

Matrix

It's the Developer eXperience.

Matrix exists because of a bit of luck, and because of my own frustration with anything that slows me down while building software. There are many successful libraries out there. The difference in Matrix, in the end, is the D/X and productivity it supports.

Nothing I say today will make you go Wow. But if you build something with it, in a day or three you may well go Wow.

I find these slides convincing, but I doubt anything other than taking Matrix for a drive will convey the magic.

Standard Reactivity

- State dependency
- State change propagation
- Side effects off change

In my promo I talked about helping folks evaluate reactive libraries (RX), given a choice. ReactJS and Flutter give you a choice. They offer modest, local state tricks, but encourage us to look elsewhere for comprehensive state management.

That's crazy. Nothing gives developers a greater advantage than powerful state management. But in the early days, ReactJS actually bragged that they just handle the view. That did not work out, but let's not digress.

So in evaluating RX, on a few things we can relax. Every state tool:

- handles state dependency tracking in some form;
- takes care of reliable de-duplicated state propagation; and
- supports side effects outside the state propagation.

Nothing to see here. Let's move along.

Not Standard Reactivity

- Granularity (object properties, not view functions)
- The OO prototype model (ad hoc properties as needed)
- Transparency (no explicit subscribe or notify)
- In-place state management (object properties, view and domain, managed directly)
- *Glitch-free* state propagation
- Unlimited state scope for rules and event handlers
- Extensible to arbitrary non-reactive libraries (XHR, routing, Postgres,...)

Sorry for the long list, but this is why we are here.

Granularity means being able to specify the derivation of individual properties, such as the enabled state of a button, or the completed date of a Todo, and have them updated individually. Hoplon/Javelin does that, as did Common Lisp Garnet/KR, and I think Clojure Nodely does.

Ad hoc properties let us use standard objects such as an HTML <Div> and attach some state we find useful. Think “hooks”.

In-place state management means having view *and* domain properties themselves managed directly, versus in a Flux pattern secondary store. The original MobX does that.

Glitch-free is an esoteric but important concern. Glitches are transient state inconsistencies arising from the state engine’s implementation details. Reagent and RxJS suffer from glitches, MobX does not, and re-frame does not at a small performance cost.

Unlimited access to state is found in Flux, including re-frame, and Matrix. Flux-like stores are global, Matrix is global with natural scope.

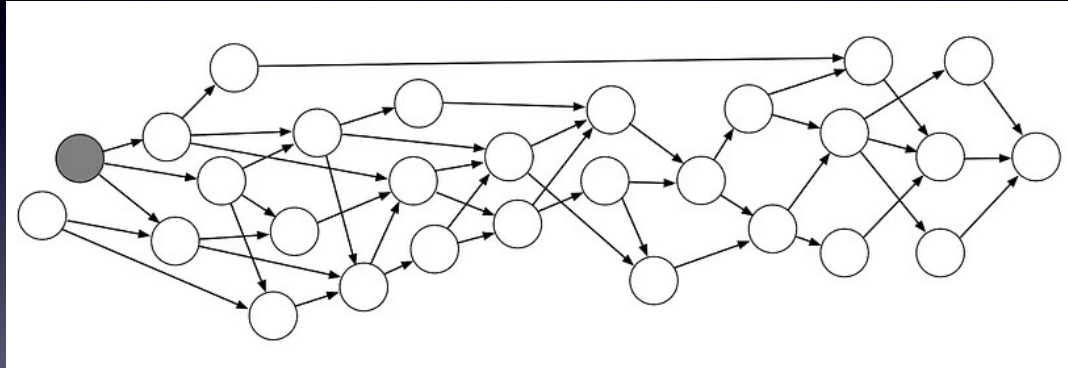
The extensible quality asks, can I use the reactive library to make my favorite, say, routing library work reactively, with more or less work?

By the way, apologies to anyone I leave out. But these are the things to look for.

Matrix

- ✓ Accurate, complete, de-duplicated state propagation
- ✓ Side effects off change, outside the DAG propagation
- ✓ Granularity
- ✓ Transparency
- ✓ In-place state management
- ✓ Glitch-free state propagation
- ✓ Unlimited scope

Pratītyasamutpāda



Everything is connected.

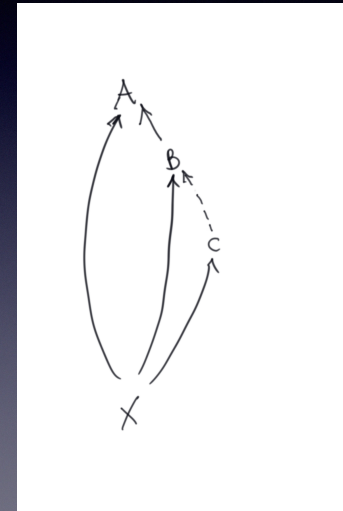
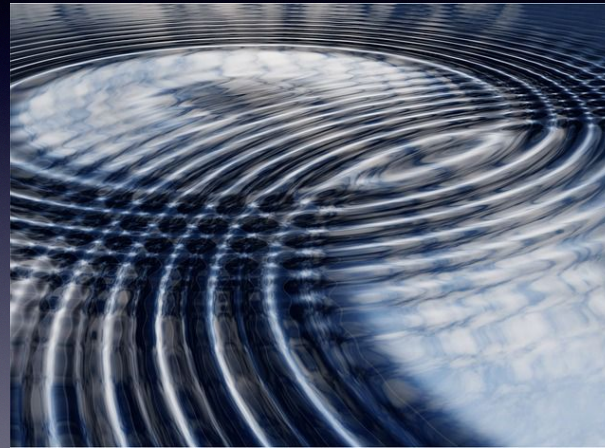
I said earlier that I thought it was crazy for UI frameworks to treat state management as a second class citizen.

Pratītyasamutpāda is the Buddhist concept of interconnectedness. Confirmed by physicists, who say a quantum event in a different galaxy can affect a billiards shot here on Earth. I read that in the NY Times Science section.

World interconnectedness is a cyclic graph. Computer applications can be less ambitious, stick to acyclic. Matrix, we will see today, lets us more or less painlessly discover the acyclic graph implicit in the problems we hope to automate, and transparently benefit from that graph to easily build apps in the face of unpredictable change.

Pictured is the RoboCup Simulation League DAG. Player client apps receive two JSON BLOB over a UDP socket every 100 ms. One is coaching from an optional coach bot. The other is a comprehensive sensory dump showing what a player bot can see. That one atomic input drives all player analysis, with streams of information branching and rejoining, forming interference patterns.

Glitches



And it is not just RoboCup that makes state hard with a stream of unpredictable state. Users do that. Apps with multiple inputs do that. This is why Facebook gave up on MVC and created Flux.

Why is changing state hard? We must update everything affected, of course. We would like also not to waste cycles on things not affected. And we must update things in the right order. There's the rub.

In this diagram, we see why. The solid arrows show known dependencies at some point in time. When X changes to X!, it can see that it needs to tell each of A, B, and C to recalculate. But in what order should it tell them? If it tells A first, A will use an obsolete value of B. Not good. So it must tell B before A. It can leave C until the end, it thinks, because of that dotted line. The dotted line means B *might* someday read C, but a conditional branch in B, based on the value X, did not access C.

So B gets updated, and something happens. Based on the new value X!, B decides to read C. But C is obsolete. B thus computes a value B? inconsistent with the new value X!. So does A, and we have A?. Now X finally tells C, who at this point knows B has used it. (Imagine the dotted line is now solid.) Now B! and A! get updated a second time, wasting cycles, but at least this time to values B! And A!, consistent with X!.

We wasted processing time, and if the RX offers side effects outside the DAG, we may have initiated an irretrievable action.

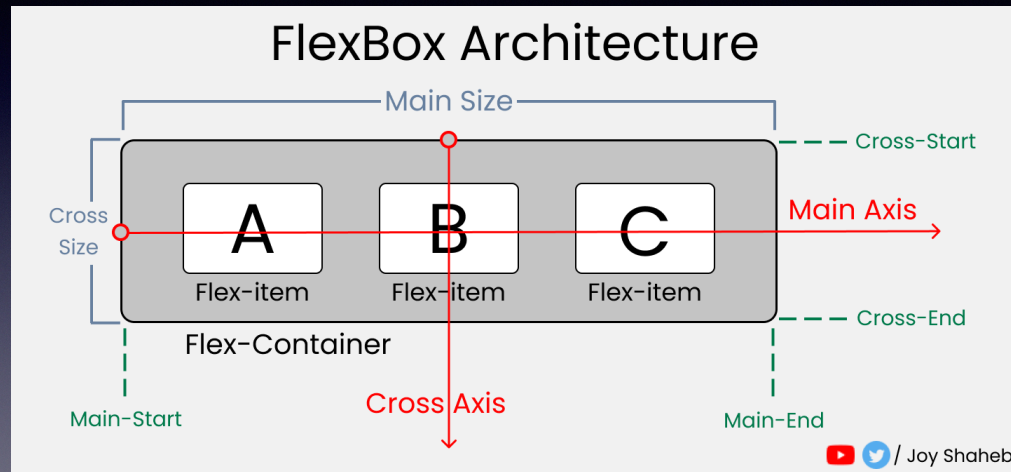
The Birth of Matrix

A hand-drawn diagram illustrating a fraction with annotations for alignment. The fraction is $\frac{23}{4567}$. Above the numerator '23', there is a double-headed arrow with a checkmark to its right, indicating a width measurement. To the left of the '2' in the numerator is a double-headed arrow with a question mark above it, indicating a question about the left margin. Below the denominator '4567', there is a double-headed arrow with a checkmark to its right, indicating a width measurement. Below the entire fraction, there is a long double-headed arrow with a question mark below it, indicating a question about the overall width or alignment of the fraction.

So how did we get to Matrix? Twenty-five years ago, my brilliant scheme for centering a numerator horizontally in a fraction collapsed in a heap. The specs were tough: just let the student type, make the software handle the layout seamlessly.

The target was struggling students, they did not need to be fighting the GUI to get their work centered.

We did not have Flexbox



This was before Flexbox.

We did not have MathJax

[illegible]

		x^3	$-2x^2$	$+4x$	-6
$x^2 - 3$)	x^5	$-2x^4$	$+x^3$		
				$-8x$	$+18$
	$-x^5$		$+3x^3$		
		$-2x^4$	$+4x^3$		
		$2x^4$		$-6x^2$	
			$4x^3$	$-6x^2$	$-8x$
			$-4x^3$		$+12x$
				$-6x^2$	$+4x$
				$6x^2$	-18
				$4x$	

And before MathJax. This was Mac OS 9 Quickdraw, and we had to tell it every screen coordinate.

My failed scheme:

Compute children then parent

$N\text{-offset-hz} = (F\text{-width} - N\text{-width})/2$

But $F\text{-width} = \max(N\text{-right}, D\text{-right})$

And $N\text{-right} = N\text{-offset-hz} + \text{string-width}(N.\text{value})$

Oops. Hello, Cycle

“Make everything as simple as possible, but no simpler.”
— Albert Einstein

But I could see that fraction layout was not a cyclic calculation.

Hello, Matrix

```
N.left = 0  
N.right = string-width( N.value )  
D.right = string-width( D.value )  
F.width = max( N.right, D.right )  
N.offset-h = (F.width - N.right)/2
```

There was no cycle, if we worked property by property.

Working property by property was also trivial. To write, to read, and to debug if it came to that.

Win #1: The automatic ordering of the functional paradigm, with values calculated JIT.

And the declarative quality.

And because we just decide a property, the rules are small and easy.

Fraction geometry


```
(make-mx
 :type :ratio
 :width (c-rule
         (apply max (map width (parts me))))
 :parts (c-rule
         (make-mx ;; numerator
          :value (c-input nil)
          :left 0
          :right (c-rule (string-width (value me)))
          :offset-hz (c-rule (/ (- (width parent)
                                   (width me)) 2)))
         (make-mx ;; denominator
          ...similar)))
```

Three years later...

tilton's algebra Log In/Register Welcome, Visitor.

[Home](#) > [Guided Practice](#) > [Numeric Fractions: Adding & Subtracting](#) Feedback

Congratulations. 1 2 3 4 5 6 7 8 9 10 11




Your work so far:

1.
$$\begin{array}{r} \frac{3}{10} + \frac{7}{16} \\ 48 + 70 \\ \hline 160 \\ 118 \\ 160 \\ 59 \\ \hline 80 \end{array}$$

Simplify if possible:

$$\frac{\frac{3}{10} + \frac{7}{16}}{\frac{48+70}{160}}$$
$$\frac{118}{160}$$
$$\frac{59}{80}$$

simplest form



<http://tiltonsalgebra.com/#>

Not Just the UI





<https://www.robocup.org/leagues/23>


This GUI app is just a playback tool we can use to visualize games.

Games are played by clients exchanging JSON sensory input and player reactions every 100ms over a UDP socket.

AskHN Who's Hiring Browser

 Ask HN: Who Is Hiring? [Pro tips](#)

October, 2023  Total jobs: 149

Filters 

- ☒ REMOTE ☐ ONSITE ☐ INTERNS ☐ VISA
- ☐ Starred ☐ Noted ☐ Applied ☐ Excluded ☒ Unreviewed

Search

Title Only

Regex for title search

Full Names


clojure or lisp or scheme

☐ match case ☒ allow or/and [help](#)

Sort [Creation](#) [Stars](#) [Company](#)

Jobs: 2 Show: 42 [Collapse All](#)

Strategic Blue | London, UK | Clojure Developer | REMOTE (UK) | VISA

☆☆☆ ☐ Applied Notes 

Domain: FinOps / Cloud Pricing Culture: Transparent / Learning / Diverse / Collaborative / Pairing Stack: Clojure / ClojureScript (re-frame/reagent) / AWS Fargate / AWS CDK / Event Sourcing Apply: tech-hn@strategic-blue.com

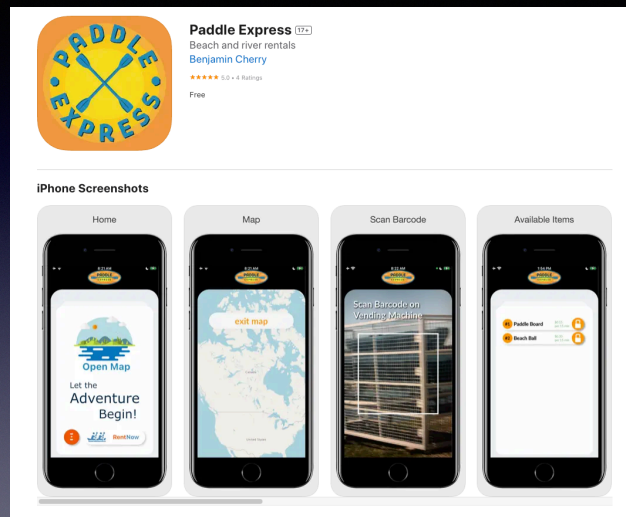
We are cloud FinOps experts, helping our customers to reduce cloud spend using innovative commitment trading techniques. We have several internal products, and a customer-facing web portal.

We are proud of our culture of openness and transparency, with an emphasis on learning. We operate autonomous self-organised teams, and we value everyone's ideas and opinions. We do a lot of pair programming to share knowledge and experience. We currently work fully remote, with occasional visits to our London office for collaborative work that helps us get to know each other better.

We seek enthusiastic developers who either know or would like to learn Clojure. We currently have a diverse team of nine full-stack devs, but we have plenty of front-end and back-end work, so we can accommodate people who have a preference.

Built with the JS port of Matrix.

Paddle Express



A Flutter/MX App

Back to Properties

```
(defn td-completed [todo]
  (mget todo :completed))

(defn clear-completed-button []
  (fx/visibility
   {:visible (cF (when-let [tds (todo/app-todos (my-app))]
                   (some td-completed tds))))
   (fx/text-button
    {:onPressed (mswap! (mget (fasc :app) :todo-list) :kids
                        (partial remove td-completed))}
    (fx/text "Clear done"))))
```

The TodoMVC spec required the “Clear completed” button to be hidden unless at least one Todo has been marked “completed”.

The “visible” rule runs when a Todo is created or deleted, because it reads the “app-todos” property of the app. It also runs if a Todo it read during the “some” traversal changes.

We include also how the button works, once visible. It will remove all Todos marked completed.

Reactivity and Persistence

```
(deftype ToDo []  
  :extends Model  
  PObserver  
  (observe [this prop me new-value prior-value cell]  
    (storage/td-rewrite  
      (select-keys @td [:stg-id :created-at :title :completed])))))
```

OOP is not all bad.

Here is an example of the “extensibility” mentioned earlier. The Hive package is wrapped by a bit of glue I called “storage”. Storage uses Matrix reactive objects, so the rest of the app can act as if Hive were reactive.

The OO Prototype Model

```
(defn todo-list-item [todo]
  (fx/visibility
    {:visible (cF (mget me :selected?))}
    {:selected? (cF (case (fmu :todo-routing :selection)
                        :all true
                        :active (not (td-completed todo))
                        :done (td-completed todo))))}
    (fx/container
      (fx/gesture-detector
        {:onDoubleTap (as-dart-callback []
                                          (mset! me :editing? true))}
        {:name :item-control
         :editing? (c-input false)}
        (cF ;; hello responsive UI: widget children vary appropriately
          (if (mget me :editing?)
            (to-do-editor me todo)
            (to-do-display todo)))))))
```

We can add custom properties in a second map as needed.

The TodoMVC spec requires a radio group choice to see All Todos or only those that have been completed, or those still Active.

We make a Todo visible only if its completed status matches the user selection.

Unrelated, we see the UI swapping in a Todo editor if they double-click an item.

Navigation

Utilities `fasc` and `fm*` search the DAG
From one model to any other so we can act on it.

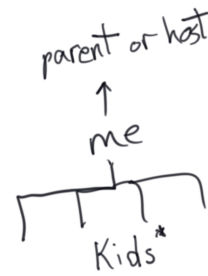
```
(defn to-do-editor [me todo]
  (fx/list-tile
    {:title (cF (fx/focus-scope
      { :onFocusChange
        (cF (as-dart-callback [focused?]
          (when (not focused?)
            (mset! (fasc :item-control) :editing? false))))))
      (fx/text-field {}
        { :name :todo-editor
          :value (cI title)})))))
```

The TodoMVC spec says to terminate editing and save any changes if they move the focus off the Todo.

Simply changing “editing?” to false handles everything, reverting the edit widget to a display widget, and updating the Hive DB.

Navigation and Natural Scope

Each model



A tree of models



Navigation is easy. MX is just a tree, although with Flutter we also have “delegate” objects in model properties other than “children”. Those are “host”ed by models.

How it works (simplified)

```
(defn cell-get [c]
  (let [prior-value (c-value c)
        new-value (ensure-value-is-current c :c-read nil)]
    (when *depender*
      (record-dependency c))
    new-value))

(defn compute-cell [c]
  (when (some #{c} *call-stack*)
    (mx-throw "COMPUTE_CYCLE_DETECTED"))
  (binding [*depender* c
            *call-stack* (cons c *call-stack*)]
    (unlink-from-used c)
    ((c-formula c) c)))

(defn cell-change [c new-value]
  (let [prior-value (c-value c)]
    (cell-pulse-bump c)
    (md-prop-value-store (c-model c) (c-prop c) new-value)
    (propagate-to-callers c callers)
    (cell-watch c prior-value)))
```

Now when sth changes, the engine knows exactly what to recalculate. Without us explicitly subscribing. Or notifying.

EVL: what does subscription/notification look like in a framework you might use. To get a derivation, do you need to code up a new little subscription? Does that bookkeeping ever let up?

Transparency

- Subscriptions: transparent and automatic
- Notify: transparent and automatic
- Navigation: modest hinting a good idea; definite risk of cycles. Hinting avoids cycles. Otherwise, just say what we seek.

Moral: we can have our one-way, acyclic DAG and ignore it, too.

When I Use Matrix, and When Not

Matrix helps build any application involving:

- an interesting amount...
- ...of long-lived state maintained over time; and
- a stream of unpredictable inputs.

Examples:

- a human working on Algebra problems in a UI
- A bot playing RoboCup with a UDP game server

Counter-example:

- ETL/ELT: each record handled in isolation.

I do not use Matrix for ETL/ELT because the large amount of unpredictable data is not used to maintain a long-lived summary analysis. Each record is processed independently of the others.

Matrix Evolving

Every new project brings new ideas.

- Flutter's propensity for async led to the `:async?` option.
- This summer a stream analyzing project led to a cell “freeze” capability.
- Preparing this talk led to a relaxation of a rule enjoining mutation of a formula.
- An app using a physics engine led to dependencies on hash-table entities.

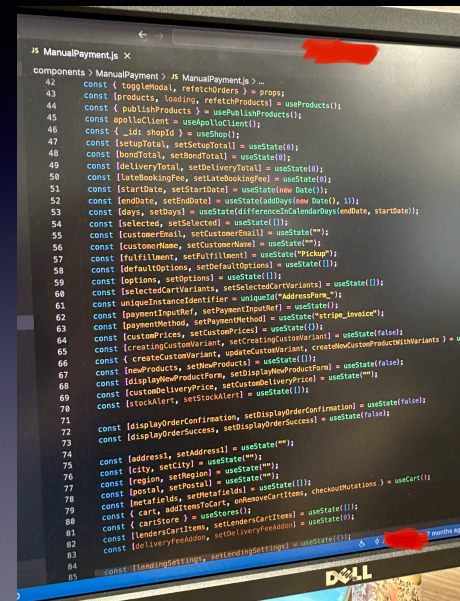
There is more, for another day...

- Formulas become inputs.
- Formulas become constants.
- Lazy cells.
- Synapses.
- Cells deciding to freeze.
- Without-dependency,
- Unchanged-if.
- Standalone cells.
- Client queue handler.

Summary: the Matrix Difference

Granularity	Read and write domain/view <i>properties</i>
Transparency	No explicit subscribe or notify; dependencies identified automatically
In-place state management	Direct maintenance of domain and view properties; no separate store
Navigation 1	Formulas can read any app state
Navigation 2	Event handlers can modify any app state
Extensibility	Readily applied to external libraries

Where Flux Went Wrong



```
42 const { toggleModal, refetchOrders } = props;
43 const { products, loading, refetchProducts } = useProducts();
44 const { publishProducts } = usePublishProducts();
45 const apolloClient = useApolloClient();
46 const { _id: shopId } = useShop();
47 const { setupTotal, setSetupTotal } = useState(0);
48 const { bondTotal, setBondTotal } = useState(0);
49 const { deliveryTotal, setDeliveryTotal } = useState(0);
50 const { lateBookingFee, setLateBookingFee } = useState(0);
51 const { startDate, setStartDate } = useState(new Date());
52 const { endDate, setEndDate } = useState(addDays(new Date(), 1));
53 const { days, setDays } = useState(differenceInCalendarDays(endDate, startDate));
54 const { selected, setSelected } = useState(1);
55 const { customerEmail, setCustomerEmail } = useState("");
56 const { customerName, setCustomerName } = useState("");
57 const { fulfillment, setFulfillment } = useState("Pickup");
58 const { defaultOptions, setDefaultOptions } = useState(1);
59 const { options, setOptions } = useState(1);
60 const { selectedCartVariants, setSelectedCartVariants } = useState(1);
61 const { uniqueInstanceIdentifier, setUniqueInstanceIdentifier } = useState(1);
62 const { paymentType, setPaymentType } = useState("stripe_invoice");
63 const { paymentMethod, setPaymentMethod } = useState(1);
64 const { customPrices, setCustomPrices } = useState(false);
65 const { creatingCustomerVariant, setCreatingCustomerVariant } = useState(false);
66 const { createCustomerVariant, setCreateCustomerVariant } = useState(false);
67 const { newProducts, setNewProducts } = useState(false);
68 const { displayNewProductForm, setDisplayNewProductForm } = useState(false);
69 const { customDeliveryPrice, setCustomDeliveryPrice } = useState("");
70 const { stockAlert, setStockAlert } = useState(1);
71 const { displayOrderConfirmation, setDisplayOrderConfirmation } = useState(false);
72 const { displayOrderSuccess, setDisplayOrderSuccess } = useState(false);
73 const { address, setAddress } = useState("");
74 const { city, setCity } = useState("");
75 const { region, setRegion } = useState("");
76 const { postal, setPostal } = useState(1);
77 const { latitude, setLatitude } = useState(1);
78 const { longitude, setLongitude } = useState(1);
79 const { cart, addItemsToCart, removeCartItems, checkoutMutations } = useCart();
80 const { cartItems, setCartItems } = useState(1);
81 const { cartItems, setCartItems } = useState(1);
82 const { deliveryFeeAddon, setDeliveryFeeAddon } = useState(1);
83 const { deliveryFeeAddon, setDeliveryFeeAddon } = useState(1);
84 const { deliveryFeeAddon, setDeliveryFeeAddon } = useState(1);
85 const { deliveryFeeAddon, setDeliveryFeeAddon } = useState(1);
```

"Just one more state variable bro. Just one more hook and the page will have everything it needs. Just one more state variable please bro. Bro? Add one more state variable please bro"

This makes me wanna go back to MVC and separation of concerns. Where did we go wrong.

— Heard on Twitter Oct. 2023