How to learn every frontend framework at once

An ill-considered guide to surviving the inflationary frontend epoch

Sound check

Hi. I'm Sam Bleckley.

sambleckley.com

vistimo.com

Who this is for

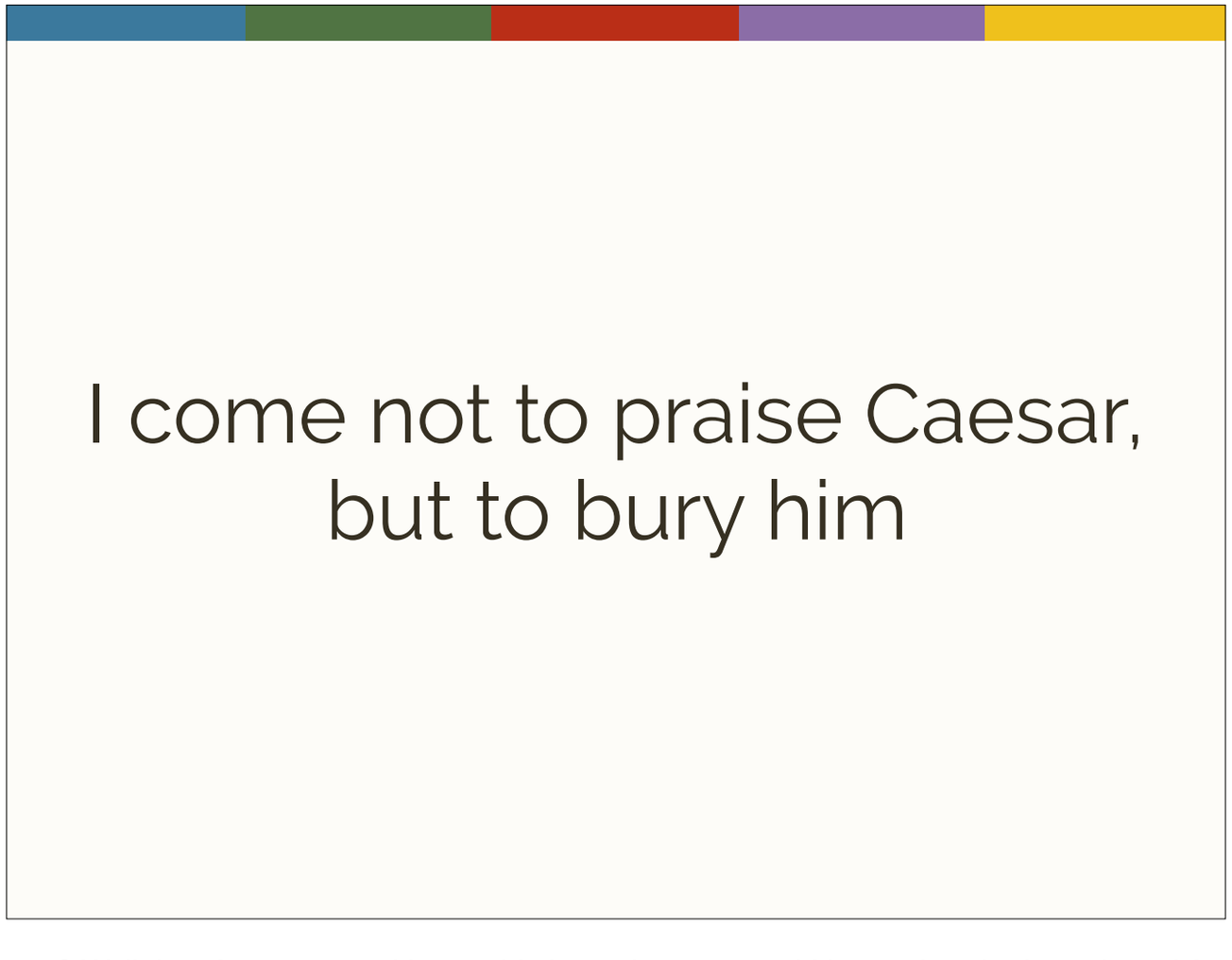
- If you already know 2+ frontend frameworks, you're not gonna get a whole lotta new knowledge dumped on 'ya. Sit tight, heckle me when I lie, and wait for the references slide.
- If you don't know HTML, CSS, and JS, you're gonna be so far outta your depth you won't notice you're drowning. Enjoy the ride, but learn the basics before attempting on your own.
- If you know JS, but have used zero or one frontend frameworks, now is the time to lean forward in interest.

It's OK if what I say seems obvious.

My goal is to say true things in a clear, memorable way. If they feel obvious in retrospect, I've done my job.

It's OK if what I say seems like gibberish

We're all at different places in our practice, and the words I use are not going to be the right words for everyone.



Oh, you've downloaded my slides, have you? Well, just for you: yes, I know this is a misquote; and I know that the intention of Antony's speech is *absolutely* to praise Caesar. Make of my knowledge, and this comment, what you will.

compare and contrast frameworks

I come not to ~~praise Caesar~~,
but to bury him

compare and contrast frameworks

I come not to ~~praise Caesar~~,
but to ~~bury him~~.

discuss their underlying principles.

If you understand the designer's goals, and the strategies available, then the architectural choices will seem obvious.

Our goal is to step inside the shoes of the author

Learn a system most intimately by climbing inside its creators

Learn a system most intimately by climbing inside its creators

Like some kind of serial killer.

Cast yourself back in time

cue wavy transition effect

Back in MY day

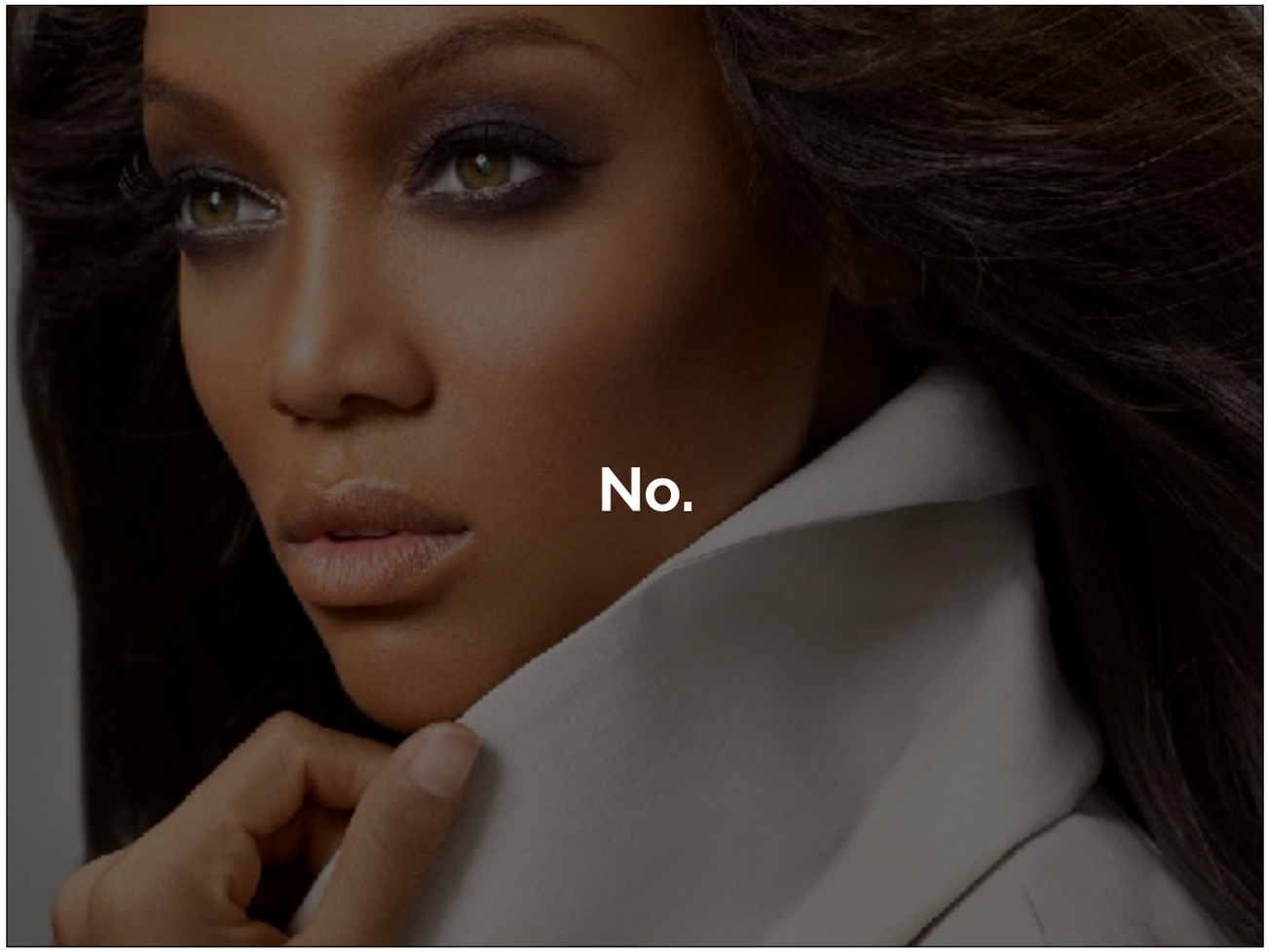
- Store all data in the DOM
- use jquery to manipulate the DOM and the data simultaneously
- The unit of interest is a whole page
- We wrote javascript uphill, both ways, in a cold cubicle, and we *liked* it.

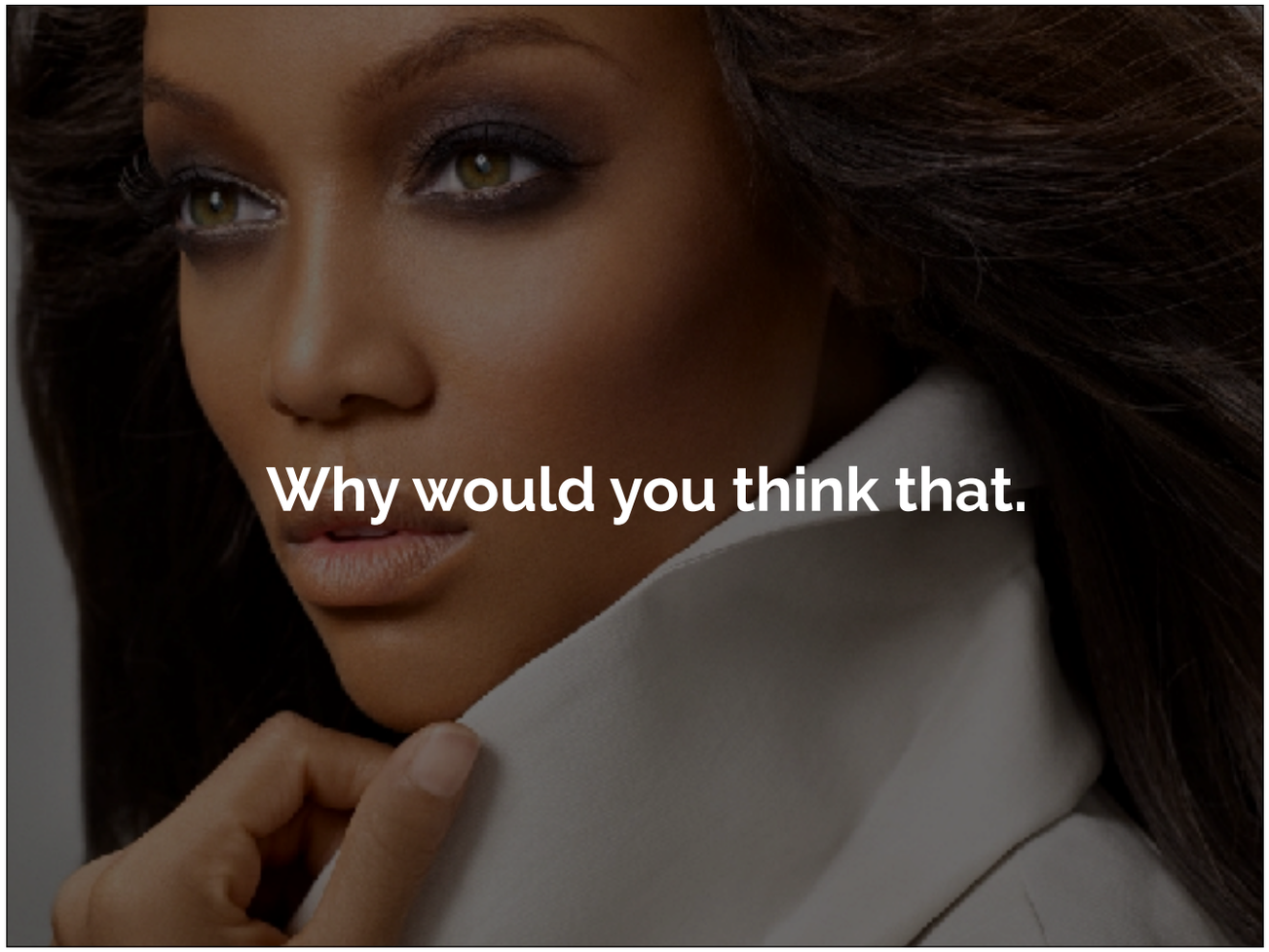
This was a bummer.

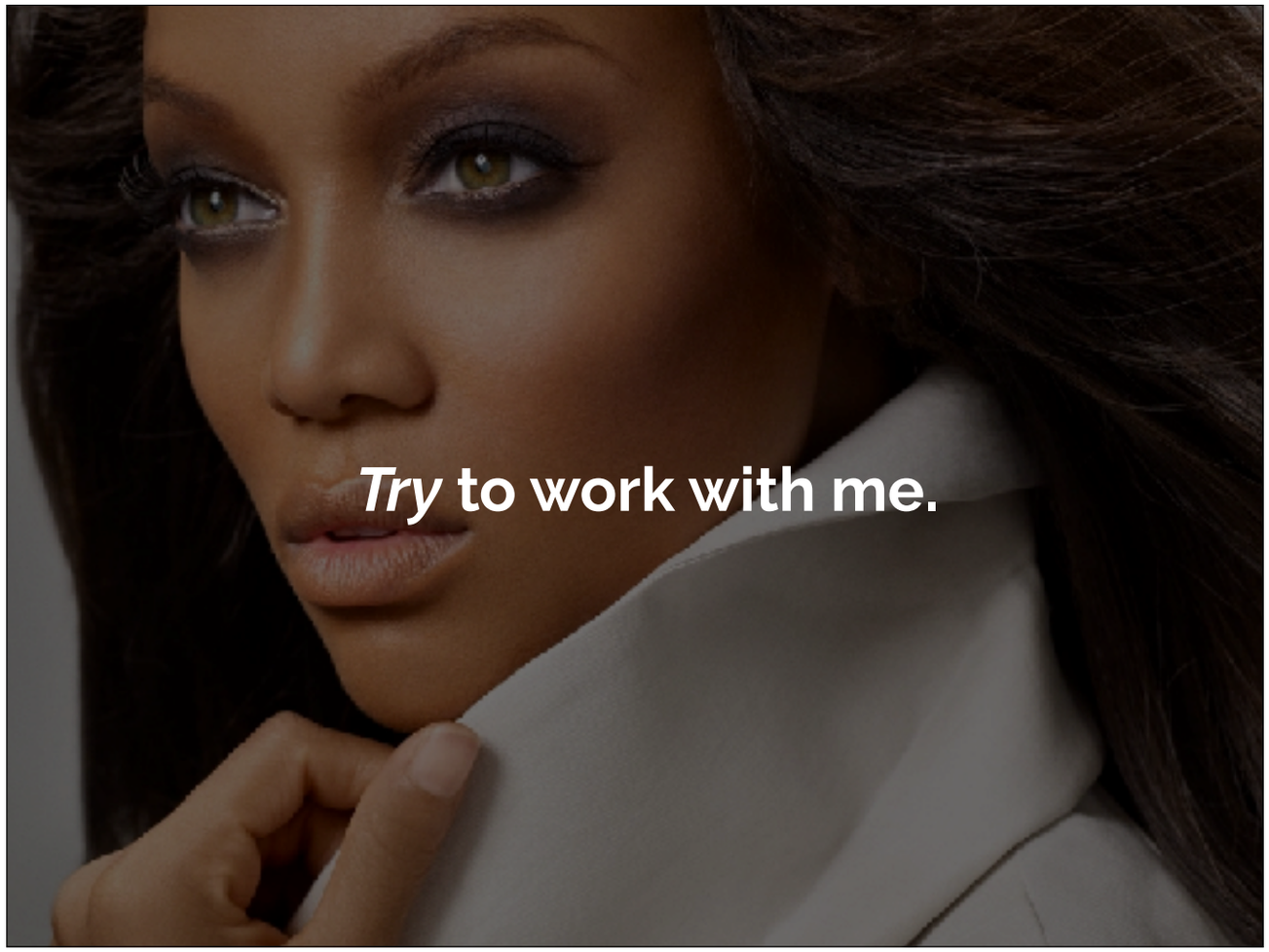
- if you want to change the way data looks, you must also rewrite the way it behaves
- the majority of dev time is spent in DOM-manipulation hell
- collaboration is tremendously hard

How do we fix it?

There's a model.


No.


Why would you think that.


Try to work with me.

This, by itself, is a bit of a revolution; FE in the past has loved to store its model right on the DOM. MVC is less'n 10 years old in the frontend com'ty.

An independent model is much easier to understand, to test, to update, to sync with a server. It's a good thing!

‘Model’ includes data:

- User: name, email, last login, permissions level
- Comment: author, date, text, context

But also actions:

- User: logout, change email, update password
- Comment: create, reply, like, flag

Including data and actions *doesn’t* mean it has to be object oriented — it can be a pure data structure and a family of functions that apply to that structure — but we’re gonna lump it all under the label ‘model’.

It can *do* all the *doing-stuff* and *know* all the *knowing-stuff* your user wants, just without any UI to show it to them.

Now that I have a model, I have to update the model AND update the DOM every time something changes. This is dumb. You're dumb, Sam.

But there's a problem.

THE FIRST BIG IDEA

Make the UI automatically sync to changes in a model

Great. How does this magic ‘thing’
know that the model’s been updated?

But there’s a problem.

Strategies

- A special ‘setter’ function — you may only update the model using the ‘setter’ (variant: ES6 setters)
- Special ‘getter’ functions; track every model dependency in the view (automatically or manually)
- Dirty checking: every time something *might* have changed, re-check every dependency in the view

Mix to taste.

“Manual” dependency tracking

```
let Person = EmberObject.extend({
  init() {
    this._super(...arguments);

    this.firstName = 'Betty';
    this.lastName = 'Jones';
  },

  fullName: computed('firstName', 'lastName', function() {
    return `${this.get('firstName')} ${this.get('lastName')}`;
  })
});
```

“Automatic” dependency tracking

```
var numbers = observable([1,2,3]);
var sum = computed(() => numbers.reduce((a, b) => a + b, 0));

var disposer = autorun(() => console.log(sum.get()));
// prints '6'
numbers.push(4);
// prints '10'

disposer();
numbers.push(5);
// won't print anything, nor is `sum` re-evaluated
```

Dirty checking: when ‘might’ something have changed?

- Build replacements for every async and user-driven operation (fetch, timeout, click etc) and expect users to use those things instead. Escape hatch ($apply) for the rest
- Monkey-patch every async/user-driven operation
- Monkey-patch every async operation, but in a Sophisticated Computer Science-y way (“zones”)

Some effective recipes

- Ember: special setters, special getters, manually list dependencies in the getters
- AngularJS, Aurelia: Dirty checking w/replacement async all the way
- Angular: Dirty checking w/“zones”
- React: special setters, plain-old getters
- Mobx: ES6 getters and auto-tracked ES6 getters

What about collaboration, reuse, testing, and other headaches?

Headaches like your attitude?

THE SECOND BIG IDEA

Break the UI into small, modular components

Easy to understand,
hard to enforce

Strategies

- Components
- Nested templates
- Dependency injection
- Directives
- Higher-order components

Components have won.

In old Ember and AngularJS codebases, you may still see raw templates, but in modern FEFs components rule the day.

Component routing for SPAs

Make them smaller
than you think

Even smaller than that.

Make them automatically

Architect so you can copy/paste them

III: SOME SMALL IDEAS

Upon picking up a new framework, once you've determined the recipes used for the Two Big Ideas, here are some other things you can look into to get a deeper sense of how the creators think.

Talking to remote APIs

What's a service?

Formalized in AngularJS and Angular, but conceptually useful in every framework; code unrelated to the DOM and likely shared with many components.

"TodoItemService", "AlertMessageService", etc.

We know Aurelia and AngularJS will offer their own HTTP services, because of their recipe for BI#1.

(Angular does too, but doesn't *need* to; you can use fetch as-is in Angular, but in AngularJS you'd need to add an $apply().)

SPA Routing

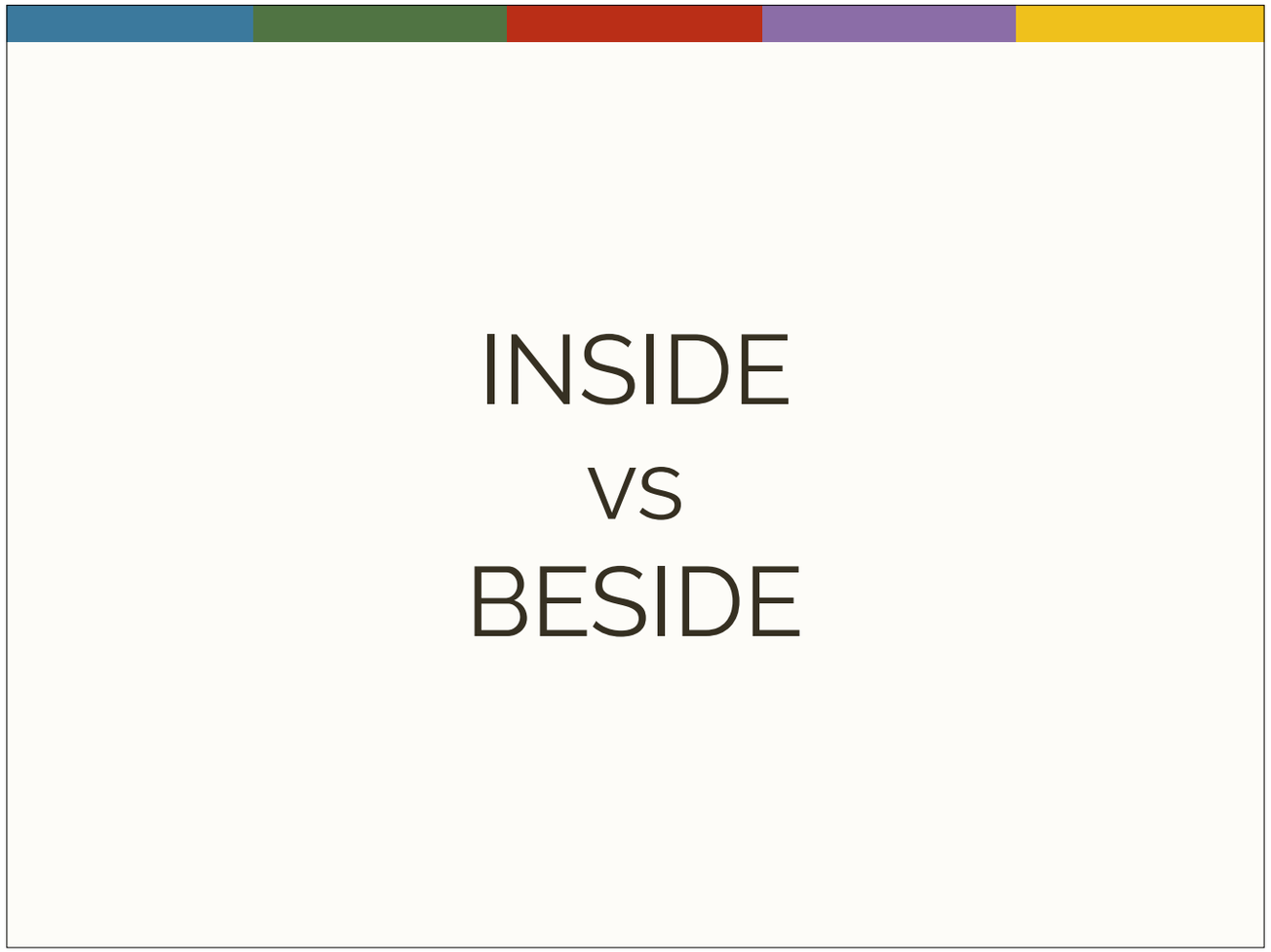
CSS Isolation

Sharing data across remote parts of the DOM

Animation

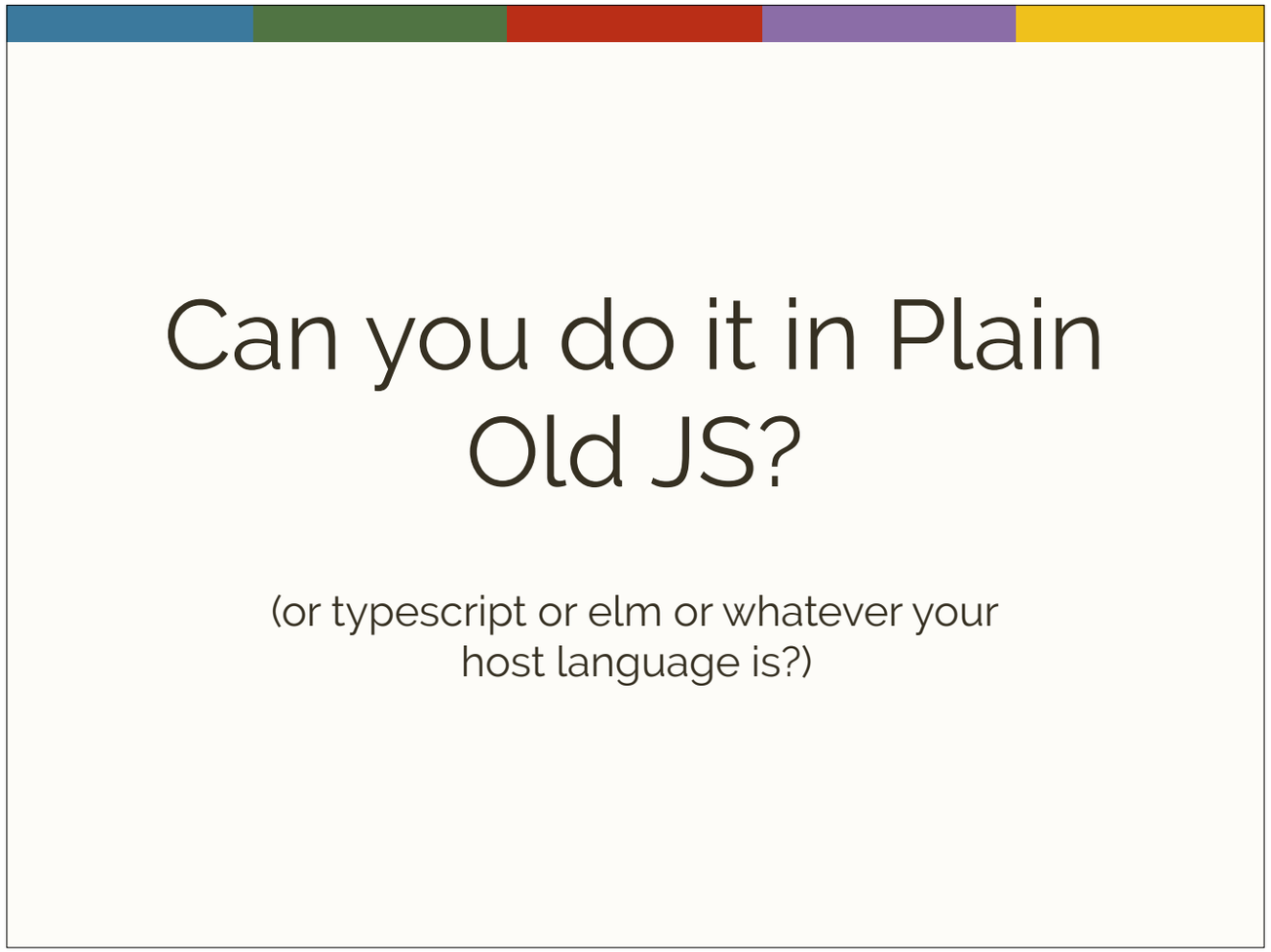
particularly of components leaving the DOM

Other Survival Techniques



Once you start using a framework — particularly the big megalithic ones like Ember and Angular — it starts to feel like you must do *everything* inside the framework, using its language and its tools.

This is an illusion.



If the thing you're trying to do *doesn't* have to do with the Two Big Ideas, maybe the framework isn't helping you.

JS is powerful, and has gotten MORE powerful in the past few years!

If you can be framework-agnostic, then when you, or your team, decide to change frameworks, you're dumping very little; if what you've built inside the framework is mostly small, modular components, then you already have a blueprint of what to rebuild — and if all the Fancy Pants stuff is in agnostic libraries, you only need to translate the wrapper.

MutationObserver

is pretty cool

```
function observe(mutations: MutationRecord[]) {
  mutations.forEach(m => {
    m.addedNodes.forEach(n => {
      if (!(n instanceof HTMLElement)) return;
      dispatch(n);
    });
    if (m.type === "attributes" && m.target instanceof HTMLElement) {
      if (m.target.dataset.phocusAction) {
        dispatch(m.target);
      } else {
        removeTrigger(m.target);
      }
    }
  });
}

var observer: MutationObserver;
export function startPhocus(elt: HTMLElement) {
  dispatch(elt);
  document.addEventListener("keydown", keydown);
  document.addEventListener("mousemove", setMoving);
  observer = new MutationObserver(observe);
  observer.observe(elt, {
    childList: true,
    subtree: true,
    attributes: true,
    attributeFilter: ["data-phocus-action", "data-phocus-on-mouseover"]
  });
  initTouchDeviceListeners();
}
```

(This is the directive/
attribute-based mixin pattern,
just ignorant of Angular)

What to take away from all of this

Frontend frameworks really only do two things: they sync the DOM to a data model, and they break a UI into modular components. If you understand the options for doing those things, you can make educated guesses about how any given framework will behave.

If you remember that a framework is a tool, not the whole universe, you can build in such a way to make it easy (if not necessarily fast) to migrate between frameworks if and when you need to.

References

AngularJS (also work, essentially unchanged, as first-order approximate descriptions of Aurelia and the rendering portions of Vue)

- https://blog.mgechev.com/2015/03/09/build-learn-your-own-light-lightweight-angularjs/
- https://teropa.info/build-your-own-angular/

React

- https://engineering.hexacta.com/didact-learning-how-react-works-by-building-it-from-scratch-51007984e5c5 (The author says “You can skip the first part.” That’s a lie. The bit on Fiber is interesting and useful, but it’s not the Heart of the Matter. Grok the recursive version first, then the queue’d version if you’re interested.)

Angular

- Somewhere there’s a google drive full of architectural plans for Angular. I can’t bloody find the thing, been looking for weeks.

Ember

- I don’t have a good resource for this. If anyone knows of a ‘build your own ember’-style tutorial, please let me know.

My work

- Libraries I’ve written, some in the style I’ve recommended in this talk: https://www.npmjs.com/~diiq

Don’t worry about trying to capture all this. These slides will be uploaded to the meet up.

IMO the BEST resource is a ‘build your own’ tutorial. These don’t walk you there *using* angular, or react, or whatever — they walk you through building a toy version of the whole framework. It’s less practical knowledge if you need to build a SPA by tomorrow at noon — but if you want to be a stronger frontend dev next *year*, it’s invaluable.