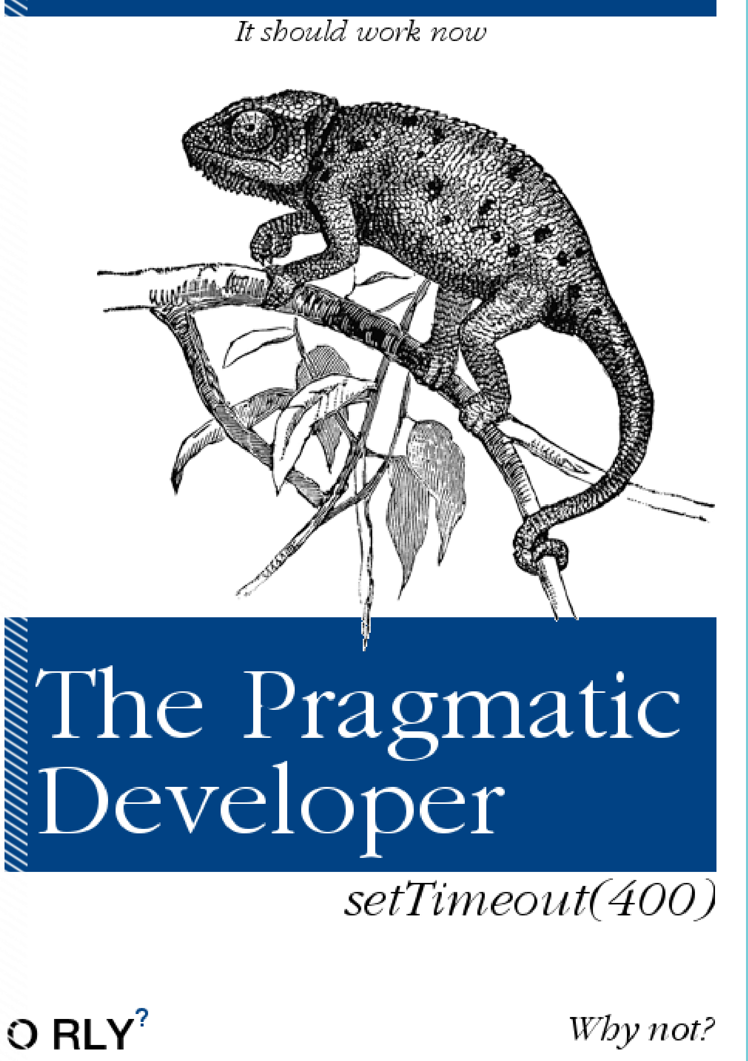
Web API - part II
Event loop, History and, Location APIs

Frontend Junior Program - 2022

CONFIDENTIAL | © 2022 EPAM Systems, Inc.



Agenda

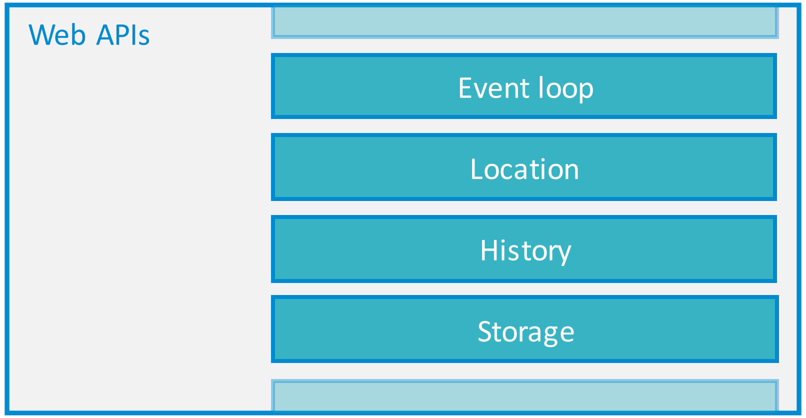
CONFIDENTIAL | © 2022 EPAM Systems, Inc.

Web APIs

There are many different Web APIs

Most of them are used in specific cases, however, some are utilized on every project: now, we are focusing on the Event loop, the Location and the History APIs.

All these are concerns of the host environment (browser) and are not the part of the JavaScript, but the HTML Standard.



because these browser interfaces are accessible on objects, sometimes this is called BOM (Browser Object Model)

EVENT LOOP, SETTIMEOUT

Event Loop

A developer usually meets with the event loop in 2 cases:

1., when they run into a bug which occurs only occasionally, and they desperately try to *make it just work* using setTimeout with some random delay;

2., when their PR with that random delay will be declined;

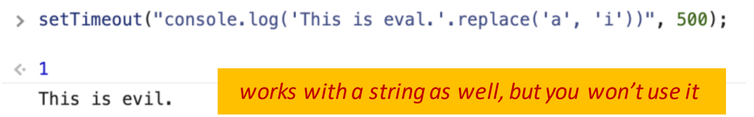
They are usually wondering – "*why, when it is working for me perfectly?*"

Here we are to answer.

setTimeout

It is easier to understand what is going on if we play a bit with setTimeout

This code seems pretty straightforward: the browser tries to execute the function after the timer expires.



```
> setTimeout("console.log('This is eval.'.replace('a', 'i'))", 500);
<· 1
  This is evil.
```

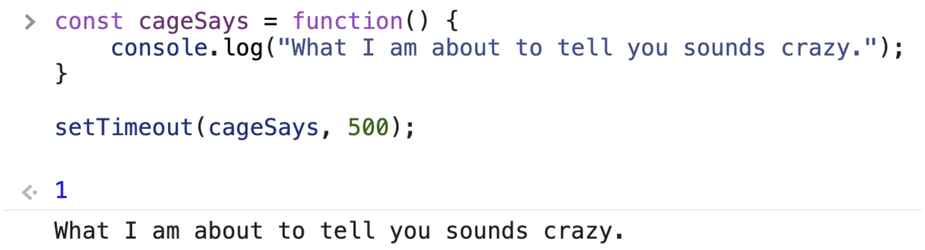
works with a string as well, but you won't use it

The callback function to be executed.

the delay is 500 ms

returns a timer ID
(with this the timer can be cancelled)

after a delay, it "~~executes~~"* the code



```
> const cageSays = function() {
      console.log("What I am about to tell you sounds crazy.");
  }

  setTimeout(cageSays, 500);
<· 1
  What I am about to tell you sounds crazy.
```

** actually, this is not the case – we will see the details now*

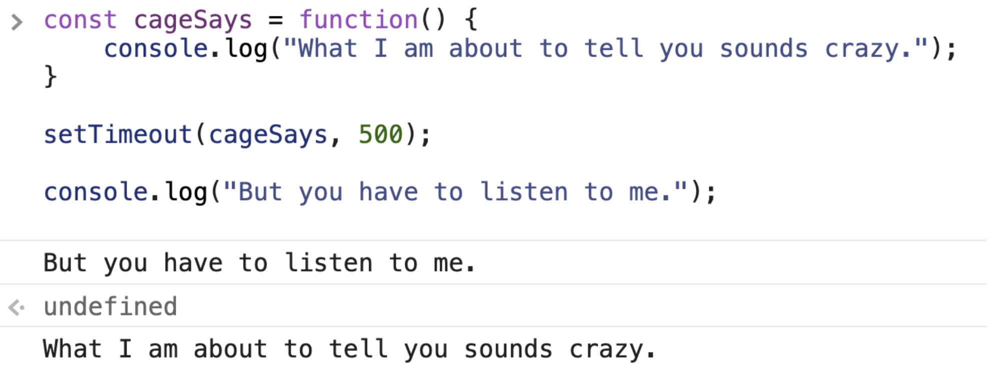
setTimeout – does not wait, it is async!

Things start to be complicated when we realize that setTimeout does not actually wait

When we set a callback to be executed it actually does what the name suggests: it only sets up a timer and the execution continues without any waiting.

So, when it will actually run? Well, after a 500ms delay, that's for sure, but when exactly?

You may ask: how to sleep the execution in JavaScript then? The short answer is: there is no internal function for sleep – you have to implement that.



```
> const cageSays = function() {
      console.log("What I am about to tell you sounds crazy.");
  }

  setTimeout(cageSays, 500);

  console.log("But you have to listen to me.");
```

this will be immediately executed

tadam!

```
  But you have to listen to me.
<· undefined
  What I am about to tell you sounds crazy.
```

a sleep() function at your disposal

Here you are, a simple sleep function

What is interesting here, however, is not that awesome sleep function, but
while the browser is working on that loop, it stops doing anything else.

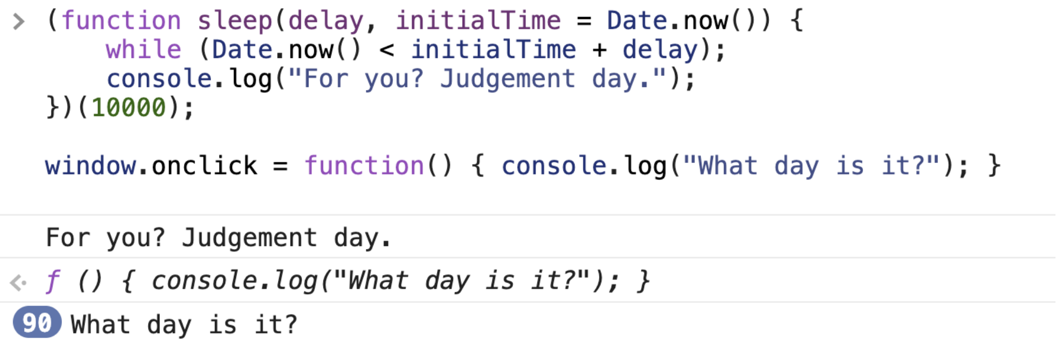
It does it because the JavaScript is a single threaded language.

try to click as much as you can!

it does not really care...

until it finishes that pretty while loop

the good news is the event loop is
still registering the click events
(will be explained a bit later)



```
> (function sleep(delay, initialTime = Date.now()) {
      while (Date.now() < initialTime + delay);
      console.log("For you? Judgement day.");
  })(10000);

  window.onclick = function() { console.log("What day is it?"); }

  For you? Judgement day.
<· f () { console.log("What day is it?"); }
90 What day is it?
```

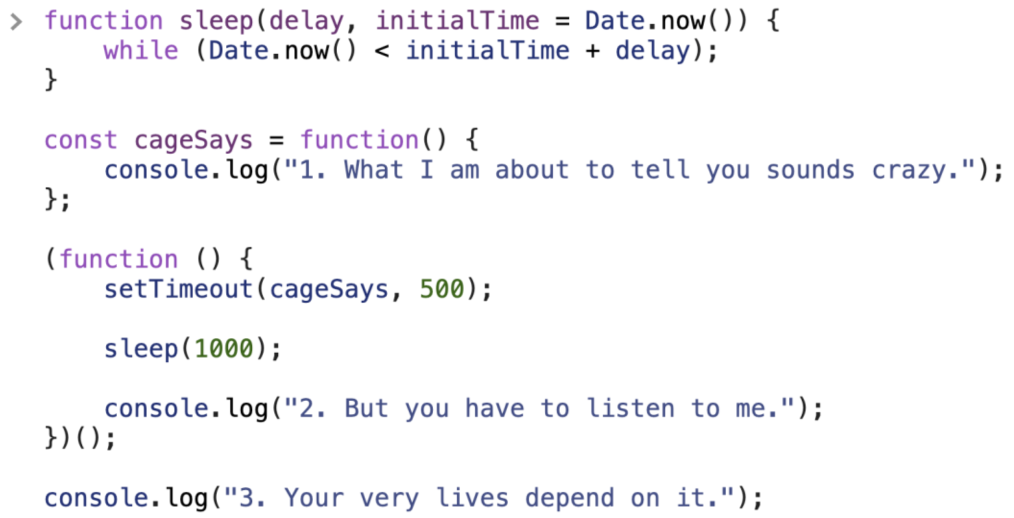
JavaScript is single threaded - new jobs must sit and wait…

Also, running long tasks could be very exhausting for the browser and could be a problem for your async calls (will see).

setTimeout – runs async, later, but much how later?

Let’s develop it heuristically!

I am sure that after a bit of trial and error we will figure it out, shall we? Our presumptions are:



```
> function sleep(delay, initialTime = Date.now()) {
      while (Date.now() < initialTime + delay);
  }

  const cageSays = function() {
      console.log("1. What I am about to tell you sounds crazy.");
  };

  (function () {
      setTimeout(cageSays, 500);

      sleep(1000);

      console.log("2. But you have to listen to me.");
  })();

  console.log("3. Your very lives depend on it.");
```

we schedule 500ms waiting here

500ms should be finished for now!

so every console.logs should run according to the numbers, right?

setTimeout – could run much later

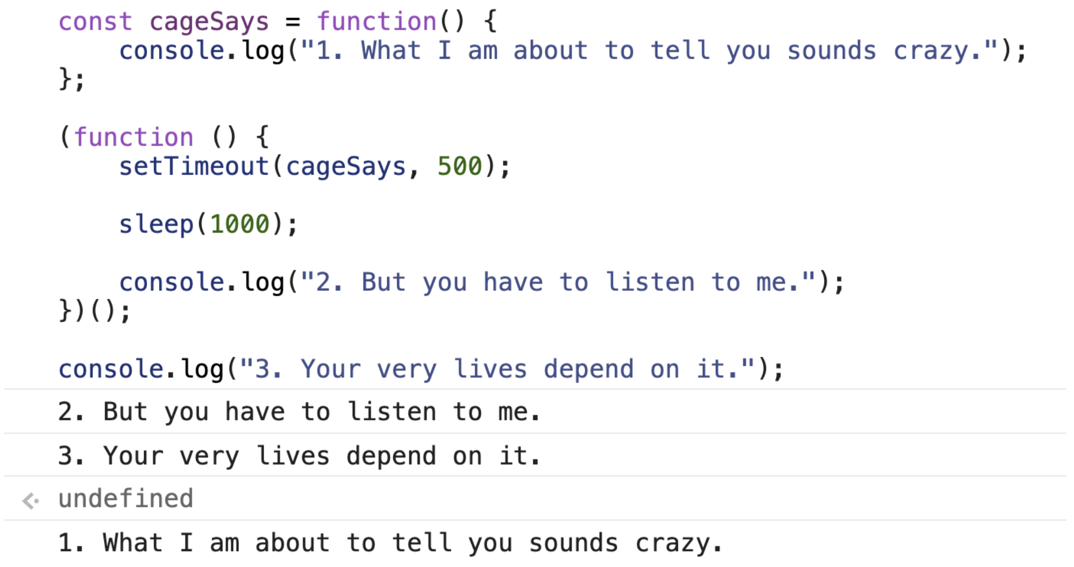
Wrong!

Hmm, it seems it waits until the function finishes…

But wait, the console.log placed outside the function runs before as well!

these look in order…

but our poor callback runs last



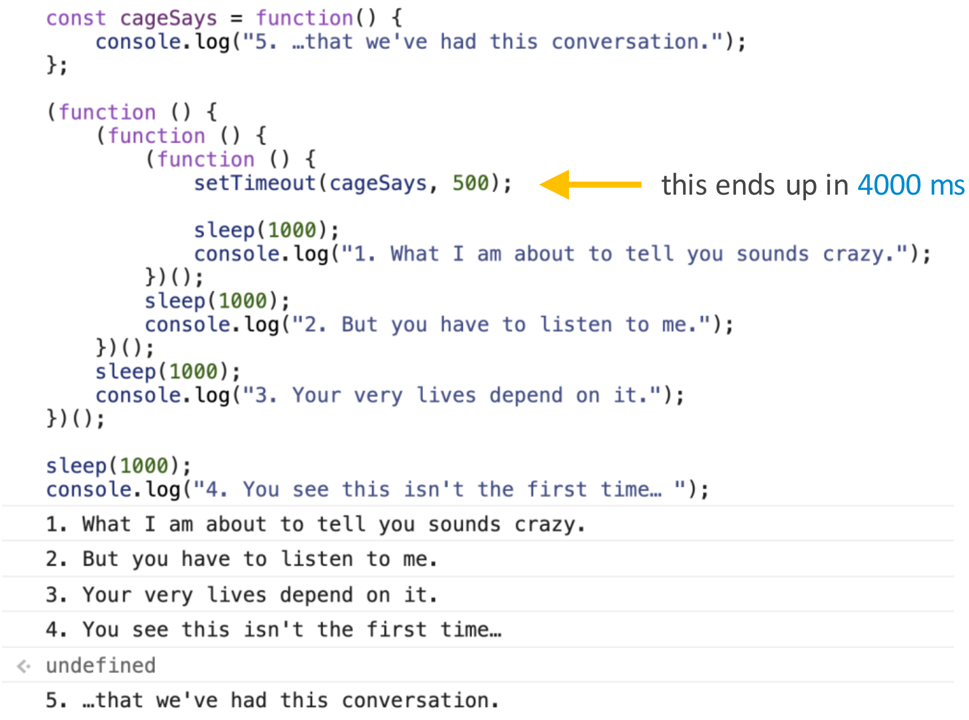
```
const cageSays = function() {
    console.log("1. What I am about to tell you sounds crazy.");
};

(function () {
    setTimeout(cageSays, 500);

    sleep(1000);

    console.log("2. But you have to listen to me.");
})();

console.log("3. Your very lives depend on it.");
```

```
2. But you have to listen to me.
3. Your very lives depend on it.
<· undefined
1. What I am about to tell you sounds crazy.
```

setTimeout – later, and we don’t know, when

All these are the result of the event loop

Once we understand how the event loop works, it will all make sense!

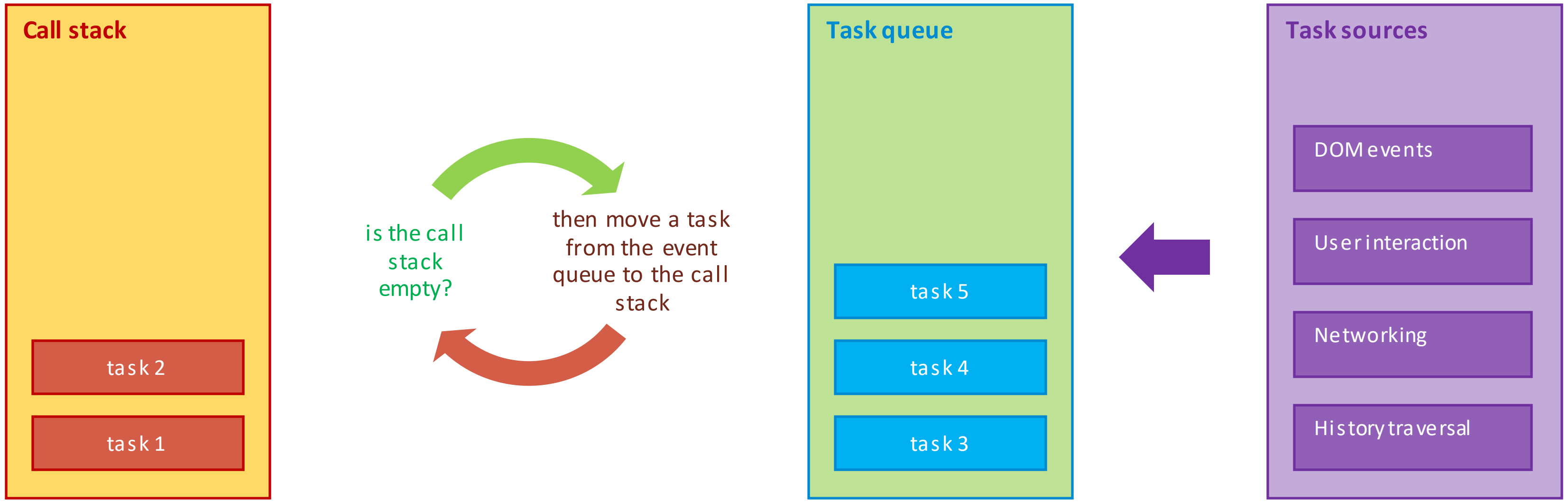


```
const cageSays = function() {
    console.log("5. …that we've had this conversation.");
};

(function () {
    (function () {
        (function () {
            setTimeout(cageSays, 500);

            sleep(1000);
            console.log("1. What I am about to tell you sounds crazy.");
        })();
        sleep(1000);
        console.log("2. But you have to listen to me.");
    })();
    sleep(1000);
    console.log("3. Your very lives depend on it.");
})();

sleep(1000);
console.log("4. You see this isn't the first time… ");
```

```
1. What I am about to tell you sounds crazy.
2. But you have to listen to me.
3. Your very lives depend on it.
4. You see this isn't the first time…
<· undefined
5. …that we've had this conversation.
```

this ends up in 4000 ms

Event loop



the setTimeout callback must wait until the entire call stack will be empty

the setTimeout callback will be placed here *after the delay time*

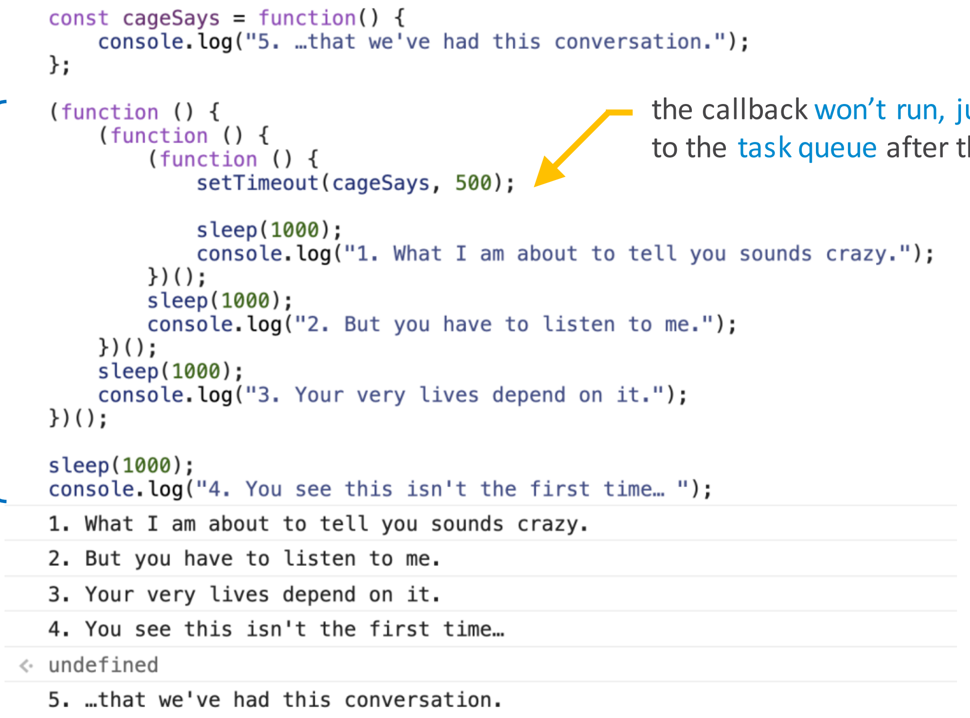
the events come from different sources, and the browser could prioritise between them

http://latentflip.com/loupe/

setTimeout – breakdown

these all are built on the call stack already

the callback won’t run, just will be moved to the task queue after the delay



```
const cageSays = function() {
    console.log("5. …that we've had this conversation.");
};

(function () {
    (function () {
        (function () {
            setTimeout(cageSays, 500);

            sleep(1000);
            console.log("1. What I am about to tell you sounds crazy.");
        })();
        sleep(1000);
        console.log("2. But you have to listen to me.");
    })();
    sleep(1000);
    console.log("3. Your very lives depend on it.");
})();

sleep(1000);
console.log("4. You see this isn't the first time… ");
```

```
1. What I am about to tell you sounds crazy.
2. But you have to listen to me.
3. Your very lives depend on it.
4. You see this isn't the first time…
< undefined
5. …that we've had this conversation.
```

setTimeout – async calls if meet…

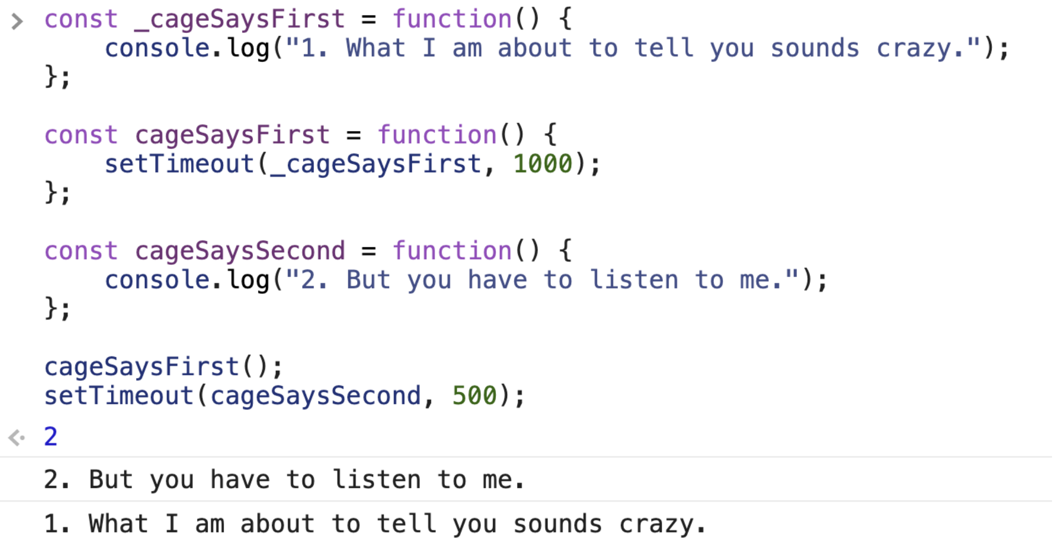
Let’s see a real-world situation: we have to execute a task after something has been finished

Can we use the setTimeout for that?

it could be anything (server call, component rendering), we can’t see its internals, we just have to wait for that

so we wait a bit

but it could not be enough – and while *it may work* on your workstation, it could fail at the visitor



```
> const _cageSaysFirst = function() {
      console.log("1. What I am about to tell you sounds crazy.");
  };

  const cageSaysFirst = function() {
      setTimeout(_cageSaysFirst, 1000);
  };

  const cageSaysSecond = function() {
      console.log("2. But you have to listen to me.");
  };

  cageSaysFirst();
  setTimeout(cageSaysSecond, 500);
<· 2
  2. But you have to listen to me.
  1. What I am about to tell you sounds crazy.
```

A key takeaway –
"soldier, you never use setTimeout to wait for anything async, am I clear?"

Use a callback or an event for that – but setTimeout is never a solution.

SETINTERVAL

setInterval

setInterval schedules a task for running periodically

And it does it really, but due to the event loop, it could lead into surprises.

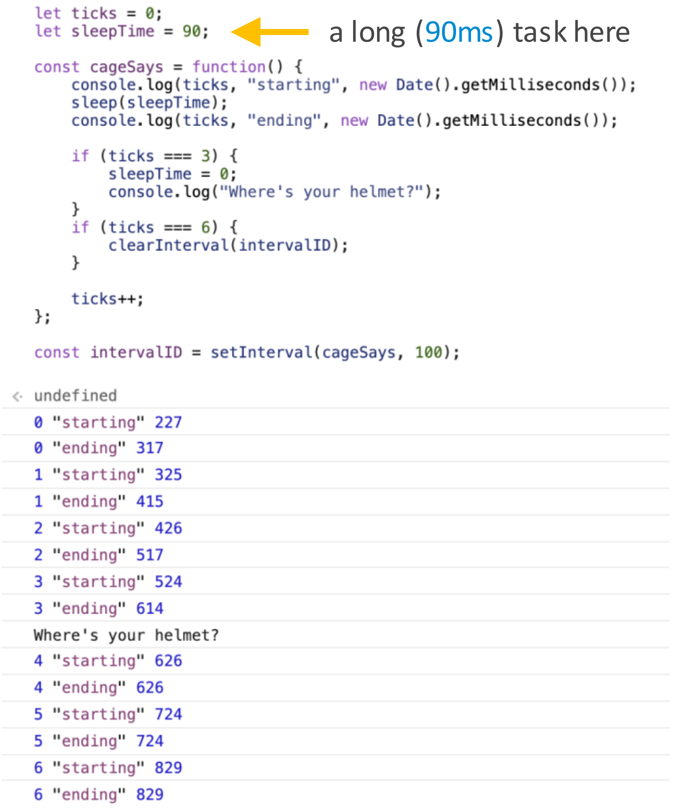
a long (90ms) task here

100ms interval could mean...

Just ~10ms between the end of the task and the start of the task.

If the scheduled task takes longer, then the interval loop will consume the whole CPU time

now it is fine, but you never rely on that the task will be finished on time

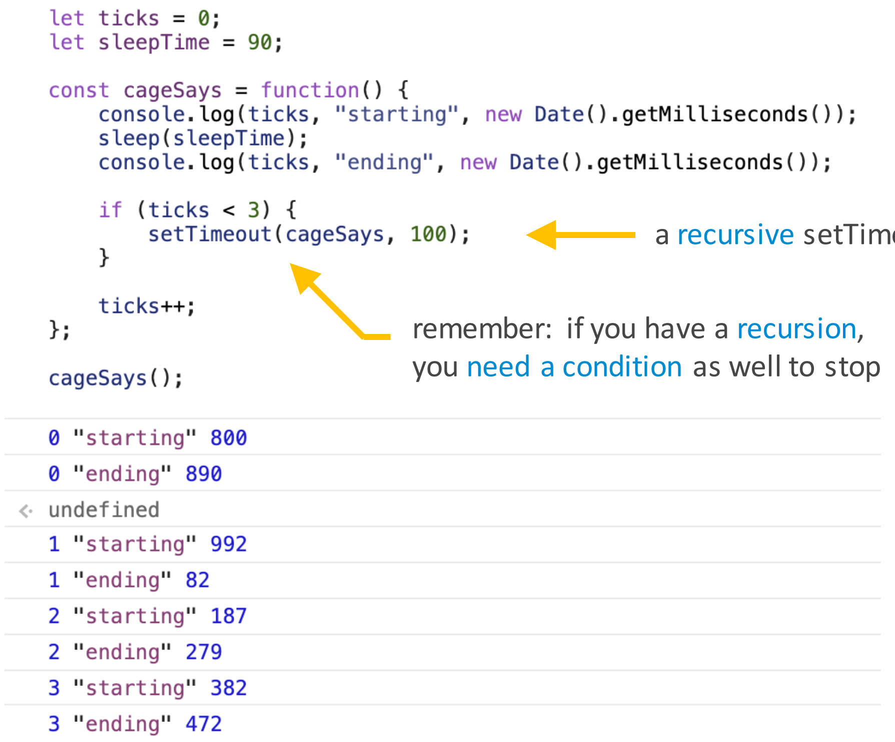


```
let ticks = 0;
let sleepTime = 90;

const cageSays = function() {
    console.log(ticks, "starting", new Date().getMilliseconds());
    sleep(sleepTime);
    console.log(ticks, "ending", new Date().getMilliseconds());

    if (ticks === 3) {
        sleepTime = 0;
        console.log("Where's your helmet?");
    }
    if (ticks === 6) {
        clearInterval(intervalID);
    }

    ticks++;
};

const intervalID = setInterval(cageSays, 100);
```

```
< undefined
0 "starting" 227
0 "ending" 317
1 "starting" 325
1 "ending" 415
2 "starting" 426
2 "ending" 517
3 "starting" 524
3 "ending" 614
Where's your helmet?
4 "starting" 626
4 "ending" 626
5 "starting" 724
5 "ending" 724
6 "starting" 829
6 "ending" 829
```

instead of setInterval – use recursive setTimeout

It is a usual pattern using recursive setTimeout calls instead of setIntervals



```
let ticks = 0;
let sleepTime = 90;

const cageSays = function() {
    console.log(ticks, "starting", new Date().getMilliseconds());
    sleep(sleepTime);
    console.log(ticks, "ending", new Date().getMilliseconds());

    if (ticks < 3) {
        setTimeout(cageSays, 100);
    }

    ticks++;
};

cageSays();
```

```
0 "starting" 800
0 "ending" 890
<· undefined
1 "starting" 992
1 "ending" 82
2 "starting" 187
2 "ending" 279
3 "starting" 382
3 "ending" 472
```

a recursive setTimeout call

remember: if you have a recursion, you need a condition as well to stop

proper 100ms intervals between the end and the start

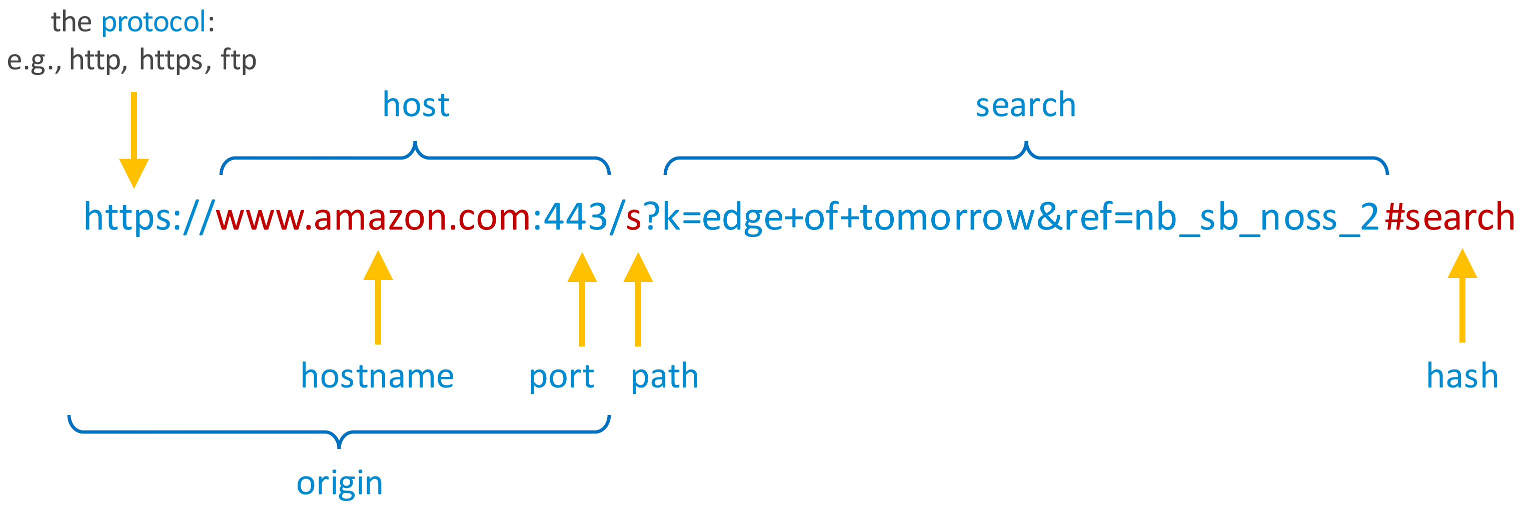
LOCATION

The location API is important basically in 2 cases:

1., when you want to know your current position

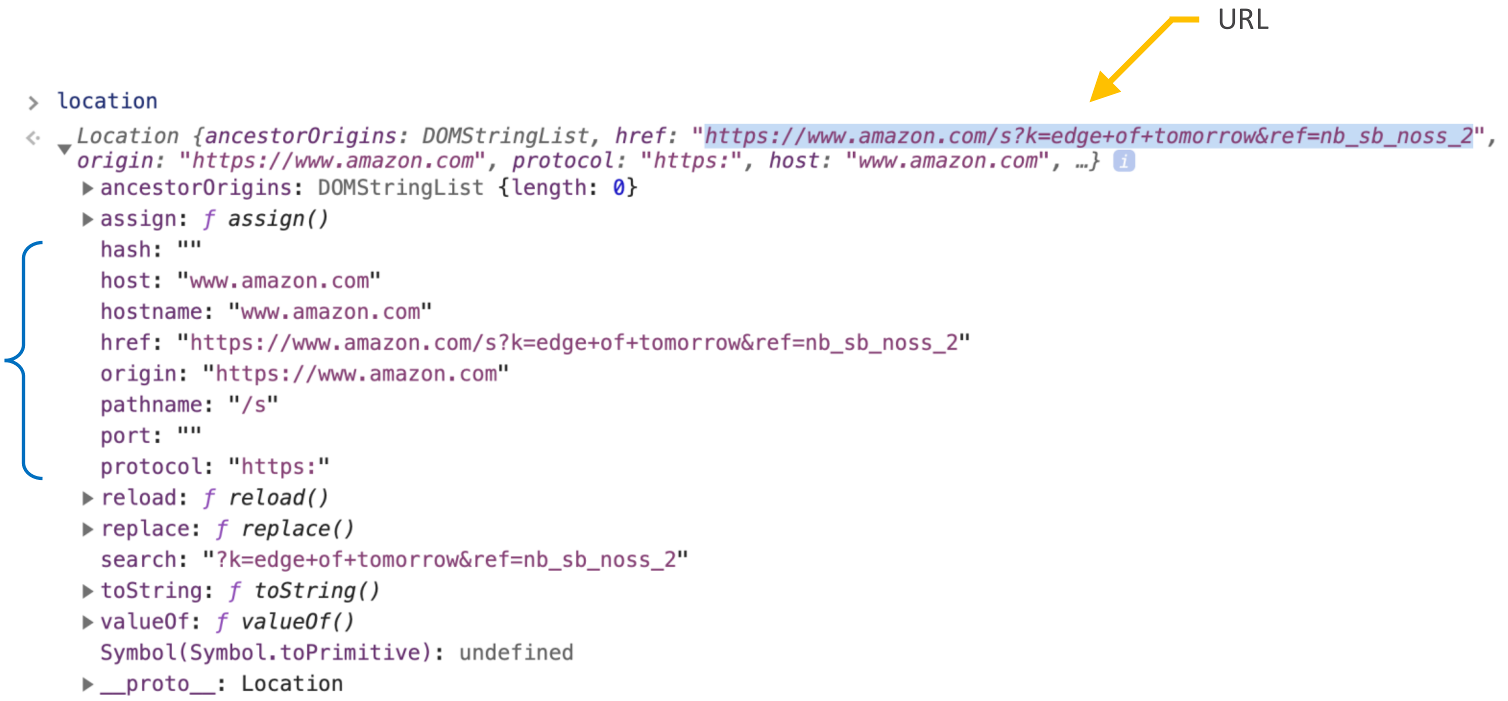
2., when you want to set your target

Location - URL


the protocol:
e.g., http, https, ftp
host
search
https://www.amazon.com:443/s?k=edge+of+tomorrow&ref=nb_sb_noss_2#search
hostname
port
path
hash
origin

the most important concept about location is the URL (Uniform Resource Locator)

Location



```
> location
<· ▼Location {ancestorOrigins: DOMStringList, href: "https://www.amazon.com/s?k=edge+of+tomorrow&ref=nb_sb_noss_2",
   origin: "https://www.amazon.com", protocol: "https:", host: "www.amazon.com", …}
    ▶ancestorOrigins: DOMStringList {length: 0}
    ▶assign: ƒ assign()
     hash: ""
     host: "www.amazon.com"
     hostname: "www.amazon.com"
     href: "https://www.amazon.com/s?k=edge+of+tomorrow&ref=nb_sb_noss_2"
     origin: "https://www.amazon.com"
     pathname: "/s"
     port: ""
     protocol: "https:"
    ▶reload: ƒ reload()
    ▶replace: ƒ replace()
     search: "?k=edge+of+tomorrow&ref=nb_sb_noss_2"
    ▶toString: ƒ toString()
    ▶valueOf: ƒ valueOf()
     Symbol(Symbol.toPrimitive): undefined
    ▶__proto__: Location
```

the URL parts are accessible via the location object

Location - Hash

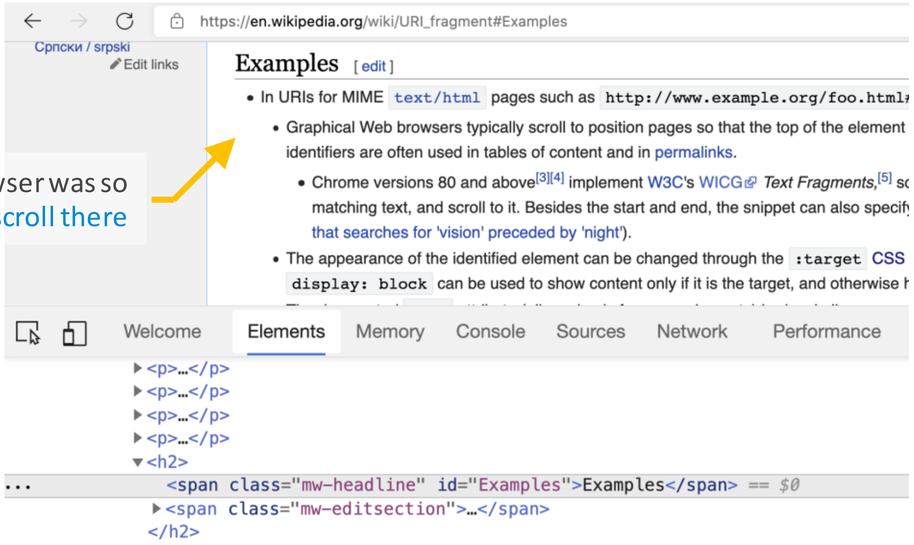
Having a hash value in the URL the browser should scroll to the relevant part of the document

This happens at page reload, but hash can be added in JavaScript as well, and when this code runs, the browser should scroll as well.

hash: #Examples

the browser was so kind to scroll there

the hash refers to an id



https://en.wikipedia.org/wiki/URI_fragment#Examples

Српски / srpski
Edit links

Examples [edit]

- In URIs for MIME `text/html` pages such as `http://www.example.org/foo.html`
  - Graphical Web browsers typically scroll to position pages so that the top of the element identifiers are often used in tables of content and in permalinks.
    - Chrome versions 80 and above[3][4] implement W3C's WICG Text Fragments,[5] s matching text, and scroll to it. Besides the start and end, the snippet can also specif that searches for 'vision' preceded by 'night').
  - The appearance of the identified element can be changed through the `:target` CSS `display: block` can be used to show content only if it is the target, and otherwise h

```
▸<p>…</p>
▸<p>…</p>
▸<p>…</p>
▸<p>…</p>
▾<h2>
...   <span class="mw-headline" id="Examples">Examples</span> == $0
     ▸<span class="mw-editsection">…</span>
  </h2>
```

Location - Hash

Hash seems harmless at the first sight, however, it opens up a whole lot of new possibilities (and bugs)

Sometimes, there is a request from the Product Owner, that the page should scroll to a specified position after the page loading.

While the #hash can be used for that, it is the browser’s concern to decide how and when to process that. Relying on the hash scroll could lead to surprises.

window.scrollTo(x-coord, y-coord) also can be used, but you need to be very careful *when to do that* – the page rendering takes time.

working with a #hash could be a serious task

Hash also can be used for special purposes (e.g., communication between iframes – many times you will need to integrate 3rd party iframes, and while now there are other methods for that, it is still can be used by those iframes)

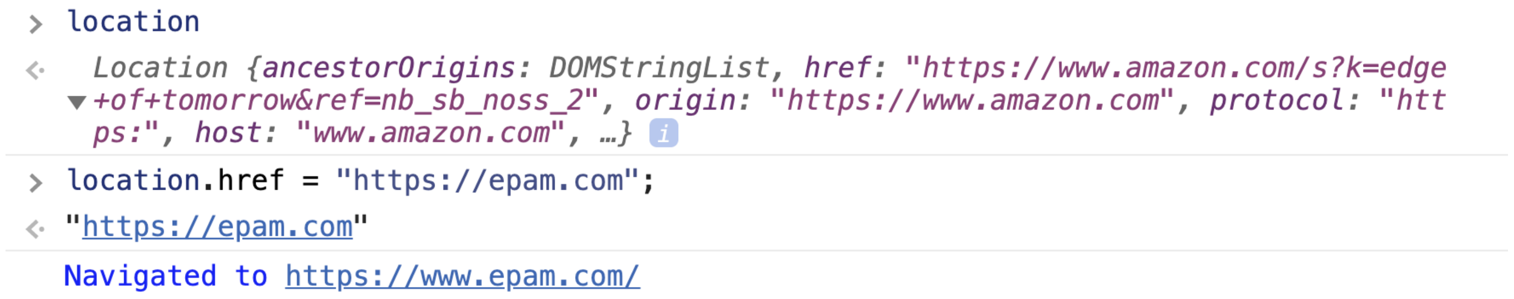
even there is an event for that

```
> window.onhashchange = function(event) {
      console.log("My middle name... is... " + location.hash);
  };

  location.hash="Rose";
< "Rose"
  My middle name... is... #Rose
```

Location - navigate

When setting the location.href, the browser will navigate to a new URL

Navigation could be possible with location.replace() as well - the difference is, that location.href will save the original URL to the browser’s history.



```
> location
<· ▼ Location {ancestorOrigins: DOMStringList, href: "https://www.amazon.com/s?k=edge
     +of+tomorrow&ref=nb_sb_noss_2", origin: "https://www.amazon.com", protocol: "htt
     ps:", host: "www.amazon.com", …} i
> location.href = "https://epam.com";
<· "https://epam.com"
   Navigated to https://www.epam.com/
```

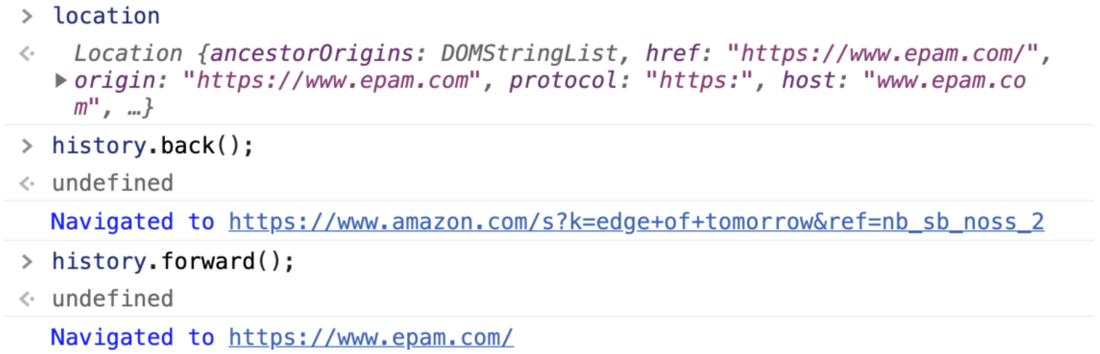
HISTORY

History

The browser history can be modified, also, actions can be performed (back, forward, go)

This is a significant responsibility in Single Page Applications, as the navigation between "pages" in SPAs are not natively supported by browsers (there is only one page from the browser's perspective).

a simple bug in the history management of a web application – no worries, business as usual



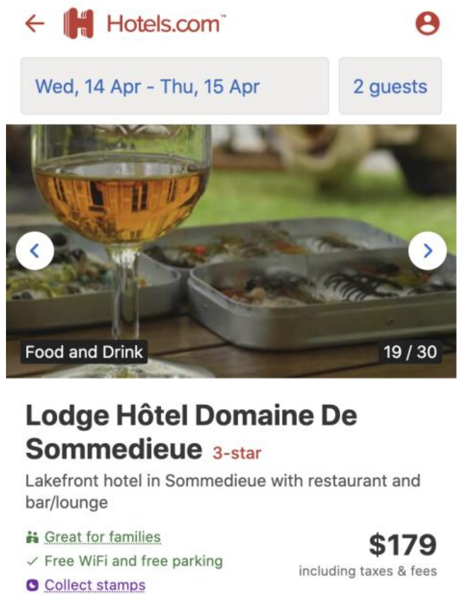
```
> location
<· Location {ancestorOrigins: DOMStringList, href: "https://www.epam.com/",
  ▶origin: "https://www.epam.com", protocol: "https:", host: "www.epam.co
   m", …}
> history.back();
<· undefined
  Navigated to https://www.amazon.com/s?k=edge+of+tomorrow&ref=nb_sb_noss_2
> history.forward();
<· undefined
  Navigated to https://www.epam.com/
```

History

that innocent looking back button can cause a lot of issues, be aware!

Also, mobile web applications tend to have navigation UI elements

Those would need special attention to handle properly, as there are a lot of edge cases there - please test the application thoroughly.



Lodge Hôtel Domaine De Sommedieue 3-star

Lakefront hotel in Sommedieue with restaurant and bar/lounge

Great for families
Free WiFi and free parking
Collect stamps

$179
including taxes & fees

Q&A

edu_hu@epam.com