Houdini's Magic Trick for the Enterprise

> Talk by Markus Walther, AXA DevCon, June 4, 2024 @ Winterthur, Switzerland

I am Markus, but Who is Houdini?

Harry Houdini was a professional American magician and escape artist.

He was famous at the start of the last century for his stunts in which he invariably escaped impossible-looking situations.

In the Washington DC prison escape of 19-oh-6 it took him all of 18 minutes to defeat his handcuffs, open 5 locks of his cell, and get his clothes from the neighbouring cell.

> "Stone walls and chains do not make a prison - for Houdini"

reads the caption of his photo.

And What is Enterprise?

Enterprise IT serves the needs of a large business organization — an enterprise — not just an individual user.

Things have grown over the years, one has to protect against hostile actors from outside, auditing requirements need to be met, users have access rights and restrictions.

For a frontend software engineer working in Enterprise, terms like stonewalls, chains, prison feel familiar - your hands are often tied, you can't move fast and break things, things are locked down, yet the show must go on and your web site must be up and running 24/7.

Where's Houdini in that situation?

What is Enterprise Frontend like?

As everybody knows, frontend deals with HTML - the structure, 'what is what' -, CSS - the styling, 'how it looks', and JavaScript - the interactivity, 'how it responds'.

Now typical enterprise restrictions are:

- traditional, web pages dominate, lots and lots of HTML. **Not** SPAs, **not** PWAs.
- you don't own the whole page
- you can't run JavaScript on the server
- SEO is crucial

So, out goes:

- most frontend frameworks, and
- whole-page performance optimization

Houdini, please help!

Here comes The Magic Trick

Whenever you're stuck in yet another impossible Enterprise situation, do this:

> write a web component!

That's it, that's the escape, that's Houdini's magic trick for the enterprise, you can do it basically every single time.

<web-component>s are HTML tags with a dash in the name that you define yourself with a little JavaScript. All major browsers support them.

And since the foundation is always HTML, we can always use that trick.

How?

I'll show you now.

Here's five typical situations where the trick works

1. Situation: UI components

Most people know this one already.

A design system is good for enterprise brand identity and includes UI components like buttons, form input elements, date pickers and so on.

AXA's design system is open source and built with web components since 2019.

It's called the AXA Pattern Library.

Those web components are live on every page of axa.ch.

Houdini's trick simply works. These days many enterprises do the same.

Therefore, let's move on to the next situation.

2. Situation: Slow Page Performance

So your boss tells you to make things load faster, because the Google rating dropped...

"Oh", he says, "and can you please make it happen by tomorrow?"

But you don't own the pages, hundreds of kilobytes of JavaScript and CSS assets must stay.

Those assets are loaded via `<script>` and `<link>` tags, as usual, which *also* have to stay in place.

Should you give up, or can you escape the impossible situation?

Of course, you can escape!

Just use Houdini's trick!

Step 1: wish yourself a new HTML tag called script-with-a-dash (`<scri-pt>`) with new behaviour.

Here we want at least two things.

One is to delay loading an asset until the browser is idle. Idle means the browser is not busy doing more important things like painting pixels.

The other behaviour is to delay loading until you scroll down *so much* so that the new tag becomes visible.

Step 2: invent your own attributes to control that new behaviour.

Because "script when visible, or script when idle" sounds good, we decide to invent the `when` attribute:

```
<scri-pt when="visible">...</scri-pt>
<scri-pt when="idle">...</scri-pt>
```

Step 3: compose your markup as a mix of old tags and the new one.

Because nothing loads automatically inside a `<template>` tag, we can wrap the things to load with a template and then put script-with-a-dash around it:

```
<scri-pt when="visible">
  <template>
    <script src="big-fat-bundle.js"></script>
  </template>
</scri-pt>
```

That snippet means:

load that big fat bundle *only* when the user scrolled down *so much* that script-with-a-dash is visible.

Step 4: now do-it-yourself, *define* the new behaviour.

Keep it simple.

No framework, just plain old JavaScript:

```
class ScriPt extends HTMLElement {
  connectedCallback() {
    if (this.getAttribute("when") === "visible") {
      const observer = new IntersectionObserver( entries => entries[0].isInte
      return observer.observe(this);
    }
    requestIdleCallback(this.#load);
  }

  #load = () => this.appendChild(this.firstElementChild.content.cloneNode(tru
}

customElements.define("scri-pt", ScriPt);
```

Aaand... in just 11 lines of code, Houdini's escape is perfect: script-with-a-dash does the trick!

To see a production version of this *live* — under the alias axa-script,— just visit *any* axa.ch page.

3. Situation: Microfrontends

Microfrontends boil down to one simple idea:

> embed apps in HTML.

Such apps are written in JavaScript and placed *somewhere* on a web page.

They could be good for anything, like calculating a mortgage or show your insurances or whatever.

If you do it right, teams can then deploy and maintain such apps *independent* of your site releases, which is a huge win.

But not all apps are alike.

For example, some might first require login for user authentication.

Only *after* login can the app start and render itself.

Others require *waiting* till all markup is parsed by the browser, also known as 'DOM loaded'.

How can you embed *all* such types of microfrontend apps in HTML?

You guessed it: easy-peasy if you use Houdini's trick!

First, do step 1 again: wish yourself micro-dash-frontend (`<micro-frontend>`).

That is to say, a web component which knows how to embed *and* start any type of app.

Step 2, add attributes.

Use `type` to indicate which kind of microfrontend it is:

for example use `type="auth"` for authentication-first, or `type="dom-loaded"` and so on.

Step 3, compose:

```
<micro-frontend type="dom-loaded">
  <scri-pt when="visible" for="parentNode">
    <template>
      <script type="module" src="big-app-bundle.js"></script>
    </template>
  </scri-pt>
</micro-frontend>
```

That snippet means: place a microfrontend of type dom-loaded here.

The app will start and render under micro-dash-frontend once we are visible and the big app bundle has loaded.

Did you notice something?

Yes, that's right: we did not have to define app-loading!

We simply reused script-with-a-dash...

That's the power of composition at work!

Finally Step 4: do-it-yourself, define micro-dash-frontend in pure JavaScript.

One noteworthy detail is having to wait for the app-loading child element to finish, which can be done with a Promise.

I will spare you further details, but it's only 3-point-4 kiloBytes compressed.

Finally, I invite you to see micro-dash-frontend for yourself!

It's *live* on axa.ch pages, under the alias webhub-pod.

Situation 4. Internationalization

Imagine customer research has found that some users *hate* you.

They click on that link in their email, and don't understand the German page that opens.

Then they use the language selector to switch to English or whatever.

And *then* they *really* hate you because they need to wait while the page loads *again*.

Your boss yells: "no reload, but don't you dare and change anything else".

"And no performance regression, please."

Impossible situation again, or not?

How about... a little inspiration from Mister Houdini?

Step 1: wish yourself trans-dash-late (`<trans-late>`).

A web component which knows how to translate text when the page language changes.

Skip step 2: no attributes are needed!

Why? Because we can just observe updates of the `lang` attribute:

```
<html lang="en">
```

And changing *that* requires 1 line of JavaScript in the language selector.

Moving on to Step 3: compose:

```
<trans-late>Houdini's Magic Trick for the Enterprise
    <template lang="de">Houdini's Magischer Trick für Enterprise Frontend</te
```

```
</trans-late>
```

Anything under template is not rendered by the browser, so we're fine visually.

The HTML is so clear, it speaks for itself.

Finally, Step 4: do-it-yourself, define trans-dash-late in JavaScript.

```
const textNodes = new Map();

const getPageLanguage = () => document.documentElement.getAttribute("lang");

let pageLanguage = getPageLanguage();

const observer = new MutationObserver(() => {
  const language = getPageLanguage();
  for (let [textNode, translations] of textNodes) {
    textNode.textContent = translations[language];
  }
});

observer.observe(document.documentElement, {
  attributes: true,
  attributeFilter: ["lang"],
});

class TransLate extends HTMLElement {
  connectedCallback() {
    const text = this.textContent.trim();
    const textNode = new Text(text);
    const translations = { [pageLanguage]: text };
    this.querySelectorAll("template").forEach(template => {
      const language = template.getAttribute("lang");
      const text = template.content.textContent.trim();
      translations[language] = text;
    });
    textNodes.set(textNode, translations);
    this.replaceWith(textNode); // vanish!
  }
}

customElements.define("trans-late", TransLate);
```

Yes, that's all! 28 lines of easy-to-understand JavaScript for Houdini's trick are enough here.

But did you notice something?

Ladies and Gentlemen, a new stunt by Houdini: the self-vanishing web component!

I say that because here trans-dash-late *replaces* itself *with* the text.

How can that work?

Well, Houdini's tricks were all secret, but this is open source, so here's how:

Text is *also* a DOM node, so we can store a global reference to it before we remove the web component.

Then, when the page language changes, we just update all the stored nodes, and bingo — no page reloads!

The take-home message is this:

You are in *full* control of your web component's behaviour, how cool is *that*?!!

5. Situation: Server-Side Rendering for the Enterprise

So you discover it's impossible to do Server-Side Rendering in your organization.

You know, that frontend hype thingie where you run JavaScript server-side to pump out web pages.

You can't do that because you are rendering server-side already!

Your super-expensive Enterprise CMS written in Java is the one that pumps out web pages!

And they are composed out of server-side components, available since forever.

So you whine because you work in the enterprise and your boring CV doesn't mention the latest frontend hypes.

You dream:

> If I could only sprinkle a little magic interactivity here and there in the components' HTML, I could escape the impossible situation...
>
> And there's this other hot frontend hype, signals: fine-grained reactive updates for values that change over time. Can I please have that too?

Dream no more, Houdini to the rescue!

Step 1: Wish yourself sig-dash-nal (< `sig-nal`).

A web component which knows how to react and apply updates to a certain place in your HTML.

Step 2: add attributes.

Use `ref` to reference a place by name:

```
<signal ref="inc">+</sig-nal>
```

By default, that place is the parent node of `<sig-nal>` .

Use `new` to do the same but also create a named signal:

```
<signal new="counter">Initial value of counter</sig-nal>
```

Use `type` to convert the initial signal value:

```
<signal new="counter" type="number">3</sig-nal>
```

Use `hydrate` to express your interactivity:

```
<sig-nal hydrate>
      { /* JavaScript object */
      }
</sig-nal>
```

That JavaScript object will pair names of DOM nodes with reactions.

You might react to user input, for example clicks, or when a signal value is updated.

We see an example right now, in

Step 3: compose

```
    <template shadowrootmode="open">
      <link rel="stylesheet" href="./counterdemo.css"/>
      <h3>Interactive island</h3>
      <p>The server supplied the initial value <b>3.</b></p>
      <div>
        <button @click="counter"><sig-nal ref="dec">-</sig-nal></button>
        <div .textContent="counter"><sig-nal new="counter" type="number">3</s
        <button @click="counter"><sig-nal ref="inc">+</sig-nal></button>
        <sig-nal hydrate>
          {
          counter: { ".textContent": rerender },
          inc: { "@click": ({ signal }) => signal.value++ },
          dec: { "@click": ({ signal }) => signal.value-- }
          }
        </sig-nal>
      </div>
    </template>
```

That reads as follows:

when the counter value updates, update the text content of that `<div>`. Initially it's 3.

When you click on the plus-labelled button, increment the signal named counter. 3 becomes 4, you click again, it becomes 5, you get the idea.

Likewise, when hitting that minus-labelled button, you decrement the counter signal. 3 becomes 2, and so on.

It's this short, but it will work without any compiler — I promise!

Step 4: do-it-yourself, define sig-nal in JavaScript. Under 1 KB compressed, that's all you need. I open-sourced it here.

This is the last example, this is Houdini's trick at its best.

It's pure no-build goodness.

And it gives you that awesome Lighthouse-100 performance rating.

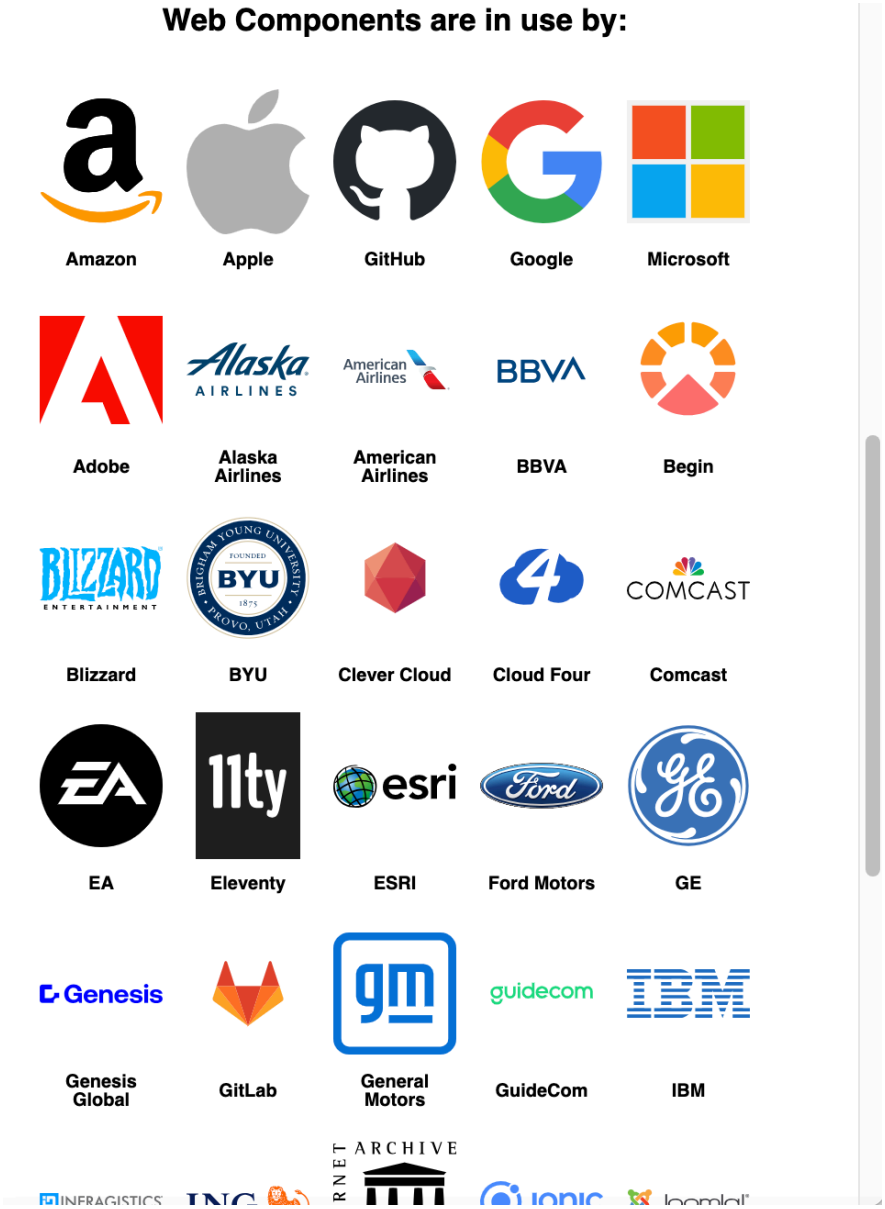
Let me demo it real quick.

Haters gonna hate, but Enterprise knows better

I am not living under a rock.

Of course I know about blog posts like that famous one from the creator of Svelte, Rich Harris: Why I don't use web components.

But it seems Enterprise has overwhelmingly voted with their feet, as arewebcomponentsathingyet.com shows:



So haters gonna hate, but they can't touch Houdini.

Happy escaping!

Let me summarize real quick.

I work in Enterprise frontend, that's where the jobs are.

And I discovered Houdini's magic trick:

There's an ~~app~~ web component for that!

I am happy again because I can help myself in every impossible situation.

Yes, I might not be able to change much around me, but I can *always* can bend HTML to my will.

That's what web components do for me, that's why I recommend them to *you*:

they give you a principled way to escape.

So, happy escaping to all of you, and thanks for listening!