ALPINE.js HANDBOOK

FLAVIO COPES

Preface

This book aims to be an introduction to Alpine.js, a very cool library you can use to provide client-side interactivity to your pages in an easy way.

If you're unfamiliar with JavaScript or TypeScript, before reading this book I highly recommend reading my handbooks on those topics.

This book was published in late 2024.

Legal

Introduction

Alpine is a lightweight frontend library that I've used with a lot of happiness, specifically to add client-side interactivity to a page.

Things like "if I click this button, show this modal" or "close the modal if I click outside of it", which are always kind of unfriendly to do with vanilla JavaScript and DOM APIs. Not hard, but I have to think a bit about how to do some things, while with Alpine, it's very straightforward.

Compared to most UI libraries you see these days, like React, Svelte, Vue, Angular, and so on, Alpine is A LOT simpler, and it's 100% focused on so-called "JavaScript sprinkles", which are little bits of interactivity.

As opposed to taking control of the entire UI like most of those bigger frameworks tend to do (although you can *also* use them for tiny portions of a web page).

Alpine is 100% focused on this, and it's very good at it.

In this book I'll explain the basics of Alpine, so you can learn the fundamentals of this super handy library.

To learn more, the official documentation is the best place to go: https://alpinejs.dev/start-here

Installing Alpine

Installing Alpine can be as simple as including the script tag in your HTML page:

```
<script defer src="
<https://cdn.jsdelivr.net/npm/alpinejs@3.x.x/dist/cdn.min.js>"></script>
```

Or, you can add it as a JavaScript module into your project:

```
npm install alpinejs
```

And you can initialize it with

```
import Alpine from 'alpinejs'

window.Alpine = Alpine

Alpine.start()
```

You can choose the way you prefer.

Adding the script tag is the simplest as you automatically have the Alpine object available in the global object, and already initialized (you don't have to manually call `Alpine.start()`)

The basics of Alpine

The basics of Alpine are pretty straightforward: you define some data inside your markup using the `x-data` attribute, and then you can show this data to the users using the `x-text` attribute on a tag, like this:

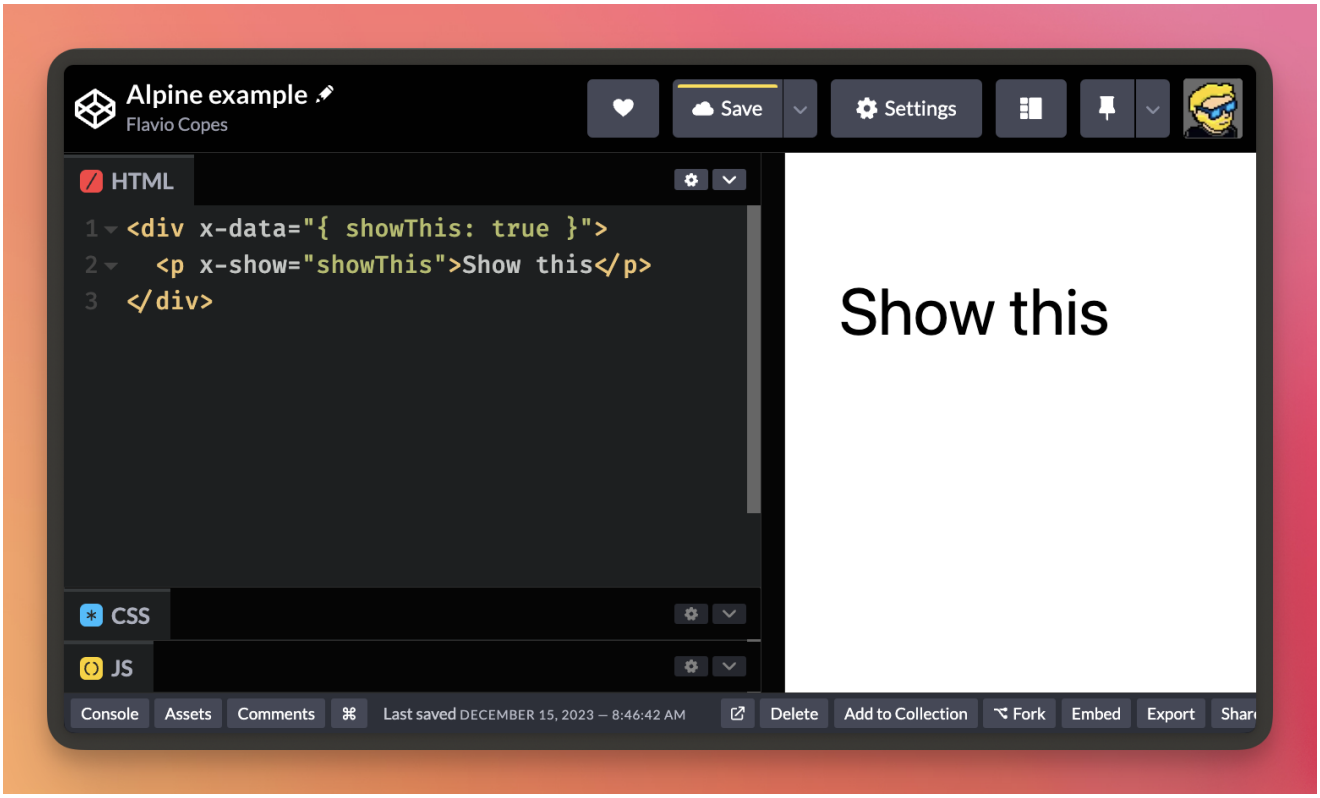
```
<div x-data="{ name: 'Joe' }">

</div>
```

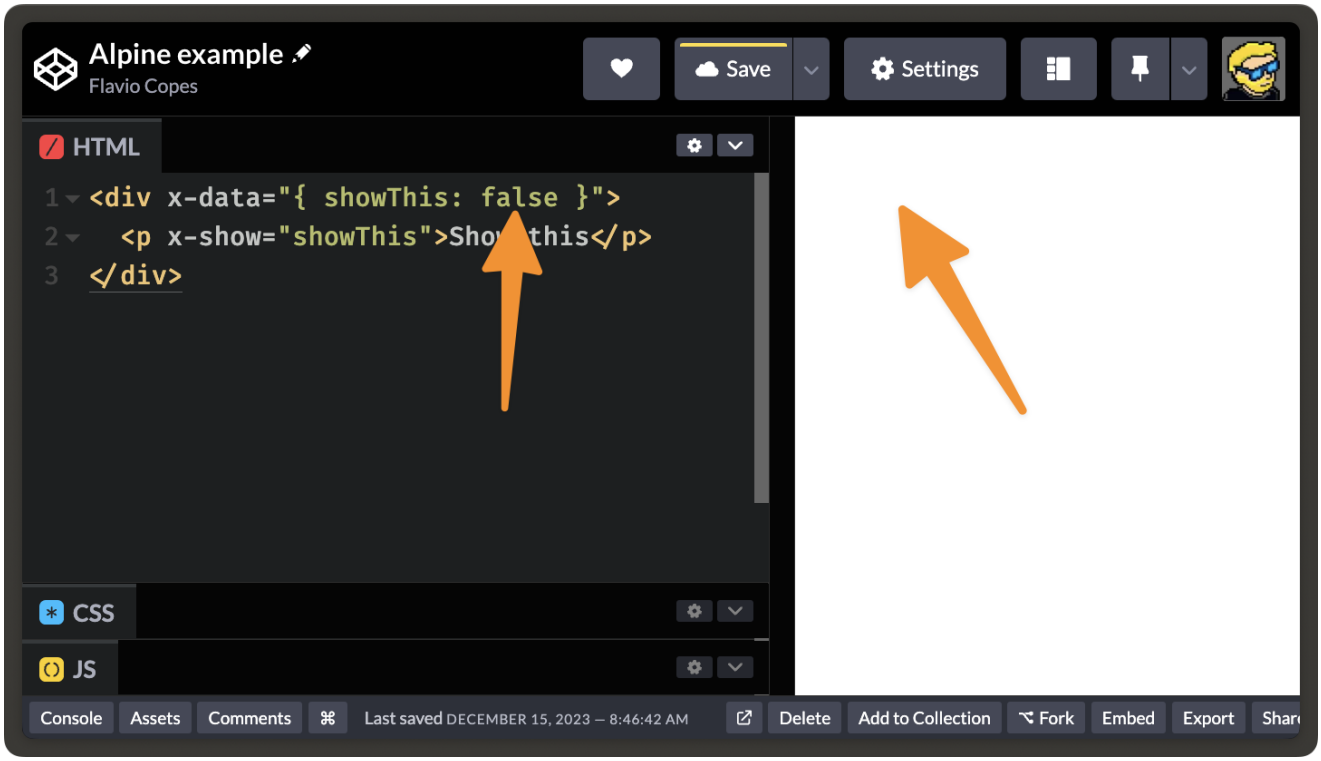
```
<div x-data="{ name: 'Joe' }">
  <p x-text="name"></p>
</div>
```

Using boolean state and the `x-show` attribute you can easily hide or show some portions of the UI:

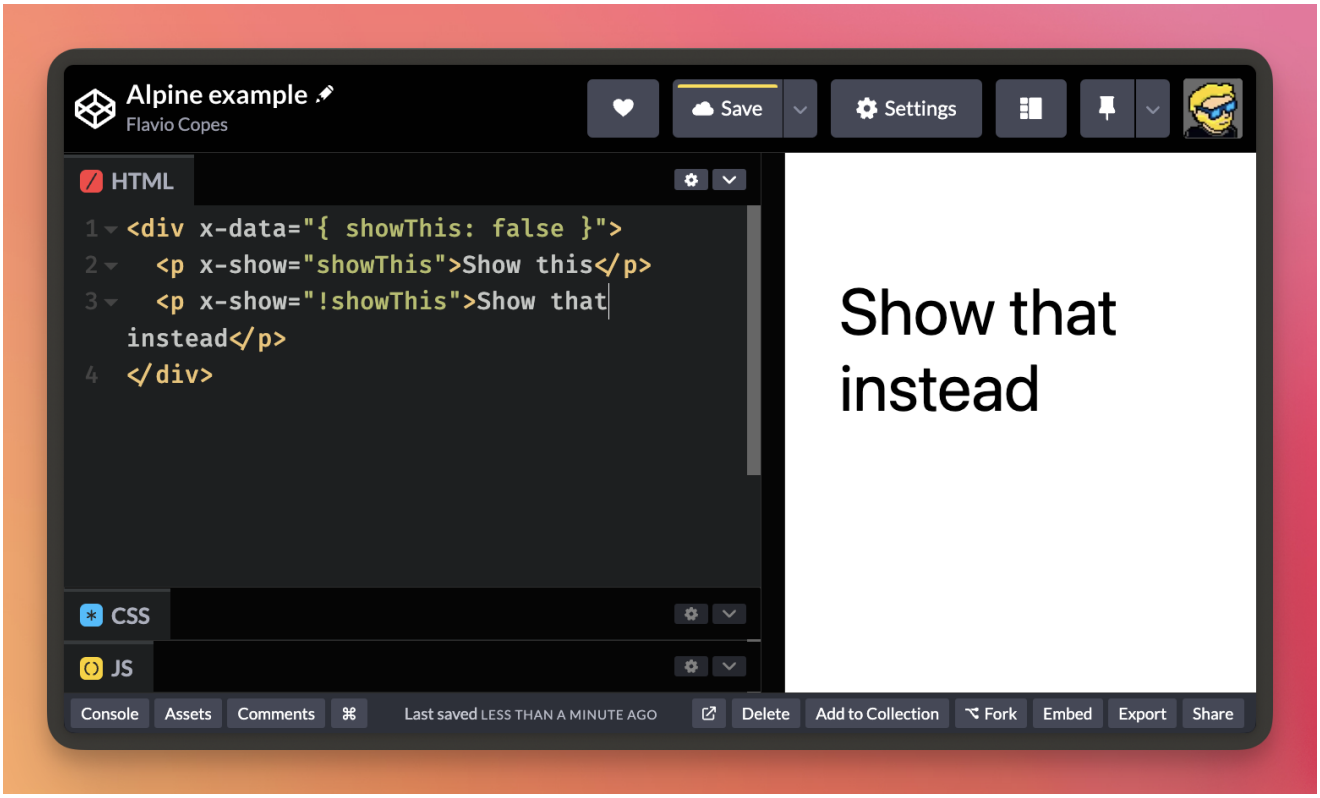
```
<div x-data="{ showThis: false }">
  <p x-show="showThis">Show this</p>
</div>
```



If you set the `showThis` variable to `false`, nothing will be displayed:

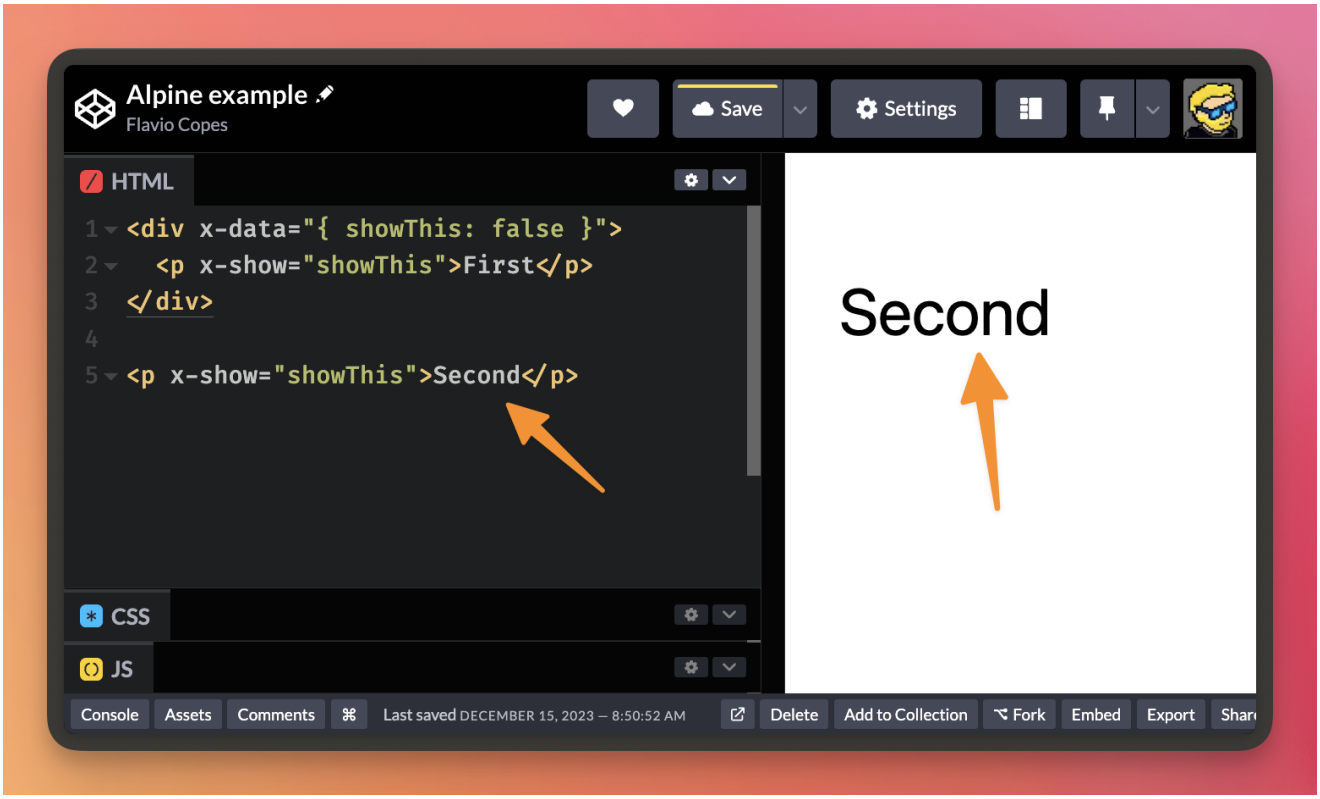


You can use `x-show="!showThis"` to do the opposite:



One thing to the data is only available on child elements of the tag with the `x-data` attribute.

For example in this case the second `p` tag doesn't have access to the `showThis` value, and it's shown regardless of its value defined in the `div` (it's isolated):

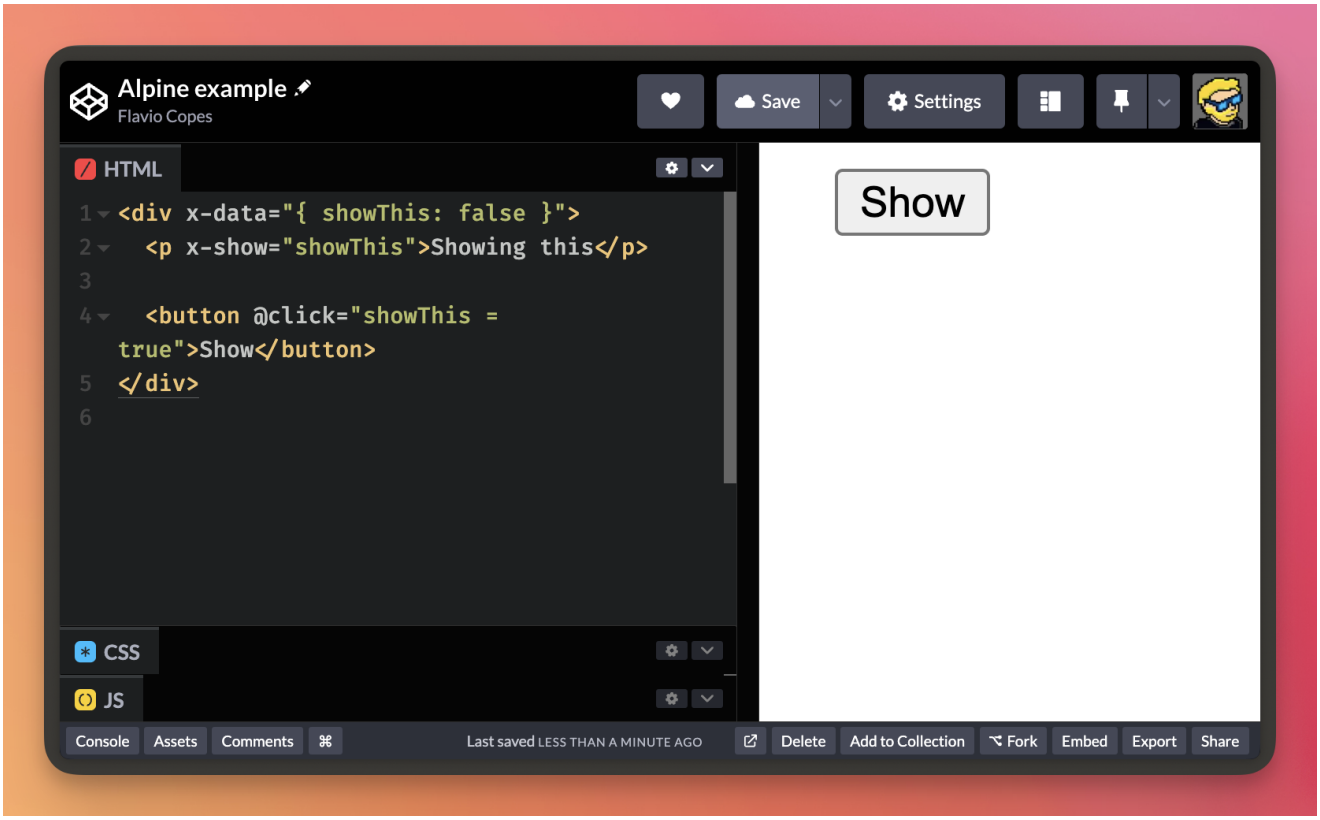


Events

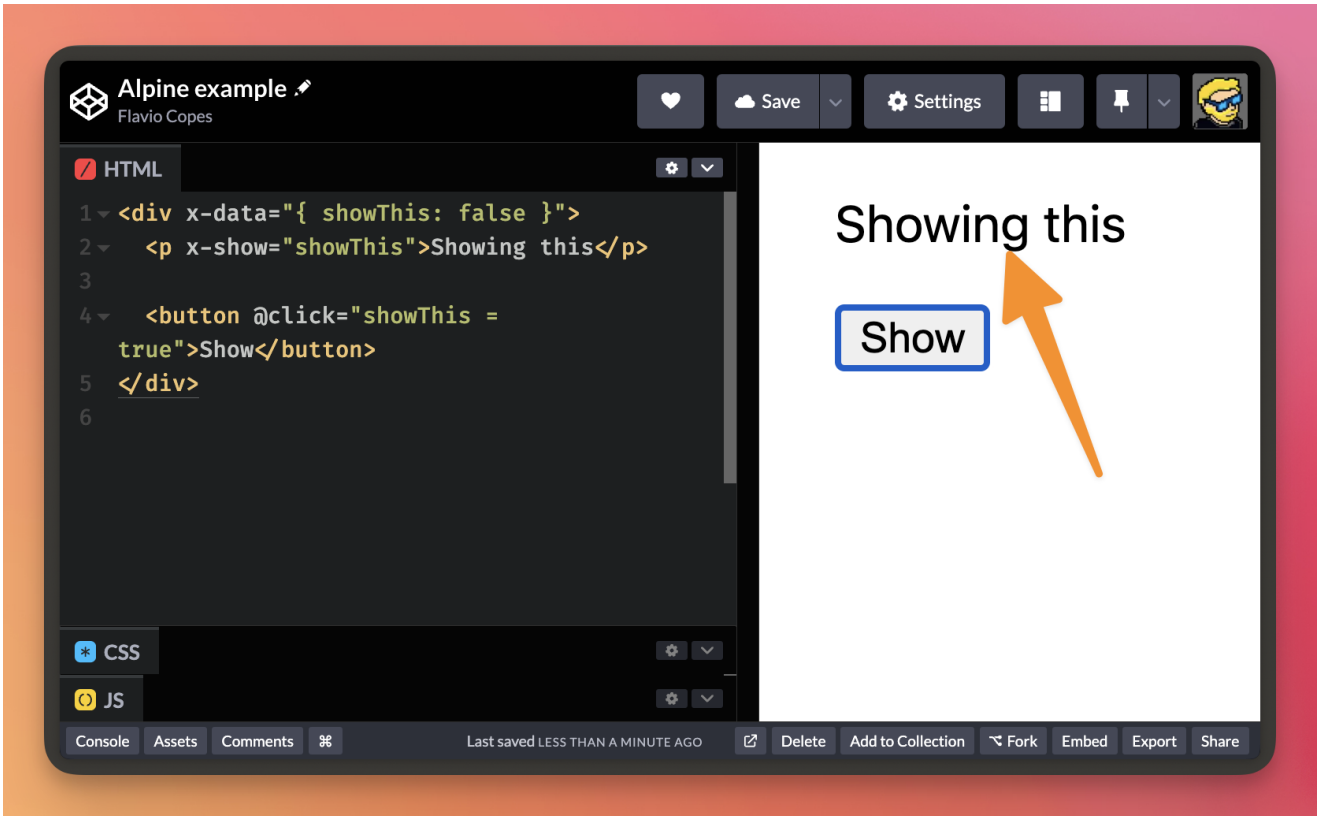
The User Interface we define inside the browser is event-based. People do things on the page, like clicking buttons, and we react to those events with an event handler to do *something*.

You can use the `@click` attribute to listen for clicks:

```
<button @click="showThis = true">Show</button>
```



Click the button, and `showThis` turns to `true`, and the `p` tag is displayed:



`@click` is a shorthand for `x-on:click`.

The `x-on` attribute lets you define any kind of event, but I like the shorthand form more.

We have others like

- `@keyup`
- `@keydown`
- `@dblclick`
- `@submit`
- ...

We can use modifiers on the event listener to change the behavior.

For example it's common sometimes to call `event.preventDefault()` when listening on a form submit event, to prevent the browser from sending the data to the server.

Something like:

```
<form>...</form>

<script>
        const form = document.querySelector('form')

        form.addEventListener('submit', event => {
          event.preventDefault()
          //..client-side logic...
          showThis = true
```

```
        })
</script>
```

With Alpine we would write

```
<form @submit.prevent="showThis = true">
  ...
```

Much simpler, right?

I've also seen it used as `@click.prevent` to prevent the default behavior on link clicks.

One modifier I used was `@click.outside` to hide a modal when clicking outside of it.

I used something like this:

```
<div x-show="showModal" @click.outside='showModal = false'>
  ....the modal content
</div>
```

to the div containing the modal.

Another one I used was `@keydown.enter` to detect the "enter key" being pressed.

Another useful modifier is `.window`. Using it we can bind an event listener to the window, super useful for example to detect pressing the esc key to close a modal window:

```
<div
  x-on:keydown.escape.window="showModalAddTask = false">
  ...
</div>
```

Without `.window`, you wouldn't be able to detect the keypress because unless the user is focused on an input field, key press events are sent to the global window.

Defining data

We've seen how to define data using `x-data`:

```
<div x-data="{ showThis: false }">
  <p x-show="showThis">Show this</p>
</div>
```

The string passed to `x-data` contains a JavaScript object, and we can add more variables into it:

```
<div x-data="{ showThis: false, name: '' }">
  <p x-show="showThis">Show this</p>
</div>
```

But it's kind of difficult to write JavaScript in this way, especially because it's not syntax highlighted and being inside a string it's easy to do errors and the editor cannot detect them for us.

You can do this instead:

```
Alpine.data('mydata', () => ({
  showThis: false,
  name: ''
}))
```

and then you associate `mydata` to an HTML tag:

```
<div x-data="mydata">

</div>
```

and in its children (and on the element itself, if you need) you have access to the variables defined in `mydata`:

```
<div x-data="mydata">
  <p x-show="showThis">Show this</p>
</div>
```

This is very useful if you need to add methods that you can call when an event occurs, or when you have more complex data processing.

Simple example, I have a shopping cart and in addition to the data, I have some utility functions to format this data nicely in my UI:

```
Alpine.data('shop', () => ({
  showCartOverlay: false,

  cart: JSON.parse(localStorage.getItem('cart')) || [],

  subtotal: 0,

  subtotalFormatted() {
```

```
    return this.formatAsPrice(this.subtotal)
  },

  formatAsPrice(price) {
    return Intl.NumberFormat('en-US', { style: 'currency', currency:
'USD' }).format(price)
  },

  recalculateSubtotal() {
    this.subtotal = 0
    this.cart.map((item) => (this.subtotal = (parseInt(this.subtotal *
100) + parseInt(item.price * 100)) / 100))
  }
})
```

…you got the idea.

Now I can use functions like this:

```
<div x-data="shop">
  ...
  <p class='ml-4' x-text='formatAsPrice(item.price)'></p>
</div>
```

It's much cleaner than adding all those functions inside `x-data`.

Looping

If you store some `x-data` variable as an array, you can loop through it and render a template multiple times using `x-for`, like this:

```
<ul x-data="shop">
  <template x-for='(item, index) in cart'>
    <li>
      <p x-text='item.title'></p>
      <p x-text='formatAsPrice(item.price)'></p>
      <button @click=`removeFromCart(index)`>
        Remove
      </button>
    </li>
  </template>
</ul>
```

The caveat is that `x-for` must be put on a `<template>` tag, and this must have a single child element (for example, wrap all tags in a `div` if you have more).

The `<template>` tag is a special tag used to hold content that's not rendered immediately on the page, but rendered using JavaScript (see MDN).

```
<div x-data="shop">
  ...
  <p class='ml-4' x-text='formatAsPrice(item.price)'></p>
</div>
```

Watching variables change

You can watch/listen for changes in an Alpine data variable, and run a function when a change happens.

You do so in the `x-init` attribute which contains code that's run when Alpine is initialized.

For example I have some HTML that belongs to a modal, and any time this modal is opened I want to clear the `name` data, so the input field is always empty:

```
<div
  x-data="{ name: '' }"
  x-init="$watch('showModal', () => { name = '' })">
  <form>
    <input name="name" x-model="name" />
  </form>
</div>
```

Stores

Stores are a state management solution you can use to create a centralized data "storage" for your UI.

To avoid confusion, data stored in a store is not "stored" anywhere. When you reload the page, the store is reinitialized.

But I found stores very useful for one particular thing: accessing a data variable using JavaScript in a clean way.

In my case I had to hide a modal after an htmx HTTP POST request.

Something like this:

```
document.addEventListener('htmx:afterRequest', (event) => {
  showModal = false
})
```

But since `showModal` was defined as `x-data="{ showModal: false }"` I had no way to access it using JavaScript.

So instead of initializing that variable with `x-data`, I created a store:

```
document.addEventListener('alpine:init', () => {
  Alpine.store('mystore', {
    showModal: false,
  })
})
```

And then I was able to access (and edit) this variable using:

```
document.addEventListener('htmx:afterRequest', (event) => {
  Alpine.store('mystore').showModal = false
})
```