Hypermedia In Action

9. 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 in Contact.app, mainly because the functionality we implemented so far has not required it. We have shown that, contrary to what some people content, hypermedia is not simply for "documents", but is also an excellent technology for building "applications", with many affordances for building compelling interactive experiences. A goal of this book is to show that it is possible to build sophisticated web applications using the original technology of the web, hypermedia, without the application developer needing to reach for the abstractions provided by JavaScript frameworks.

On the other hand, htmx *itself* is written in JavaScript, and we don’t want the message of this book 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 using scripting for the web?"

Scripting has been a massive force multiplier of the power of the web. Using scripting, web application developers are not only able to enhance their hypertext-based websites, but are able to create fully-fledged client-side applications that can often compete with native, thick

client applications. However, with this power, the browser has often become a distribution mechanism for what are effectively cross-platform thick clients, with the original, REST-ful web architecture being discarded in favor of a more JavaScript-centric and data-oriented approach.

Note, again, we do not claim this is a bad thing! It is a testament to the power of the web and, in particular, of modern browsers, that developers are able to build such sophisticated JavaScript-centric applications.

However, we want to show that, in addition to this more JavaScript-centric style, scripting *can also* be done in a manner compatible and consistent with Hypermedia-Driven Applications.

We feel that you are scripting in an HDA-compatible manner if the following two constraints are adhered to:

- First, the main data format exchanged between server and client must be hypermedia, the same as it would be with no scripting.
- Second, client-side state, outside the DOM itself, is kept to a minimum.

Satisfying these constraints requires us to adopt different practices and patterns than what is typically recommended for JavaScript, since the most common advice often comes, naturally, from JavaScript-centric applications.

Simply listing "best practices" is rarely convincing or edifying (and, lets be frank, it is boring). Instead, we will demonstrate them by implementing client-side features in Contact.app:

- We will add an overflow menu to hold the *Edit*, *View* and *Delete* actions, to clean up visual clutter in our list of contacts
- We will add an improved interface for bulk deletion
- We will add a keyboard shortcut for focusing the search box

The important concept in the implementation of each of these features is that, while they are implemented entirely client-side using scripting, they don’t exchange information with the server using, for example, JSON, and they don’t store a significant amount of state outside the DOM itself.

By satisfying these two constraints with our scripting, we will keep these features within the bounds of the Hypermedia Driven Application approach.

9.1. 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 at the start of this chapter 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 many languages to JavaScript, such as TypeScript, Dart, Kotlin, ClojureScript, F#. There is also the WebAssembly (WASM) bytecode format, which is supported as a compilation target for C, Rust, and the WASM-first language AssemblyScript.

However, most of these options 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 existing C/C++ libraries from JavaScript.

Because of this, we are going to focus on three client-side scripting technologies that *are* geared, or at least highly-compatible with, the Hypermedia Driven Application approach:

- 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.

Note that, as with CSS, we are not going to take a deep dive into any of these technologies. Insteat, we are going to show just enough to give you a flavor of how they work and, we hope, spark your interest in looking into each of them more extensively.

9.2. 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 "Vanilla" came into vogue as a play on the fact that there are 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.

(Ironically, at the same time, SPAs have become more popular, which require even more elaborate JavaScript frameworks!)

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

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

The message of *VanillaJS* here is: "The browser already has JavaScript baked into it, there isn't any need to download a framework for your application to function!"

With JavaScript having matured as a scripting language, this is certainly the case for many applications. It is especially true the case in HDAs, since, by using hypermedia, your application doesn't need many features typically 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

Without all this complexity being handled in JavaScript, your framework needs are dramatically reduced.

One of the best things about VanillaJS is how you install it: you don't have to! You can just

start writing JavaScript in your web application, and it will simply work. Amazing!

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 can make it a 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 has a complicated and confusing set of features for working with asynchronous code, that is, code that requires waiting for an asynchronous operation (such as an AJAX request) to complete
- Working with events in JavaScript is surprisingly difficult
- DOM APIs (a large portion of which were originally designed for Java) are verbose and frequently do not make common functionality easy to use.

None of these limitations are deal-breakers, of course, and many people prefer the "close to the browser" nature of vanilla JavaScript over more elaborate client-side scripting approaches.

9.3. A Simple Counter

To dive into vanilla JavaScript as a front end scripting option, let's create a simple counter widget. Counter widgets are a common "Hello World" example demonstrated by JavaScript frameworks, so looking at how it can be done in vanilla JavaScript will be instructive.

Our counter widget will be very simple: it will have a number, shown as text, and a button that increments the number.

Now, one problem with tackling this problem in vanilla JavaScript is that it lacks one thing most JavaScript frameworks provide: a default code and architectural style. With vanilla JavaScript, there are no rules!

This lack of structure, however, isn't all bad: it presents a great opportunity to take a small journey through various styles that people have developed for writing their JavaScript.

9.3.1. An Inline Implementation

To begin, let's start with the simplest thing imaginable: all of our JavaScript will be written inline, directly in the HTML. When the button is clicked, we will look up the `output`

element holding the number, and increment the number contained within it.

Listing 9. 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 to add an event listener

❸ Find the output via a querySelector() call

❹ JavaScript allows us use the `++` operator on strings

Not too bad!

The code is a little gronky, if you aren't used to the DOM APIs. It's a little annoying that we needed to add an `id` to the `output` element to make things work. Tje `document.querySelector()` function is a bit verbose compared with, say, the `$` function (a common function used in jQuery, an older, popular JavaScript librar.)

But (but!) it works, it's easy enough to understand, and, crucially, it doesn't require any other JavaScript libraries.

So that's the simple, inline approach.

9.3.2. Separating Our Scripting Out

While the inline implementation is simple, a more standard way to write this code would be to move the code into a separate JavaScript file. This JavaScript file would then either be linked to via a `<script src>` tag or placed into an inline `<script>` element by a build process.

Here we see the HTML and JavaScript *separated out* from one another, in different files. The HTML is now "cleaner" in that there is no JavaScript in it. The JavaScript is a bit more

complex than in our inline version: we need to look up the button using a query selector and add an *event listener* to handle the click event and increment the counter.

Listing 9. 2. Counter HTML

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

Listing 9. 3. Counter JavaScript

```
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

In moving the JavaScript out to another file, we are following a software design principle known as *Separation of Concerns (SoC).* Separation of Concerns posits that the various "concerns" (or aspects) of a software project should be divided up into multiple files, so that they don't "pollute" one another. JavaScript isn't markup, so it shouldn't be in your HTML, it should be *elsewhere*. Styling information, similarly, isn't markup, and so it belongs in a separate file as well (A CSS file, for example.) For quite some time, this Separation of Concerns was considered the "orthodox" way to build web applications.

A stated goal of Separation of Concerns is that we should be able to modify and evolve each concern independently, with confidence that we won't break any of the other concerns.

However, let's look at exactly how this principle has worked out in our simple counter example. If you look closely at the new HTML, it turns out that we've had to add a class to the button. We addedc this class so that we could look the button up in JavaScript and add in an event handler for the "click" event. Now, in both the HTML and the JavaScript, this

class name is just a string and there isn't any process to *verify* that the button has the right classes on it or its parents to ensure that the event handler is actually added to the right element.

Unfortunately, it has turned out that the careless use of CSS selectors in JavaScript can cause what is known as *jQuery soup*. jQuery soup is a situation where:

- The JavaScript that attaches a given behavior to a given element is difficult to find.
- Code reuse is difficult.
- The code ends up wildly disorganized and "flat", with lots of unrelated event handlers mixed together.

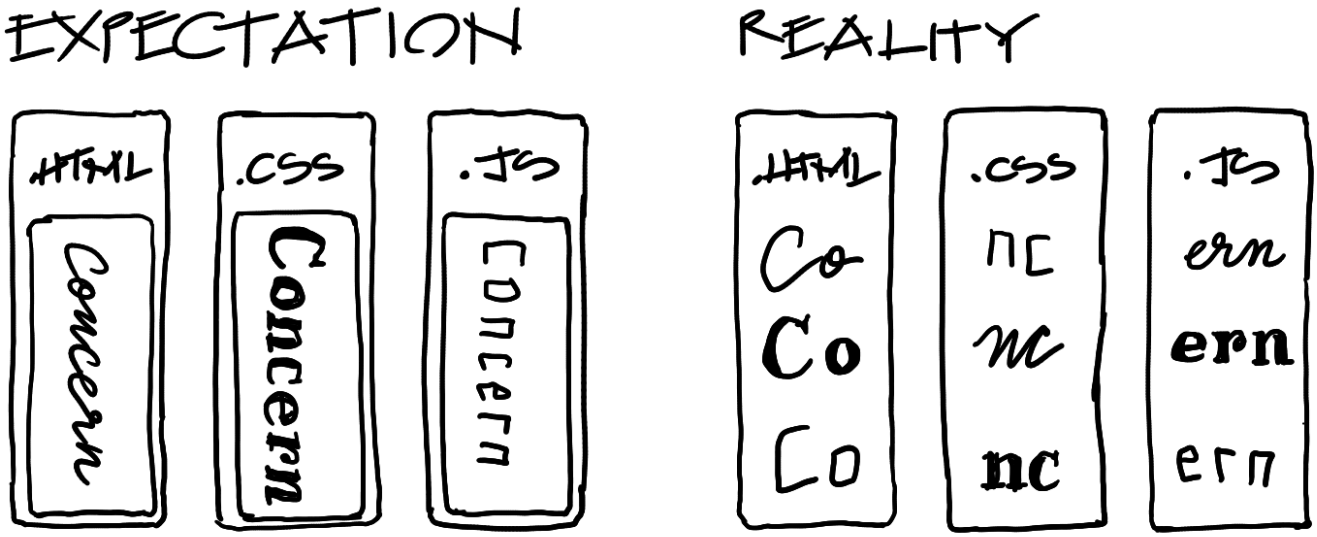
The name "jQuery Soup" comes from the fact that early JavaScript-heavy applications were typically built in jQuery, which, perhaps inadvertently, tended to encourage this style of JavaScript.

So, you can see that the notion of "Separation of Concerns" doesn't always work out as well as promised: our concerns end up intertwined pretty deeply, even when we separate them into different files.

To show that it isn't just naming between concerns that can get you into trouble, consider another small change to our HTML that demonstrates the problems with our separation of concerns. Imagine that we decide to change the number field from an `<output>` tag to an `<input type="number">`.

This small change to our HTML will break our JavaScript, despite the fact we have "separated" our concerns!

The fix for this issue is simple enough (we would need to change the `.textContent` property to `.value` property), but this demonstrates the burden of synchronizing markup changes and code changes across multiple files. Keeping everything in sync can become increasingly difficult as your application size increases .



The fact that small changes to our HTML can break our scripting indicates that the two are *tightly coupled*, despite being broken up into multiple files. This tight coupling suggests that separation between HTML and JavaScript (and CSS) is often an illusory separation of concerns: the concerns are sufficiently related to one another that they aren't easily separated.

In Contact.app we are not *concerned* with "structure", "styling" or "behavior"; we are concerned with collecting contact info and presenting it to users. SoC, in the way it's formulated in web development orthodoxy, is not really an inviolate architectural guideline, but rather a stylistic choice that, as we can see, can even become a hindrance.

9.3.3. Locality

It turns out that there is a burgeoning reaction *againts* Separation of Concerns. Consider the following web technologies and techniques:

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

Each of these technologies *colocate* code in various languages that address a single *feature* (typically a UI widget).

All of them mix *implementation* concerns together in order to present a unified abstraction to the end-user. Separating technical detail concerns just isn't as much of an, ahem, concern.

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.

In a Hypermedia Driven Application, we feel that Locality of Behavior is often more important than Separation of Concerns.

9.3.4. What To Do With Our Counter?

So, should we go back to the `onclick` attribute way of doing things? That approach certainly wins in Locality of Behavior, and has the additional benefit that it is baked into HTML.

Unfortunately, however, the `on*` JavaScript attributes also come with some 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.

Consider this common situation: you have a popup, and you want it to be dismissed when a user clicks outside of it. The listener will need to be on the body element in this situation, far away from the actual popup markup. This means that the body element would need to have listeners attached to it that deal with many unrelated components. Some of these components may not even be on the page when it was first rendered, if they are added dynamically after the initial HTML page is rendered!

So vanilla javaScript and Locality of Behavior don't seem to mesh *quite* as well as we would like them to. The situation is not hopeless, however: it's important to understand that LoB does not require behavior to be *implemented* at a use site, but merely *invoked* there. That is, we don't need to write all our code on a given element, we just need to make it clear that a given element is *invoking* some code, which can be located elsewhere.

Keeping this in mind, it *is* possible to improve LoB while writing JavaScript in a separate file, provided we have a reasonable system for structuring our JavaScript.

RSJS

RSJS (the "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 provides a solution to the lack of a standard code style for vanilla JavaScript that we mentioned earlier.

We won't reproduce all the RSJS guidelines here, but here are the ones most relevant for our counter widget:

- "Use `data-` attributes" in HTML - invoking behavior via adding data attributes makes it obvious there is JavaScript happening, as opposed to using 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

To follow the RSJS guidelines, let's restructure our current HTML and JavaScript files. First, we will use *data attributes*, that is, HTML attributes that begin with `data-`, a standard feature of HTML, to indicate that our HTML is a counter component. We will then update our JavaScript to use an attribute selector that looks for the `data-counter` attribute as the root element in our counter component and wires in the appropriate event handlers and logic. Additionally, let's rework the code to use `querySelector()` and add the counter functionality to *all* counter components found on the page. (You never know how many counter's you might want!)

Here is what our code looks like now:

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 descendant 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

Using RSJS solves, or at least alleviates, many of the problems we pointed out with our first, unstructured example of vanilla JS being split out to 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 component on the page it will just work.
- The code is **well-organized** --- one behavior per file

All in all, RSJS is a good way to structure your vanilla JavaScript in a Hypermedia Driven Application. So long as the JavaScript isn't communicating with a server via a plain data JSON API, or holding a bunch of internal state outside of the DOM, this is perfectly compatible with the HDA approach.

Let's take a look at implementing a feature in Contact.App using the RSJS/vanilla JavaScript approach.

9.3.5. Vanilla JS in action: an overflow menu

Our homepage has "Edit", "View" and "Delete" links for every contact in our table. This uses a lot of space and creates visual clutter. We're going to place these actions inside a menu with a button to open it.

Let's sketch the markup we want for our overflow menu. First, we need an element, we'll use a `div`, to enclose the entire widget and mark it as a menu component. Within this div, we will have a standard `button` that will function as the mechanism that shows and hides our menu items. Finally, we'll have another `div` that holds the menu items that we are going to show. These menu items will be simple anchor tags, as they are in the current contacts table.

Here is what our updated, RSJS-structured HTML looks like:

```
<div data-menu="closed"> ❶
  <button type="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 component

❷ This button will open and close our menu

❸ A container for our menu items

❹ Mark menu items so we can implement moving between them with arrow keys

With this HTML written, we can move on to creating

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 button = menu.querySelector("[data-menu-button]"), ❷
  body = menu.querySelector("[data-menu-items]"),
  items = Array.from(body.querySelectorAll("[data-menu-item]"));

  const isOpen = () => 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
  });


  // more code below...

})
```

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 [1] 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 resulting markup less readable.

Let's implement toggling the menu:

```
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(event) {
  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]:

```
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`.

9.4. Alpine.js

OK, so that's an in-depth look at how to structure plain "vanilla" JavaScript. Now let's turn our attention to a JavaScript framework that enables a different approach for adding dynamic behavior to your application, Alpine.js (https://alpinejs.dev). Alpine is a relatively new JavaScript library which allows you to embed your code directly in HTML, akin to the `on*` attributes available in plain HTML and JavaScript, but takes this concept much further. Alpine bills itself as a modern replacement for jQuery, a widely used, older JavaScript library, and, as you will see, it lives up to that promise.

Installing Alpine is very easy: it is a single file and is dependency-free, so you can simply include it via a CDN:

Listing 9. 4. Installing Alpine

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

You can also install it via a package manager such as NPM, or vendor it from your own server.

Alpine provides a set of HTML attributes, all of which begin with the `x-` prefix, the main one of which is `x-data`. The content of `x-data` is a JavaScript expression which evaluates to an object. The properties of this object can, then, be access within the element that the `x-data` attribute is located on.

Let's look at our counter example, and how to implement it using Alpine. For the counter, the only state we need to keep track of is the current number, so let's declare a JavaScript object with one property, `count`, in an `x-data` attribute on the div for our counter:

Listing 9. 5. Counter with Alpine, line 1

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

This defines our state, that is, the data we are going to be using to drive dynamic updates to

the DOM. With the state declared like this, we can now use it *within* the div element it is declared on. Let's add an `output` element with an `x-text` attribute. We will *bind* the `x-text` attribute to the `count` attribute we declared in the `x-data` attribute on the parent `div` element. This will have the effect of setting the text of the `output` element to whatever the value of `count` is: if `count` is updates, so will the text of the `output`. This is "reactive" programming, in that the DOM will "react" to changes to the backing data.

Listing 9. 6. Counter with Alpine, lines 1-2

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

❶ The `x-text` attribute.

Next, we need to update the count, using a button. Alpine allows you to attach event listeners with the `x-on` attribute. To specify the event to listen for, you add a colon and then the event name after the `x-on` attribute name. Then, the value of the attribute is the JavaScript you wish to execute. This is similar to the plain `on*` attributes we discussed earlier, but it turns out to be much more flexible.

We want to listen for a `click` event, and we want to increment `count` when a click occurs, so here is what our Alpine code will look like:

Listing 9. 7. 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*.

And, would you look at that, we're done already! A simple component, like a counter, should be simple, and Alpine sure delivers on that!

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

As we said, the Alpine `x-on:click` attribute (or its shorthand, the `@click` attribute) is similar to the built-in `onclick` attribute. However, it has additional features that make it

significantly 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. For example, if you wanted to listen for the `htmx:after-request` event you could write `x-on:htmx:after-request="doSomething()"`

9.4.2. Reactivity and Templating

We hope that you'll agree that the AlpineJS version of the counter widget is better, in general, than the VanillaJS implementation, which was either somewhat hacky or spread out over multiple files.

A big part of the power of AlpineJS is that it supports a notion of "reactive" variables, allowing you to bind the count of the `div` element to a variable that both the `output` and the `button` can reference, and properly updating all the dependencies when a mutation occurs. Alpine allows for much more elaborate data bindings than what we have demonstrated here, and it is an excellent general purpose client-side scripting library.

9.4.3. Alpine.js in Action: A Bulk Action Toolbar

Next, let's implement a feature in Contact.app with Alpine. As it stands currently, 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 takes up a lot of room. If we wanted to add additional "bulk" actions, this wouldn't really scale very well visually.

In this section, we'll replace this single button with a toolbar. Furthermore, the toolbar will only appear when the user starts selecting contacts. Finally, it will show how many contacts are selected and let you select all contacts in one go.

The first thing we will need to add is an `x-data` attribute, to hold the state that we will use to determine if the toolbar is visible or not. We will need to place this on a parent element

of both the toolbar that we are going to add, as well as of the checkboxes, which will be updating the state when they are checked and unchecked. The best option given our current HTML is to place the attribute on the `form` element that surrounds the contacts table. We will declare a property, `selected`, which will be an array that holds the selected contact ids, based on the checkboxes that are selected.

Here is what our form tag will look like:

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

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

Next, at the top of the contacts table, we are going to add a `template` tag. A template tag is *not* rendered by a browser, by default, so you might be surprised that we are using it. However, by adding an Alpine `x-if` attribute, we can tell Alpine: if a condition is true, show the HTML within this template.

Recall that we want to show the toolbar if and only if one or more contacts are selected. But we know that we will have the ids of the selected contacts in the `selected` property. Therefore, we can check the *length* of that array to see if there are any selected contacts, quite easily:

```
<template x-if="selected.length > 0"> ❶
  <div class="box info tool-bar">
    <slot x-text="selected.length"></slot>
    contacts selected

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

❶ Show this HTML if there are 1 or more selected contacts

❷ We will implement these buttons in just a moment

The next step is to ensure that toggling a checkbox for a given contact adds (or removes) a given contact's id from the `selected` property. To do this, we will need to use a new

Alpine attribute, `x-model`. The `x-model` attribute allows you to *bind* a given element to some underlying data, or it's "model".

In this case, we want to bind the value of the checkbox inputs to the `selected` property. This is how we do this:

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

❶ The `x-model` attribute binds the `value` of this input to the `selected` property

Now, when a checkbox is checked or unchecked, the `selected` array will be updated with the given rows contact id. Furthermore, mutations we make to the `selected` array will similarly be reflected in the checkboxes' state. This is known as a *two-way* binding.

With this code written, we can make the toolbar appear and disappear, based on whether contact checkboxes are selected! Awesome.

Implementing Actions

Now that we have the mechanics of showing and hiding the toolbar, let's look at how to implement the buttons within the toolbar.

Let's first implement the "Clear" button, because it is quite easy. All we need to do is, when the button is clicked, clear out the `selected` array. Because of the two-way binding that Alpine provides, this will uncheck all the selected contacts (and then hide the toolbar)!

Here is the code:

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

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

❶ Just reset the `selected` array

Pretty easy!

The "Delete" button, however, will be a bit more complicated. It will need to do two things:

first it will confirm if the user indeed intends to delete the contacts selected, and, if the user confirms the action, it will use the htmx JavaScript API to issue a `DELETE` request.

```
<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 Confirm the user wishes to delete the selected number of contacts

❷ Issue a `DELETE` using the htmx JavaScript API

Note that we are using the short-circuiting behavior of the `&&` operator in JavaScript to avoid the call to `htmx.ajax()` if the `confirm()` call returns false.

The `htmx.ajax()` function is just a way to directly access the normal, HTML-driven hypermedia exchange that htmx's attributes give you from JavaScript. We pass in that we want ot issue a `DELETE` to `/contacts`. We then pass in two additional pieces of information: `source` and `target`. The `source` properly is the element from which htmx will collect data to include in the request. We set this to `$root`, which is a special symbol in Alpine that will be the element that has the `x-data` attribute declared on it. In this case, it will be the form containing all of our contacts. The `target`, or where the response HTML will be placed, is just the entire document's body, since the `DELETE` handler returns a whole page when it completes.

Note that we are using Alpine here in a Hypermedia Driven Application compatible manner. We *could* have issued an AJAX request directly from Alpine and perhaps updated an `x-data` property depending on the results of that request. But, instead, we delegated to htmx's JavaScript API, which made a *hypermedia exchange* with the server. This is the key to scripting in a Hypermedia Driven Application.

So, with all of this in place, we now have a much improved experience for performing bulk actions on contacts: less visual clutter and the toolbar can be extended with more options without creating bloat in the main interface of our app.

9.5. _hyperscript

The next scripting technology we are going to look at is a bit further afield: _hyperscript

(https://hyperscript.org) While previous two examples are JavaScript-oriented, _hyperscript is an entire new scripting language for front-end development. _hyperscript has a completely different syntax than JavaScript, and is based on an older language called HyperTalk. HyperTalk was the scripting language for a technology called HyperCard, an old hypermedia system available on early Macintosh Computers.

The most noticeable thing about _hyperscript is that it resembles English prose more than it resembles other programming languages. _hyperscript was initially created as a sibling project to htmx, because it was felt that JavaScript wasn't event-oriented enough and made adding small script enhancements to htmx applications too cumbersom. Like Alpine, _hyperscript, is positioned as a modern jQuery replacement and, further, as alternative to JavaScript.

Also like Alpine, _hyperscript allows you to write your scripting inline, in HTML.

Unline Alpine, _hyprescript is _not_ reactive. It instead focuses on making DOM manipulations in response to events easy and clear. It has built-in language constructs for many DOM operations, preventing you from needing to navigate the sometimes-verbose JavaScript DOM 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 9. 8. Installing _hyperscript via CDN

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

_hyperscript uses the `_` (underscore) attribute for putting scripting on DOM elements. You may also use the `script` or `data-script` attributes, depending on your HTML validation needs.

Let's look at how to implement the simple counter component we have been looking at using _hyperscript. We will place an `output` element and a `button` inside of a `div`. To implement the counter, we will need to add a small bit of _hyperscript to the button. On a

click, the button should increment the text of the previous `output` tag.

Well, it turns out that that last sentence is almost valid _hyperscript!

Here is our code:

```
<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!

Let's go through each component of the _hyperscript:

- `on click` This is an event listener, telling the button to listen for a `click` event and then executing the remaining code
- `increment` This is a "command" in _hypsercipt that, well, increments things, similar to the `++` operator in JavaScript
- `the` "the" doesn't have any semantic meaning _hyperscript, but can used to make scripts more readable
- `textContent of` - This one form of *property access* in _hyperscript. You are probably familiar with the JavaScript syntax `a.b`, meaning "Get the property `b` on object `a`\". _hyperscript supports this syntax, but _also_ supports the forms `b of a` and `a's b`. Which one you use should depend on which one is most readable.
- `the previous` The `previous` expression in hyperscript finds the previous element in the DOM that matches some condition
- `<output />` This is a *query literal*, which is a CSS selector is wrapped between "<" and "/>"

In this code, the `previous` keyword (and the accompanying `next`) is an example of how _hyperscript makes DOM operations easier: there is no such native functionality to be found in the standard DOM API, and implementing it is actually trickier than you might think!

So, you can see, _hyperscript is very expressive, particularly when it comes to DOM manipulations. This makes it easier to embed scripts directly in HTML: since the scripting language is more powerful, scripts written in it tend to be shorter and easier to read.

Natural Language Programming?

Seasoned programmers are often suspicious of _hyperscript: There have been many "natural language programming" (NLP) projects that target non-programmers and beginner programmers, assuming that being able to read code in their "natural language" will give them the ability to write it as well. This has lead to some badly written and structured code and has failed to live up to the (often over the top) hype.

_hyperscript is _not_ an NLP programming language. Yes, its syntax is inspired in many places by the speech patterns of web developers. But _hyperscript's readability is achieved not through complex heuristics or fuzzy NLP processing, but rather through judicious use of common parsing tricks, coupled with a culture of readability.

As you can see in the above example, with the use of a *query reference*, `<output/>`, _hyperscript does not shy away from using DOM-specific, non-natural language when appropriate.

9.5.1. _hyperscript in action: a keyboard shortcut

While the counter demo is a good way to compare various approaches to scripting, the rubber meets the road when you try to actually implement a useful feature with an approach. For _hyperscript, let's add a keyboard shortcut to Contact.app: when a user hits Shift-S in our app, we will focus the search field.

Since our keyboard shortcut focuses the search input, let's put the code for it on that search input, satisfying locality.

Here is the original HTML for the search input:

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

We will add an event listener using the `on keydown` syntax, which will fire whenever a keydown occurs. Further, we can use an *event filter* syntax in _hyperscript using square brackets after the event. In the square brackets we can place a _filter expression_ that will

filter out `keydown` events we aren't interested in. In our case, we only want to consider events where the shift key is held down and where the "S" key is being pressed. We can create a boolean expression that inspects the `shiftKey` property (to see if it is `true`) and the `code` property (to see if it is `"KeyS"`) of the event to achieve this.

So far our hyperscript looks like this:

Listing 9. 9. A Start On Our Keyboard Shortcut

```
on keydown[shiftKey and code is 'KeyS'] ...
```

Now, by default, _hyperscript will listen for a given event _on the element it is declared on_. So, in this case, with the script we have so far, we would only get `keydown` events if the search box is already focused. That's not what we want! We want to have this key work *globally*, no matter which element has focus.

Not a problem! We can listen for the `keyDown` event elsewhere by using a `from` clause in our event handler. In this case we want to listen for the `keyDown` from the window, and our code ends up looking, naturally, like this:

We need to attach the listener to the whole window instead. No problem:

Listing 9. 10. Listening Globally

```
on keydown[shiftKey and code is 'KeyS'] from window ...
```

Using the `from` clause, we can attach the listener to the window while, at the same time, keeping the code on the element it logically relates to. Very nice!

Now that we've picked out the event we want to use to focus the search box, let's implement the actual focusing by calling the standard `.focus()` method.

Here is the entire script, embedded in HTML

Listing 9. 11. Our Final Script

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

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

Given all the functionality, this is surprisingly terse, and, as an English-like programming language, pretty easy to read.

9.5.2. Why a new programming language?

This is all well and good, but you may be thinking "An entirely new scripting language? That seems excessive." And, at some level, you are right: JavaScript is a decent scripting language, is very well optimized and is widely understood in web development. On the other hand, by creating an entirely new front end scripting language, we had the freedom to address some problems that we saw generating ugly and verbose code in JavaScript:

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, instead, requires either the explicit use of callbacks _or_ the use of explicit `async` annotations (which can't be mixed with synchronous code).

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, making array property access act like a flat-map operation. jQuery has a similar feature, but only for its own data structure.

Native CSS Syntax

In _hyperscript, you can use things like CSS class and ID literals, or CSS query literals, directly in the language, rather than needing to call out to a wordy DOM API, as you do in JavaScript

Deep Event Support

Working with events in _hyperscript is far more pleasant than working with them in JavaScript, with native support for responding to and sending events, as well as for common event-handling patterns such as "debouncing" or rate limiting events.

Hyperscript also provides declarative mechanisms for synchronizing events within a given element and across multiple elements.

Again we wish to stress that, in this example, we are not stepping outside the line of a Hypermedia Driven Application: we are only adding front-end, client-side functionality with our scripting. We are not creating and managing a large amount of state outside of the DOM itself, or communicating with the server in a non-hypermedia exchange. Additionally, since _hyperscript embeds so well in HTML, it keeps the focus _on the hypermedia_, rather than on the scripting logic.

Taken all together, given a certain style of scripting and certain scripting needs, _hyperscript can provide an excellent scripting experience for your Hypermedia Driven Application. Of course, it is small and obscure programming language, so we won't blame you if you decide to pass on it, but it is at least worth a look to understand what it is trying to achieve, if only out of intellectual interest.

9.6. Using Off-the-shelf Components

That concludes our look at three different options for *your* scripting infrastructure, that is, the code that *you* write to enhance your Hypermedia Driven Application. However, there is another major area to consider when discussing client side scripting: "off the shelf" components. That is, JavaScript libraries that other people have created that offer some sort of functionality, such as showing modal dialogs.

Components have become very popular in the web development works, with libraries like DataTables (https://datatables.net/) providing rich user experiences with very little JavaScript code on the part of a user. Unfortunately, if these libraries aren't integrated well into a website, they can begin to make an application feel "patched together". Furthermore, some libraries go beyond simple DOM manipulation, and require that you integrate with a server end point, almost invariably with a JSON data API. This means you are no longer building a Hypermedia Driven Application, simply because a particular widget demands something different. A shame!

9.6.1. Integration Options

The best JavaScript libraries to work with when you are building a Hypermedia Driven Application are ones that:

- Mutate the DOM but don't communicate with a server over JSON
- Respect HTML norms (e.g. using `input` elements to store values)
- Trigger many custom events, as the library updates things

The last point, triggering many custom events (over the alternative of using lots of methods and callbacks) is especially important, as these custom events can be dispatched or listened to without additional glue code written in a scripting.

Let's take a look at two different approaches to scripting, one using JavaScript call backs, and one using events.

To make things concrete, let's implement a better confirmation dialog for the `DELETE` button we created in Alpine in the previous section. In the original example we used the `confirm()` function built in to JavaScript, which shows a pretty bare-bones system confirmation dialog. We will replace this function with a popular JavaScript library, SweetAlert2, that shows a much nicern nice looking confirmation dialog. Unlink the `confirm()` function, which blocks and returns a boolean (`true` if the user confirmed, `false` otherwise), SweetAlert2 returns a `Promise` object, which is a JavaScript mechanism for hooking in a callback once an asynchronous action (such as waiting for a user to confirm or deny an action) completes.

Integrating Using Callbacks

With SweetAlert2 installed as a library, you have access to the `Swal` object, which has a `fire()` function on it to trigger showing an alert. You can pass in arguments to the `fire()` method to configure exactly what the buttons on the confirmation dialog look like, what the title of the dialog is, and so forth. We won't get into these details too much, but you will see what a dialog looks like in a bit.

So, given we have installed the SweetAlert2 library, we can swap it in place of the `confirm()` function call. We then need to restructure the code to pass a *callback* to the `then()` method on the `Promise` that `Swal.fire()` returns. A deep dive into Promises is beyond the scope of this chapter, but suffice to say that this callback will be called when a user confirms or denys the action. If the user confirmed the action, then the `result.isConfirmed` property will be `true`.

Given all that, our updated code will look like this:

Listing 9. 12. A Callback-based Confirmation Dialog

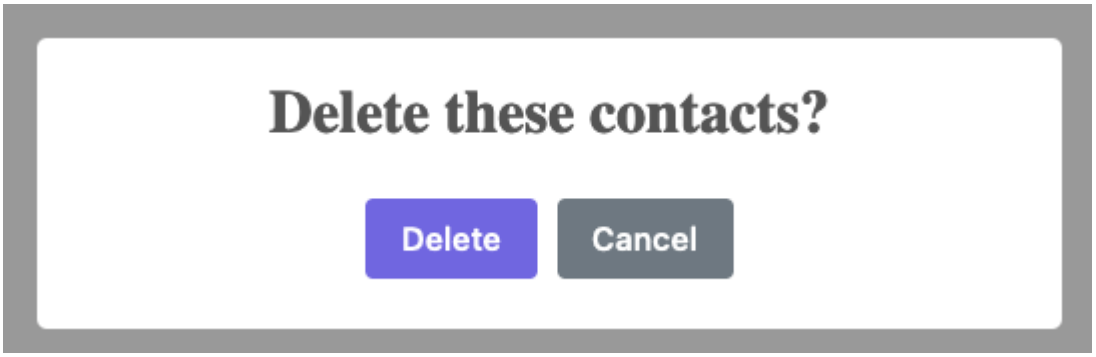
```
<button type="button" class="bad bg color border"
  @click="Swal.fire({  ❶
                  title: 'Delete these contacts?',  ❷
                  showCancelButton: true,
                  confirmButtonText: 'Delete'
                }).then((result) => { ❸
                  if (result.isConfirmed) {
                    htmx.ajax('DELETE', '/contacts', { source: $root, target:
document.body })
                  }
                });"
>Delete</button>
```

❶ Invoke the `Swal.fire()` function

❷ Configure how the dialog appears

❸ Handle the result of the users selection

And now, when this button is clicked, we get a nice looking dialog in our web application:



Much nicer than the system confirmation dialog! Still, this feels a little wrong. This is a lot of code to write just to trigger a slightly nicer `confirm()`, isn't it? And the htmx JavaScript code we are using here feels a little awkward. It would be more natural to move the htmx out to attributes on the button, as we have been doing, and then trigger the request via events.

So let's take a different approach and see how that looks.

Integrating Using Events

To clean this code up, we will pull the `Swal.fire()` code out to a custom JavaScript function we will create called `sweetConfirm()`. `sweetConfirm()` will take the dialog

options that are passed into the `fire()` method, as well as the element that is confirming an action. The big difference between the code we already have and `sweetConfirm()` is that `sweetConfirm()`, rather than calling some htmx directly, will, instead, trigger a `confirmed` event on the button when the user confirms they wish to delete.

Here is what our JavaScript function looks like:

Listing 9. 13. An Event-based Confirmation Dialog

```
function sweetConfirm(elt, config) {
      Swal.fire(config) ❶
          .then((result) => {
                  if (result.isConfirmed) {
                      elt.dispatchEvent(new Event('confirmed')); ❷
                  }
          });
}
```

❶ Pass the config through to the `fire()` function

❷ If the user confirmed the action, trigger an `confirmed` event

With this method available, we can now tighten up our delete button quite a bit. We can remove all the SweetAlert2 code that we had in the `@click` Alpine attribute, and simply call this new `sweetConfirm()` method, passing in the arguments `$el`, which is the Alpine syntax for getting "the current element" that the script is on, and then the exact configuration we want for our dialog.

If the user confirms the action, a `confirmed` event will be triggered on the button. This means that we can go back to using our trusty htmx attributes! Namely, we can move `DELETE` to an `hx-delete` attribute, and we can we can use `hx-target` to target the body. And then, and here is the crucial step, we can use the `confirmed` event that is triggered in the `sweetConfirm()` function, to trigger the request, but adding an `hx-trigger` for it!

Here is

Listing 9. 14. An Event-based Confirmation Dialog

```
<button type="button" class="bad bg color border"
        hx-delete="/contacts" hx-target="body" hx-trigger="confirmed" ❶
        @click="sweetConfirm($el, ❷
                { title: 'Delete these contacts?',  ❸
                  showCancelButton: true,
                  confirmButtonText: 'Delete'})">
```

❶ htmx attributes are back

❷ We pass the button in to the function, so an event can be triggered on it

❸ We pass through the SweetAlert2 configuration information

Now, as you can see, this event-based code is much cleaner and certainly more "HTML-ish". The key to this cleaner implementation is that our new `sweetConfirm()` function an event that htmx is able to listen for. This is why a rich event model is important to look for when choosing a library to work with, both with htmx and with Hypermedia Driven Applications in general.

Unfortunately, due to the prevalence and dominance of the JavaScript-first mindset today, many libraries are like SweetAlert2: they expect you to pass a callback in the first style. In these cases you can use the technique we have demonstrated here, wrapping the library in a function that triggers events in a callback, to make the library more hypermedia and htmx-friendly.

9.7. Pragmatic Scripting

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

We have shown you quite a few tools and techniques for scripting in a Hypermedia Driven Application. How should you pick between them? The sad truth is that there will never be a single, always correct answer to this question. Are you committed to vanilla JavaScript-only, perhaps due to company policy? Well, you can use vanilla JavaScript effectively to script your Hypermedia Driven Application. Do you have more leeway and like the look of Alpine.js? That’s a great way to add more structured, localized JavaScript to your application, and offers some nice reactive features as well.

Are you perhaps a bit more wild in your technical choices? Maybe _hypercript would be worth taking a look at. (We think so!)

Sometimes you might even consider picking two (or more!) of these approaches within an application. Each has its own strengths and weaknesses, and all of them are relatively small and self-contained, so picking the right tool for the job at hand might be the best approach.

In general, we encourage a *pragmatic* approach to scripting: whatever feels right is probably right (or, at least, right *enough*) for you. Rather than being concerned about which particular approach is taken for your scripting, we would focus with these more general concerns:

- Avoiding communicating with the server via JSON data APIs
- Avoiding storing large amounts of state outside of the DOM
- Favoring using events, rather than hard-coded callbacks or method calls

But even on these topics, sometimes a web developer has to do what a web developer has to do. If the perfect widget for your application exists but, darn it, it uses a JSON data API, that's OK. Just don't make it a habit!

9.8. Summary

- Maximize locality of behavior, sometimes at the expense of separation of concerns.
- Avoid maintaining large amounts of state outside of the DOM.
- Avoid communicating with the server using JSON data APIs.
- 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.
- 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 too much about theoretical purity.

[1] https://www.w3.org/WAI/ARIA/apg/patterns/menubutton/