Engineer

Single-page application

a web app that loads one page and changes it with JavaScript

What it is

A single-page application, also written single-page app and usually shortened to SPA, is a web application that loads one HTML document and then changes that document with JavaScript instead of asking the server for a new page. Clicking a link inside it does not unload the page. A script catches the click, fetches whatever data the next view needs, rewrites part of the DOM and updates the address bar, so the URL still says where the reader is.

No standard owns the term, and the people who build the tools admit as much. Next.js opens its guide to the subject with “The definition of a SPA varies.” and then fixes one for its own use, which it calls a strict SPA. It has two traits. A single HTML file serves the whole app, and JavaScript in the browser takes care of every route, every page transition and every request for data. And the page never reloads in full: when the route changes, scripts on the client rewrite the DOM of the page that is already open and load data when it is wanted, instead of asking the server for a fresh document. This article uses that definition because it is the most direct one on offer and names the property everything else depends on, a document that is never replaced.

The browser parts are standard. The HTML Standard’s History interface gives a page `history.pushState()`, which adds a new entry to the session history and sets the URL without loading anything, and `history.replaceState()`, which rewrites the current entry instead. When the reader goes Back or Forward between those entries, the browser fires a `popstate` event at the window and leaves the redrawing to the page. The newer Navigation API, which the standard presents as the successor to the classic location and history APIs, works from the other end: a `navigate` event fires when a navigation begins, and its `intercept()` method turns that navigation into a same-document one that the page handles itself.

The opposite is the ordinary multi-page website, where every link asks the server for a fresh document and the browser throws the old one away. Whether a page's HTML is made once by a static site generator or on each request by server-side rendering is a separate question, about how the HTML is produced rather than what happens after it arrives. Next.js blurs the line on purpose: its static export writes an HTML file for every route, and moving between them still happens on the client.

Use the concept in a decision

Put "What is a single-page application?" to work in a real decision. We can help relate the concept to the system you are building or buying. Interfaces that load quickly, work on a phone and behave the same in every browser your customers happen to use.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Directed full-stack GIS map development — full-stack mapping, from database to browser

Directed full-stack GIS map development Get in touch →

Why it matters

Because the document is never replaced, whatever lives in it survives a click: a half-filled form, a playing video, the scroll position of a side panel, data already fetched. Only the data the next view needs has to cross the network, not a whole new page with its header, styles and scripts. A change of view can be as small as rewriting one region of the page.

Next.js names the costs in the same breath as the definition. A strict SPA often has to load a large amount of JavaScript before the page can be interactive, and client data waterfalls, where one request can start only after another has come back, are hard to manage. Anything that reads the page without running scripts pays as well. With one HTML file behind every route, a crawler that does not run JavaScript gets the same nearly empty document whatever URL it asks for. Next.js makes the same point about its own browser-only components: anything that indexes the page without running scripts gets the fallback, not the component.

Its answer is to keep the client-side navigation and move the rest. Next.js splits JavaScript bundles automatically and generates multiple HTML entry points into different routes, so a visitor does not load code a route does not need. Its `next/link` component prefetches routes, which makes route changes feel as immediate as those in a fully client-rendered SPA while keeping the application's

routing state in the URL, where it can be linked to and shared. A static export goes further and writes a fully rendered page for each route, so visitors get content without waiting for the client bundle, while client-side transitions between pages stay as quick as in a strict SPA.

Nor is the choice made once and for all. Next.js describes a project that starts as a static site, or even as a strict SPA rendered entirely on the client, and adds server features as it grows, and it says an existing SPA can move across without being rewritten from scratch.

What it looks like in practice

The router below, written for this article, is the smallest working version of the idea. It assumes the page has a `<main>` element and that the server answers every app path with the same HTML document. The two views stand in for whatever the app really shows.

```
// router.js
const views = {
  "/": { title: "Orders", text: "Your recent orders." },
  "/settings": { title: "Settings", text: "Delivery preferences." },
};
const main = document.querySelector("main");

function show(path, moveFocus) {
  const view = views[path] ?? views["/"];
  const heading = document.createElement("h1");
  const body = document.createElement("p");
  heading.textContent = view.title;
  heading.tabIndex = -1;
  body.textContent = view.text;
  main.replaceChildren(heading, body);
  document.title = view.title;
  if (moveFocus) heading.focus();
}

document.addEventListener("click", (event) => {
  const link = event.target.closest("a[href]");
  const modified = event.metaKey || event.ctrlKey || event.shiftKey;
  if (!link || link.origin !== location.origin) return;
  if (modified || event.button !== 0) return;
  event.preventDefault();
  history.pushState(null, "", link.href);
  show(location.pathname, true);
});
```

```
window.addEventListener("popstate", () => show(location.pathname, true));
show(location.pathname, false);
```

The click handler takes over plain left clicks on links to the same origin and lets modified clicks through, so opening a link in a new tab still works. `preventDefault()` stops the browser's own navigation, and `pushState()` records a new session history entry with the link's URL; its middle argument is there only for historical reasons, which is why it is an empty string. `show()` replaces the view, sets the document title and moves focus to the new heading, which a `tabIndex` of `-1` makes focusable from script while keeping it out of the Tab order. The `popstate` listener redraws the matching view when the reader goes Back or Forward, and the last line draws whichever URL the page was opened at, which only works if the server sends the same document for `/settings` as for `/`. A real app would also show its own not-found view instead of quietly falling back to the start page, and would leave links to files and to other parts of the site alone.

In the founder's current work under NextMariner, the Next.js 16 frontend shows the cost side of this on a real product. Its portfolio entry describes an authenticated dashboard that had to feel quick and stay genuinely interactive, and names what happens when everything is fetched on the client: the first load drags, and the N+1 pattern appears, with every component waking up and sending its own request until a single page becomes a cascade of round trips. That is the client data waterfall the Next.js guide warns about, met in practice. The fix kept client-side interactivity where it mattered and moved the first fetch to the server. On the dashboard and the big directory listings, a factory called `createPrefetchedServerPage` hands each page to the client with its data already in place. The entry is candid about the rest: the other authenticated pages are still rendered on the client and wait their turn, so it describes a migration that works and has an obvious next page, not one that is complete.

Where it goes wrong

The Back button leaves the app. If a script swaps views without calling `pushState()`, the session history has no entry for them. Back skips every view the reader saw and leaves the app altogether, and a reload lands on whatever URL the app was first opened at. Push an entry for each view a reader would call a page, use `replaceState()` for changes that should not become steps of their

own, such as a filter being typed, and handle `popstate`. Next.js argues for keeping routing state in the URL for the same reason: it is what lets a view be linked to and shared.

A deep link meets a server that knows one file. In a strict SPA one HTML file serves every route, but the address bar shows `/settings`, and that is what the browser asks for on a reload or when someone opens a shared link. A server that only has `index.html` answers with a 404. Either configure the server to answer every app path with the document, or generate a file per route. Next.js's static export does the second: its `out` folder holds files such as `index.html`, `404.html` and `blog/post-1.html`, and its guide shows an nginx rule, `try_files $uri $uri.html $uri/ =404;`, that maps a clean URL onto the matching file.

Focus stays where the old view was. `pushState()` changes the session history and the URL and nothing else. After a script swaps the view, keyboard focus stays on the link that was clicked, or is lost if the element holding it was removed, and nothing tells assistive technology that a new view has arrived unless the page does. Set the title and move focus to the new view's heading, as the example does. The Navigation API takes on more of this itself. A navigation handled with `intercept()` resets focus by default, to the first element with `autofocus` or else to the body, and once the handler finishes, the browser signals completion to the user, which the standard says includes assistive technology. An app that places focus itself opts out with `focusReset: "manual"`.

Everything is paid for up front. A strict SPA often has to load a large amount of JavaScript before anything is interactive, and its first data requests start only once that code runs. Split the bundle by route, so that a visitor downloads only what the current view needs, and start the first requests on the server. The Next.js pattern for this fetches data in a Server Component without awaiting it and passes the promise to a Client Component, which unwraps it with `use()`. Because the request begins before the rest of the app renders, the guide says, the response can start streaming at once, and the client is spared a waterfall of requests, each waiting for the one before it.

Further reading

- How to build single-page applications with Next.js — the strict SPA definition, what it costs, and the patterns Next.js uses to keep client-side navigation while moving work to the server.

- HTML Standard: The History interface — `pushState()` , `replaceState()` and how session history entries are added and replaced.
- HTML Standard: Navigation API — the `navigate` event, `intercept()` , and the focus and scroll handling that comes with it.
- How to create a static export of your Next.js application — one HTML file per route, what a static export cannot do, and serving clean URLs with nginx.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a single-page application?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/single-page-application/