Hypermedia In Action

8. Client Side Scripting

This chapter covers

- How scripting can be effectively added to a Hypermedia Driven Application
- Adding a three-dot menu in our contacts table
- Adding a toolbar for bulk actions that appears when selecting contacts
- Adding a keyboard shortcut for focusing the search input

Scripting in Hypermedia-Driven Applications

> REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.
>
> — Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures

Thus far we have avoided writing any JavaScript for Contact.app, mainly because the functionality we implemented so far does not need it. Contrary to popular belief, hypermedia is not just for "documents" (where a document is considered essentially different to an "app"), and it has many affordances for building interactive experiences. We want to show that it is possible to build sophisticated web applications using the original model of the web without the abstractions provided by JavaScript frameworks. On the other hand, htmx itself is written in JavaScript, and we don’t want our message to be interpreted as "JavaScript bad", or, more generally, "Client-side scripting bad."

The question isn’t "Should we be scripting for the web?" but rather "How should we be scripting for the web?"

Scripting has been a massive multiplier of the Web’s capabilities. Through its use, Web application authors are not only able to enhance their hypertext-based websites, but also create full-fledged client-side applications that can compete with native apps in how they work (although they don’t always win when they do). In other terms, the Web became a distribution medium for non-REST apps in addition to being a RESTful system. When it’s not used as a replacement for the RESTful architecture provided by the Web, however,

scripting is extremely useful in Hypermedia Driven Applications.

You are scripting in a way compatible with HDAs if:

- The main data format exchanged between server of client is hypermedia, the same as it would be in an application with no scripting.
- Client-side state (other than the DOM) is minimized.

This style of scripting requires us to adopt different practices than what is typically recommended for JavaScript, as the most common advice often comes, naturally, from SPA or server-side backgrounds. We will see these new practices in action in the upcoming chapter.

Simply listing "best practices", however, is rarely convincing or edifying (and, frankly, it is often boring). So, we instead will frame them around shiny tools that work well for scripting in a HDA. We will use each of these tools to add a feature to ContactApp:

- An overflow menu to hold the *Edit*, *View* and *Delete* actions, to clean up visual clutter in our list of contacts
- Reordering contacts by dragging and dropping
- A dialog to confirm the deletion of contacts
- A keyboard shortcut for focusing the search box

The important idea in the implementation of each of these features is that they are implemented entirely client-side and yet they don’t exchange information with the server using, for example, JSON. This constraint is what will keep the features within the bounds of a proper Hypermedia Driven Application.

8.1. Scripting tools for the Web

The primary scripting language for the web is, of course, JavaScript, which is ubiquitous in web development today. A bit of interesting internet lore, however, is that JavaScript was not always the only built-in option. As the quote from Roy Fielding above indicates, *applets* written in other languages such as Java were considered part of the scripting infrastructure of the web. In addition, there was a brief period when Internet Explorer supported VBScript, a scripting language based on Visual Basic.

Today, we have a variety of *transcompilers* (often shortened to *transpilers*) that convert another language to JavaScript, such as TypeScript, Dart, Kotlin, ClojureScript, F#. There is also the WebAssembly bytecode format, which is supported as a compilation target for C, Rust, and the WASM-first language AssemblyScript. However, most of these are not geared towards an HDA-compatible style of scripting --- compile-to-JS languages are often paired with SPA-oriented libraries (Dart and AngularDart, ClojureScript and Reagent, F# and Elmish), and WASM is currently mainly geared toward linking to C/C++ libraries from JavaScript.

We bring this up because we are going to look at three different mechanisms for adding scripting to our Hypermedia Driven Application:

- Vanilla JS, that is, using JavaScript without depending on any framework.
- Alpine.js, a JavaScript library for adding behavior directly in HTML.
- _hyperscript, a non-JavaScript scripting language created alongside htmx. Like AlpineJS, it is usually embedded in HTML.

Let's take a quick look at each of these scripting options, so we know what we are dealing with. As with CSS, we are not going to deep dive into any of these options: we are going to show just enough to give you a flavor of each and, we hope, spark your interest in looking into each of them more extensively.

8.2. Vanilla JavaScript

> No code is faster than no code.
>
> — Merb

Vanilla JavaScript is simply using JavaScript in your application without any intermediate layers. The term came into vogue as a play on the fact that there were so many ".js" frameworks out there to help you write JavaScript. As JavaScript matured as a scripting language, standardized across browsers and provided more and more functionality, the utility of many of these frameworks and libraries has diminished. (At the same time, however, SPAs have become more popular, requiring more elaborate JavaScript frameworks).

A quote from the humorous website http://vanilla-js.com captures the situation well:

> Vanilla JS is the lowest-overhead, most comprehensive framework I've ever used.
>
> — http://vanilla-js.com

The message of *VanillaJS* here is that since the browser already has JavaScript baked into it, there isn't any need to download a framework for your application to function. This is true more often than we might like to admit, and is especially the case in HDAs, since hypermedia obviates many features provided by JavaScript frameworks:

- Client-side routing
- An abstraction over DOM manipulation, i.e.: templates that automatically update when referenced variables change
- Server side rendering (rendering here refers to HTML generation)
- Attaching dynamic behavior to server-rendered tags on load
- Network requests

Installation of VanillaJS couldn't be easier: you don't have to. You can just start writing JavaScript in your web application, and it will simply work.

That's the good news. The bad news is that, despite improvements over the last decade, JavaScript has some significant limitations as a scripting language that often make it less than ideal as a stand-alone scripting technology for Hypermedia Driven Applications:

- It is a relatively complex language that has accreted a lot of features and warts.
- JavaScript's asynchrony model involves *colored functions*, a concept described in Robert Nystrom's oft-cited *What Color is Your Function?* [1]
- It is surprisingly clunky to work with events.
- DOM APIs (a large portion of which were originally designed for Java) are verbose and do not make common functionality easy to use.

None of these are deal-breakers, of course, and many people prefer the "close to the metal" (for lack of a better term) nature of vanilla JavaScript to more elaborate client-side scripting approaches.

To dive into Vanilla JavaScript as a front end scripting option, let's write a simple counter [2]. It will have a number and a button that increments the number. Nothing too elaborate,

but it will give you the flavor of each of the three scripting approaches we are going to use in this chapter.

A problem with tackling this problem in Vanilla JavaScript is that it lacks something most JavaScript frameworks provide: a standardized code style. This is not an insurmountable issue, and in fact, it presents a great opportunity to take a small journey through various styles. For our counter, we will start with the simplest thing possible.

Listing 8. 1. Counter in vanilla JavaScript, inline version

```
<section class="counter">
  <output id="my-output">0</output> ❶
  <button
    onclick=" ❷
      document.querySelector('#my-output') ❸
        .textContent++ ❹
    "
  >Increment</button>
</section>
```

❶ Our output element has an ID to help us find it

❷ We use the `onclick` attribute, a brittle but quick way to add an event listener

❸ Find the output

❹ JavaScript lets us use the `++` operator on a string because it loves us

So, not too bad. It's a little annoying that we needed to add an `id` to the span to make this work and `document.querySelector` is a bit verbose compared to, say, `$` (if you are familiar with jQuery) but (but!) it works, and it doesn't require any other JavaScript libraries.

So that's the simple, inline approach. A more standard way to write this code, however, would be to move it into a separate JavaScript file, either linked via a `<script src>` tag or placed into an inline `<script>` by a build process:

Counter in vanilla JavaScript, in multiple files

```
<section class="counter">
  <output id="my-output">0</output>
  <button class="increment-btn">Increment</button>
</section>
```

```
const counterOutput = document.querySelector("#my-output") ❶
const incrementBtn  = document.querySelector(".counter .increment-btn") ❷

incrementBtn.addEventListener("click", e => { ❸
  counterOutput.innerHTML++ ❹
})
```

❶ Find the output element

❷ and the button

❸ We use `addEventListener`, which is preferable to `onclick` for many reasons

❹ The logic stays the same, only the structure around it changes

The design principle motivating separating your JavaScript out to another file is known as *Separation of Concerns (SoC)*. The idea is that the various "concerns" of a software project should be divided up so they don't "pollute" one another. Scripting isn't markup, so it should be *elsewhere*. Styling, similarly, isn't markup, and so it belongs in a separate file as well (A CSS file, for example.)

A goal of separating concerns is that we should be able to modify and evolve one concern independently, with confidence that we won't break any of the other concerns. But, on consideration, is this really the case with HTML and JavaScript?

Did you notice that the HTML in the SoC example is not simply the previous example with the onclick attribute removed? Can you spot the difference?

It turns out that we've had to add a class to the button, so that we could look it up in JavaScript and add in an even handler. In both the HTML and the JavaScript, this class is a string literal and is not subject to name resolution (the process, in compilers and interpreters, of linking names to what they reference).

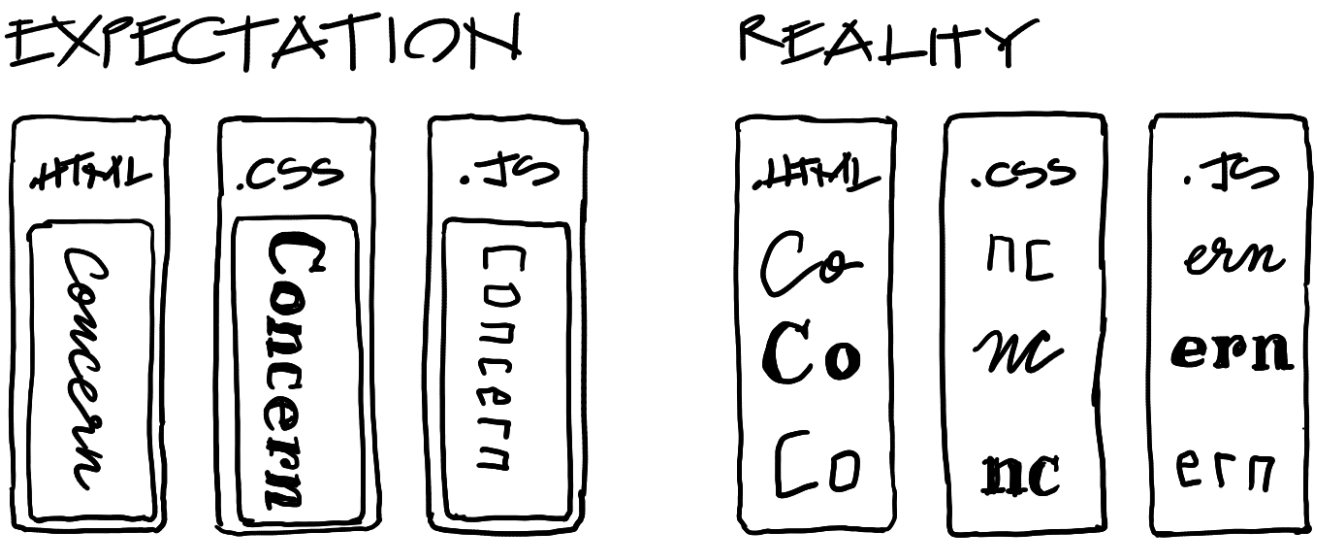
Unfortunately, careless use of CSS selectors in JavaScript can end up causing *jQuery soup*, where:

- The JavaScript that attaches a behavior to a given element is difficult to find.
- Code reuse is difficult.
- The code ends up disorganized (if we have many components, how do we separate them into files? Should we separate them at all?)

The name "jQuery Soup" comes from the fact that early JavaScript-heavy applications were often built in jQuery and ended up with many of these event handlers scattered about in an unstructured and difficult to understand mess.

But things get worse! Imagine that we want to change the number field from an `<output>` tag to an `<input type="number">`. This small change to our HTML will break our JavaScript! This, despite the fact we have "separated" our concerns!

The fix for this issue is trivial (change `.textContent` to `.value`), but it's not hard to see how the burden of synchronizing markup changes and code changes across multiple files might increase in larger components or across a whole page.



The tight coupling between files in this simple example suggests that separation between HTML and JavaScript (and CSS) is often an illusory separation of concerns. In Contact.app we are not *concerned* with "structure", "styling" or "behavior", rather we are concerned with collecting contact info and presenting it to users. We, personally, just don't see a big advantage to this design principle.

And, it turns out that we are not alone in thinking that separation of concerns has been

oversold. Consider the following technologies:

- JSX
- LitHTML
- CSS-in-JS
- Single-File Components
- Filesystem based routing

All of these technologies *colocate* code in various languages that address a single *feature* (usually, a UI widget). In order to use them effectively, we need to understand the problem domain and identify business concerns *in addition* to implementation concerns. Separating technical detail concerns isn't as much of a, ahem, concern.

8.2.1. Locality of Behavior

Locality of Behavior (LoB) is an alternative software design principle that we coined, in opposition to Separation of Concerns. It describes the following characteristic of a piece of software:

> The behaviour of a unit of code should be as obvious as possible by looking only at that unit of code.
>
> — https://htmx.org/essays/locality-of-behaviour/

In simple terms: you should be able to tell what a button does by simply looking at the code or markup that creates that button. This does not mean you need to inline the entire implementation, but that you shouldn't need to hunt for it or require prior knowledge of the codebase to find it.

We will demonstrate Locality of Behavior in all of our examples, both the counter demos and the features we add to ContactApp. Locality of behavior is an explicit design goal of both _hyperscript and Alpine.js (which we will cover later) as well as htmx. All of these tools achieve Locality of Behavior by having you embed attributes directly within your HTML, as opposed to having code look up elements in a document through CSS selectors in order to add event listeners onto them.

The `addEventListener` method in JavaScript is, in a way, a sort of "monkey-patching". It functions in much same way for event listeners as the Ruby programming language's

`define_method` functions for methods in that language:

Listing 8. 2. `define_method` **in Ruby**

```
button.define_method(:click, ->{ ❶
  count += 1 ❷
})
```

❶ When a `click` method call is received,

❷ Do this

Listing 8. 3. `addEventListener` **in JavaScript**

```
button.addEventListener('click', () => { ❶
  count++ ❷
})
```

❶ When a `click` event is received,

❷ Do this

*Note that this ruby code is deliberately unidiomatic to make it easier to understand for non-Rubyists).

Interestingly, "monkey-patching" actually used to be the default way of adding methods to things in JavaScript, by modifying a functions `prototype`. This is a long conversation and beyond the scope of this book, but after proper classes were added to JavaScript in ES2015, modifying the `prototype` of a function has been increasingly discouraged and "monkey patching" has become less and less common. No such advancement has been made for *event listeners*, however, leaving us stuck with `addEventListener`.

```
'use strict'; ❶
(function () {
  Button.prototype.click = function () {
    count++;
  }
})();
```

❶ Feeling nostalgic yet?

This is a shame, because, particularly in the case of front end scripting in a Hypermedia

Driven Application, Locality of Behavior is often far more important than Separation of Concerns.

> **2 > 1 > 2**
>
> Having two decoupled modules is better than having one big blob, but two tightly-coupled modules is worse than either.
>
> (Of course, having no code at all is the best, so 0 > 2 > 1 > 2.)

So, should we go back to the `onclick` attribute way of doing things? It certainly wins in Locality of Behavior, and is baked into HTML. Unfortunately, however, the `on*` JavaScript attributes have some pretty severe drawbacks:

- They don't support custom events.
- There is no good mechanism for associating long-lasting variables with an element --- all variables are discarded when an event listener completes executing.
- If you have multiple instances of an element, you will need to repeat the listener code on each, or use something more clever like event delegation.
- JavaScript code that directly manipulates the DOM gets verbose, and clutters the markup.
- An element cannot listen for events on another element. For example, if you want to dismiss a popup by clicking outside it, the listener will need to be on the body element. The body element will need to have listeners that deal with many unrelated components, some of which may not even be on the page if it was generated from a common template.

JavaScript and Locality of Behavior don't seem to mesh as well as we want them to, but the situation is not hopeless. it's important to be aware that LoB does not require behavior to be *defined* at the use site, but merely invoked there. Keeping this in mind, it's possible to improve LoB while writing JS in a separate file, provided we have a reasonable system for structuring our JavaScript.

8.2.2. RSJS

RSJS ("Reasonable System for JavaScript Structure", https://ricostacruz.com/rsjs/) is a set of guidelines for JavaScript architecture targeted at "a typical non-SPA website". RSJS is a

solution to the lack of a standard code style we mentioned earlier.

We won't replicate all of the guidelines here, but here are the ones most relevant to this book:

- "Use `data-` attributes" --- invoking behavior via adding data attributes makes it obvious there is JavaScript happening, as opposed to random classes or IDs that may be mistakenly removed or changed
- "One component per file" --- the name of the file should match the data attribute so that it can be found easily, a win for LoB

Counter in vanilla JavaScript, with RSJS

```
<section class="counter" data-counter> ❶
  <output id="my-output" data-counter-output>0</output> ❷
  <button class="increment-btn" data-counter-increment>Increment</button>
</section>
```

❶ Invoke a JavaScript behavior with a data attribute

❷ Mark relevant child elements

```
// counter.js ❶
document.querySelectorAll("[data-counter]") ❷
  .forEach(el => {
    const output = el.querySelector("[data-counter-output]"),
      increment = el.querySelector("[data-counter-increment]") ❸

    increment.addEventListener("click", e => output.textContent++) ❹
  })
```

❶ File should have the same name as the data attribute, so that we can locate it easily

❷ Get all elements that invoke this behavior

❸ Get any child elements we need

❹ Register event handlers

This methodology solves (or at least alleviates) many of our gripes with the previous example of vanilla JS in a separate file:

- The JS that attaches behavior to a given element is **clear** (though only through naming conventions).
- Reuse is **easy** --- you can create another counter on the page and it will just work.
- The code is **well-organized** --- one behavior per file

You may remember the problem we discussed about replacing the output tag with `<input type="number">`. That problem still remains. There is a way to solve it, but it's a bit convoluted:

Counter with vanilla JavaScript, with extra-flexible RSJS

```
<section class="counter" data-counter>
  <output id="my-output" data-counter-output="innerHTML">0</output> ❶
  <button class="increment-btn" data-counter-increment>Increment</button>
</section>
```

❶ Specify the property to put the value into

```
// counter.js
document.querySelectorAll("[data-counter]").forEach(el => {
  const output = el.querySelector("[data-counter-output]"),
    increment = el.querySelector("[data-counter-increment]")

  const outProp = output.dataset.counterOutput ❶

  increment.addEventListener("click", e => output[outProp]++) ❷
})
```

❶ Get the attribute's value

❷ Dynamically access the property to increment

If we wanted to use an input, we would change the value of `data-counter-output` to `"value"`. This would also work with `<input type="range">`!

On one hand, this is a way overengineered the solution to the problem. How often do we need to reuse a counter?

On the other, let's think about where else we could go with this. With very little work, we could let the button markup specify the increment amount --- we could go 5-at-a-time, or

decrement (increment by -1). It might be a little more puzzling to support multiple increment buttons with varying amounts if you aren't familiar with this kind of programming, but not insurmountable. As you continue hacking on this counter example, you could end up building a DSL for smart number inputs. The decoupling that is forced on us by putting our JavaScript in a separate file can lead us to invention; restriction breeds creativity.

That's enough fun, however, let's get to work on ContactApp.

Event delegation

Event delegation is a technique that makes use of bubbling in DOM events both as a form of code organization and to reduce memory usage, in situations where a large number of elements need to respond to an event in the same way. Instead of attaching event listeners to each individual element, we attach a single listener to a shared parent element. The parent listener determines which element the event arrived through.

The following is how event delegation would be usually implemented:

Listing 8. 4. With event delegation

```
ul.addEventListener('click', e => {
    const li = e.target.closest('li')
    if (!li) return

    doThingWith(li)
})
```

whereas the alternative would be:

Listing 8. 5. Without event delegation

```
ul.querySelector('li').forEach(li => {
  li.addEventListener('click', e => {
    doThingWith(li)
  })
})
```

Benefits of event delegation

- If elements are dynamically added, there is no need to add the event listener onto them (this usually requires extracting the listener to a named function, and code repeated in every place where events are added). Event delegation can simplify code quite a lot.
- Having only one event listener reduces memory use.
- When code is inline in HTML, event delegation protects us from repetition.

Drawbacks of event delegation

- The listener will execute for every click in a subtree (or other event type) when not all may be relevant.
- The listener will stay around even if no relevant elements remain.

8.2.3. Vanilla JS in action: an overflow menu

Let's sketch the markup we want for our overflow menu:

```
<div data-menu="closed"> ❶
  <button data-menu-button>Options</button> ❷
  <div data-menu-items hidden> ❸
    <a data-menu-item href="/contacts/{{ contact.id }}/edit">Edit</a> ❹
    <a data-menu-item href="/contacts/{{ contact.id }}">View</a>
  </div>
</div>
```

❶ Mark the root element of the menu. We'll reuse this attribute to store the open state of the menu.

❷ We'll hook this button up to open and close our menu.

❸ This is a container for our menu items. We add the `hidden` attribute to avoid the menu items flashing as JS loads.

❹ We mark menu items such that we can implement moving between them with arrow keys.

This is all of the HTML we'll write. The rest of our work will be in JS, making these elements act like a menu.

We start by adding ARIA attributes:

```
import nanoid from "https://unpkg.com/nanoid@4.0.0/non-secure/index.js";

document.querySelectorAll("[data-menu]").forEach(menu => { ❶
  const ❷
  button = menu.querySelector("[data-menu-button]"),
  body = menu.querySelector("[data-menu-items]"),
  items = body.querySelectorAll("[data-menu-item]");

  const isOpen = () => return menu.dataset.menu === "open";

  const bodyId = body.id ?? (body.id = nanoid()); ❸

  button.setAttribute("aria-haspopup", "menu");
  button.setAttribute("aria-controls", "bodyId");

  body.setAttribute("role", "menu");

  items.forEach(item => {
    item.setAttribute("role", "menuitem");
    item.setAttribute("tabindex", "-1"); ❹
  });
})
```

❶ With RSJS, you'll write `querySelectorAll(…).forEach` quite a lot.

❷ Get the descendants.

❸ In order to use `aria-controls`, we need the menu body to have an ID. If it doesn't, we generate one randomly.

❹ Make menu items non-tabbable, so we can manage their focus ourselves.

This is based on the Menu Button example from the cite:[ARIA Authoring Practices Guide]. We haven't made the menu work yet, so these attributes are wrong for now.

HTML ID Soup

Some features of HTML such as ARIA require you to assign unique IDs to elements. When pages are generated from templates dynamically, avoiding name conflicts in large apps can be difficult, as HTML IDs are not scoped the way identifiers in programming languages are.

Randomized IDs with a tool like https://npmjs.com/nanoid can let you avoid the issue, but they also make templates more complex and .

Let's implement toggling the menu:

```
// ...
  items.forEach(item => item.setAttribute("role", "menuitem"));

  function toggleMenu(open = !isOpen()) { ❶
    if (open) {
      menu.dataset.menu = "open"
      body.hidden = false
      button.setAttribute("aria-expanded", "true")
      items[0].focus() ❷
    } else {
      menu.dataset.menu = "closed"
      body.hidden = true
      button.setAttribute("aria-expanded", "false")
    }
  }

  toggleMenu(isOpen()) ❸
  button.addEventListener("click", () => toggleMenu()) ❹
})
```

❶ Optional parameter to specify desired state. This allows us to use one function to open, close, or toggle the menu.

❷ Focus first item of menu when opened.

❸ Call `toggleMenu` with current state, to initialize element attributes.

❹ Toggle menu when button is clicked.

Let's also make the menu close when we click outside it:

```
// ...
  button.addEventListener("click", () => toggleMenu())

  window.addEventListener("click", function clickAway() {
    if (!menu.isConnected) window.removeEventListener("click", clickAway); ❶
    if (menu.contains(event.target)) return; ❷
    toggleMenu(false); ❸
  })
})
```

❶ Clean up event listener if menu has been removed

❷ If the click is inside the menu, do not do anything

❸ Close the menu

You should be able to open, close, and dismiss the menu now, and may be tempted to ship this code to production. Don't! We're not done yet because our menu fails many requirements for menu interactions:

- You can't navigate between menu items using arrow keys
- You can't activate a menu item with the Space key

These factors make our menu annoying and possibly unusable for many people. Let's fix it with the guidance of the venerable cite:[ARIA Authoring Practices Guide]:

```
// ...
    toggleMenu(false); ❸
  })

  const currentIndex = () => { ❶
    const idx = items.indexOf(document.activeElement);
    if (idx === -1) return 0;
    return idx;
  }

  menu.addEventListener("keydown", e => {
    if (e.key === "ArrowUp") {
      items[currentIndex() - 1]?.focus(); ❷

    } else if (e.key === "ArrowDown") {
      items[currentIndex() + 1]?.focus(); ❸

    } else if (e.key === "Space") {
      items[currentIndex()].click(); ❹

    } else if (e.key === "Home") {
      items[0].focus(); ❺

    } else if (e.key === "End") {
      items[items.length - 1].focus(); ❻

    } else if (e.key === "Escape") {
      toggleMenu(false); ❼
      button.focus(); ❽
    }
  })
})
```

❶ Helper: Get the index in the items array of the currently focused menu item (0 if none).

❷ Move focus to the previous menu item when the up arrow key is pressed

❸ Move focus to the next menu item when the down arrow key is pressed

❹ Activate the currently focused element when the space key is pressed

❺ Move focus to the first menu item when Home is pressed

❻ Move focus to the last menu item when End is pressed

❼ Close menu when Escape is pressed

❽ Return focus to menu button when closing menu

I'm pretty sure that covers all our bases. That's a lot of code! But it's code that encodes a lot of behavior.

Though, we still don't support submenus, or menu items being added or removed dynamically. If we need more features, it might make more sense to use an off-the-shelf library --- for instance, GitHub's `details-menu-element`.

8.3. Alpine.js

Alpine.js (https://alpinejs.dev) is a relatively new JavaScript library that allows you to embed your code directly in HTML. It bills itself as a modern replacement for jQuery, a widely used but quite old JavaScript library, and it lives up to that promise.

Installing AlpineJS is a breeze, you can simply include it via a CDN:

Listing 8. 6. Installing AlpineJS

```
<script src="https://unpkg.com/alpinejs"></script>
```

You can also install it from npm, or vendor it from your own server.

The main interface of Alpine is a set of HTML attributes, the main one of which is `x-data`. The content of `x-data` is a JavaScript expression which evaluates to an object, whose properties we can access in the element. For our counter, the only state is the current number, so let's create an object with one property:

Listing 8. 7. Counter with Alpine, line 1

```
<div class="counter" x-data="{ count: 0 }">
```

We've defined our state, let's actually use it:

Listing 8. 8. Counter with Alpine, lines 1-2

```
<div class="counter" x-data="{ count: 0 }">
  <output x-text="count"></output> ❶
```

❶ The `x-text` attribute.

This attribute sets the text content of an element to a given expression. Notice that we can access the data of a parent element.

To attach event listeners, we use `x-on`:

Listing 8. 9. Counter with Alpine, the full thing

```
<div class="counter" x-data="{ count: 0 }">
  <output x-text="count"></output>

  <button x-on:click="count++">Increment</button> ❶
</div>
```

❶ With `x-on`, we specify the attribute in the attribute *name*.

Would you look at that, we're done already! (It's almost as though we wrote a trivial example). What we created is, incidentally, nearly identical to the second code example in Alpine's documentation --- available at https://alpinejs.dev/start-here.

8.3.1. `x-on:click` *vs.* `onclick`

The `x-on:click` attribute (or its shorthand `@click`) differs from the browser built-in `onclick` attribute in significant ways that make it much more useful:

- You can listen for events from other elements. For example, the `.outside` modifier lets you listen to any click event that is **not** within the element.
- You can use other modifiers to
  - throttle or debounce event listeners,
  - ignore events that are bubbled up from descendant elements, or
  - attach passive listeners.
- You can listen to custom events, such as those dispatched by htmx.

8.3.2. Reactivity and templating

As you can see, this code is much tighter than the VanillaJS implementation. It helps that AlpineJS supports a notion of variables, allowing you to bind the visibility of the `span` element to a variable that both it and the button can access. Alpine allows for much more

elaborate data bindings as well, it is an excellent general purpose client-side scripting library.

8.3.3. Alpine.js in action: A bulk action toolbar

Right now, Contact.app has a "Delete Selected Contacts" button at the very bottom of the page. This button has a long name, is not easy to find and the UI pattern is not conducive to adding more bulk actions as they would clutter the page. In this section, we'll replace this with a toolbar that only appears when the user starts selecting contacts. It will show how many contacts are selected and let you select all contacts in one go.

Let's lay down some markup:

```
<template
  x-data="{ selected: [] }" ❶
  x-if="selected.length > 0"> ❷
  <div class="box info tool-bar">
    <slot x-text="selected.length"></slot>
    contacts selected

    <button type="button" class="bad bg color border">Delete</button>
    <hr aria-orientation="vertical">
    <button type="button">Cancel</button>
  </div>
</template>
```

❶ Our data: the set of selected contacts.

❷ Only show toolbar if at least one item is selected.

We needed to wrap our component in a `<template>` to use the `x-if` attribute. There is an alternative that doesn't require this (`x-show`) which hides an element through styling rather than removing it from the DOM.

Right now, there are no observable changes to the page, because the `selected` set is always empty. Let's write the code that adds to it:

```
<td><input type="checkbox" name="selected_contact_ids" value="{{ contact.id }}"
      x-model="selected"></td>
```

We used the `x-model` attribute to create a two-way binding between the `selected` array

and the checkboxes. When a checkbox is checked or unchecked, the array will be updated, and mutations we make to the array will similarly be reflected in the checkboxes' state.

Unfortunately, it doesn't work, and we're getting errors about `selected` not being defined. This is because the element on which we defined our data (the `<template>`) is not an ancestor of this checkbox. We need to lift that state up to a common ancestor:

```
<form x-data="{ selected: [] }"> ❶
    <template ...
```

❶ This is the form that was wrapped around the contacts table.

Now, you should see the toolbar appearing when you select a contact --- great! Let's implement the buttons:

The delete button needs to interact with htmx to work:

```
<button type="button" class="bad bg color border"
  @click="confirm(`Delete ${selected.length} contacts?`) && ❶
    htmx.ajax('DELETE', '/contacts', { source: $root, target: document.body })" ❷
>Delete</button>
```

❶ We can use a dynamic confirmation message.

❷ Use the htmx JS API to send a request and swap it into the body from Alpine.

Let's look closer at the call to `htmx.ajax`. `source` is the element from which htmx will collect data to include in the request. We set this to $root, the element which has the x-data attribute --- the form containing all of our contacts. As the `DELETE /contacts` endpoint returns the contents of whole body, we tell htmx to swap it as such.

For the *Cancel* button, our job is quite simple:

```
<button type="button" @click="selected = []">Cancel</button>
```

We set `selected` to an empty array. Our use of x-model means this will be reflected in the actual checkboxes.

With all of this in place, we have a much improved experience for performing bulk actions

on contacts that can be extended with more options without creating bloat in the main interface of our app.

8.4. _hyperscript

While previous two examples are JavaScript-oriented, _hyperscript (https://hyperscript.org, the underscore is part of the name but not pronounced) is a entire new scripting language for front-end development. It has a completely different syntax than JavaScript, derived from an older language called HyperTalk, which was the scripting language of HyperCard, an old hypermedia system, along with IDE and WYSIWYG editor on the Macintosh Computer. The most noticeable thing about _hyperscript is that it resembles English prose more than it does code. It was initially created as a sister project to htmx, to handle events and modify the document in htmx-based applications. Currently, it positions itself as a modern jQuery replacement and alternative to JavaScript.

Like Alpine, _hyperscript allows you to program inline in HTML, but instead of using JavaScript, it has a syntax designed to be embedded into other languages.

What it eschews is a reactive mechanism, instead focusing on making manual DOM manipulation easier. It has built-in constructs for many DOM operations, preventing you from needing to navigate sometimes-verbose APIs.

We will not be doing a deep dive on the language, but again just want to give you a flavor of what scripting in _hyperscript is like, so you can pursue the language in more depth later if you find it interesting.

Like htmx and AlpineJS, _hyperscript can be installed via a CDN or from npm (package name `hyperscript.org`):

Listing 8. 10. Installing _hyperscript via CDN

```
<script src="//unpkg.com/hyperscript.org"></script>
```

Like AlpineJS, in _hyperscript you put attributes directly in your HTML. Unlike AlpineJS, there is only one attribute for _hyperscript: the `_` (underscore) attribute [3]. This is where all the code responsible for an element goes.

```
<div class="counter">
  <output>0</output>
  <button _="on click increment the textContent of the previous <output
/>">Increment</button> ❶
</div>
```

❶ This is what _hyperscript looks like, believe it or not!

Seasoned JavaScript programmers are often suspicious of _hyperscript: There have been many "natural language programming" projects that usually target non-programmers and beginner programmers, assuming that being able to read code will give you the ability to write it as well. (The authors' views on the usefulness of natural language for teaching programming are nuanced and out of scope for this book). It should be noted that _hyperscript is openly a programming language, in fact, its syntax is inspired in many places by the speech patterns of web developers. In addition, _hyperscript's readability is achieved not through complex heuristics or NLP, but common parsing tricks and a culture of readability.

As you can see in the above example, _hyperscript does not shy away from using punctuation when appropriate. We'll come across quite a lot of new syntax we use as we go. To get our feet wet, here's an annotated version of the script above:

```
on click -- Event listener
  increment -- This command (built into the language) increments things
    the -- "the" is ignored
    textContent of -- "b of a" and "a's b" are alternative forms of "a.b"
    the previous -- "previous x" == element before me in the DOM that matches x
    <output /> -- A CSS selector is wrapped between "<" and "/>"
```

The `previous` keyword (and the accompanying `next`) are an example of how _hyperscript makes DOM operations easier. As an exercise, you can try to implement a function `previous(selector: string): Node` that does the same.

8.4.1. _hyperscript in action: a keyboard shortcut

Since our keyboard shortcut focuses a search input, let's put the code on that search input. Here it is:

```
<input id="search" name="q" type="search" placeholder="Search Contacts">
```

We begin with an event listener, which, as we explained, starts with `on`:

```
<input id="search" name="q" type="search" placeholder="Search Contacts"
  _="on keydown [shiftKey and code is 'KeyS'] ❶❷❸❹
       -- ...">
```

❶ The square bracket notation is *event filtering* --- any event for which the expression inside the brackets is falsey will be ignored by this listener.

❷ Inside the event filter, properties of the event can be directly accessed.

❸ `and` is `&&` in JavaScript.

❹ `is` is `==` in JavaScript.

We are using event filtering to listen to only the events we are interested in, i.e. the user pressing kbd:[Shift+S]. There is a problem, however: Keyboard events will only be sent to this input element if it is already focused. We need to attach the listener to the whole window instead. No problem:

```
<input id="search" name="q" type="search" placeholder="Search Contacts"
  _="on keydown [shiftKey and code is 'KeyS'] from the window ❶
       -- ...">
```

❶ "from" is part of the "on" feature and lets us listen to events from other objects.

We can attach the listener to the body while keeping its code on the element it logically relates to. Let's actually focus that element now:

```
<input id="search" name="q" type="search" placeholder="Search Contacts"
  _="on keydown[shiftKey and code is 'KeyS'] from the window
       focus() me"> ❶❷
```

❶ Any method of any object can be used as a command. (This is called a "pseudocommand" in _hyperscript lingo). This line is equivalent to `me.focus()` (which is also valid syntax in _hyperscript).

❷ "me" refers to the element that the script is written on.

There's our code! Surprisingly terse for an English-like programming language, compared to the equivalent JavaScript:

```
const search = document.querySelector("#search")
window.addEventListener("keydown", e => {
  if (e.shiftKey && e.code === "KeyS") search.focus();
})
```

8.4.2. Why a new programming language?

Being an interpreter written in JavaScript, the _hyperscript runtime has a lot of overhead. One might wonder why it isn't implemented as a JavaScript library. A new programming language allows us to provide features and fix warts in a way that wouldn't be possible otherwise:

Async transparency

In _hyperscript, asynchronous functions (i.e. functions that return `Promise` instances) can be invoked as if they were synchronous. Changing a function from sync to async does not break any _hyperscript code that calls it. This is achieved by checking for a Promise when evaluating any expression, and suspending the running script if one exists (only the current event handler is suspended and the main thread is not blocked). JavaScript does not allow us to hook into expression evaluation at the level of granularity needed to achieve this.

Array property access

In _hyperscript, accessing a property on an array (other than `length` or a number) will return an array of the values of property on each member of that array --- in other terms, `a.name` is equivalent to `a.map(el ⍰ el.name)`. jQuery has a similar feature, but only for its own data structure.

8.4.3. Reusable behavior in _hyperscript

The main mechanism for reuse in _hyperscript is _behaviors_ --- named collections of *features* (event listeners, function definitions etc.) that can be *installed* as follows:

```
<div _="install ToggleableMenu(button: .menu-button in me, menu: #contents)"> ❶
  <button class="menu-button">Options</button>
  <div id="contents">
```

❶ Behaviors can accept arguments.

A nice aspect of _hyperscript behaviors is that any element's script can be refactored into a reusable behavior on a copy-paste basis:

Listing 8. 11. The search bar keyboard shortcut code, extracted into a behavior

```
behavior SearchShortcut
  on keydown[shiftKey and code is 'KeyS'] from the window
    focus() me
  end
end
```

Prime examples of behavior usage can be found on Ben Pate's *Hyperscript Widgets* collection (https://github.com/benpate/hyperscript-widgets). Reproduced here with minor cleanup is a rich text editor implemented in 68 lines:

Listing 8. 12. wysiwyg._hs

```
behavior wysiwyg(name)

    -- WYSIWYG setup
    init
        -- save links to important DOM nodes
        set :form to closest <form />
        set :input to form.elements[name]
        set :editor to first .wysiwyg-editor in me

        -- configure related DOM nodes
        add [@tabIndex=0] to :editor
        add [@contentEditable=true] to :editor

        tell <button/> in me
            add [@type="button"]
        end

    -- Clicking a toolbar button triggers a command on the content
    on click(target)
        if target's @data-command is null then
```

```
            set target to the closest <[data-command]/> to target
            if target is null then
                exit
            end
        end

        set command to target's @data-command

        -- special handling for inertLink
        if command is "createLink" then
            get prompt("Enter Link URL")
            call document.execCommand(command, false, result)
            exit
        end

        -- fall through to all other commands
        set value to target's @data-command-value
        call document.execCommand(command, false, value)
    end

    -- Show the toolbar when focused
    on focus(target) from the .wysiwyg-editor in me
    remove @hidden from the .wysiwyg-toolbar in me
    end

    -- Hide the toolbar when blured
    on blur from the .wysiwyg-editor in me
        wait 200ms
        if (<:focus/> in me) is empty then
      add [@hidden=true] to the .wysiwyg-toolbar in me
        end
    end

    -- Autosave the WYSIWYG after 15s of inactivity
    on input debounced at 15s
        send updated to form
    end

    -- Autosave the WYSIWYG whenever it loses focus
    on blur from the .wysiwyg-editor in me
        send updated to form
    end

    -- Push the value directly into the XHR request before it's sent.
    on htmx:configRequest(parameters) from closest <form/>
```

```
        set value to the editor's innerHTML
        Object.defineProperty(parameters, name, {value: value, writable:'true'})
    end
```

You can try the editor on https://benpate.github.io/hyperscript-widgets/wysiwyg/.

_hyperscript, being a whole programming language, goes a lot deeper than what was introduced here. Further information is available at https://hyperscript.org/docs.

> In keeping with general htmx principles, we will endeavor to create code that is:
>
> - Usable
> - Accessible
> - Un-Scalable
>
> — https://benpate.github.io/hyperscript-widgets/

8.5. Using off-the-shelf components

8.6. Events and the DOM

One thing that you will notice in all the scripting that we add to Contact.app is the heavy use of *events*. This is not an accident; scripting in a Hypermedia Driven Application should be oriented around events --- mostly listening to DOM events, but also dispatching custom events. Since htmx allows requests to be triggered upon any type of event, custom events provide an excellent bridge between client-side scripts and the hypermedia exchanges that define a RESTful Hypermedia Driven Application.

Another thing you might notice about the scripting examples is that they all mutate the DOM in some way, such as showing or hiding elements, changing the text content of an element, or moving focus. In many cases this change in state isn't synchronized with the server, which may, at first, seem to discredit our aim of using hypermedia is the engine of application state.

Both the use of events and the prevalence of DOM mutations point to the fact that the purpose of scripting in a HDA is to enhance UI interaction. Use of events reflects that we are mainly concerned with responding to the user's actions. DOM mutations make up a large portion of our code because we are concerned with UI as opposed to business logic

---

when we write scripts. The state retained by client-side scripts should be an extension of state retained by the browser (e.g.: the value of an input element before it is submitted), not closely tied to the application domain, and *ephemeral*. Scripts may use tools like localStorage to keep some user-specific data; what they should not do is alter a canonical data store without going through a hypermedia channel. (As a **very rough** rule of thumb, this means that scripts should avoid making non-GET requests to your server).

8.7. Being pragmatic

> In case of conflict, consider users over authors over implementors over specifiers over theoretical purity.
>
> — W3C, HTML Design Principles § 3.2 Priority of Constituencies

The sad truth is that there will never be a general theory of web development. Any guideline, methodology, or rule of thumb will hit degenerate cases. When this happens, there are a few ways to react:

Denial

Why would we want to implement this feature, anyway? Invent reasons why the problem should not be solved.

Anger

Vehemently refuse to abandon your principles and implement the feature with your method, without regard for the consequences.

Bargaining

Try to invent a new theory to accommodate the feature. It will be incoherent.

Depression

Fantasize about leaving the software industry.

Acceptance

Implement the feature the way you always knew it should be. Leave a comment for any future developer who might be compulsed to "refactor" it.

8.8. Summary

Maximize locality of behavior, sometimes at the expense of separation of concerns.

Remember that "concerns" are not the same thing as filetypes.

Avoid using scripts to directly modify system state. Reserve it for UI state.

Use progressive enhancement.

If you’re mostly going to write reusable, generalized components: use vanilla JavaScript with RSJS.

If you’re mostly going to write one-off, specialized components: use Alpine.js or _hyperscript. Alpine supports one- or two-way data binding and lets you use the programming language known by the highest number of people (though with some extensions). _hyperscript offers a concise, readable syntax, especially for DOM operations, and makes async operations easy.

If you need a common UI pattern that isn’t built into HTML: use a library. If you’re going to write such a library yourself, use vanilla JS with RSJS.

Don’t worry much about theoretical purity.

[1] https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

[2] The counter is a common example widget for UI development tools, a trend that seems to have been started by React. İt’s unclear if the "counterexample" pun was intentional.

[3] You can also use a `script` attribute, or `data-script` to please HTML validators.