

Popups and window methods

A popup window is one of the oldest methods to show additional documents to users.

Basically, you just run:

```
window.open('https://javascript.info/')
```

...And it will open a new window with the given URL. Most modern browsers are configured to open new tabs instead of separate windows.

Popups exist from really ancient times. The initial idea was to show another content without closing the main window. As of now, there are other ways to do that: we can load content dynamically with `fetch` and show it in a dynamically generated `<div>`. So, popups is not something we use everyday.

Also, popups are tricky on mobile devices that don't show multiple windows simultaneously.

Still, there are tasks where popups are still used, e.g. for OAuth authorization (login with Google/Facebook/...), because:

1. A popup is a separate window with its own independent JavaScript environment. So opening a popup with a third-party non-trusted site is safe.
2. It's very easy to open a popup.
3. A popup can navigate (change URL) and send messages to the opener window.

Popup blocking

In the past, evil sites abused popups a lot. A bad page could open tons of popup windows with ads. So now most browsers try to block popups and protect the user.

Most browsers block popups if they are called outside of user-triggered event handlers like `onclick`.

```
// popup blocked  
window.open('https://javascript.info');
```

```
// popup allowed  
button.onclick = () => {
```

```
window.open('https://javascript.info');  
};
```

This way users are somewhat protected from unwanted popups, but the functionality is not disabled totally.

What if the popup opens from onclick, but after setTimeout? That's a bit tricky.

Try this code:

```
// open after 3 seconds  
setTimeout(() => window.open('http://google.com'), 3000);
```

The popup opens in Chrome, but gets blocked in Firefox.

...If we decrease the delay, the popup works in Firefox too:

```
// open after 1 seconds  
setTimeout(() => window.open('http://google.com'), 1000);
```

The difference is that Firefox treats a timeout of 2000ms or less as acceptable, but after it – removes the “trust”, assuming that now it’s “outside of the user action”. So the first one is blocked, and the second one is not.

window.open

The syntax to open a popup is: `window.open(url, name, params)`:

url

An URL to load into the new window.

name

A name of the new window. Each window has a `window.name`, and here we can specify which window to use for the popup. If there's already a window with such name – the given URL opens in it, otherwise a new window is opened.

params

The configuration string for the new window. It contains settings, delimited by a comma. There must be no spaces in params, for instance: `width:200,height=100`.

Settings for params:

- Position:

`left/top` (numeric) – coordinates of the window top-left corner on the screen. There is a limitation: a new window cannot be positioned offscreen.

`width/height` (numeric) – width and height of a new window. There is a limit on minimal width/height, so it's impossible to create an invisible window.

- Window features:

`menubar` (yes/no) – shows or hides the browser menu on the new window.

`toolbar` (yes/no) – shows or hides the browser navigation bar (back, forward, reload etc) on the new window.

`location` (yes/no) – shows or hides the URL field in the new window. FF and IE don't allow you to hide it by default.

`status` (yes/no) – shows or hides the status bar. Again, most browsers force it to show.

`resizable` (yes/no) – allows to disable the resize for the new window. Not recommended.

`scrollbars` (yes/no) – allows to disable the scrollbars for the new window. Not recommended.

There are also a number of less supported browser-specific features, which are usually not used. Check [window.open in MDN](#) for examples.

Example: a minimalistic window

Let's open a window with minimal set of features just to see which of them browser allows to disable:

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,
width=0,height=0,left=-1000,top=-1000`;
open('/', 'test', params);
```

Here most “window features” are disabled and the window is positioned offscreen. Run it and see what really happens. Most browsers “fix” odd things like zero `width/height` and offscreen `left/top`. For instance, Chrome opens such a window with full width/height, so that it occupies the full screen.

Let's add normal positioning options and reasonable `width`, `height`, `left`, `top` coordinates:

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,
width=600,height=300,left=100,top=100`;
open('/', 'test', params);
```

Most browsers show the example above as required.

Rules for omitted settings:

- If there is no 3rd argument in the `open` call, or it is empty, then the default window parameters are used.
- If there is a string of params, but some `yes/no` features are omitted, then the omitted features are assumed to have `no` value. So if you specify params, make sure you explicitly set all required features to `yes`.
- If there is no `left/top` in params, then the browser tries to open a new window near the last opened window.
- If there is no `width/height`, then the new window will be the same size as the last opened.

Accessing popup from window

The `open` call returns a reference to the new window. It can be used to manipulate its properties, change location and even more.

In this example, we generate popup content from JavaScript:

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");
newWin.document.write("Hello, world!");
```

And here we modify the contents after loading:

```
let newWindow = open('/', 'example', 'width=300,height=300')
newWindow.focus();
alert(newWindow.location.href); // (*) about:blank, loading hasn't started yet
newWindow.onload = function() {
  let html = `<div style="font-size:30px">Welcome!</div>`;
  newWindow.document.body.insertAdjacentHTML('afterbegin', html);
};
```

Please note: immediately after `window.open`, the new window isn't loaded yet. That's demonstrated by `alert` in line (*). So we wait for `onload` to modify it. We could also use the `DOMContentLoaded` handler for the `newWin.document`.

Closing a popup

To close a window: `win.close()`.

To check if a window is closed: `win.closed`.

Technically, the `close()` method is available for any window, but `window.close()` is ignored by most browsers if the window is not created with `window.open()`. So it'll only work on a popup.

The `closed` property is `true` if the window is closed. That's useful to check if the popup (or the main window) is still open or not. A user can close it anytime, and our code should take that possibility into account.

This code loads and then closes the window:

```
let newWindow = open('/', 'example', 'width=300,height=300');
newWindow.onload = function() {
  newWindow.close();
  alert(newWindow.closed); // true
};
```

Cross-window communication

The “Same Origin” (same site) policy limits access of windows and frames to each other.

The idea is that if a user has two pages open: one from `john-smith.com`, and another one is `gmail.com`, then they wouldn't want a script from `john-smith.com` to read our mail from `gmail.com`. So, the purpose of the “Same Origin” policy is to protect users from information theft.

Same Origin

Two URLs are said to have the “same origin” if they have the same protocol, domain and port.

These URLs all share the same origin:

- `http://site.com`
- `http://site.com/`
- `http://site.com/my/page.html`

These ones do not:

- `http://www.site.com` (another domain: `www.` matters)
- `http://site.org` (another domain: `.org` matters)
- `https://site.com` (another protocol: `https`)
- `http://site.com:8080` (another port: `8080`)

The “Same Origin” policy states that:

- if we have a reference to another window, e.g. a popup created by `window.open` or a window inside `<iframe>`, and that window comes from the same origin, then we have full access to that window.
- otherwise, if it comes from another origin, then we can’t access the content of that window: variables, document, anything. The only exception is `location`: we can change it (thus redirecting the user). But we cannot *read* `location` (so we can’t see where the user is now, no information leak)

In action: iframe

An `<iframe>` tag hosts a separate embedded window, with its own separate document and window objects.

We can access them using properties:

- `iframe.contentWindow` to get the window inside the `<iframe>`.
- `iframe.contentDocument` to get the document inside the `<iframe>`, shorthand for `iframe.contentWindow.document`.

When we access something inside the embedded window, the browser checks if the iframe has the same origin. If that’s not so then the access is denied (writing to `location` is an exception, it’s still permitted).

For instance, let’s try reading and writing to `<iframe>` from another origin:

```
<iframe src="https://example.com" id="iframe"></iframe>
```

```

<script>
  iframe.onload = function() {
    // we can get the reference to the inner window
    let iframeWindow = iframe.contentWindow; // OK
    try {
      // ...but not to the document inside it
      let doc = iframe.contentDocument; // ERROR
    } catch(e) {
      alert(e); // Security Error (another origin)
    }
    // also we can't READ the URL of the page in iframe
    try {
      // Can't read URL from the Location object
      let href = iframe.contentWindow.location.href; // ERROR
    } catch(e) {
      alert(e); // Security Error
    }
    // ...we can WRITE into location (and thus load something else into the iframe)!
    iframe.contentWindow.location = '/'; // OK
    iframe.onload = null; // clear the handler, not to run it after the location change
  };
</script>

```

The code above shows errors for any operations except:

- Getting the reference to the inner window `iframe.contentWindow` – that's allowed.
- Writing to `location`.

Contrary to that, if the `<iframe>` has the same origin, we can do anything with it:

```

<!-- iframe from the same site -->
<iframe src="/" id="iframe"></iframe>
<script>
  iframe.onload = function() {
    // just do anything

```

```
    iframe.contentDocument.body.prepend("Hello, world!");  
};  
</script>
```

iframe.onload vs iframe.contentWindow.onload

The `iframe.onload` event (on the `<iframe>` tag) is essentially the same as `iframe.contentWindow.onload` (on the embedded window object). It triggers when the embedded window fully loads with all resources.

...But we can't access `iframe.contentWindow.onload` for an `iframe` from another origin, so using `iframe.onload`.

iframe: wrong document pitfall

When an `iframe` comes from the same origin, and we may access its `document`, there's a pitfall. It's not related to cross-origin things, but important to know.

Upon its creation an `iframe` immediately has a document. But that document is different from the one that loads into it!

So if we do something with the document immediately, that will probably be lost.

Here, look:

```
<iframe src="/" id="iframe"></iframe>  
<script>  
    let oldDoc = iframe.contentDocument;  
    iframe.onload = function() {  
        let newDoc = iframe.contentDocument;  
        // the loaded document is not the same as initial!  
        alert(oldDoc == newDoc); // false  
    };  
</script>
```

We shouldn't work with the document of a not-yet-loaded `iframe`, because that's the *wrong document*. If we set any event handlers on it, they will be ignored.

How to detect the moment when the document is there?

The right document is definitely at place when `iframe.onload` triggers. But it only triggers when the whole iframe with all resources is loaded.

We can try to catch the moment earlier using checks in `setInterval`:

```
<iframe src="/" id="iframe"></iframe>
<script>
  let oldDoc = iframe.contentDocument;
  // every 100 ms check if the document is the new one
  let timer = setInterval(() => {
    let newDoc = iframe.contentDocument;
    if (newDoc == oldDoc) return;
    alert("New document is here!");
    clearInterval(timer); // cancel setInterval, don't need it any more
  }, 100);
</script>
```

Collection: `window.frames`

An alternative way to get a window object for `<iframe>` – is to get it from the named collection `window.frames`:

- By number: `window.frames[0]` – the window object for the first frame in the document.
- By name: `window.frames.iframeName` – the window object for the frame with `name="iframeName"`.

For instance:

```
<iframe src="/" style="height:80px" name="win" id="iframe"></iframe>
<script>
  alert(iframe.contentWindow == frames[0]); // true
  alert(iframe.contentWindow == frames.win); // true
</script>
```

An iframe may have other iframes inside. The corresponding `window` objects form a hierarchy.

Navigation links are:

- `window.frames` – the collection of “children” windows (for nested frames).
- `window.parent` – the reference to the “parent” (outer) window.
- `window.top` – the reference to the topmost parent window.

For instance:

```
window.frames[0].parent === window; // true
```

We can use the `top` property to check if the current document is open inside a frame or not:

```
if (window === top) { // current window === window.top?
    alert('The script is in the topmost window, not in a frame');
} else {
    alert('The script runs in a frame!');
}
```

The “sandbox” iframe attribute

The `sandbox` attribute allows for the exclusion of certain actions inside an `<iframe>` in order to prevent it executing untrusted code. It “sandboxes” the iframe by treating it as coming from another origin and/or applying other limitations.

There’s a “default set” of restrictions applied for `<iframe sandbox src=“...”>`. But it can be relaxed if we provide a space-separated list of restrictions that should not be applied as a value of the attribute, like this: `<iframe sandbox=“allow-forms allow-popups”>`.

In other words, an empty “`sandbox`” attribute puts the strictest limitations possible, but we can put a space-delimited list of those that we want to lift.

Here’s a list of limitations:

allow-same-origin

By default “`sandbox`” forces the “different origin” policy for the iframe. In other words, it makes the browser treat the iframe as coming from another origin, even if its `src` points to the same site. With all implied restrictions for scripts. This option removes that feature.

allow-top-navigation

Allows the `iframe` to change `parent.location`

.

allow-forms

Allows to submit forms from `iframe`.

allow-scripts

Allows to run scripts from the `iframe`.

allow-popups

Allows to `window.open` popups from the `iframe`

See the manual for more.

The example below demonstrates a sandboxed `iframe` with the default set of restrictions:

`<iframe sandbox src="...">`. It has some JavaScript and a form.

Please note that nothing works. So the default set is really harsh:

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <div>The iframe below is has <code>sandbox</code> attribute.</div>
  <iframe sandbox src="sandboxed.html" style="height:60px;width:90%"></iframe>
</body>
</html>
```

//sanboxed.html

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
</head>
```

```
<body>
  <button onclick="alert(123)">Click to run a script (doesn't work)</button>
  <form action="http://google.com">
    <input type="text">
    <input type="submit" value="Submit (doesn't work)">
  </form>
</body>
</html>
```

Cross-window messaging

The `postMessage` interface allows windows to talk to each other no matter which origin they are from.

So, it's a way around the "Same Origin" policy. It allows a window from `john-smith.com` to talk to `gmail.com` and exchange information, but only if they both agree and call corresponding JavaScript functions. That makes it safe for users.

The interface has two parts.

postMessage

The window that wants to send a message calls the `postMessage` method of the receiving window. In other words, if we want to send the message to `win`, we should call `win.postMessage(data, targetOrigin)`.

Arguments:

data

The data to send. Can be any object, the data is cloned using the "structured cloning algorithm". IE supports only strings, so we should `JSON.stringify` complex objects to support that browser.

targetOrigin

Specifies the origin for the target window, so that only a window from the given origin will get the message.

The `targetOrigin` is a safety measure. Remember, if the target window comes from another origin, we can't read its `location` in the sender window. So we can't be sure which site is open in the intended window right now: the user could navigate away, and the sender window has no idea about it.

Specifying `targetOrigin` ensures that the window only receives the data if it's still at the right site. Important when the data is sensitive.

For instance, here `win` will only receive the message if it has a document from the origin `http://example.com`:

```
<iframe src="http://example.com" name="example">
<script>
  let win = window.frames.example;
  win.postMessage("message", "http://example.com");
</script>
```

If we don't want that check, we can set `targetOrigin` to `*`.

```
<iframe src="http://example.com" name="example">
<script>
  let win = window.frames.example;
  win.postMessage("message", "*");
</script>
```

onmessage

To receive a message, the target window should have a handler on the `message` event. It triggers when `postMessage` is called (and `targetOrigin` check is successful).

The event object has special properties:

data

The data from `postMessage`.

origin

The origin of the sender, for instance `http://javascript.info`.

source

The reference to the sender window. We can immediately `source.postMessage(...)` back if we want.

To assign that handler, we should use `addEventListener`, a short syntax `window.onmessage` does not work.

Here's an example:

```
window.addEventListener("message", function(event) {  
  if (event.origin !== 'http://javascript.info') {  
    // something from an unknown domain, let's ignore it  
    return;  
  }  
  alert( "received: " + event.data );  
  // can message back using event.source.postMessage(...)  
});  
// index.html
```

```
<!doctype html>  
<html>  
<head>  
  <meta charset="UTF-8">  
</head>  
<body>  
  <form id="form">  
    <input type="text" placeholder="Enter message" name="message">  
    <input type="submit" value="Click to send">  
  </form>  
  <iframe src="iframe.html" id="iframe" style="display:block;height:60px"></iframe>  
  <script>  
    form.onsubmit = function() {  
      iframe.contentWindow.postMessage(this.message.value, '*');  
      return false;  
    };  
  </script>
```

```
</body>
```

```
</html>
```

```
// iframe.html
```

```
<!doctype html>
```

```
<html>
```

```
<head>
```

```
  <meta charset="UTF-8">
```

```
</head>
```

```
<body>
```

```
  Receiving iframe.
```

```
  <script>
```

```
    window.addEventListener('message', function(event) {
```

```
      alert(`Received ${event.data} from ${event.origin}`);
```

```
    });
```

```
  </script>
```

```
</body>
```

```
</html>
```

Fetch

JavaScript can send network requests to the server and load new information whenever it's needed.

For example, we can use a network request to:

- Submit an order,
- Load user information,
- Receive latest updates from the server,
- ...etc.

...And all of that without reloading the page!

There's an umbrella term "AJAX" (abbreviated **A**synchronous **J**avaScript **A**nd **X**ML) for network requests from JavaScript. We don't have to use XML though: the term comes from old times, that's why that word is there. You may have heard that term already.

There are multiple ways to send a network request and get information from the server.

The `fetch()` method is modern and versatile, so we'll start with it. It's not supported by old browsers (can be polyfilled), but very well supported among the modern ones.

The basic syntax is:

```
let promise = fetch(url, [options])
```

- **url** – the URL to access.
- **options** – optional parameters: method, headers etc.

Without options, that is a simple GET request, downloading the contents of the `url`.

The browser starts the request right away and returns a promise that the calling code should use to get the result.

Getting a response is usually a two-stage process.

First, the `promise`, returned by `fetch`, resolves with an object of the built-in `Response` class as soon as the server responds with headers.

At this stage we can check HTTP status, to see whether it is successful or not, check headers, but don't have the body yet.

The promise rejects if the `fetch` was unable to make HTTP-request, e.g. network problems, or there's no such site. Abnormal HTTP-statuses, such as 404 or 500 do not cause an error.

We can see HTTP-status in response properties:

- **status** – HTTP status code, e.g. 200.
- **ok** – boolean, `true` if the HTTP status code is 200-299.

For example:

```
let response = await fetch(url);
if (response.ok) { // if HTTP-status is 200-299
  // get the response body (the method explained below)
```



```
    let json = await response.json();  
  } else {  
    alert("HTTP-Error: " + response.status);  
  }  
}
```

Second, to get the response body, we need to use an additional method call.

Response provides multiple promise-based methods to access the body in various formats:

- **response.text()** – read the response and return as text,
- **response.json()** – parse the response as JSON,
- **response.formData()** – return the response as FormData object (explained in the next chapter),
- **response.blob()** – return the response as Blob (binary data with type),
- **response.arrayBuffer()** – return the response as ArrayBuffer (low-level representation of binary data),
- Additionally, `response.body` is a ReadableStream object, it allows you to read the body chunk-by-chunk, we'll see an example later.

For instance, let's get a JSON-object with latest commits from GitHub:

```
let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits';  
let response = await fetch(url);  
let commits = await response.json(); // read response body and parse as JSON  
alert(commits[0].author.login);
```

Or, the same without `await`, using pure promises syntax:

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')  
  .then(response => response.json())  
  .then(commits => alert(commits[0].author.login));
```

To get the response text, `await response.text()` instead of `.json()`:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');
```

```
let text = await response.text(); // read response body as text
alert(text.slice(0, 80) + '...');
```

As a show-case for reading in binary format, let's fetch and show a logo image of “fetch” specification (see chapter Blob for details about operations on Blob):

```
let response = await fetch('/article/fetch/logo-fetch.svg');
let blob = await response.blob(); // download as Blob object
// create <img> for it
let img = document.createElement('img');
img.style = 'position:fixed;top:10px;left:10px;width:100px';
document.body.append(img);
// show it
img.src = URL.createObjectURL(blob);
setTimeout(() => { // hide after three seconds
  img.remove();
  URL.revokeObjectURL(img.src);
}, 3000);
```

We can choose only one body-reading method.

If we've already got the response with `response.text()`, then `response.json()` won't work, as the body content has already been processed.

```
let text = await response.text(); // response body consumed
let parsed = await response.json(); // fails (already consumed)
```

Response headers

The response headers are available in a Map-like headers object in `response.headers`.

It's not exactly a Map, but it has similar methods to get individual headers by name or iterate over them:

```
let response = await fetch('https://api.github.com/repos/javascript-
tutorial/en.javascript.info/commits');
// get one header
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8
```

```
// iterate over all headers
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

Request headers

To set a request header in `fetch`, we can use the `headers` option. It has an object with outgoing headers, like this:

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'secret'
  }
});
```

...But there's a list of forbidden HTTP headers that we can't set:

- Accept-Charset, Accept-Encoding
- Access-Control-Request-Headers
- Access-Control-Request-Method
- Connection
- Content-Length
- Cookie, Cookie2
- Date
- DNT
- Expect
- Host
- Keep-Alive
- Origin
- Referer
- TE
- Trailer
- Transfer-Encoding
- Upgrade
- Via
- Proxy-*

- Sec-*

These headers ensure proper and safe HTTP, so they are controlled exclusively by the browser

POST requests

To make a POST request, or a request with another method, we need to use fetch options:

- **method** – HTTP-method, e.g. POST,
- **body** – the request body, one of:
 - a string (e.g. JSON-encoded),
 - FormData object, to submit the data as form/multipart,
 - Blob/BufferSource to send binary data,
 - URLSearchParams, to submit the data in x-www-form-urlencoded encoding, rarely used.

The JSON format is used most of the time.

For example, this code submits user object as JSON:

```
let user = {  
  name: 'John',  
  surname: 'Smith'  
};  
let response = await fetch('/article/fetch/post/user', {  
  method: 'POST',  
  headers: {  
    'Content-Type': 'application/json;charset=utf-8'  
  },  
  body: JSON.stringify(user)  
});  
let result = await response.json();  
alert(result.message);
```

Please note, if the request body is a string, then the Content-Type header is set to text/plain;charset=UTF-8 by default.

But, as we're going to send JSON, we use the headers option to send `application/json` instead, the correct `Content-Type` for JSON-encoded data.

Sending an image

We can also submit binary data with `fetch` using `Blob` or `BufferSource` objects.

In this example, there's a `<canvas>` where we can draw by moving a mouse over it. A click on the "submit" button sends the image to the server:

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>
  <input type="button" value="Submit" onclick="submit()">
  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };
    async function submit() {
      let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
      let response = await fetch('/article/fetch/post/image', {
        method: 'POST',
        body: blob
      });
      // the server responds with confirmation and the image size
      let result = await response.json();
      alert(result.message);
    }
  </script>
</body>
```

Please note, here we don't set `Content-Type` header manually, because a `Blob` object has a built-in type (here `image/png`, as generated by `toBlob`). For `Blob` objects that type becomes the value of `Content-Type`.

The `submit()` function can be rewritten without `async/await` like this:

```
function submit() {  
  canvasElem.toBlob(function(blob) {  
    fetch('/article/fetch/post/image', {  
      method: 'POST',  
      body: blob  
    })  
    .then(response => response.json())  
    .then(result => alert(JSON.stringify(result, null, 2)))  
  }, 'image/png');  
}
```

FormData

This chapter is about sending HTML forms: with or without files, with additional fields and so on.

FormData objects can help with that. As you might have guessed, it's the object to represent HTML form data.

The constructor is:

```
let formData = new FormData([form]);
```

If an HTML form element is provided, it automatically captures its fields.

The special thing about FormData is that network methods, such as fetch, can accept a FormData object as a body. It's encoded and sent out with Content-Type: multipart/form-data.

From the server point of view, that looks like a usual form submission.

Sending a simple form

Let's send a simple form first.

As you can see, that's almost one-liner:

```
<form id="formElem">  
  <input type="text" id="name" name="name" value="John">  
  <input type="text" id="surname" name="surname" value="Smith">  
  <input type="submit">
```

```
</form>
```

```
<script>
```

```
  formElem.onsubmit = async (e) => {  
    e.preventDefault();  
    var formData = new FormData(formElem);  
    for (var [key, value] of formData.entries()) {  
      console.log(key, value);  
    }  
    let response = await fetch('/testing', {  
      method: 'POST',  
      body: formData  
    })  
    let result = await response.json();  
    alert(result.message);  
  };  
};
```

```
</script>
```

In this example, the server code is not presented, as it's beyond our scope. The server accepts the POST request and replies "User saved".

FormData Methods

We can modify fields in `FormData` with methods:

- `formData.append(name, value)` – add a form field with the given name and value,
- `formData.append(name, blob, fileName)` – add a field as if it were `<input type="file">`, the third argument `fileName` sets file name (not form field name), as it were a name of the file in user's filesystem,
- `formData.delete(name)` – remove the field with the given name,
- `formData.get(name)` – get the value of the field with the given name,
- `formData.has(name)` – if there exists a field with the given name, returns `true`, otherwise `false`

A form is technically allowed to have many fields with the same name, so multiple calls to `append` add more same-named fields.

There's also method `set`, with the same syntax as `append`. The difference is that `.set` removes all fields with the given name, and then appends a new field. So it makes sure there's only one field with such name, the rest is just like `append`:

- `formData.set(name, value),`
- `formData.set(name, blob, fileName).`

Sending a form with a file

The form is always sent as `Content-Type: multipart/form-data`, this encoding allows you to send files. So, `<input type="file">` fields are sent also, similar to a usual form submission.

Here's an example with such form:

```
<form id="formElem">
  <input type="text" name="firstName" value="John">
  Picture: <input type="file" name="picture" accept="image/*">
  <input type="submit">
</form>
<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();
    let response = await fetch('/article/formdata/post/user-avatar', {
      method: 'POST',
      body: new FormData(formElem)
    });
    let result = await response.json();
    alert(result.message);
  };
</script>
```

Sending a form with Blob data

As we've seen in the chapter `Fetch`, it's easy to send dynamically generated binary data e.g. an image, as `Blob`. We can supply it directly as `fetch` parameter `body`.

In practice though, it's often convenient to send an image not separately, but as a part of the form, with additional fields, such as "name" and other metadata.

Also, servers are usually more suited to accept multipart-encoded forms, rather than raw binary data.

This example submits an image from `<canvas>`, along with some other fields, as a form, using `FormData`:

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>
  <input type="button" value="Submit" onclick="submit()">
  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };
    async function submit() {
      let imageBlob = await new Promise(resolve => canvasElem.toBlob(resolve,
'image/png'));
      let formData = new FormData();
      formData.append("firstName", "John");
      formData.append("image", imageBlob, "image.png");
      let response = await fetch('/article/formdata/post/image-form', {
        method: 'POST',
        body: formData
      });
      let result = await response.json();
      alert(result.message);
    }
  </script>
</body>
```

Fetch: Download progress

The `fetch` method allows you to track *download* progress.

Please note: there's currently no way for `fetch` to track *upload* progress. For that purpose, please use `XMLHttpRequest`, we'll cover it later.

To track download progress, we can use `response.body` property. It's `ReadableStream` – a special object that provides body chunk-by-chunk, as it comes. Readable streams are described in the Streams API specification.

Unlike `response.text()`, `response.json()` and other methods, `response.body` gives full control over the reading process, and we can count how much is consumed at any moment.

Here's the sketch of code that reads the response from `response.body`:

```
// instead of response.json() and other methods
const reader = response.body.getReader();
// infinite loop while the body is downloading
while(true) {
  // done is true for the last chunk
  // value is Uint8Array of the chunk bytes
  const {done, value} = await reader.read();
  if (done) {
    break;
  }
  console.log(`Received ${value.length} bytes`)
}
```

The result of `await reader.read()` call is an object with two properties:

- **done** – true when the reading is complete, otherwise false.
- **value** – a typed array of bytes: `Uint8Array`

Please note:

Streams API also describes asynchronous iteration over `ReadableStream` with `for await..of` loop, but it's not yet widely supported (see browser issues), so we use `while` loop.

We receive response chunks in the loop, until the loading finishes, that is: until `done` becomes `true`.

To log the progress, we just need for every received fragment `value` to add its length to the counter.

Here's the full working example that gets the response and logs the progress in console, more explanations to follow:

```
// Step 1: start the fetch and obtain a reader
```

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits?per_page=100');
const reader = response.body.getReader();
```

```
// Step 2: get total length
```

```
const contentLength = +response.headers.get('Content-Length');
```

```
// Step 3: read the data
```

```
let receivedLength = 0; // received that many bytes at the moment
let chunks = []; // array of received binary chunks (comprises the body)
while(true) {
  const {done, value} = await reader.read();
  if (done) {
    break;
  }
  chunks.push(value);
  receivedLength += value.length;
  console.log(` Received ${receivedLength} of ${contentLength} `)
}
```

```
// Step 4: concatenate chunks into single Uint8Array
```

```
let chunksAll = new Uint8Array(receivedLength); // (4.1)
let position = 0;
for(let chunk of chunks) {
  chunksAll.set(chunk, position); // (4.2)
  position += chunk.length;
}
```

```
// Step 5: decode into a string
let result = new TextDecoder("utf-8").decode(chunksAll);
// We're done!
let commits = JSON.parse(result);

alert(commits[0].author.login);
```

Let's explain that step-by-step:

1. We perform `fetch` as usual, but instead of calling `response.json()`, we obtain a stream reader `response.body.getReader()`.
Please note, we can't use both these methods to read the same response: either use a reader or a response method to get the result.
2. Prior to reading, we can figure out the full response length from the `Content-Length` header.
It may be absent for cross-origin requests (see chapter Fetch: Cross-Origin Requests) and, well, technically a server doesn't have to set it. But usually it's at place.
3. Call `await reader.read()` until it's done.
We gather response chunks in the array `chunks`. That's important, because after the response is consumed, we won't be able to "re-read" it using `response.json()` or another way (you can try, there'll be an error).
4. At the end, we have `chunks` – an array of `Uint8Array` byte chunks. We need to join them into a single result. Unfortunately, there's no single method that concatenates those, so there's some code to do that:
 1. We create `chunksAll = new Uint8Array(receivedLength)` – a same-typed array with the combined length.
 2. Then use `.set(chunk, position)` method to copy each chunk one after another in it.
5. We have the result in `chunksAll`. It's a byte array though, not a string.
To create a string, we need to interpret these bytes. The built-in `TextDecoder` does exactly that. Then we can `JSON.parse` it, if necessary.
What if we need binary content instead of a string? That's even simpler. Replace steps 4 and 5 with a single line that creates a `Blob` from all chunks:
`let blob = new Blob(chunks);`

Fetch: Abort

As we know, fetch returns a promise. And JavaScript generally has no concept of “aborting” a promise. So how can we abort a fetch?

There’s a special built-in object for such purposes: `AbortController`, that can be used to abort not only fetch, but other asynchronous tasks as well.

The usage is pretty simple:

- Step 1: create a controller:

```
let controller = new AbortController();
```

A controller is an extremely simple object.

- It has a single method `abort()`, and a single property `signal`.
- When `abort()` is called:
 - `abort` event triggers on `controller.signal`
 - `controller.signal.aborted` property becomes `true`.

All parties interested to learn about `abort()` call set listeners on `controller.signal` to track it.

Like this (without fetch yet):

```
let controller = new AbortController();
let signal = controller.signal;
// triggers when controller.abort() is called
signal.addEventListener('abort', () => alert("abort!"));
controller.abort(); // abort!
alert(signal.aborted); // true
```

- Step 2: pass the `signal` property to `fetch` option:

```
let controller = new AbortController();
fetch(url, {
  signal: controller.signal
});
```

The `fetch` method knows how to work with `AbortController`, it listens to `abort` on `signal`.

- Step 3: to abort, call `controller.abort()`:

```
controller.abort();
```

We're done: `fetch` gets the event from `signal` and aborts the request.

When a `fetch` is aborted, its promise rejects with an error `AbortError`, so we should handle it, e.g. in `try..catch`:

```
// abort in 1 second
let controller = new AbortController();
setTimeout(() => controller.abort(), 1000);
try {
  let response = await fetch('/article/fetch-abort/demo/hang', {
    signal: controller.signal
  });
} catch(err) {
  if (err.name == 'AbortError') { // handle abort()
    alert("Aborted!");
  } else {
    throw err;
  }
}
```

AbortController is scalable, it allows to cancel multiple fetches at once.

For instance, here we fetch many urls in parallel, and the controller aborts them all:

```
let urls = [...]; // a list of urls to fetch in parallel
let controller = new AbortController();
let fetchJobs = urls.map(url => fetch(url, {
  signal: controller.signal
}));
let results = await Promise.all(fetchJobs);
// if controller.abort() is called from elsewhere,
// it aborts all fetches
```

Fetch: Cross-Origin Requests

If we send a fetch request to another web-site, it will probably fail.

For instance, let's try fetching `http://example.com`:

```
try {  
  await fetch('http://example.com');  
} catch(err) {  
  alert(err); // Failed to fetch  
}
```

Fetch fails, as expected.

The core concept here is *origin* – a domain/port/protocol triplet.

Cross-origin requests – those sent to another domain (even a subdomain) or protocol or port – require special headers from the remote side.

That policy is called “CORS”: Cross-Origin Resource Sharing.

Simple requests

There are two types of cross-origin requests:

1. Simple requests.
2. All the others.

Simple Requests are, well, simpler to make, so let's start with them.

A simple request is a request that satisfies two conditions:

1. Simple method: GET, POST or HEAD
2. Simple headers – the only allowed custom headers are:
 - Accept,
 - Accept-Language,
 - Content-Language,
 - Content-Type with the value `application/x-www-form-urlencoded`, `multipart/form-data` or `text/plain`.

Any other request is considered “non-simple”. For instance, a request with `PUT` method or with an `API-Key` HTTP-header does not fit the limitations.

The essential difference is that a “simple request” can be made with a `<form>` or a `<script>`, without any special methods.

CORS for simple requests

If a request is cross-origin, the browser always adds the Origin header to it.

For instance, if we request `https://anywhere.com/request` from `https://javascript.info/page`, the headers will be like:

GET /request

Host: anywhere.com

Origin: https://javascript.info

...

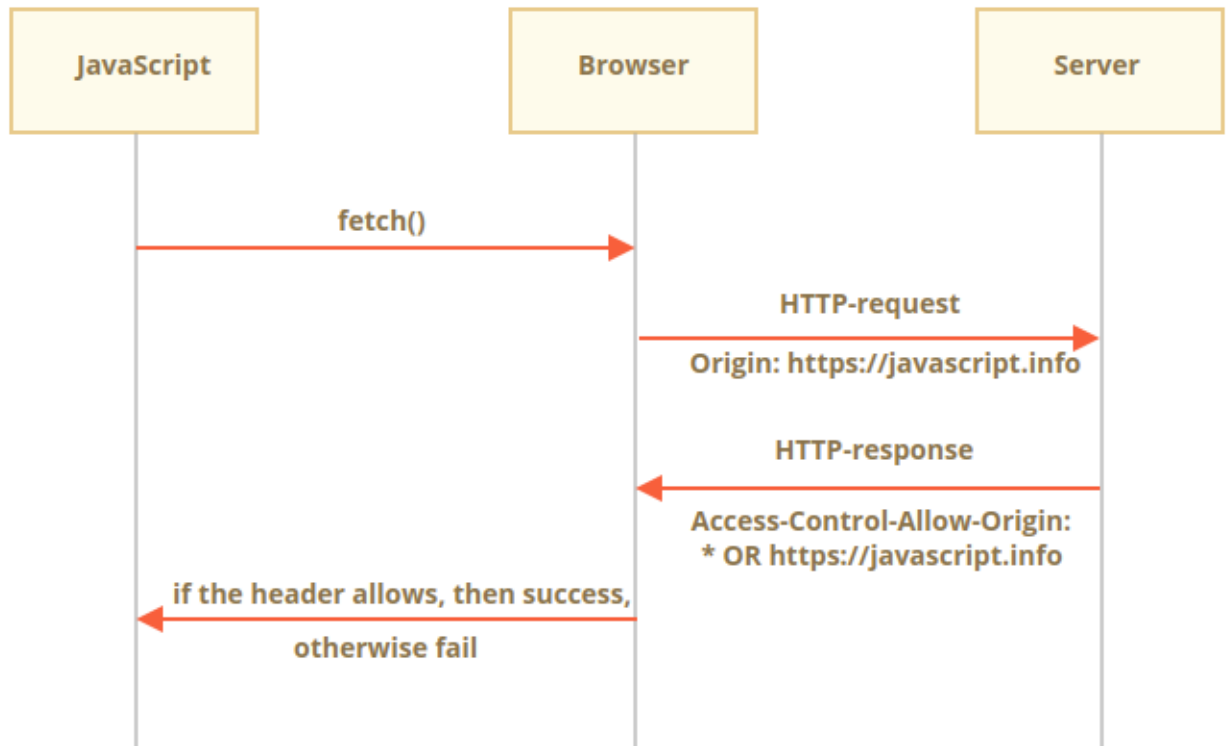
As you can see, the Origin header contains exactly the origin (domain/protocol/port), without a path.

The server can inspect the Origin and, if it agrees to accept such a request, adds a special header `Access-Control-Allow-Origin` to the response. That header should contain the allowed origin (in our case `https://javascript.info`), or a star `*`. Then the response is successful, otherwise an error.`https://javascript.info`

The browser plays the role of a trusted mediator here:

1. It ensures that the correct Origin is sent with a cross-origin request.

2. It checks for permitting `Access-Control-Allow-Origin` in the response, if it exists, then JavaScript is allowed to access the response, otherwise it fails with an error.



Here's an example of a permissive server response:

200 OK

Content-Type:text/html; charset=UTF-8

Access-Control-Allow-Origin: https://javascript.info

Response headers

For cross-origin request, by default JavaScript may only access so-called "simple" response headers:

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified
- Pragma

Accessing any other response header causes an error.

To grant JavaScript access to any other response header, the server must send `Access-Control-Expose-Headers` header. It contains a comma-separated list of non-simple header names that should be made accessible.

For example:

200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 12345

API-Key: 2c9de507f2c54aa1

Access-Control-Allow-Origin: https://javascript.info

Access-Control-Expose-Headers: Content-Length, API-Key

With such `Access-Control-Expose-Headers` header, the script is allowed to read `Content-Length` and `API-Key` headers of the response.

“Non-simple” requests

We can use any HTTP-method: not just `GET/POST`, but also `PATCH`, `DELETE` and others.

Some time ago no one could even imagine that a webpage could make such requests. So there may still exist web services that treat a non-standard method as a signal: “That’s not a browser”. They can take it into account when checking access rights.

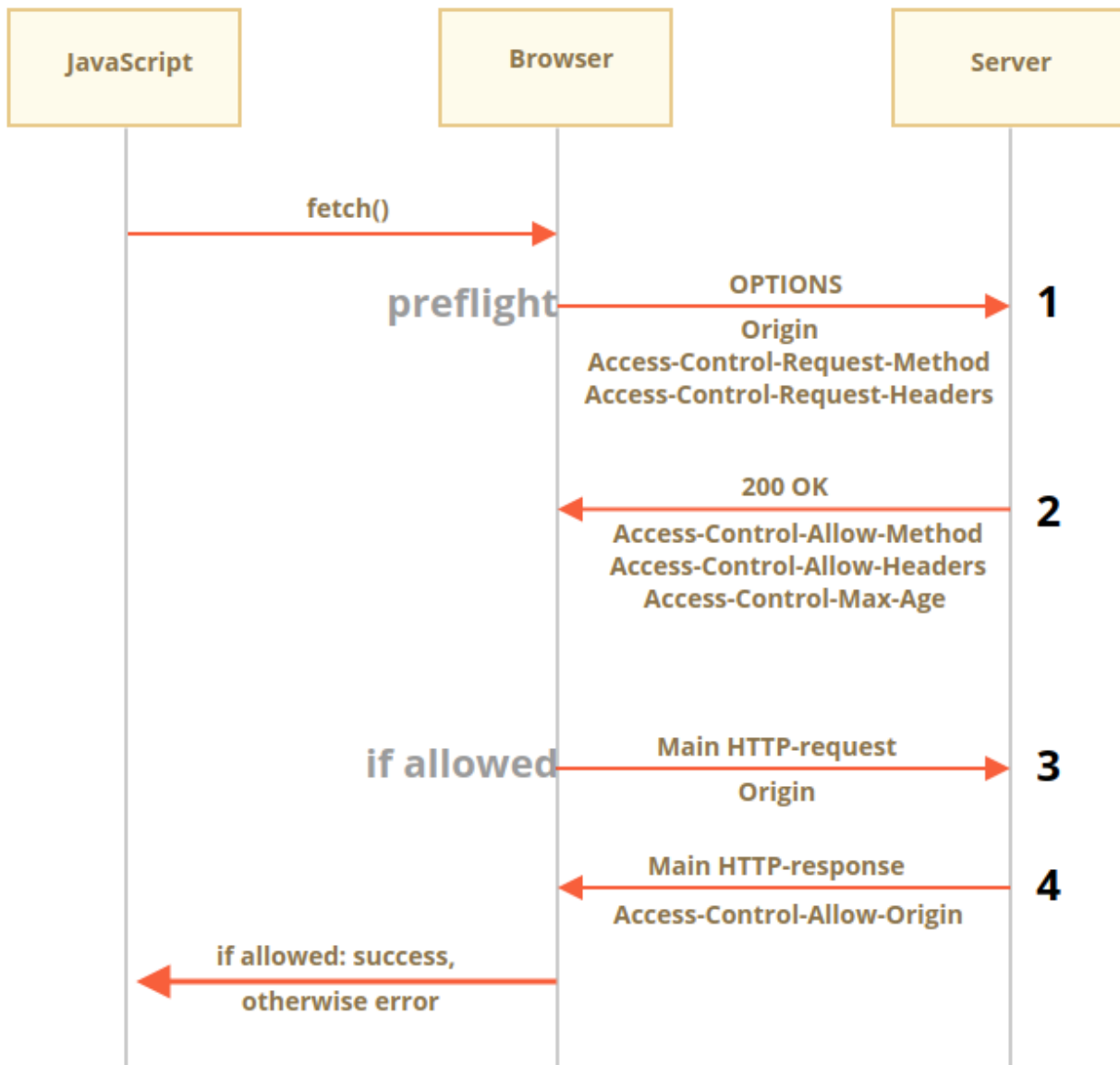
So, to avoid misunderstandings, any “non-simple” request – that couldn’t be done in the old times, the browser does not make such requests right away. Before it sends a preliminary, so-called “preflight” request, asking for permission.

A preflight request uses method `OPTIONS`, no body and two headers:

- The `Access-Control-Request-Method` header has the method of the non-simple request.
- `Access-Control-Request-Headers` header provides a comma-separated list of its non-simple HTTP-headers.

If the server agrees to serve the requests, then it should respond with empty body, status 200 and headers:

- `Access-Control-Allow-Methods` must have the allowed method.
- `Access-Control-Allow-Headers` must have a list of allowed headers.
- Additionally, the header `Access-Control-Max-Age` may specify a number of seconds to cache the permissions. So the browser won't have to send a preflight for subsequent requests that satisfy given permissions.



Let's see how it works step-by-step on example, for a cross-origin `PATCH` request (this method is often used to update data):

```
let response = await fetch('https://site.com/service.json', {
  method: 'PATCH',
```

```
headers: {  
  'Content-Type': 'application/json',  
  'API-Key': 'secret'  
}  
});
```

There are three reasons why the request is not simple (one is enough):

- Method PATCH
- Content-Type is not one of: application/x-www-form-urlencoded, multipart/form-data, text/plain.
- “Non-simple” API-Key header.

Step 1 (preflight request)

Prior to sending such request, the browser, on its own, sends a preflight request that looks like this:

```
OPTIONS /service.json  
Host: site.com  
Origin: https://javascript.info  
Access-Control-Request-Method: PATCH  
Access-Control-Request-Headers: Content-Type,API-Key
```

- Method: OPTIONS.
- The path – exactly the same as the main request: /service.json.
- Cross-origin special headers:
 - Origin – the source origin.
 - Access-Control-Request-Method – requested method.
 - Access-Control-Request-Headers – a comma-separated list of “non-simple” headers.

Step 2 (preflight response)

The server should respond with status 200 and headers:

- Access-Control-Allow-Methods: PATCH

- Access-Control-Allow-Headers: Content-Type,API-Key.

That allows future communication, otherwise an error is triggered.

If the server expects other methods and headers in the future, it makes sense to allow them in advance by adding to the list:

200 OK

Access-Control-Allow-Methods: PUT,PATCH,DELETE

Access-Control-Allow-Headers: API-Key,Content-Type,If-Modified-Since,Cache-Control

Access-Control-Max-Age: 86400

Now the browser can see that PATCH is in Access-Control-Allow-Methods and Content-Type,API-Key are in the list Access-Control-Allow-Headers, so it sends out the main request.

Besides, the preflight response is cached for time, specified by Access-Control-Max-Age header (86400 seconds, one day), so subsequent requests will not cause a preflight. Assuming that they fit the cached allowances, they will be sent directly.

Step 3 (actual request)

When the preflight is successful, the browser now makes the main request. The algorithm here is the same as for simple requests.

The main request has Origin header (because it's cross-origin):

PATCH /service.json

Host: site.com

Content-Type: application/json

API-Key: secret

Origin: https://javascript.info

Step 4 (actual response)

The server should not forget to add Access-Control-Allow-Origin to the main response. A successful preflight does not relieve from that:

Access-Control-Allow-Origin: https://javascript.info

Then JavaScript is able to read the main server response.

Please note:

Preflight request occurs “behind the scenes”, it’s invisible to JavaScript.

JavaScript only gets the response to the main request or an error if there’s no server permission.

Credentials

A cross-origin request by default does not bring any credentials (cookies or HTTP authentication).

That’s uncommon for HTTP-requests. Usually, a request to `http://site.com` is accompanied by all cookies from that domain. But cross-origin requests made by JavaScript methods are an exception.

For example, `fetch('http://another.com')` does not send any cookies, even those (!) that belong to `another.com` domain.

Why?

That’s because a request with credentials is much more powerful than without them. If allowed, it grants JavaScript the full power to act on behalf of the user and access sensitive information using their credentials.

Does the server really trust the script that much? Then it must explicitly allow requests with credentials with an additional header.

To send credentials in `fetch`, we need to add the option `credentials: "include"`, like this:

```
fetch('http://another.com', {  
  credentials: "include"  
});
```

Now `fetch` sends cookies originating from `another.com` without request to that site.

If the server agrees to accept the request *with credentials*, it should add a header `Access-Control-Allow-Credentials: true` to the response, in addition to `Access-Control-Allow-Origin`.

For example:

`200 OK`

`Access-Control-Allow-Origin: https://javascript.info`

Access-Control-Allow-Credentials: true

Please note: Access-Control-Allow-Origin is prohibited from using a star * for requests with credentials. Like shown above, it must provide the exact origin there. That's an additional safety measure, to ensure that the server really knows who it trusts to make such requests.

Fetch API

So far, we know quite a bit about fetch. Let's see the rest of API, to cover all its abilities.

Here's the full list of all possible fetch options with their default values (alternatives in comments):

```
let promise = fetch(url, {
  method: "GET", // POST, PUT, DELETE, etc.
  headers: {
    // the content type header value is usually auto-set
    // depending on the request body
    "Content-Type": "text/plain;charset=UTF-8"
  },
  body: undefined // string, FormData, Blob, BufferSource, or URLSearchParams
  referrer: "about:client", // or "" to send no Referer header,
  // or an url from the current origin
  referrerPolicy: "no-referrer-when-downgrade", // no-referrer, origin, same-origin...
  mode: "cors", // same-origin, no-cors
  credentials: "same-origin", // omit, include
  cache: "default", // no-store, reload, no-cache, force-cache, or only-if-cached
  redirect: "follow", // manual, error
  integrity: "", // a hash, like "sha256-abcdef1234567890"
  keepalive: false, // true
  signal: undefined, // AbortController to abort request
  window: window // null
});
```

An impressive list, right?

We fully covered `method`, `headers` and `body` in the chapter `Fetch`.

Now let's explore the rest of capabilities.

referrer, referrerPolicy

These options govern how `fetch` sets HTTP `Referer` header. Usually that header is set automatically

To send no referer, set an empty string:

```
fetch('/page', {  
  referrer: "" // no Referer header  
});
```

To set another url within the current origin:

```
fetch('/page', {  
  // assuming we're on https://javascript.info  
  // we can set any Referer header, but only within the current origin  
  referrer: "https://javascript.info/anotherpage"  
});
```

The `referrerPolicy` option sets general rules for `Referer`.

Requests are split into 3 types:

1. Request to the same origin.
2. Request to another origin.
3. Request from HTTPS to HTTP (from safe to unsafe protocol).

Unlike the `referrer` option that allows to set the exact `Referer` value, `referrerPolicy` tells the browser general rules for each request type.

Possible values are described in the `Referrer Policy` specification:

- **"no-referrer-when-downgrade"** – the default value: full `Referer` is sent always, unless we send a request from HTTPS to HTTP (to less secure protocol).
- **"no-referrer"** – never send `Referer`.

- **"origin"** – only send the origin in Referer, not the full page URL, e.g. only http://site.com instead of http://site.com/path.
- **"origin-when-cross-origin"** – send full Referer to the same origin, but only the origin part for cross-origin requests (as above).
- **"same-origin"** – send full Referer to the same origin, but no referer for for cross-origin requests.
- **"strict-origin"** – send only origin, don't send Referer for HTTPS→HTTP requests.
- **"strict-origin-when-cross-origin"** – for same-origin send full Referer, for cross-origin send only origin, unless it's HTTPS→HTTP request, then send nothing.
- **"unsafe-url"** – always send full url in Referer, even for HTTPS→HTTP requests.

Here's a table with all combinations:

Value	To same origin	To another origin	HTTPS → HTTP
"no-referrer"	-	-	-
"no-referrer-when-downgrade" or "" (default)	full	full	-
"origin"	origin	origin	origin
"origin-when-cross-origin"	full	origin	origin
"same-origin"	full	-	-
"strict-origin"	origin	origin	-
"strict-origin-when-cross-origin"	full	origin	-
"unsafe-url"	full	full	full

Let's say we have an admin zone with URL structure that shouldn't be known from outside of the site.

If we send a fetch, then by default it always sends the Referer header with the full url of our page (except when we request from HTTPS to HTTP, then no Referer).

E.g. Referer: https://javascript.info/admin/secret/paths.

If we'd like other websites know only the origin part, not URL-path, we can set the option:

```
fetch('https://another.com/page', {
  // ...
  referrerPolicy: "origin-when-cross-origin" // Referer: https://javascript.info
});
```

We can put it to all fetch calls, maybe integrate into the JavaScript library of our project that does all requests and uses fetch inside.

Its only difference compared to the default behavior is that for requests to another origin `fetch` sends only the origin part of the URL (e.g. `https://javascript.info`, without path). For requests to our origin we still get the full `Referer` (maybe useful for debugging purposes).

mode

The `mode` option is a safe-guard that prevents occasional cross-origin requests:

- **"cors"** – the default, cross-origin requests are allowed, as described in `Fetch: Cross-Origin Requests`,
- **"same-origin"** – cross-origin requests are forbidden,
- **"no-cors"** – only simple cross-origin requests are allowed.

This option may be useful when the URL for `fetch` comes from a 3rd-party, and we want a “power off switch” to limit cross-origin capabilities.

credentials

The `credentials` option specifies whether `fetch` should send cookies and `HTTP-Authorization` headers with the request.

- **"same-origin"** – the default, don't send for cross-origin requests,
- **"include"** – always send, requires `Accept-Control-Allow-Credentials` from cross-origin server in order for JavaScript to access the response, that was covered in the chapter `Fetch: Cross-Origin Requests`,
- **"omit"** – never send, even for same-origin requests.

cache

By default, `fetch` requests make use of standard HTTP-caching. That is, it honors `Expires`, `Cache-Control` headers, sends `If-Modified-Since`, and so on. Just like regular HTTP-requests do.

The `cache` options allows to ignore HTTP-cache or fine-tune its usage:

- **"default"** – `fetch` uses standard HTTP-cache rules and headers,
- **"no-store"** – totally ignore HTTP-cache, this mode becomes the default if we set a header `If-Modified-Since`, `If-None-Match`, `If-Unmodified-Since`, `If-Match`, or `If-Range`,

- **"reload"** – don't take the result from HTTP-cache (if any), but populate cache with the response (if response headers allow),
- **"no-cache"** – create a conditional request if there is a cached response, and a normal request otherwise. Populate HTTP-cache with the response,
- **"force-cache"** – use a response from HTTP-cache, even if it's stale. If there's no response in HTTP-cache, make a regular HTTP-request, behave normally,
- **"only-if-cached"** – use a response from HTTP-cache, even if it's stale. If there's no response in HTTP-cache, then error. Only works when mode is "same-origin".

redirect

Normally, fetch transparently follows HTTP-redirects, like 301, 302 etc.

The `redirect` option allows to change that:

- **"follow"** – the default, follow HTTP-redirects,
- **"error"** – error in case of HTTP-redirect,
- **"manual"** – don't follow HTTP-redirect, but `response.url` will be the new URL, and `response.redirected` will be `true`, so that we can perform the redirect manually to the new URL (if needed).

integrity

The `integrity` option allows to check if the response matches the known-ahead checksum.

As described in the specification, supported hash-functions are SHA-256, SHA-384, and SHA-512, there might be others depending on a browser.

For example, we're downloading a file, and we know that it's SHA-256 checksum is "abcdef" (a real checksum is longer, of course).

We can put it in the `integrity` option, like this:

```
fetch('http://site.com/file', {
  integrity: 'sha256-abcdef'
});
```

Then fetch will calculate `SHA-256` on its own and compare it with our string. In case of a mismatch, an error is triggered.

keepalive

The `keepalive` option indicates that the request may “outlive” the webpage that initiated it.

For example, we gather statistics about how the current visitor uses our page (mouse clicks, page fragments he views), to analyze and improve user experience.

When the visitor leaves our page – we’d like to save the data at our server.

We can use `window.onunload` event for that:

```
window.onunload = function() {  
    fetch('/analytics', {  
        method: 'POST',  
        body: "statistics",  
        keepalive: true  
    });  
};
```

Normally, when a document is unloaded, all associated network requests are aborted. But `keepalive` option tells the browser to perform the request in background, even after it leaves the page. So this option is essential for our request to succeed.

It has few limitations:

- We can’t send megabytes: the body limit for `keepalive` requests is 64kb.
 - If we gather more data, we can send it out regularly in packets, so that there won’t be a lot left for the last `onunload` request.
 - The limit is for all currently ongoing requests. So we can’t cheat it by creating 100 requests, each 64kb.
- We can’t handle the server response if the request is made on `onunload`, because the document is already unloaded at that time, functions won’t work.
 - Usually, the server sends an empty response to such requests, so it’s not a problem.

URL objects

The built-in `URL` class provides a convenient interface for creating and parsing URLs.

There are no networking methods that require exactly a `URL` object, strings are good enough. So technically we don't have to use `URL`. But sometimes it can be really helpful.

Creating a URL

The syntax to create a new `URL` object:

`new URL(url, [base])`

- **url** – the full URL or only path (if base is set, see below),
- **base** – an optional base URL: if set and url argument has only path, then the URL is generated relative to base.

For example:

```
let url = new URL('https://javascript.info/profile/admin');
```

These two URLs are same:

```
let url1 = new URL('https://javascript.info/profile/admin');
let url2 = new URL('/profile/admin', 'https://javascript.info');
alert(url1); // https://javascript.info/profile/admin
alert(url2); // https://javascript.info/profile/admin
```

We can easily create a new URL based on the path relative to an existing URL:

```
let url = new URL('https://javascript.info/profile/admin');
let newUrl = new URL('tester', url);
alert(newUrl); // https://javascript.info/profile/tester
```

The `URL` object immediately allows us to access its components, so it's a nice way to parse the url, e.g.:

```
let url = new URL('https://javascript.info/url');
alert(url.protocol); // https:
alert(url.host);    // javascript.info
alert(url.pathname); // /url
```

Here's the cheat sheet for URL components:

href					
origin					
protocol	host		pathname	search	hash
	hostname	port			
https:	site.com	8080	/path/page	?p1=v1&p2=v2...	#hash

- href is the full url, same as `url.toString()`
- protocol ends with the colon character :
- search – a string of parameters, starts with the question mark ?
- hash starts with the hash character #
- there may also be user and password properties if HTTP authentication is present:
http://login:password@site.com (not painted above, rarely used)

We can pass URL objects to networking (and most other) methods instead of a string

We can use a URL object in `fetch` or `XMLHttpRequest`, almost everywhere where a URL-string is expected.

Generally, URL objects can be passed to any method instead of a string, as most methods will perform the string conversion, that turns a URL object into a string with full URL.

SearchParams “?...”

Let’s say we want to create a url with given search params, for instance,
`https://google.com/search?query=JavaScript`

We can provide them in the URL string:

```
new URL('https://google.com/search?query=JavaScript')
```

So there’s URL property for that: `url.searchParams`, an object of type `URLSearchParams`.

It provides convenient methods for search parameters:

- **append(name, value)** – add the parameter by name,
- **delete(name)** – remove the parameter by name,
- **get(name)** – get the parameter by name,
- **getAll(name)** – get all parameters with the same name (that's possible, e.g. `?user=John&user=Pete`),
- **has(name)** – check for the existence of the parameter by name,
- **set(name, value)** – set/replace the parameter,
- **sort()** – sort parameters by name, rarely needed,
- ...and it's also iterable, similar to Map.

An example with parameters that contain spaces and punctuation marks:

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!'); // added parameter with a space and !
alert(url); // https://google.com/search?q=test+me%21
url.searchParams.set('tbs', 'qdr:y'); // added parameter with a colon :
// parameters are automatically encoded
alert(url); // https://google.com/search?q=test+me%21&tbs=qdr%3Ay
// iterate over search parameters (decoded)
for(let [name, value] of url.searchParams) {
  alert(`${name}=${value}`); // q=test me!, then tbs=qdr:y
}
```

Encoding

There's a standard RFC3986 that defines which characters are allowed in URLs and which are not.

Those that are not allowed, must be encoded, for instance non-latin letters and spaces – replaced with their UTF-8 codes, prefixed by %, such as %20 (a space can be encoded by +, for historical reasons, but that's an exception).

The good news is that URL objects handle all that automatically. We just supply all parameters unencoded, and then convert the URL to string:

```
// using some cyrillic characters for this example
```

```
let url = new URL('https://ru.wikipedia.org/wiki/Тест');
url.searchParams.set('key', 'б');
alert(url); //https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82?key=%D1%8A
```

As you can see, both `Тест` in the url path and `б` in the parameter are encoded.

The URL became longer, because each cyrillic letter is represented with two bytes in UTF-8, so there are two `%..` entities

Encoding strings

In old times, before URL objects appeared, people used strings for URLs.

As of now, URL objects are often more convenient, but strings can still be used as well. In many cases using a string makes the code shorter.

If we use a string though, we need to encode/decode special characters manually.

There are built-in functions for that:

- `encodeURIComponent` – encodes URL as a whole.
- `decodeURIComponent` – decodes it back.
- `encodeURIComponent` – encodes a URL component, such as a search parameter, or a hash, or a pathname.
- `decodeURIComponent` – decodes it back.

A natural question is: “What’s the difference between `encodeURIComponent` and `encodeURIComponent`?”

When we should use either?”

That’s easy to understand if we look at the URL, that’s split into components in the picture above:

```
https://site.com:8080/path/page?p1=v1&p2=v2#hash
```

As we can see, characters such as `:`, `?`, `=`, `&`, `#` are allowed in the URL.

...On the other hand, if we look at a single URL component, such as a search parameter, these characters must be encoded, not to break the formatting.

- `encodeURIComponent` encodes only characters that are totally forbidden in URL.

- `encodeURIComponent` encodes same characters, and, in addition to them, characters #, \$, &, +, ,, /, :, ;, =, ? and @.

So, for a whole URL we can use `encodeURIComponent`:

```
// using cyrillic characters in url path
let url = encodeURIComponent('http://site.com/привет');
alert(url); // http://site.com/%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D1%82
```

...While for URL parameters we should use `encodeURIComponent` instead:

```
let music = encodeURIComponent('Rock&Roll');
let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock%26Roll
```

Compare it with `encodeURIComponent`:

```
let music = encodeURIComponent('Rock&Roll');
let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock&Roll
```

As we can see, `encodeURIComponent` does not encode &, as this is a legit character in URL as a whole.

But we should encode & inside a search parameter, otherwise, we get `q=Rock&Roll` – that is actually `q=Rock` plus some obscure parameter `Roll`. Not as intended.

So we should use only `encodeURIComponent` for each search parameter, to correctly insert it in the URL string. The safest is to encode both name and value, unless we're absolutely sure that it has only allowed characters.

XMLHttpRequest

`XMLHttpRequest` is a built-in browser object that allows you to make HTTP requests in JavaScript.

Despite having the word “XML” in its name, it can operate on any data, not only in XML format. We can upload/download files, track progress and much more.

Right now, there's another, more modern method `fetch`, that somewhat deprecates `XMLHttpRequest`.

In modern web-development `XMLHttpRequest` is used for three reasons:

1. Historical reasons: we need to support existing scripts with `XMLHttpRequest`.
2. We need to support old browsers, and don't want polyfills (e.g. to keep scripts tiny).
3. We need something that `fetch` can't do yet, e.g. to track upload progress.

Does that sound familiar? If yes, then all right, go on with `XMLHttpRequest`. Otherwise, please head on to `Fetch`.

The basics

`XMLHttpRequest` has two modes of operation: synchronous and asynchronous.

Let's see the asynchronous first, as it's used in the majority of cases.

To do the request, we need 3 steps:

1. Create `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest();
```

The constructor has no arguments.

2. Initialize it, usually right after `new XMLHttpRequest`:

```
xhr.open(method, URL, [async, user, password])
```

This method specifies the main parameters of the request:

- `method` – HTTP-method. Usually "GET" or "POST".
- `URL` – the URL to request, a string, can be URL object.
- `async` – if explicitly set to `false`, then the request is synchronous, we'll cover that a bit later.
- `user, password` – login and password for basic HTTP auth (if required).

Please note that `open` call, contrary to its name, does not open the connection. It only configures the request, but the network activity only starts with the call of `send`.

3. Send it out.

```
xhr.send([body])
```

This method opens the connection and sends the request to the server. The optional `body` parameter contains the request body.

Some request methods like `GET` do not have a body. And some of them, like `POST` use body to send the data to the server. We'll see examples of that later.

4. Listen to xhr events for response.

These three events are the most widely used:

- `load` – when the request is complete (even if HTTP status is like 400 or 500), and the response is fully downloaded.
- `error` – when the request couldn't be made, e.g. network down or invalid URL.
- `progress` – triggers periodically while the response is being downloaded, reports how much has been downloaded.

```
xhr.onload = function() {  
    alert(`Loaded: ${xhr.status} ${xhr.response}`);  
};  
  
xhr.onerror = function() { // only triggers if the request couldn't be made at all  
    alert(`Network Error`);  
};  
  
xhr.onprogress = function(event) { // triggers periodically  
    // event.loaded - how many bytes downloaded  
    // event.lengthComputable = true if the server sent Content-Length header  
    // event.total - total number of bytes (if lengthComputable)  
    alert(`Received ${event.loaded} of ${event.total}`);  
};
```

Here's a full example. The code below loads the URL at `/article/xmlhttprequest/example/load` from the server and prints the progress:

```
// 1. Create a new XMLHttpRequest object  
let xhr = new XMLHttpRequest();  
// 2. Configure it: GET-request for the URL /article/.../load  
xhr.open('GET', '/article/xmlhttprequest/example/load');  
  
// 3. Send the request over the network  
xhr.send();  
  
// 4. This will be called after the response is received
```

```

xhr.onload = function() {
    if (xhr.status != 200) { // analyze HTTP status of the response
        alert(`Error ${xhr.status}: ${xhr.statusText}`); // e.g. 404: Not Found
    } else { // show the result
        alert(`Done, got ${xhr.response.length} bytes`); // responseText is the server
    }
};

xhr.onprogress = function(event) {
    if (event.lengthComputable) {
        alert(`Received ${event.loaded} of ${event.total} bytes`);
    } else {
        alert(`Received ${event.loaded} bytes`); // no Content-Length
    }
};

xhr.onerror = function() {
    alert("Request failed");
};

```

Once the server has responded, we can receive the result in the following `xhr` properties:

status

HTTP status code (a number): 200, 404, 403 and so on, can be 0 in case of a non-HTTP failure.

statusText

HTTP status message (a string): usually OK for 200, Not Found for 404, Forbidden for 403 and so on.

URL search parameters

To add parameters to URL, like `?name=value`, and ensure the proper encoding, we can use `URL` object:

```

let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!');
// the parameter 'q' is encoded
xhr.open('GET', url); // https://google.com/search?q=test+me%21

```

Response Type

We can use `responseType` property to set the response format:

- `""` (default) – get as string,
- `"text"` – get as string,
- `"arraybuffer"` – get as `ArrayBuffer` (for binary data, see chapter `ArrayBuffer`, binary arrays),
- `"blob"` – get as `Blob` (for binary data, see chapter `Blob`),
- `"document"` – get as XML document (can use `XPath` and other XML methods),
- `"json"` – get as JSON (parsed automatically).

For example, let's get the response as JSON:

```
let xhr = new XMLHttpRequest();
xhr.open('GET', '/article/xmlhttprequest/example/json');
xhr.responseType = 'json';
xhr.send();
// the response is {"message": "Hello, world!"}
xhr.onload = function() {
  let responseObj = xhr.response;
  alert(responseObj.message); // Hello, world!
};
```

Ready states

`XMLHttpRequest` changes between states as it progresses. The current state is accessible as `xhr.readyState`.

All states, as in the specification:

```
UNSENT = 0; // initial state
OPENED = 1; // open called
HEADERS_RECEIVED = 2; // response headers received
LOADING = 3; // response is loading (a data packet is received)
DONE = 4; // request complete
```

An XMLHttpRequest object travels in the order $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \dots \rightarrow 3 \rightarrow 4$. State 3 repeats every time a data packet is received over the network.

We can track them using readystatechange event:

```
xhr.onreadystatechange = function() {  
  if (xhr.readyState == 3) {  
    // loading  
  }  
  if (xhr.readyState == 4) {  
    // request finished  
  }  
};
```

You can find readystatechange listeners in really old code, it's there for historical reasons, as there was a time when there were no load and other events. Nowadays, load/error/progress handlers deprecate it.

Aborting request

We can terminate the request at any time. The call to xhr.abort() does that:

```
xhr.abort(); // terminate the request
```

That triggers an abort event, and xhr.status becomes 0.

Synchronous requests

If in the open method the third parameter async is set to false, the request is made synchronously.

In other words, JavaScript execution pauses at send() and resumes when the response is received. Somewhat like alert or prompt commands.

Here's the rewritten example, the 3rd parameter of open is false:

```
let xhr = new XMLHttpRequest();  
xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);  
try {  
  xhr.send();  
  if (xhr.status != 200) {  
    alert(`Error ${xhr.status}: ${xhr.statusText}`);  
  } else {
```

```
    alert(xhr.response);  
  }  
} catch(err) { // instead of onerror  
  alert("Request failed");  
}
```

It might look good, but synchronous calls are used rarely, because they block in-page JavaScript till the loading is complete. In some browsers it becomes impossible to scroll. If a synchronous call takes too much time, the browser may suggest to close the “hanging” webpage.

Many advanced capabilities of XMLHttpRequest, like requesting from another domain or specifying a timeout, are unavailable for synchronous requests. Also, as you can see, no progress indication.

Because of all that, synchronous requests are used very sparingly, almost never. We won't talk about them any more.

HTTP-headers

XMLHttpRequest allows both to send custom headers and read headers from the response.

There are 3 methods for HTTP-headers:

setRequestHeader(name, value)

Sets the request header with the given name and value.

For instance:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

Headers limitations

Several headers are managed exclusively by the browser, e.g. Referer and Host. The full list is in the specification.

XMLHttpRequest is not allowed to change them, for the sake of user safety and correctness of the request.

Can't remove a header

Another peculiarity of XMLHttpRequest is that one can't undo setRequestHeader.

Once the header is set, it's set. Additional calls add information to the header, don't overwrite it.

For instance:

```
xhr.setRequestHeader('X-Auth', '123');  
xhr.setRequestHeader('X-Auth', '456');  
// the header will be:  
// X-Auth: 123, 456
```

getResponseHeader(name)

Gets the response header with the given name (except Set-Cookie and Set-Cookie2).

For instance:

```
xhr.getResponseHeader('Content-Type')
```

getAllResponseHeaders()

Returns all response headers, except Set-Cookie and Set-Cookie2.

Headers are returned as a single line, e.g.:

```
Cache-Control: max-age=31536000  
Content-Length: 4260  
Content-Type: image/png  
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

The line break between headers is always `"\r\n"` (doesn't depend on OS), so we can easily split it into individual headers. The separator between the name and the value is always a colon followed by a space `": "`. That's fixed in the specification.

So, if we want to get an object with name/value pairs, we need to throw in a bit of JS.

Like this (assuming that if two headers have the same name, then the latter one overwrites the former one):

```
let headers = xhr  
  .getAllResponseHeaders()  
  .split("\r\n")  
  .reduce((result, current) => {
```



```
    let [name, value] = current.split(': ');
    result[name] = value;
    return result;
  }, {});
// headers['Content-Type'] = 'image/png'
```

POST, FormData

To make a POST request, we can use the built-in FormData object.

The syntax:

```
let formData = new FormData([form]); // creates an object, optionally fill from <form>
formData.append(name, value); // appends a field
```

We create it, optionally fill from a form, append more fields if needed, and then:

1. `xhr.open('POST', ...)` – use POST method.
2. `xhr.send(formData)` to submit the form to the server.

For instance:

```
<form name="person">
  <input name="name" value="John">
  <input name="surname" value="Smith">
</form>
<script>
  // pre-fill FormData from the form
  let formData = new FormData(document.forms.person);
  // add one more field
  formData.append("middle", "Lee");
  // send it out
  let xhr = new XMLHttpRequest();
  xhr.open("POST", "/article/xmlhttprequest/post/user");
  xhr.send(formData);
  xhr.onload = () => alert(xhr.response);
</script>
```

The form is sent with `multipart/form-data` encoding.

Or, if we like JSON more, then `JSON.stringify` and send as a string.

Just don't forget to set the header `Content-Type: application/json`, many server-side frameworks automatically decode JSON with it:

```
let xhr = new XMLHttpRequest();
let json = JSON.stringify({
  name: "John",
  surname: "Smith"
});
xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');
xhr.send(json);
```

The `.send(body)` method is pretty omnivore. It can send almost anybody, including `Blob` and `BufferSource` objects.

Upload progress

The `progress` event triggers only on the downloading stage.

That is: if we POST something, `XMLHttpRequest` first uploads our data (the request body), then downloads the response.

If we're uploading something big, then we're surely more interested in tracking the upload progress. But `xhr.onprogress` doesn't help here.

There's another object, without methods, exclusively to track upload events: `xhr.upload`.

It generates events, similar to `xhr`, but `xhr.upload` triggers them solely on uploading:

- `loadstart` – upload started.
- `progress` – triggers periodically during the upload.
- `abort` – upload aborted.
- `error` – non-HTTP error.

- `load` – upload finished successfully.
- `timeout` – upload timed out (if `timeout` property is set).
- `loadend` – upload finished with either success or error.

Example of handlers:

```
xhr.upload.onprogress = function(event) {
    alert(`Uploaded ${event.loaded} of ${event.total} bytes`);
};

xhr.upload.onload = function() {
    alert(`Upload finished successfully.`);
};

xhr.upload.onerror = function() {
    alert(`Error during the upload: ${xhr.status}`);
};
```

Here's a real-life example: file upload with progress indication:

```
<input type="file" onchange="upload(this.files[0])">
<script>
    function upload(file) {
        let xhr = new XMLHttpRequest();
        // track upload progress
        xhr.upload.onprogress = function(event) {
            console.log(`Uploaded ${event.loaded} of ${event.total}`);
        };
        // track completion: both successful or not
        xhr.onloadend = function() {
            if (xhr.status == 200) {
                console.log("success");
            } else {
                console.log("error " + this.status);
            }
        }
    }
}
```

```
    };  
    xhr.open("POST", "/article/xmlhttprequest/post/upload");  
    xhr.send(file);  
  }  
</script>
```

Cross-origin requests

XMLHttpRequest can make cross-origin requests, using the same CORS policy as fetch.

Just like fetch, it doesn't send cookies and HTTP-authorization to another origin by default. To enable them, set `xhr.withCredentials` to `true`:

```
let xhr = new XMLHttpRequest();  
xhr.withCredentials = true;  
xhr.open('POST', 'http://anywhere.com/request');  
...
```

Long polling

Long polling is the simplest way of having persistent connection with a server, that doesn't use any specific protocol like WebSocket or Server Side Events.

Being very easy to implement, it's also good enough in a lot of cases.

Regular Polling

The simplest way to get new information from the server is periodic polling. That is, regular requests to the server: "Hello, I'm here, do you have any information for me?". For example, once in 10 seconds.

In response, the server first takes a notice to itself that the client is online, and second – sends a packet of messages it got till that moment.

That works, but there are downsides:

1. Messages are passed with a delay up to 10 seconds (between requests).

2. Even if there are no messages, the server is bombed with requests every 10 seconds, even if the user switched somewhere else or is asleep. That's quite a load to handle, speaking performance-wise.

So, if we're talking about a very small service, the approach may be viable, but generally, it needs an improvement.

Long polling

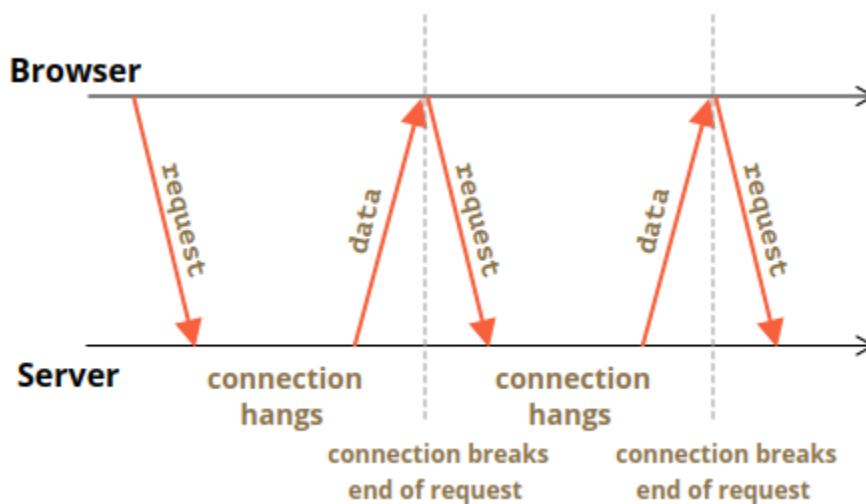
So-called “long polling” is a much better way to poll the server.

It's also very easy to implement, and delivers messages without delays.

The flow:

1. A request is sent to the server.
2. The server doesn't close the connection until it has a message to send.
3. When a message appears – the server responds to the request with it.
4. The browser makes a new request immediately.

The situation when the browser sent a request and has a pending connection with the server, is standard for this method. Only when a message is delivered, the connection is reestablished.



If the connection is lost, because of, say, a network error, the browser immediately sends a new request.

A sketch of client-side `subscribe` function that makes long requests:

```

async function subscribe() {
  let response = await fetch("/subscribe");
  if (response.status == 502) {
    // Status 502 is a connection timeout error,
    // may happen when the connection was pending for too long,
    // and the remote server or a proxy closed it
    // let's reconnect
    await subscribe();
  } else if (response.status != 200) {
    // An error - let's show it
    showMessage(response.statusText);
    // Reconnect in one second
    await new Promise(resolve => setTimeout(resolve, 1000));
    await subscribe();
  } else {
    // Get and show the message
    let message = await response.text();
    showMessage(message);
    // Call subscribe() again to get the next message
    await subscribe();
  }
}
subscribe();

```

As you can see, `subscribe` function makes a fetch, then waits for the response, handles it and calls itself again.

Server should be ok with many pending connections

The server architecture must be able to work with many pending connections.

Certain server architectures run a process per connect. For many connections there will be as many processes, and each process takes a lot of memory. So many connections just consume it all.

Demo: a chat

Here's a demo chat, you can also download it and run locally (if you're familiar with Node.js and can install modules):

Download link: <https://javascript.info/tutorial/zipview/longpoll.zip?plunkId=1LAJIEQpn9jFLCd5>

WebSocket

The WebSocket protocol, described in the specification RFC 6455 provides a way to exchange data between browser and server via a persistent connection. The data can be passed in both directions as “packets”, without breaking the connection and additional HTTP-requests.

WebSocket is especially great for services that require continuous data exchange, e.g. online games, real-time trading systems and so on.

Refer: <https://javascript.info/websocket>

Server Sent Events

The Server-Sent Events specification describes a built-in class `EventSource`, that keeps connection with the server and allows it to receive events from it.

Similar to `WebSocket`, the connection is persistent.

But there are several important differences:

WebSocket	EventSource
Bi-directional: both client and server can exchange messages	One-directional: only server sends data
Binary and text data	Only text
WebSocket protocol	Regular HTTP

`EventSource` is a less-powerful way of communicating with the server than `WebSocket`.

Why should one ever use it?

The main reason: it's simpler. In many applications, the power of `WebSocket` is a little bit too much.

We need to receive a stream of data from the server: maybe chat messages or market prices, or whatever. That's what `EventSource` is good at. Also it supports auto-reconnect, something we need to implement manually with `WebSocket`. Besides, it's a plain old HTTP, not a new protocol.

Getting messages

To start receiving messages, we just need to create `new EventSource(url)`.

The browser will connect to `url` and keep the connection open, waiting for events.

The server should respond with status `200` and the header `Content-Type: text/event-stream`, then keep the connection and write messages into it in the special format, like this:

```
data: Message 1
data: Message 2
data: Message 3
data: of two lines
```

- A message text goes after `data:`, the space after the colon is optional.
- Messages are delimited with double line breaks `\n\n`.
- To send a line break `\n`, we can immediately send one more `data:` (3rd message above).

In practice, complex messages are usually sent JSON-encoded. Line-breaks are encoded as `\n` within them, so multiline `data:` messages are not necessary.

For instance:

```
data: {"user":"John","message":"First line\n Second line"}
```

...So we can assume that one `data:` holds exactly one message.

For each such message, the `message` event is generated:

```
let eventSource = new EventSource("/events/subscribe");
eventSource.onmessage = function(event) {
  console.log("New message", event.data);
  // will log 3 times for the data stream above
};
```



```
// or eventSource.addEventListener('message', ...)
```

Cross-origin requests

EventSource supports cross-origin requests, like `fetch` any other networking methods. We can use any URL:

```
let source = new EventSource("https://another-site.com/events");
```

The remote server will get the `Origin` header and must respond with `Access-Control-Allow-Origin` to proceed.

To pass credentials, we should set the additional option `withCredentials`, like this:

```
let source = new EventSource("https://another-site.com/events", {  
  withCredentials: true  
});
```

Please see the chapter `Fetch: Cross-Origin Requests` for more details about cross-origin headers.

Message id

When a connection breaks due to network problems, either side can't be sure which messages were received, and which weren't.

To correctly resume the connection, each message should have an `id` field, like this:

```
data: Message 1  
id: 1
```

```
data: Message 2  
id: 2
```

```
data: Message 3  
data: of two lines  
id: 3
```

When a message with `id` is received, the browser:

- Sets the property `eventSource.lastEventId` to its value.
- Upon reconnection sends the header `Last-Event-ID` with that id, so that the server may resend following messages.

Connection status: `readyState`

The `EventSource` object has `readyState` property, that has one of three values:

`EventSource.CONNECTING = 0; // connecting or reconnecting`

`EventSource.OPEN = 1; // connected`

`EventSource.CLOSED = 2; // connection closed`

When an object is created, or the connection is down, it's always `EventSource.CONNECTING` (equals 0).

We can query this property to know the state of `EventSource`.

Event types

By default `EventSource` object generates three events:

- `message` – a message received, available as `event.data`.
- `open` – the connection is open.
- `error` – the connection could not be established, e.g. the server returned HTTP 500 status.

The server may specify another type of event with the `event: ...` at the event start.

For example:

`event: join`

`data: Bob`

`data: Hello`

`event: leave`

`data: Bob`

To handle custom events, we must use `addEventListener`, not `onmessage`:

```
eventSource.addEventListener('join', event => {
  alert(`Joined ${event.data}`);
});
```

```

eventSource.addEventListener('message', event => {
  alert(` Said: ${event.data}`);
});
eventSource.addEventListener('leave', event => {
  alert(` Left ${event.data}`);
});

```

Full example

Here's the server that sends messages with 1, 2, 3, then bye and breaks the connection.

Then the browser automatically reconnects.

```

//server.js

let http = require('http');
let url = require('url');
let querystring = require('querystring');
function onDigits(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/event-stream; charset=utf-8',
    'Cache-Control': 'no-cache'
  });
  let i = 0;
  let timer = setInterval(write, 1000);
  write();
  function write() {
    i++;
    if (i == 4) {
      res.write('event: bye\n');
      res.write('data: bye-bye\n\n');
      clearInterval(timer);
      res.end();
      return;
    }
    res.write('data: ' + i + '\n\n');
  }
}

```

```

}
function accept(req, res) {
  if (req.url == '/digits') {
    onDigits(req, res);
    return;
  }
  fileServer.serve(req, res);
}

```

```

if (!module.parent) {
  http.createServer(accept).listen(8080);
} else {
  exports.accept = accept;
}

```

```

// index.html
<!DOCTYPE html>
<script>
  let eventSource;
  function start() { // when "Start" button pressed
    if (!window.EventSource) {
      // IE or an old browser
      alert("The browser doesn't support EventSource.");
      return;
    }
    eventSource = new EventSource('digits');
    eventSource.onopen = function(e) {
      log("Event: open");
    };
    eventSource.onerror = function(e) {
      log("Event: error");
      if (this.readyState == EventSource.CONNECTING) {

```

```

        log(`Reconnecting (readyState=${this.readyState})...`);
    } else {
        log("Error has occurred.");
    }
};

eventSource.addEventListener('bye', function(e) {
    log("Event: bye, data: " + e.data);
});

eventSource.onmessage = function(e) {
    log("Event: message, data: " + e.data);
};
}

function stop() { // when "Stop" button pressed
    eventSource.close();
    log("eventSource.close()");
}

function log(msg) {
    logElem.innerHTML += msg + "<br>";
    document.documentElement.scrollTop = 99999999;
}
</script>
<button onclick="start()">Start</button> Press the "Start" to begin.
<div id="logElem" style="margin: 6px 0"></div>
<button onclick="stop()">Stop</button> "Stop" to finish.

```

Cookies, document.cookie

Cookies are small strings of data that are stored directly in the browser. They are a part of HTTP protocol, defined by RFC 6265 specification.

Cookies are usually set by a web-server using the response `Set-Cookie` HTTP-header. Then the browser automatically adds them to (almost) every request to the same domain using `Cookie` HTTP-header.

One of the most widespread use cases is authentication:

1. Upon sign in, the server uses `Set-Cookie` HTTP-header in the response to set a cookie with a unique “session identifier”.
2. Next time when the request is set to the same domain, the browser sends the cookie over the net using `Cookie` HTTP-header.
3. So the server knows who made the request.

We can also access cookies from the browser, using `document.cookie` property. There are many tricky things about cookies and their options.

Reading from `document.cookie`

Does your browser store any cookies from this site? Let's see:

```
// At javascript.info, we use Google Analytics for statistics,  
// so there should be some cookies  
alert( document.cookie ); // cookie1=value1; cookie2=value2;...
```

The value of `document.cookie` consists of `name=value` pairs, delimited by `;`. Each one is a separate cookie.

To find a particular cookie, we can split `document.cookie` by `;`, and then find the right name. We can use either a regular expression or array functions to do that.

Writing to `document.cookie`

We can write to `document.cookie`. But it's not a data property, it's an accessor (getter/setter). An assignment to it is treated specially.

A write operation to `document.cookie` updates only cookies mentioned in it, but doesn't touch other cookies.

For instance, this call sets a cookie with the name `user` and value `John`:

```
document.cookie = "user=John"; // update only cookie named 'user'  
alert(document.cookie); // show all cookies
```

If you run it, then probably you'll see multiple cookies. That's because `document.cookie=` operation does not overwrite all cookies. It only sets the mentioned cookie `user`.

Technically, name and value can have any characters, to keep the valid formatting they should be escaped using a built-in `encodeURIComponent` function:

```
// special characters (spaces), need encoding
let name = "my name";
let value = "John Smith"
// encodes the cookie as my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURIComponent(value);
alert(document.cookie); // ...; my%20name=John%20Smith
```

Limitations

There are few limitations:

- The `name=value` pair, after `encodeURIComponent`, should not exceed 4kb. So we can't store anything huge in a cookie.
- The total number of cookies per domain is limited to around 20+, the exact limit depends on a browser.

Cookies have several options, many of them are important and should be set.

The options are listed after `key=value`, delimited by `;`, like this:

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT"
```

path

- `path=/mypath`

The url path prefix, the cookie will be accessible for pages under that path. Must be absolute. By default, it's the current path.

If a cookie is set with `path=/admin`, it's visible at pages `/admin` and `/admin/something`, but not at `/home` or `/adminpage`.

Usually, we should set a path to the root: `path=/` to make the cookie accessible from all website pages.

domain

- `domain=site.com`

A domain where the cookie is accessible. In practice though, there are limitations. We can't set any domain.

By default, a cookie is accessible only at the domain that set it. So, if the cookie was set by `site.com`, we won't get it `other.com`.

...But what's more tricky, we also won't get the cookie at a subdomain `forum.site.com`!

```
// at site.com
document.cookie = "user=John"
// at forum.site.com
alert(document.cookie); // no user
```

There's no way to let a cookie be accessible from another 2nd-level domain, so `other.com` will never receive a cookie set at `site.com`.

It's a safety restriction, to allow us to store sensitive data in cookies, that should be available only on one site.

...But if we'd like to allow subdomains like `forum.site.com` get a cookie, that's possible. When setting a cookie at `site.com`, we should explicitly set `domain` option to the root domain:
`domain=site.com`:

```
// at site.com
// make the cookie accessible on any subdomain *.site.com:
document.cookie = "user=John; domain=site.com"
// later
// at forum.site.com
alert(document.cookie); // has cookie user=John
```

For historical reasons, `domain=.site.com` (a dot before `site.com`) also works the same way, allowing access to the cookie from subdomains. That's an old notation, should be used if we need to support very old browsers.

So, `domain` option allows to make a cookie accessible at subdomains.

expires, max-age

By default, if a cookie doesn't have one of these options, it disappears when the browser is closed. Such cookies are called "session cookies"

To let cookies survive browser close, we can set either `expires` or `max-age` option.

- **`expires=`Tue, 19 Jan 2038 03:14:07 GMT**

Cookie expiration date, when the browser will delete it automatically.

The date must be exactly in this format, in GMT timezone. We can use `date.toUTCString` to get it. For instance, we can set the cookie to expire in 1 day:

```
// +1 day from now
```

```
let date = new Date(Date.now() + 86400e3);
```

```
date = date.toUTCString();
```

```
document.cookie = "user=John; expires=" + date;
```

If we set `expires` to a date in the past, the cookie is deleted.

- **`max-age=3600`**

An alternative to `expires`, specifies the cookie expiration in seconds from the current moment.

If zero or negative, then the cookie is deleted:

```
// cookie will die +1 hour from now
```

```
document.cookie = "user=John; max-age=3600";
```

```
// delete cookie (let it expire right now)
```

```
document.cookie = "user=John; max-age=0";
```

secure

- `secure`

The cookie should be transferred only over HTTPS.

By default, if we set a cookie at `http://site.com`, then it also appears at `https://site.com` and vice versa.

That is, cookies are domain-based, they do not distinguish between the protocols.

With this option, if a cookie is set by `https://site.com`, then it doesn't appear when the same site is accessed by HTTP, as `http://site.com`. So if a cookie has sensitive content that should never be sent over unencrypted HTTP, then the flag is the right thing.

```
// assuming we're on https:// now
// set the cookie secure (only accessible if over HTTPS)
document.cookie = "user=John; secure";
```

samesite

That's another security attribute `samesite`. It's designed to protect from so-called XSRF (cross-site request forgery) attacks.

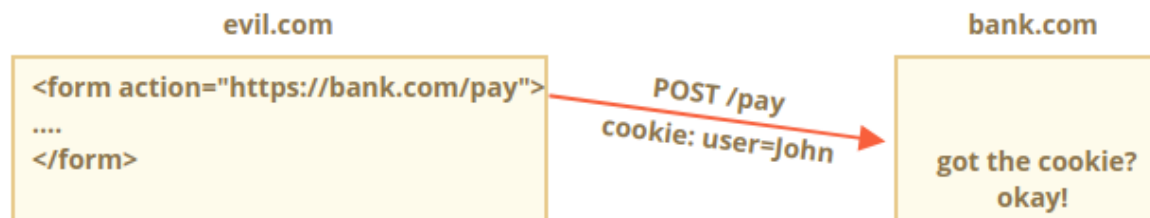
To understand how it works and when it's useful, let's take a look at XSRF attacks.

XSRF attack

Imagine, you are logged into the site `bank.com`. That is: you have an authentication cookie from that site. Your browser sends it to `bank.com` with every request, so that it recognizes you and performs all sensitive financial operations.

Now, while browsing the web in another window, you accidentally come to another site `evil.com`. That site has JavaScript code that submits a form `<form action="https://bank.com/pay">` to `bank.com` with fields that initiate a transaction to the hacker's account.

The browser sends cookies every time you visit the site `bank.com`, even if the form was submitted from `evil.com`. So the bank recognizes you and actually performs the payment.



That's called a "Cross-Site Request Forgery" (in short, XSRF) attack.

Real banks are protected from it of course. All forms generated by `bank.com` have a special field, so called "XSRF protection token", that an evil page can't generate or extract from a

remote page (it can submit a form there, but can't get the data back). And the site `bank.com` checks for such token in every form it receives.

But such protection takes time to implement: we need to ensure that every form has the token field, and we must also check all requests.

Enter cookie `samesite` option

The cookie `samesite` option provides another way to protect from such attacks, that (in theory) should not require "xsrp protection tokens".

It has two possible values:

- **`samesite=strict` (same as `samesite` without value)**

A cookie with `samesite=strict` is never sent if the user comes from outside the same site.

In other words, whether a user follows a link from their mail or submits a form from `evil.com`, or does any operation that originates from another domain, the cookie is not sent.

If authentication cookies have `samesite` option, then XSRF attack has no chances to succeed, because a submission from `evil.com` comes without cookies. So `bank.com` will not recognize the user and will not proceed with the payment.

The protection is quite reliable. Only operations that come from `bank.com` will send the `samesite` cookie, e.g. a form submission from another page at `bank.com`.

Although, there's a small inconvenience.

When a user follows a legitimate link to `bank.com`, like from their own notes, they'll be surprised that `bank.com` does not recognize them. Indeed, `samesite=strict` cookies are not sent in that case.

We could work around that by using two cookies: one for "general recognition", only for the purposes of saying: "Hello, John", and the other one for data-changing operations with `samesite=strict`. Then a person coming from outside of the site will see a welcome, but payments must be initiated from the bank website, for the second cookie to be sent.

- **`samesite=lax`**

A more relaxed approach that also protects from XSRF and doesn't break user experience.

Lax mode, just like `strict`, forbids the browser to send cookies when coming from outside the site, but adds an exception.

A `samesite=lax` cookie is sent if both of these conditions are true:

1. The HTTP method is “safe” (e.g. GET, but not POST).
The full list of safe HTTP methods is in the RFC7231 specification. Basically, these are the methods that should be used for reading, but not writing the data. They must not perform any data-changing operations. Following a link is always GET, the safe method.
2. The operation performs top-level navigation (changes URL in the browser address bar).
That’s usually true, but if the navigation is performed in an `<iframe>`, then it’s not top-level. Also, JavaScript methods for network requests do not perform any navigation, hence they don’t fit.

So, what `samesite=lax` does is basically allows a most common “go to URL” operation to have cookies. E.g. opening a website link from notes satisfies these conditions.

But anything more complicated, like a network request from another site or a form submission loses cookies.

If that’s fine for you, then adding `samesite=lax` will probably not break the user experience and add protection.

Overall, `samesite` is a great option, but it has an important drawback:

- `samesite` is ignored (not supported) by old browsers, year 2017 or so.

So if we solely rely on `samesite` to provide protection, then old browsers will be vulnerable.

But we surely can use `samesite` together with other protection measures, like xsrf tokens, to add an additional layer of defence and then, in the future, when old browsers die out, we’ll probably be able to drop xsrf tokens.

httpOnly

This option has nothing to do with JavaScript, but we have to mention it for completeness.

The web-server uses the `Set-Cookie` header to set a cookie. And it may set the `httpOnly` option.

This option forbids any JavaScript access to the cookie. We can't see such cookie or manipulate them using `document.cookie`.

Appendix: Cookie functions

Here's a small set of functions to work with cookies, more convenient than a manual modification of `document.cookie`.

There exist many cookie libraries for that, so these are for demo purposes. Fully working though.

getCookie(name)

The shortest way to access cookie is to use a regular expression.

The function `getCookie(name)` returns the cookie with the given `name`:

```
// returns the cookie with the given name,  
// or undefined if not found  
function getCookie(name) {  
    let matches = document.cookie.match(new RegExp(  
        "(?:^|; )" + name.replace(/[. $?*\{\}\(\)\[\]\\\V+^]/g, "\\$1') + "=[^;]*"  
    ));  
    return matches ? decodeURIComponent(matches[1]) : undefined;  
}
```

Here `new RegExp` is generated dynamically, to match `; name=<value>`.

Please note that a cookie value is encoded, so `getCookie` uses a built-in `decodeURIComponent` function to decode it.

setCookie(name, value, options)

Sets the cookie `name` to the given `value` with `path=/` by default (can be modified to add other defaults):

```
function setCookie(name, value, options = {}) {  
    options = {  
        path: '/',  

```

```

    // add other defaults here if necessary
    ...options
};
if (options.expires instanceof Date) {
    options.expires = options.expires.toUTCString();
}
let updatedCookie = encodeURIComponent(name) + "=" + encodeURIComponent(value);
for (let optionKey in options) {
    updatedCookie += "; " + optionKey;
    let optionValue = options[optionKey];
    if (optionValue !== true) {
        updatedCookie += "=" + optionValue;
    }
}
document.cookie = updatedCookie;
}
// Example of use:
setCookie('user', 'John', {secure: true, 'max-age': 3600});

```

deleteCookie(name)

To delete a cookie, we can call it with a negative expiration date:

```

function deleteCookie(name) {
    setCookie(name, "", {
        'max-age': -1
    })
}

```

Updating or deleting must use same path and domain

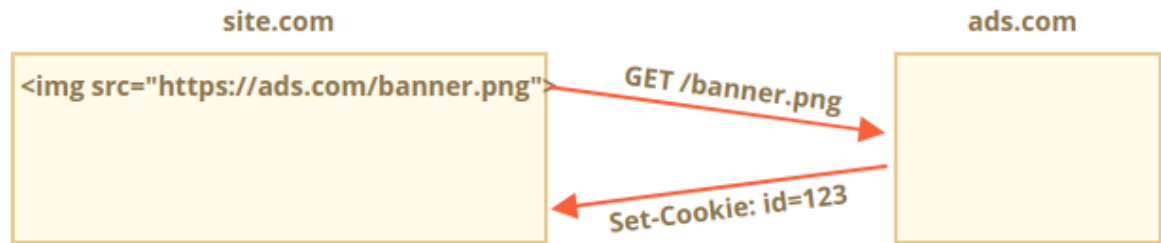
Please note: when we update or delete a cookie, we should use exactly the same path and domain options as when we set it.

Appendix: Third-party cookies

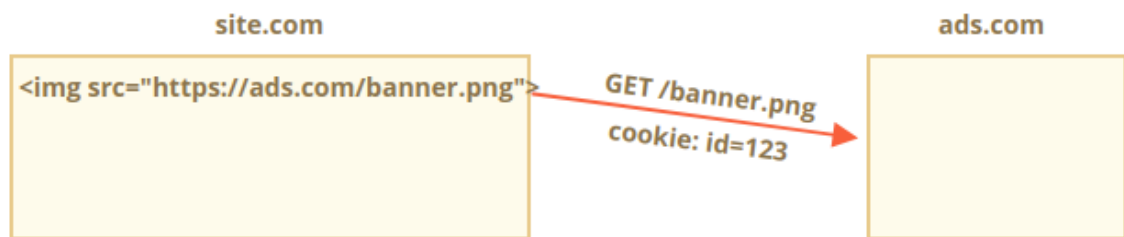
A cookie is called “third-party” if it’s placed by a domain other than the page the user is visiting.

For instance:

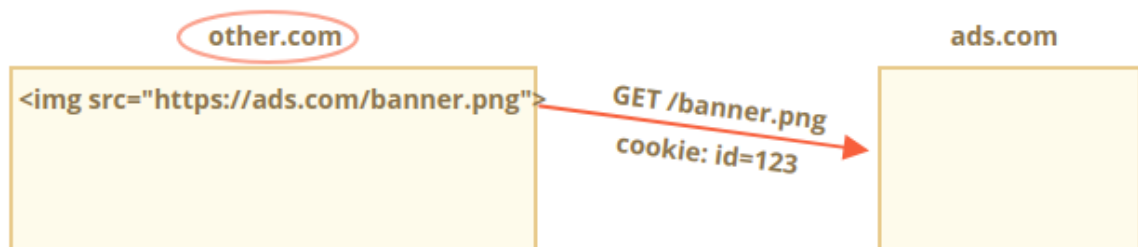
1. A page at `site.com` loads a banner from another site: ``.
2. Along with the banner, the remote server at `ads.com` may set a `Set-Cookie` header with cookie like `id=1234`. Such cookie originates from `ads.com` domain, and will only be visible at `ads.com`:



3. Next time when `ads.com` is accessed, the remote server gets the `id` cookie and recognizes the user:



4. What's even more important, when the users moves from `site.com` to another site `other.com` that also has a banner, then `ads.com` gets the cookie, as it belongs to `ads.com`, thus recognizing the visitor and tracking him as he moves between sites:



Third-party cookies are traditionally used for tracking and ads services, due to their nature. They are bound to the originating domain, so `ads.com` can track the same user between different sites, if they all access it.

Naturally, some people don't like being tracked, so browsers allow you to disable such cookies.

Also, some modern browsers employ special policies for such cookies:

- Safari does not allow third-party cookies at all.
- Firefox comes with a "black list" of third-party domains where it blocks third-party cookies.

Please note:

If we load a script from a third-party domain, like `<script src="https://google-analytics.com/analytics.js">`, and that script uses `document.cookie` to set a cookie, then such cookie is not third-party.

If a script sets a cookie, then no matter where the script came from – the cookie belongs to the domain of the current webpage.

LocalStorage, sessionStorage

Web storage objects `localStorage` and `sessionStorage` allow to save key/value pairs in the browser.

What's interesting about them is that the data survives a page refresh (for `sessionStorage`) and even a full browser restart (for `localStorage`). We'll see that very soon.

We already have cookies. Why additional objects?

- Unlike cookies, web storage objects are not sent to the server with each request. Because of that, we can store much more. Most browsers allow at least 2 megabytes of data (or more) and have settings to configure that.
- Also unlike cookies, the server can't manipulate storage objects via HTTP headers. Everything's done in JavaScript.
- The storage is bound to the origin (domain/protocol/port triplet). That is, different protocols or subdomains infer different storage objects, they can't access data from each other.

Both storage objects provide same methods and properties:

- `setItem(key, value)` – store key/value pair.
- `getItem(key)` – get the value by key.
- `removeItem(key)` – remove the key with its value.
- `clear()` – delete everything.
- `key(index)` – get the key on a given position.
- `length` – the number of stored items.

As you can see, it's like a `Map` collection (`setItem/getItem/removeItem`), but also allows access by index with `key(index)`.

localStorage demo

The main features of `localStorage` are:

- Shared between all tabs and windows from the same origin.
- The data does not expire. It remains after the browser restart and even OS reboot.

For instance, if you run this code...

```
localStorage.setItem('test', 1);
```

...And close/open the browser or just open the same page in a different window, then you can get it like this:

```
alert( localStorage.getItem('test') ); // 1
```

We only have to be on the same origin (domain/port/protocol), the url path can be different.

The `localStorage` is shared between all windows with the same origin, so if we set the data in one window, the change becomes visible in another one.

Object-like access

We can also use a plain object way of getting/setting keys, like this:

```
// set key
localStorage.test = 2;
// get key
```

```
alert( localStorage.test ); // 2
// remove key
delete localStorage.test;
```

That's allowed for historical reasons, and mostly works, but generally not recommended, because:

1. If the key is user-generated, it can be anything, like `length` or `toString`, or another built-in method of `localStorage`. In that case `getItem/setItem` work fine, while object-like access fails:

```
let key = 'length';
localStorage[key] = 5; // Error, can't assign length
```
2. There's a `storage` event, it triggers when we modify the data. That event does not happen for object-like access. We'll see that later in this chapter.

Looping over keys

As we've seen, the methods provide "get/set/remove by key" functionality. But how to get all saved values or keys?

Unfortunately, storage objects are not iterable.

One way is to loop over them as over an array:

```
for(let i=0; i<localStorage.length; i++) {
  let key = localStorage.key(i);
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Another way is to use `for key in localStorage` loop, just as we do with regular objects.

It iterates over keys, but also outputs few built-in fields that we don't need:

```
// bad try
for(let key in localStorage) {
  alert(key); // shows getItem, setItem and other built-in stuff
}
```

...So we need either to filter fields from the prototype with `hasOwnProperty` check:

```
for(let key in localStorage) {
```

```

    if (!localStorage.hasOwnProperty(key)) {
        continue; // skip keys like "setItem", "getItem" etc
    }
    alert(`${key}: ${localStorage.getItem(key)}`);
}

```

...Or just get the “own” keys with `Object.keys` and then loop over them if needed:

```

let keys = Object.keys(localStorage);
for(let key of keys) {
    alert(`${key}: ${localStorage.getItem(key)}`);
}

```

The latter works, because `Object.keys` only returns the keys that belong to the object, ignoring the prototype.

Strings only

Please note that both key and value must be strings.

If were any other type, like a number, or an object, it gets converted to string automatically:

```

sessionStorage.user = {name: "John"};
alert(sessionStorage.user); // [object Object]

```

We can use `JSON` to store objects though:

```

sessionStorage.user = JSON.stringify({name: "John"});
// sometime later
let user = JSON.parse( sessionStorage.user );
alert( user.name ); // John

```

sessionStorage

The `sessionStorage` object is used much less often than `localStorage`.

Properties and methods are the same, but it’s much more limited:

- The `sessionStorage` exists only within the current browser tab.
 - Another tab with the same page will have a different storage.

- But it is shared between iframes in the same tab (assuming they come from the same origin).
- The data survives page refresh, but not closing/opening the tab.
-

Let's see that in action. Run this code...

```
sessionStorage.setItem('test', 1);
```

...Then refresh the page. Now you can still get the data:

```
alert( sessionStorage.getItem('test') ); // after refresh: 1
```

...But if you open the same page in another tab, and try again there, the code above returns null, meaning “nothing found”.

That's exactly because `sessionStorage` is bound not only to the origin, but also to the browser tab. For that reason, `sessionStorage` is used sparingly.

Storage event

When the data gets updated in `localStorage` or `sessionStorage`, storage event triggers, with properties:

- `key` – the key that was changed (null if `.clear()` is called).
- `oldValue` – the old value (null if the key is newly added).
- `newValue` – the new value (null if the key is removed).
- `url` – the url of the document where the update happened.
- `storageArea` – either `localStorage` or `sessionStorage` object where the update happened.

The important thing is: the event triggers on all `window` objects where the storage is accessible, except the one that caused it.

Let's elaborate.

Imagine, you have two windows with the same site in each. So `localStorage` is shared between them.

You might want to open this page in two browser windows to test the code below.

If both windows are listening for `window.onstorage`, then each one will react to updates that happened in the other one.

```
// triggers on updates made to the same storage from other documents
window.onstorage = event => { // same as window.addEventListener('storage', () => {
  if (event.key !== 'now') return;
  alert(event.key + ':' + event.newValue + " at " + event.url);
});
localStorage.setItem('now', Date.now());
```

Please note that the event also contains: `event.url` – the url of the document where the data was updated.

Also, `event.storageArea` contains the storage object – the event is the same for both `sessionStorage` and `localStorage`, so `event.storageArea` references the one that was modified.

That allows different windows from the same origin to exchange messages.

Modern browsers also support the Broadcast channel API, the special API for same-origin inter-window communication; it's more full featured, but less supported. There are libraries that polyfill that API, based on `localStorage`, that make it available everywhere.

Client-side Vs. Server-side Rendering: What to choose when?

The web page rendering dilemma

The discussion about a web page rendering has come to light only in recent years. Earlier, the websites and web applications had a common strategy to follow. They prepared the HTML content to be sent to the browsers at the server-side; this content was then rendered as an HTML with CSS based styling on the browser.

With the advent of JavaScript frameworks, came in a completely different approach to web development. JavaScript frameworks brought in the possibility of shedding burden off the server.

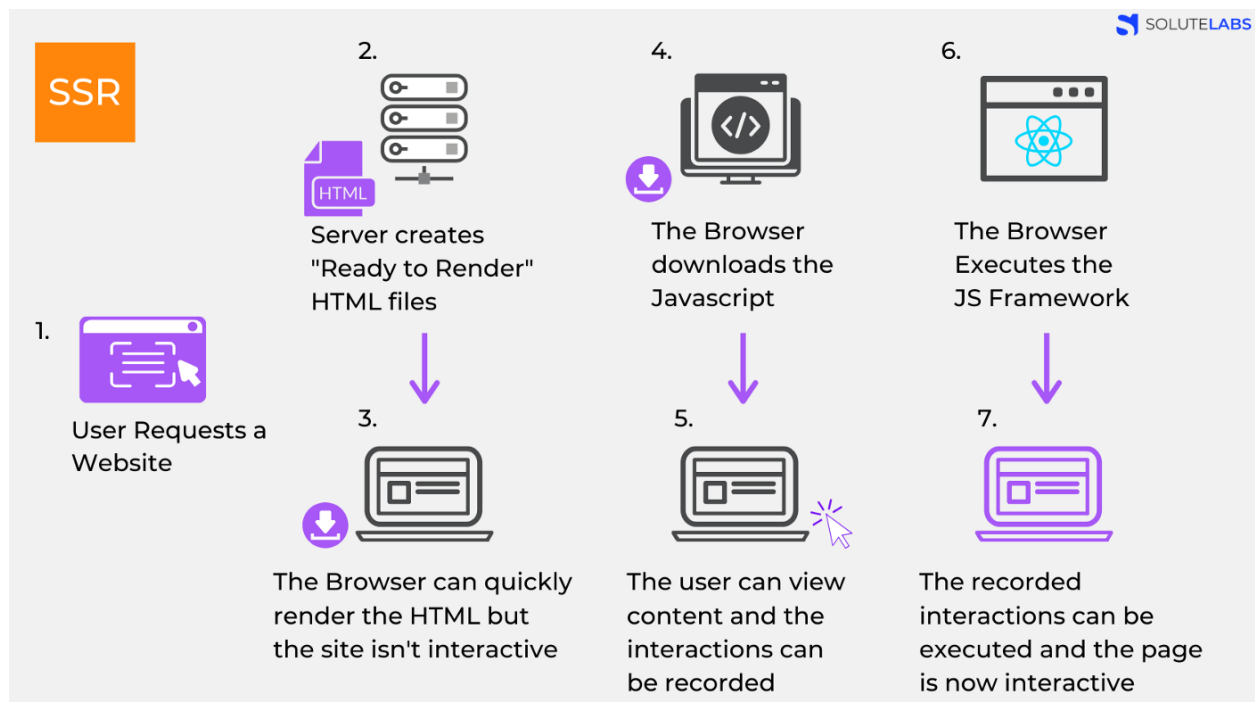
With the power of JavaScript frameworks, it became possible to render dynamic content right from the browser by requesting just for the content that is required. The server, in this scenario, only served with the base HTML wrapper that was necessary. This transformation gave a

seamless user experience to the visitors since there is very little time taken for loading the web page. Moreover, once loaded, the web page does not reload itself again.

What is server-side rendering (SSR)?

Server-side rendering or SSR is the conventional way of rendering web pages on the browser. As discussed above, the traditional way of rendering dynamic web content follows the below steps:

1. The user sends a request to a website (usually via a browser)
2. The server checks the resource, compiles and prepares the HTML content after traversing through server-side scripts lying within the page.
3. This compiled HTML is sent to the client's browser for further rendering and display.
4. The browser downloads the HTML and makes the site visible to the end-user
5. The browser then downloads the Javascript (JS) and as it executes the JS, it makes the page interactive



In this process, all the burden of getting the dynamic content, converting it to HTML, and sending it to the browser remains on the server. Hence, this process is called server-side rendering (SSR).

This responsibility of rendering the complete HTML in advance comes with a burden on memory and processing power on the Server.

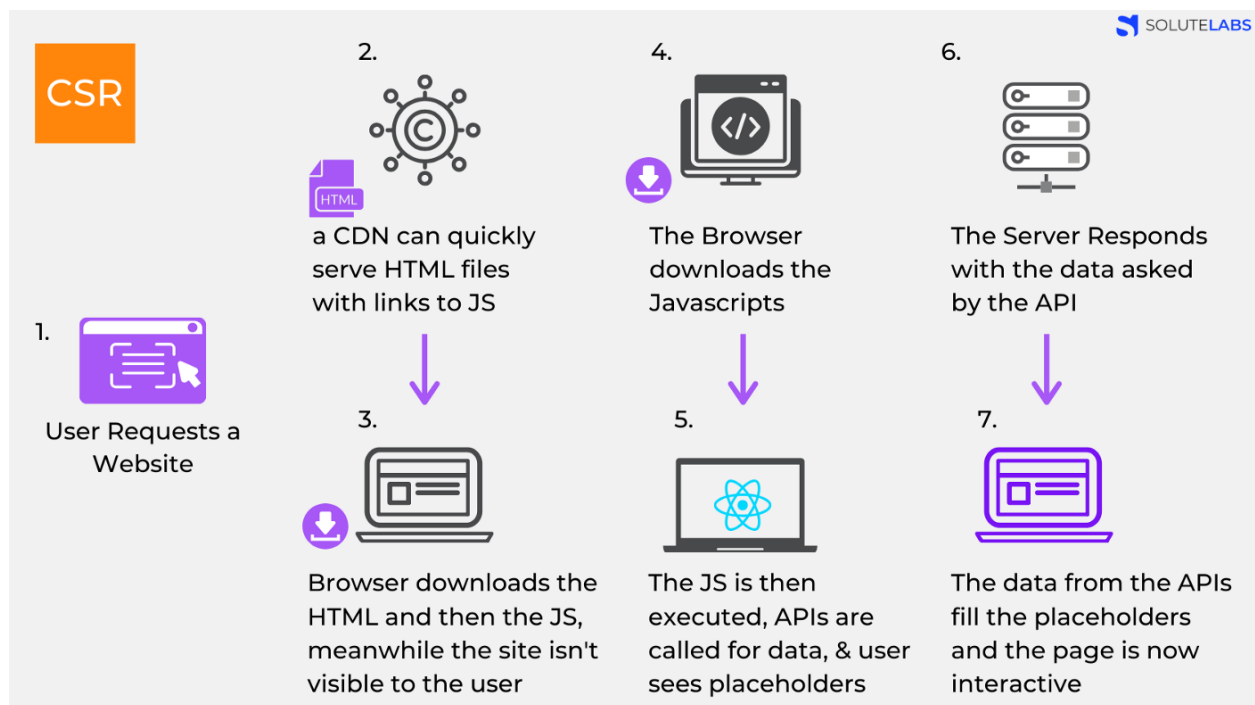
What is client-side rendering (CSR)?

Client-side rendering or CSR is a different approach to how the web page is processed for display on the browser. In the CSR, the burden of compiling dynamic content and generate HTML for them is transferred to the client browser.

This approach is powered with JavaScript frameworks and libraries like ReactJS, VueJS, and Angular. The normal flow of web page rendering for a client-side rendering scenario follows the below steps:

1. The user sends a request to a website (usually via a browser)

2. Instead of a Server, a CDN (Content Delivery Network) can be used to serve static HTML, CSS and supporting files to the user
3. The browser downloads the HTML and then the JS, meanwhile the user sees a loading symbol
4. After the browser fetches the JS, it makes API requests via AJAX to fetch the dynamic content and processes it to render the final content
5. After the server responds, the final content is rendered using DOM processing on the client browser



Since this process involves fetching and processing data on the client front, the process is called client-side rendering.

Client-side rendering (CSR) vs. Server-side rendering (SSR) — Comparison

Since both the approaches are different in the way the content is processed, each method has its benefits making it difficult to decide what to choose- CSR or SSR.

Let us find out the difference between server-side rendering and client-side rendering from a user as well as the web perspective.

Web page load time

The web page load time is the time taken between when a request is sent to the server and when it is rendered on the browser. This is an important aspect when it comes to user experience (UX) for your website or web application. The web page load times for CSR v/s SSR are different in two scenarios:

First page load time

The first page load time is the average time taken when the user loads your website for the first time. On the first load, in CSR, the browser loads base HTML, CSS, and all the scripts required at once and then compile HTML to usable content on the browser.

This time usually is more than getting a pre-compiled HTML and the corresponding scripts from the server. Thus, SSR takes lesser time normally when it comes to the first page load time.

Second and further page load time

The second page load time is the average time taken to navigate from one page to another. In this scenario, since all the supporting scripts are loaded in advance for CSR, the load time is lesser for CSR (and thus better performance). It does not send a request to the server unless a lazy module JavaScript needs to be loaded.

However, for SSR, the complete request cycle followed in the first-page load is repeated. This means that there is hardly any impact on web page load time when it comes to SSR. Thus, in this scenario, CSR responds faster.

Impact on SEO

For a business website, optimizing it for search engines is essential. Search engines read and understand your websites using automated bots called crawlers. These crawlers are interested in the metadata of your website more than the actual content. Hence, it becomes vital that your web page reflects the right metadata for the search engines.

With CSR, the web page content is dynamically generated using JavaScript. This means the change of **metadata from one page to another relies on JavaScript execution**. In the past search engines preferred not to run JavaScript while crawlers crawled through the sites. However, with Google accepting to run JavaScript, the trend is changing.

With CSR, you need to utilize and make an additional effort to ensure that the page metadata changes from one page to the other. This calls for the use of plugins like React Helmet for ReactJS and the use of library modules like Meta from @angular/browser library for Angular framework. You need to put in extra efforts for the metadata to be set for each page and render it on the client-side.

With SSR, the complete page is compiled with the right metadata and sent to the frontend only after getting the final HTML content. This ensures the page metadata is always accurate irrespective of whether the crawler allows the use of JavaScript or not. This makes SSR a better solution to search-engine-optimized pages compared to CSR

Choosing the right path for you

Lesser choices are always the simplest. Conventionally, you had a single choice — SSR. With CSR coming into the picture, the question arises which one is the right one for your application or website. Let us understand where each of them is beneficial.

Dynamic content loading

A server normally resides on a machine with higher compute power and considerably higher networking speeds. With this speed and power, it never runs out of juice while processing the expected number of requests for processing. As a result, the fetching of content on the server front becomes comparatively faster.

Client machines, on the other end, have limited compute power and might take longer for fetching and rendering the dynamic content on the client-side. This means the overall time consumed to get the content rendered will be more. Thus, if your website involves repeated dynamic content rendering, SSR is a better choice over CSR.

Web application UX v/s Website UX

Although they appear almost the same, web applications and websites are two different formats of web content. A web application is a complete application that can be used for purposes like accounting, CRM, HR management, Project management, etc. A website, on the other hand, is informative content about the business.

A web application involves far more user interaction compared to a website since the user performs data entry and generates reports in a web application. In a scenario where user interaction is more, it is crucial to ensure that the clicks don't take long. **So, CSR works better for web applications compared to SSR.**

On the other hand, for a website, a customer is okay if the new web page loads on every click since the caching would typically take care of speeding up the rendering. Moreover, SSR also ensures the right metadata for crawlers — this makes **SSR better for websites compared to CSR.**

Best of Both Worlds

After going through the above, you might be wondering if there were a way to get the benefits of SSR's quick first loads and better SEO performance with a near-native feeling of a CSR. You're in luck! — There are frameworks that work on a hybrid approach such as Gatsby.

What it essentially does is that while the first page is always loaded with SSR, it caches the other pages after the load is done. So, the rest of the pages are pre-rendered and cached making it feel like you're using CSR approach on the subsequent pages! Check out our website, which is also built using Gatsby.

Patterns and flags

Regular expressions are patterns that provide a powerful way to search and replace in text.

In JavaScript, they are available via the RegExp object, as well as being integrated in methods of strings.

Regular Expressions

A regular expression (also “regex”, or just “reg”) consists of a *pattern* and optional *flags*.

There are two syntaxes that can be used to create a regular expression object.

The “long” syntax:

```
regex = new RegExp("pattern", "flags");
```

And the “short” one, using slashes `/`:

```
regex = /pattern/; // no flags
```

```
regex = /pattern/gmi; // with flags g,m and i (to be covered soon)
```

Slashes `/.../` tell JavaScript that we are creating a regular expression. They play the same role as quotes for strings.

In both cases `regex` becomes an instance of the built-in `RegExp` class.

The main difference between these two syntaxes is that pattern using slashes `/.../` does not allow for expressions to be inserted (like string template literals with `${...}`). They are fully static.

Slashes are used when we know the regular expression at the code writing time – and that’s the most common situation. While `new RegExp`, is more often used when we need to create a `regex` “on the fly” from a dynamically generated string. For instance:

```
let tag = prompt("What tag do you want to find?", "h2");
```

```
let regex = new RegExp(`<${tag}>`); // same as /<h2>/ if answered "h2" in the prompt above
```

Flags

Regular expressions may have flags that affect the search.

There are only 6 of them in JavaScript:

i

With this flag the search is case-insensitive: no difference between `A` and `a` (see the example below).

g

With this flag the search looks for all matches, without it – only the first match is returned.

m

Multiline mode (covered in the chapter Multiline mode of anchors `^` `$`, flag `"m"`).

s

Enables “dotall” mode, that allows a dot `.` to match newline character `\n` (covered in the chapter Character classes).

u

Enables full unicode support. The flag enables correct processing of surrogate pairs. More about that in the chapter Unicode: flag “u” and class `\p{...}`.

y

“Sticky” mode: searching at the exact position in the text (covered in the chapter Sticky flag “y”, searching at position)

Searching: str.match

As mentioned previously, regular expressions are integrated with string methods.

The method `str.match(regex)` finds all matches of `regex` in the string `str`.

It has 3 working modes:

- If the regular expression has flag `g`, it returns an array of all matches:

```
let str = "We will, we will rock you";  
alert( str.match(/we/gi) ); // We,we (an array of 2 substrings that match)
```

Please note that both `We` and `we` are found, because flag `i` makes the regular expression case-insensitive.
- If there's no such flag it returns only the first match in the form of an array, with the full match at index `0` and some additional details in properties:

```
let str = "We will, we will rock you";  
let result = str.match(/we/i); // without flag g  
alert( result[0] ); // We (1st match)  
alert( result.length ); // 1  
// Details:  
alert( result.index ); // 0 (position of the match)
```

```
alert( result.input ); // We will, we will rock you (source string)
```

The array may have other indexes, besides 0 if a part of the regular expression is enclosed in parentheses. We'll cover that in the chapter Capturing groups.

- And, finally, if there are no matches, `null` is returned (doesn't matter if there's flag `g` or not).

This is a very important nuance. If there are no matches, we don't receive an empty array, but instead receive `null`. Forgetting about that may lead to errors, e.g.:

```
let matches = "JavaScript".match(/HTML/); // = null
if (!matches.length) { // Error: Cannot read property 'length' of null
  alert("Error in the line above");
}
```

If we'd like the result to always be an array, we can write it this way:

```
let matches = "JavaScript".match(/HTML/) || [];
if (!matches.length) {
  alert("No matches"); // now it works
}
```

Replacing: `str.replace`

The method `str.replace(regex, replacement)` replaces matches found using `regex` in string `str` with `replacement` (all matches if there's flag `g`, otherwise, only the first one).

For instance:

```
// no flag g
alert( "We will, we will".replace(/we/i, "I") ); // I will, we will
```

```
// with flag g
alert( "We will, we will".replace(/we/ig, "I") ); // I will, I will
```

The second argument is the replacement string. We can use special character combinations in it to insert fragments of the match:

Symbols	Action in the replacement string
<code>\$&</code>	inserts the whole match
<code>\$`</code>	inserts a part of the string before the match
<code>\$'</code>	inserts a part of the string after the match
<code>\$n</code>	if <code>n</code> is a 1-2 digit number, then it inserts the contents of <code>n</code> -th parentheses, more about it in the chapter Capturing groups
<code>\$<name></code>	inserts the contents of the parentheses with the given <code>name</code> , more about it in the chapter Capturing groups
<code>\$\$</code>	inserts character <code>\$</code>

An example with `$&`

```
alert( "I love HTML".replace(/HTML/, "$& and JavaScript") ); // I love HTML and JavaScript
```

Testing: `regexp.test`

The method `regexp.test(str)` looks for at least one match, if found, returns `true`, otherwise `false`.

```
let str = "I love JavaScript";
let regexp = /LOVE/i;
alert( regexp.test(str) ); // true
```

Character classes

Consider a practical task – we have a phone number like `" +7(903)-123-45-67"`, and we need to turn it into pure numbers: `79031234567`.

To do so, we can find and remove anything that's not a number. Character classes can help with that.

A *character class* is a special notation that matches any symbol from a certain set.

For the start, let's explore the "digit" class. It's written as `\d` and corresponds to "any single digit".

For instance, let's find the first digit in the phone number:

```
let str = "+7(903)-123-45-67";
let regexp = /\d/;
alert( str.match(regexp) ); // 7
```

Without the flag `g`, the regular expression only looks for the first match, that is the first digit `\d`.

Let's add the `g` flag to find all digits:

```
let str = "+7(903)-123-45-67";
let regexp = /\d/g;
alert( str.match(regexp) ); // array of matches: 7,9,0,3,1,2,3,4,5,6,7
// let's make the digits-only phone number of them:
alert( str.match(regexp).join("") ); // 79031234567
```

That was a character class for digits. There are other character classes as well.

Most used are:

`\d` (“d” is from “digit”)

A digit: a character from 0 to 9.

`\s` (“s” is from “space”)

A space symbol: includes spaces, tabs `\t`, newlines `\n` and few other rare characters, such as `\v`, `\f` and `\r`.

`\w` (“w” is from “word”)

A “wordly” character: either a letter of Latin alphabet or a digit or an underscore `_`. Non-Latin letters (like cyrillic or hindi) do not belong to `\w`.

For instance, `\d\s\w` means a “digit” followed by a “space character” followed by a “wordly character”, such as `1 a`.

A regexp may contain both regular symbols and character classes.

For instance, `CSS\d` matches a string `CSS` with a digit after it:

```
let str = "Is there CSS4?";
let regexp = /CSS\d/
alert( str.match(regexp) ); // CSS4
```

Also we can use many character classes:

```
alert( "I love HTML5!".match(/\s\w\w\w\w\d/) ); // ' HTML5'
```


The match (each regexp character class has the corresponding result character):

I love HTML5

(Note: In the original image, the string "love" is highlighted in orange, and "HTML5" is highlighted in brown. Above "love", the text "\s \w \w \w \w \d" is written in orange, indicating the match for each character in the string.)

Inverse classes

For every character class there exists an “inverse class”, denoted with the same letter, but uppercased.

The “inverse” means that it matches all other characters, for instance:

\D

Non-digit: any character except `\d`, for instance a letter.

\S

Non-space: any character except `\s`, for instance a letter.

\W

Non-wordly character: anything but `\w`, e.g a non-latin letter or a space.

In the beginning of the chapter we saw how to make a number-only phone number from a string like `+7(903)-123-45-67`: find all digits and join them.

```
let str = "+7(903)-123-45-67";  
alert( str.match(/\d/g).join("") ); // 79031234567
```

An alternative, shorter way is to find non-digits `\D` and remove them from the string:

```
let str = "+7(903)-123-45-67";  
alert( str.replace(/\D/g, "") ); // 79031234567
```

A dot is “any character”

A dot `.` is a special character class that matches “any character except a newline”.

For instance:

```
alert( "Z".match(/./) ); // Z
```

Or in the middle of a regexp:

```
let regexp = /CS.4/;  
alert( "CSS4".match(regexp) ); // CSS4  
alert( "CS-4".match(regexp) ); // CS-4  
alert( "CS 4".match(regexp) ); // CS 4 (space is also a character)
```

Please note that a dot means “any character”, but not the “absence of a character”. There must be a character to match it:

```
alert( "CS4".match(/CS.4/) ); // null, no match because there's no character for the dot
```

Dot as literally any character with “s” flag

By default, a dot doesn't match the newline character `\n`.

For instance, the regexp `A.B` matches `A`, and then `B` with any character between them, except a newline `\n`:

```
alert( "A\nB".match(/A.B/) ); // null (no match)
```

There are many situations when we'd like a dot to mean literally “any character”, newline included.

That's what flag `s` does. If a regexp has it, then a dot `.` matches literally any character:

```
alert( "A\nB".match(/A.B/s) ); // A\nB (match!)
```

Pay attention to spaces

Usually we pay little attention to spaces. For us strings `1-5` and `1 - 5` are nearly identical. But if a regexp doesn't take spaces into account, it may fail to work. Let's try to find digits separated by a hyphen:

```
alert( "1 - 5".match(/\d-\d/) ); // null, no match!
```

Let's fix it adding spaces into the regexp `\d - \d`:

```
alert( "1 - 5".match(/\d - \d/) ); // 1 - 5, now it works
```

// or we can use `\s` class:

```
alert( "1 - 5".match(/\d\s-\s\d/) ); // 1 - 5, also works
```

A space is a character. Equal in importance with any other character.

We can't add or remove spaces from a regular expression and expect to work the same.

In other words, in a regular expression all characters matter, spaces too.

Anchors: string start ^ and end \$

The caret ^ and dollar \$ characters have special meaning in a regexp. They are called “anchors”.

The caret ^ matches at the beginning of the text, and the dollar \$ – at the end.

For instance, let's test if the text starts with Mary:

```
let str1 = "Mary had a little lamb";  
alert( /^Mary/.test(str1) ); // true
```

The pattern ^Mary means: “string start and then Mary”.

Similar to this, we can test if the string ends with snow using snow\$:

```
let str1 = "it's fleece was white as snow";  
alert( /snow$/.test(str1) ); // true
```

In these particular cases we could use string methods `startsWith/endsWith` instead. Regular expressions should be used for more complex tests.

Testing for a full match

Both anchors together ^...\$ are often used to test whether or not a string fully matches the pattern. For instance, to check if the user input is in the right format.

Let's check whether or not a string is a time in 12:34 format. That is: two digits, then a colon, and then another two digits.

In regular expressions language that's \d\d:\d\d:

```
let goodInput = "12:34";  
let badInput = "12:345";  
let regexp = /^\\d\\d:\\d\\d$/;  
alert( regexp.test(goodInput) ); // true
```

```
alert( regexp.test(badInput) ); // false
```

Here the match for `\d\d:\d\d` must start exactly after the beginning of the text `^`, and the end `$` must immediately follow.

The whole string must be exactly in this format. If there's any deviation or an extra character, the result is false. Anchors behave differently if flag `m` is present.

Multiline mode of anchors `^`, `$`, flag `"m"`

The multiline mode is enabled by the flag `m`.

It only affects the behavior of `^` and `$`.

In the multiline mode they match not only at the beginning and the end of the string, but also at start/end of line.

Searching at line start `^`

In the example below the text has multiple lines. The pattern `/^d/gm` takes a digit from the beginning of each line:

```
let str = `1st place: Winnie
2nd place: Piglet
3rd place: Eeyore`;
alert( str.match(/^d/gm) ); // 1, 2, 3
```

Without the flag `m` only the first digit is matched:

```
let str = `1st place: Winnie
2nd place: Piglet
3rd place: Eeyore`;
alert( str.match(/^d/g) ); // 1
```

That's because by default a caret `^` only matches at the beginning of the text, and in the multiline mode – at the start of any line.

Please note:

“Start of a line” formally means “immediately after a line break”: the test `^` in multiline mode matches at all positions preceded by a newline character `\n`.

And at the text start.

Searching at line end `$`

The dollar sign `$` behaves similarly.

The regular expression `\d$` finds the last digit in every line

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;
alert( str.match(/\d$/gm) ); // 1,2,3
```

Without the flag `m`, the dollar `$` would only match the end of the whole text, so only the very last digit would be found.

Please note:

“End of a line” formally means “immediately before a line break”: the test `$` in multiline mode matches at all positions succeeded by a newline character `\n`.

And at the text end

Searching for `\n` instead of `^` `$`

To find a newline, we can use not only anchors `^` and `$`, but also the newline character `\n`.

What's the difference? Let's see an example.

Here we search for `\d\n` instead of `\d$`:

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;
```

```
alert( str.match(/\d\n/gm) ); // 1\n,2\n
```

As we can see, there are 2 matches instead of 3.

That's because there's no newline after 3 (there's text end though, so it matches \$).

Another difference: now every match includes a newline character `\n`. Unlike the anchors `^` and `$`, that only test the condition (start/end of a line), `\n` is a character, so it becomes a part of the result.

So, a `\n` in the pattern is used when we need newline characters in the result, while anchors are used to find something at the beginning/end of a line.

Word boundary: `\b`

A word boundary `\b` is a test, just like `^` and `$`.

When the regexp engine (program module that implements searching for regexps) comes across `\b`, it checks that the position in the string is a word boundary.

There are three different positions that qualify as word boundaries:

- At string start, if the first string character is a word character `\w`.
- Between two characters in the string, where one is a word character `\w` and the other is not.
- At string end, if the last string character is a word character `\w`.

For instance, regexp `\bJava\b` will be found in `Hello, Java!`, where `Java` is a standalone word, but not in `Hello, JavaScript!`.

```
alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, JavaScript!".match(/\bJava\b/) ); // null
```

In the string `Hello, Java!` following positions correspond to `\b`:



Hello, Java!

So, it matches the pattern `\bHello\b`, because:

1. At the beginning of the string matches the first test `\b`.
2. Then matches the word `Hello`.
3. Then the test `\b` matches again, as we're between `o` and a space.

The pattern `\bJava\b` would also match. But not `\bHell\b` (because there's no word boundary after `l`) and not `Java!\b` (because the exclamation sign is not a wordly character `\w`, so there's no word boundary after it).

```
alert( "Hello, Java!".match(/\bHello\b/) ); // Hello
alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, Java!".match(/\bHell\b/) ); // null (no match)
alert( "Hello, Java!".match(/\bJava!\b/) ); // null (no match)
```

We can use `\b` not only with words, but with digits as well.

For example, the pattern `\b\d\d\b` looks for standalone 2-digit numbers. In other words, it looks for 2-digit numbers that are surrounded by characters different from `\w`, such as spaces or punctuation (or text start/end).

```
alert( "1 23 456 78".match(/\b\d\d\b/g) ); // 23,78
alert( "12,34,56".match(/\b\d\d\b/g) ); // 12,34,56
```

Escaping, special characters

As we've seen, a backslash `\` is used to denote character classes, e.g. `\d`. So it's a special character in regexps (just like in regular strings).

There are other special characters as well, that have special meaning in a regexp. They are used to doing more powerful searches. Here's a full list of them: `[\ ^ $. | ? * + ()`.

Don't try to remember the list – soon we'll deal with each of them separately and you'll know them by heart automatically.

Escaping

Let's say we want to find literally a dot. Not "any character", but just a dot. To use a special character as a regular one, prepend it with a backslash: `\.`

That's also called "escaping a character".

For example:

```
alert( "Chapter 5.1".match(/\d\.\d/) ); // 5.1 (match!)  
alert( "Chapter 511".match(/\d\.\d/) ); // null (looking for a real dot \.)
```

Parentheses are also special characters, so if we want them, we should use `\(`. The example below looks for a string `"g()"`:

```
alert( "function g()".match(/g\(\)/) ); // "g()"
```

If we're looking for a backslash `\`, it's a special character in both regular strings and regexps, so we should double it.

```
alert( "1\\2".match(/\\/) ); // \"
```

A slash

A slash symbol `/` is not a special character, but in JavaScript it is used to open and close the regexp: `/...pattern.../`, so we should escape it too.

Here's what a search for a slash `/` looks like:

```
alert( "/" .match(/\//) ); // '/'
```

On the other hand, if we're not using `/.../`, but create a regexp using `new RegExp`, then we don't need to escape it:

```
alert( "/" .match(new RegExp("/")) ); // finds /
```

new RegExp

If we are creating a regular expression with `new RegExp`, then we don't have to escape `/`, but need to do some other escaping.

For instance, consider this:


```
let regexp = new RegExp("\\d\\.\\d");
alert( "Chapter 5.1".match(regexp) ); // null
```

The similar search in one of previous examples worked with `^\\d\\.\\d/`, but `new RegExp("\\d\\.\\d")` doesn't work, why?

The reason is that backslashes are “consumed” by a string. As we may recall, regular strings have their own special characters, such as `\\n`, and a backslash is used for escaping.

Here's how `"\\d\\.\\d"` is perceived:

```
alert("\\d\\.\\d"); // d.d
```

String quotes “consume” backslashes and interpret them on their own, for instance:

- `\\n` – becomes a newline character,
- `\\u1234` – becomes the Unicode character with such code,
- ...And when there's no special meaning: like `\\d` or `\\z`, then the backslash is simply removed.

So `new RegExp` gets a string without backslashes. That's why the search doesn't work!

To fix it, we need to double backslashes, because string quotes turn `\\` into `\\`:

```
let regStr = "\\d\\.\\d";
alert(regStr); // \\d\\.\\d (correct now)
let regexp = new RegExp(regStr);
alert( "Chapter 5.1".match(regexp) ); // 5.1
```

Sets and ranges [...]

Several characters or character classes inside square brackets [...] mean to “search for any character among given”.

Sets

For instance, `[eao]` means any of the 3 characters: `'a'`, `'e'`, or `'o'`.

That's called a *set*. Sets can be used in a regexp along with regular characters:

```
// find [t or m], and then "op"
alert( "Mop top".match(/[tm]op/gi) ); // "Mop", "top"
```

Please note that although there are multiple characters in the set, they correspond to exactly one character in the match.

So the example below gives no matches:

```
// find "V", then [o or i], then "la"  
alert( "Voila".match(/V[oi]la/) ); // null, no matches
```

The pattern searches for:

- V,
- then *one* of the letters [oi],
- then la.

So there would be a match for Vola or Vila.

Ranges

Square brackets may also contain *character ranges*.

For instance, [a-z] is a character in range from a to z, and [0-5] is a digit from 0 to 5.

In the example below we're searching for "x" followed by two digits or letters from A to F:

```
alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) ); // xAF
```

Here [0-9A-F] has two ranges: it searches for a character that is either a digit from 0 to 9 or a letter from A to F.

If we'd like to look for lowercase letters as well, we can add the range a-f: [0-9A-Fa-f]. Or add the flag i.

We can also use character classes inside [...].

For instance, if we'd like to look for a wordly character \w or a hyphen -, then the set is [\w-].

Combining multiple classes is also possible, e.g. [\s\d] means "a space character or a digit".

Character classes are shorthands for certain character sets

For instance:

- `\d` – is the same as `[0-9]`,
- `\w` – is the same as `[a-zA-Z0-9_]`,
- `\s` – is the same as `[\t\n\v\f\r]`, plus a few other rare unicode space characters.

Excluding ranges

Besides normal ranges, there are “excluding” ranges that look like `[^...]`. They are denoted by a caret character `^` at the start and match any character *except the given ones*.

For instance:

- `[^aeyo]` – any character except 'a', 'e', 'y' or 'o'.
- `[^0-9]` – any character except a digit, the same as `\D`.
- `[^s]` – any non-space character, same as `\S`.

The example below looks for any characters except letters, digits and spaces:

```
alert( "alice15@gmail.com".match(/[^d\sA-Z]/gi) ); // @ and .
```

Escaping in [...]

Usually when we want to find exactly a special character, we need to escape it like `\.`. And if we need a backslash, then we use `\\`, and so on.

In square brackets we can use the vast majority of special characters without escaping:

- Symbols `.` `+` `()` never need escaping.
- A hyphen `-` is not escaped in the beginning or the end (where it does not define a range).
- A caret `^` is only escaped in the beginning (where it means exclusion).
- The closing square bracket `]` is always escaped (if we need to look for that symbol).

In other words, all special characters are allowed without escaping, except when they mean something for square brackets.

A dot `.` inside square brackets means just a dot. The pattern `[.,]` would look for one of characters: either a dot or a comma.

In the example below the regexp `[-().^+]` looks for one of the characters `-().^+`:

```
// No need to escape
```

```
let regexp = /[(-).^+]/g;
alert( "1 + 2 - 3".match(regexp) ); // Matches +, -
```

Quantifiers +, *, ? and {n}

Let's say we have a string like `+7(903)-123-45-67` and want to find all numbers in it. But unlike before, we are interested not in single digits, but full numbers: 7, 903, 123, 45, 67.

A number is a sequence of 1 or more digits `\d`. To mark how many we need, we can append a *quantifier*.

Quantity {n}

The simplest quantifier is a number in curly braces: `{n}`. A quantifier is appended to a character (or a character class, or a [...] set etc) and specifies how many we need.

It has a few advanced forms, let's see examples:

The exact count: {5}

`\d{5}` denotes exactly 5 digits, the same as `\d\d\d\d\d`.

The example below looks for a 5-digit number:

```
alert( "I'm 12345 years old".match(/\d{5}/) ); // "12345"
```

We can add `\b` to exclude longer numbers: `\b\d{5}\b`.

The range: {3,5}, match 3-5 times

To find numbers from 3 to 5 digits we can put the limits into curly braces: `\d{3,5}`

```
alert( "I'm not 12, but 1234 years old".match(/\d{3,5}/) ); // "1234"
```

We can omit the upper limit.

Then a regexp `\d{3,}` looks for sequences of digits of length 3 or more:

```
alert( "I'm not 12, but 345678 years old".match(/\d{3,}/) ); // "345678"
```

Let's return to the string `+7(903)-123-45-67`.

A number is a sequence of one or more digits in a row. So the regexp is `\d{1,}`:

```
let str = "+7(903)-123-45-67";
let numbers = str.match(/\d{1,}/g);
alert(numbers); // 7,903,123,45,67
```

Shorthands

There are shorthands for most used quantifiers:

+

Means “one or more”, the same as {1,}. For instance, `\d+` looks for numbers:

```
let str = "+7(903)-123-45-67";  
alert( str.match(/\d+/g) ); // 7,903,123,45,67
```

?

Means “zero or one”, the same as {0,1}. In other words, it makes the symbol optional. For instance, the pattern `ou?r` looks for `o` followed by zero or one `u`, and then `r`.

So, `colou?r` finds both `color` and `colour`:

```
let str = "Should I write color or colour?";  
alert( str.match(/colou?r/g) ); // color, colour
```

Means “zero or more”, the same as {0,}. That is, the character may repeat any times or be absent.

For example, `\d0*` looks for a digit followed by any number of zeroes (may be many or none):

```
alert( "100 10 1".match(/\d0*/g) ); // 100, 10, 1
```

Compare it with `+` (one or more):

```
alert( "100 10 1".match(/\d0+/g) ); // 100, 10  
// 1 not matched, as 0+ requires at least one zero
```

More examples

Quantifiers are used very often. They serve as the main “building block” of complex regular expressions, so let’s see more examples.

Regexp for decimal fractions (a number with a floating point): `\d+\.\d+`

In action:

```
alert( "0 1 12.345 7890".match(/\d+\.\d+/g) ); // 12.345
```

Regexp for an “opening HTML-tag without attributes”, such as `` or `<p>`.

1. The simplest one: `/<[a-z]+>/i`

```
alert( "<body> ... </body>".match(/<[a-z]+>/gi) ); // <body>
```

The regexp looks for character '`<`' followed by one or more Latin letters, and then '`>`'.

2. Improved: `/<[a-z][a-z0-9]*>/i`

According to the standard, HTML tag name may have a digit at any position except the first one, like `<h1>`.

```
alert( "<h1>Hi!</h1>".match(/<[a-z][a-z0-9]*>/gi) ); // <h1>
```

Regexp “opening or closing HTML-tag without attributes”: `/<V?[a-z][a-z0-9]*>/i`

We added an optional slash `/?` near the beginning of the pattern. Had to escape it with a backslash, otherwise JavaScript would think it is the pattern end.

```
alert( "<h1>Hi!</h1>".match(/<V?[a-z][a-z0-9]*>/gi) ); // <h1>, </h1>
```

Greedy and lazy quantifiers

Quantifiers are very simple from the first sight, but in fact they can be tricky.

We should understand how the search works very well if we plan to look for something more complex than `\d+/.`

Let's take the following task as an example.

We have a text and need to replace all quotes `"..."` with guillemet marks: `«...»`. They are preferred for typography in many countries.

For instance: `"Hello, world"` should become `«Hello, world»`. There exist other quotes, such as `„Witam, świat!”`(Polish) or `「你好，世界」` (Chinese), but for our task let's choose `«...»`.

The first thing to do is to locate quoted strings, and then we can replace them.

A regular expression like `/".+"/g` (a quote, then something, then the other quote) may seem like a good fit, but it isn't!

Let's try it:

```
let regexp = /".+"/g;
let str = 'a "witch" and her "broom" is one';
alert( str.match(regexp) ); // "witch" and her "broom"
```

...We can see that it works not as intended!

Instead of finding two matches "witch" and "broom", it finds one: "witch" and her "broom".

That can be described as “greediness is the cause of all evil”.

Greedy search

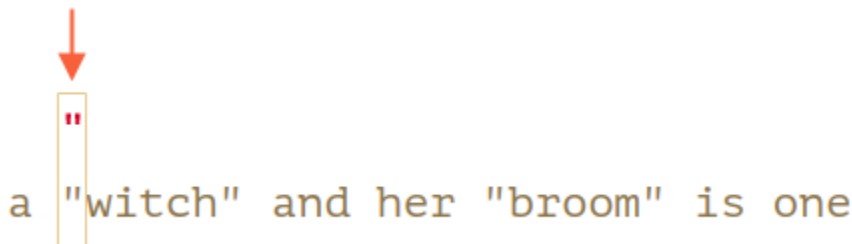
To find a match, the regular expression engine uses the following algorithm:

- For every position in the string
 - Try to match the pattern at that position.
 - If there's no match, go to the next position.

These common words do not make it obvious why the regexp fails, so let's elaborate how the search works for the pattern `".+"`.

1. The first pattern character is a quote `"`. The regular expression engine tries to find it at the zero position of the source string `a "witch" and her "broom" is one`, but there's a `a` there, so there's immediately no match.

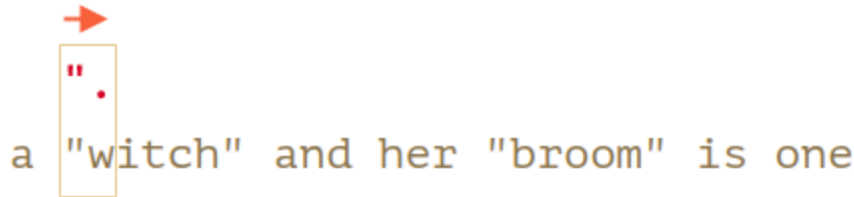
Then it advances: goes to the next positions in the source string and tries to find the first character of the pattern there, fails again, and finally finds the quote at the 3rd position:



a "witch" and her "broom" is one

2. The quote is detected, and then the engine tries to find a match for the rest of the pattern. It tries to see if the rest of the subject string conforms to `".+"`.

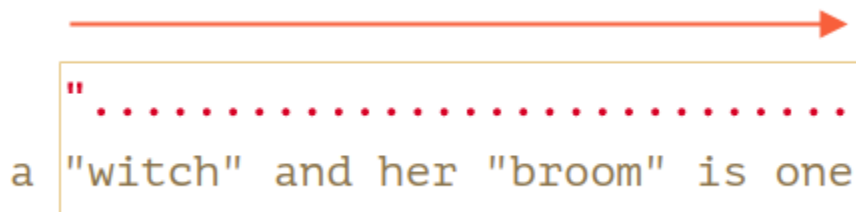
In our case the next pattern character is `.` (a dot). It denotes “any character except a newline”, so the next string letter `w` fits:



a "witch" and her "broom" is one

3. Then the dot repeats because of the quantifier `.+`. The regular expression engine adds to the match one character after another.

...Until when? All characters match the dot, so it only stops when it reaches the end of the string



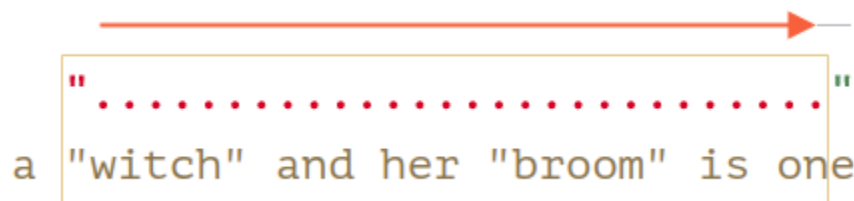
a "witch" and her "broom" is one

:

4. Now the engine has finished repeating `.+` and tries to find the next character of the pattern. It's the quote `"`. But there's a problem: the string has finished, there are no more characters!

The regular expression engine understands that it took too many `.+` and starts to *backtrack*.

In other words, it shortens the match for the quantifier by one character:

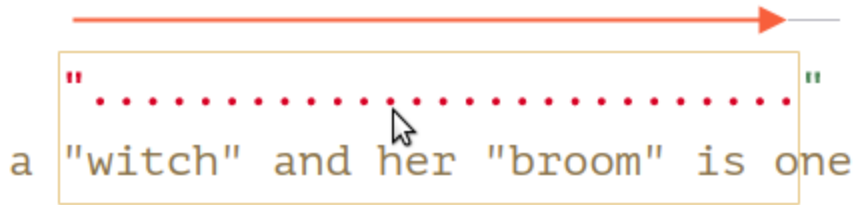


a "witch" and her "broom" is one

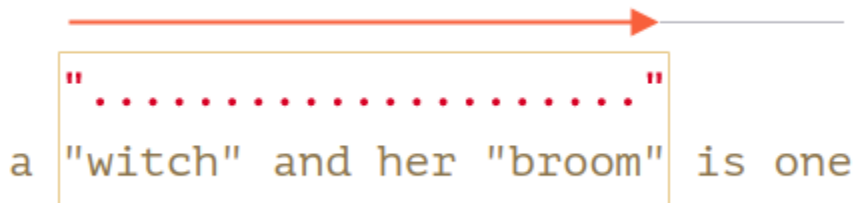
Now it assumes that `.+` ends one character before the string ends and tries to match the rest of the pattern from that position.

If there were a quote there, then the search would end, but the last character is 'e', so there's no match.

5. ...So the engine decreases the number of repetitions of `.+` by one more character:



6. The engine keep backtracking: it decreases the count of repetition for '.' until the rest of the pattern (in our case '') matches:



7. The match is complete.
8. So the first match is "witch" and her "broom". If the regular expression has flag g, then the search will continue from where the first match ends. There are no more quotes in the rest of the string is one, so no more results.

That's probably not what we expected, but that's how it works.

In the greedy mode (by default) a quantifier is repeated as many times as possible.

The regexp engine adds to the match as many characters as it can for .+, and then shortens that one by one, if the rest of the pattern doesn't match.

For our task we want another thing. That's where a lazy mode can help.

Lazy mode

The lazy mode of quantifiers is an opposite to the greedy mode. It means: "repeat minimal number of times".

We can enable it by putting a question mark '?' after the quantifier, so that it becomes *? or +? or even ?? for '?'.

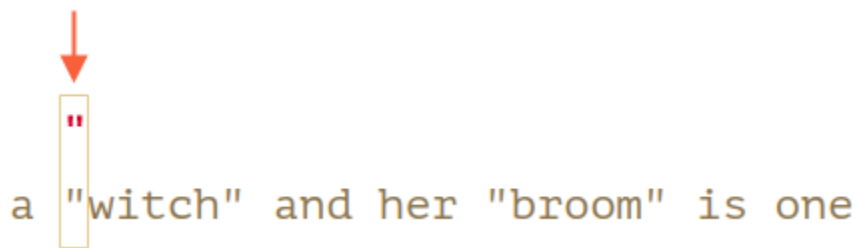
To make things clear: usually a question mark `?` is a quantifier by itself (zero or one), but if added *after another quantifier (or even itself)* it gets another meaning – it switches the matching mode from greedy to lazy.

The regexp `/"(?:.+?)/g` works as intended: it finds `"witch"` and `"broom"`:

```
let regexp = /"(?:.+?)/g;  
let str = 'a "witch" and her "broom" is one';  
alert( str.match(regexp) ); // witch, broom
```

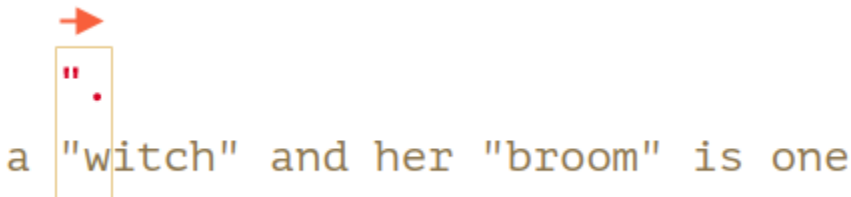
To clearly understand the change, let's trace the search step by step.

1. The first step is the same: it finds the pattern start `"` at the 3rd position:



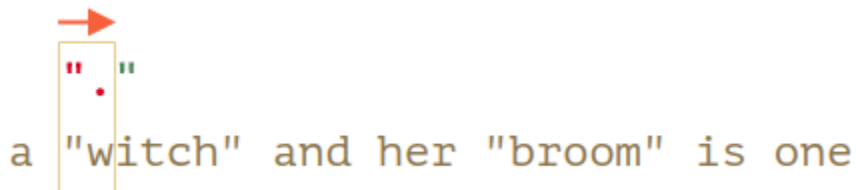
a "witch" and her "broom" is one

2. The next step is also similar: the engine finds a match for the dot `.`:



a "witch" and her "broom" is one

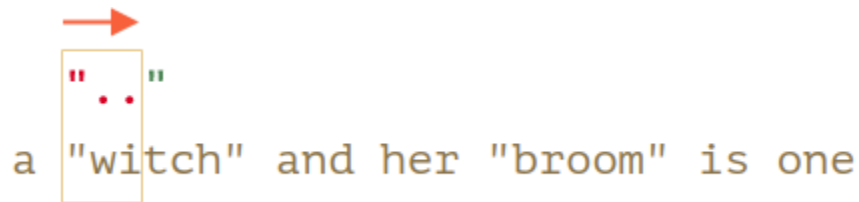
3. And now the search goes differently. Because we have a lazy mode for `+?`, the engine doesn't try to match a dot one more time, but stops and tries to match the rest of the pattern `"` right now:



a "witch" and her "broom" is one

If there were a quote there, then the search would end, but there's `i`, so there's no match.

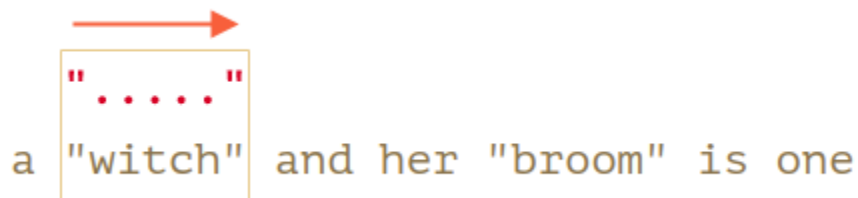
4. Then the regular expression engine increases the number of repetitions for the dot and tries one more time:



a "witch" and her "broom" is one

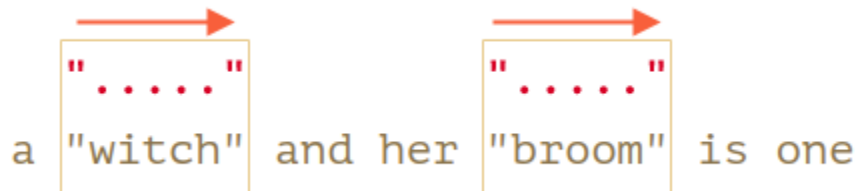
Failure again. Then the number of repetitions is increased again and again...

5. ...Till the match for the rest of the pattern is found:



a "witch" and her "broom" is one

6. The next search starts from the end of the current match and yield one more result:



a "witch" and her "broom" is one

In this example we saw how the lazy mode works for `+`. Quantifiers `*?` and `??` work the similar way – the regexp engine increases the number of repetitions only if the rest of the pattern can't match on the given position.

Laziness is only enabled for the quantifier with `?`.

Other quantifiers remain greedy.

For instance:

```
alert( "123 456".match(/d+ \d+?/) ); // 123 4
```

1. The pattern `\d+` tries to match as many digits as it can (greedy mode), so it finds `123` and stops, because the next character is a space `' '`.
2. Then there's a space in the pattern, it matches.
3. Then there's `\d+?`. The quantifier is in lazy mode, so it finds one digit `4` and tries to check if the rest of the pattern matches from there.
...But there's nothing in the pattern after `\d+?`.
The lazy mode doesn't repeat anything without a need. The pattern finished, so we're done. We have a match `123 4`.

Alternative approach

With regexps, there's often more than one way to do the same thing.

In our case we can find quoted strings without lazy mode using the regexp `"[^\"]+"`:

```
let regexp = /"[^"]+"/g;
let str = 'a "witch" and her "broom" is one';
alert( str.match(regexp) ); // witch, broom
```

The regexp `"[^\"]+"` gives correct results, because it looks for a quote `"` followed by one or more non-quotes `[^\"]`, and then the closing quote.

When the regexp engine looks for `[^\"]+` it stops the repetitions when it meets the closing quote, and we're done.

Please note, that this logic does not replace lazy quantifiers!

It is just different. There are times when we need one or another.

Let's see an example where lazy quantifiers fail and this variant works right.

For instance, we want to find links of the form ``, with any href.

Which regular expression to use?

The first idea might be: `//g`.

Let's check it:

```
let str = '...<a href="link" class="doc">...';
```

```
let regexp = /<a href=".*" class="doc">/g;
// Works!
alert( str.match(regexp) ); // <a href="link" class="doc">
```

It worked. But let's see what happens if there are many links in the text?

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href=".*" class="doc">/g;
// Whoops! Two links in one match!
alert( str.match(regexp) ); // <a href="link1" class="doc">... <a href="link2" class="doc">
```

Now the result is wrong for the same reason as our “witches” example. The quantifier `.*` took too many characters.

The match looks like this:

```
<a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Let's modify the pattern by making the quantifier `.*` lazy:

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href=".*?" class="doc">/g;

// Works!
alert( str.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

Now it seems to work, there are two matches:

```
<a href="....." class="doc"> <a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

...But let's test it on one more text input:

```
let str = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let regexp = /<a href=".*?" class="doc">/g;
// Wrong match!
```

```
alert( str.match(regex) ); // <a href="link1" class="wrong">... <p style="" class="doc">
```

Now it fails. The match includes not just a link, but also a lot of text after it, including `<p...>`.

Why?

That's what's going on:

1. First the regexp finds a link start `<a href=`.
2. Then it looks for `.*?`: takes one character (lazily!), checks if there's a match for `" class="doc">` (none).
3. Then takes another character into `.*?`, and so on... until it finally reaches `" class="doc">`.

But the problem is: that's already beyond the link `<a...>`, in another tag `<p>`. Not what we want. Here's the picture of the match aligned with the text:

```
<a href="....." class="doc">
<a href="link1" class="wrong">... <p style="" class="doc">
```

So, we need the pattern to look for ``, but both greedy and lazy variants have problems.

The correct variant can be: `href="^[^"]*" class="doc">`. It will take all characters inside the href attribute till the nearest quote, just what we need.

A working example:

```
let str1 = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let str2 = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href="^[^"]*" class="doc">/g;

// Works!
alert( str1.match(regexp) ); // null, no matches, that's correct
alert( str2.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

Capturing groups

A part of a pattern can be enclosed in parentheses (...). This is called a “capturing group”.

That has two effects:

1. It allows you to get a part of the match as a separate item in the result array.
2. If we put a quantifier after the parentheses, it applies to the parentheses as a whole.

Examples

Let's see how parentheses work in examples.

Example: gogogo

Without parentheses, the pattern `go+` means `g` character, followed by `o` repeated one or more times. For instance, `goooo` or `goooooooooo`.

Parentheses group characters together, so `(go)+` means `go`, `gogo`, `gogogo` and so on.

```
alert( 'Gogogo now!'.match(/(go)+/i) ); // "Gogogo"
```

Example: domain

Let's make something more complex – a regular expression to search for a website domain. For example:

`mail.com`

`users.mail.com`

`smith.users.mail.com`

As we can see, a domain consists of repeated words, a dot after each one except the last one.

In regular expressions that's `(\w+\.)+\w+`:

```
let regexp = /(\w+\.)+\w+/g;
```

```
alert( "site.com my.site.com".match(regexp) ); // site.com,my.site.com
```

The search works, but the pattern can't match a domain with a hyphen, e.g. `my-site.com`, because the hyphen does not belong to class `\w`.

We can fix it by replacing `\w` with `[\w-]` in every word except the last one: `([\w-]+\.)+\w+`.

Example: email

The previous example can be extended. We can create a regular expression for emails based on it.

The email format is: `name@domain`. Any word can be the name, hyphens and dots are allowed. In regular expressions that's `[-.\w]+`.

The pattern:

```
let regexp = /[-.\w]+@[([\w-]+\.)+[\w-]+/g;  
alert("my@mail.com @ his@site.com.uk".match(regexp)); // my@mail.com, his@site.com.uk
```

That regexp is not perfect, but mostly works and helps to fix accidental mistypes. The only truly reliable check for an email can only be done by sending a letter.

Parentheses contents in the match

Parentheses are numbered from left to right. The search engine memorizes the content matched by each of them and allows to get it in the result.

The method `str.match(regexp)`, if `regexp` has no flag `g`, looks for the first match and returns it as an array:

1. At index 0: the full match.
2. At index 1: the contents of the first parentheses.
3. At index 2: the contents of the second parentheses.
4. ...and so on...

For instance, we'd like to find HTML tags `<.*?>`, and process them. It would be convenient to have tag content (what's inside the angles), in a separate variable.

Let's wrap the inner content into parentheses, like this: `<(.*?)>`.

Now we'll get both the tag as a whole `<h1>` and its contents `h1` in the resulting array:

```
let str = '<h1>Hello, world!</h1>';  
let tag = str.match(/<(.*?)>/);  
alert( tag[0] ); // <h1>  
alert( tag[1] ); // h1
```

Nested groups

Parentheses can be nested. In this case the numbering also goes from left to right.

For instance, when searching a tag in `` we may be interested in:

1. The tag content as a whole: `span class="my"`.
2. The tag name: `span`.
3. The tag attributes: `class="my"`.

Let's add parentheses for them: `<(([a-z]+\s*([^\s]*))>`.

Here's how they are numbered (left to right, by the opening paren):

Diagram illustrating the numbering of parentheses in the regex `<(([a-z]+\s*([^\s]*))>`:

- 1. The entire tag content: `span class="my"`
- 2. The tag name: `span`
- 3. The tag attributes: `class="my"`

In action:

```
let str = '<span class="my">';
let regexp = /<(([a-z]+\s*([^\s]*))>/;
let result = str.match(regexp);
alert(result[0]); // <span class="my">
alert(result[1]); // span class="my"
alert(result[2]); // span
alert(result[3]); // class="my"
```

Optional groups

Even if a group is optional and doesn't exist in the match (e.g. has the quantifier `(...)?`), the corresponding result array item is present and equals `undefined`.

For instance, let's consider the regexp `a(z)?(c)?`. It looks for "a" optionally followed by "z" optionally followed by "c".

If we run it on the string with a single letter `a`, then the result is:

```
let match = 'a'.match(/a(z)?(c)?/);
alert( match.length ); // 3
alert( match[0] ); // a (whole match)
alert( match[1] ); // undefined
alert( match[2] ); // undefined
```

The array has the length of 3, but all groups are empty.

And here's a more complex match for the string `ac`:

```
let match = 'ac'.match(/a(z)?(c)?/)  
alert( match.length ); // 3  
alert( match[0] ); // ac (whole match)  
alert( match[1] ); // undefined, because there's nothing for (z)?  
alert( match[2] ); // c
```

The array length is permanent: 3. But there's nothing for the group `(z)?`, so the result is `["ac", undefined, "c"]`

Searching for all matches with groups: `matchAll`

When we search for all matches (flag `g`), the `match` method does not return contents for groups.

For example, let's find all tags in a string:

```
let str = '<h1> <h2>';  
let tags = str.match(/<(.*?)>/g);  
alert( tags ); // <h1>,<h2>
```

The result is an array of matches, but without details about each of them. But in practice we usually need the contents of capturing groups in the result.

To get them, we should search using the method `str.matchAll(regex)`.

It was added to the JavaScript language long after the `match`, as its “new and improved version”.

Just like `match`, it looks for matches, but there are 3 differences:

1. It returns not an array, but an iterable object.
2. When the flag `g` is present, it returns every match as an array with groups.
3. If there are no matches, it returns not `null`, but an empty iterable object.

For instance:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);  
// results - is not an array, but an iterable object  
alert(results); // [object RegExp String Iterator]
```

```
alert(results[0]); // undefined (*)
results = Array.from(results); // let's turn it into array
alert(results[0]); // <h1>,h1 (1st tag)
alert(results[1]); // <h2>,h2 (2nd tag)
```

There's no need in `Array.from` if we're looping over results:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
for(let result of results) {
  alert(result);
  // first alert: <h1>,h1
  // second: <h2>,h2
}
```

...Or using destructuring:

```
let [tag1, tag2] = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
```

Named groups

Remembering groups by their numbers is hard. For simple patterns it's doable, but for more complex ones counting parentheses is inconvenient. We have a much better option: give names to parentheses.

That's done by putting `?<name>` immediately after the opening paren.

For example, let's look for a date in the format "year-month-day":

```
let dateRegExp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "2019-04-30";
let groups = str.match(dateRegExp).groups;
alert(groups.year); // 2019
alert(groups.month); // 04
alert(groups.day); // 30
```

As you can see, the groups reside in the `.groups` property of the match.

To look for all dates, we can add flag `g`.

We'll also need `matchAll` to obtain full matches, together with groups:

```
let dateRegex = /(?!<year>[0-9]{4})-(?!<month>[0-9]{2})-(?!<day>[0-9]{2})/g;
let str = "2019-10-30 2020-01-01";
let results = str.matchAll(dateRegex);
for(let result of results) {
  let {year, month, day} = result.groups;
  alert(`${day}.${month}.${year}`);
  // first alert: 30.10.2019
  // second: 01.01.2020
}
```

Capturing groups in replacement

Method `str.replace(regex, replacement)` that replaces all matches with `regex` in `str` allows to use parentheses contents in the `replacement` string. That's done using `$n`, where `n` is the group number.

For example,

```
let str = "John Bull";
let regex = /(\\w+) (\\w+)/;
alert( str.replace(regex, '$2, $1') ); // Bull, John
```

For named parentheses the reference will be `$<name>`.

For example, let's reformat dates from "year-month-day" to "day.month.year":

```
let regex = /(?!<year>[0-9]{4})-(?!<month>[0-9]{2})-(?!<day>[0-9]{2})/g;
let str = "2019-10-30, 2020-01-01";
alert( str.replace(regex, '$<day>.$<month>.$<year>') );
// 30.10.2019, 01.01.2020
```

Non-capturing groups with ?:

Sometimes we need parentheses to correctly apply a quantifier, but we don't want their contents in results.

A group may be excluded by adding `?:` in the beginning.

For instance, if we want to find `(go)+`, but don't want the parentheses contents `(go)` as a separate array item, we can write: `(?:go)+`.

In the example below we only get the name `John` as a separate member of the match:

```
let str = "Gogogo John!";  
// ?: excludes 'go' from capturing  
let regexp = /(?:go)+ (\w+)/i;  
let result = str.match(regexp);  
alert( result[0] ); // Gogogo John (full match)  
alert( result[1] ); // John  
alert( result.length ); // 2 (no more items in the array)
```

Backreferences in pattern: `\N` and `\k<name>`

We can use the contents of capturing groups (...) not only in the result or in the replacement string, but also in the pattern itself.

Backreference by number: `\N`

A group can be referenced in the pattern using `\N`, where `N` is the group number.

To make clear why that's helpful, let's consider a task.

We need to find quoted strings: either single-quoted `'...'` or a double-quoted `"..."` – both variants should match.

How to find them?

We can put both kinds of quotes in the square brackets: `["'](.*)["']`, but it would find strings with mixed quotes, like `"...' and '..."`. That would lead to incorrect matches when one quote appears inside other ones, like in the string `"She's the one!"`:

```
let str = `He said: "She's the one!".`;   
let regexp = /["'](.*)["']/g;  
// The result is not what we'd like to have  
alert( str.match(regexp) ); // "She'
```

As we can see, the pattern found an opening quote `"`, then the text is consumed till the other quote `'`, that closes the match.

To make sure that the pattern looks for the closing quote exactly the same as the opening one, we can wrap it into a capturing group and backreference it: `(["'])(.*?)\1`.

Here's the correct code:

```
let str = `He said: "She's the one!".`;
let regexp = /(["'])(.*?)\1/g;
alert( str.match(regexp) ); // "She's the one!"
```

Now it works! The regular expression engine finds the first quote `(["'])` and memorizes its content. That's the first capturing group.

Further in the pattern `\1` means “find the same text as in the first group”, exactly the same quote in our case.

Similar to that, `\2` would mean the contents of the second group, `\3` – the 3rd group, and so on.

Note : If we use `?` in the group, then we can't reference it. Groups that are excluded from capturing `(?:...)` are not memorized by the engine.

Don't mess up: in the pattern `\1`, in the replacement: `$1`

In the replacement string we use a dollar sign: `$1`, while in the pattern – a backslash `\1`.

Backreference by name: `\k<name>`

If a regexp has many parentheses, it's convenient to give them names.

To reference a named group we can use `\k<имя>`.

In the example below the group with quotes is named `?<quote>`, so the backreference is `\k<quote>`:

```
let str = `He said: "She's the one!".`;
let regexp = /(?<quote>["'])(.*?)\k<quote>/g;
alert( str.match(regexp) ); // "She's the one!"
```

Alternation (OR) |

Alternation is the term in regular expression that is actually a simple “OR”.

In a regular expression it is denoted with a vertical line character |.

For instance, we need to find programming languages: HTML, PHP, Java or JavaScript.

The corresponding regexp: `html|php|java(script)?`.

A usage example:

```
let regexp = /html|php|css|java(script)?/gi;
let str = "First HTML appeared, then CSS, then JavaScript";
alert( str.match(regexp) ); // 'HTML', 'CSS', 'JavaScript'
```

We already saw a similar thing – square brackets. They allow you to choose between multiple characters, for instance `gr[ae]y` matches `gray` or `grey`.

Square brackets allow only characters or character sets. Alternation allows any expressions. A regexp `A|B|C` means one of expressions A, B or C.

For instance:

- `gr(a|e)y` means exactly the same as `gr[ae]y`.
- `gr|ey` means `gra` or `ey`.

Example: regexp for time

In previous articles there was a task to build a regexp for searching time in the form `hh:mm`, for instance `12:00`. But a simple `\d\d:\d\d` is too vague. It accepts `25:99` as the time (as 99 seconds match the pattern, but that time is invalid).

How can we make a better pattern?

We can use more careful matching. First, the hours:

- If the first digit is 0 or 1, then the next digit can be any: `[01]\d`.
- Otherwise, if the first digit is 2, then the next must be `[0-3]`.
- (no other first digit is allowed)

We can write both variants in a regexp using alternation: `[01]\d|2[0-3]`.

Next, minutes must be from 00 to 59. In the regular expression language that can be written as `[0-5]\d`: the first digit 0-5, and then any digit.

If we glue minutes and seconds together, we get the pattern: `[01]\d|2[0-3]:[0-5]\d`.

We're almost done, but there's a problem. The alternation `|` now happens to be between `[01]\d` and `2[0-3]:[0-5]\d`.

That is: minutes are added to the second alternation variant, here's a clear picture:

`[01]\d | 2[0-3]:[0-5]\d`

That pattern looks for `[01]\d` or `2[0-3]:[0-5]\d`.

But that's wrong, the alternation should only be used in the "hours" part of the regular expression, to allow `[01]\d` OR `2[0-3]`. Let's correct that by enclosing "hours" into parentheses: `([01]\d|2[0-3]):[0-5]\d`.

The final solution:

```
let regexp = /([01]\d|2[0-3]):[0-5]\d/g;
alert("00:00 10:10 23:59 25:99 1:2".match(regexp)); // 00:00,10:10,23:59
```

Lookahead and lookbehind

Sometimes we need to find only those matches for a pattern that are followed or preceded by another pattern.

There's a special syntax for that, called "lookahead" and "lookbehind", together referred to as "lookaround".

For the start, let's find the price from the string like `1 turkey costs 30€`. That is: a number, followed by `€` sign.

Lookahead

The syntax is: `X(?=Y)`, it means "look for X, but match only if followed by Y". There may be any pattern instead of X and Y.

For an integer number followed by `€`, the regexp will be `\d+(?=.*€)`:


```
let str = "1 turkey costs 30€";  
alert( str.match(/\d+(?=€)/) ); // 30, the number 1 is ignored, as it's not followed by €
```

Please note: the lookahead is merely a test, the contents of the parentheses (`?=...`) is not included in the result 30.

When we look for `X(?=Y)`, the regular expression engine finds `X` and then checks if there's `Y` immediately after it. If it's not so, then the potential match is skipped, and the search continues.

More complex tests are possible, e.g. `X(?=Y)(?=Z)` means:

1. Find `X`.
2. Check if `Y` is immediately after `X` (skip if isn't).
3. Check if `Z` is also immediately after `X` (skip if isn't).
4. If both tests passed, then the `X` is a match, otherwise continue searching.

In other words, such a pattern means that we're looking for `X` followed by `Y` and `Z` at the same time.

That's only possible if patterns `Y` and `Z` aren't mutually exclusive.

For example, `\d+(?=\s)(?=.*30)` looks for `\d+` only if it's followed by a space, and there's 30 somewhere after it:

```
let str = "1 turkey costs 30€";  
alert( str.match(/\d+(?=\s)(?=.*30)/) ); // 1
```

In our string that exactly matches the number 1.

Negative lookahead

Let's say that we want a quantity instead, not a price from the same string. That's a number `\d+`, NOT followed by `€`.

For that, a negative lookahead can be applied.

The syntax is: `X(?!Y)`, it means "search `X`, but only if not followed by `Y`".

```
let str = "2 turkeys cost 60€";  
alert( str.match(/\d+(?!€)/) ); // 2 (the price is skipped)
```

Lookbehind

Lookahead allows you to add a condition for “what follows”.

Lookbehind is similar, but it looks behind. That is, it allows a pattern to match only if there’s something before it.

The syntax is:

- Positive lookbehind: `(?<=Y)X`, matches `X`, but only if there’s `Y` before it.
- Negative lookbehind: `(?<!Y)X`, matches `X`, but only if there’s no `Y` before it.

For example, let’s change the price to US dollars. The dollar sign is usually before the number, so to look for `$30` we’ll use `(?<=\\$)\\d+` – an amount preceded by `$`:

```
let str = "1 turkey costs $30";  
// the dollar sign is escaped \\$  
alert( str.match(/(?<=\\$)\\d+/) ); // 30 (skipped the sole number)
```

And, if we need the quantity – a number, not preceded by `$`, then we can use a negative lookbehind `(?<!\\$)\\d+`:

```
let str = "2 turkeys cost $60";  
alert( str.match(/(?<!\\$)\\d+/) ); // 2 (skipped the price)
```

Capturing groups

Generally, the contents inside look around parentheses does not become a part of the result.

E.g. in the pattern `\\d+(?=€)`, the `€` sign doesn’t get captured as a part of the match. That’s natural: we look for a number `\\d+`, while `(?=€)` is just a test that should be followed by `€`.

But in some situations we might want to capture the lookahead expression as well, or a part of it. That’s possible. Just wrap that part into additional parentheses.

In the example below the currency sign `(€|kr)` is captured, along with the amount:

```
let str = "1 turkey costs 30€";  
let regexp = /\\d+(?= (€|kr))/; // extra parentheses around €|kr  
alert( str.match(regexp) ); // 30, €
```

And here’s the same for lookbehind:

```
let str = "1 turkey costs $30";  
let regexp = /(?!<=(\${£}))\d+;/  
alert( str.match(regexp) ); // 30, $
```

Sticky flag "y", searching at position

The flag `y` allows to perform the search at the given position in the source string.

To grasp the use case of `y` flag, and see how great it is, let's explore a practical use case.

One of common tasks for regexps is "lexical analysis": we get a text, e.g. in a programming language, and analyze it for structural elements.

For instance, HTML has tags and attributes, JavaScript code has functions, variables, and so on.

Writing lexical analyzers is a special area, with its own tools and algorithms, so we don't go deep in there, but there's a common task: to read something at the given position.

E.g. we have a code string `let varName = "value"`, and we need to read the variable name from it, that starts at position 4.

We'll look for variable name using regexp `\w+`. Actually, JavaScript variable names need a bit more complex regexp for accurate matching, but here it doesn't matter.

A call to `str.match(/\w+/)` will find only the first word in the line. Or all words with the flag `g`. But we need only one word at position 4.

To search from the given position, we can use the method `regexp.exec(str)`.

If the `regexp` doesn't have flags `g` or `y`, then this method looks for the first match in the string `str`, exactly like `str.match(regexp)`. Such a simple no-flags case doesn't interest us here.

If there's flag `g`, then it performs the search in the string `str`, starting from position stored in its `regexp.lastIndex` property. And, if it finds a match, then sets `regexp.lastIndex` to the index immediately after the match.

When a regexp is created, its `lastIndex` is 0.

So, successive calls to `regexp.exec(str)` return matches one after another. An example (with flag `g`):

```

let str = 'let varName';
let regexp = /\w+/g;
alert(regexp.lastIndex); // 0 (initially lastIndex=0)
let word1 = regexp.exec(str);
alert(word1[0]); // let (1st word)
alert(regexp.lastIndex); // 3 (position after the match)
let word2 = regexp.exec(str);
alert(word2[0]); // varName (2nd word)
alert(regexp.lastIndex); // 11 (position after the match)
let word3 = regexp.exec(str);
alert(word3); // null (no more matches)
alert(regexp.lastIndex); // 0 (resets at search end)

```

Every match is returned as an array with groups and additional properties.

We can get all matches in the loop:

```

let str = 'let varName';
let regexp = /\w+/g;
let result;
while (result = regexp.exec(str)) {
    alert( `Found ${result[0]} at position ${result.index}` );
    // Found let at position 0, then
    // Found varName at position 4
}

```

Such use of `regexp.exec` is an alternative to method `str.matchAll`.

Unlike other methods, we can set our own `lastIndex`, to start the search from the given position.

For instance, let's find a word, starting from position 4:

```

let str = 'let varName = "value"';
let regexp = /\w+/g; // without flag "g", property lastIndex is ignored
regexp.lastIndex = 4;
let word = regexp.exec(str);
alert(word); // varName

```

We performed a search of `\w+`, starting from position `regexp.lastIndex = 4`.

Please note: the search starts at position `lastIndex` and then goes further. If there's no word at position `lastIndex`, but it's somewhere after it, then it will be found:

```
let str = 'let varName = "value";  
let regexp = /\w+/g;  
regexp.lastIndex = 3;  
let word = regexp.exec(str);  
alert(word[0]); // varName  
alert(word.index); // 4
```

...So, with flag `g` property `lastIndex` sets the starting position for the search.

Flag `y` makes `regexp.exec` to look exactly at position `lastIndex`, not before, not after it.

Here's the same search with flag `y`:

```
let str = 'let varName = "value";  
let regexp = /\w+/y;  
regexp.lastIndex = 3;  
alert( regexp.exec(str) ); // null (there's a space at position 3, not a word)  
regexp.lastIndex = 4;  
alert( regexp.exec(str) ); // varName (word at position 4)
```

As we can see, `regexp /\w+/y` doesn't match at position 3 (unlike the flag `g`), but matches at position 4.

Imagine, we have a long text, and there are no matches in it, at all. Then searching with flag `g` will go till the end of the text, and this will take significantly more time than the search with flag `y`.

In such tasks like lexical analysis, there are usually many searches at an exact position. Using flag `y` is the key for a good performance.

Methods of RegExp and String

In this article we'll cover various methods that work with regexps in-depth.

`str.match(regexp)`

The method `str.match(regex)` finds matches for `regex` in the string `str`.

It has 3 modes:

1. If the `regex` doesn't have flag `g`, then it returns the first match as an array with capturing groups and properties `index`(position of the match), `input` (input string, equals `str`):

```
let str = "I love JavaScript";
let result = str.match(/Java(Script)/);
alert( result[0] ); // JavaScript (full match)
alert( result[1] ); // Script (first capturing group)
alert( result.length ); // 2

// Additional information:
alert( result.index ); // 0 (match position)
alert( result.input ); // I love JavaScript (source string)
```

2. If the `regex` has flag `g`, then it returns an array of all matches as strings, without capturing groups and other details.

```
let str = "I love JavaScript";
let result = str.match(/Java(Script)/g);
alert( result[0] ); // JavaScript
alert( result.length ); // 1
```

3. If there are no matches, no matter if there's flag `g` or not, `null` is returned.

That's an important nuance. If there are no matches, we don't get an empty array, but `null`. It's easy to make a mistake forgetting about it, e.g.:

```
let str = "I love JavaScript";
let result = str.match(/HTML/);
alert(result); // null
alert(result.length); // Error: Cannot read property 'length' of null
```

If we want the result to be an array, we can write like this:

```
let result = str.match(regex) || [];
```

str.matchAll(regex)

The method `str.matchAll(regex)` is a “newer, improved” variant of `str.match`.

It's used mainly to search for all matches with all groups.

There are 3 differences from `match`:

1. It returns an iterable object with matches instead of an array. We can make a regular array from it using `Array.from`.
2. Every match is returned as an array with capturing groups (the same format as `str.match` without flag `g`).
3. If there are no results, it returns not `null`, but an empty iterable object.

Usage example:

```
let str = '<h1>Hello, world!</h1>';
let regexp = /<(.*?)>/g;
let matchAll = str.matchAll(regexp);
alert(matchAll); // [object RegExp String Iterator], not array, but an iterable
matchAll = Array.from(matchAll); // array now
let firstMatch = matchAll[0];
alert( firstMatch[0] ); // <h1>
alert( firstMatch[1] ); // h1
alert( firstMatch.index ); // 0
alert( firstMatch.input ); // <h1>Hello, world!</h1>
```

If we use `for..of` to loop over `matchAll` matches, then we don't need `Array.from` any more.

str.split(regex|substr, limit)

Splits the string using the `regex` (or a substring) as a delimiter.

We can use `split` with strings, like this:

```
alert('12-34-56'.split('-')) // array of [12, 34, 56]
```

But we can split by a regular expression, the same way:

```
alert('12, 34, 56'.split(/,\s*/)) // array of [12, 34, 56]
```

str.search(regex)

The method `str.search(regex)` returns the position of the first match or `-1` if none found:

```
let str = "A drop of ink may make a million think";  
alert( str.search( /ink/i ) ); // 10 (first match position)
```

The important limitation: search only finds the first match.

If we need positions of further matches, we should use other means, such as finding them all with `str.matchAll(regex)`.

str.replace(str|regex, str|func)

This is a generic method for searching and replacing, one of most useful ones. The swiss army knife for searching and replacing.

We can use it without regexps, to search and replace a substring:

```
// replace a dash by a colon  
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

There's a pitfall though.

When the first argument of replace is a string, it only replaces the first match.

You can see that in the example above: only the first `"-"` is replaced by `":"`.

To find all hyphens, we need to use not the string `"-"`, but a regexp `/-/g`, with the obligatory `g` flag:

```
// replace all dashes by a colon  
alert( '12-34-56'.replace( /-/g, ":" ) ) // 12:34:56
```

The second argument is a replacement string. We can use special character in it:

Symbols	Action in the replacement string
\$&	inserts the whole match
\$`	inserts a part of the string before the match
\$'	inserts a part of the string after the match
\$n	if <code>n</code> is a 1-2 digit number, then it inserts the contents of <code>n</code> -th parentheses, more about it in the chapter Capturing groups
\$<name>	inserts the contents of the parentheses with the given <code>name</code> , more about it in the chapter Capturing groups
\$\$	inserts character <code>\$</code>

For instance:

```
let str = "John Smith";
// swap first and last name
alert(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

For situations that require “smart” replacements, the second argument can be a function.

It will be called for each match, and the returned value will be inserted as a replacement.

The function is called with arguments `func(match, p1, p2, ..., pn, offset, input, groups)`:

1. `match` – the match,
2. `p1, p2, ..., pn` – contents of capturing groups (if there are any),
3. `offset` – position of the match,
4. `input` – the source string,
5. `groups` – an object with named groups.

If there are no parentheses in the regexp, then there are only 3 arguments: `func(str, offset, input)`.

For example, let's uppercase all matches:

```
let str = "html and css";
let result = str.replace(/html|css/gi, str => str.toUpperCase());
alert(result); // HTML and CSS
```

Replace each match by its position in the string:

```
alert("Ho-Ho-ho".replace(/ho/gi, (match, offset) => offset)); // 0-3-6
```

In the example below there are two parentheses, so the replacement function is called with 5 arguments: the first is the full match, then 2 parentheses, and after it (not used in the example) the match position and the source string:

```
let str = "John Smith";
let result = str.replace(/(\w+) (\w+)/, (match, name, surname) => `${surname}, ${name}`);
alert(result); // Smith, John
```

If there are many groups, it's convenient to use rest parameters to access them:

```
let str = "John Smith";
let result = str.replace(/(\w+) (\w+)/, (...match) => `${match[2]}, ${match[1]}`);
alert(result); // Smith, John
```

Or, if we're using named groups, then `groups` object with them is always the last, so we can obtain it like this:

```
let str = "John Smith";
let result = str.replace(/(?<name>\w+) (?<surname>\w+)/, (...match) => {
  let groups = match.pop();
  return `${groups.surname}, ${groups.name}`;
});
alert(result); // Smith, John
```

Using a function gives us the ultimate replacement power, because it gets all the information about the match, has access to outer variables and can do everything.

regex.exec(str)

The method `regex.exec(str)` method returns a match for `regex` in the string `str`. Unlike previous methods, it's called on a `regex`, not on a string.

It behaves differently depending on whether the `regex` has flag `g`.

If there's no `g`, then `regex.exec(str)` returns the first match exactly as `str.match(regex)`. This behavior doesn't bring anything new.

But if there's flag `g`, then:

- A call to `regexp.exec(str)` returns the first match and saves the position immediately after it in the property `regexp.lastIndex`.
- The next such call starts the search from position `regexp.lastIndex`, returns the next match and saves the position after it in `regexp.lastIndex`.
- ...And so on.
- If there are no matches, `regexp.exec` returns null and resets `regexp.lastIndex` to 0.

So, repeated calls return all matches one after another, using property `regexp.lastIndex` to keep track of the current search position.

In the past, before the method `str.matchAll` was added to JavaScript, calls of `regexp.exec` were used in the loop to get all matches with groups:

```
let str = 'More about JavaScript at https://javascript.info';
let regexp = /javascript/ig;
let result;
while (result = regexp.exec(str)) {
  alert( `Found ${result[0]} at position ${result.index}` );
  // Found JavaScript at position 11, then
  // Found javascript at position 33
}
```

This works now as well, although for newer browsers `str.matchAll` is usually more convenient.

We can use `regexp.exec` to search from a given position by manually setting `lastIndex`.

For instance:

```
let str = 'Hello, world!';
let regexp = /\w+/g; // without flag "g", lastIndex property is ignored
regexp.lastIndex = 5; // search from 5th position (from the comma)
alert( regexp.exec(str) ); // world
```

If the `regexp` has flag `y`, then the search will be performed exactly at the position `regexp.lastIndex`, not any further.

Let's replace flag `g` with `y` in the example above. There will be no matches, as there's no word at position 5:

```
let str = 'Hello, world!';
let regexp = /\w+/y;
regexp.lastIndex = 5; // search exactly at position 5
alert( regexp.exec(str) ); // null
```

That's convenient for situations when we need to “read” something from the string by a regexp at the exact position, not somewhere further.

regexp.test(str)

The method `regexp.test(str)` looks for a match and returns `true/false` whether it exists.

For instance:

```
let str = "I love JavaScript";
// these two tests do the same
alert( /love/i.test(str) ); // true
alert( str.search(/love/i) !== -1 ); // true
```

An example with the negative answer:

```
let str = "Bla-bla-bla";
alert( /love/i.test(str) ); // false
alert( str.search(/love/i) !== -1 ); // false
```

If the regexp has flag `g`, then `regexp.test` looks from the `regexp.lastIndex` property and updates this property, just like `regexp.exec`.

So we can use it to search from a given position:

```
let regexp = /love/gi;
let str = "I love JavaScript";
// start the search from position 10:
regexp.lastIndex = 10;
alert( regexp.test(str) ); // false (no match)
```
