Engineer

Progressive enhancement

building pages that work in plain HTML, then adding more

What it is

Progressive enhancement is a way of building web pages in layers, starting from the one that always works. The base is HTML that carries all of the content and completes every task on its own: links lead somewhere, forms submit to a server, and the page can be read and used with nothing else loaded. Style sheets go on top to lay the page out and make it pleasant to look at. Scripts come last, to make it quicker or smoother to use. Each layer improves the one beneath it, and the page does not depend on any layer above the HTML to do its job. The GOV.UK Service Manual sums the idea up as making a page work with HTML first.

The name comes from a talk by Steven Champeon and Nick Finck, *Inclusive Web Design For the Future with Progressive Enhancement*, which their paper says was delivered on 11 March 2003 at SXSW in Austin, Texas. The paper writes it with capitals, Progressive Enhancement, and shortens it to PE. It sets the idea against graceful degradation, the older habit of building for the most capable browser and hoping that the others could still get at the content. PE, as the paper describes it, starts with a document that the least capable devices can use, then adds presentation as separate logic, so that the basic devices are not weighed down by what the modern browsers get.

The difference between the two is the direction of the work. Graceful degradation starts from the full experience and asks what is left when parts of it are missing. Progressive enhancement starts from what is always there and asks what can be added. In a current browser on a good connection both can end at the same page. They differ when something fails: a page built from the baseline up still has its baseline, and a page built from the top down has whatever its authors remembered to provide.

It is a way of working, not a library or a feature. Nothing in HTML, CSS or JavaScript is called progressive enhancement, but the standards leave room for it. A browser skips a CSS declaration it does not understand and ignores a script whose type it does not recognise, and the `@supports` rule lets a style sheet ask whether a feature is there before relying on it.

Use the concept in a decision

Put “What is progressive enhancement?” 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

The layers fail in different ways, and the GOV.UK Service Manual spells out how. The CSS layer is reasonably forgiving: a browser skips a declaration it does not recognise and carries on, although a selector it does not know makes it drop the whole rule. The JavaScript layer is not forgiving at all. When a script uses syntax or an interface the browser lacks, the rest of that script does not run. A page that needs its script for its main task is therefore only as reliable as every step between the server and the reader.

There are a lot of those steps. The same page lists what can stop the CSS or the JavaScript from loading or running: a temporary network error, an ad blocker, a content delivery network that is down, a DNS failure, a bug in a browser update, a corporate or a personal firewall. None of them has anything to do with an old browser, and the reader cannot tell which one it was. They see a button that does nothing. The manual gives no figures for how often each one happens; its argument rests on the list, not on a percentage.

Some readers also switch things off on purpose. The manual says that some users deliberately turn off features in their browsers and that their choice should be respected. The HTML Standard says that many users disable scripting, and for that reason, and because declarative features are often easier to maintain, it encourages authors to use them in place of script where they can. Its example is

the `details` element, which shows and hides a section of a page without a line of JavaScript. The standard also asks authors to make their applications degrade gracefully when there is no scripting support.

For UK government services the manual turns this into a rule: they should be functional using only HTML, and JavaScript, where it is used, should only enhance the HTML and CSS so that people can still use the service when the script fails. That rule is written for one government's services, not for the whole web, but its reasoning carries over to any site whose readers its authors cannot choose.

There is a gain for the team as well. When an enhancement breaks, the reader loses the improvement, not the task, so a fault in a script is an annoyance rather than an outage. A new feature can be shipped to the browsers that support it without a plan for the ones that do not, because those keep the page they already had.

What it looks like in practice

The HTML Standard's own example of graceful behaviour without script is a table whose column headings re-sort it, and it is a fair test of the approach. The baseline is a table whose headings are ordinary links. A click asks the server for the same table sorted by that column, and the server sends it back in that order. This works in any browser, with or without a script.

```
<table id="orders">
  <thead>
    <tr>
      <th><a href="?sort=date">Date</a></th>
      <th><a href="?sort=total">Total</a></th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>2026-10-20</td>
      <td>120</td>
    </tr>
    <tr>
      <td>2026-10-24</td>
      <td>45</td>
    </tr>
  </tbody>
</table>
<script type="module" src="sort.mjs"></script>
```

The script is the enhancement. It finds the same links, cancels the navigation and reorders the rows in the page, which saves a round trip to the server.

```
const table = document.querySelector("#orders");
const body = table.tBodies[0];
const collator = new Intl.Collator("en", { numeric: true });

function sortBy(column) {
  const text = (row) => row.cells[column].textContent;
  const rows = [...body.rows];
  rows.sort((a, b) => collator.compare(text(a), text(b)));
  body.append(...rows);
}

for (const link of table.tHead.querySelectorAll("a")) {
  link.addEventListener("click", (event) => {
    event.preventDefault();
    sortBy(link.closest("th").cellIndex);
  });
}
```

Failures before a listener is attached leave that link working. If `sort.mjs` never arrives, the links do what they did before. If the script throws halfway, the headings whose listeners it never reached fall back to the server. A browser that does not support module scripts treats `type="module"` as a script type it does not know and, as the HTML Standard says, ignores it. A module script without `async` is run once the page has finished parsing, so the table is there by the time the script looks for it. This example does not recover if an attached click handler throws after `preventDefault()` has cancelled navigation.

The price is on the server: the baseline needs an endpoint that can sort, and the two ways of sorting have to agree. That is the usual cost of the approach, paid once. The intended benefit is a page whose main task remains usable through the server fallback; an attached handler must preserve that fallback when its own work fails.

This site makes its baseline explicit. Its accessibility statement describes complete HTML, CSS and SVG pages with no executable JavaScript. The search code is retained but disabled, so there is no in-page filter to depend on. The statement also names a remaining limit: the small labels shown on icon hover or focus cannot be dismissed with a key, which would need browser code the site does not ship. That is why it meets 1.4.13 Content on Hover or Focus only in part.

Where it goes wrong

A `noscript` block stands in for a baseline. The `noscript` element shows its content only when scripting is disabled, and the HTML Standard is blunt about it: “The noscript element is a blunt instrument.” Scripting can be enabled and the page’s script can still fail, and then the reader gets neither the scripted page nor the fallback. Start from a page that works without the script and let the script change it, as the standard’s own example does when its script adds a live result to a form and removes the form’s submit button.

One unsupported feature stops the whole script. Because the JavaScript layer is not fault-tolerant, a single newer piece of syntax in a bundle can take every enhancement in that bundle down with it, including those that would have worked. Keep the baseline independent of the script, so that a failure costs the enhancement and nothing else. Load modern code with `type="module"`, which older browsers skip, and if older browsers need an enhanced version at all, give them a separate classic bundle marked `nomodule`, which browsers that support modules do not run.

A new selector takes good rules down with it. A browser that meets a selector it does not know drops the entire rule, declarations and all. Listing a recent pseudo-class next to selectors that every browser understands can therefore remove styles from browsers that would have applied them. Give a selector that some browsers may not support a rule of its own.

Fallback styles leak into the enhanced layout. Widths written for browsers without a newer layout system can fight that layout system in the browsers that have it. CSS Conditional Rules Level 3 shows the fix with a flex layout and an `@supports not (display: flex)` block for the alternative, and notes that the width declarations could harm the flex layout, which is why they belong only in the styles for browsers without it.

Images are chosen by script. The HTML Standard points out that if a page names no image in its markup and leaves the choice to a script, no image is downloaded at all for readers with scripting disabled, and if a script replaces an image the markup already named, the browser may start two downloads. Declare the candidates in HTML with `srcset` and `sizes`, or with `picture`, and let the browser choose.

Behaviour with nothing underneath. Champeon and Finck's strategy list asks for a fallback for all behaviour: a real `href` behind every click handler, and a check on the server behind every check in the browser. A control that exists only as a click handler, or a form that is validated only in the browser, has no baseline at all. Use real links and real form submissions first, always validate on the server, and treat validation in the browser as a convenience added on top.

Treating it as a matter of old browsers. A team that thinks the baseline is for old browsers drops it once those browsers are gone. The GOV.UK list of causes is about networks, blockers, firewalls and broken updates, which current browsers meet as well. Keep the baseline as a path that is used and tested: load the page with scripts blocked now and then, and carry out its main task from start to finish.

Further reading

- Using progressive enhancement — the GOV.UK Service Manual's guidance, last updated 27 September 2024: the three layers, how each one fails, and why government services should work with HTML alone.
- HTML Standard: Scripting — the `script` element with `type="module"` and `nomodule`, and the `noscript` element with the advice against relying on it.
- CSS Conditional Rules Module Level 3 — the W3C Candidate Recommendation Draft of 15 August 2024 that defines `@supports` with its `not`, `and` and `or`.
- HTML Standard: Images — `srcset`, `sizes` and `picture`, and why choosing an image with script fails readers who have scripting turned off.

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 progressive enhancement?” 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/progressive-enhancement/