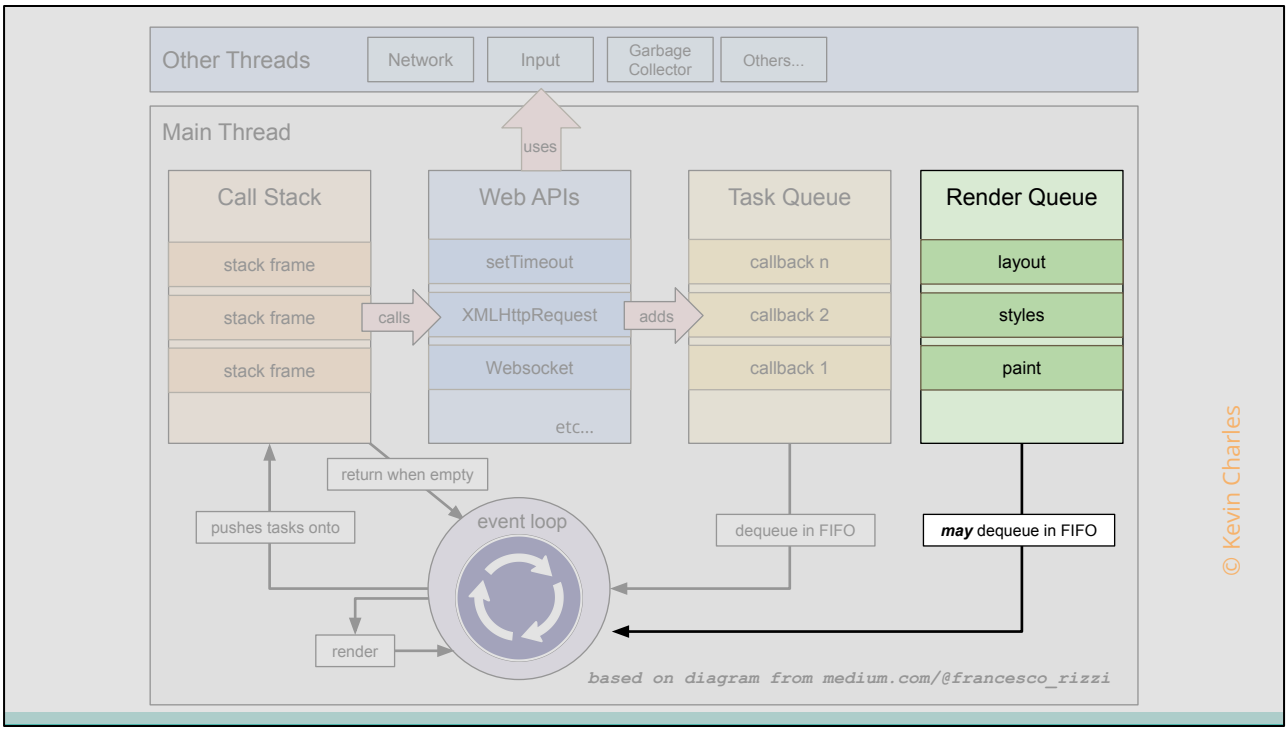


* up next is the task queue

    * as i said web, apis will queue tasks to run callbacks here

    * this is how any asynchronous code you encounter in javascript will invoke a callback (...by queueing a task)

    * and callbacks in the task queue are always run in the order they are put into the queue



* and then we have the render queue

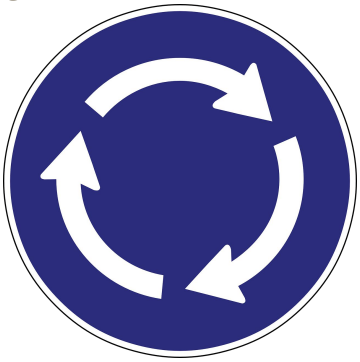
    * i don't know too much about the render queue

    * other than the browser has an algorithm to determine when it needs to render

    * and typically won't render more times than screen can refresh (typically 60htz)

Event Loop

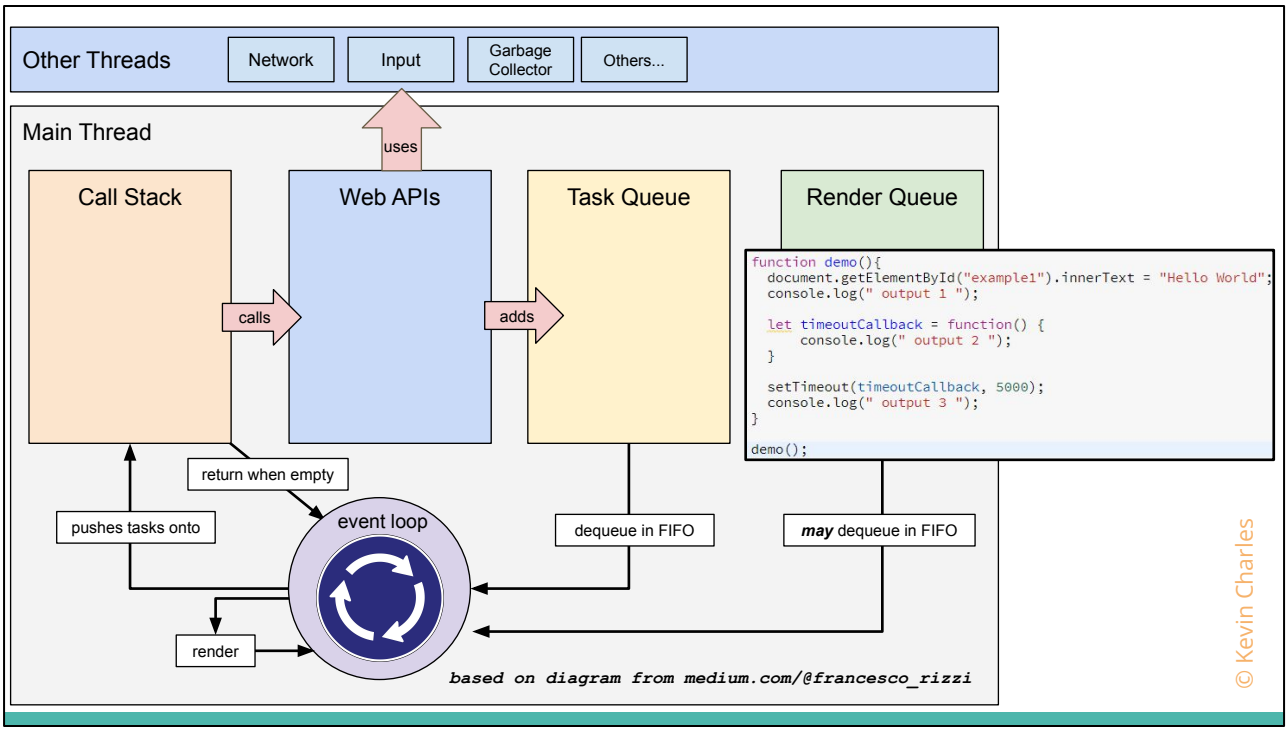
- Event loop either running javascript or rendering
- Dispatches events to code



* so with all these pieces you can see it's a cycle with the event loop at the centre
    * either executing javascript or rendering

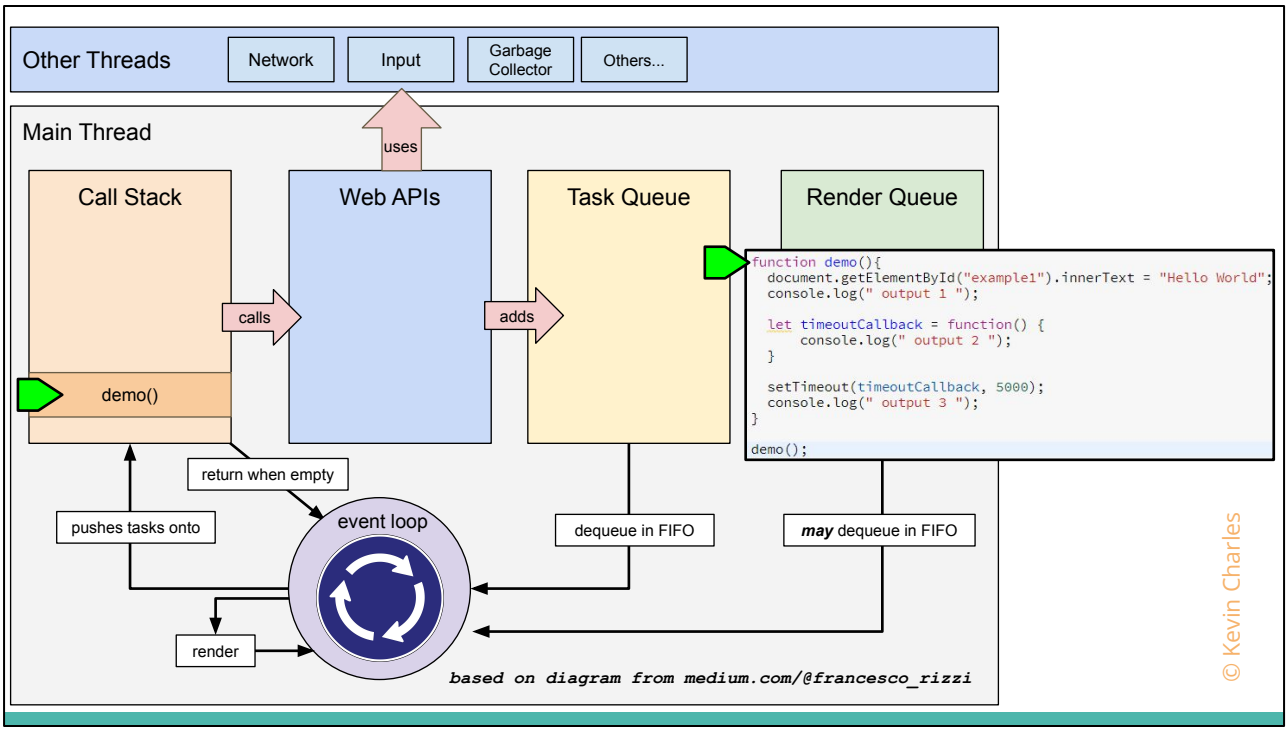
- It's called the event loop because it dispatches events to the code that handles those events

* now i'm going to go back to that example we had earlier and see how it fits into this diagram



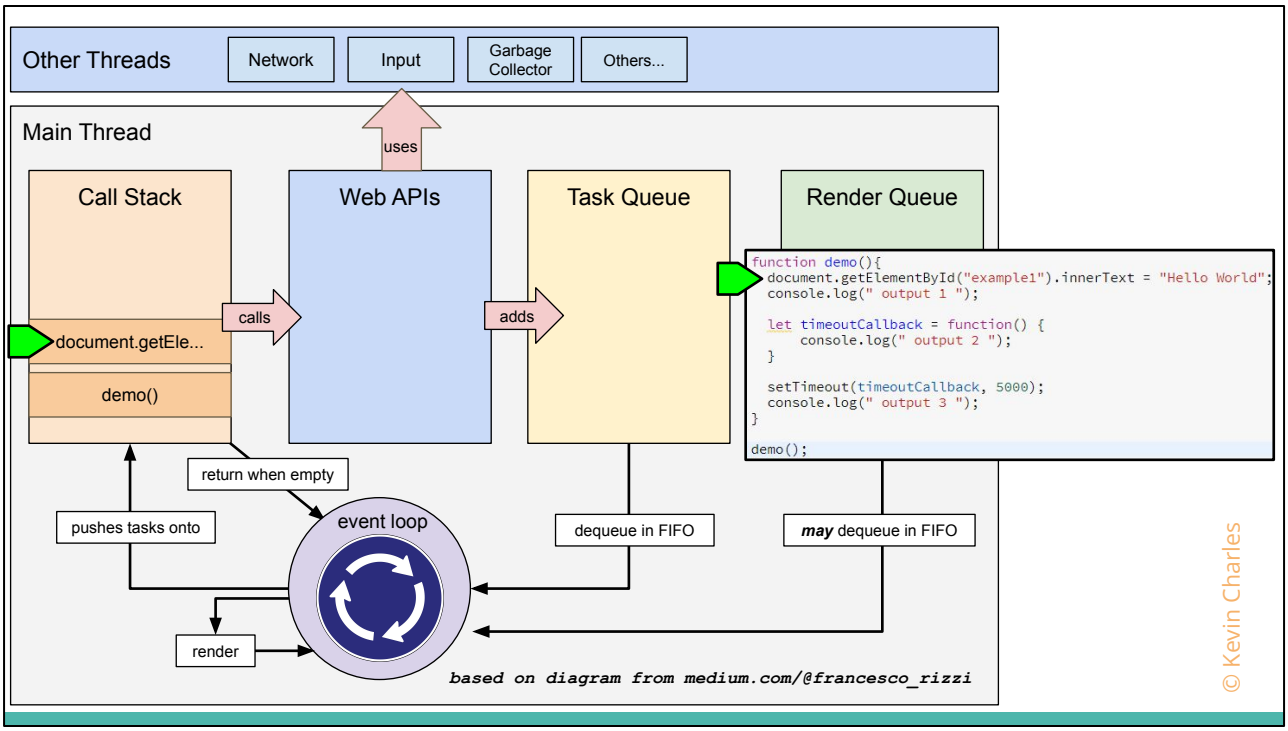
* now we can see the diagram and the javascript code

* i'm not going to show the html or the render queue to save space, so just imagine it's there

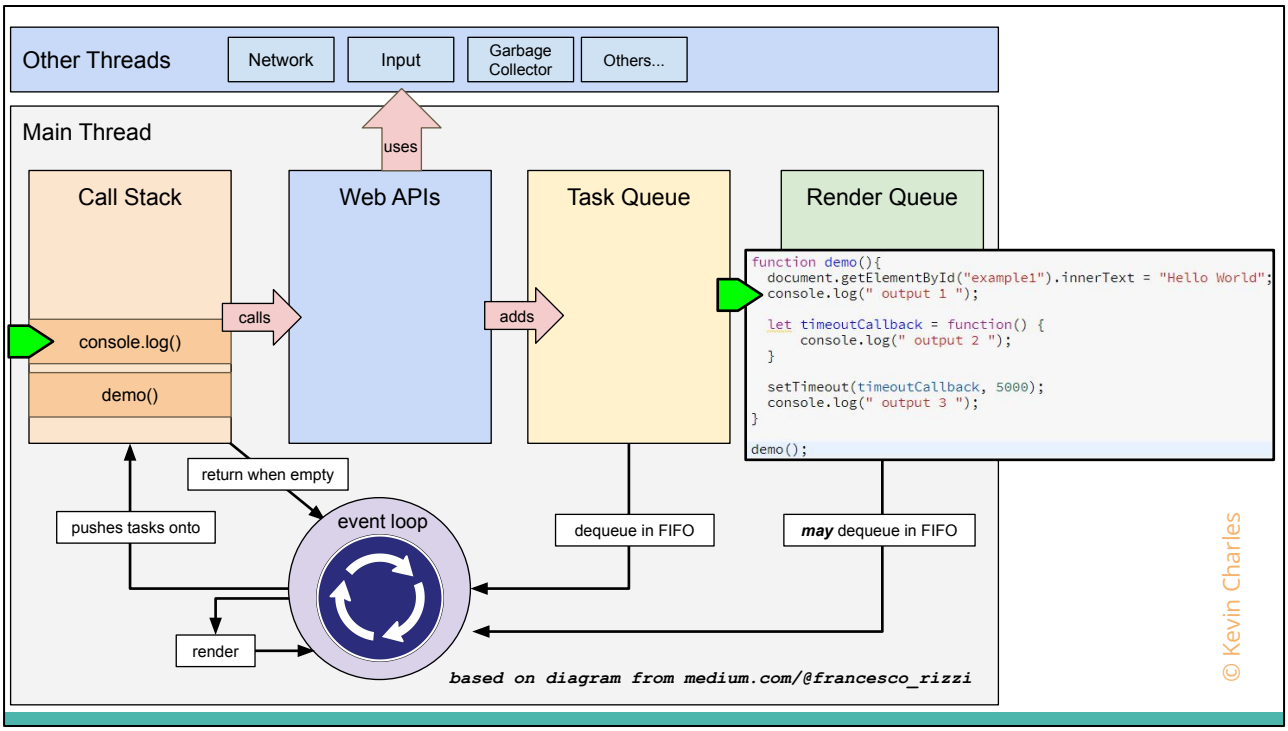


* so we have our function we've defined called demo

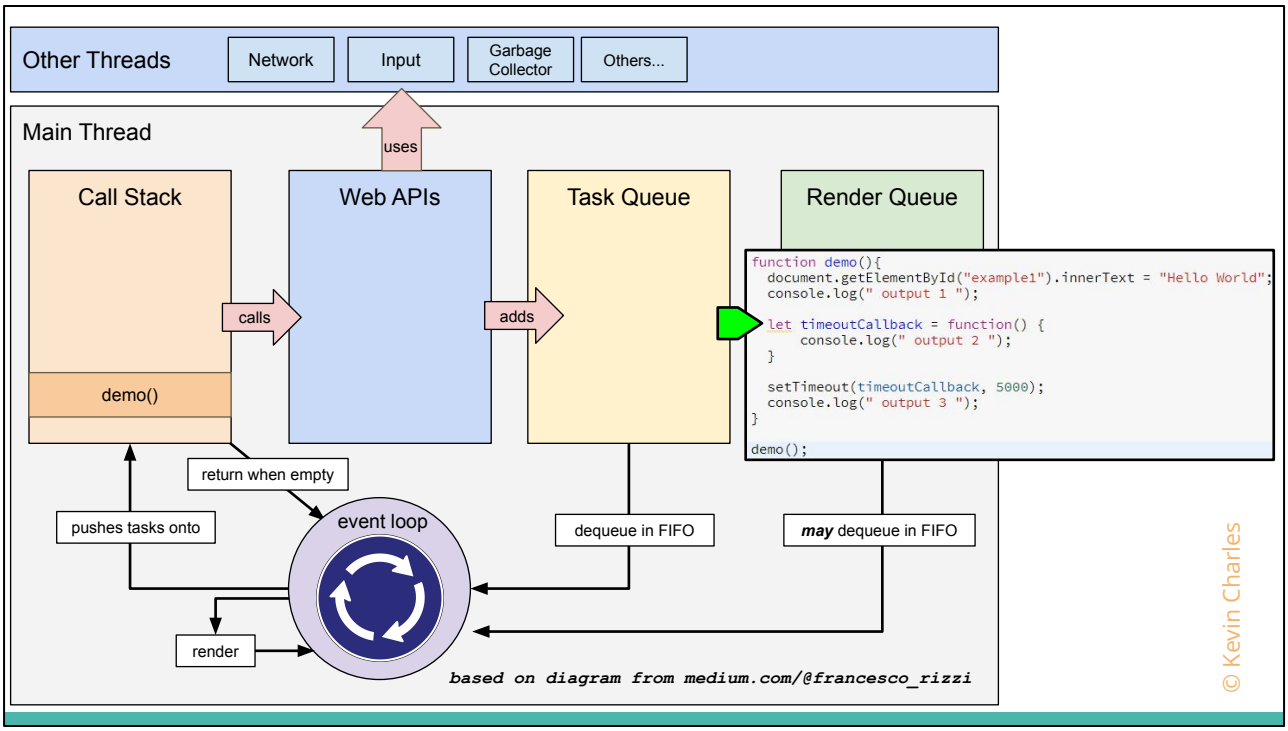
* this gets pushed onto the stack by the event loop which begins executing it



* each function call within demo also gets pushed on and off the stack as they are run, line by line
    * first the Text for the h1 element is updated

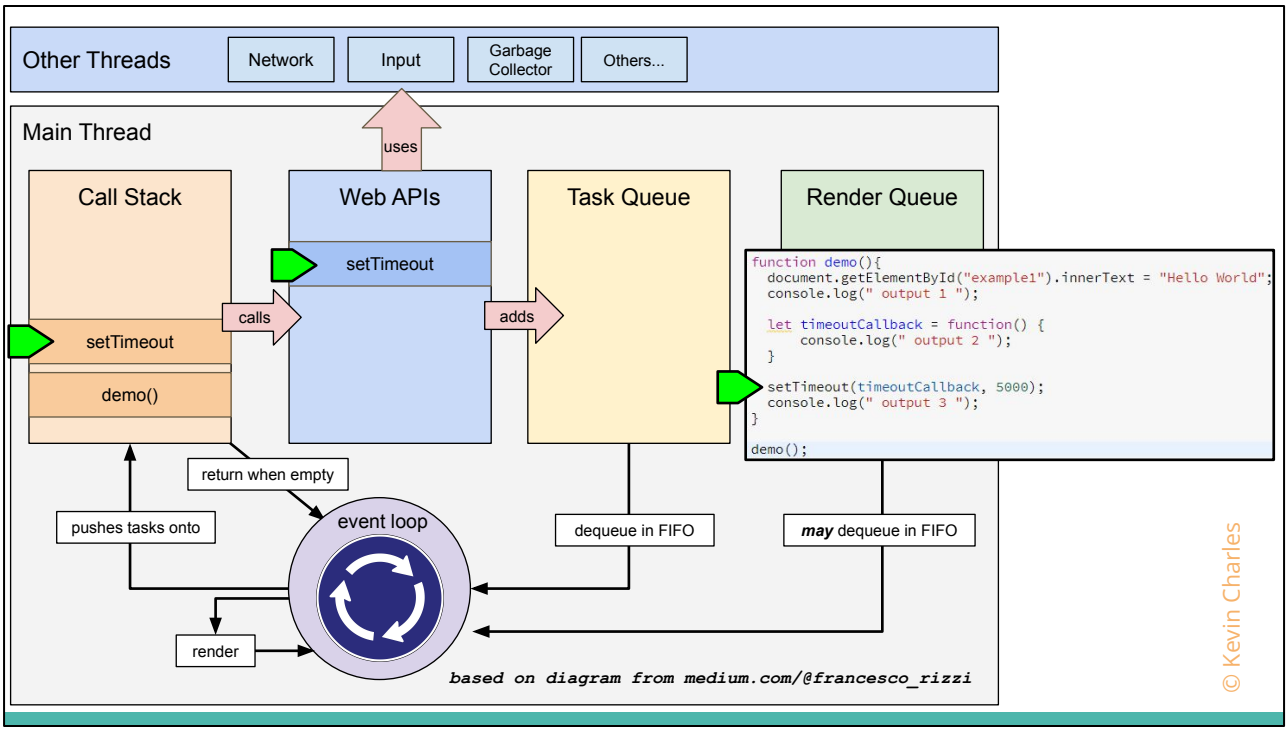


* we then log to the console

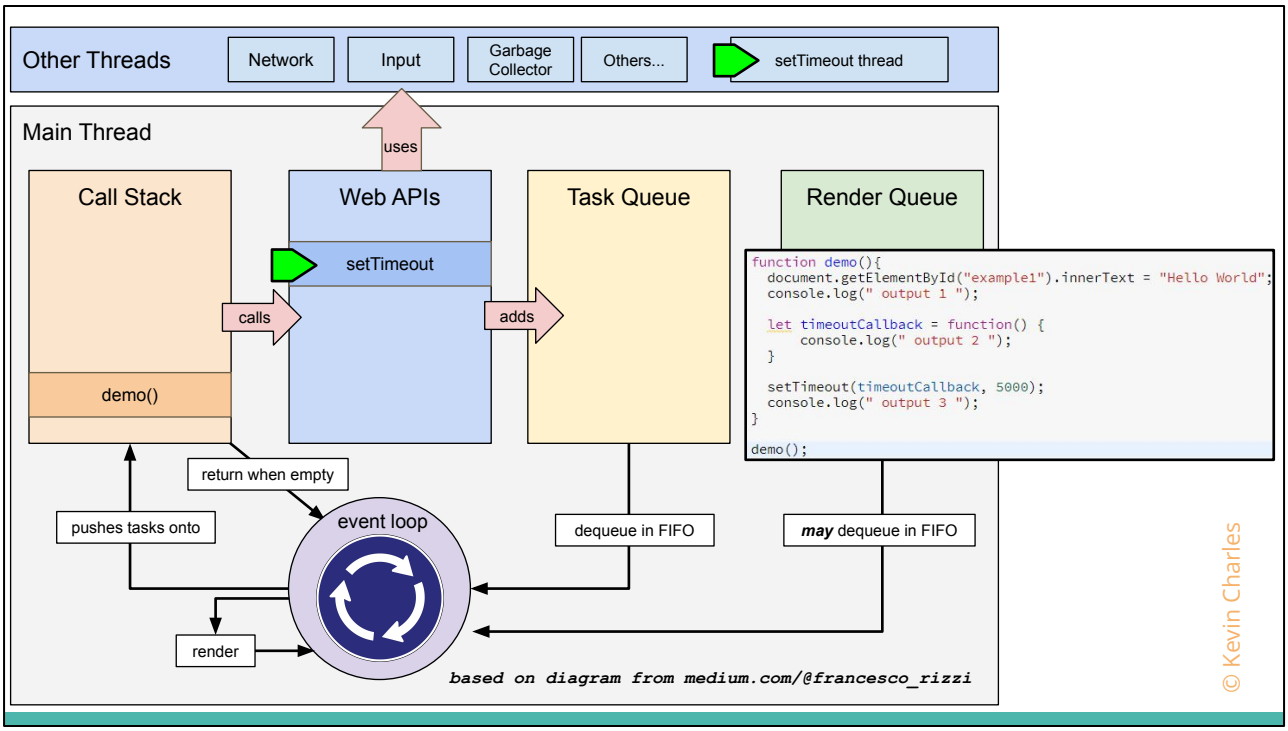


* the callback for the timeout is defined and stored in a variable

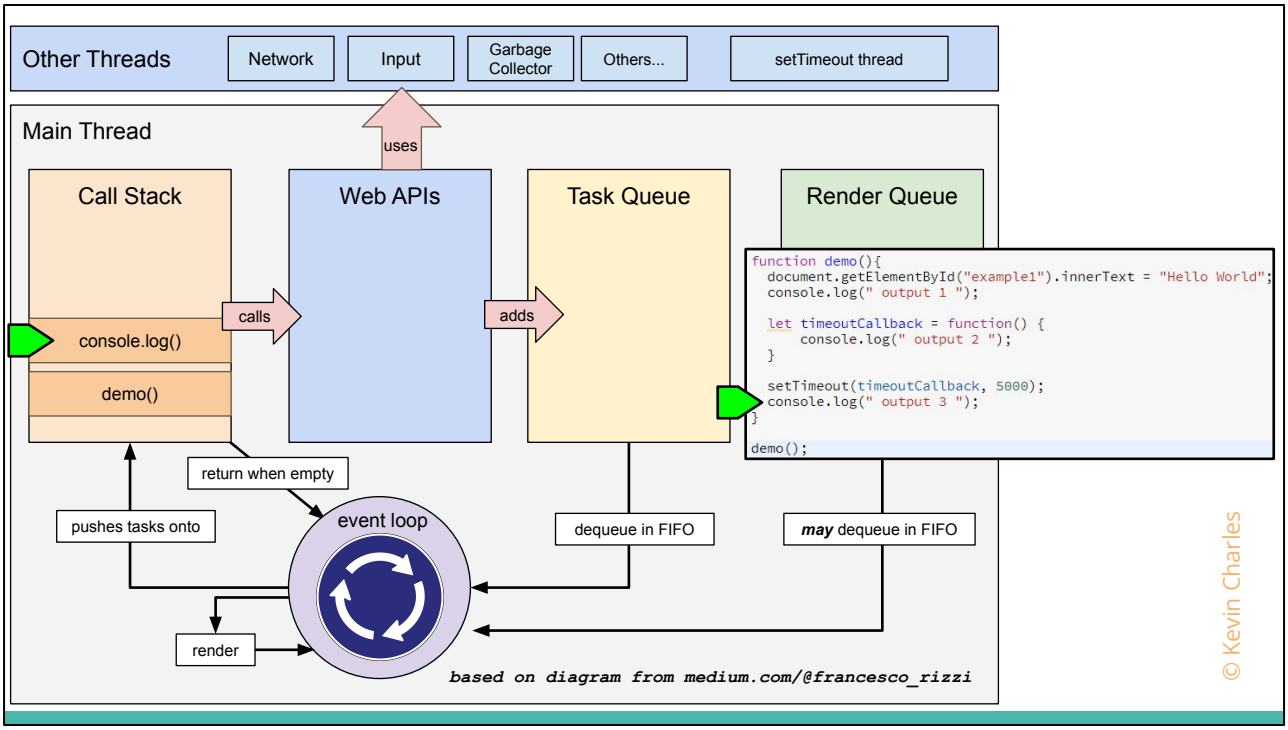
    * this isn't a function call so nothing is placed on the stack



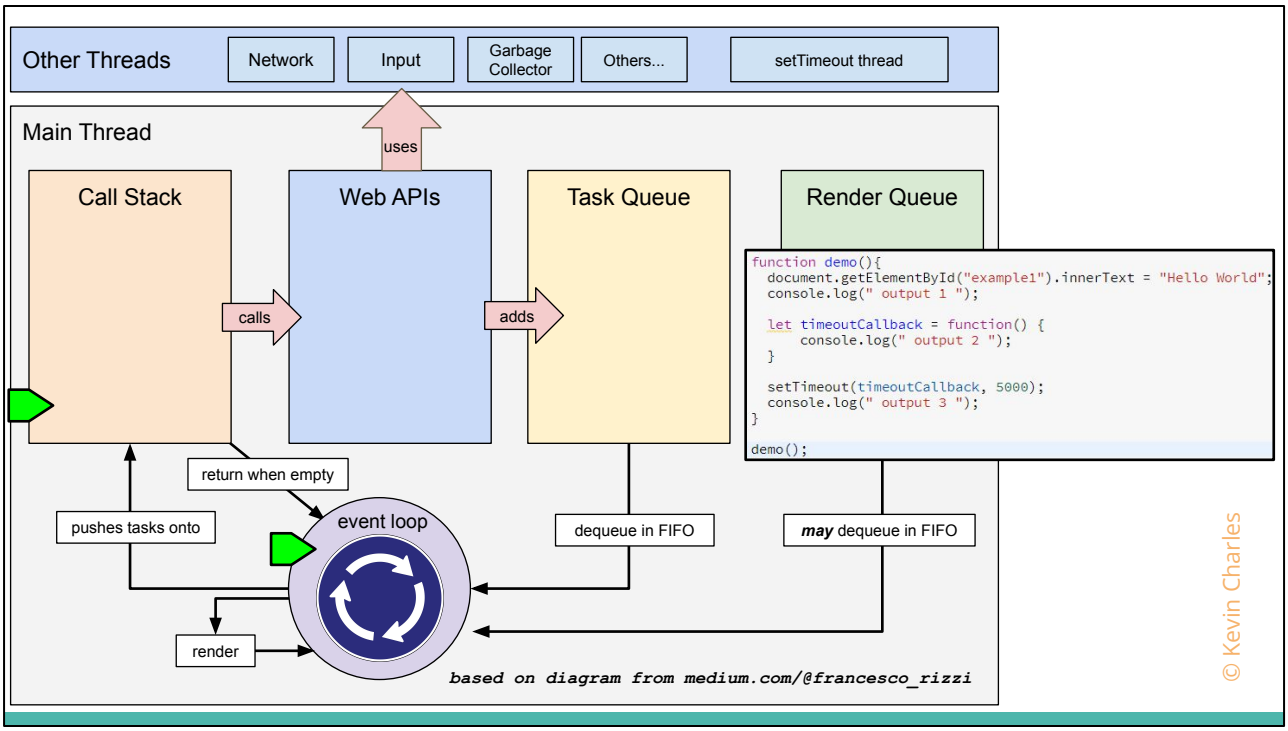
* we call set timeout which takes our call back and returns immediately



* Behind the scenes another thread handles the timeout period in parallel the main thread

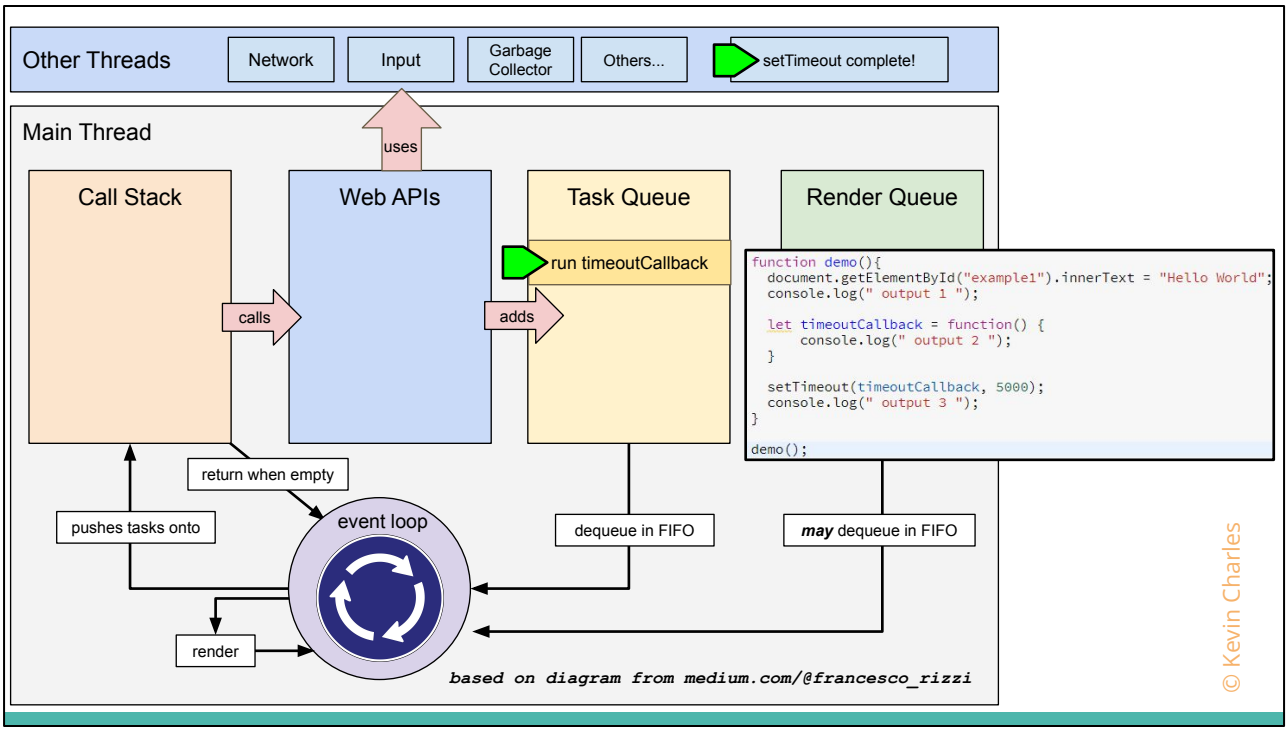


* we log to the console again

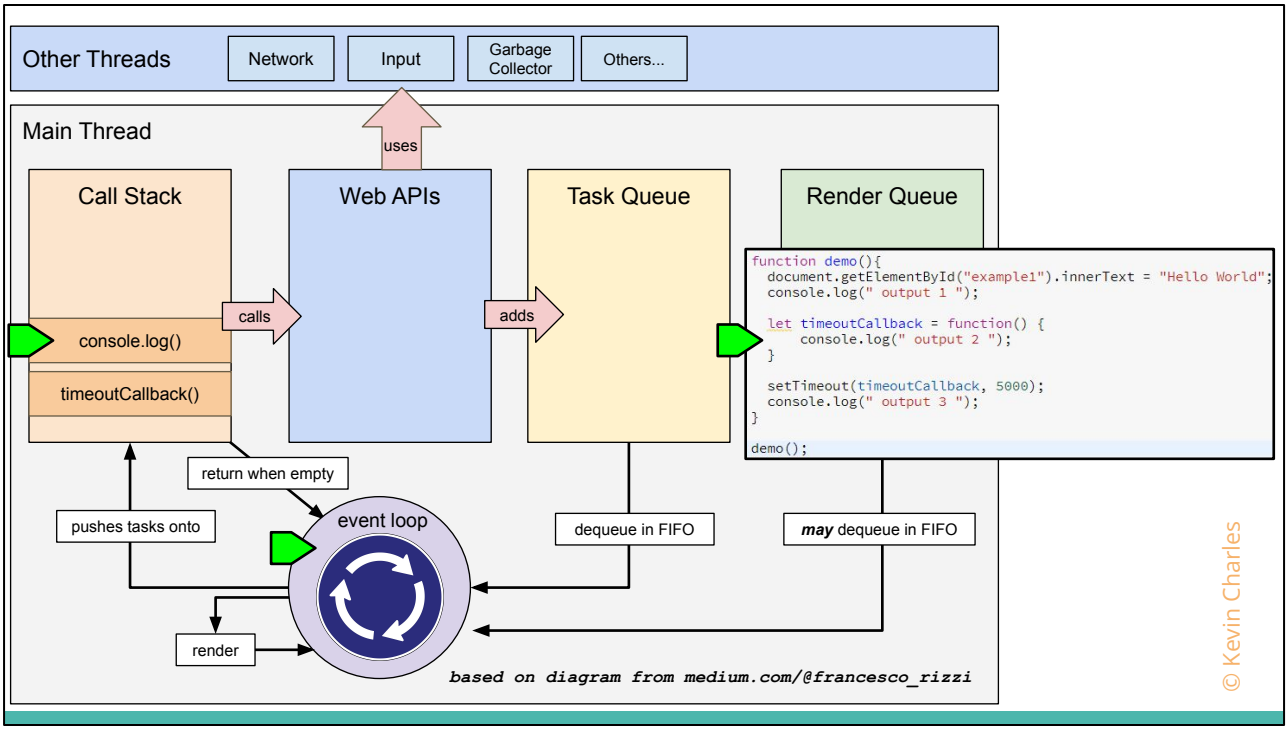


* the demo function finishes running and is popped off the stack

* control is returned to the event loop (which can now render)



* when the thread running set timeout is completed it queues a task to run the callback we gave it



* the event loop then pushes this callback onto the stack

* and we log to the console for the final time

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 5000);
  console.log(" output 3 ");
}

demo();
```

```
function demo(){
  document.getElementById("example1").innerText = "Hello World";
  console.log(" output 1 ");

  let timeoutCallback = function() {
      console.log(" output 2 ");
  }

  setTimeout(timeoutCallback, 0);
  console.log(" output 3 ");
}

demo();
```


* these exact steps will always happen regardless of the length of time you give to set timeout
* even if it set to zero
* as the call stack must be emptied before the timeoutCallback can be run

>>> show both code examples

* this explains the behaviour we saw earlier in the examples

* i'll show you another quick example taking input from a button

Button example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <input id="example2" type="button" value="Click Me!">
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
let button = document.getElementById("example2");

let clickCallback = function () {
  console.log("Hello World");
}

button.addEventListener("click", clickCallback);
```


>>> show code example

* so here we have an html button

Button example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <input id="example2" type="button" value="Click Me!">
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
let button = document.getElementById("example2");

let clickCallback = function () {
  console.log("Hello World");
}

button.addEventListener("click", clickCallback);
```


* and in the javascript we
    * get the button element

Button example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <input id="example2" type="button" value="Click Me!">
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

```
let button = document.getElementById("example2");

let clickCallback = function () {
  console.log("Hello World");
}

button.addEventListener("click", clickCallback);
```


* define a call back called clickCallback then
    * in the callback we will simply print to the console

Button example

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
    <input id="example2" type="button" value="Click Me!">
</body>
<script src="./script.js"></script>
</html>
```

JavaScript

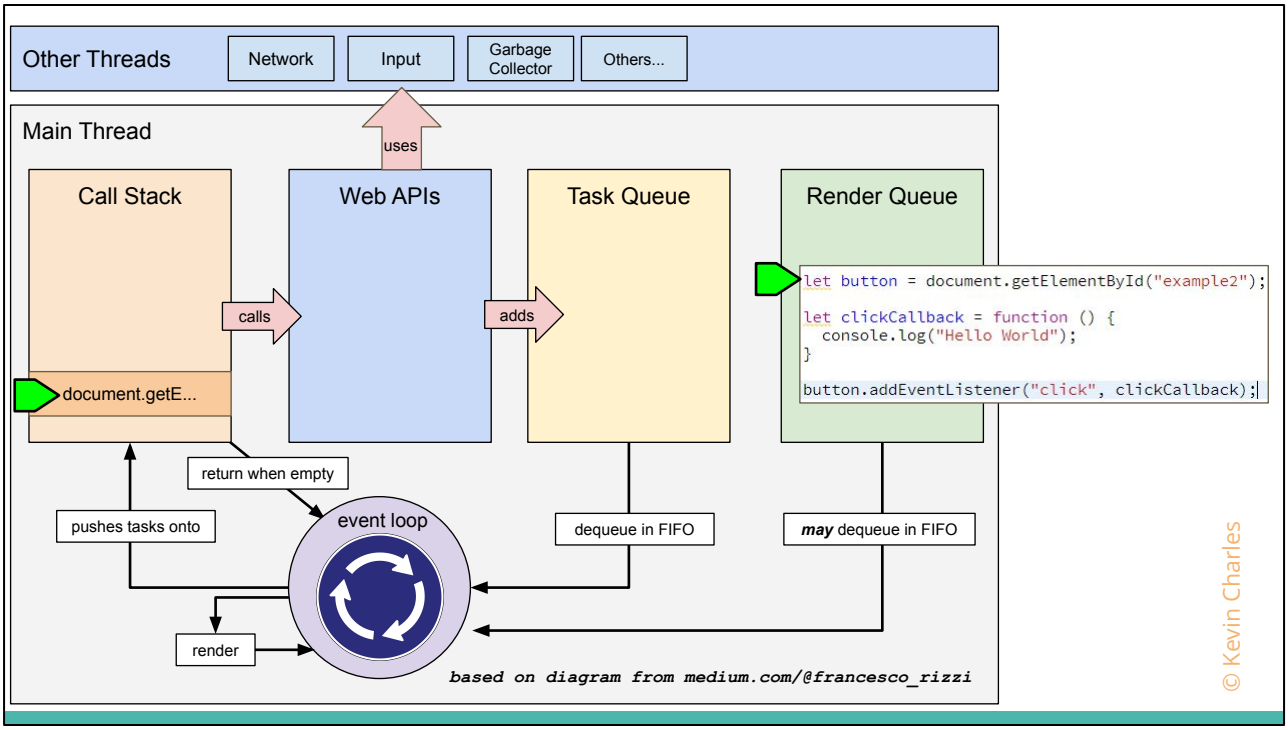
```
let button = document.getElementById("example2");

let clickCallback = function () {
  console.log("Hello World");
}

button.addEventListener("click", clickCallback);
```


- We set the callback as the event handler for the on click event for this button
- I will show you quickly what this code looks like when it's running

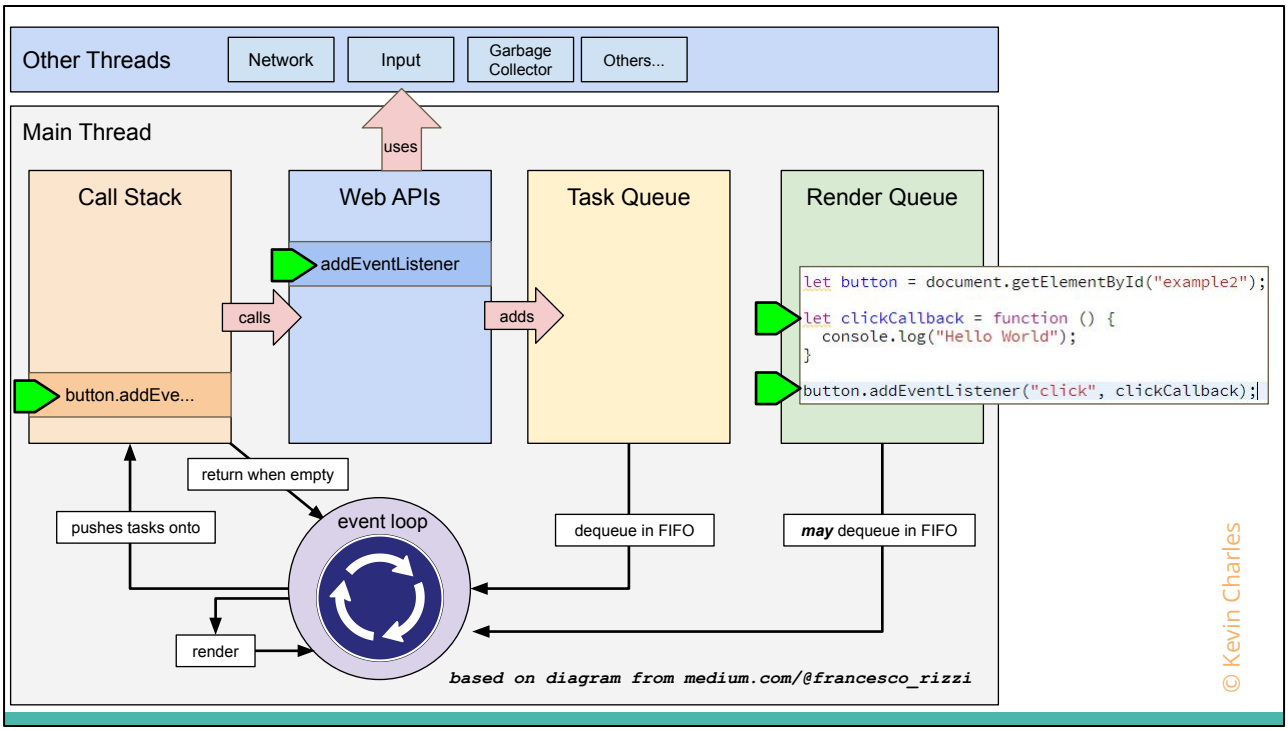
Button example

- So as you can see each time I press the button, we log to the console

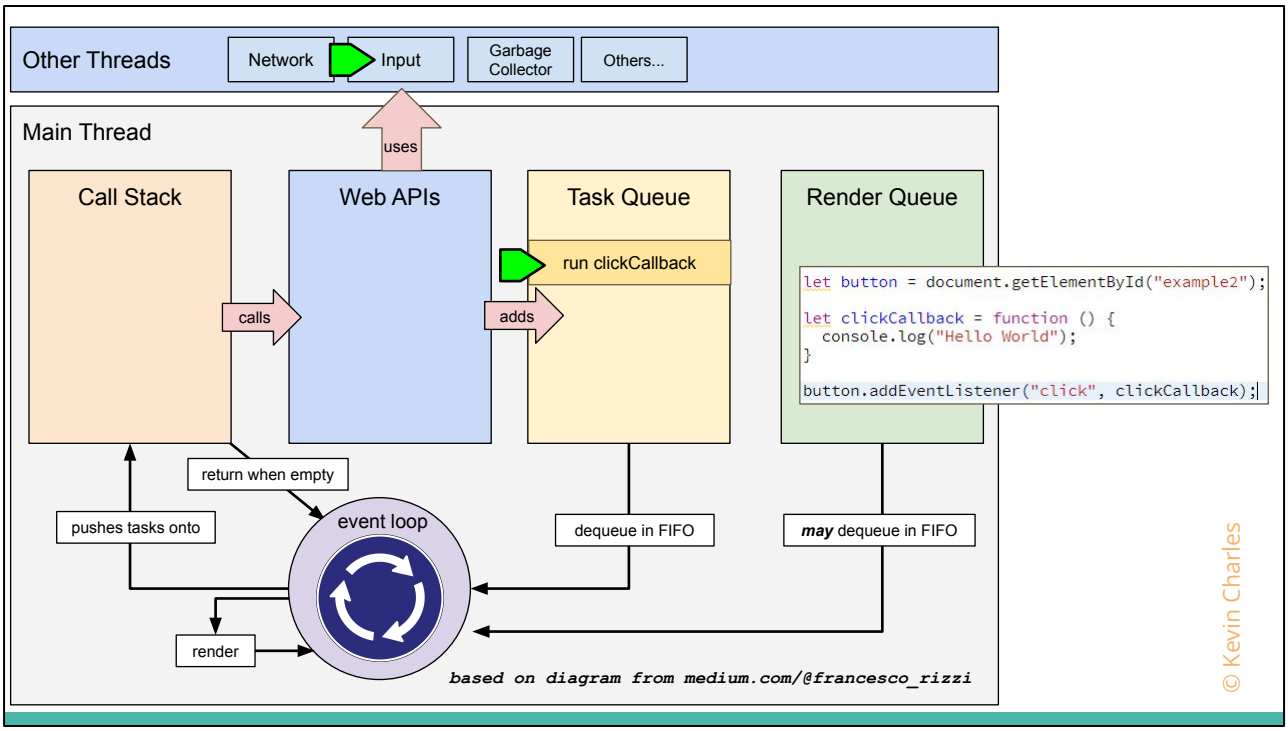
* if we look at what goes on using the diagram it's fairly simple



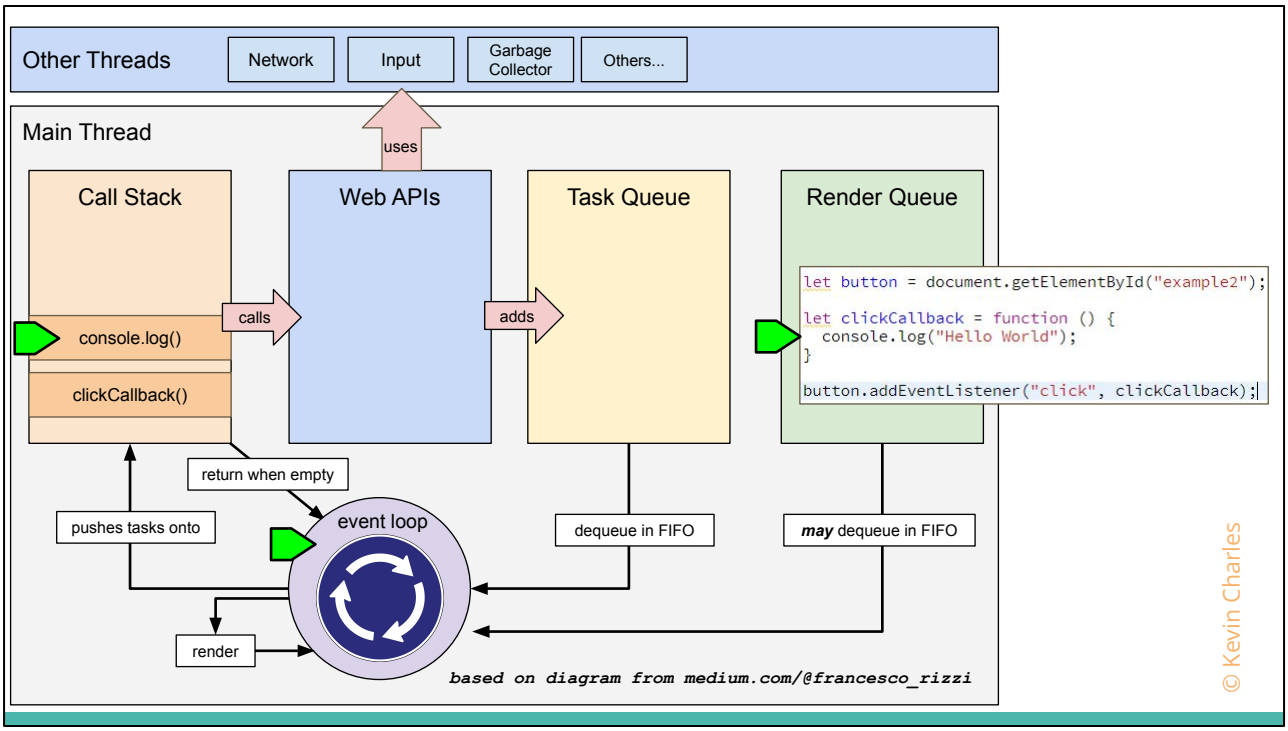
* when we run this, the event loop will place our code in the call stack line by line
 * so we get the button element



* register the callback as an event handler for the click



* when the click happens, another thread handling input events queues a task to run clickCallback



* the event loop will push clickCallback onto the call stack executing it
* we log hello world to the console

And that's an example of how input events work

Websocket example

So this same pattern of registering callbacks is used for some more advanced web apis
take for example a websocket

* In this example we create a web socket and connect to an end point,

* then we create and register callbacks to handle events that the web socket may receive

Websocket example

* There's a callback registered for the on open event (that gets called when the connection is established)

Websocket example

* And another callback for the onmessage event which gets called when the websocket receives data

Websocket example

```javascript
// create WebSocket connection.
const socket = new WebSocket('ws://localhost:1337');

// event - connection opened
socket.addEventListener('open', function (event) {
    socket.send('Hello Server!');
});

// event - websocket message received
socket.addEventListener('message', function (event) {
    console.log('Message from server ', event.data);
});
```


- I won't use the diagram to illustrate this but

* behind the scenes there's a thread that's waiting for the connection to open and data to be received on the web socket

* if either of these events happen the thread will then queue a task to call the registered callback

Web Workers

- No explicit control over web API threads
- How to create your own thread?

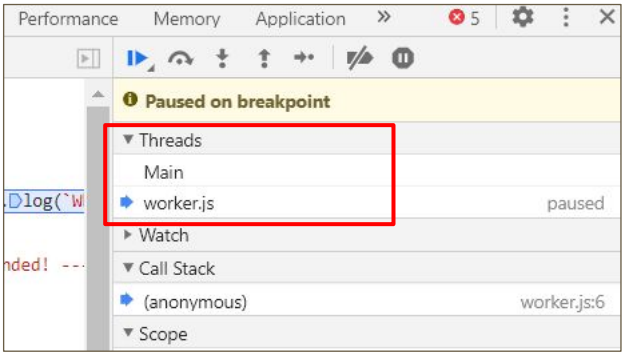
* So as we've seen you don't have explicit control over how web apis manage threads in the background

* so what happens if you want to spawn your own thread to run in parallel with the main thread?

* This is where web workers come in

Web Workers

```
JavaScript ▾
let myWorker = new Worker('worker.js');
```

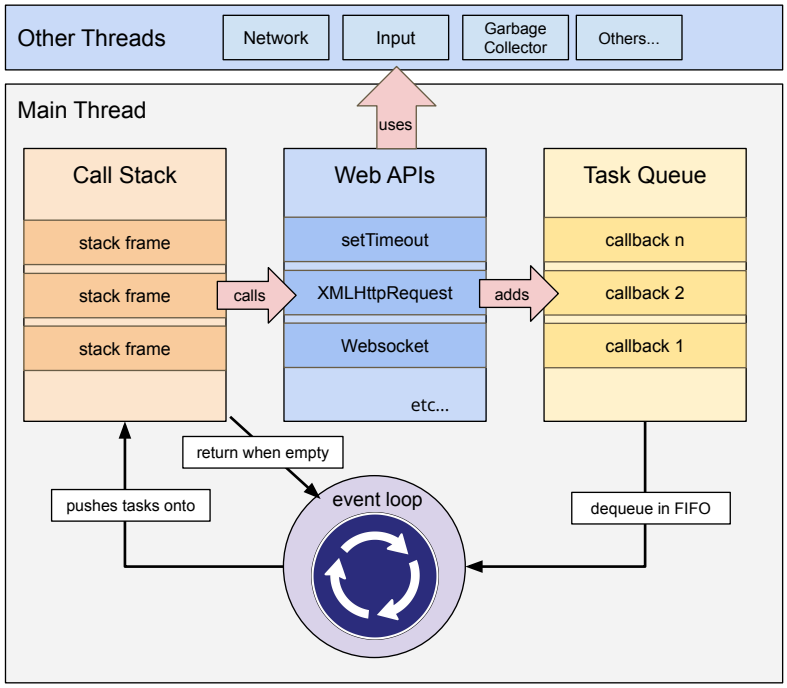


* You can spawn web workers from the main thread like this
    * you call the constructor using the name of the file you want to run in the new thread

>>>show code

* this creates a completely separate thread from the main one

* we use a worker thread In the [redacted] to improve performance by taking things like [redacted] off the main thread



* Web workers are almost identical to the main thread, they have a similar event loop with one key difference - you can't manipulate the DOM from a worker thread
* so as you can see in the diagram there is no rendering queue

>>> show modified diagram

Web Workers -> Main Thread Communication

Main Thread

```javascript
let myWorker = new Worker('worker.js');

// handle message from worker
myWorker.onmessage = function(msg) {
  let data = msg.data;
  console.log('Message received from worker');
}

// send message to worker
myWorker.postMessage([first.value,second.value]);
```

Worker Thread

```javascript
// handle message from main thread
onmessage = function(msg) {
  let data = msg.data;
  console.log('Message received from main script');
}

// send message to main thread
postMessage(workerResult);
```


* a web worker and the main thread can communicate using a simple API

- Each side can send a message using a function called postMessage
- And each side can receive a message by creating an event handler called onmessage
- There's a little bit more code involved in the main thread as it has to create the worker

>>> show code

Web Worker - Data Transfer

- postMessage / onmessage API can transfer data
- Data is copied not shared
- Some datatypes support transferring ownership
  - ArrayBuffer
  - MessagePort
  - ImageBitmap
  - OffscreenCanvas
- Transferred objects not available in original thread

* You can use this API to transfer data between each thread and for most types this data is copied from one thread to the other and not shared

* There are a few data types however where ownership of an object can be transferred from one thread to another - which means no copying

* as of now the four types are
    * ArrayBuffer
    * MessagePort
    * ImageBitmap
    * OffscreenCanvas

* when an object gets transferred from one thread to another, you can't refer to it in the original thread again (unless you transfer it back)

Wrapping up - Javascript Applications

- Collection of event handlers
- Organised into objects and functions

* to wrap up I want to say that you can think of a javascript application

* as collection of what you want to happen when certain events occur in the browser (organised into objects and functions)

Slide removed


```
Slide removed
```

Thanks