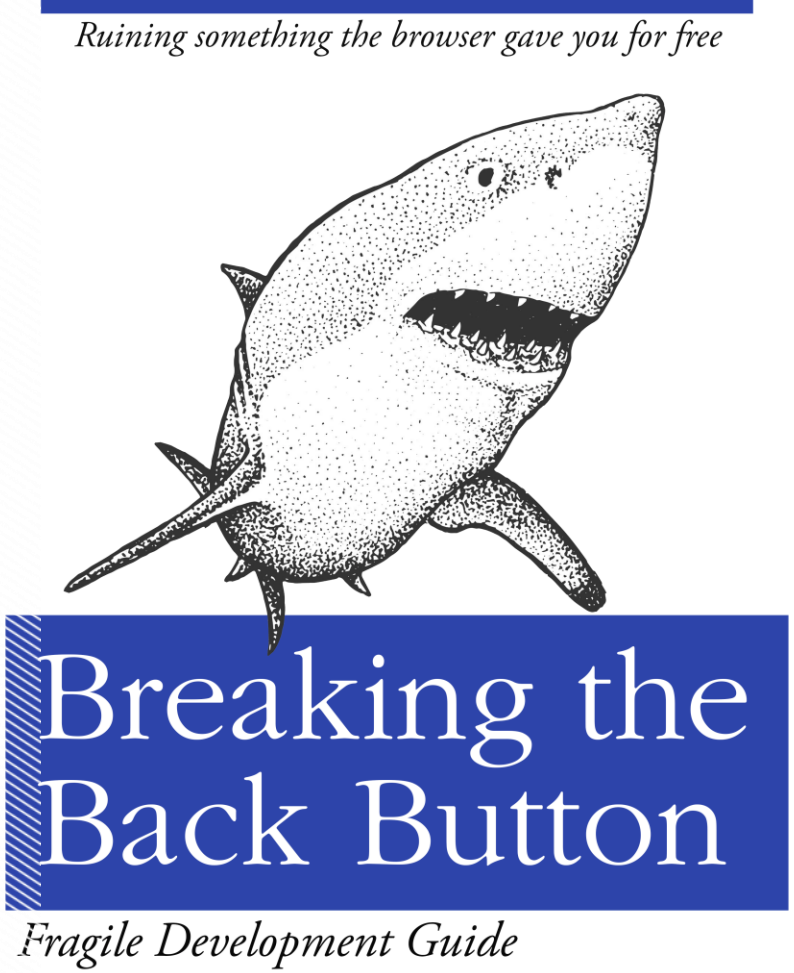
Web API - part I

DOM Events,
Errors,
Storages

Frontend Junior Program - 2022



Agenda

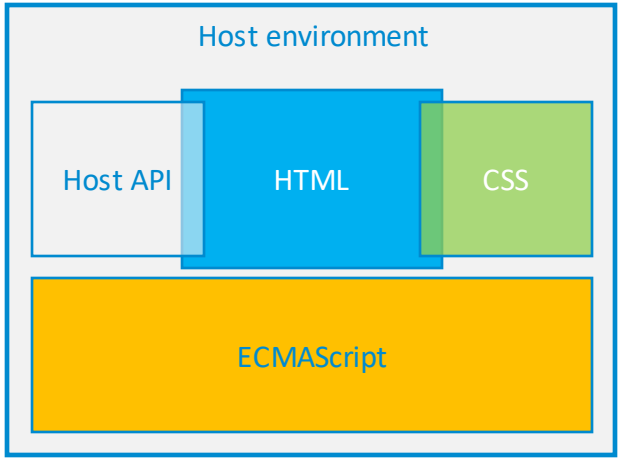
1. Intro
2. Events
3. Event propagation
4. Event delegation
2. Error object
3. Throw an exception
4. Try, Catch, Finally
5. Cookies
6. Local and Session Storages

Introduction

The web uses different technologies and standards. We could be already familiar with HTML, CSS and JavaScript (these are all well defined standards).

However, these are just really the top of the iceberg, and boundaries between these are not always clear.

host: web browser, webview, Node.JS …



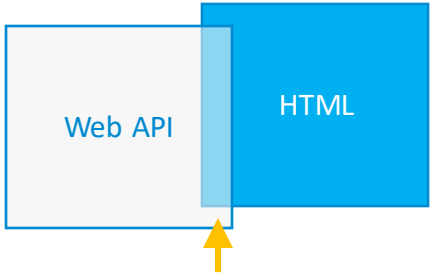
Node.JS lacks of rendering engine, therefore there is no need for HTML, CSS

Console API is part of the browser's Web API

Core JavaScript vs Web API

Usually, we refer JavaScript as a code, written in the JavaScript language. However, there are parts provided by the standard (core JavaScript), yet others are added by the host environment.

In this lecture, we will depart from the core JS and learn about an important parts of the Web API: DOM Events, Cookies and Web storage. Also, we will get familiar with the built-in Error object of the core JavaScript.



Storage is a Web API, however, also a part of the HTML Standard as well.

setTimeout*, however, is not part of the JS core, it is added by the browser.

Error is a built-in object in JavaScript, such as Array, Number, Boolean, etc.

```
> setTimeout(function() {
      throw new Error("A déjà vu is usually a glitch in the Matrix.");
  }, 0);
<· 42
⊗ ▶Uncaught Error: A déjà vu is usually a glitch in the Matrix.  VM7632:2
      at <anonymous>:2:11
```

* as you may guessed, there is a setTimeout in Node.JS, slightly different, though

Host API

It is easier to reason about the host APIs, if we imagine that the browser declares these before running our code, like this:

```
> window.setTimeout = function() { ... }
  window.localStorage = { ... }
  window.sessionStorage = { ... }
  window.console = { ... };
  ...

  // here comes your code
```

This will be very important when you try to write unit tests, because those are running in Node.JS, therefore the browser API needs to be emulated to be able to understand your code.

```
> global.setTimeout
[Function: setTimeout] {
  [Symbol(nodejs.util.promisify.custom)]: [Function]
}
> global.sessionStorage
undefined
>
```

in Node.JS, setTimeout added to the global object, but there is no support for the storage API

EVENTS

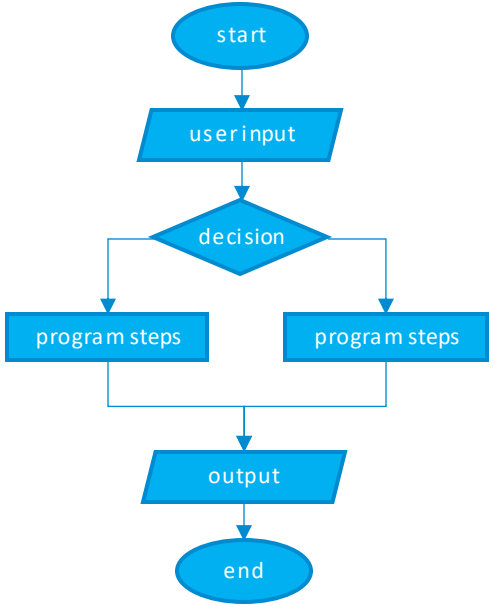
Program flow

What are events and why do we need them?

The real question is why would we need anything other than events, as programming is, in its essence, about event management.

This, however, is not obvious at first, if you consider programs, like this:

Usually, however, we develop different kind of programs…



a program flow depicted in programming books

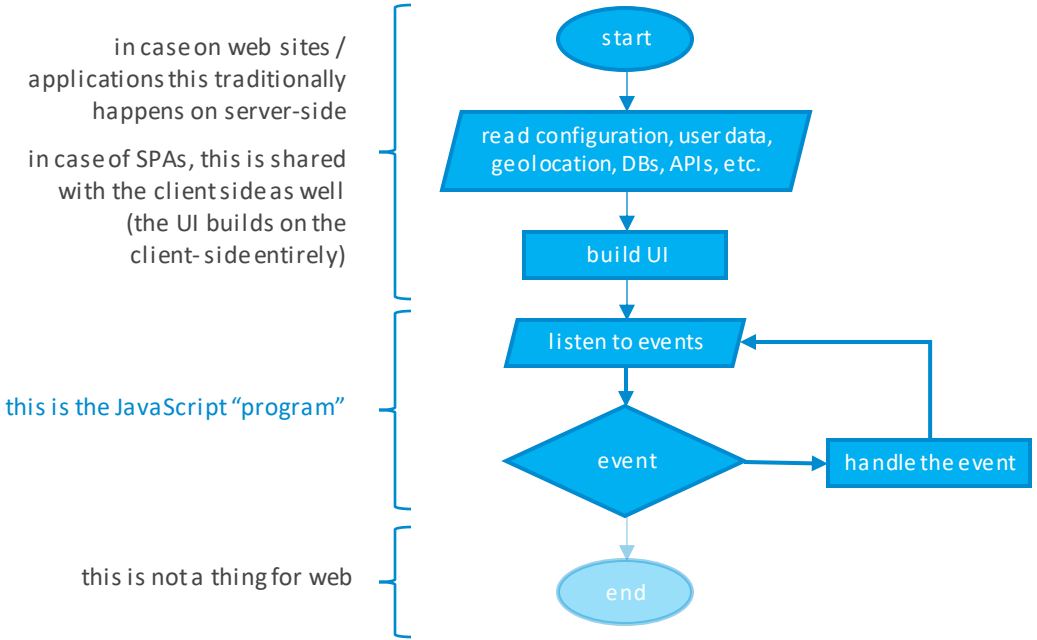
Program flow – application with User Interface

Real-life* applications rely heavily on user input

The application itself is nothing else than configuring the UI components (buttons, input fields, scrollbars) and setting up the event handlers for these.

In past, writing event handlers was *the "programming"* in JavaScript, but with SPAs, the site-build part is also done on client-side.

Still, writing event handlers is a crucial part of the web development.



a general program flow of every app with UI

**we less often write programs for mars rovers or for nuclear power plants*

Events

An event is a signal that something has happened that the application need to react on

Events can be user input / UI events (keyboard, pointer, input field, click, touch, scroll, submit, focus), environmental (history, network, sockets, messaging) or application dispatched events.

Event names starts with “on”.

This snippet returns the available events in your browser:

```
function getEvents(obj, events = []) {
  for (var prop in obj) if (prop.indexOf('on') === 0) events.push(prop);
  return events;
}

function getAllEvents(events = {}) {
  events['window'] = getEvents(window);
  Object.getOwnPropertyNames(window).forEach(prop => {
    try {
      const arr = getEvents(window[prop].prototype);
      events = {...events, ...(arr.length && {[prop]: arr})};
    }
    catch {}
  });

  return events;
}
console.log(getAllEvents());
```

one simply cannot declare a variable in the parameter list in prod code, believe me ;)

this is, however, a pretty common way to add a property conditionally

```
▼{window: Array(104), Option: Array(97), Image: Array(97), Audio: Array(99),
  ▶AbortSignal: ["onabort"]
  ▶Animation: (3) ["onfinish", "oncancel", "onremove"]
  ▶Audio: (99) ["onencrypted", "onwaitingforkey", "onabort", "onblur", "onca
  ▶AudioBufferSourceNode: ["onended"]
  ▶AudioContext: ["onstatechange"]
  ▶AudioScheduledSourceNode: ["onended"]
  ▶AudioWorkletNode: ["onprocessorerror"]
  ▶BackgroundFetchRegistration: ["onprogress"]
  ▶BaseAudioContext: ["onstatechange"]
  ▶BatteryManager: (4) ["onchargingchange", "onchargingtimechange", "ondisch
  ▶BroadcastChannel: (2) ["onmessage", "onmessageerror"]
  ▶CSSAnimation: (3) ["onfinish", "oncancel", "onremove"]
  ▶CSSTransition: (3) ["onfinish", "oncancel", "onremove"]
  ▶CanvasCaptureMediaStreamTrack: (3) ["onmute", "onunmute", "onended"]
  ▶ConstantSourceNode: ["onended"]
  ▶Document: (103) ["onreadystatechange", "onpointerlockchange", "onpointerl
  ▶Element: (9) ["onbeforecopy", "onbeforecut", "onbeforepaste", "onsearch",
  ▶EventSource: (3) ["onopen", "onmessage", "onerror"]
  ▶FileReader: (6) ["onloadstart", "onprogress", "onload", "onabort", "onerr
  ▶HTMLAnchorElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "o
  ▶HTMLAreaElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "onc
  ▶HTMLAudioElement: (99) ["onencrypted", "onwaitingforkey", "onabort", "onb
  ▶HTMLBRElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "oncan
  ▶HTMLBaseElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "onc
  ▶HTMLBodyElement: (113) ["onblur", "onerror", "onfocus", "onload", "onresi
  ▶HTMLButtonElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "o
  ▶HTMLCanvasElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "o
  ▶HTMLDListElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "on
  ▶HTMLDataElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "onc
  ▶HTMLDataListElement: (97) ["onabort", "onblur", "oncancel", "oncanplay",
  ▶HTMLDetailsElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "
  ▶HTMLDialogElement: (97) ["onabort", "onblur", "oncancel", "oncanplay", "o
```

Event handlers

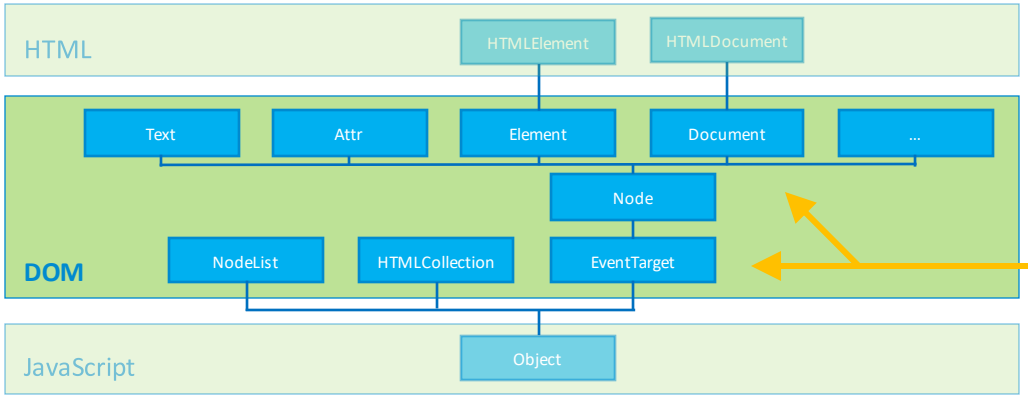
An event handler is a function, which can be assigned to the relevant property on a HTML element

In this lecture, we are primarily focusing on browsers as a host environment. There are events in Node.js as well, but those are different from many aspects – and Node.js definitely does not connected to any DOM. Remember this chart?

With vanilla JS you will never use this method, because your HTML and JavaScript should be separated.

With React, you will almost always use this. ¯_(ツ)_/¯

```
> document.body
<·   <body onclick="console.log('Snap!')"></body>
  Snap!
```



the document object itself is an EventTarget

Event handlers – running in a different time and space

Let me stop here just for a moment – because this *snap* will have significant consequences

The code part that we assign to the event handler, will run later – we don't know exactly when. Also, we don't know that the state we expect (DOM element, data values, user state, etc.) will exist at all when the event finally will be fired.

What we do know, that all these could be completely different, even in a way we are not prepared for.

an event occurs here…

then somewhere in the universe things start to happen…

… so that it will end in an unintended way.

A classic example

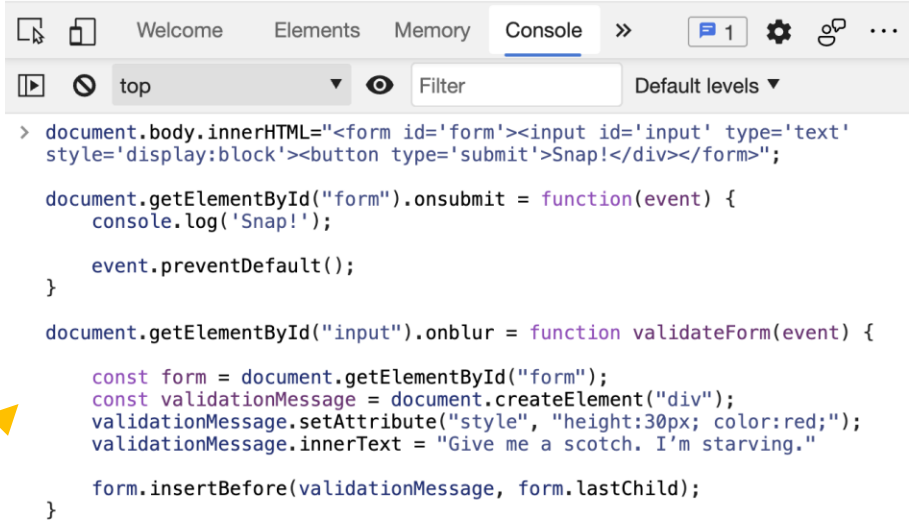
Should you be curious, here you are an example – a form

The expectation is, that when clicking on the submit button, the *onsubmit* event should fire; not necessarily to submit the form, but we probably need some additional check, or logging.

So, the *onsubmit* event should fire no matter what.

Now, guess what will happen if we write something into the input field, then click on the “Snap!” button?

a form, an input field and a button – nothing special…



… also, we have a validation message, which triggers on the onblur event of the input field – again, pretty usual

A classic example - the result

What happened?

Well, the *onsubmit* event did not fire; why?
Let's go through step-by-step:

1., we click the "Snap!" button

2., but before that, we actually leave the input field, so the *onblur* event fires first

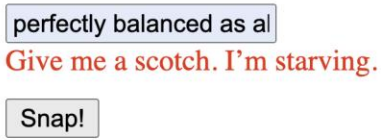
3., we show a validation message

4., but the message pushes down the button

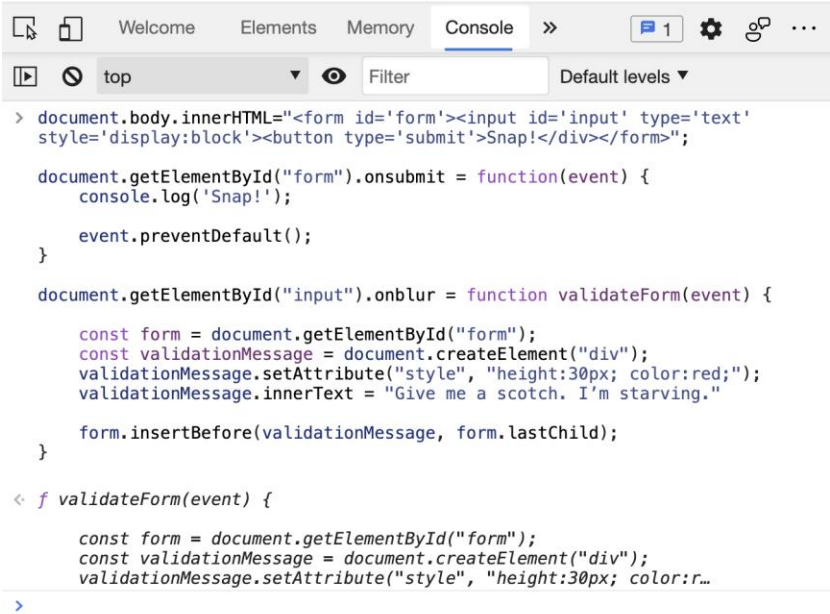
5., so when the click would arrive to the button

6., the button is not there anymore

7., so there is no click, there is no *onsubmit* event



now we have a validation message, but that is expected



Let’s analyse this!

the onclick attribute...

will be set as property...

therefore we can overwrite it

```
> document.body
<- <body onclick="console.log('Snap!')"></body>
> document.body.onclick
<- ƒ onclick(event) {
console.log('Snap!')
}
> document.body.onclick = function() {
  console.log('Another snap!');
}
<- ƒ () {
  console.log('Another snap!');
}
Another snap!
```

Not this way

the onclick property…

…won't be synchronized with the onclick attribute. It is not obvious, because many properties (e.g. id) will.

```
> document.body
<  <body></body>
> document.body.onclick = function() {
    console.log("snap!");
  }
  document.body.id = "thanos";
<  "thanos"
  snap!
> document.body
<  <body id="thanos"></body>
```

Event object parameter

Events receives an event object as a parameter

This object's properties provide useful information about the event.

here we have a MouseEvent

```
> document.body.onclick = function(event) {
    console.log(event);
  }
<· ƒ (event) {
    console.log(event);
  }
▼MouseEvent {isTrusted: true, screenX: 762, screenY: 294, clientX: 223, clientY: 156, …}
    altKey: false
    bubbles: true
    button: 0
    buttons: 0
    cancelBubble: false
    cancelable: true
    clientX: 223
    clientY: 156
```

Event interfaces

The browser provide several event interfaces available through objects

These events are browser dependent, several of them available only in a particular browser.

this snippet returns the available event objects of your browser:

```
Object.getOwnPropertyNames(window)
  .filter(e => window[e] && Object.getPrototypeOf(window[e]).prototype === Event.prototype)
  .sort();
```

AnimationEvent
AudioProcessingEvent
BeforeInputEvent
BeforeUnloadEvent
BlobEvent
ClipboardEvent
CloseEvent
CompositionEvent
CSSFontFaceLoadEvent
CustomEvent
DeviceLightEvent
DeviceMotionEvent
DeviceOrientationEvent
DeviceProximityEvent
DOMTransactionEvent
DragEvent
EditingBeforeInputEvent
ErrorEvent
FetchEvent
FocusEvent
GamepadEvent
HashChangeEvent
IDBVersionChangeEvent
InputEvent
KeyboardEvent
MediaStreamEvent
MessageEvent
MouseEvent
MutationEvent
OfflineAudioCompletionEvent
OverconstrainedError
PageTransitionEvent
PaymentRequestUpdateEvent
PointerEvent
PopStateEvent
ProgressEvent
RelatedEvent
RTCDataChannelEvent
RTCIdentityErrorEvent
RTCIdentityEvent
RTCPeerConnectionIceEvent
SensorEvent
StorageEvent
SVGEvent
SVGZoomEvent
TimeEvent
TouchEvent
TrackEvent
TransitionEvent
UIEvent
UserProximityEvent
WebGLContextEvent
WheelEvent

.addEventListener

Event handlers can be added with .addEventListener as well

With this method, multiple handlers can be registered.

not onclick, just click!

```
> document.body.addEventListener("click", function() {
    console.log('Thanos: Snap!');
  });
<· undefined
> document.body.addEventListener("click", function() {
    console.log('Iron man: Hold my beer...');
  });
<· undefined
  Thanos: Snap!
  Iron man: Hold my beer...
```

.removeEventListener

Event handlers can be removed as well

it is cleaner to define a handler function separately and register that

```
> const thanosSnapHandler = function() {
    console.log("Snap!");
  };

  document.body.addEventListener("click", thanosSnapHandler);
<· undefined
  Snap!
> const ironManSnapHandler = function() {
    console.log("Hold my beer...");
  };

  document.body.addEventListener("click", ironManSnapHandler);

  document.body.removeEventListener("click", thanosSnapHandler);
<· undefined
  Hold my beer...
```

.preventDefault

The browser's default event can be prevented

this won't check now

Thanos Snap! ☐

.preventDefault()

Welcome | Elements | Memory | Console | Sources | Network | Performance

top | Filter | Default levels

```
> document.body
  ▼<body>
    ▼<form>
       <label for="id-checkbox">Thanos Snap!</label>
       <input type="checkbox" id="id-checkbox">
     </form>
   </body>

> document.querySelector("#id-checkbox").addEventListener("click", function(event) {
    console.log("Part Of The Journey Is The End.");
    event.preventDefault();
  });
  undefined
  Part Of The Journey Is The End.
```

EVENT PROPAGATION

Events are bubbling

Events are bubbling (by default)

If event handlers were registered for the same event for both the parent and the child, then both handler will run: first the child, then the parent.

This will continue, until the browser will reach the root element.

we still don't use innerHTML in prod

```
> document.body.innerHTML="<div style='height:100%'></div>";

  const thanosSnapHandler = function() {
    console.log("Snap!");
  };

  const ironManSnapHandler = function() {
    console.log("Hold my beer...");
  };

  document.getElementsByTagName("div")[0].addEventListener("click", thanosSnapHandler);
  document.body.addEventListener("click", ironManSnapHandler);
  document.body;

<·  ▼<body>
      <div style="height:100%"></div>
    </body>

  Snap!

  Hold my beer...
```

first the child's then the parent's handler run: this is the event bubbling

Events can be capturing as well

There is an opposite mechanism as well: the capturing

By setting the third parameter of the addEventListener, the process can be reversed: first the parent's handler will run, and it will continue, until we reach the target element.

```
> document.body.innerHTML="<div style='height:100%'></div>";

  const thanosSnapHandler = function() {
    console.log("Snap!");
  };

  const ironManSnapHandler = function() {
    console.log("Hold my beer...");
  };

  document.getElementsByTagName("div")[0].addEventListener("click", thanosSnapHandler, true);
  document.body.addEventListener("click", ironManSnapHandler, true);
  document.body;

<·  ▼<body>
      <div style="height:100%"></div>
    </body>
  Hold my beer...
  Snap!
```

the useCapture parameter

when the parent's handler run first
that is the capturing

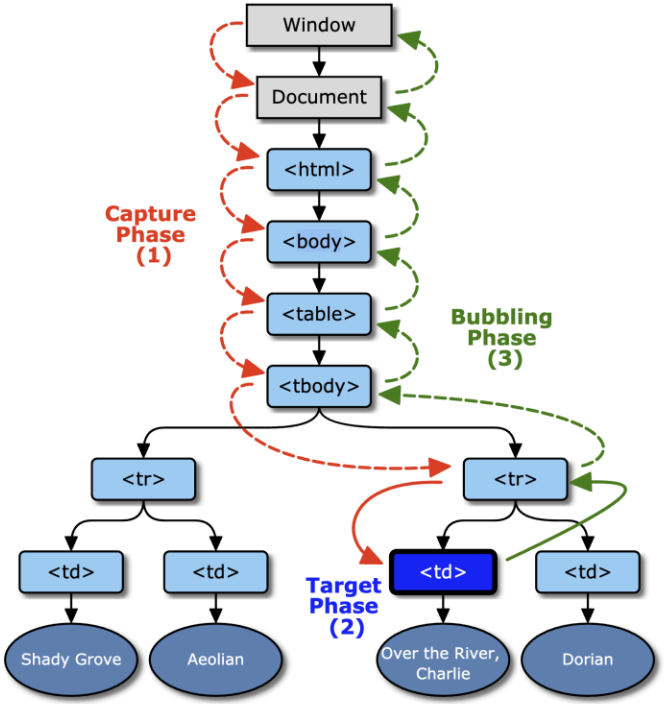
Event phases - bubbling, capturing, target

There are 3 different event phases defined in the Standard

The capture phase: The event object propagates through the target's ancestors from the Window to the target's parent.

The target phase: The event object arrives at the event object's event target. This phase is also known as the at-target phase. If the event type indicates that the event doesn't bubble, then the event object will halt after completion of this phase.

The bubble phase: The event object propagates through the target's ancestors in reverse order, starting with the target's parent and ending with the Window. This phase is also known as the bubbling phase.



EVENT DELEGATION

Event delagation

Because events will be propagated to the parent, we can have the event handler exlusively on the parent

This could be very useful when you have several, similar elements in the parent. In fact, sometimes you must use this: for example, you have a google map with many points to click - let's say 100 for a city.

Just imagine, what will happen, when the user starts to zoom out… Soon, the browser must deal with thousands of event handlers (don't guess, it will crush), and you must deal with a bug ticket.

```
> document.body.innerHTML="<div id='target-div' style='height:100%'></div>";

  const parentClickHandler = function({type, bubbles, defaultPrevented, targe
    console.log({type, bubbles, defaultPrevented, target, path});
  };

  document.body.addEventListener("click", parentClickHandler);
  document.body;

<·  ▼<body>
      <div id="target-div" style="height:100%"></div>
    </body>

  ▼{type: "click", bubbles: true, defaultPrevented: false, target: div#targe
    bubbles: true
    defaultPrevented: false
    ▶path: (5) [div#target-div, body, html, document, Window]
    ▶target: div#target-div
    type: "click"
    ▶__proto__: Object
```

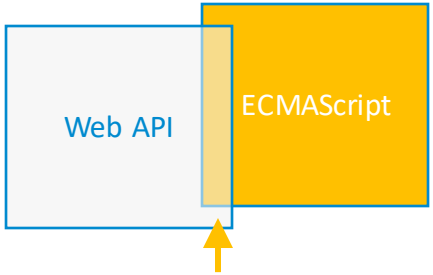
with event delegation we have to know the source of the event

Error OBJECT

Error object

Error is an object

As you may already get used to it, important parts of the core JavaScript can be accessed via built-in objects. Error is one of them.



While Error is a built-in object, defined by the ECMA Standard, implementations are slightly in different browsers.

Error object

When you see an error in a console, that is essentially the result of:

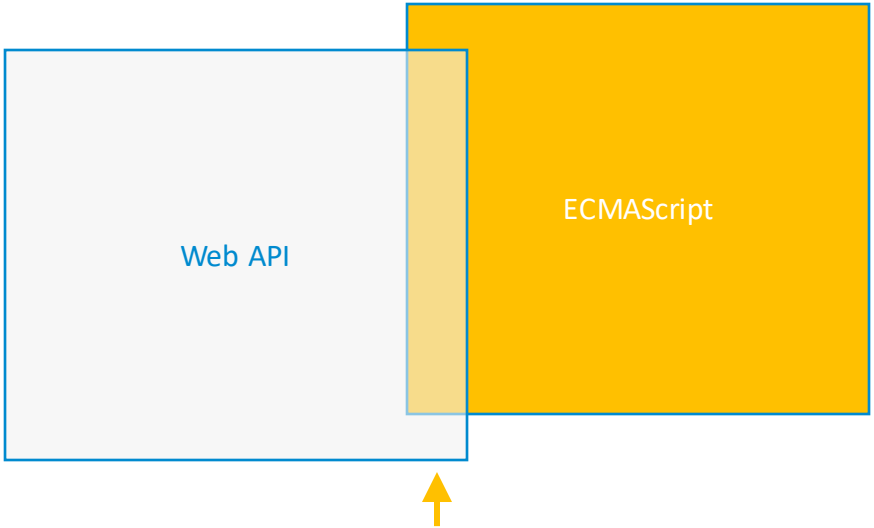
- creating a new error object instance
- throwing that
- (and not catching the error)

```
> let neo = {
      contacts: {
          rhineheart: {},
      }
  }

  // later...

  neo.contacts.morpheus.talksTo();
```

```
Uncaught TypeError: Cannot read property 'talksTo' of undefined
    at <anonymous>:9:23
```

nah, not yet...

Browser compatibility



While Error is a built-in object, defined by the ECMA Standard, implementations are slightly different in browsers.

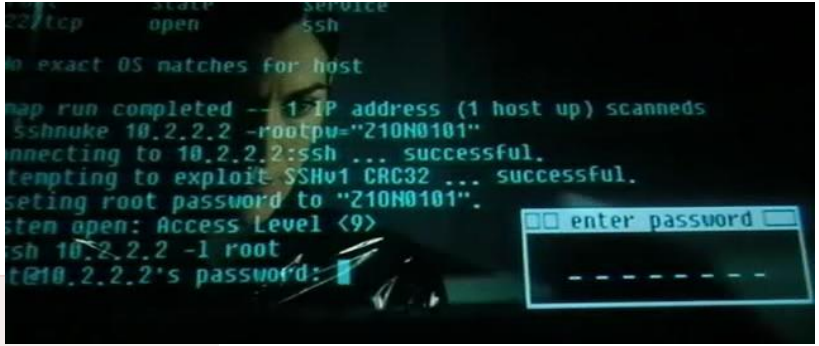
And differences in browser implementations are always an infinite source of lot of ~~fun~~ nightmare in web development.

Error types

There are several types of predefined error objects exist, and a custom error could be created, as well.

RangeError	*a numeric variable or parameter is outside of its valid range*
ReferenceError	*referencing an invalid reference*
SyntaxError	*represents a syntax error*
TypeError	*a variable or parameter is not of a valid type*
URIError	*encodeURI() or decodeURI() are passed invalid parameters*
AggregateError	*represents several errors wrapped in a single error when multiple errors need to be reported by an operation (e.g., Promise.any()).*

Examples




```
> Array(-1)
✕ ▶Uncaught RangeError: Invalid array length
      at <anonymous>:1:1
> equilibrium++;
✕ ▶Uncaught ReferenceError: equilibrium is not defined
      at <anonymous>:1:1
> Wake up Samurai, the Matrix is everywhere. It is all around us.
✕ Uncaught SyntaxError: Unexpected identifier
> ({ choice: { is: { an: "illusion" } } }).choice.is.not.illusion;
✕ ▶Uncaught TypeError: Cannot read property 'illusion' of undefined
      at <anonymous>:1:56
> decodeURIComponent('%root#10.2.2.2');
✕ ▶Uncaught URIError: URI malformed
      at decodeURIComponent (<anonymous>)
      at <anonymous>:1:1
```

THROW AN EXCEPTION

How to create ~~bugs~~ errors

Error occurs in predefined cases, however, errors can be triggered programmatically, too.

All you need is to create a new instance of your choice of error objects:

the error constructors return the error object even when called without *new*, so it can be used both ways

```
> let pillColor = "yellow";

  switch (pillColor) {
      case "red":
          message = "you wake up in your bed and believe whatever you want to believe.";
          break;
      case "blue":
          message = "you stay in Wonderland and I show you how deep the rabbit hole goes.";
          break;
      default:
          throw new RangeError("Remember, all I'm offering is the truth, nothing more");
  }

  ▶Uncaught RangeError: Remember, all I'm offering is the truth, nothing more
      at <anonymous>:11:15
```

Throwing an exception

That being said, we can throw an exception without an error object as well.

Execution will stop (the statements after throw won't be executed), and control will be passed to the first catch block in the call stack. If no catch block exists among caller functions, the program will terminate.

```
throw {
    name: "ChickenError",
    message: "I can't do this"
}
```

the throw value can be any expression

```
> (function goToTheScaffold() {
    let whiteRabbit = "\u{0001F407}";

    throw "I can't do this";

    console.log(whiteRabbit);
  })();

  console.log("Welcome to the real world");
✕ ▶Uncaught I can't do this
```

this will not run

nor this

no rabbit, no real world :/

let's have a catch block then…

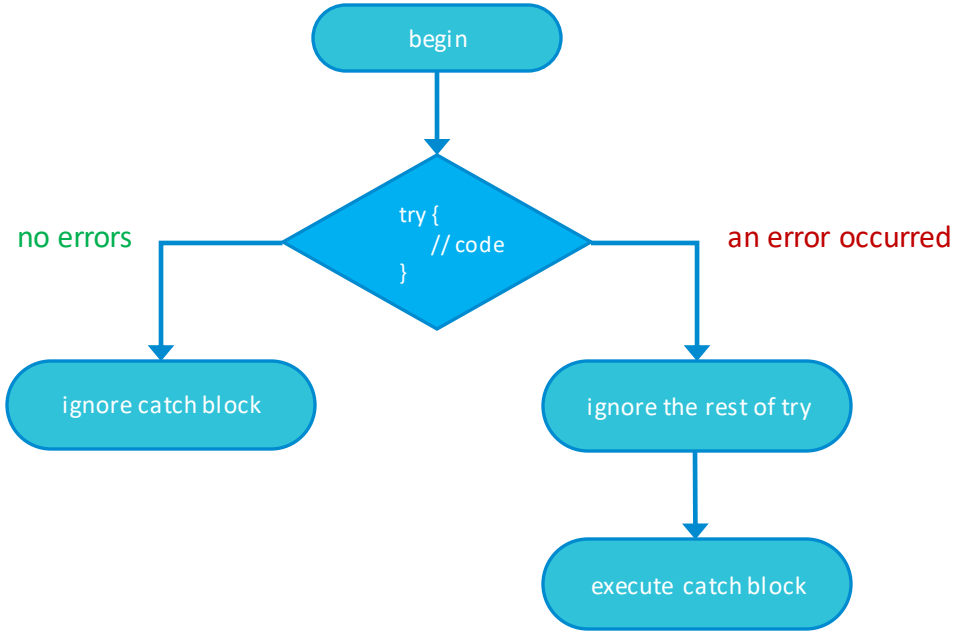
TRY, CATCH, FINALLY

Try ... Catch

Throwing errors is just the half of the story

An uncaught error leads to less user-friendly error message in the console, and generally is a sign that the developer team left some loose ends.

The mechanism for handling errors is the try-catch.



MDN: try...catch

Try … Catch

The try … catch block allows to ”catch” errors, and instead of terminating, do something more reasonable.

we have an exception here

… so this won’t run

but we handle it

… so we won’t miss the important part

```
> try {
    (function goToTheScaffold() {
        let whiteRabbit = "\u{0001F407}";

        throw "I can't do this";

        console.log(whiteRabbit);
    })();
  }
  catch {
    console.log("Trinity removes the bug");
  }

  console.log("Welcome to the real world");
```

```
Trinity removes the bug
Welcome to the real world
```

finally…

Try … Catch … Finally

We could also have a finally block, which will run in either case.

```
> try {
      let whiteRabbit = "\u{0001F407}";

      throw "I can't do this";

      console.log(whiteRabbit);
  }
  catch {
      console.log("Trinity removes the bug");
  }
  finally {
      console.log("Got the Red pill");
  }

  console.log("Welcome to the real world");
  Trinity removes the bug
  Got the Red pill
  Welcome to the real world
<· undefined
```

we could throw, or not

the *finally* block runs anyway

```
> try {
      let whiteRabbit = "\u{0001F407}";

      // throw "I can't do this";

      console.log(whiteRabbit);
  }
  catch {
      console.log("Trinity removes the bug");
  }
  finally {
      console.log("Got the Red pill");
  }

  console.log("Welcome to the real world");
  🐇
  Got the Red pill
  Welcome to the real world
<· undefined
```

so why do we need for an error object then?…

Differential error handling

Errors being an object is useful, because we can handle different errors in different ways.

Also, an error object can provide useful information for debugging in its properties.

also, movieMismatchErrors

```
> let chickenError = {
      name: "ChickenError",
      message: "I can't do this"
  }

  try {
      let whiteRabbit = "\u{0001F407}";

      throw chickenError;

      console.log(whiteRabbit);
  }

  catch (error) {
      switch (error.name) {
          case "RangeError":
              console.log("Got the wrong pill");
              break;
          case "ChickenError":
              console.log("What's wrong McFly? Chicken!");
              break;
          default:
              console.log("Houston, we have a problem");
      }
  }

  What's wrong McFly? Chicken!
```

Do we use try … catch for error handling?

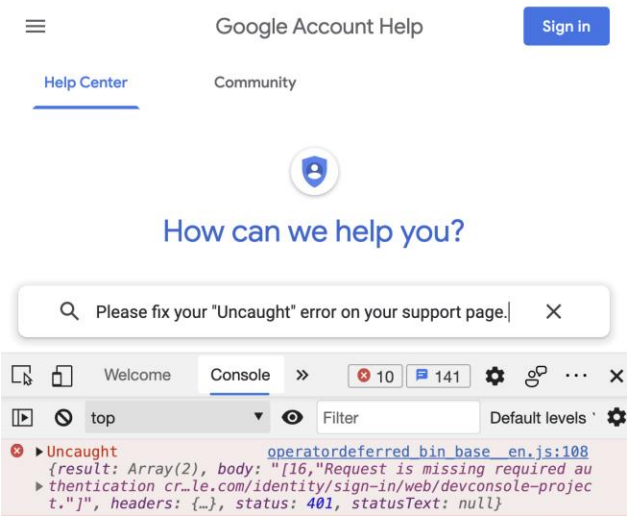
Depends on the project. Generally speaking, *error handling* is one of the most complex areas, and different projects require different approaches.

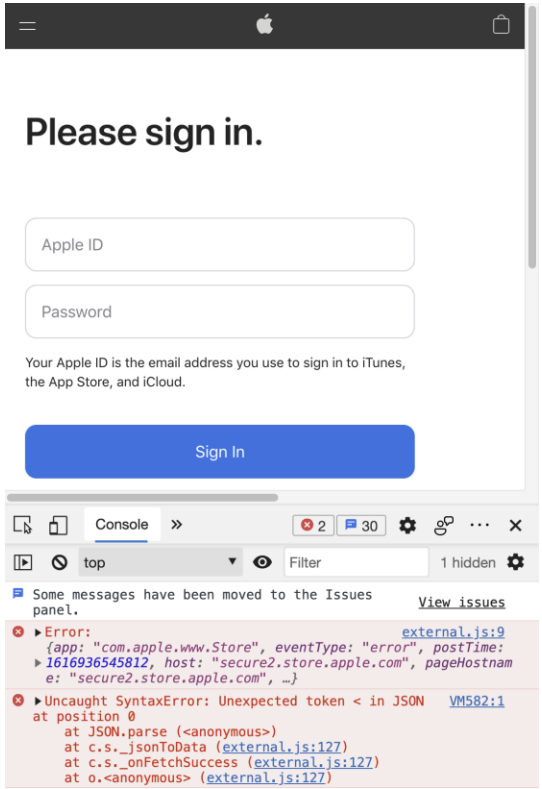
That being said, having errors on the console reflects less competent engineering than expected.

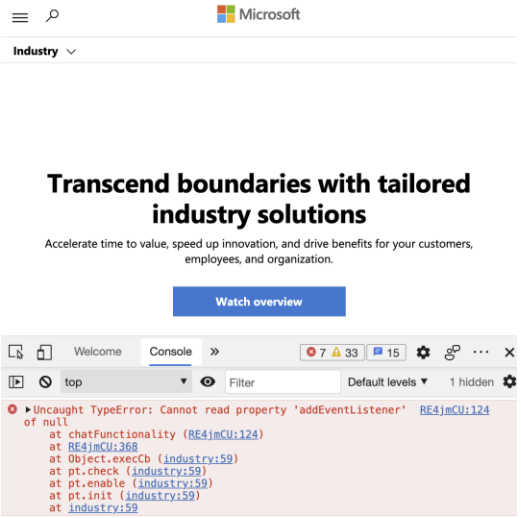
Interestingly, relying on TypeScript (false) safety in runtime could lead to more TypeErrors if backend data is not validated properly (because you probably won't *check?.the?.existence?.of?.every?.property).*

No worries, though, *you will have errors* on console (see the next slide), and if you are prepared for that, you can handle as well.

It happens with the best…







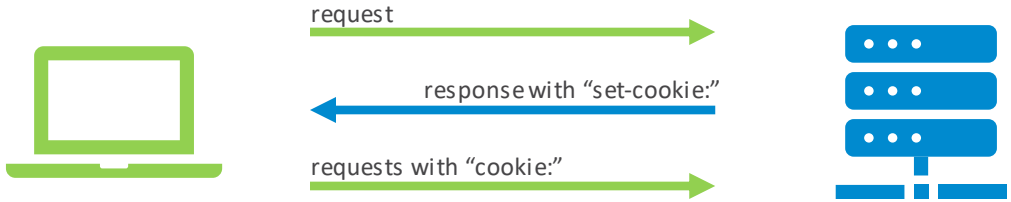
COOKIES

Cookies

Cookie is a string containing a semicolon-separated list of *key = pair* values

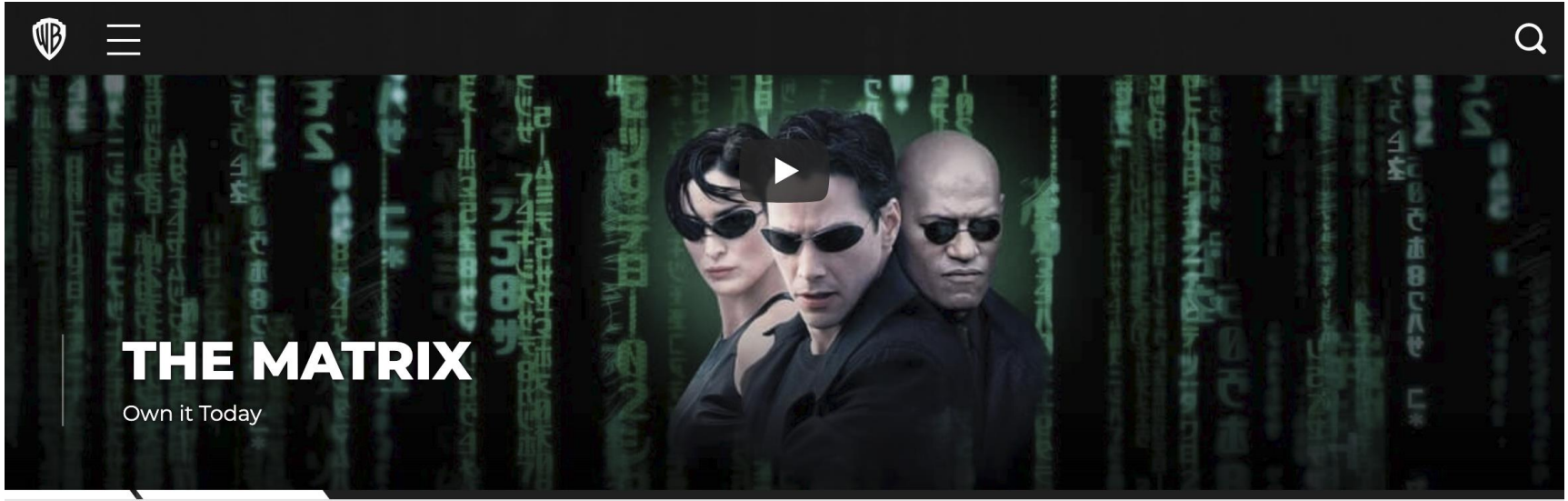
How do cookies work? Cookies are transferred in the http header (you can check at the network tab) between the server and the client. After a cookie was set, it will be stored in the browser for a specified time. The cookie then will be sent to server with *every* subsequent requests, even with requests for images*.

mr_anderson=neo; neo=not_the_one;



**because of this, cookies can be used for tricky activities, such as analyzing website traffic, communication between iframes, etc.*

MDN: Cookies



Welcome | Elements | Memory | Console | Sources | Network | Performance | Application | Security | Lighthouse | AdBlock | 82

Application
- Manifest
- Service Workers
- Storage

Storage
- Local Storage
- Session Storage
- IndexedDB
- Web SQL
- Cookies
  - https://www.warnerbros.com

Filter | Only show cookies with an issue

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Priority
mr_smith	application error	www.warnerbros.com	/	Session	25				Mediun
mr_anderson	neo	www.warnerbros.com	/	Session	14				Mediun

don't believe your eye (#1): a cookie value cannot contain whitespaces

Cookie - attributes

Different attributes
can be set for a cookie:

;path=*path*
;domain=*domain*
;max-age=*max-age-in-seconds*
;expires=*date-in-GMTString-format*
;secure
;httponly
;samesite

In case you wondered: there is a standard for cookies – here is the new, the draft version, signed by Google and Apple:

Workgroup:	HTTP	
Internet-Draft:	draft-ietf-httpbis-rfc6265bis-07	
Obsoletes:	6265 (if approved)	
Published:	7 December 2020	
Intended Status:	Standards Track	
Expires:	10 June 2021	
Authors:	M. West, Ed. *Google, Inc*	J. Wilander, Ed. *Apple, Inc*

Cookies: HTTP State Management Mechanism

Cookie – accessing from JavaScript

Cookies can be get/set with JavaScript as well (except for cookies with *;httponly* attributes) - one cookie at once, only: it's not a data property, it's an accessor (getter/setter).

An assignment to it is treated specially.

```
> document.cookie="WMF-Last-Access-Global=Because as we both know, without
  purpose, we would not exist.; domain=.wikipedia.org; path=/";
<· "WMF-Last-Access-Global=Because as we both know, without purpose, we would
  not exist.; domain=.wikipedia.org; path=/"
```

don't believe your eye (#2): it is the value of the assignment operator, not the new value of the cookie: it is an *httponly* cookie on wikipedia, so cannot be set; also whitespaces.

LOCAL AND SESSION STORAGES

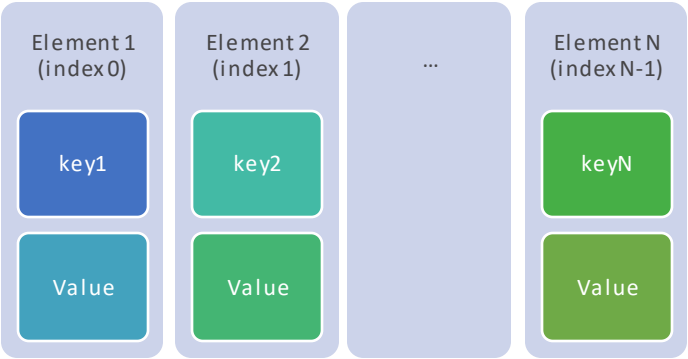
Web storage

The Web Storage API provides mechanisms by which browsers can securely store key/value pairs.

Storage elements are simple key-value stores (strings).

2 different storages exist:

- sessionStorage a separate storage area for the duration of the page session, including page reloads.
- localStorage does the same thing but persists even when the browser is closed and reopened.



MDN: Web storage API

Storage – properties

Method	Description
length	The number of elements in the storage
key(n)	Returns the name of the nth key in the storage
getItem(keyname)	Returns the value of the specified key name
setItem(keyname, value)	Adds that key to the storage, or updates that key's value if it already exists. Also, it will throw an exception if the size quota is reached.
removeItem(keyname)	Removes that key from the storage
clear()	Empties all key out of the storage

Storage - examples

```
> const storageSupportedStr = window.sessionStorage && window.localStorage ? "is" : "is not";
  `Storage API ${storageSupportedStr} supported`;
<· "Storage API is supported"
```

```
> localStorage.setItem("We need guns", "Lots of guns");
<· undefined
> Array.from({length:localStorage.length}, function(_,i) { return localStorage.key(i)})
<· ▶ (10) ["as_tex", "ls-opt-out", "as-fcs1", "We need guns", "ac-storage-ac-store-cache",
    "as-fcs2", "mk_epub_expiry", "test", "fl_products_507829", "mk_epub"]
> localStorage.getItem("We need guns");
<· "Lots of guns"
> localStorage.removeItem("We need guns");
<· undefined
> localStorage.getItem("We need guns");
<· null
> localStorage.clear();
<· undefined
> Array.from({length:localStorage.length}, function(_,i) { return localStorage.key(i)})
<· ▶ []
```

Storage – storage event

Also, there is a special event for storages, it's called "storage".

This storage event is triggered each time a value in a storage is modified from another page.

```
> window.addEventListener(
      'storage',
      function({ url, key, newValue: value }) {
          console.log(`'${key}' => '${value}' on '${url}'`);
      }
  );
<· undefined
  'Neo' => 'I know Kung Fu!' on 'https://www.epam.com/' VM424:4
```

different browser window / tab

```
> localStorage.setItem('Neo', 'I know Kung Fu!');
<· undefined
>
```

Cookies vs Storage API

Cookie	localStorage / sessionStorage
~ 4 kB	~ 5 MB
data lives until the end of expiration date / until it is deleted	data lives until deleted (sessionStorage: until the end of the session)
no event	storage event
document.cookie setter/getter	methods for writing and reading
automatically sent to the server	-
does not rely on JavaScript	requires JavaScript

Q&A

edu_hu@epam.com