Hypermedia In Action

8. Client Side Scripting

This chapter covers

- How scripting can be effectively added to a Hypermedia Driven Application
- Adding a javascript-based confirmation dialog for deleting contacts
- Adding a three-dot menu in our contacts table
- Adding a keyboard shortcut for focusing the search input
- Adding support for re-ordering contacts via drag-and-drop

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."

[htmx loves javascript] | *htmx-loves-javascript.png*

So 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 different practices than what is usually recommended for JavaScript, as the most common advice often comes from SPA or server-side backgrounds. We will see these new practices in action in the upcoming chapter.

However, listing "best practices" is rarely convincing or edifying (and often boring). So, we instead 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 entirely client-side and don't exchange information with the server using, for example, JSON. This is what will keep them all 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.

I 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 became more popular, requiring better 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 like to admit and especially so in HDAs as hypermedia obviates many features provided by 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 JavaScript has some 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.
- Its model for asynchrony 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 in the language.
- 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.

As our "hello world" example to showcase each of our scripting options, let's write a 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.

We have a problem, however, as one of the things frameworks provide is still missing: a standardized code style. This is not an insurmountable problem, and in fact a great opportunity to take a small journey through various styles, starting 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, `$` but (but!) it works and it doesn't require any other JavaScript libraries.

A more "standard" way to write the above would be to put the above in a separate file, either linked via a `<script src>` 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 main reason people do this is for the sake of *separation of concerns*.

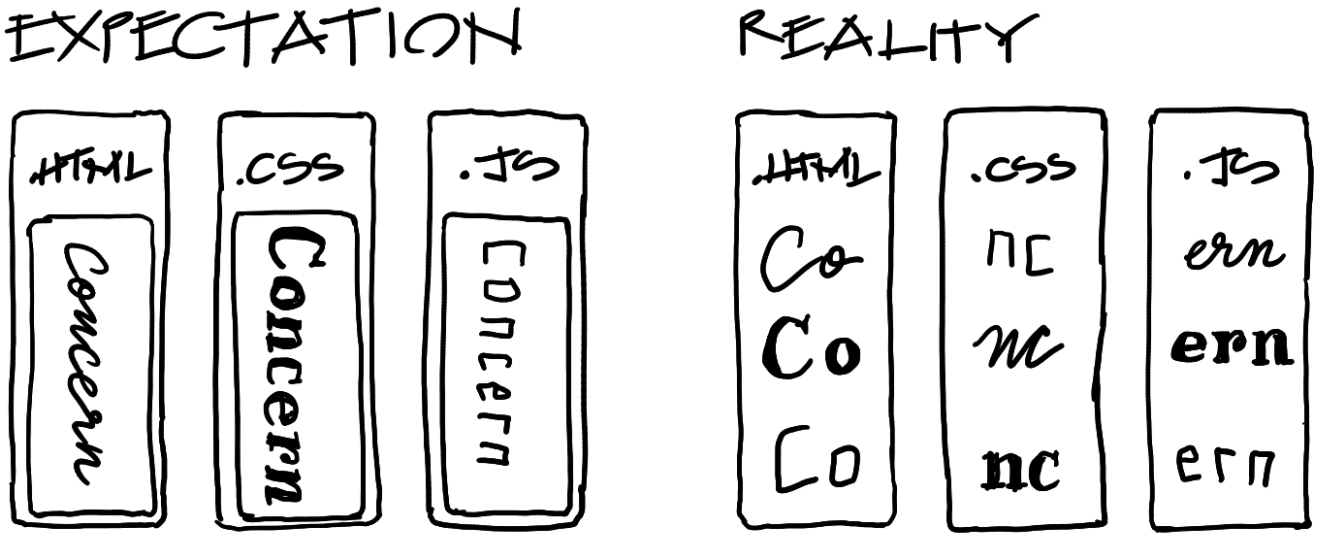
The purpose of separating concerns is that we will be able to modify one with confidence that we won't break any other. Is this really the case with HTML and JS?

Notice that the HTML in the above example is not just the previous example with the onclick attribute removed. Can you spot the difference?

We've had to add a class to the button so that we could find it in JS. In both the HTML and the JS, this class is a string literal not subject to name resolution (the process, in compilers and interpreters, of linking names to what they reference). The careless use of CSS selectors in JavaScript causes *jQuery soup*, where:

- The JS that attaches behavior to a given element is unclear (though developer tools in browsers help with this).
- Reuse is difficult.
- The code is disorganized (if we have many components, how do we separate them into files (if at all?))

Furthermore, imagine that we want to change the number field from an `<output>` tag to an `<input type="number">`. This change to our HTML will break our JavaScript. The fix is trivial (change `.textContent` to `.value`), but it's not hard to see how the burden of synchronizing markup and code across files would 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. Contact.app is is not *concerned* with "structure", "styling" or "behavior", it's concerned with collecting contact info and presenting it.

Our suspicion is validated by developments in the JS framework world:

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

All of these solutions 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 adition to implementation concerns.

8.2.1. Locality of Behavior

Locality of Behavior (LoB) is a software design principle that we coined to describe 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. It is a design goal of both _hyperscript and Alpine.js (which we will cover later) as well as htmx. These tools achieve it through having you embed attributes and directly within your HTML, as opposed to having code pluck elements out of a document through CSS selectors and add event listeners onto them.

The `addEventListener` method is, in a way, monkey-patching. Its functionality is the same for event listeners as ruby's `define_method`:

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

(The Ruby code is deliberately unidiomatic to make it easier to understand for non-Rubyists).

Monkey-patching actually used to be the default way of adding methods in JavaScript. After classes were added in ES2015, however, modifying the `prototype` of a function (which, confusingly, is not the function's prototype but of objects the function returns) is increasingly discouraged. No such advancement has been made for event listeners, however, leaving us stuck with `addEventListener` and `onclick`.

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

❶ Feeling nostalgic yet?

This is a shame, because in the case of front end scripting in a HDA, locality of behavior is often the more important principle over 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` way of doing things? It certainly wins in the Locality of Behavior category. Unfortunately, JavaScript in `on*` attributes are not a great way to program:

- 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 JavaScript in action: A confirmation dialog

Right now, clicking the `Delete` link on a contact instantly deletes it, making it prone to accidents. We'll write some JavaScript to add confirmation dialogs to elements and use it on the delete button.

We'll write the JavaScript first before adding anything to our markup.

Listing 8. 6. Confirmation dialog with Vanilla JS & RSJS

```
document.querySelectorAll("[data-confirm]") ❶
  .forEach(el => {
    // ...
  })
```

❶ Find relevant elements. Our attribute is `data-confirm`, so we'll write this code in a file named `confirm.js`.

We need to show a confirmation dialog. There are libraries that let us show styled, rich alert dialogs, but let's just use `confirm()` for now. Adding in a library later will be a good test of how maintainable our code is.

```
document.querySelectorAll("[data-confirm]")
  .forEach(el => {
    el.addEventListener("...", e => { ❶
      const didConfirm = confirm()
      if (!didConfirm) {
        event.stopImmediatePropagation(); ❷
        event.stopPropagation(); ❸
      }
    })
  })
```

❶ **What event?**

❷ Prevent listeners on this element from running

❸ Prevent listeners on parent elements from running

We need to decide what event we need to listen to:

- Hardcode `"click"`. It's simple and it covers most cases. However, there's not a clear escape hatch if you need a different event.
- Try to sniff what event you need to listen to based on the element. Complex and fragile (but I repeat myself).
- Let the author specify in the attribute. This is what we'll do.

```
el.addEventListener(
  el.dataset.confirm || "click", ❶
  e => {
    // ...
  }
)
```

❶ Specify a default for convenience.

In 9 lines of code, we have a generic confirmation library that we can use for any element as follows. It's definitely overengineered as a result of the forced decoupling, just like the counter earlier, but it works well and was reasonably fun to write.

```
<button type="submit" data-confirm>Delete</button>
<input type="radio" name="volume" value="100" data-confirm="input">
```

Async ruins everything

In the confirmation dialog code we wrote, we use `confirm()`, which is convenient, but displays a barebones dialog that cannot contain rich text. Can we write a similar script using a fancy alert dialog library, like SweetAlert2?

```
document.querySelectorAll("[data-confirm]")
  .forEach(el => {
    el.addEventListener("click", e => {
      const result = await Swal.fire("Are you sure?", "", "question")
      const didConfirm = result.isConfirmed
      if (!didConfirm) {
        event.stopImmediatePropagation();
        event.stopPropagation();
      }
    })
  })
```

```
Uncaught SyntaxError: await is only valid in async functions, async generators
and modules
```

Right. Let's fix that…

```
document.querySelectorAll("[data-confirm]")
  .forEach(el => {
    el.addEventListener("click", async e => {
      const result = await Swal.fire("Are you sure?", "", "question")
      const didConfirm = result.isConfirmed
      if (!didConfirm) {
        event.stopImmediatePropagation();
        event.stopPropagation();
      }
    })
  })
```

No more errors, but this code no longer works. This is because by the time we call `stopPropagation` and `stopImmediatePropagation`, the event has already propagated. We can avoid this when using the built-in `confirm` function because it has the privilege of blocking the main thread.

There is no general solution to this problem.

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. 7. 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. 8. Counter with Alpine, line 1

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

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

Listing 8. 9. 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. 10. 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 in action: an overflow menu

An overflow menu only has one bit of state: whether it is open.

```
<div x-data="{ open: false }"> ❶
  <button>Options</button> ❷
  <div>
    <a href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a href="/contacts/{{ contact.id }}">View</a>
  </div>
</div>
```

❶ Define the initial state

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

While we have only one bit of state, we have many parts that depend on it. This is where *reactivity* shines:

```
<div x-data="{ open: false }">
  <button
    aria-haspopup="menu" ❶
    aria-controls="contents" ❷
    x-bind:aria-expanded="open" ❸
  >Options</button>
  <template x-if="open"> ❹
    <div id="contents"> ❺
      <a href="/contacts/{{ contact.id }}/edit">Edit</a>
      <a href="/contacts/{{ contact.id }}">View</a>
    </div>
  </template>
</div>
```

❶ Declare that this button will cause a menu to open,

❷ and that the menu that this button *controls* is the one with ID `contents`

❸ Indicate the current open state of the menu, using x-bind to reference our data

❹ Only show the menu if it is open

❺ Add an ID to the menu, so that we can reference it in the aria-controls attribute

This is based on the Menu Button example from the cite:[ARIA Authoring Practices Guide]. We haven't made the menu work yet, just the button that opens it.

The use `x-bind` means that as we change the open state, the `aria-expanded` attribute

will update to match. The same holds for the `x-show` on the div with the contents, and indeed for most of Alpine. In order to see this in action, let's actually change that state:

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.

Some developers in the SPA world use randomized IDs with a tool like https://npmjs.com/nanoid to avoid the issue.

```
<div x-data="{ open: false }">
  <button
    aria-haspopup="menu"
    aria-controls="contents"
    x-bind:aria-expanded="open"
    x-on:click="open = !open" ❶
  >Options</button>
  <template x-if="open">
    <div id="contents" x-on:click.outside="open = false"> ❷
      <a href="/contacts/{{ contact.id }}/edit">Edit</a>
      <a href="/contacts/{{ contact.id }}">View</a>
    </div>
  </template>
</div>
```

❶ Toggle the open state when the button is clicked

❷ Dismiss the menu by clicking away

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

- It does not have the `menu` or `menuitem` roles applied properly, which makes life harder for users of assistive software
- 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 even unusable for many people. Let's fix it with the guidance of the venerable cite:[ARIA Authoring Practices Guide]:

```
<div x-data="{ open: false }">
  <button
    aria-haspopup="menu"
    aria-controls="contents"
    x-bind:aria-expanded="open"
    x-on:click="open = !open"
  >Options</button>
  <div role="menu" ❶
    id="contents" x-show="open"
    x-on:click.outside="open = false"
    x-on:keydown.up="document.activeElement.previousElementSibling?.focus()" ❷
    x-on:keydown.down="document.activeElement.nextElementSibling?.focus()" ❸
    x-on:keydown.space="document.activeElement.click()" ❹
    x-effect="if (open) requestAnimationFrame(() => $el.firstElementChild.focus())"
❺❻
    x-on:keydown="$event.key === 'Home'
      ? $el.firstElementChild.focus()
      : $event.key === 'End'
      ? $el.lastElementChild.focus()
      : null" ❼
    >
    <a role="menuitem" ❽
      tabindex="-1" ❾
      href="/contacts/{{ contact.id }}/edit">Edit</a>
    <a role="menuitem" tabindex="-1" href="/contacts/{{ contact.id }}">View</a>
  </div>
</div>
```

❶ Put the `menu` role on the menu root

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

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

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

❺ Access the div itself through the Alpine-supplied `$el` variable

❻ Focus the first item when `show` changes

❼ Handle the remaining cases that Alpine doesn't have modifiers for

❽ Put the `menuitem` role on the individual items

❾ Make the menu items non-tabbable

`x-effect` is a cool attribute that lets you perform side-effects when a piece of element state changes. It automatically detects which state is accessed in the effect. However, it can also complicate our code --- in this example, we need to use `requestAnimationFrame` because otherwise, the effect is executed before the `x-show` attribute reveals the element to focus.

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. Not to mention that we still made some assumptions to make our code shorter:

- All children are menu items with no wrappers, dividers, etc.
- There are no submenus

As we need more features, it might make more sense to use a library --- for instance, GitHub's `details-menu-element`.

8.3.4. Reusable behavior in Alpine

Our menu component has a lot of attributes that will currently be repeated in every item of the table. This is hard to maintain when manually writing HTML and increases payload sizes when generating it via a template. We can rectify this using an nifty feature of the `x-bind` attribute:

> x-bind allows you to bind an object of different directives and attributes to an element.
>
> The object keys can be anything you would normally write as an attribute name in Alpine. This includes Alpine directives and modifiers, but also plain HTML attributes. The object values are either plain strings, or in the case of dynamic Alpine directives, callbacks to be evaluated by Alpine.
>
> — https://alpinejs.dev/directives/bind#bind-directives

It's far easier to understand what this means after seeing the attribute in use. To begin, we create a JavaScript function which will encapsulate all of our menu's behavior:

```
function menu() {
  return {
    role: "menu",
    "x-show"() { ❶
      return this.open; ❷
    },
    "x-on:click.outside"() { this.open = false },
    "x-on:keydown.up"() { document.activeElement.previousElementSibling?.focus() },
    "x-on:keydown.down"() { document.activeElement.nextElementSibling?.focus() },
    "x-on:keydown.space"() { document.activeElement.click() },
    "x-effect"() { if (this.open) this.$el.firstElementChild.focus() },
    "x-on:keydown"(event) { ❸
      if (event.key === 'Home') $el.firstChild.focus()
      else if (event.key === 'End') $el.lastChild.focus()
    },
  }
}
```

❶ JavaScript allows any string literal to be the name of an object member. This even works with classes!

❷ Values that would be globally accessible in an attribute are accessed through `this` in a function.

❸ We can clean up longer functions.

The return value is a map of attribute names to values, with Alpine attributes having functions as values instead of strings of code. We can then reference this function in HTML as follows:

```
<div id="contents" x-bind="menu()">
  <a role="menuitem" tabindex="-1" href="/contacts/{{ contact.id }}/edit">Edit</a>
  <a role="menuitem" tabindex="-1" href="/contacts/{{ contact.id }}">View</a>
</div>
```

This requires the function `menu` to be global. We can avoid that with `Alpine.data`, which is a function to make any data accessible to Alpine expressions:

```
Alpine.data("menu", () => {
  return {
    role: "menu",
    "x-show"() { return this.open; },
    // ...
  }
})
```

Another useful tool in factoring Alpine code is calling functions in `x-data` as follows:

```
Alpine.data("toggleableMenu", () => ({ open: false }))
```

```
<div x-data="toggleableMenu()">
  <button
    aria-haspopup="menu"
    ...>
```

You can combine the two techniques:

```
Alpine.data("toggleableMenu", () => ({
  open: false,
  menuBehavior: { ❶
    role: "menu",
    "x-show"() { return this.open; },
    // ...
  },
  buttonBehavior: { ❷
    "aria-haspopup": "menu",
    "aria-controls": "contents",
    "x-bind:aria-expanded"() { return this.open },
    "x-on:click"() { this.open = !this.open },
  }
}))
```

❶ The object that we bind to the menu has been moved into the data.

❷ We can encapsulate the button's behavior in the same way.

```
<div x-data="toggleableMenu()">
  <button x-bind="buttonBehavior">Options</button> ❶
  <div id="contents" x-bind="menuBehavior"> ❷
    <!-- ... -->
```

❶ Access the button behavior object from the data.

❷ Same for the menu… hey, does this look familiar?

You may notice that the markup for the `x-bind` style quite resembles RSJS. Combined with Alpine's reactivity and concise syntax, it's quite a powerful style for writing localized as well as decoupled code.

Factoring our behavior in this way reduces the locality in our code, as it requires us to locate the `menu` and `toggleableMenu` functions to understand what our code does. You can use named files similarly to RSJS to somewhat alleviate this issue, but it's a tradeoff that needs to be considered.

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. 11. 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. 12. 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. 13. 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

8.5.1. Off-the-shelf components in action: drag to reorder

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), ephemeral, 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

Use progressive enhancement.

Maximize locality of behavior, sometimes at the expense of separation of concerns. Remember that "concerns" are not the same thing as filetypes.

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

If you’re mostly going to write one-off, specialized components: use Alpine.js or _hyperscript.

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.

Alpine lets things auto-update based on changes to state and lets you use the programming language known by the most people.

_hyperscript offers a concise, readable syntax, especially for DOM operations, and makes async operations easy.

Events are cool.

Do not use scripts to directly modify system state. Reserve it for UI state.

[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.