JavaScript

Table of Contents

[1. Introduction to JavaScript for Developers 2](#_Toc71875914)

[2. JavaScript Objects and Prototypes In-depth 2](#_Toc71875915)

[3. JavaScript Scopes and Closures In-depth 2](#_Toc71875916)

[4. AJAX 3](#_Toc71875917)

[5. Common Ways to Send Http Requests 4](#_Toc71875918)

[5.1. XmlHttpRequest 4](#_Toc71875919)

[5.2. Fetch-API 4](#_Toc71875920)

[5.3. Axios 4](#_Toc71875921)

[6. JQuery 5](#_Toc71875922)

[7. Media Queries, Responsive Web Design 5](#_Toc71875923)

[8. HTML 5 Web Storage 6](#_Toc71875924)

[9. XML Namespaces 7](#_Toc71875925)

[10. Identifying a Browser and communication between Tabs 7](#_Toc71875926)

[11. Map, reduce, filter … 7](#_Toc71875927)

[12. JavaScript Event Loop 8](#_Toc71875928)

[13. Asynchronous JavaScript 9](#_Toc71875929)

[13.1. Callbacks 9](#_Toc71875930)

[13.2. Promises 11](#_Toc71875931)

[1.1.1 Chained Promises 13](#_Toc71875932)

[1.1.2 Creating a Promise 14](#_Toc71875933)

[1.1.3 Promises versus callbacks 15](#_Toc71875934)

[13.3. Async-Await 16](#_Toc71875935)

[14. Moment 16](#_Toc71875936)

[1.1.4 Asd 16](#_Toc71875937)

Resources used:

* JavaScript for Developers – Java Brains: <https://www.youtube.com/watch?v=s6R0VEdoVt4&list=PLqq-6Pq4lTTYFJxC9NLJ7dSTI5Z1WWB6K>

JavaScript Basics

JavaScript is a blocking, … programming language

1. Introduction to JavaScript for Developers
2. JavaScript Objects and Prototypes In-depth

<https://www.youtube.com/watch?v=fBpPfPjxOhc&list=PLqq-6Pq4lTTaflXUL0v3TSm86nodn0c_u>

1. JavaScript Scopes and Closures In-depth

<https://www.youtube.com/watch?v=O312eN5J2bc&list=PLqq-6Pq4lTTZ_LyvzfrndUOkIvOF4y-_c>

TODOs

* Js, how to return from a for loop
* Learn more about the window object
* Firebase promise video, promises work with only the traditional for loop?

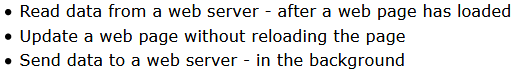
JavaScript Concepts

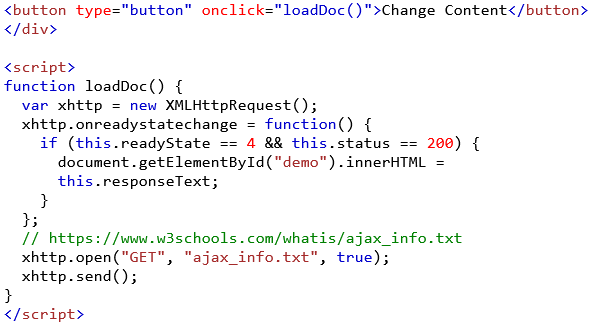
1. AJAX

Ajax (also AJAX, short for asynchronous JavaScript and XML) is a set of web development techniques using many web technologies on the client side to create asynchronous web applications. With Ajax, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows web pages and, by extension, web applications, to change content dynamically without the need to reload the entire page. In practice, modern implementations commonly utilize JSON instead of XML.

Ajax is not a single technology, but rather a group of technologies. HTML and CSS can be used in combination to mark up and style information. The webpage can then be modified by JavaScript to dynamically display—and allow the user to interact with—the new information. The **built-in XMLHttpRequest object**, **or since 2017 the new** "**fetch()**" function within JavaScript is commonly used to execute Ajax on webpages allowing websites to load content onto the screen without refreshing the page. Ajax is not a new technology, or different language, just existing technologies used in new ways.

AJAX first appeared in 1999. In the **early-to-mid 1990s**, most Web sites were based on complete HTML pages. Each user action required that a completely new page be loaded from the server. This process was inefficient, as reflected by the user experience: all page content disappeared, then the new page appeared. Each time the browser reloaded a page because of a partial change, all of the content had to be re-sent, even though only some of the information had changed. This placed additional load on the server and made bandwidth a limiting factor on performance.





1. Common Ways to Send Http Requests
   1. XmlHttpRequest

TODO: <https://www.youtube.com/watch?v=4K33w-0-p2c&t=3s>

* 1. Fetch-API

TODO: <https://www.youtube.com/watch?v=23hrM4saaMk>

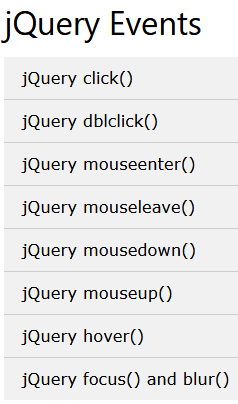
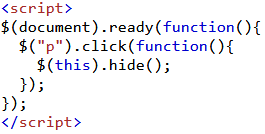
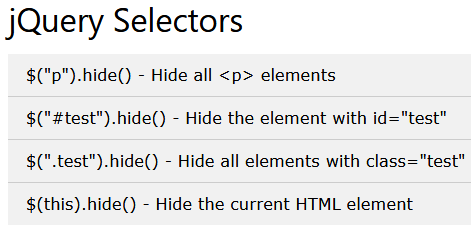
* 1. Axios

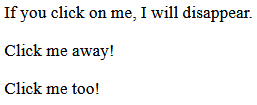
TODO: <https://www.youtube.com/watch?v=qM4G1Ai2ZpE>

1. JQuery

jQuery is a JavaScript library designed to simplify HTML DOM tree traversal and manipulation, as well as event handling, CSS animation, and Ajax. It is free, open-source software using the permissive MIT License. As of May 2019, jQuery is used by **73%** of the **10 million** most popular websites.

jQuery greatly simplifies JavaScript programming. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications. jQuery also provides capabilities for developers to create plug-ins on top of the JavaScript library. This enables developers to create abstractions for low-level interaction and animation, advanced effects and high-level, themeable widgets.





There are many many more functions.

1. Media Queries, Responsive Web Design

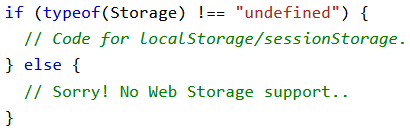
**TODO: srcsets**

1. HTML 5 Web Storage

With web storage, web applications can store data locally within the user's browser.

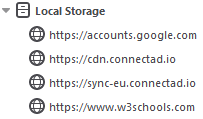
**Before HTML5**, application data had to be stored in cookies, included in every server request. Web storage is more secure, and large amounts of data can be stored locally, without affecting website performance.

Unlike cookies, the storage limit is far larger (at least 5MB) and information is never transferred to the server. Web storage is **per origin (per domain and protocol)**. All pages, from one origin, can store and access the same data.

Before using web storage, check browser support for localStorage and sessionStorage.

**HTML Web Storage Objects**

HTML web storage provides two objects for storing data on the client:

**window.localStorage** (shared across tabs(browser sessions)): The localStorage object stores the data with no expiration date. The data will not be deleted when the browser is closed, and will be available the next day, week, or year.



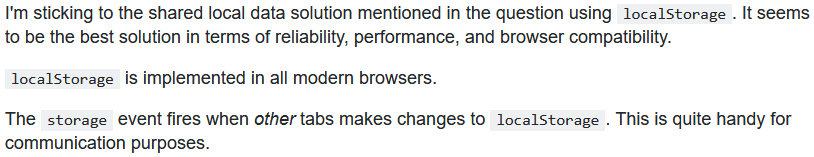




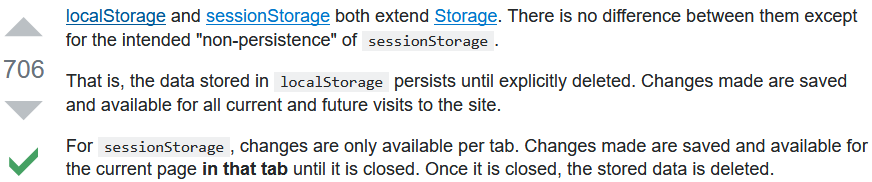
**Note**: Name/value pairs are always stored as strings. Remember to convert them to another format when needed!

**window.sessionStorage** (**NOT** shared across tabs(browser sessions)): The sessionStorage object is equal to the localStorage object, **except** that it stores the data for only one session. The data is deleted when the user closes the specific browser tab. Stays saved for the current tab even with normal or hard refresh

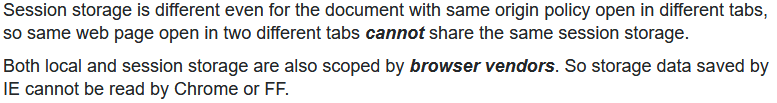












1. XML Namespaces

XML Namespaces provide a way to avoid element name conflicts.

A namespace name is a uniform resource identifier (URI). Typically, the URI chosen for the namespace of a given XML vocabulary describes a resource under the control of the author or organization defining the vocabulary, such as a URL for the author's Web server. However, the namespace specification does not require nor suggest that the namespace URI be used to retrieve information; it is simply treated by an XML parser as a string. For example, the document at http://www.w3.org/1999/xhtml itself does not contain any code. It simply describes the XHTML namespace to human readers. Using a URI (such as "http://www.w3.org/1999/xhtml") to identify a namespace, rather than a simple string (such as "xhtml"), reduces the probability of different namespaces using duplicate identifiers.

1. Identifying a Browser and communication between Tabs

TODO

1. Map, reduce, filter …

TODO

1. JavaScript Event Loop

* Javabrains event loop: [https://www.youtube.com/watch?v=EI7sN1dDwcY](https://www.youtube.com/watch?v=EI7sN1dDwcY" \t "_top)
* <https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif>
* Three strategies for the JavaScript event loop: [https://www.youtube.com/watch?v=IvLltoCt8QU](https://www.youtube.com/watch?v=IvLltoCt8QU" \t "_top)

1. Asynchronous JavaScript

JavaScript, generally speaking is **single-threaded**. Even with multiple cores, you could only get it to run tasks on a single thread, called the **main thread**.

After some time, JavaScript gained some tools to help with such problems. **Web workers** allow you to send some of the JavaScript processing off to a separate thread, called a worker so that you can run multiple JavaScript chunks simultaneously. You'd generally use a worker to run expensive processes off the main thread so that user interaction is not blocked.

**Synchronous code**: Why is this difficult to get to work using synchronous code? Let's look at a quick example. When you fetch an image from a server, you can't return the result immediately. That means that the following (pseudocode) wouldn't work:

|  |
| --- |
| **let** response **=** fetch('myImage.png'); *// fetch is asynchronous*  **let** blob **=** response.blob();  *// display your image blob in the UI somehow* |

That's because you don't know how long the image will take to download, so when you come to run the second line it will throw an error (possibly intermittently, possibly every time) because the response is not yet available. Instead, you need your code to wait until the response is returned before it tries to do anything else to it.

You can't include an async code block (i.e., fetch()) that returns a result, which you then rely on later in a sync code block. You just can't guarantee that the async function will return before the browser has processed the sync block.

There are two main types of asynchronous code style you'll come across in JavaScript code:

* old-style **callbacks**
* newer **promises**
  1. Callbacks

Callbacks were used for a long time until **ES-6 (ES-2015)** was released, where **Promises** were introduced to the language. Promises gives you a more elegant way to handle asynchronous calls.

**Async callbacks** are functions that are specified as arguments when calling a function which will start executing code in the background. When the background code finishes running, it calls the callback function to let you know the work is done, or to let you know that something of interest has happened. Using callbacks is slightly old-fashioned now, but you'll still see them in use in a number of older-but-still-commonly-used APIs.

An example of an async callback is the second parameter of the addEventListener() method:

|  |
| --- |
| btn.addEventListener('click', () => {  **let** pElem **=** document.createElement('p');  pElem.textContent **=** 'This is a newly-added paragraph.';  document.body.appendChild(pElem);  }); |

The first parameter is the type of event to be listened for, and the second parameter is a callback function that is invoked when the event is fired.

When we pass a callback function as an argument to another function, we are only passing the function's reference as an argument, i.e, the callback function is **not** executed immediately. It is “called back” (hence the name) asynchronously somewhere inside the containing function’s body. The containing function is responsible for executing the callback function when the time comes.

You can write your own function containing a callback easily enough. Let's look at another example that loads a resource via the XMLHttpRequest API:

|  |
| --- |
| **function** loadAsset(url, type, callback) {  **let** xhr **=** **new** XMLHttpRequest();  xhr.open('GET', url);  xhr.responseType **=** type;  xhr.onload **=** **function**() {  callback(xhr.response);  };  xhr.send();  }  **function** displayImage(blob) {  **let** objectURL **=** **URL**.createObjectURL(blob);  **let** image **=** document.createElement('img');  image.src **=** objectURL;  document.body.appendChild(image);  }  loadAsset('coffee.jpg', 'blob', displayImage); |

Here we create a displayImage() function that represents a blob passed to it as an object URL, then creates an image to display the URL in, appending it to the document's <body>. However, we then create a loadAsset() function that takes a callback as a parameter, along with a URL to fetch and a content type. It uses XMLHttpRequest (often abbreviated to "XHR") to fetch the resource at the given URL, then pass the response to the callback to do something with. In this case the callback is waiting on the XHR call to finish downloading the resource (using the onload event handler) before it passes it to the callback.

Callbacks are versatile — not only do they allow you to control the order in which functions are run and what data is passed between them, they also allow you to pass data to different functions depending on circumstance. So you could have different actions to run on the response downloaded, such as processJSON(), displayText(), etc.

**Note that not all callbacks are async** — some run synchronously. An example is when we use Array.prototype.forEach() to loop through the items in an array

|  |
| --- |
| **const** **gods** **=** ['Apollo', 'Artemis', 'Ares', 'Zeus'];  gods.forEach(**function** (eachName, index){  console.log(index **+** '. ' **+** eachName);  }); |

In this example we loop through an array of Greek gods and print the index numbers and values to the console. The expected parameter of forEach() is a callback function, which itself takes two parameters, a reference to the array name and index values. However, it doesn't wait for anything — it runs immediately.

* 1. Promises

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

* pending: initial state, neither fulfilled nor rejected.
* fulfilled: meaning that the operation was completed successfully.
* rejected: meaning that the operation failed.

A pending promise can either be fulfilled with a value or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained.

Diagram

Description automatically generated

* **Not to be confused with**: Several other languages have mechanisms for lazy evaluation and deferring a computation, which they also call "promises", e.g. Scheme. Promises in JavaScript represent processes that are already happening, which can be chained with callback functions. If you are looking to lazily evaluate an expression, consider the arrow function with no arguments: f = () => expression to create the lazily-evaluated expression, and f() to evaluate.
* **Note**: A promise is said to be settled if it is either fulfilled or rejected, but not pending. You will also hear the term resolved used with promises — this means that the promise is settled or “locked-in” to match the state of another promise. States and fates contain more details about promise terminology.

Promises are the new style of async code that you'll see used in modern Web APIs. A good example is the fetch() API, which is basically like a modern, more efficient version of XMLHttpRequest. Let's look at a quick example, from our Fetching data from the server article:

|  |
| --- |
| fetch('products.json')  .then(**function**(response) {  **return** response.json();  }).then(**function**(json) {  products **=** json;  initialize();  }).catch(**function**(err) {  console.log('Fetch problem: ' **+** err.message);  }); |

Here we see fetch() taking a single parameter — the URL of a resource you want to fetch from the network — and returning a promise (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>). The promise is an object representing the completion or failure of the async operation. It represents an intermediate state. In essence, it's the browser's way of saying "I promise to get back to you with the answer as soon as I can," hence the name "promise."

This concept can take practice to get used to; it feels a little like Schrödinger's cat in action. Neither of the possible outcomes have happened yet, so the fetch operation is currently waiting on the result of the browser trying to complete the operation at some point in the future. We've then got three further code blocks chained onto the end of the fetch():

* Two then() blocks. Both contain a callback function that will run if the previous operation is successful, and each callback receives as input the result of the previous successful operation, so you can go forward and do something else to it. Each .then() block returns another promise, meaning that you can chain multiple .then() blocks onto each other, so multiple asynchronous operations can be made to run in order, one after another.
* The catch() block at the end runs if any of the .then() blocks fail — in a similar way to synchronous try...catch blocks, an error object is made available inside the catch(), which can be used to report the kind of error that has occurred. Note however that synchronous try...catch won't work with promises, although it will work with async/await, as you'll learn later on.

**TODO** what does .catch() catch exactly?

* + 1. Chained Promises

The methods promise.then(), promise.catch(), and promise.finally() are used to associate further action with a promise that becomes settled.

The .then() method takes up to two arguments;

* the first argument is a callback function for the **resolved** **case** of the promise, and
* the second argument is a callback function for the **rejected** **case**.

Each .then() returns a newly generated promise object, which can optionally be used for chaining; for example:

|  |
| --- |
| **const** **myPromise** **=** **new** Promise((resolve, reject) => {  setTimeout(() => {  resolve('foo');  }, 300);  });  myPromise  .then(handleResolvedA, handleRejectedA)  .then(handleResolvedB, handleRejectedB)  .then(handleResolvedC, handleRejectedC); |

Processing continues to the next link of the chain even when a .then() lacks a callback function that returns a Promise object. Therefore, a chain can safely omit (leave out) every rejection callback function until the final .catch().

Handling a rejected promise in each .then() has consequences further down the promise chain. Sometimes there is no choice, because an error must be handled immediately. In such cases we must throw an error of some type to maintain error state down the chain. On the other hand, in the absence of an immediate need, it is simpler to leave out error handling until a final .catch() statement. A .catch() is really just a .then() without a slot for a callback function for the case when the promise is resolved.

|  |
| --- |
| myPromise  .then(handleResolvedA)  .then(handleResolvedB)  .then(handleResolvedC)  .catch(handleRejectedAny); |

Using Arrow Function Expressions for the callback functions, an implementation of a promise chain might look something like this:

|  |
| --- |
| promise1  .then(value => { **return** value **+** ' and bar'; })  .then(value => { **return** value **+** ' and bar again'; })  .then(value => { **return** value **+** ' and again'; })  .then(value => { **return** value **+** ' and again'; })  .then(value => { console.log(value) })  .catch(err => { console.log(err) }); |

The termination condition of a promise determines the "settled" state of the next promise in the chain. A "resolved" state indicates a successful completion of the promise, while a "rejected" state indicates a lack of success.

* The return value of each resolved promise in the chain is passed along to the next .then(), while the reason for **rejection** is passed along to the next **rejection-handler function** in the chain.
  + 1. Creating a Promise

Chaining: you can chain another .then after a .catch but does that make sense?

Errors or rejections can be cought either directly in the promise or we the first catch block will handle it.

We also have finally

Error handling: if one .then fails, the error will be handled by the first catch which preceeds that .then?

Promise.all

Creating Promises

|  |
| --- |
| *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *// .finally(() => {*  *// // TODO is finally and then same here?*  *// console.log('is this a cheap finally implementation?')*  *// }).finally(() => {*  *// console.log('is this a cheap finally implementation?')*  *// }).then(() => {*  *// console.log('is this a cheap finally implementation?')*  *// })*  **this**.gimmePromise()  .then(**this**.success, **this**.customErrorHandler)  .then(() => console.log('pass through Promise<void> 1'))  .then(() => console.log('pass through Promise<void> 2'))  .catch(**this**.globalErrorHandler)  .finally(() => console.log('finally'))  }  success (value**:** string) {  console.log('success: ', value)  }  customErrorHandler (error**:** any) {  console.log('customErrorHandler: ' **+** error)  }  globalErrorHandler (error**:** any) {  console.log('globalErrorHandler: ' **+** error)  }  gimmePromise ()**:** Promise<string> {  *// TODO does the language pass resolve and reject?*  **return** **new** Promise((resolve, reject) => {  **const** **num** **=** **-**1  **if** (num **>** 0) {  resolve('some string value')  } **else** {  *// TODO so if an error happens, or if you reject a promise, either way, a Promise will return, containing either the error object or the value which was passed to reject as an argument. Therefore usually an error is given to the reject function and eslint was complaining when I passed 123 as a reject value.*  **throw** **new** Error('some error')  reject(123)  }  })  } |

* + 1. Promises versus callbacks

Promises have some similarities to old-style callbacks. They are essentially a returned object to which you attach callback functions, rather than having to pass callbacks into a function.

However, promises are specifically made for handling async operations, and have many advantages over old-style callbacks:

* You can chain multiple async operations together using multiple .then() operations, passing the result of one into the next one as an input. This is much harder to do with callbacks, which often ends up with a messy "pyramid of doom" (also known as callback hell).
* Promise callbacks are always called in the strict order they are placed in the event queue.
* Error handling is much better — all errors are handled by a single .catch() block at the end of the block, rather than being individually handled in each level of the "pyramid".
* Promises avoid inversion of control, unlike old-style callbacks, which lose full control of how the function will be executed when passing a callback to a third-party library.

Callbacks makes sense for events (user clicks mouse), promises makes more sense for working and chaining asynchronous operations.

* 1. Async-Await

Introduced in ES-7 (?).

**Firebase Async Await** (should watch): https://www.youtube.com/watch?v=vn3tm0quoqE

1. Moment

**TODO:** Moment is deprecated, which one to use now?

* + 1. Asd

Asd

Asd

asd