Fetch

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`**, this 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,
- **`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 :

```
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);
```

Important:
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}`);
}
```

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.

Why is cors needed a brief history

CORS exists to protect the internet from evil hackers.

Seriously. Let's make a very brief historical digression.

For many years a script from one site could not access the content of another site.

That simple, yet powerful rule was a foundation of the internet security. E.g. an evil script from website hacker.com could not access the user’s mailbox at website gmail.com. People felt safe.

JavaScript also did not have any special methods to perform network requests at that time. It was a toy language to decorate a web page.

But web developers demanded more power. A variety of tricks were invented to work around the limitation and make requests to other websites.

Using forms

One way to communicate with another server was to submit a <form> there. People submitted it into <iframe>, just to stay on the current page, like this:

```
<!-- form target -->
<iframe name="iframe"></iframe>

<!-- a form could be dynamically generated and submitted by JavaScript -->
<form target="iframe" method="POST" action="http://another.com/…">
  ...
</form>
```

So, it was possible to make a GET/POST request to another site, even without networking methods, as forms can send data anywhere. But as it’s forbidden to access the content of an <iframe> from another site, it wasn’t possible to read the response.

Using scripts

Another trick was to use a script tag. A script could have any src, with any domain, like <script src="http://another.com/…">. It’s possible to execute a script from any website.

If a website, e.g. another.com intended to expose data for this kind of access, then a so-called “JSONP (JSON with padding)” protocol was used.

Here’s how it worked.

Let’s say we, at our site, need to get the data from http://another.com, such as the weather:

1. First, in advance, we declare a global function to accept the data, e.g. gotWeather.

```
// 1. Declare the function to process the weather data
function gotWeather({ temperature, humidity }) {
  alert(`temperature: ${temperature}, humidity: ${humidity}`);
}
```

2. Then we make a <script> tag with src="http://another.com/weather.json?callback=gotWeather", using the name of our function as the callback URL-parameter.

```
let script = document.createElement('script');
script.src = `http://another.com/weather.json?callback=gotWeather`;
document.body.append(script);
```

3. The remote server another.com dynamically generates a script that calls gotWeather(...) with the data it wants us to receive.

```
// The expected answer from the server looks like this:
gotWeather({
 temperature: 25,
   humidity: 78
});
```

4. When the remote script loads and executes, gotWeather runs, and, as it's our function, we have the data.

Safe requests

There are two types of cross-origin requests:

- Safe requests.
- All the others.

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

The essential difference is that a safe request can be made with a <form> **or a** <script>**, without any special methods.**

So, even a very old server should be ready to accept a safe request.

Contrary to that, requests with non-standard headers or e.g. method DELETE can't be created this way. For a long time JavaScript was unable to do such requests. So an old server may assume that such requests come from a privileged source, "because a webpage is unable to send them".

When we try to make a unsafe request, the browser sends a special "preflight" request that asks the server – does it agree to accept such cross-origin requests, or not?

And, unless the server explicitly confirms that with headers, an unsafe request is not sent.
Now we'll go into details.

Cors for safe 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 look like:

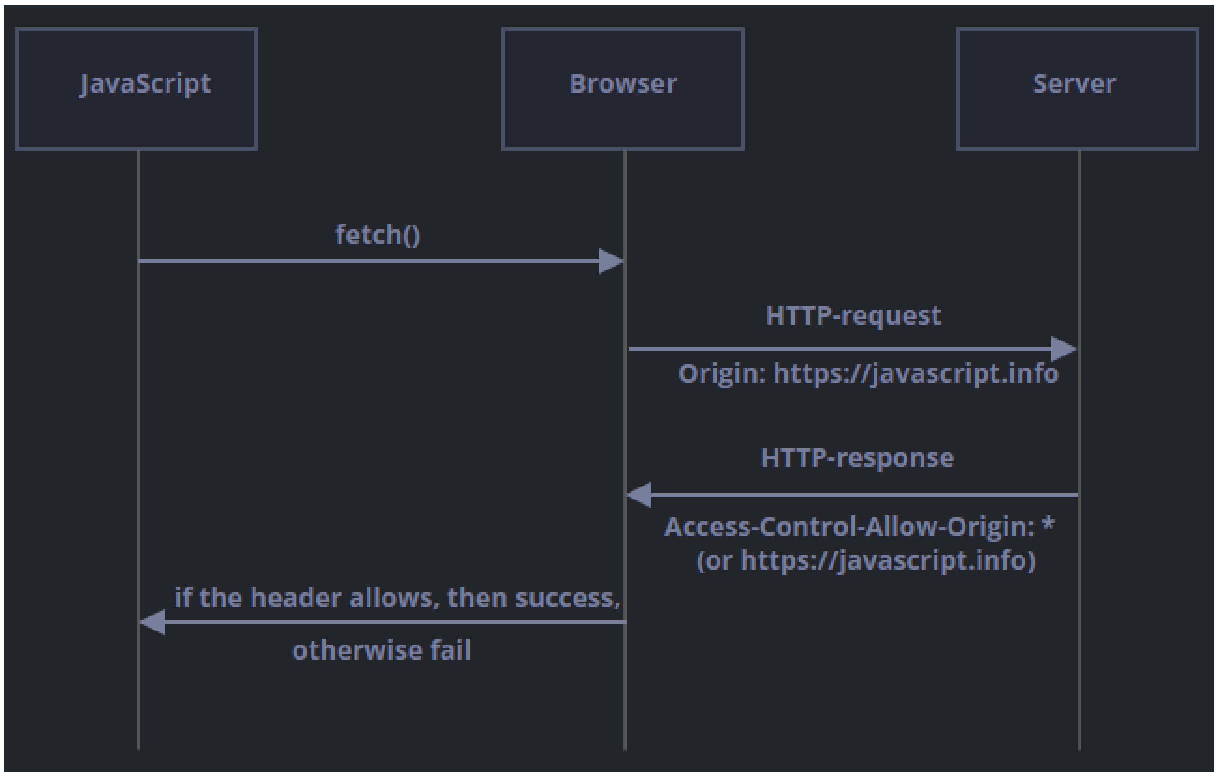
```
GET /request
Host: anywhere.com
Origin: https://javascript.info
...
```

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, add 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 it's an error.

The browser plays the role of a trusted mediator here:

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


JavaScript
Browser
Server
fetch()
HTTP-request
Origin: https://javascript.info
HTTP-response
Access-Control-Allow-Origin: *
(or https://javascript.info)
if the header allows, then success,
otherwise fail

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
```

Unsafe 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 webservices that treat a non-standard method as a signal: "That's not a browser". They can take it into account when checking access rights.

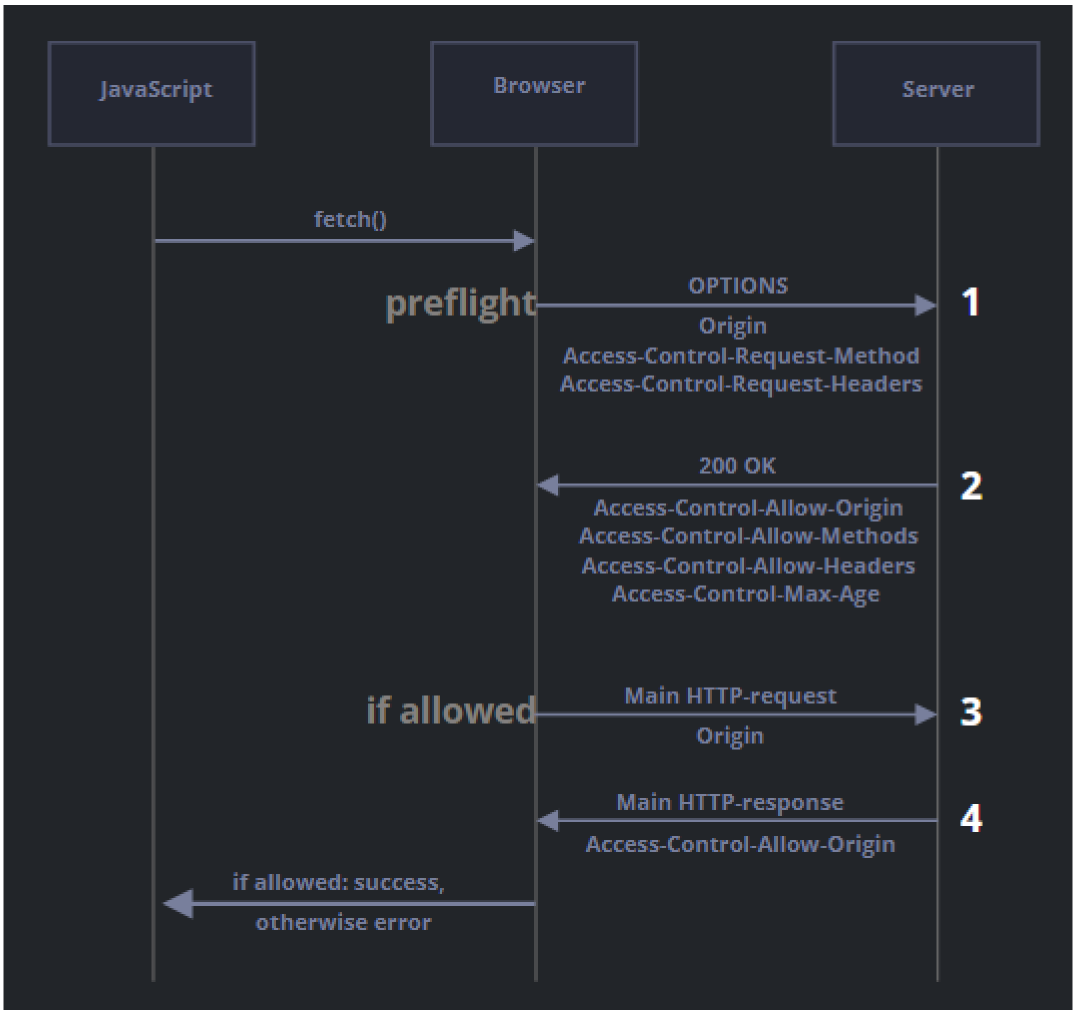
So, to avoid misunderstandings, any "unsafe" request – that couldn't be done in the old times, the browser does not make such requests right away. First, it sends a preliminary, so-called "preflight" request, to ask for permission.

A preflight request uses the method OPTIONS, no body and three headers:

- Access-Control-Request-Method header has the method of the unsafe request.
- Access-Control-Request-Headers header provides a comma-separated list of its unsafe HTTP-headers.
- Origin header tells from where the request came. (such as https://javascript.info)

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

- Access-Control-Allow-Origin must be either * or the requesting origin, such as https://javascript.info, to allow it.
- 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 the example of 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 unsafe (one is enough):

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

Rest parameters and spread syntax

Many JavaScript built-in functions support an arbitrary number of arguments.
For instance:

- `Math.max(arg1, arg2, ..., argN)` – returns the greatest of the arguments.
- `Object.assign(dest, src1, ..., srcN)` – copies properties from `src1..N` into `dest`.

…and so on.

In this chapter we’ll learn how to do the same. And also, how to pass arrays to such functions as parameters.

Rest parameters

A function can be called with any number of arguments, no matter how it is defined.
Like here:

```
function sum(a, b) {
  return a + b;
}
alert( sum(1, 2, 3, 4, 5) );
```

There will be no error because of “excessive” arguments. But of course in the result only the first two will be counted, so the result in the code above is `3`.

The rest of the parameters can be included in the function definition by using three dots `...` followed by the name of the array that will contain them. The dots literally mean “gather the remaining parameters into an array”.

For instance, to gather all arguments into array args:

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

We can choose to get the first parameters as variables, and gather only the rest.

Here the first two arguments go into variables and the rest go into `titles` array:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar

  // the rest go into titles array
  // i.e. titles = ["Consul", "Imperator"]
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

The rest parameters must be at the end

The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

```
function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
  // error
}
```

The `...rest` must always be last.

The arguments variable

There is also a special array-like object named `arguments` that contains all arguments by their index.
For instance:

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // it's iterable
  // for(let arg of arguments) alert(arg);
}

// shows: 2, Julius, Caesar
showName("Julius", "Caesar");

// shows: 1, Ilya, undefined (no second argument)
showName("Ilya");
```

In old times, rest parameters did not exist in the language, and using `arguments` was the only way to get all arguments of the function. And it still works, we can find it in the old code.

But the downside is that although `arguments` is both array-like and iterable, it's not an array. It does not support array methods, so we can't call `arguments.map(...)` for example.

Also, it always contains all arguments. We can't capture them partially, like we did with rest parameters.

So when we need these features, then rest parameters are preferred.

Arrow functions do not have `"arguments"`
If we access the `arguments` object from an arrow function, it takes them from the outer "normal" function.
Here's an example:

```
function f() {
  let showArg = () => alert(arguments[0]);
  showArg();
}

f(1); // 1
```

As we remember, arrow functions don't have their own `this`. Now we know they don't have the special `arguments` object either.

Spread syntax

We've just seen how to get an array from the list of parameters.

But sometimes we need to do exactly the reverse.

For instance, there's a built-in function Math/max that returns the greatest number from a list:

```
alert( Math.max(3, 5, 1) ); // 5
```

Now let's say we have an array `[3, 5, 1]`. How do we call `Math.max` with it?

Passing it "as is" won't work, because `Math.max` expects a list of numeric arguments, not a single array:

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

And surely we can't manually list items in the code `Math.max(arr[0], arr[1], arr[2])`, because we may be unsure how many there are. As our script executes, there could be a lot, or there could be none. And that would get ugly.

Spread syntax to the rescue! It looks similar to rest parameters, also using `...`, but does quite the opposite.

When `...arr` is used in the function call, it "expands" an iterable object `arr` into the list of arguments.

For `Math.max`:

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of
argument
```

How Single-Page Applications Work

An overview of how browser APIs enable single-page applications

11 avril 2018

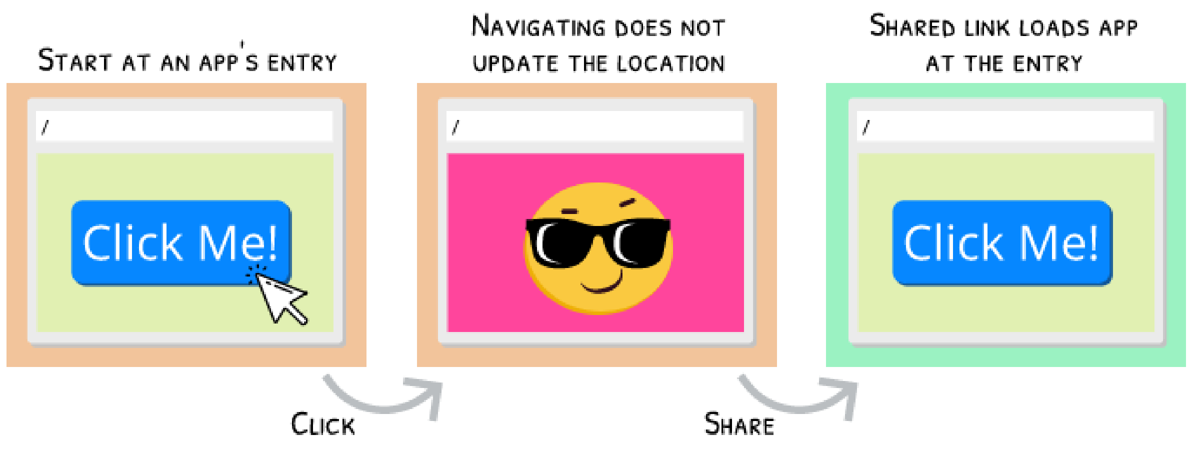
A single-page application (SPA) is a website that re-renders its content in response to navigation actions (e.g. clicking a link) without making a request to the server to fetch new HTML.

While single-page application implementations vary, most rely on the same browser behavior and native APIs to enable the core functionality. Understanding these is key to knowing how single-page applications work.

What Type of SPA?

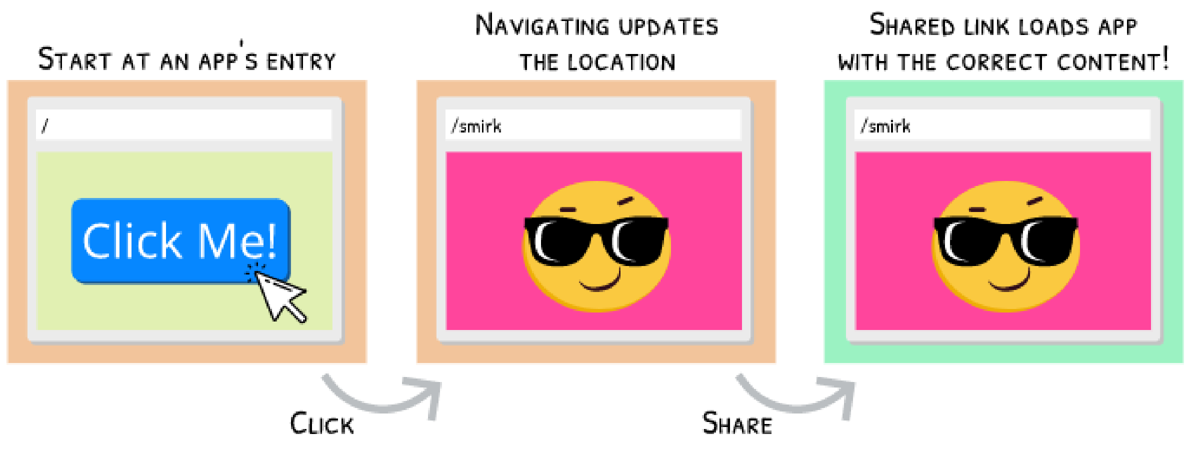
Single-page applications can use state from an external source (i.e. the URL location) or track state internally. This article will be focusing on location-based single-page applications. Why?

Internal state SPAs are limited because there is only one "entry". A single entry means that you always start at the root when you enter the application. When you navigate within an internal-state app, there is no external representation. If you want to share some content, the other person loading the app will start at the root, so you'll have to explain how to get to desired content.


Start at an app's entry
/
Click Me!
Navigating does not update the location
/
Shared link loads app at the entry
/
Click Me!
Click
Share

An internal state SPA can only load the app's entry

With location-based SPAs, you can share a link and be confident that anyone opening that link will see the same thing as you because the location is always updating as you navigate (assuming they have the same authorization to view the content).


Start at an app's entry
/
Click Me!
Navigating updates the location
/smirk
Shared link loads app with the correct content!
/smirk
Click
Share

Route Matching

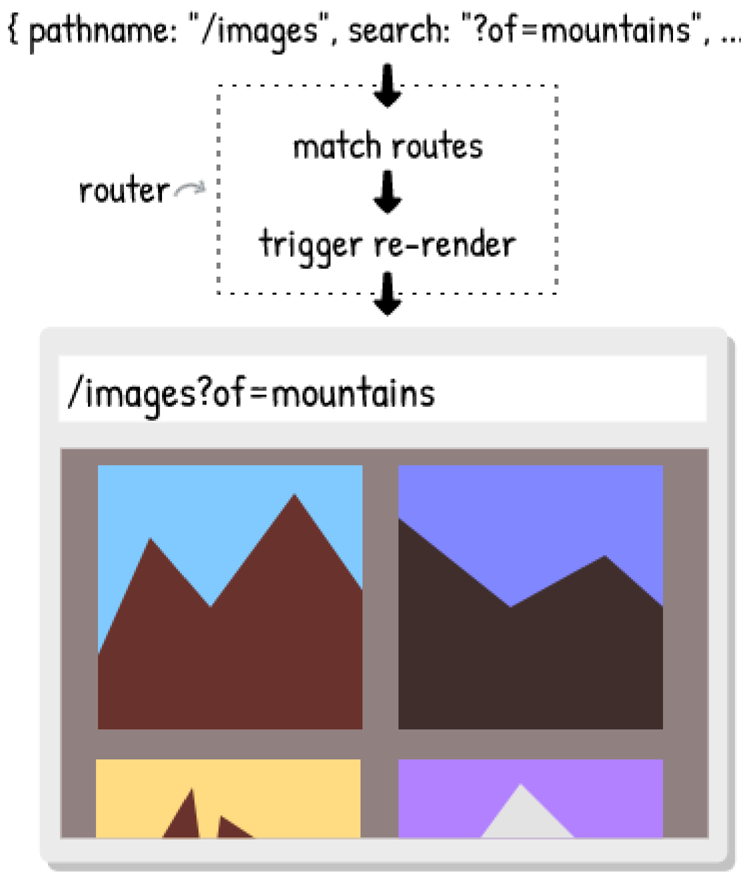
Single-page application generally rely on a router.

Routers are made up of routes, which describe the location that they should match. These can be static (`/about`) or dynamic (`/album/:id`, where the value of `:id` can be any number of possibilities) paths. The path-to-regexp package is a very popular solution for creating these paths.

```
const routes = [

  { path: '/' },

  { path: '/about' },

  { path: '/album/:id' }
];
```

Route matching is comparing the current location (usually only its pathname) against the router's routes to find one that matches.

After matching a route, the router will trigger a re-render of the application. Actual implementations for this are largely up to the router. For example, a router might use the observer pattern, where you give the router a function that knows how to trigger a re-render and the router will call it after it matches a route.



The application renders based on the route that matches the location

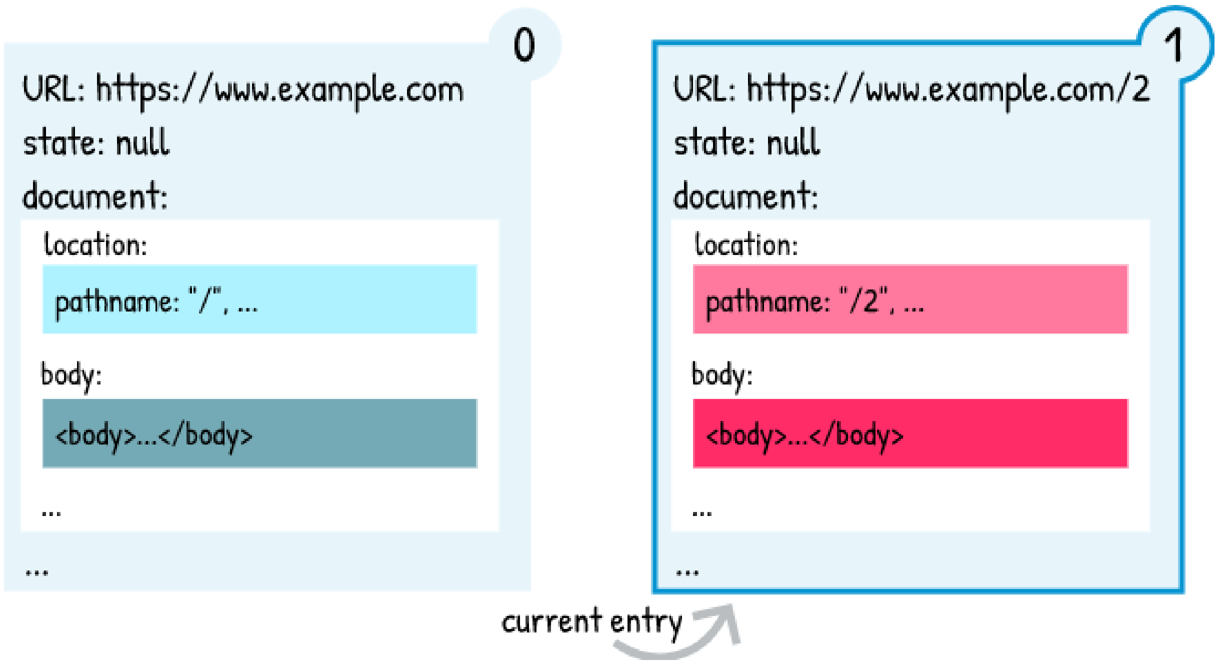
In-App Navigation

Navigation is the more interesting problem. When you click an anchor, the browser has native behavior attached to the event to trigger navigation. However, you can also attach your own click handler and override the native behavior (using `event.preventDefault()` in modern browsers). Without the native behavior the location will not change, so it is up to the click handler to trigger the navigation.

Before we can get to how click handlers can trigger navigation, we need to know a bit about how browsers normally handle navigation.

How Browsers Handle Locations

Each browser tab has a “browsing context”. The browsing context maintains a “session history”, which is essentially an array of location entries. An entry stores information about a location: its URL, the associated `Document`, serialized state, and a few more properties. An entry’s index specifies its position in the session history array. The browsing context also keeps track of which entry is currently active.



The session history is made up of entries

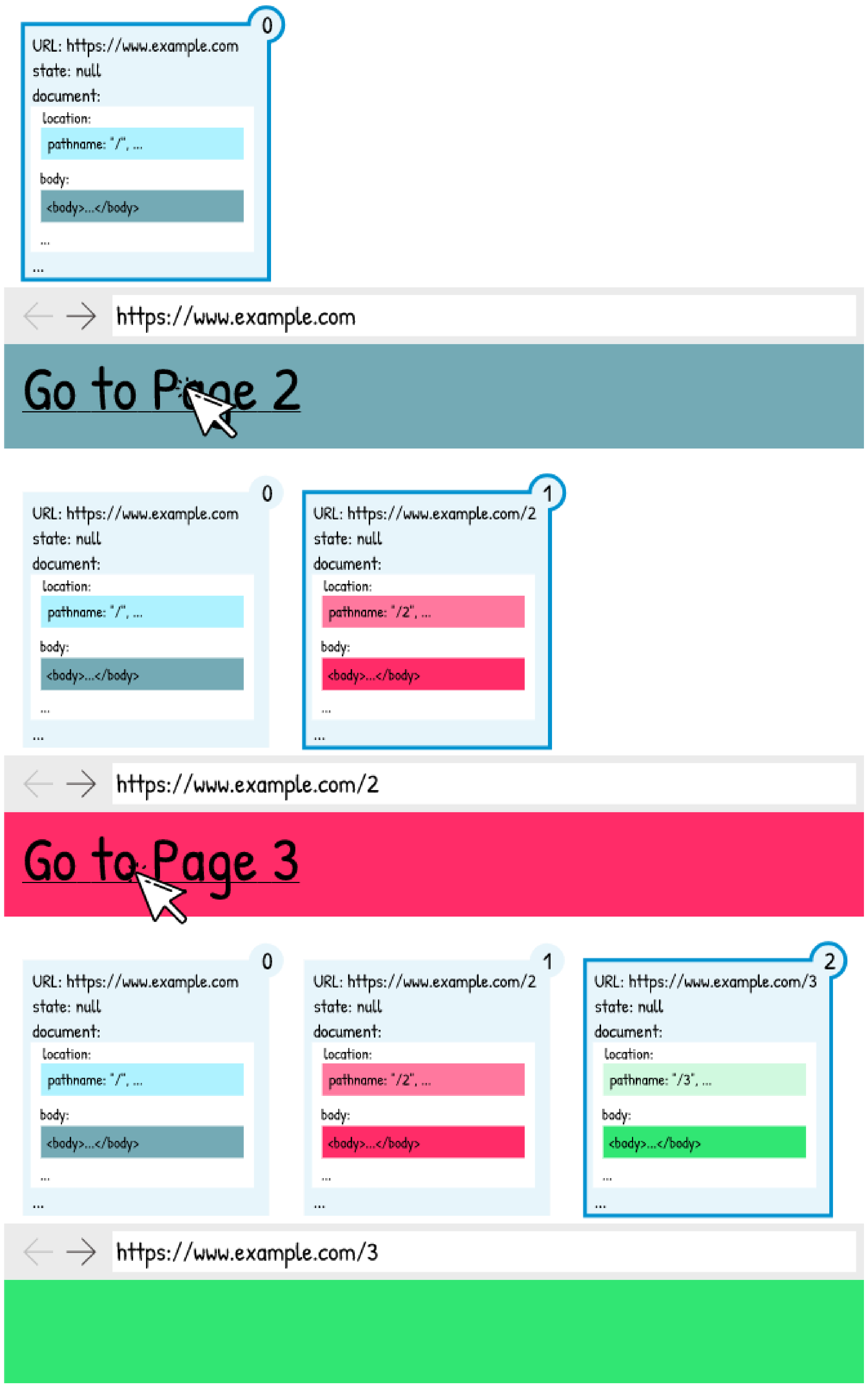
Document?

When a browser navigates, a request is sent to a server and the browser uses the response to create a `Document` object. The `Document` describes the page (the DOM) and provides methods for interacting with it. `window.document` is a browser tab’s active `Document`.

A Document is created from a Response

Session History

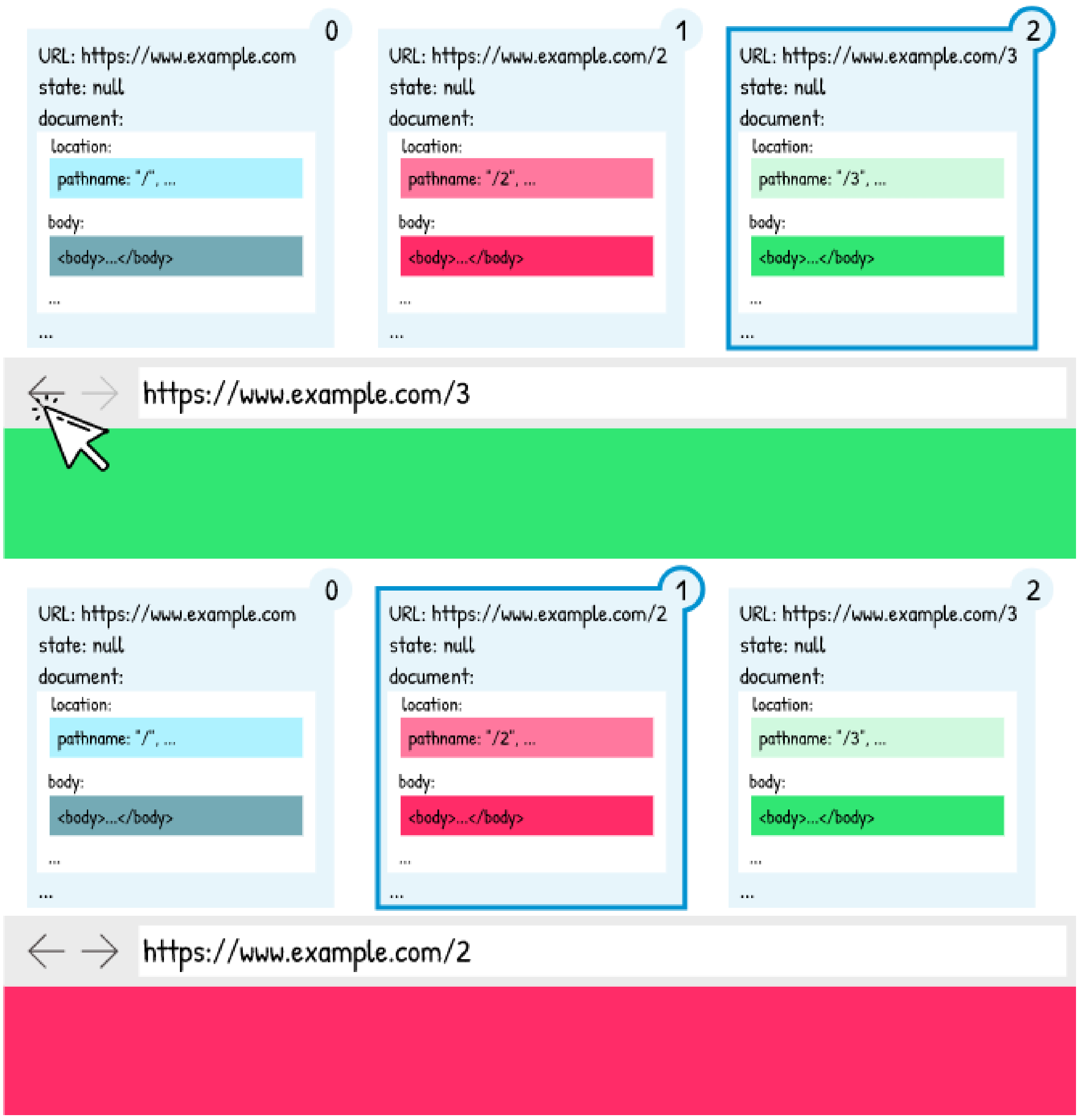
As you’re clicking links and navigating through pages, the browser tab is building up a session history. Each navigation makes a request to a server and creates a new entry (including a new `Document`).


0
URL: https://www.example.com
state: null
document:
location:
pathname: "/", ...
body:
<body>...</body>
...
...
https://www.example.com
Go to Page 2
0
URL: https://www.example.com
state: null
document:
location:
pathname: "/", ...
body:
<body>...</body>
...
...
1
URL: https://www.example.com/2
state: null
document:
location:
pathname: "/2", ...
body:
<body>...</body>
...
...
https://www.example.com/2
Go to Page 3
0
URL: https://www.example.com
state: null
document:
location:
pathname: "/", ...
body:
<body>...</body>
...
...
1
URL: https://www.example.com/2
state: null
document:
location:
pathname: "/2", ...
body:
<body>...</body>
...
...
2
URL: https://www.example.com/3
state: null
document:
location:
pathname: "/3", ...
body:
<body>...</body>
...
...
https://www.example.com/3

Each navigation appends a new entry to the session history

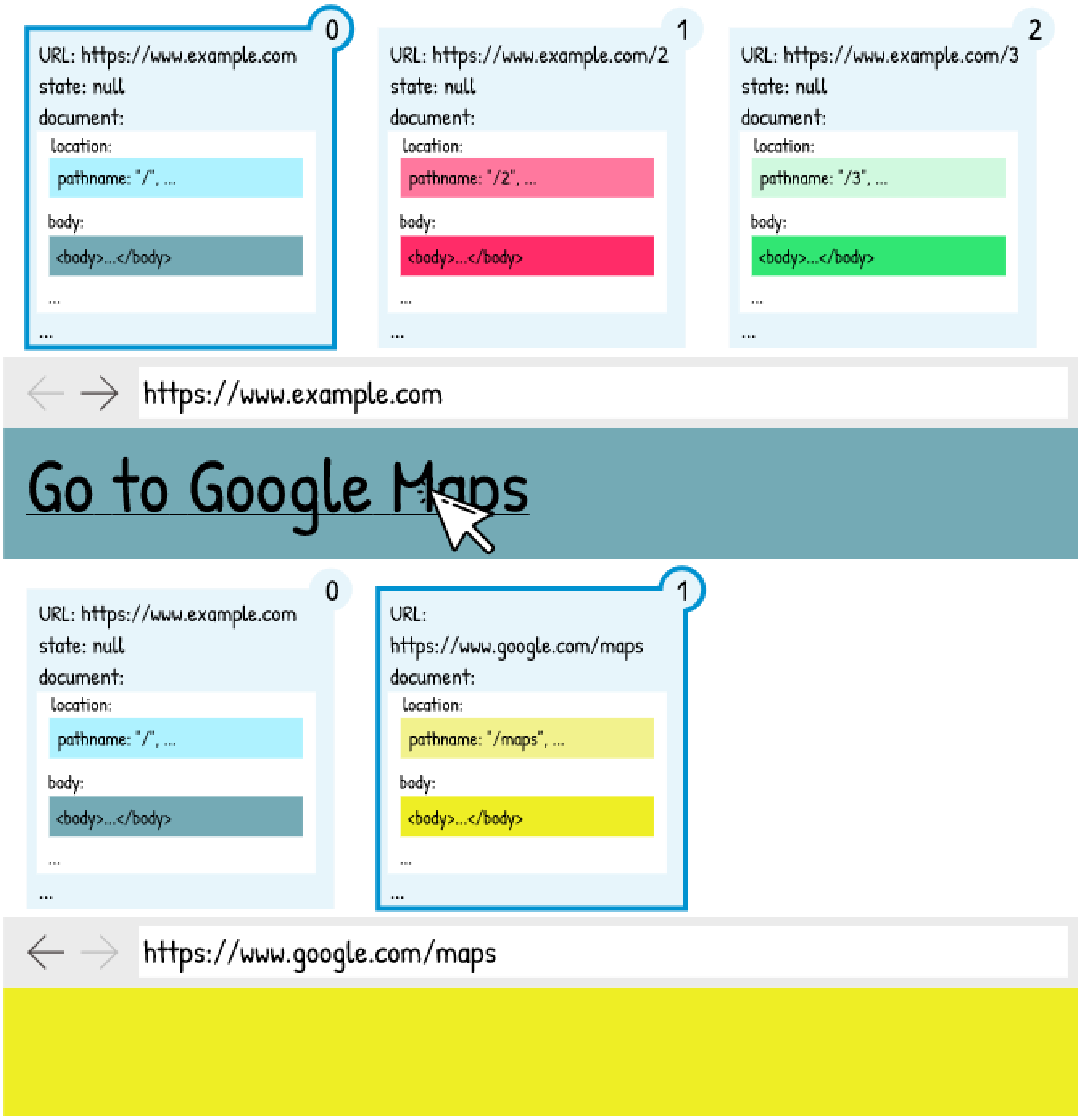
Note: Unique colors in the entry images imply different documents.

If you press the browser's back button, the browser will use the current entry to determine the "new" current entry (`current.index - 1`). The "old" current entry's `Document` will be unloaded and the "new" current entry's `Document` will be loaded.


0
URL: https://www.example.com
state: null
document:
location:
pathname: "/", ...
body:
<body>...</body>
1
URL: https://www.example.com/2
pathname: "/2", ...
2
URL: https://www.example.com/3
pathname: "/3", ...
https://www.example.com/3
https://www.example.com/2

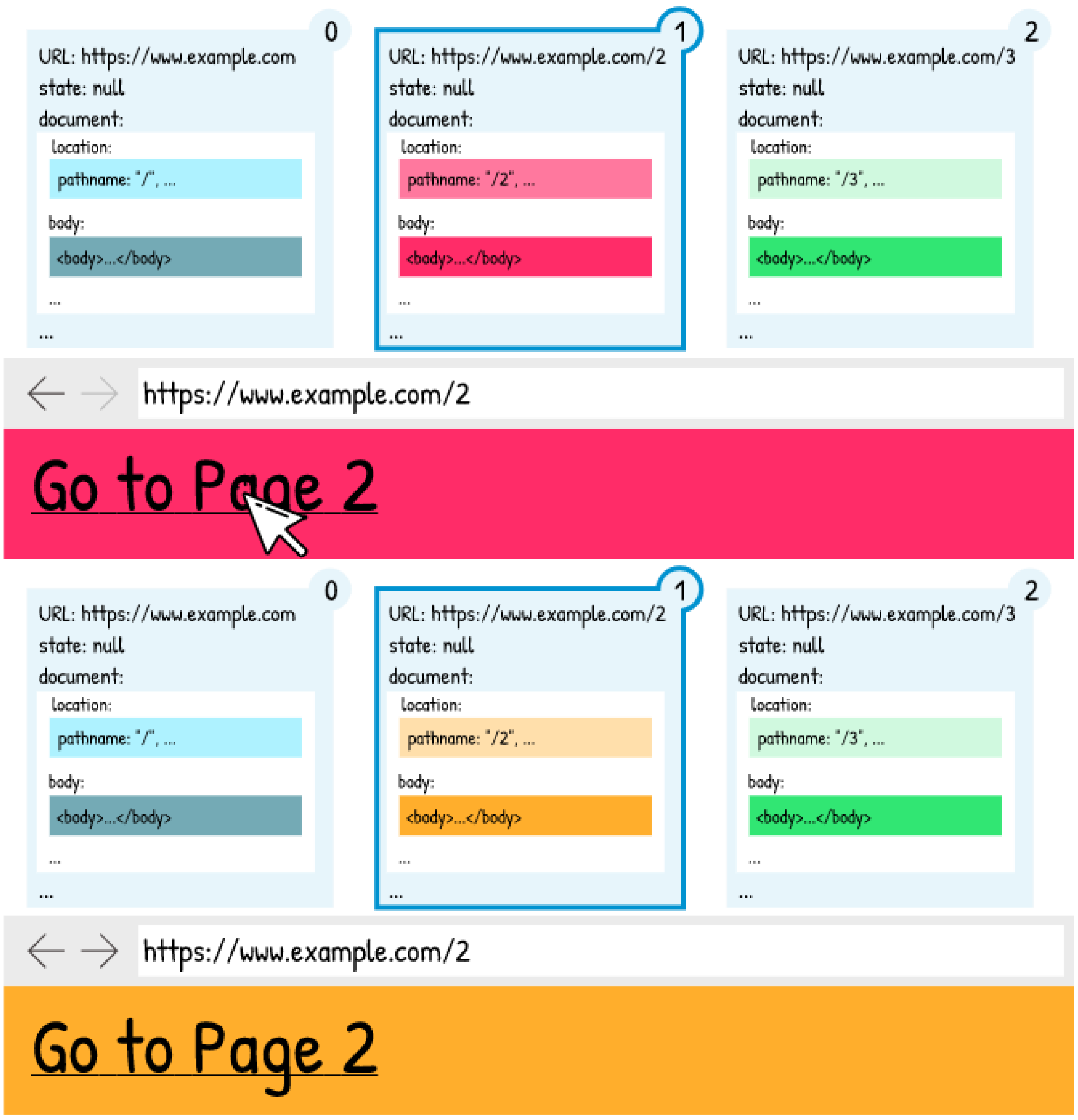
Clicking the back button switches entries (and Documents). The forward button has the same behavior.

When you click a link and there are entries after the current entry, they will be dropped.



Entries after the current entry are dropped when clicking a link

However, if you navigate to the exact same location as the current location (`pathname + search + hash`), then the current entry will be replaced without affecting succeeding entries.



Clicking a link with the exact same location as the current location will replace the current entry

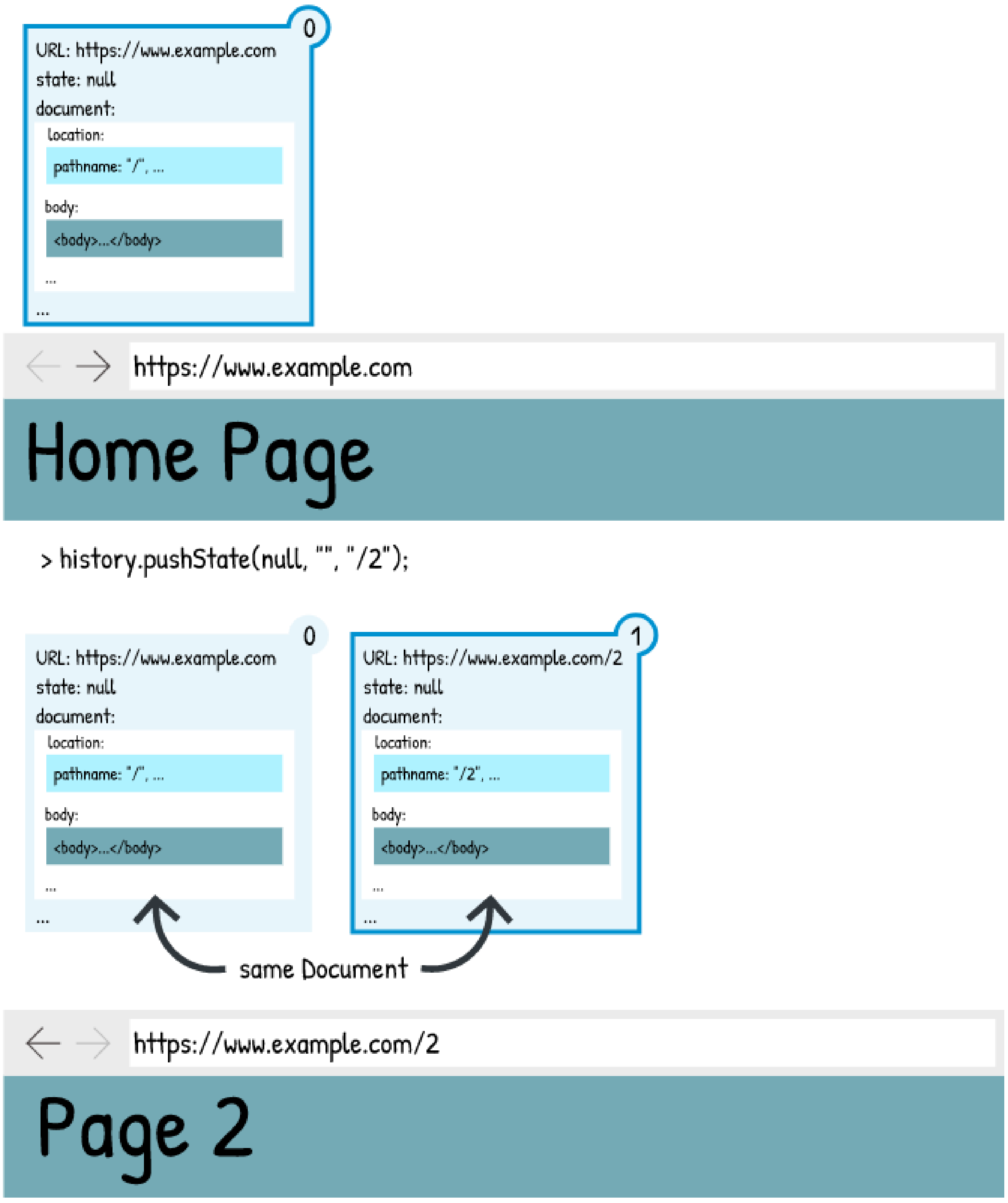
That is how native navigation works, but the point of a single-page application is to avoid requests hitting the server. What do SPAs do to avoid this?

The History API

Early single-page applications relied on the fact that you could change a location's hash and the browser would create a new location entry without sending a request to the server. This worked, but wasn't ideal, so the History API was developed to add first-class support for single-page applications.

Instead of creating a new `Document` for every location, the History API re-uses the active `Document` by updating it to reflect the new location.

The History API has three core functions: `pushState()`, `replaceState()`, and `go()`. These (and the rest of the API) are accessed via `window.history`.



Navigation with History API calls re-uses the active document

Note: Wondering about browser support? All major modern browsers support the History API. IE ≤9 does not support it, but these browsers are no longer supported themselves. Opera Mini does not run JavaScript on the client, which is why it doesn't support the History API.

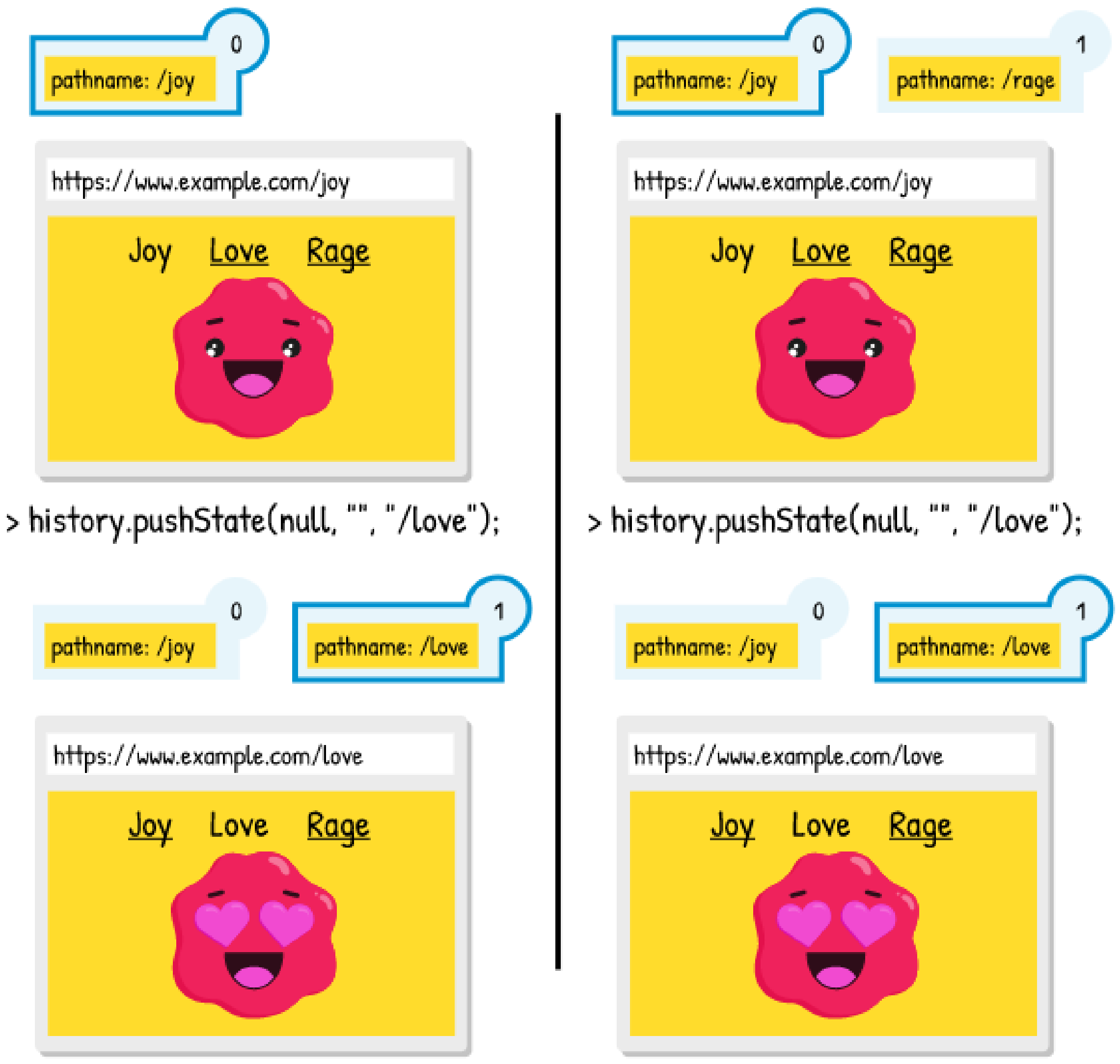
pushState() and replaceState()

Both `pushState()` and `replaceState()` have the same function signature.

1. The first argument is `state`. If you do not want to pass any state, pass `null`. It may be tempting to keep application state here, but there are some caveats that will be discussed later.
2. The second is a `title` string, but no browsers actually use this yet.
3. The third argument is the `path` that you want to navigate to. This can be a full URL, an absolute path, or a relative path, but it must be within the same application (same protocol and hostname). If it is not, a `DOMException` error will be thrown.

```
history.pushState(null, '', '/next-location');

history.replaceState(null, '', '/replace-location');

// attaching state to an entry

history.pushState({ msg: 'Hi!' }, '', '/greeting');

// while on medium.com

history.pushState(null, '', 'https://www.google.com');

// throws a DOMException
```

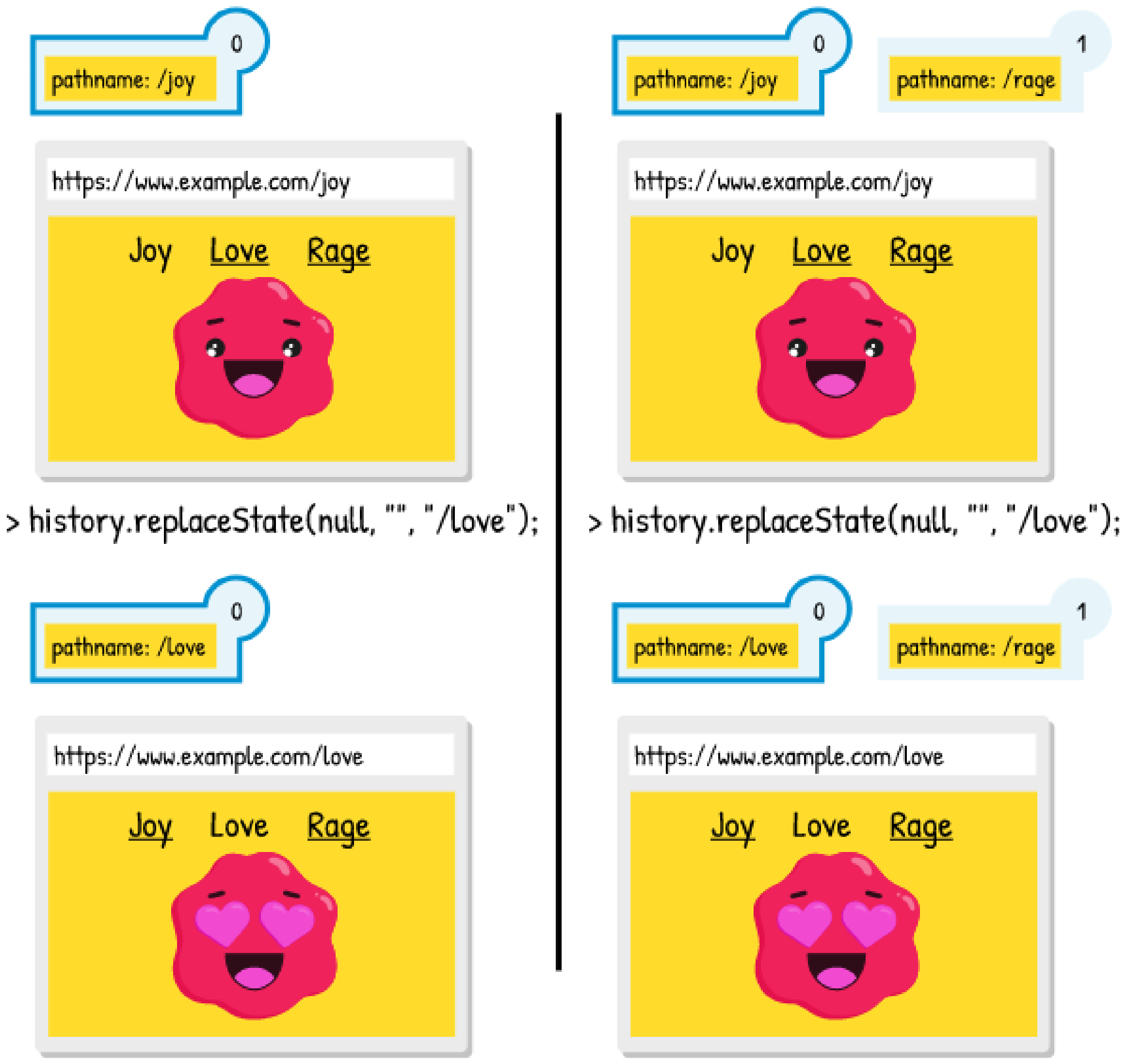
`history.pushState()` adds an entry to the session history after the current entry. If there are entries after the current entry, those are lost when you push a new location. This is the normal behavior for when you click an anchor.



0
pathname: /joy
https://www.example.com/joy
Joy Love Rage
> history.pushState(null, "", "/love");
0
pathname: /joy
1
pathname: /love
https://www.example.com/love
Joy Love Rage
0
pathname: /joy
1
pathname: /rage
https://www.example.com/joy
Joy Love Rage
> history.pushState(null, "", "/love");
0
pathname: /joy
1
pathname: /love
https://www.example.com/love
Joy Love Rage

pushState() pushes an entry after the current entry

`history.replaceState()` replaces the current entry in the session history. Any entries after the current entry are unaffected. This is similar to the behavior for clicking an anchor whose `href` is exactly the same as the current URL, except `replaceState()` can replace the current entry with any location.

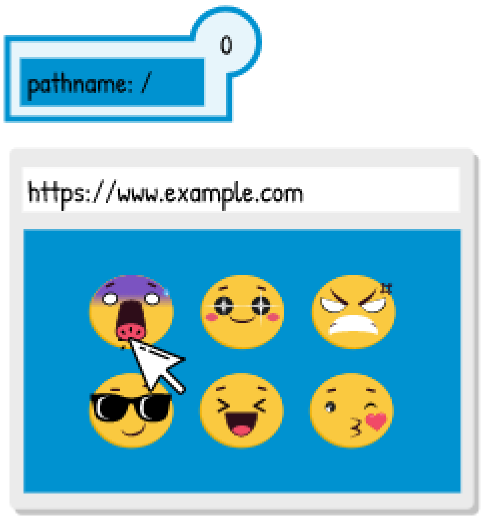

0
pathname: /joy
https://www.example.com/joy
Joy Love Rage
> history.replaceState(null, "", "/love");
0
pathname: /love
https://www.example.com/love
Joy Love Rage
0
pathname: /joy
1
pathname: /rage
https://www.example.com/joy
Joy Love Rage
> history.replaceState(null, "", "/love");
0
pathname: /love
1
pathname: /rage
https://www.example.com/love
Joy Love Rage

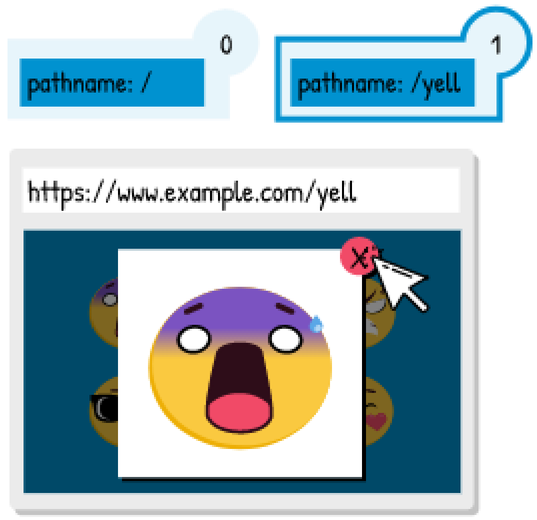
replaceState() replaces the current entry

go()

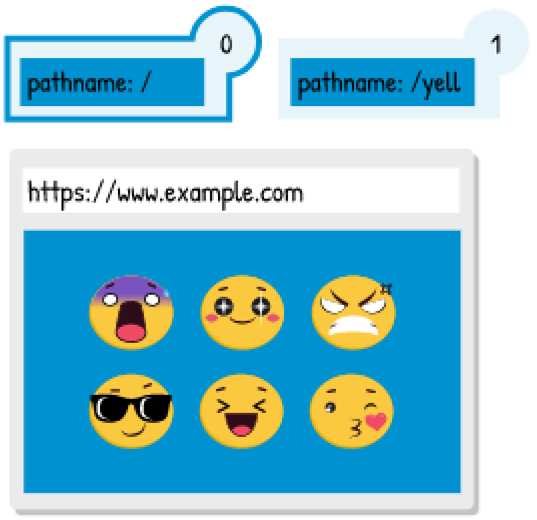
`go()` is the programmatic way of performing the same task as the browser's forward and back buttons.

`go()` can be useful when you know there is a previous page, like closing a location-aware modal, but I don't find many uses for it.





clicking the modal's close button calls:
go(-1);



go() is useful when you know that you can jump back to a previous location. If you aren't positive there is a location to go back to, you might accidentally leave your site!

`go()` takes a single argument, a number, which is the number of entries away from the current one you want to navigate. Positive numbers go forward, negative numbers go backwards, and zero (or `undefined`) reloads the page.

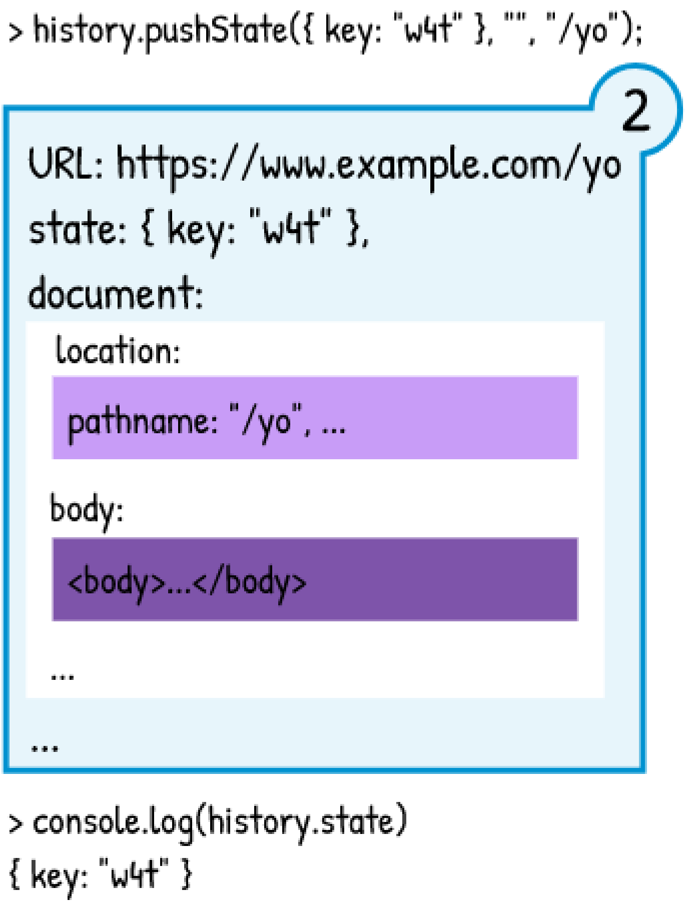
```
go(-1); // go back one entry

go(1); // go forward one entry

go(-10); // go way back

go(0); // reload

go(); // reload
```

There are also `history.back()` and `history.forward()` methods, which are the same as `history.go(-1)` and `history.go(1)`.

State

When entries were first mentioned, one of their mentioned properties was state. The `pushState()` and `replaceState()` methods also mention state directly in their name and we breezed over their first argument, which is state. So what is state?

State is data that is attached to any entry. It persists navigation, so if you add state to an entry, navigate away, and then go back to the entry, the state will still be there. State is attached to an entry with `pushState()` and `replaceState()`. You can access the current entry's state using `history.state`.



```
> history.pushState({ key: "w4t" }, "", "/yo");
```

```
> console.log(history.state)
{ key: "w4t" }
```

State can be attached to locations

There are some limitations to what state can be. First, it has to be serializable, which means that the data can be turned into a string. Second, there are size limitations (640k characters in Firefox), which you probably don't have to worry about, but do exist. Finally, when you navigate directly to a location its state is `null`, so if a page relies on state to render it will have issues with direct navigation. A good rule of thumb is that state should be used for non-rendered data, like a key to uniquely identify the entry or the URL to redirect to after a user logs in.

Navigating in SPAs using the History API

How do single-page applications take advantage of the History API? As mentioned above, we can add a click handler to anchors that overrides the native behavior using `event.preventDefault()`. The handler can call `history.pushState()/history.replaceState()` to perform the navigation without triggering a server request. However, the History API is only updating the session history, so the handler will also need to interact with the router to let it know the new location. Many routers use a History API wrapper to merge these steps.

There are various ways to add the handler to anchors. If you are using a rendering framework with components like React or Vue, you could write a special link component and attach the click handler directly to the component.

```
// React example

const Link = ({ children, href }) => (

  <a

    href={href}

    onClick={event => {

      // override native behavior

      event.preventDefault();

      // navigate using the History API

      // (ignoring replaceState() for brevity)

      history.pushState(null, '', href);

      // finally, let the router know navigation happened!

    }

  >

    {children}

  </a>

);

<Link href="/somewhere">Somewhere</Link>

// renders

<a href="/somewhere">Somewhere</a>

// but clicking on it will trigger a history.pushState() call
```

For more vanilla implementations, you could add a global event listener for clicks that detects in-app navigations, overrides the default behavior, and replaces it with a History API call.

Actual implementations are slightly more complicated because not all clicks should be overridden. If the user does a modified click (clicks while holding the ctrl, alt, shift, or meta keys), we want the browser to handle the navigation. The same goes for an anchor with a target attribute.

The History API combined with overriding native click behavior makes in-app navigation easy. However, we have another type of navigation to worry about: a user pressing the browser's forward and back buttons.

Detecting back/forward button navigation

When the back and forward buttons are clicked (as well as when `history.go()` is called), the browser emits a `popstate` event. In order to detect these, we can add an event listener to the window object.

```
window.addEventListener('popstate', event => {

  // let the router know navigation happened!

}, false);
```

The session history will already be updated by the time the event listener is called, so all we need to do is let the router know that the location has changed.

Manually navigating with the address bar

If a user updates the location manually using the address bar, that navigation will create a new `Document`. The History API only prevents reloads with entries that share the same Document. This means that calling `history.go()` and clicking the forward/back buttons to navigate between them will cause full page reloads!

Review

Single-page applications control navigation so that we re-use the active `Document` instead of sending a request to a server.

Routers are typically used to power route matching within a single-page application. Route matching is done when the location changes, determines which route matches the new location, and then triggers a re-render of the application.

In-app navigation is performed using the History API. The default click behavior for anchors is overridden to use `pushState()` and `replaceState()` to navigate. A `popstate` event listener is used to detect navigation with the browser's forward/back buttons. The click handler and event listener should both inform the router about the navigation to trigger the route matching cycle.

What about the server?

While most of a single-page application runs on the client, the files do have to come from somewhere.Single-Page Applications and the Server covers the various ways that you can serve a single-page application.

The problem with single page apps

Recently, I've had a few folks ask me how I typically handle routing with vanilla JS and single-page apps.

If you build a JavaScript app with more than one page or view, you'll probably need a way to determine which UI or layout to show based on the URL. That's routing.

We over-complicate this #

Single-page apps (or SPAs as they're sometimes called) serve all of the code for an entire multi-UI app from a single index.html file.

They use JavaScript to handle URL routing with real URLs. For this to work, you need to:

- Configure the server to point all paths on a domain back to the root index.html file. For example, todolist.com and todolist.com/lists should both point to the same file.
- Suppress the default behavior when someone clicks a link that points to another page in the app.
- Use more JavaScript—history.pushState()—to update the URL without triggering a page reload.
- Match the URL against a map of routes, and serve the right content based on it.
- If your URL has variable information in it (like a todolist ID, for example), parse that data out of the URL.
- Detect when someone clicks the browser's back button/forward button, and update the URL and UI.
- Update the title element on the page.
- Use even more JavaScript to dynamically focus the content area when the content changes (for screen-reader users).

You end up recreating with JavaScript a lot of the features the browser gives you out-of-the-box.

This becomes more code to maintain, more complexity to manage, and more things to break. It makes the whole app more fragile and bug-prone than it has to be.

I'm going to share some alternatives that I prefer.
That said, if using JavaScript to handle routing is something you're interested in, open sourced a routing plugin, Navigo, based on the article.

A simpler approach

If you don't use JavaScript to handle routing, what else can you do?

Let the browser load real HTML files located at the actual URLs. Instead of a single-page app, you build a multi-page app.

Looking at our list-making app, let's say we wanted users to have a settings page where they can choose whether or show items as a bulleted or numbered list, and clear all of their list data.

Create a directory in your app called settings, and add an index.html file there (this lets you create pretty URLs like list-maker.com/settings/ instead of having something like list-maker.com/settings.html).

To render the UI, you include a unique selector in the markup that describes the current view (such as [data-app="lists"] or [data-app="settings"]).

In your JavaScript file, you can do something like this to determine which template to use.

```
// Get the app container
var app = document.querySelector('[data-app]');

// Determine the view/UI
var page = app.getAttribute('data-app');

// Render the correct UI
if (page === 'lists') {
	// Render the homepage...
}

if (page === 'settings') {
	// Render the settings page...
}
```

Why is this better than using JavaScript routing?

A few reasons:

- Support for the browser's forward and backward buttons are baked in. You don't need to do anything to make that work.
- You don't need to intercept clicks and determine if the clicked link points to an internal link or an external one. You just let them all resolve.
- You don't need to use complex regex patterns or another library to parse the URL and determine which view or UI to render. It's baked into the markup already.
- It's simpler and easier to implement.

A counter argument might be that using JavaScript routing results in faster apps because you avoid a page reload.

That can be true, but if you use front end performance best practices and load static HTML files, I find that page loads using this approach feel nearly instantaneous.