GLEAM GATHERING 2026

INSIDE THE LUSTE RUNTIME

👋 HI, HELLO!

- i'm ~~yoshie~~ rebecca
- i work on this little Gleam library called Lustre!
- joined Hayleigh/Lustre almost exactly 1 year ago 🎉
- mostly work on the internals

LUSTRE IS AN HTML TEMPLATING ENGINE

```
pub fn view(_model: Model) {
  html.div([], [
    html.h1([], [html.text("Hi 👋 I'm Rebecca!")]),
    html.p([class("pronouns")], [html.text("(she/her)")]),
    html.nav([class("social-links")], [
      social_link(
        "https://bsky.app/profile/becca.monster", "Bluesky", icons.bluesky()
      ),
    ])
  ])
}
```

LUSTRE IS A FRONTEND APPLICATION FRAMEWORK

```
pub fn init(_flags: Nil) -> #(Model, Effect(Message)) {
  // ... initial application state ...
}
```

LUSTRE IS A FRONTEND APPLICATION FRAMEWORK

```
pub fn init(_flags: Nil) -> #(Model, Effect(Message)) {
  // ... initial application state ...
}

pub fn update(model: Model, message: Message) -> #(Model, Effect(Message)) {
  // ... how does the state change when `message` happened? ...
}
```

LUSTRE IS A FRONTEND APPLICATION FRAMEWORK

```
pub fn main() {
  let app = lustre.application(init:, update:, view:)
  let assert Ok(_) = lustre.start(app, "#app", Nil)
}

pub fn init(_flags: Nil) -> #(Model, Effect(Message)) {
  // ... initial application state ...
}

pub fn update(model: Model, message: Message) -> #(Model, Effect(Message)) {
  // ... how does the state change when `message` happened? ...
}
```

LUSTRE LETS YOU RUN YOUR CODE WHERE IT BELONGS

```
pub fn main() {
  let app = lustre.application(init:, update:, view:)
  let assert Ok(_) = lustre.start_server_component(app, Nil)
}
```

Run interactive components on the server too!

LUSTRE LETS YOU RUN YOUR CODE WHERE IT BELONGS

```
pub fn main() {
  let app = lustre.application(init:, update:, view:)
  let assert Ok(_) = lustre.start_server_component(app, Nil)
}
```

Run interactive components on the server too!

Now also available with supervision support!

LUSTRE LETS YOU RUN YOUR CODE WHERE IT BELONGS

```
pub fn main() {
  let app = lustre.application(init:, update:, view:)
  let assert Ok(_) = lustre.start_server_component(app, Nil)
}
```

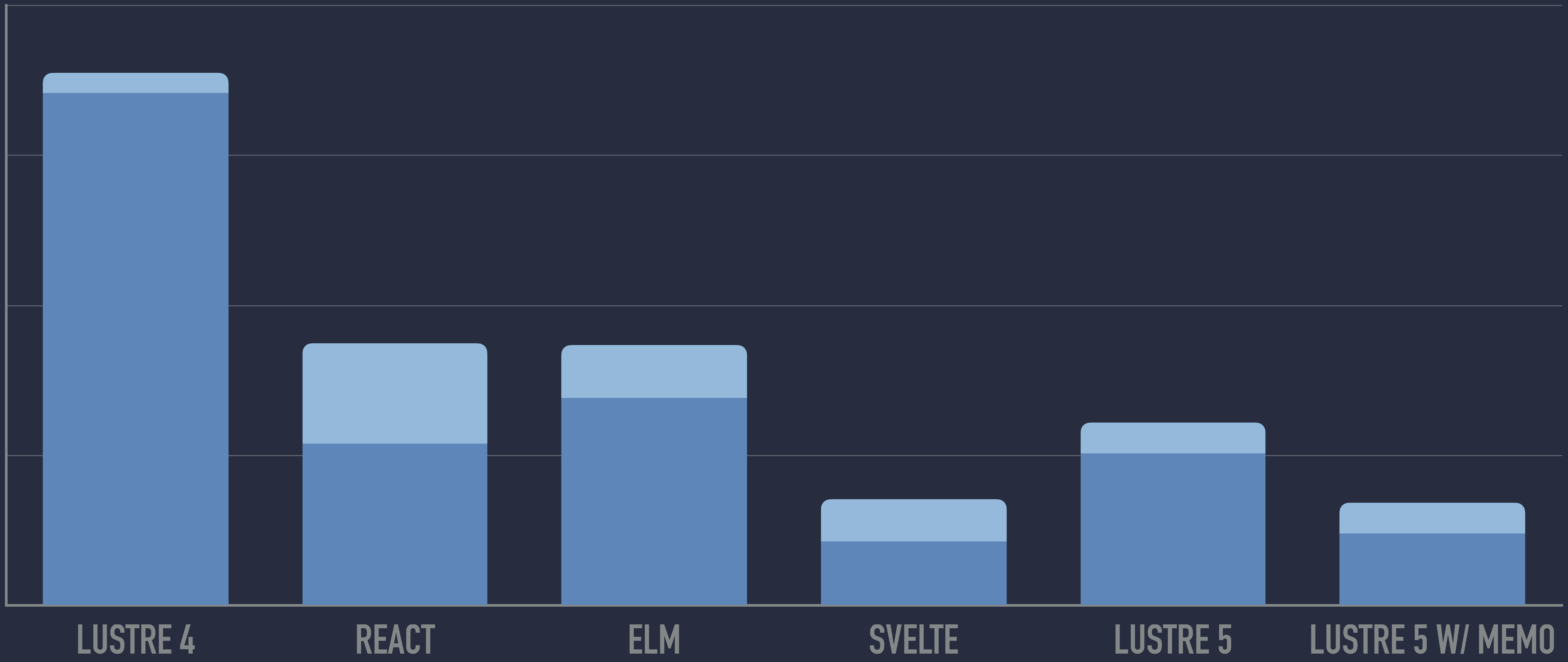
Run interactive components on the server too!

Ask your hex repository now for lustre@5.6.0!

Now also available with supervision support!

that is **lustre@5.6.0!**

WE'RE DOING ALRIGHT I THINK



HOW DO WE DO THIS?

- Almost entirely* Gleam!
- Almost entirely* pure functional code!

*terms and conditions apply :-)

HOW DO WE DO THIS?

- Almost entirely* Gleam!
- Almost entirely* pure functional code!

*terms and conditions apply :-)

LUSTRE IS 50% FASTER NOW!

- Decision Trees
- IIFE removal
- Fast record updates
- Standard library improvements
- Improved equality checking
- Prefer pattern matches over use

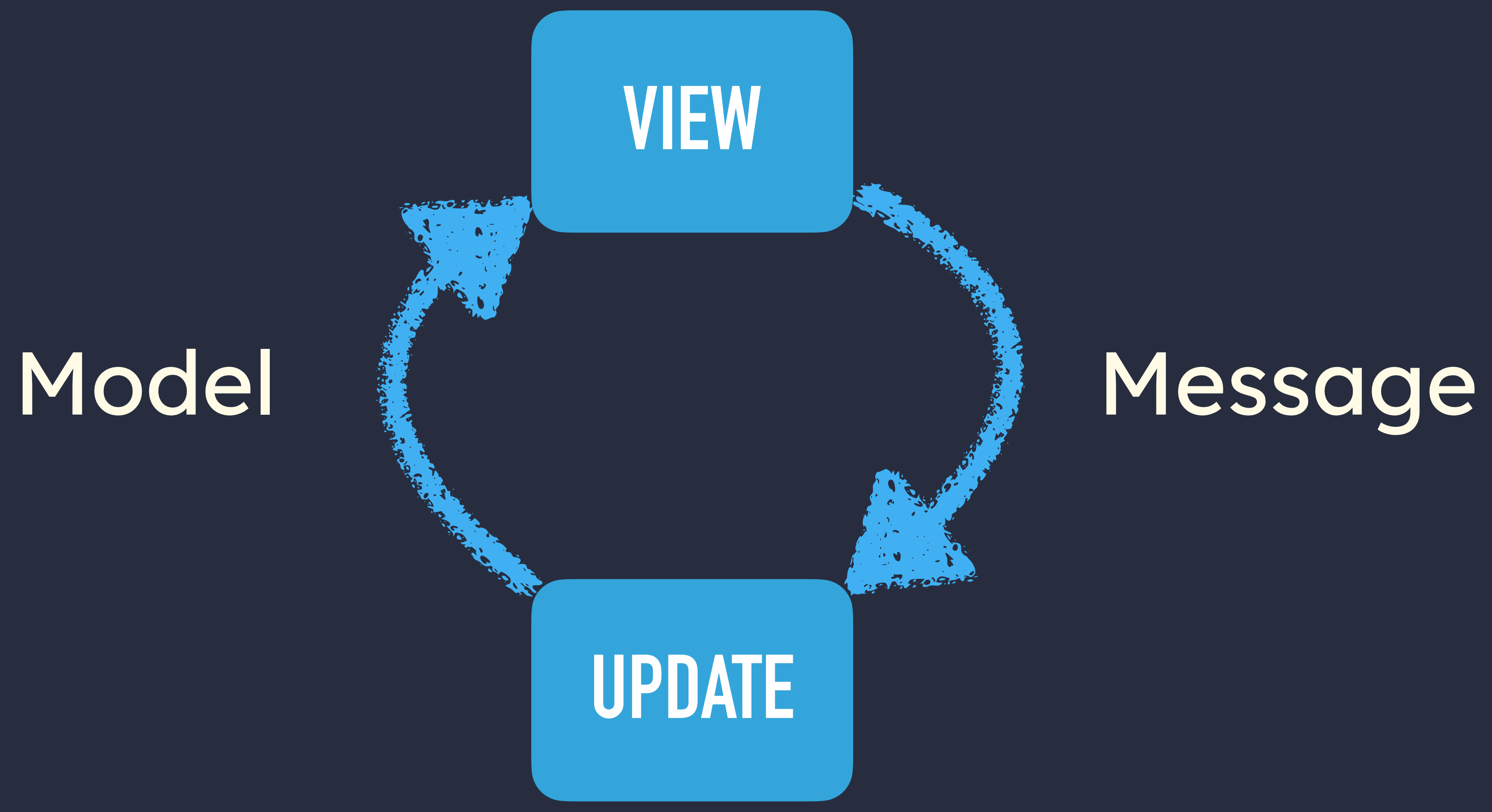
thanks Gears and Jak and all contributors!!! 💜 💕

I THINK THAT'S SOOO COOL!

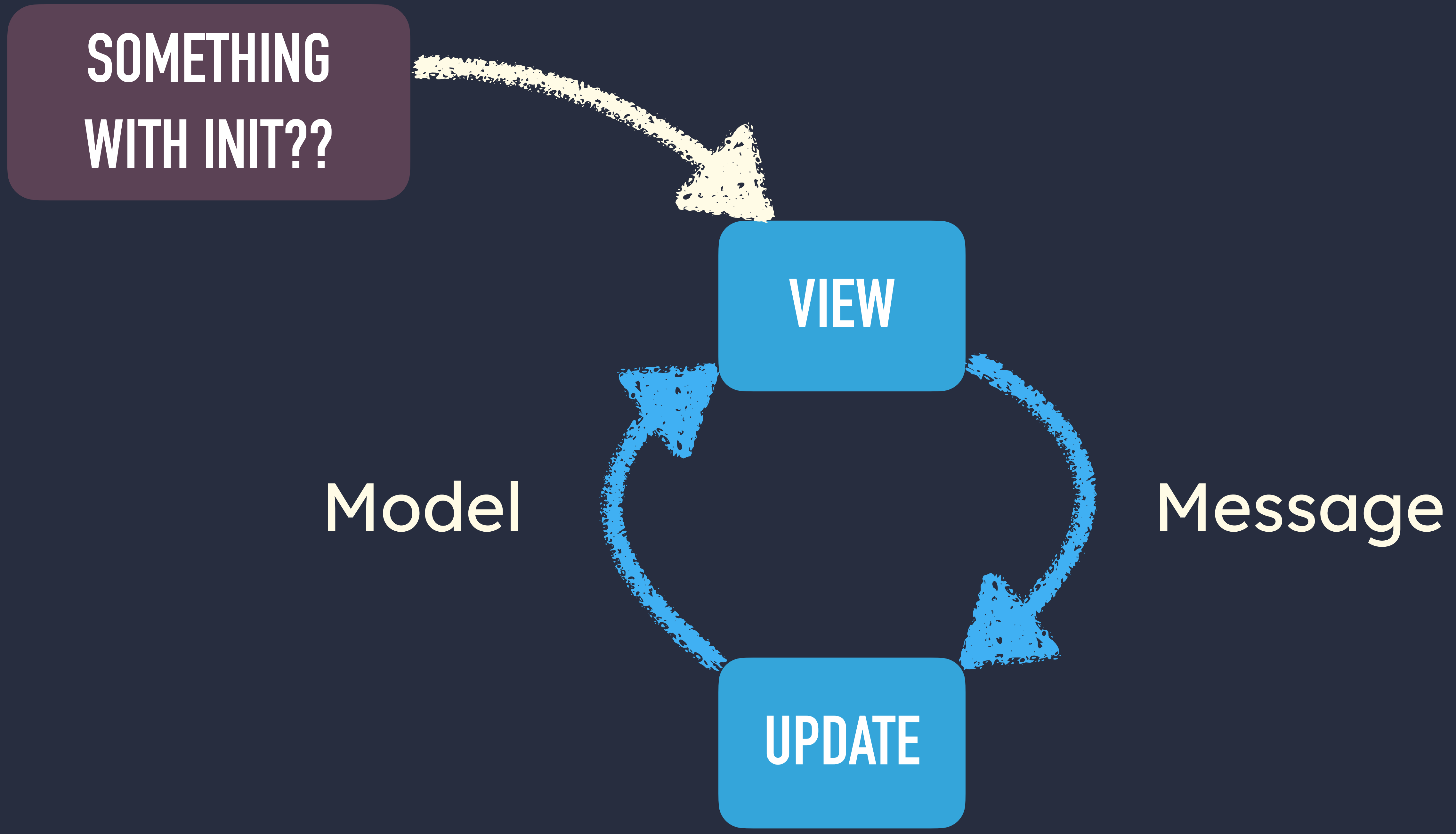
WHAT THIS TALK IS

- advanced lustre tutorial
- skip over all the basics
- immediately go to break the abstraction
- understand how a modern frontend framework works in general

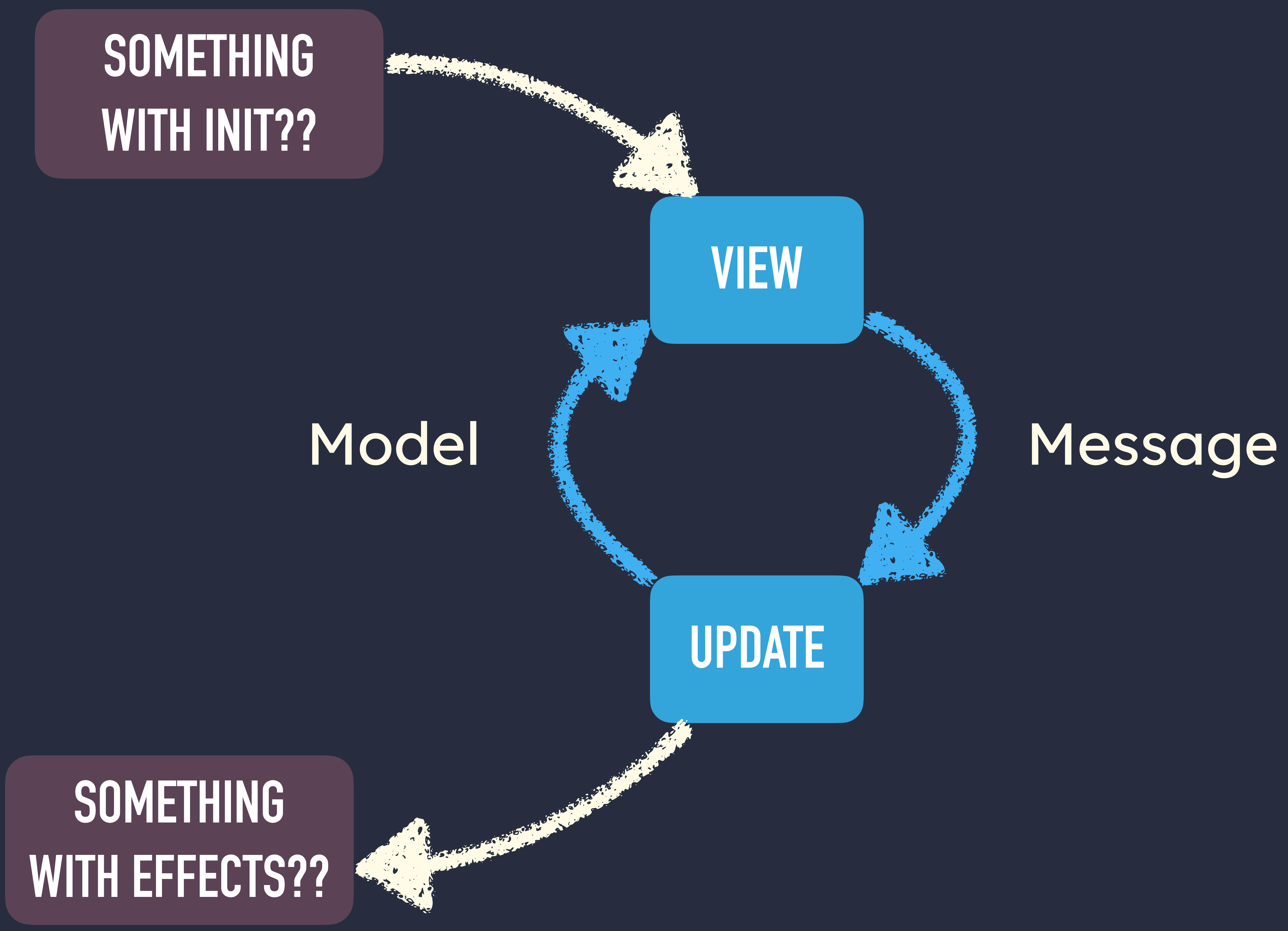
THE APPLICATION LIFECYCLE



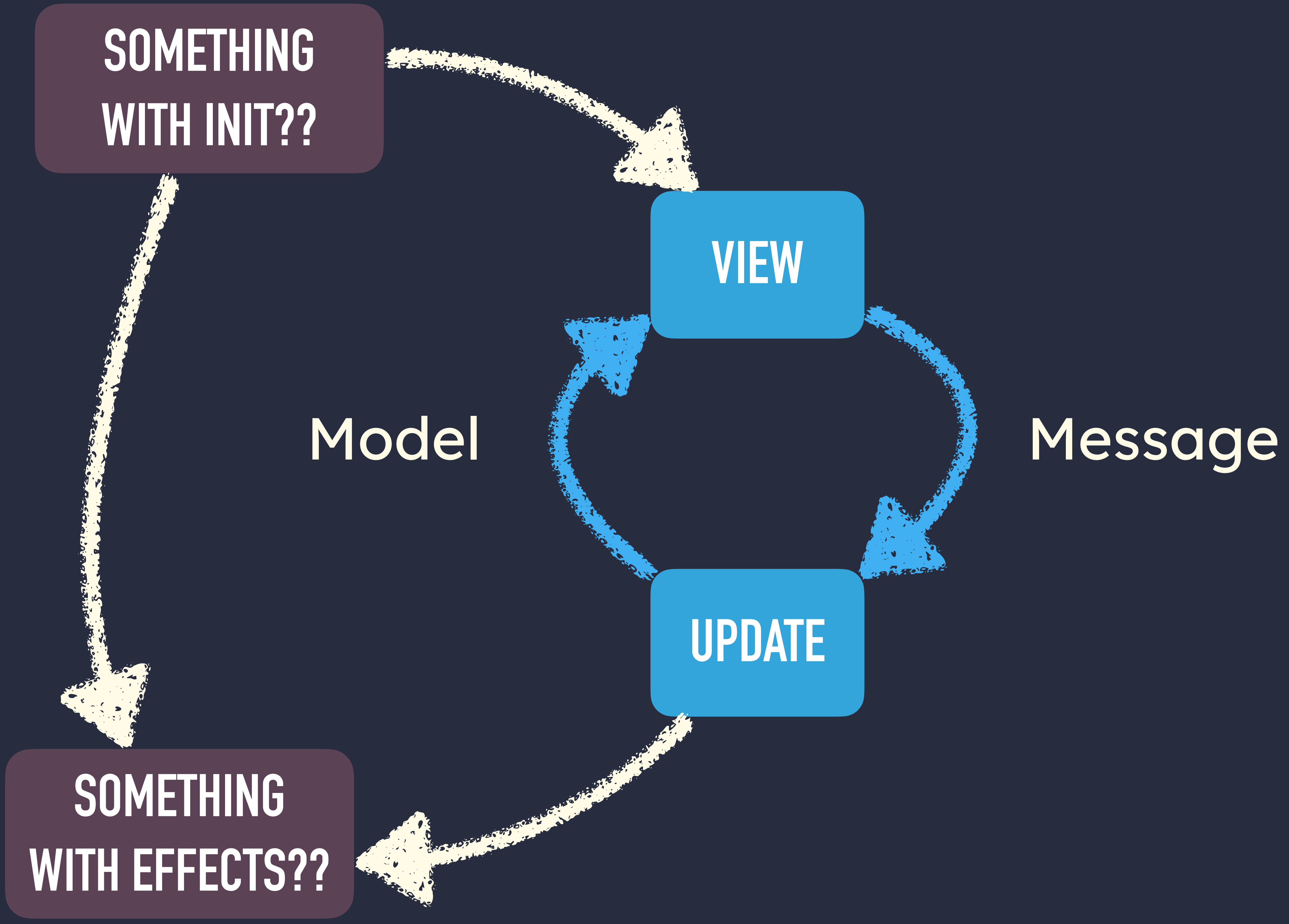
THE APPLICATION LIFECYCLE



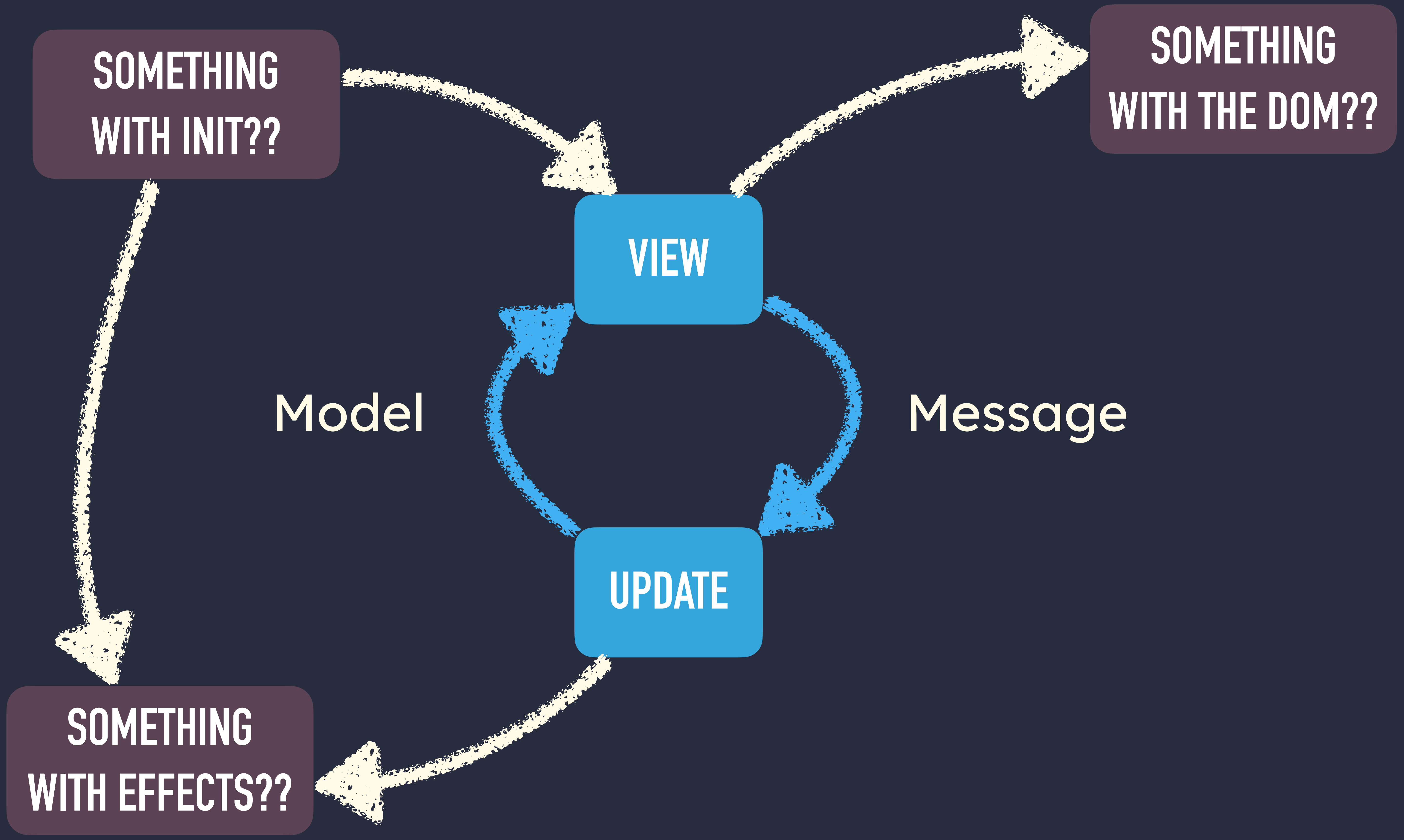
THE APPLICATION LIFECYCLE



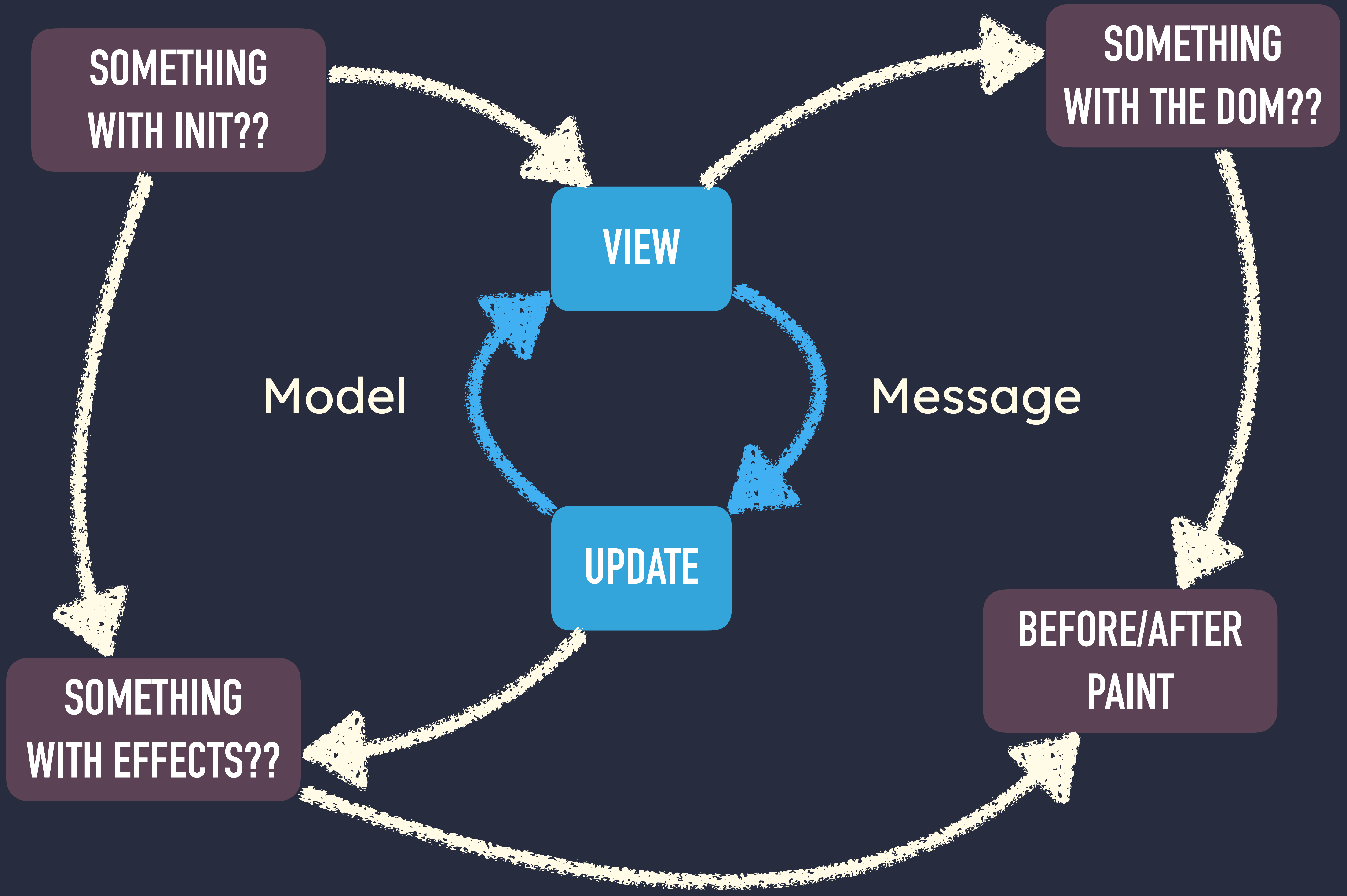
THE APPLICATION LIFECYCLE



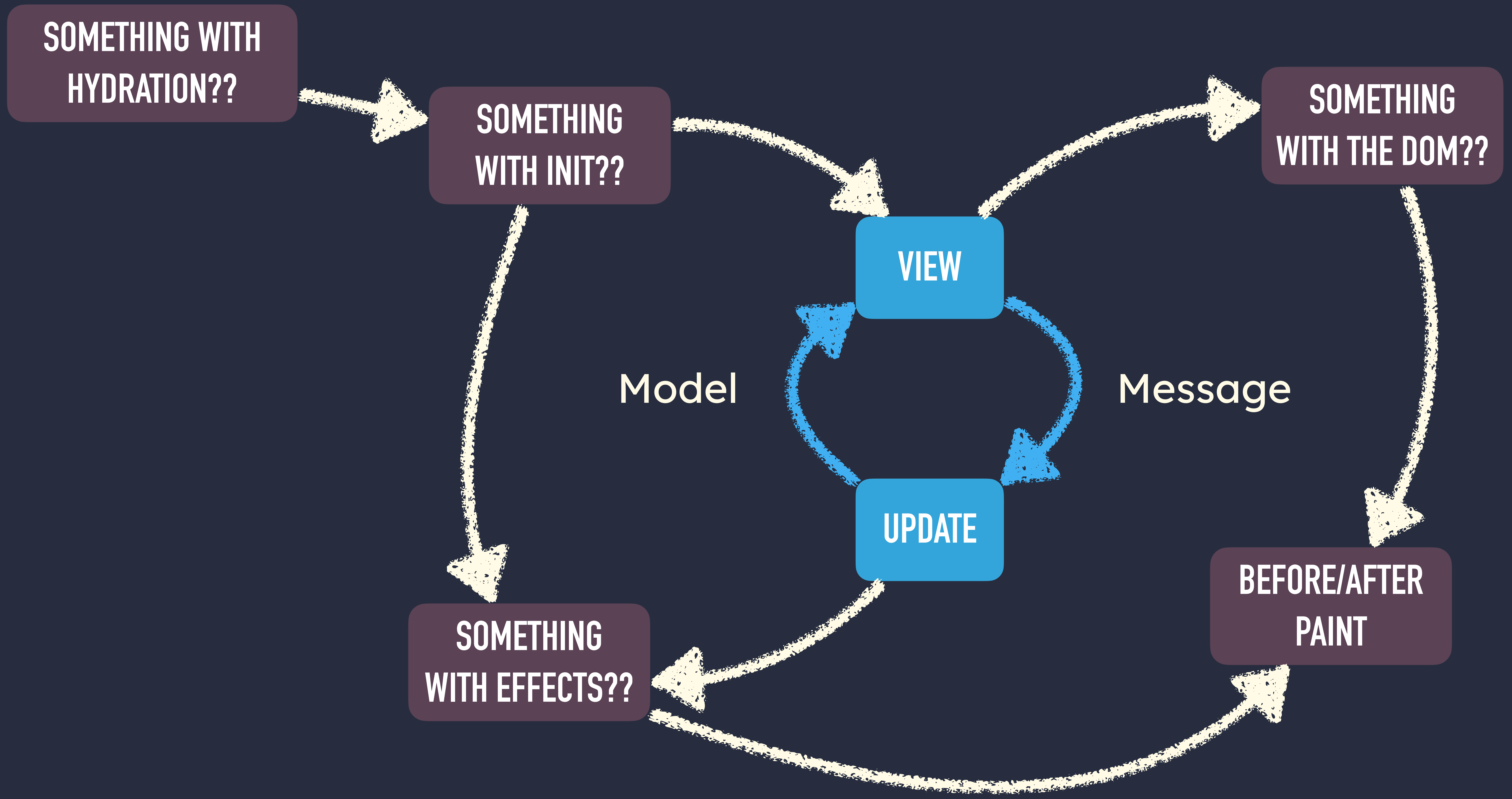
THE APPLICATION LIFECYCLE



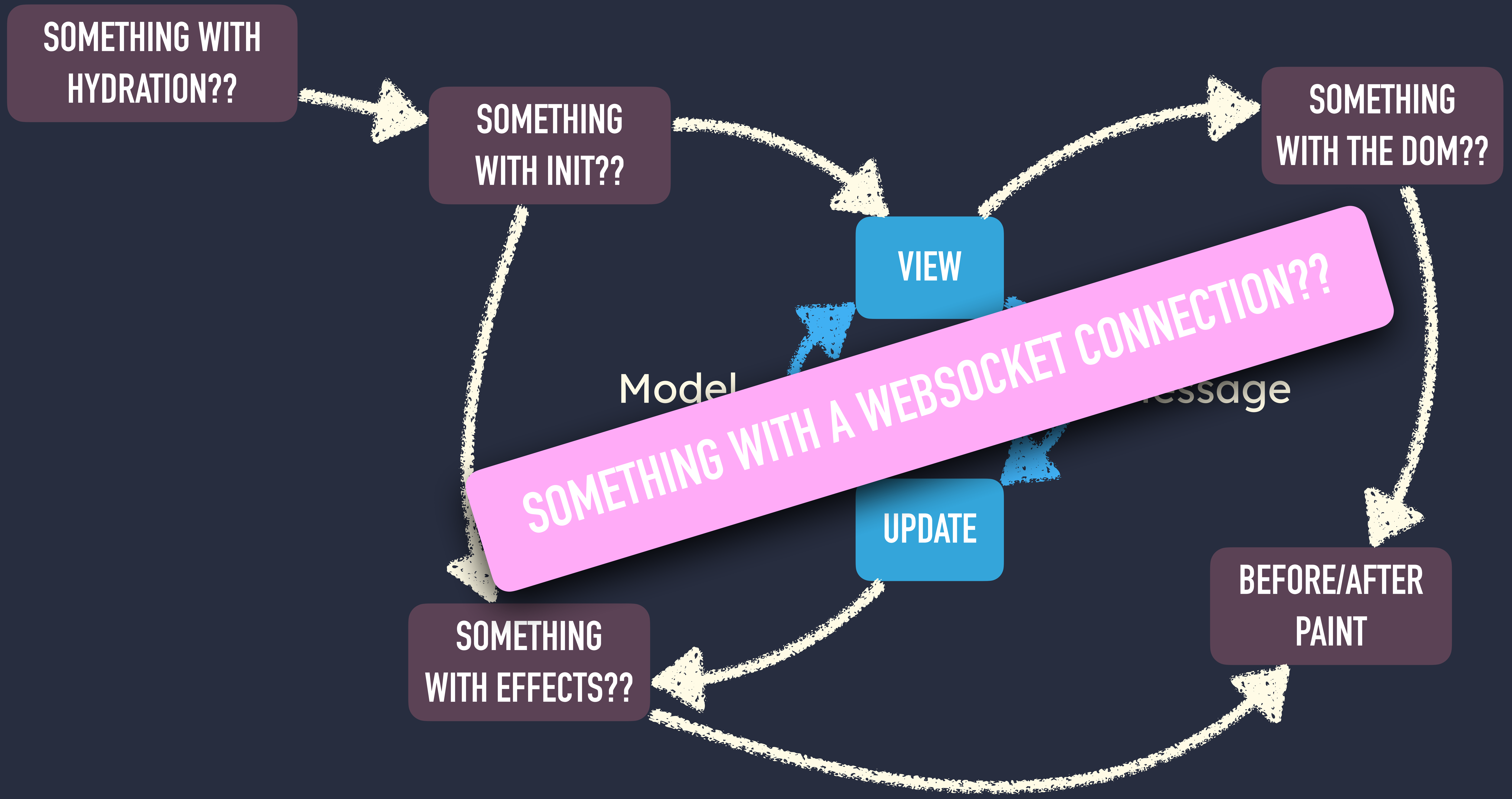
THE APPLICATION LIFECYCLE



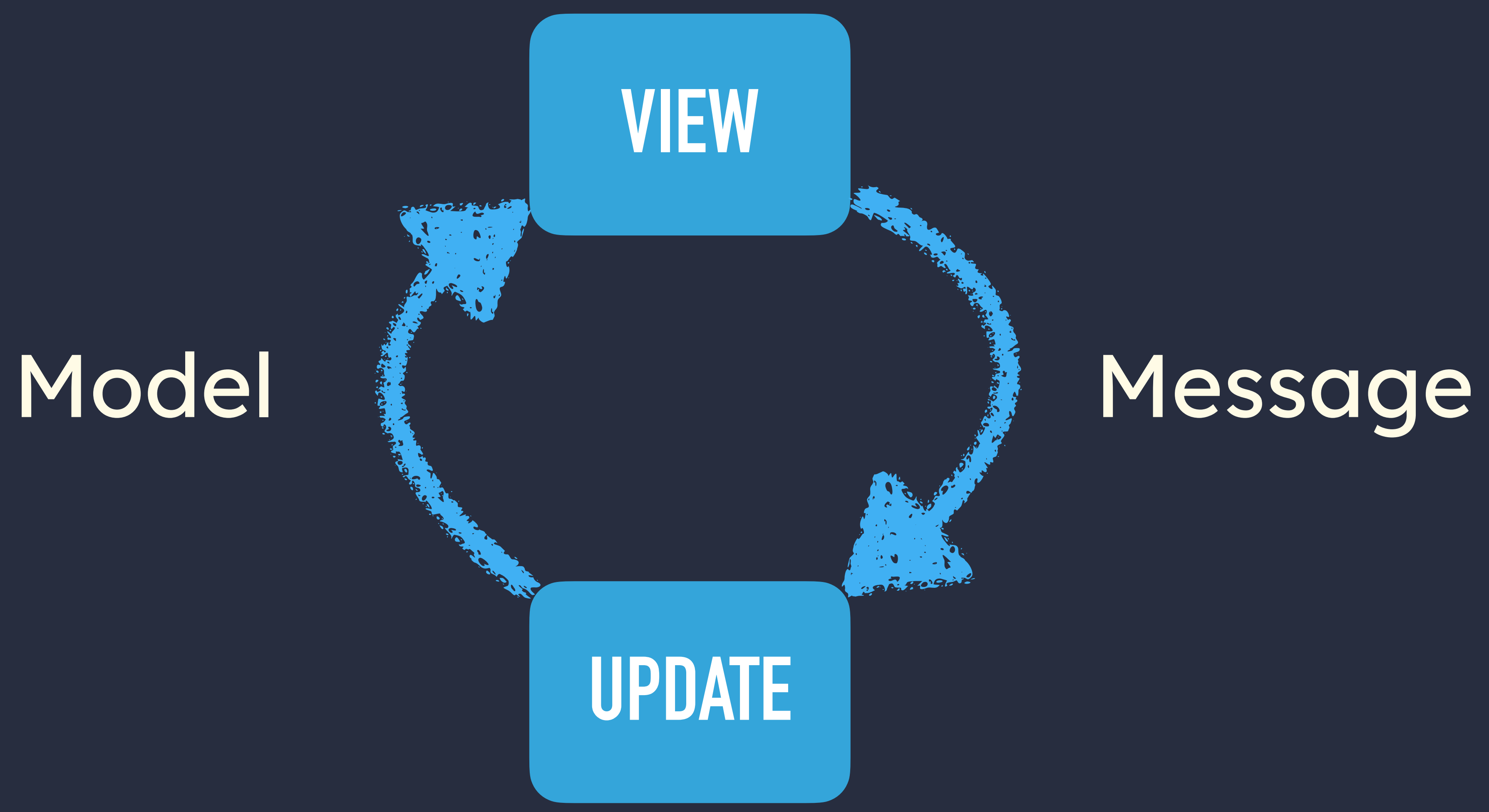
THE APPLICATION LIFECYCLE



THE APPLICATION LIFECYCLE



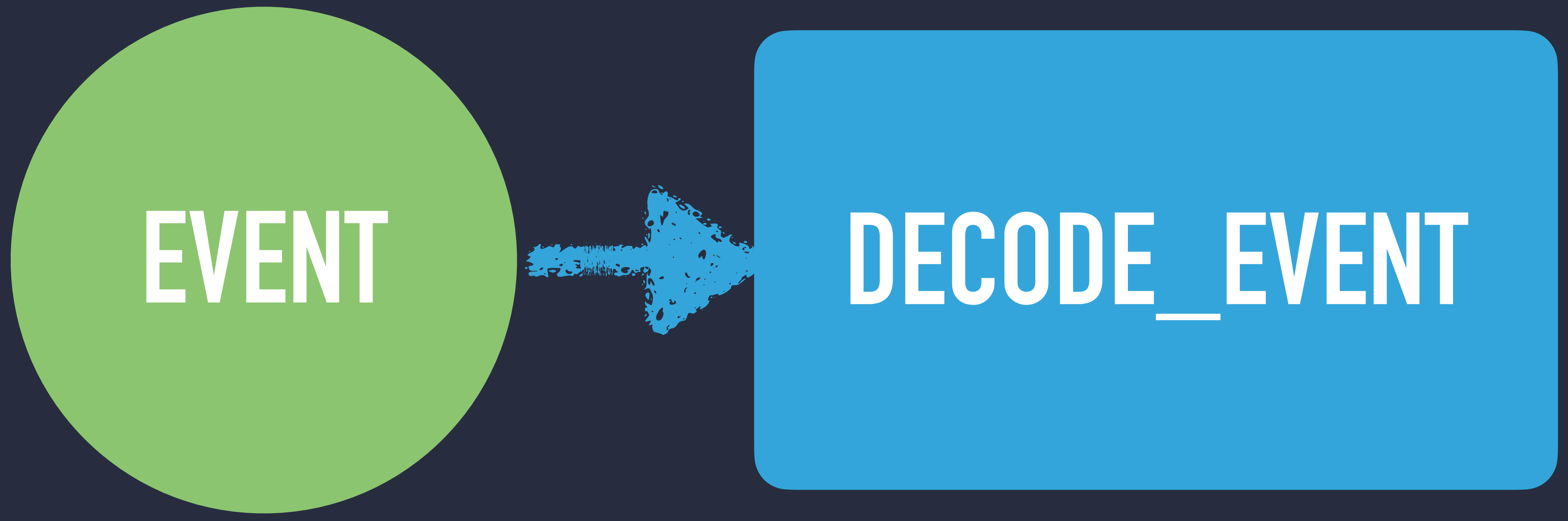
THE APPLICATION LIFECYCLE

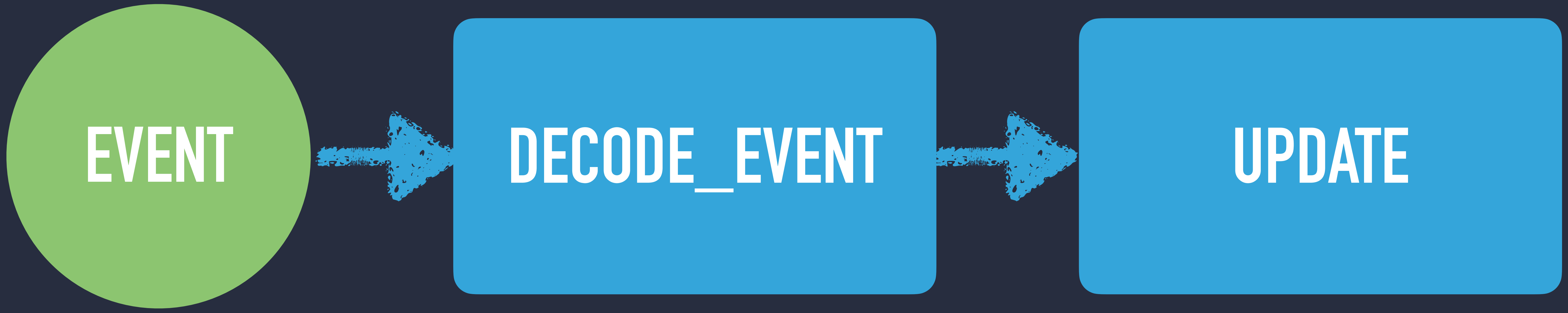


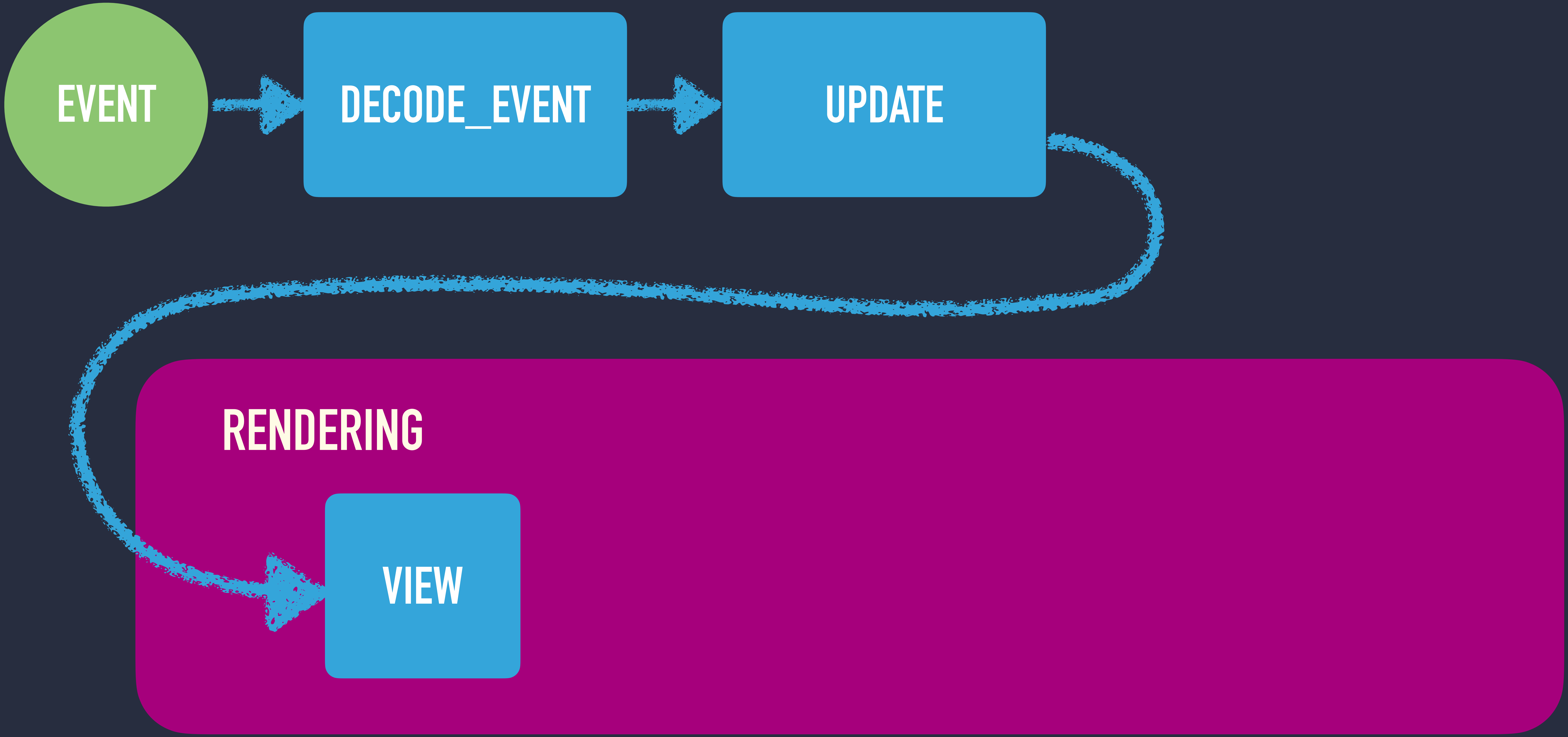
We can just go look!

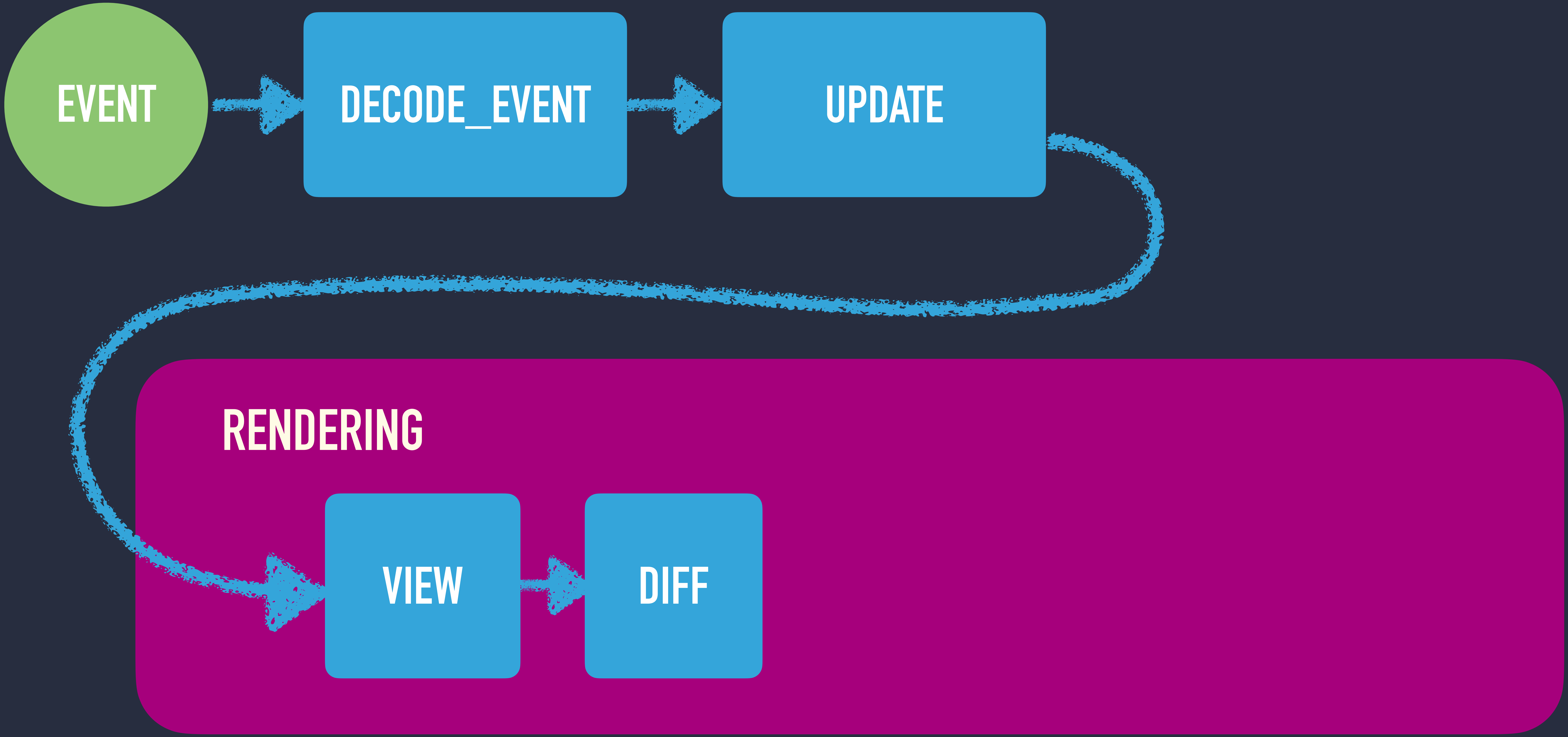
(demo time 😱)

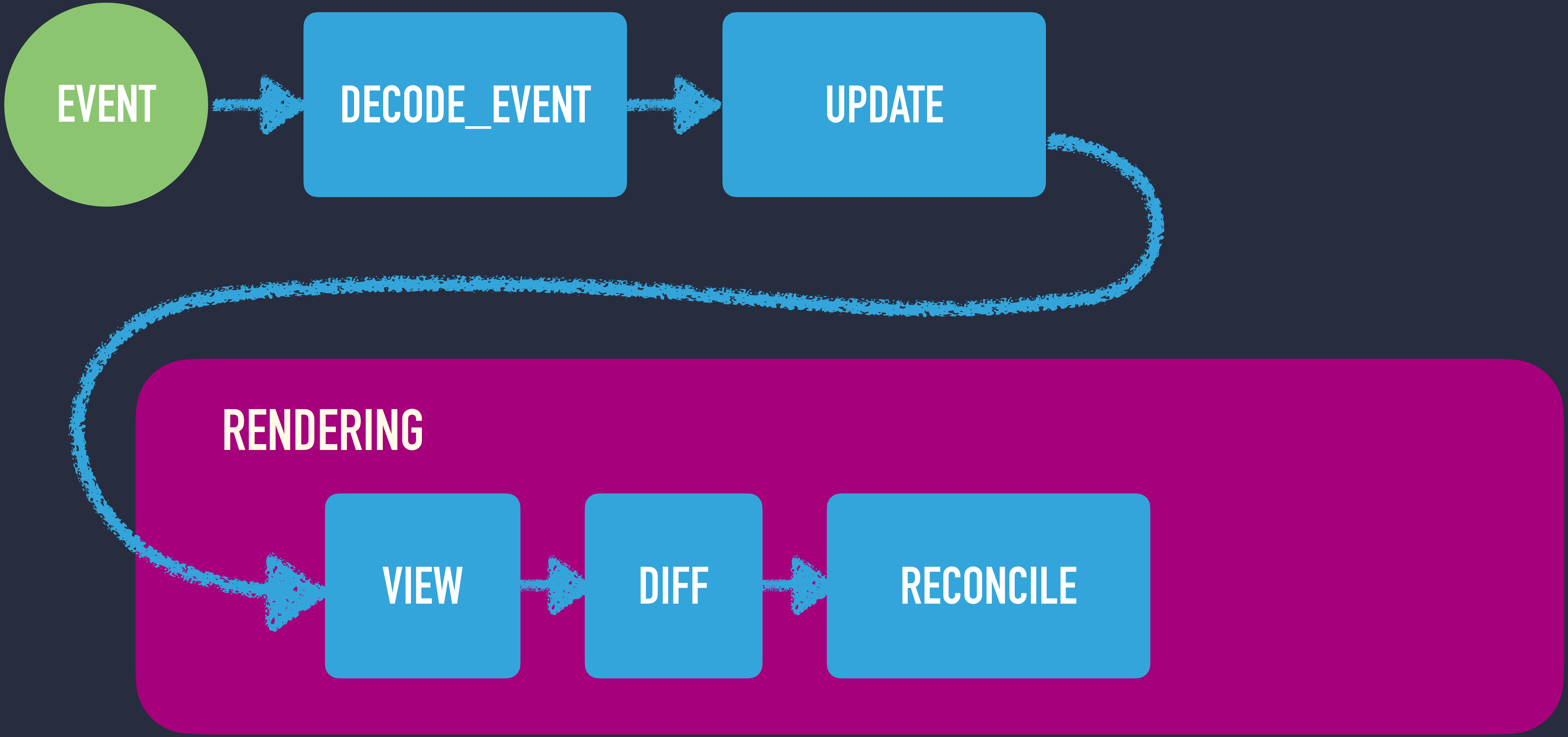
EVENT

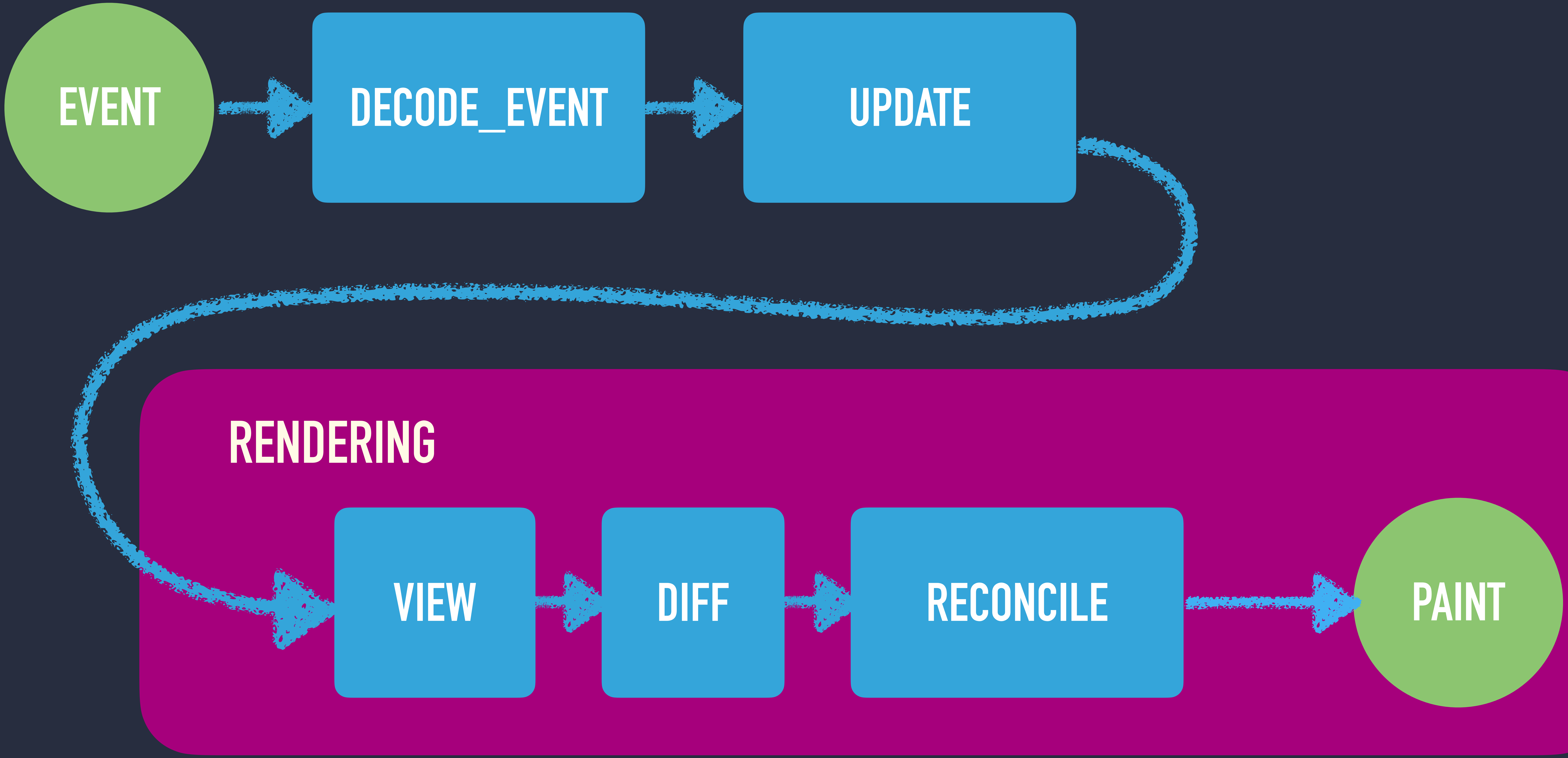

EVENT
DECODE_EVENT

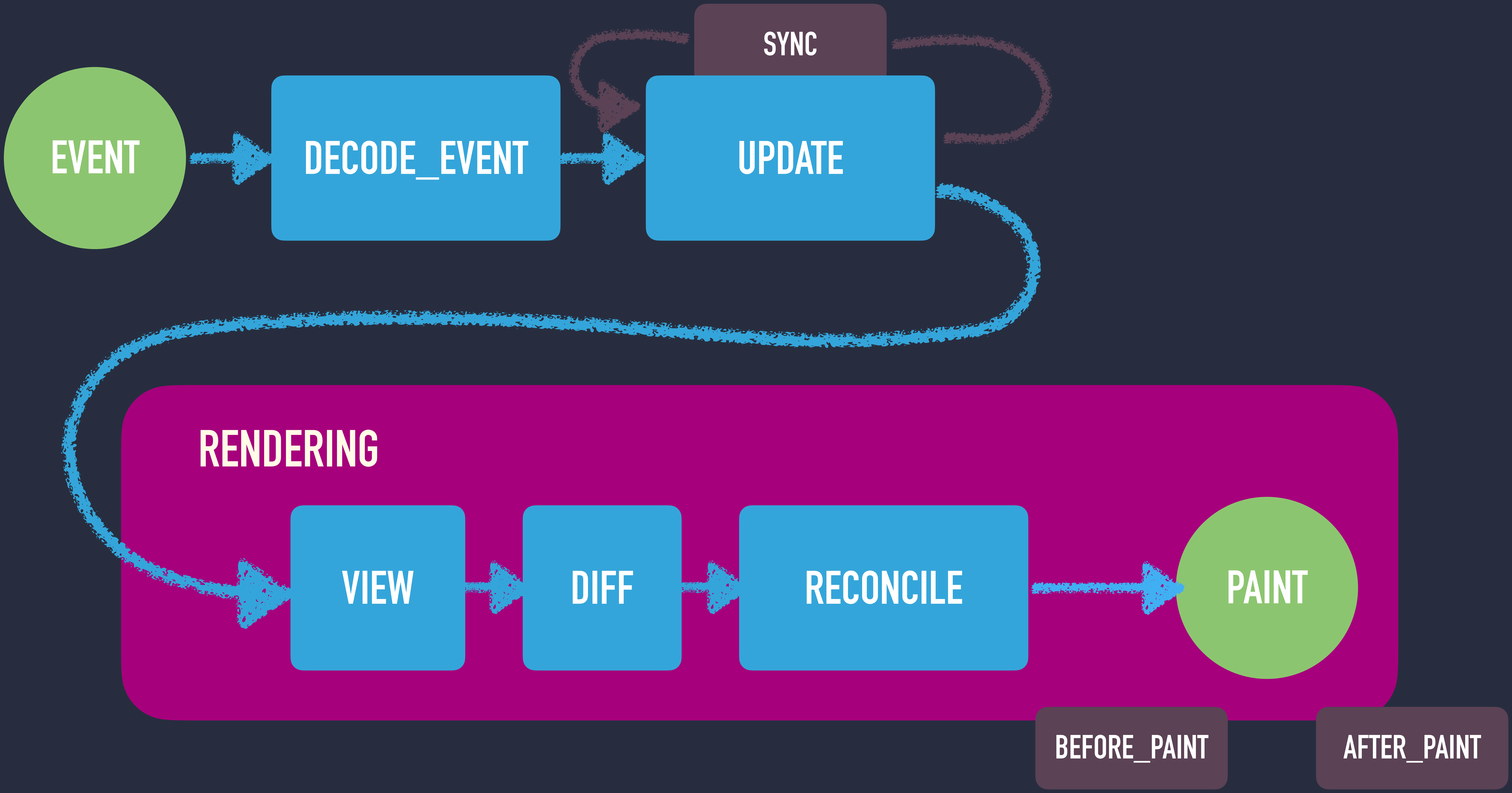

EVENT
DECODE_EVENT
UPDATE

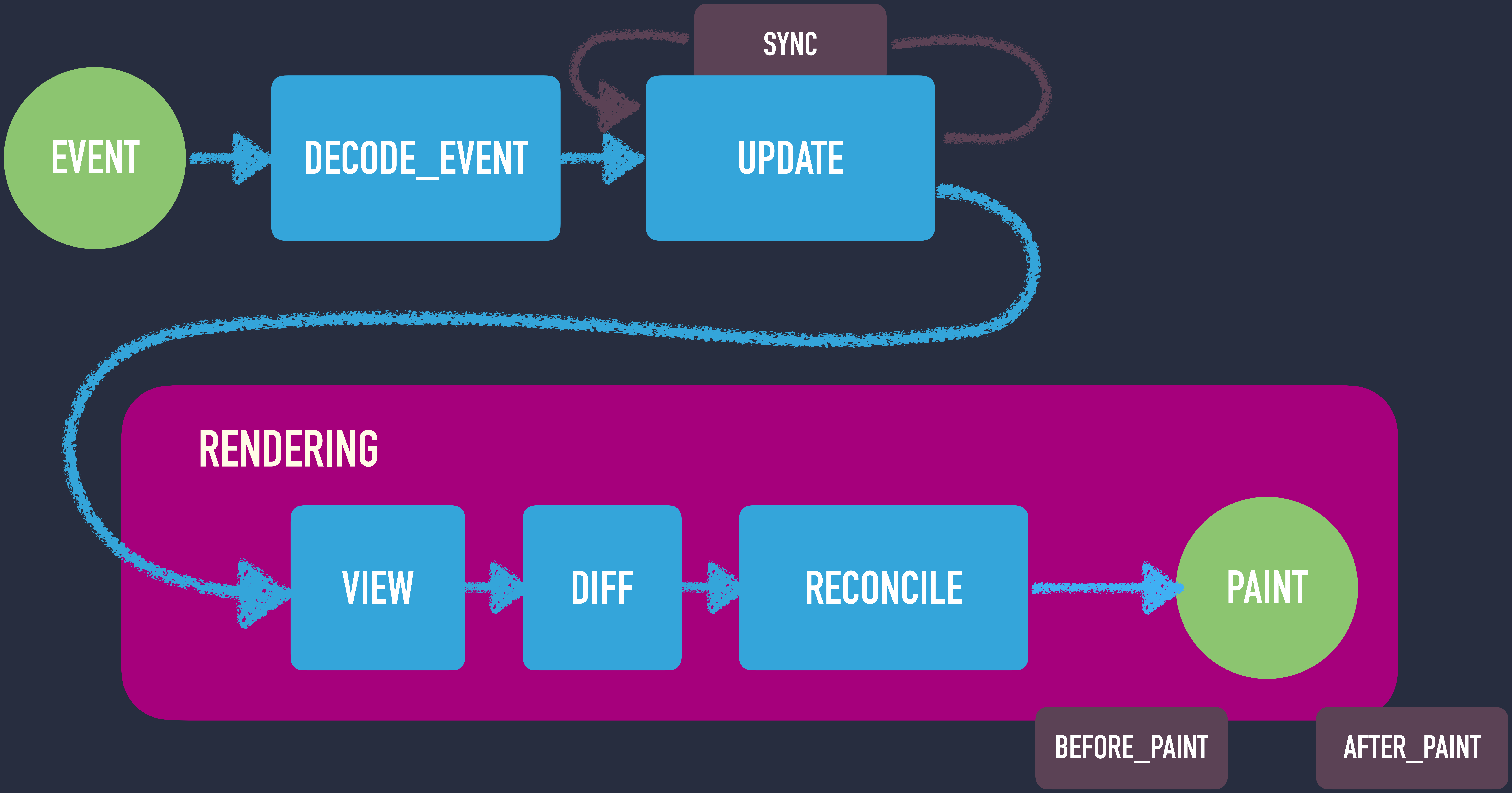

EVENT
DECODE_EVENT
UPDATE
RENDERING
VIEW


EVENT
DECODE_EVENT
UPDATE
RENDERING
VIEW
DIFF

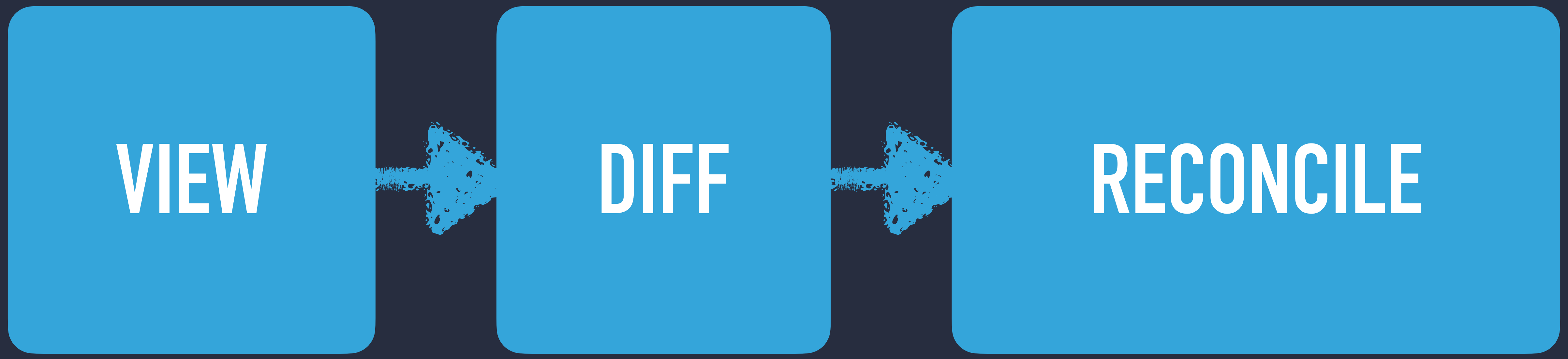

EVENT
DECODE_EVENT
UPDATE
RENDERING
VIEW
DIFF
RECONCILE


EVENT
DECODE_EVENT
UPDATE
RENDERING
VIEW
DIFF
RECONCILE
PAINT

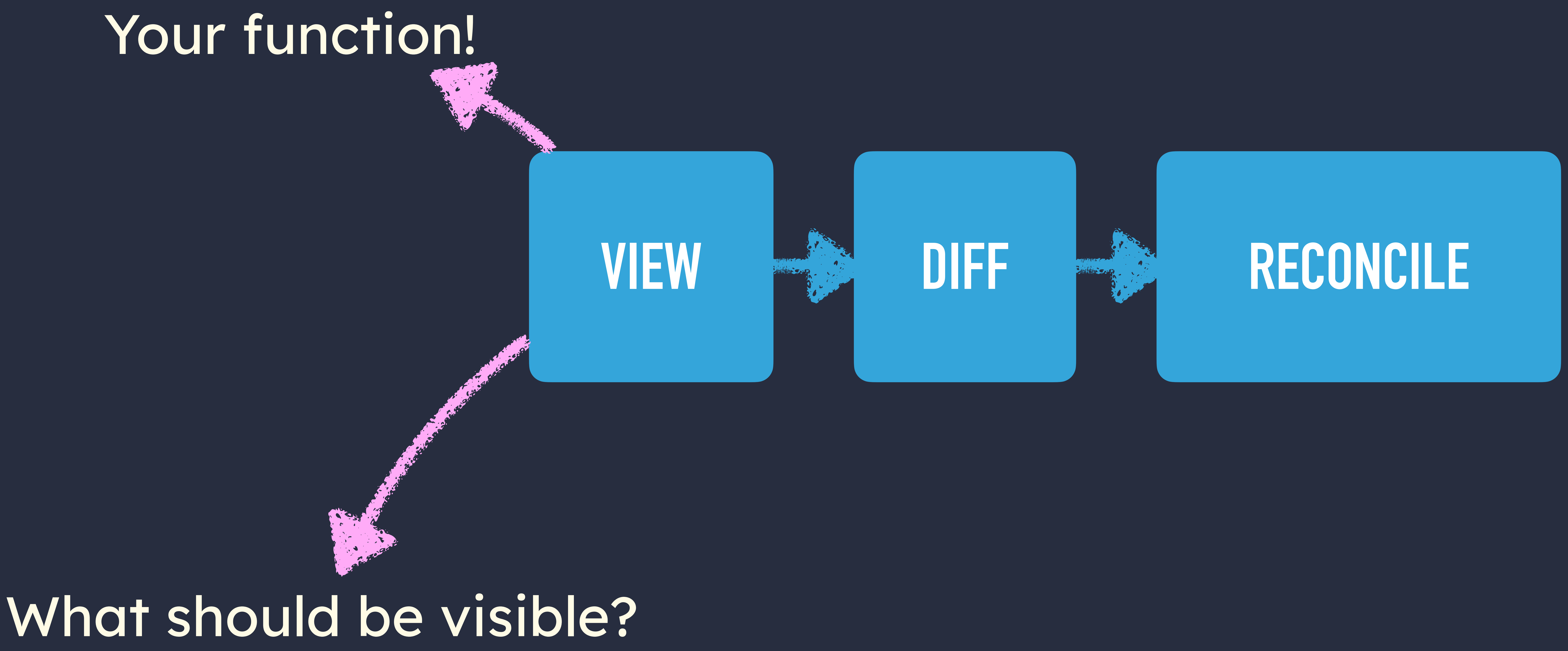

SYNC
EVENT
DECODE_EVENT
UPDATE
RENDERING
VIEW
DIFF
RECONCILE
PAINT
BEFORE_PAINT
AFTER_PAINT


SYNC
EVENT
DECODE_EVENT
UPDATE
RENDERING
VIEW
DIFF
RECONCILE
PAINT
BEFORE_PAINT
AFTER_PAINT

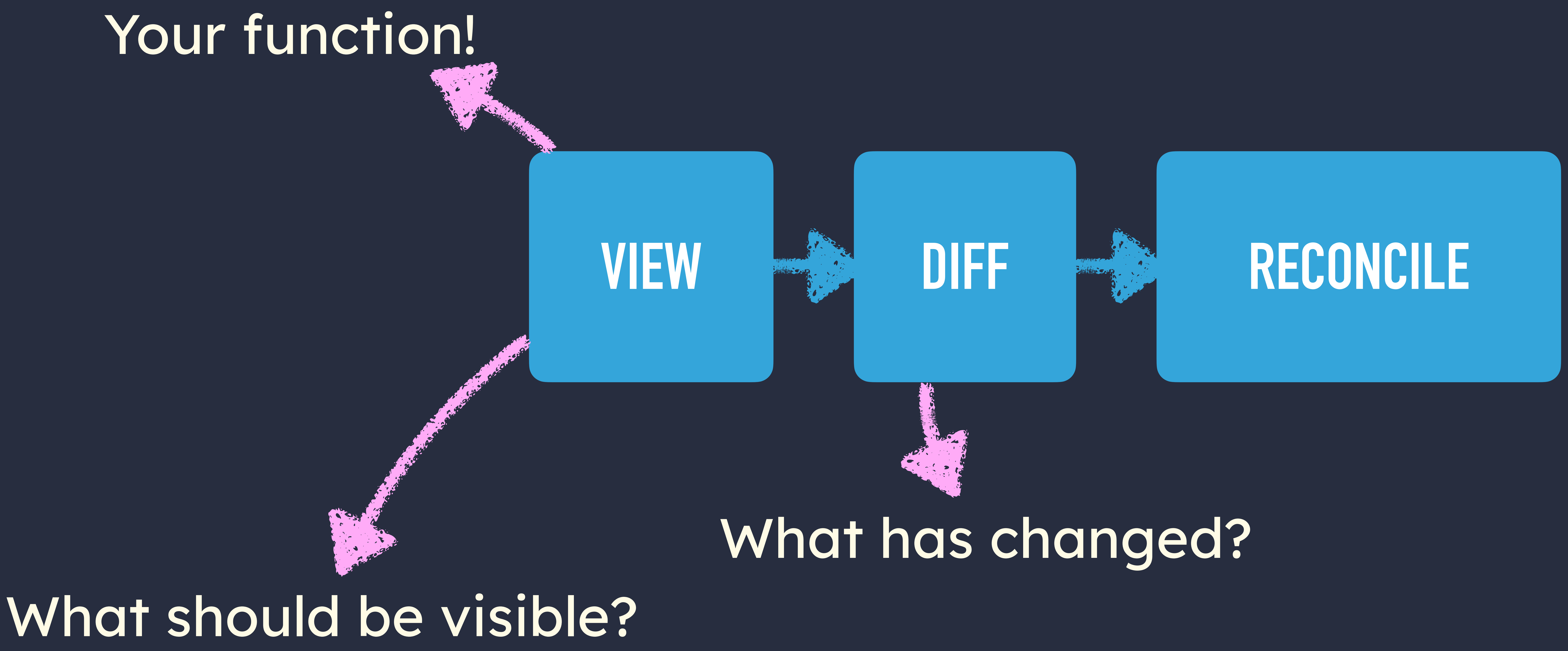
RENDERING



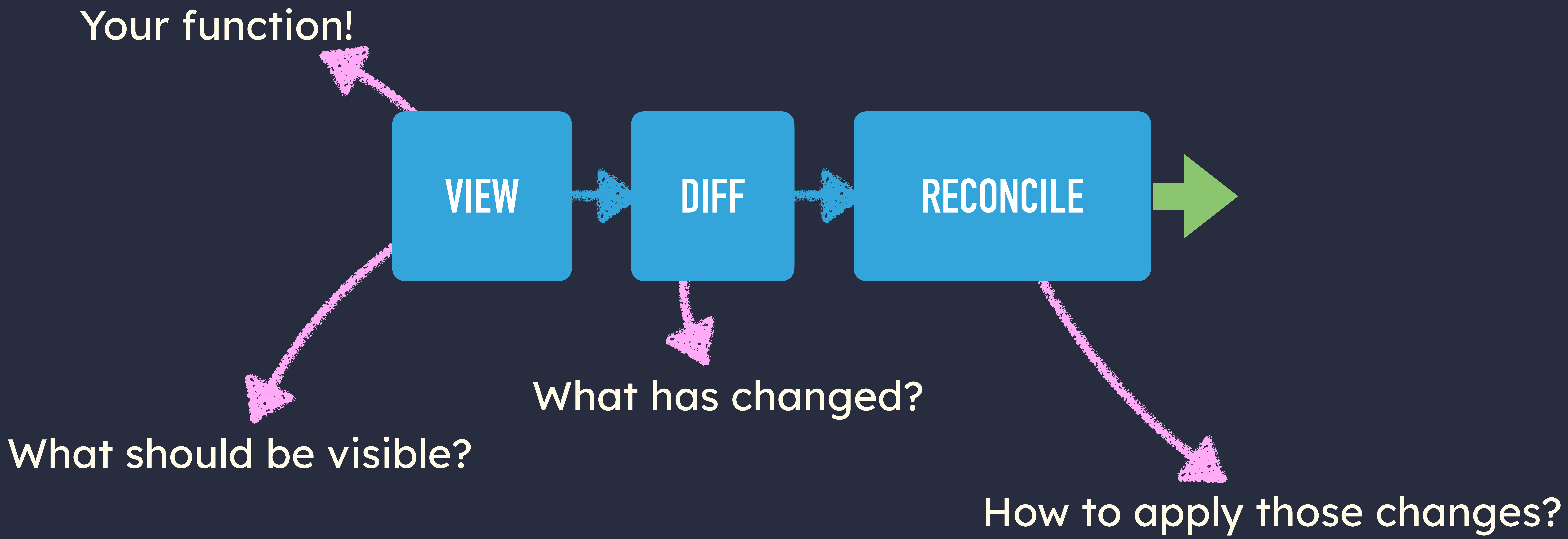
RENDERING



RENDERING


Your function!
VIEW
DIFF
RECONCILE
What has changed?
What should be visible?

RENDERING

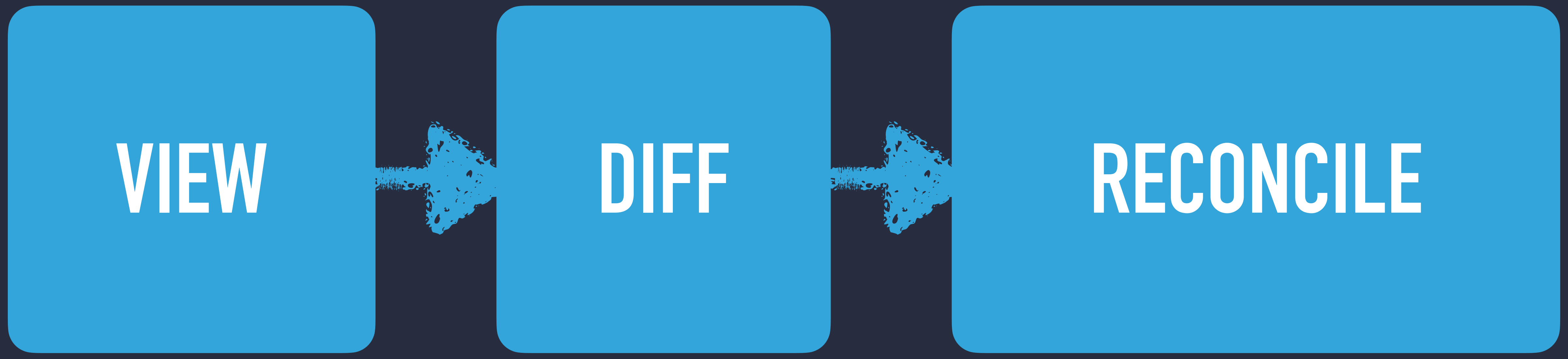

Your function!
VIEW
DIFF
RECONCILE
What has changed?
What should be visible?
How to apply those changes?

RENDERING

CLIENT OR SERVER
ALWAYS ON THE CLIENT
Your function!
VIEW
DIFF
RECONCILE
What should be visible?
What has changed?
How to apply those changes?

NEW OBJECTIVE:

- Where do we spend our time during rendering?
- How does diff work?
- How to produce "good" diffs?
- Can we influence it?



RECONCILER

- "How to make the new view visible"

RECONCILER

- "How to make the new view visible"

A (bad) reconciler implementation:

```
pub fn reconcile() {
  let root = view(model)
  let html = element.to_string(root)
  document.querySelector("#app").innerHTML = html
}
```

RECONCILER

- "How to make the new view visible"

A (bad) reconciler implementation:

```
pub fn reconcile() {
  let root = view(model)
  let html = element.to_string(root)
  document.querySelector("#app").innerHTML = html
}
```

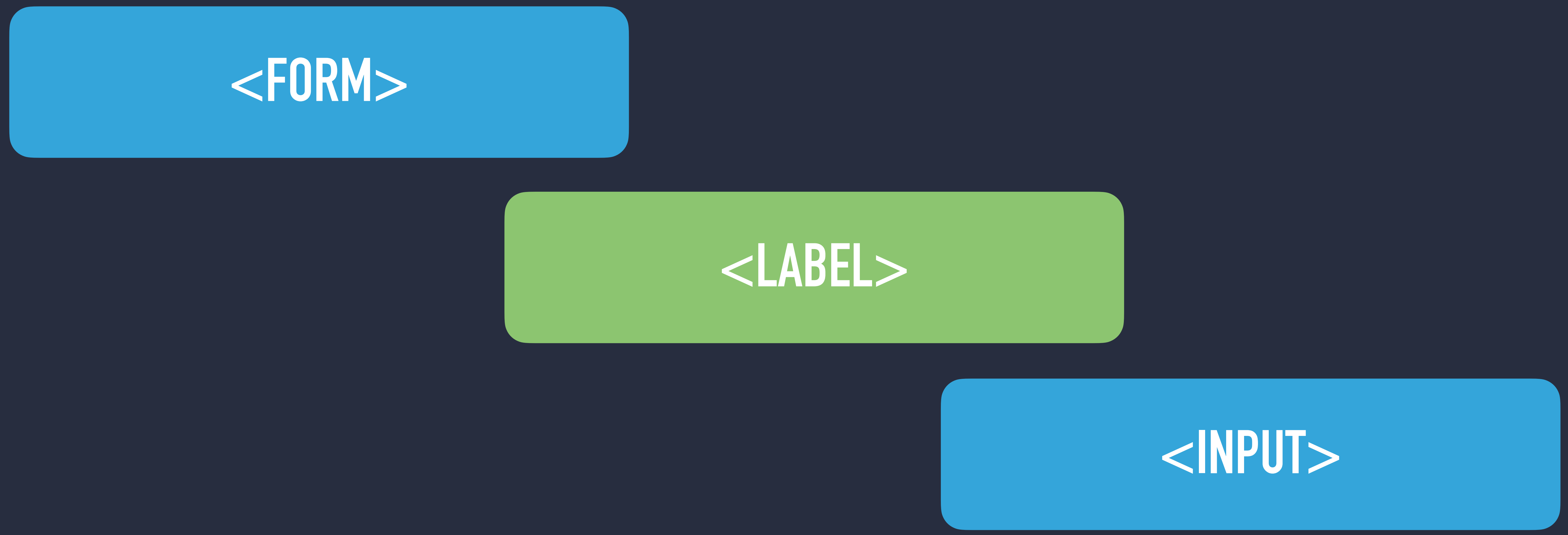
- Does not preserve state, replaces **entire html** on every update

RECONCILER

- "How to make the new view visible with minimal changes"

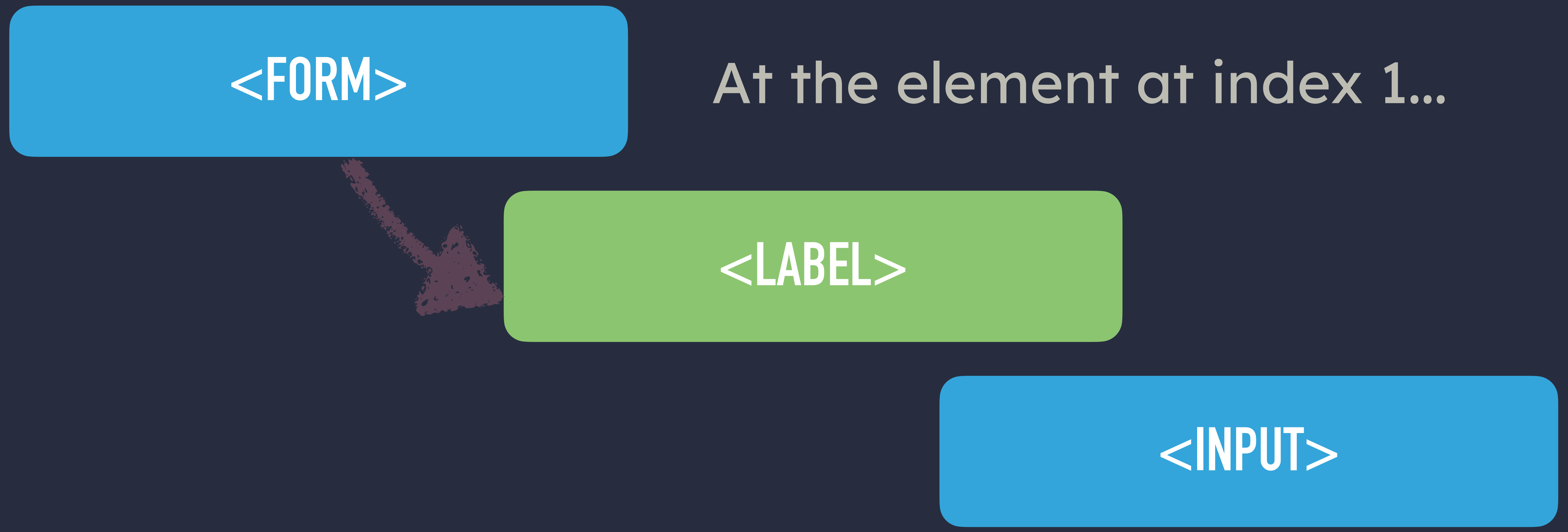
RECONCILER

- "How to make the new view visible with minimal changes"



RECONCILER

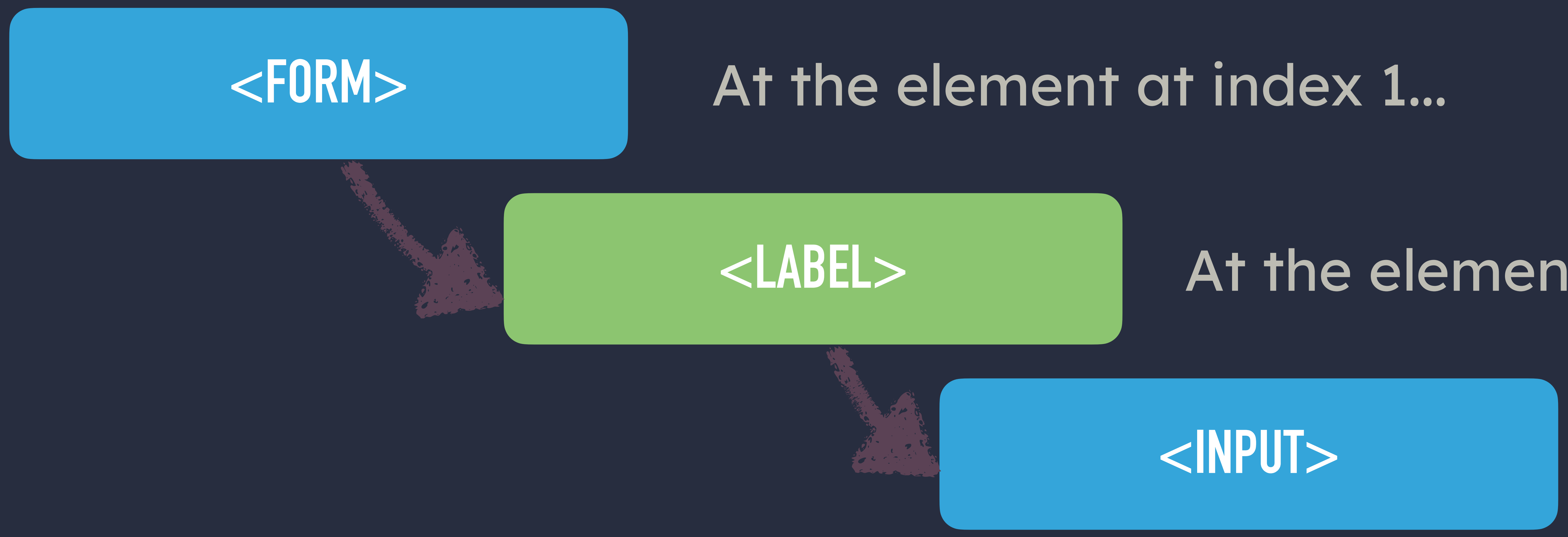
- "How to make the new view visible with minimal changes"



At the element at index 1...

RECONCILER

- "How to make the new view visible with minimal changes"

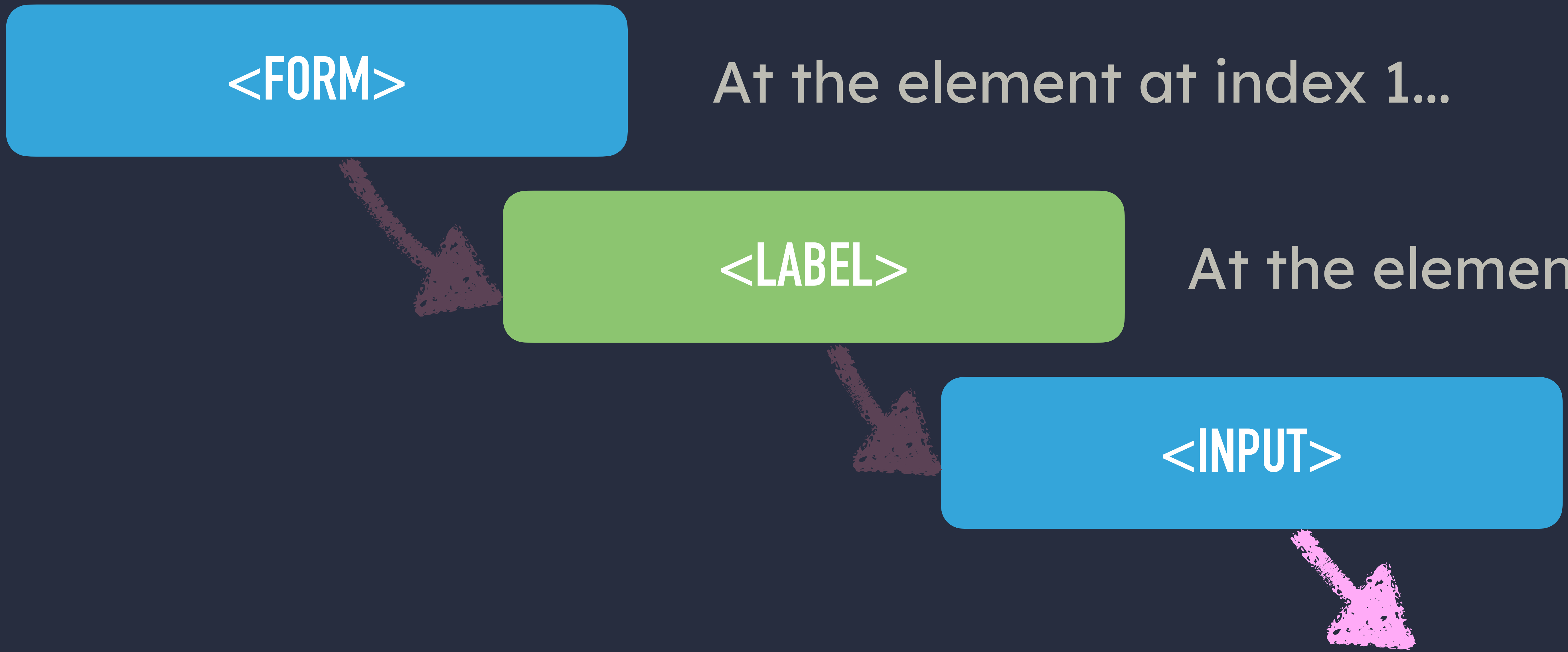


At the element at index 1...

At the element with key `first_name`...

RECONCILER

- "How to make the new view visible with minimal changes"



At the element at index 1…

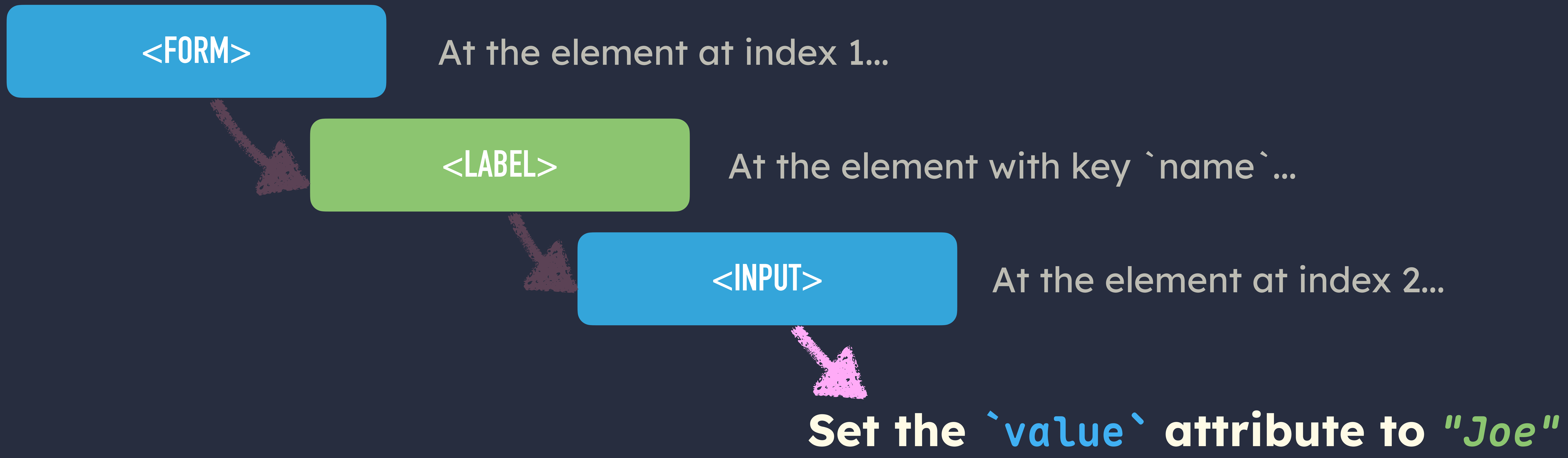
At the element with key `name`…

At the element at index 2…

Set the `value` attribute to *"Joe"*

RECONCILER

- "How to make the new view visible with minimal changes"


<FORM>
At the element at index 1…
<LABEL>
At the element with key `name`…
<INPUT>
At the element at index 2…
Set the `value` attribute to "Joe"

- Might mean replacing the entire view!

DIFFS

DIFFS

- Textual diffs

```diff
- io.println("Hello world!")
+ io.println("Hello Joe!")
```

- LiveView, HTMX, Datastar (all w/ some structural elements)

INPUT

- Textual diffs

```
- io.println("Hello world!")
+ io.println("Hello Joe!")
```

- LiveView, HTMX, Datastar (all w/ some structural elements)

- Structural diffs

```
{ "1": { "name": { "2": {
    "$SET": { "value": "Joe" }
} } } }
```

- Elm, React, Vue, Lustre, Svelte, Solid, …

THE VIRTUAL DOM

- Represent the Element structure directly as a Gleam value

THE VIRTUAL DOM

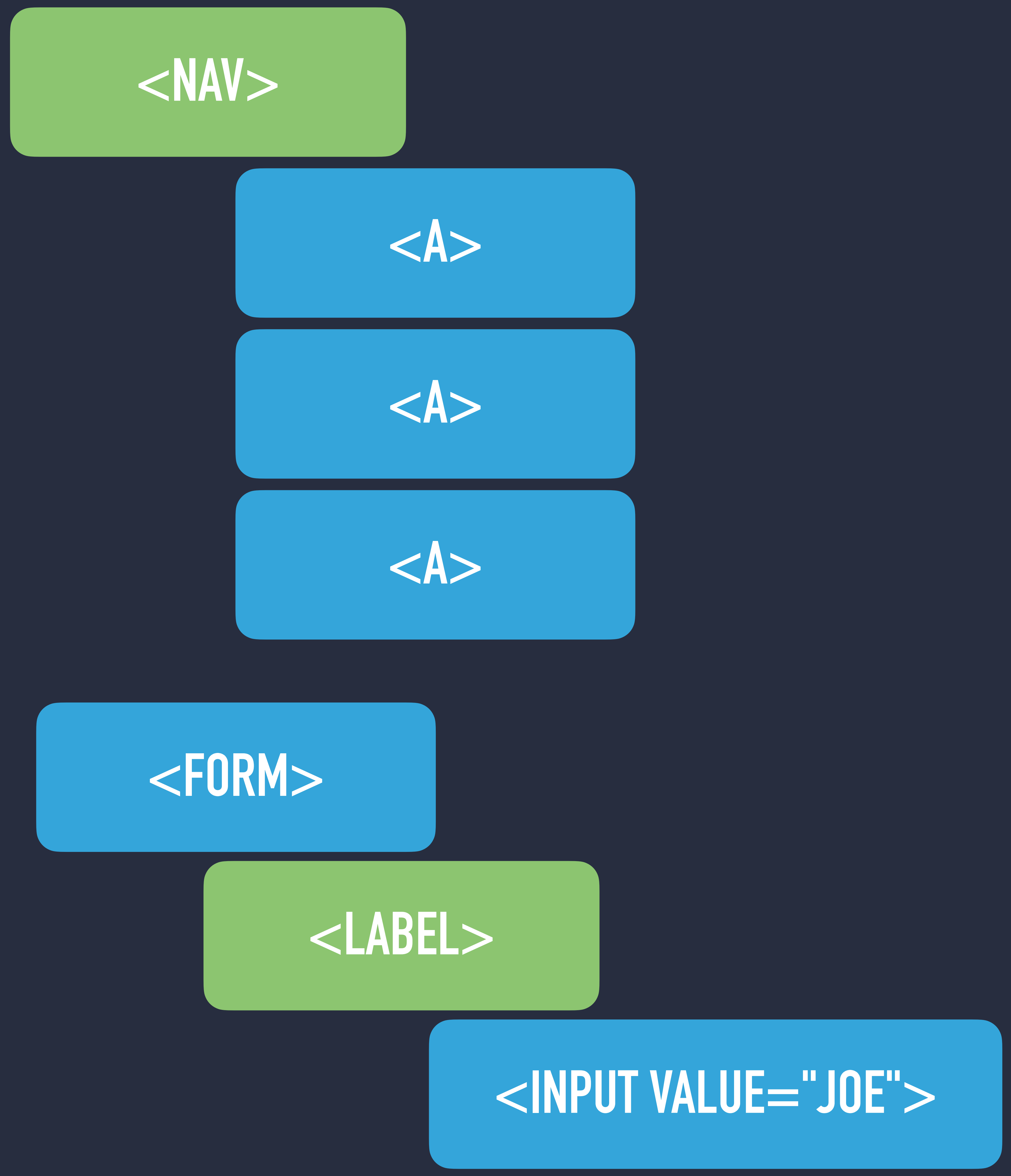
- Represent the Element structure directly as a Gleam value

```
pub opaque type Element(message) {
  Node(
    tag: String,
    attributes: List(Attribute(message)),
    children: List(Element(message)),
  )
  Text(content: String)
}
```

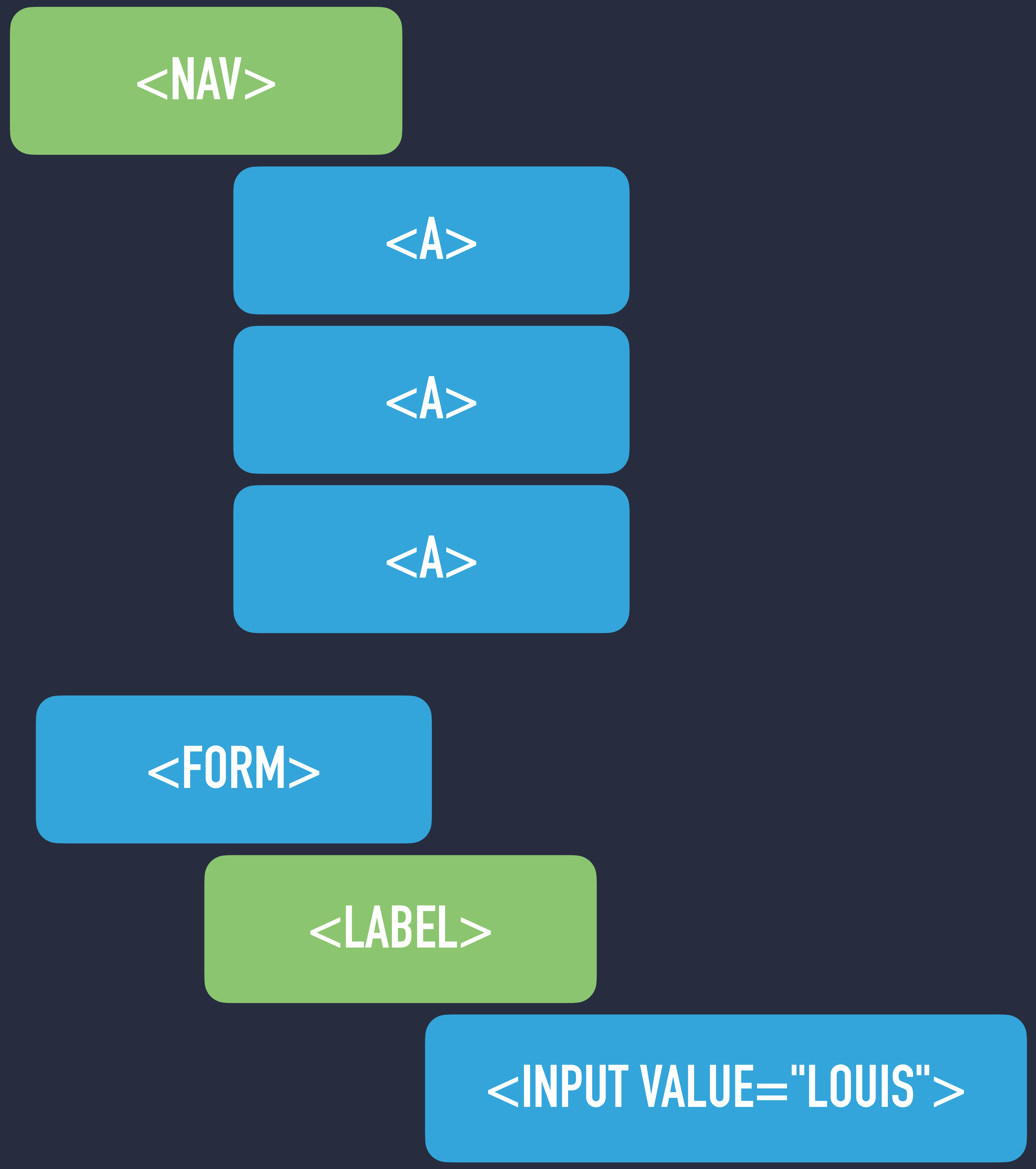
All we need to do is find the minimum edit distance between 2 trees with a dynamic cost function, easy

COMPARING VIRTUAL DOMS

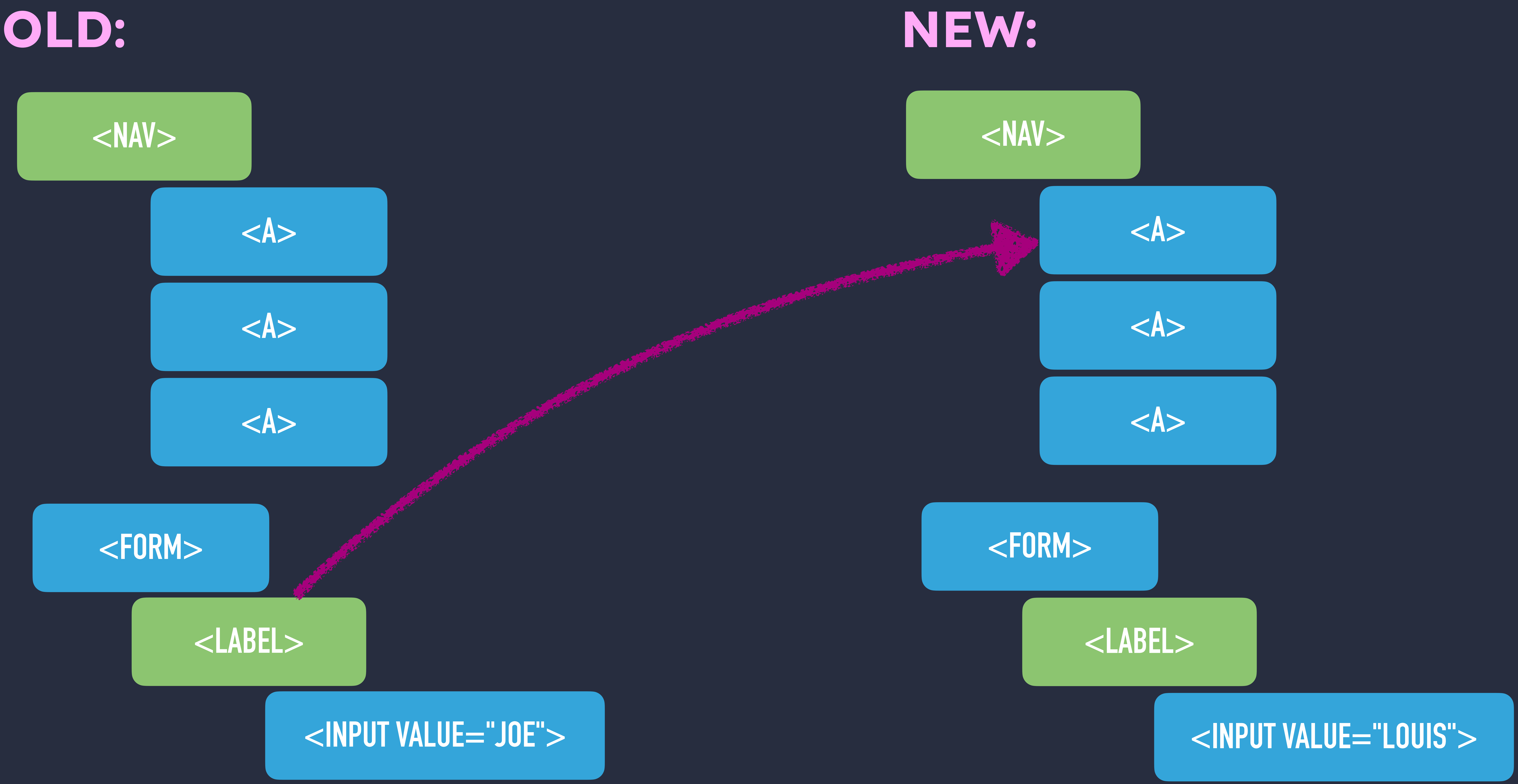
OLD:



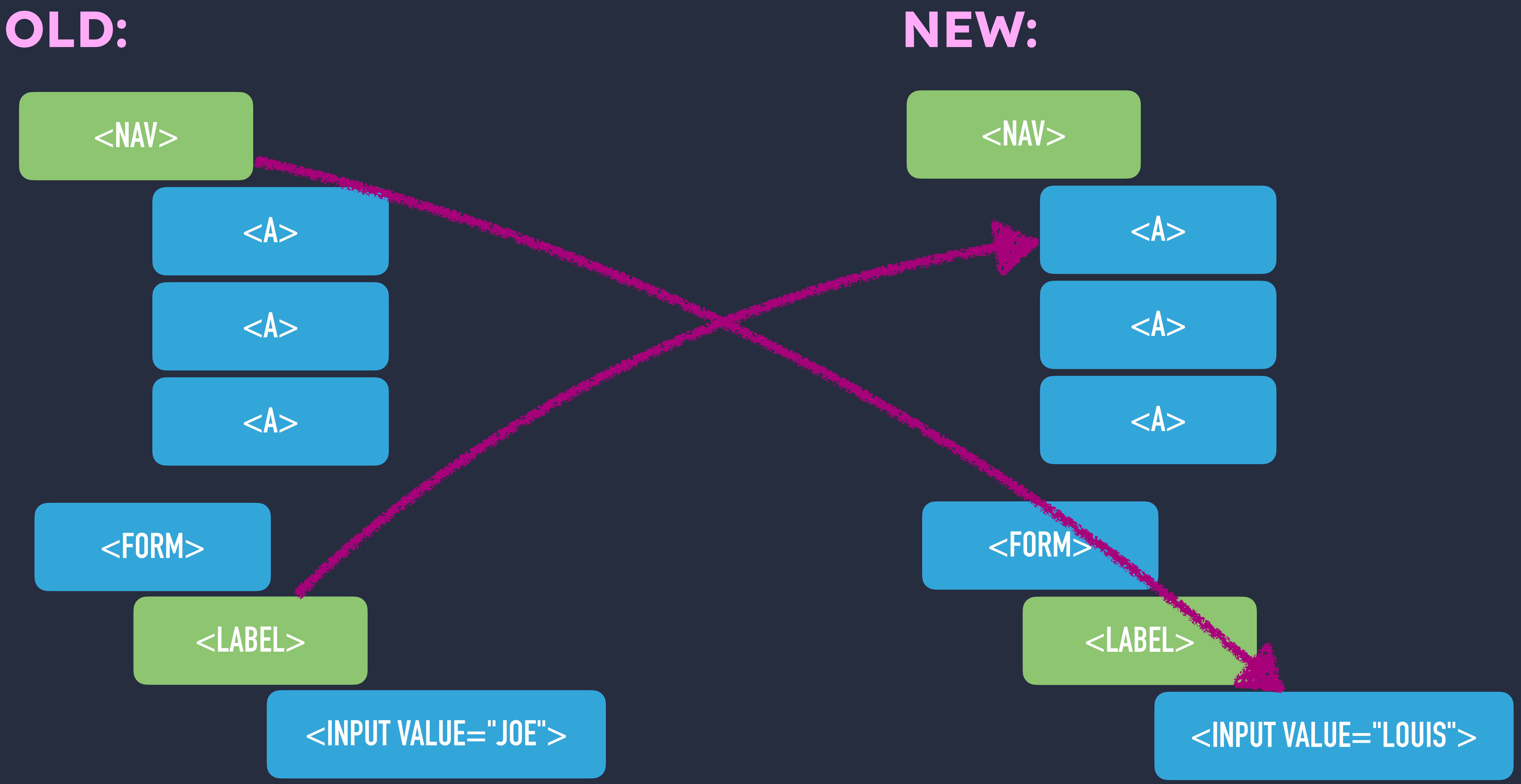
NEW:



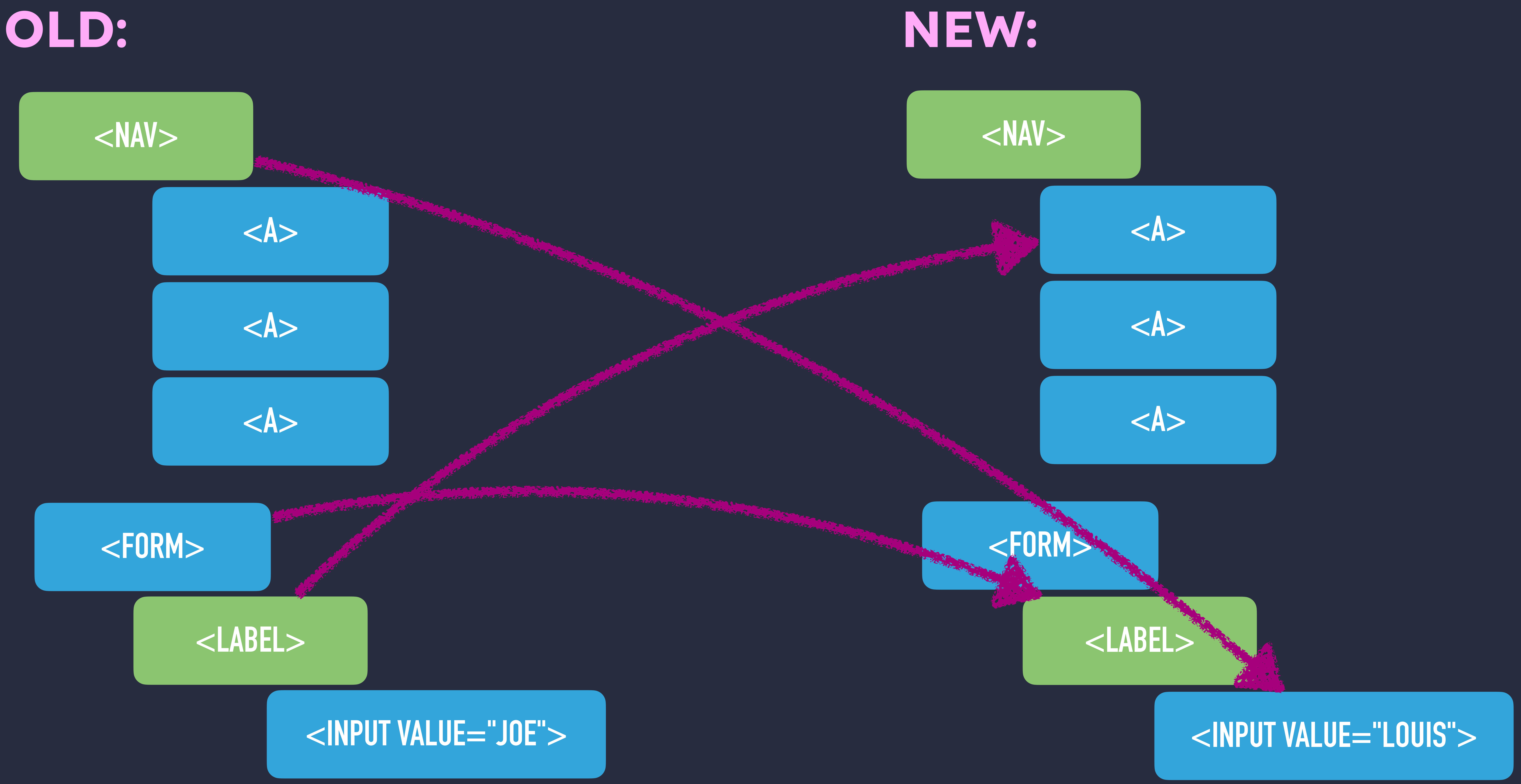
COMPARING VIRTUAL DOMS



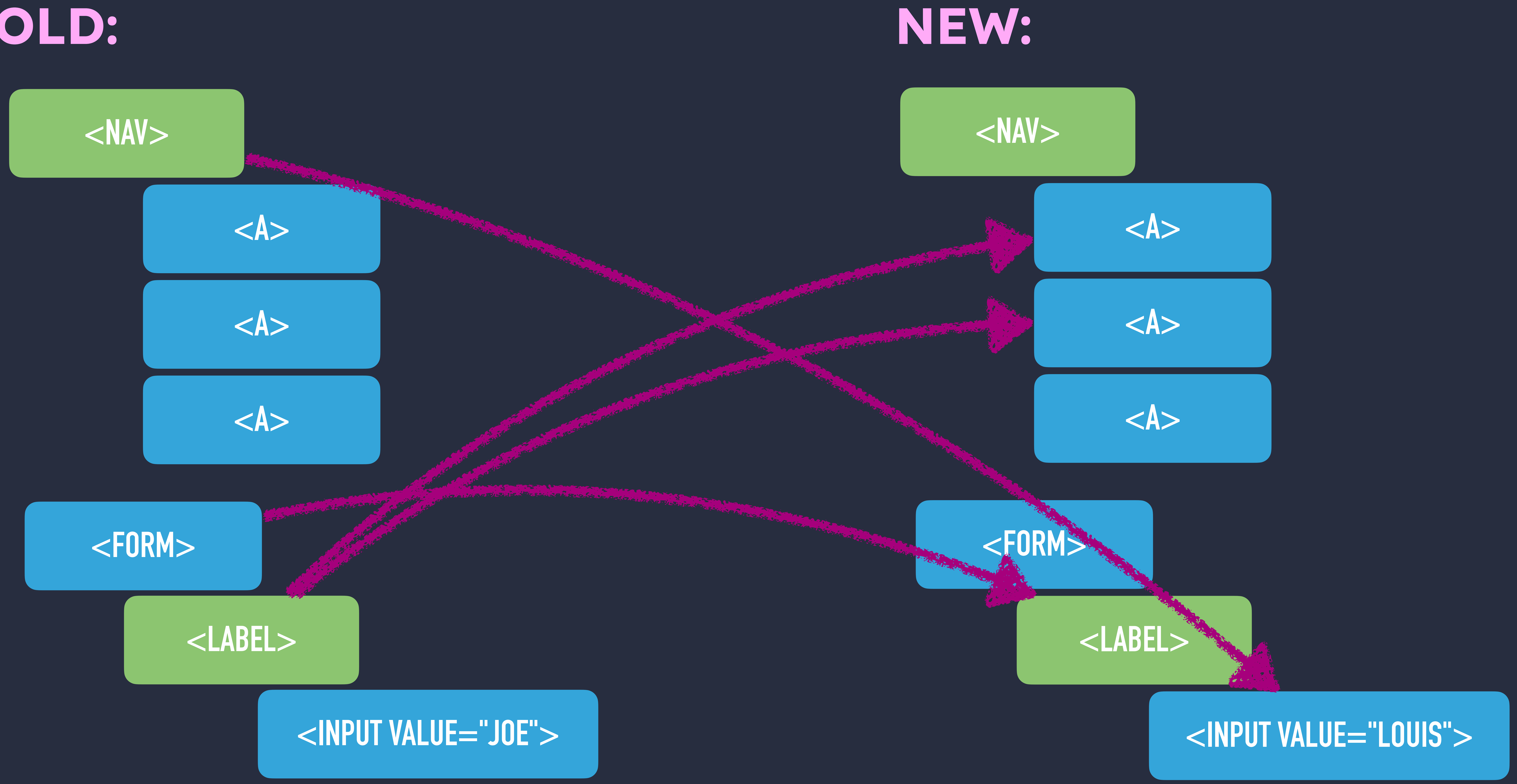
COMPARING VIRTUAL DOMS



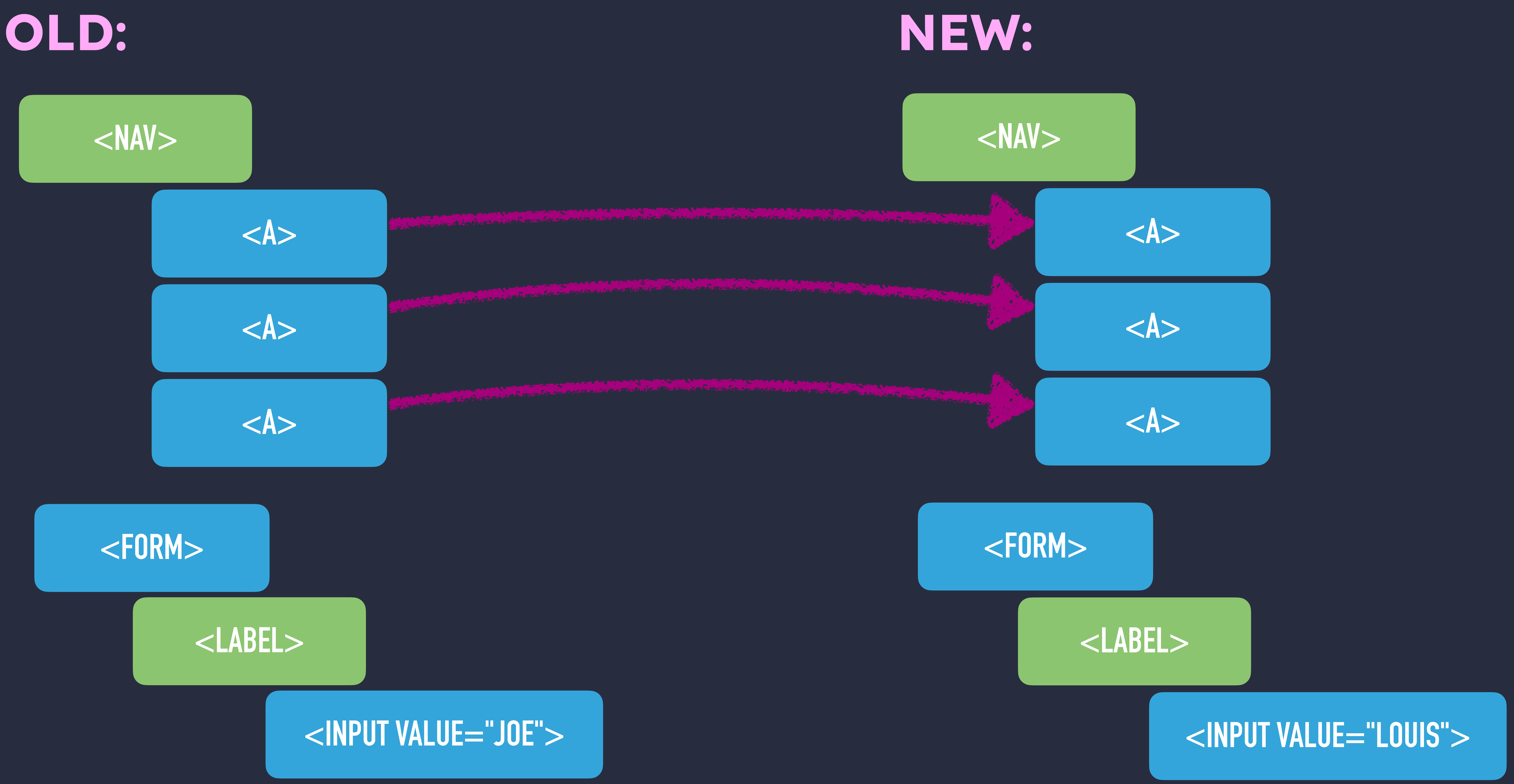
COMPARING VIRTUAL DOMS



COMPARING VIRTUAL DOMS



COMPARING VIRTUAL DOMS



HEURISTICS

- **Local**: Elements can be compared w/o looking at the global context
- Most of the view stays the same and preserves it's structure
- Changing the element **type** means the element got **replaced**
- Changing the **tag name** or **key** means the element got **replaced**
- Inserts are more likely at the **end** of elements
- Different element **types** let you **influence** the process

HEURISTICS

- **Local**: Elements can be compared w/o looking at the global context
- Most of the view stays the same and preserves it's structure
- Changing the element **type** means the element got **replaced**
- Changing the **tag name** or **key** means the element got **replaced**
- Inserts are more likely at the **end** of elements
- Different element **types** let you **influence** the process

→ Diff is a single pass over the tree!

VIEW

NORMAL ELEMENTS
FRAGMENTS

KEYED ELEMENTS
KEYED FRAGMENTS

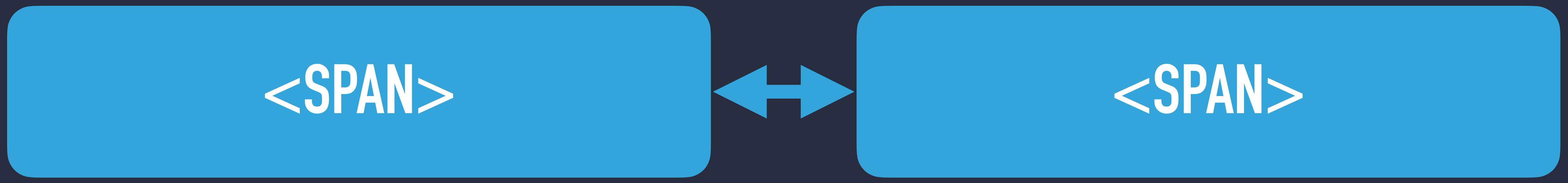
MAP NODES

MEMO NODES

NORMAL ELEMENTS & FRAGMENTS

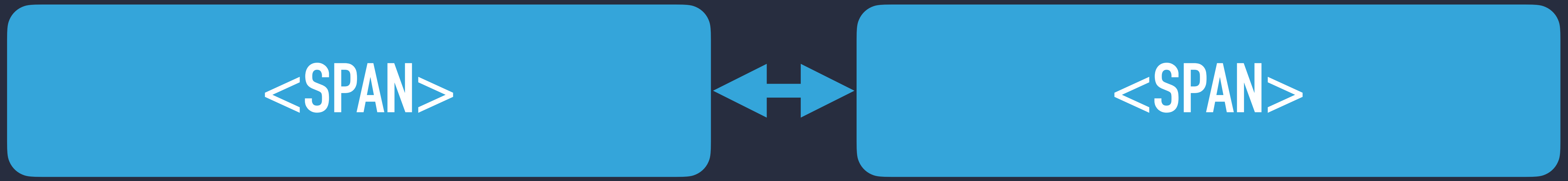
```
html.div([], [
  html.label([], [
    html.span([], [text("Your name:")]),
    html.input([
      attribute.value(model.name),
      attribute.on_input(UserChangedName)
    ])
  ])
])
```

NORMAL ELEMENTS & FRAGMENTS



```
html.div([], [
  html.label([], [
    html.span([], [text("Your name:")]),
    html.input([
      attribute.value(model.name),
      attribute.on_input(UserChangedName)
    ])
  ])
])
```

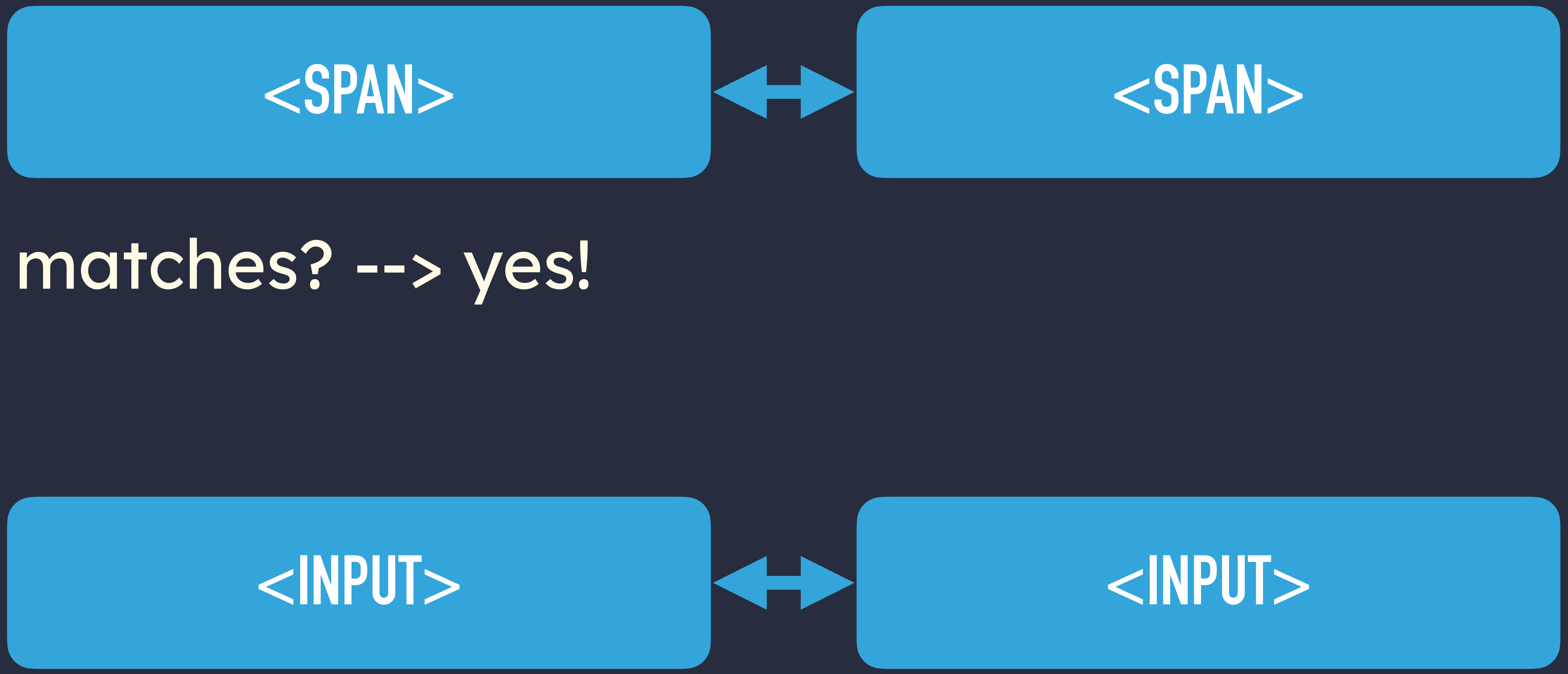
NORMAL ELEMENTS & FRAGMENTS



matches? --> yes!

```
html.div([], [
  html.label([], [
    html.span([], [text("Your name:")]),
    html.input([
      attribute.value(model.name),
      attribute.on_input(UserChangedName)
    ])
  ])
])
```

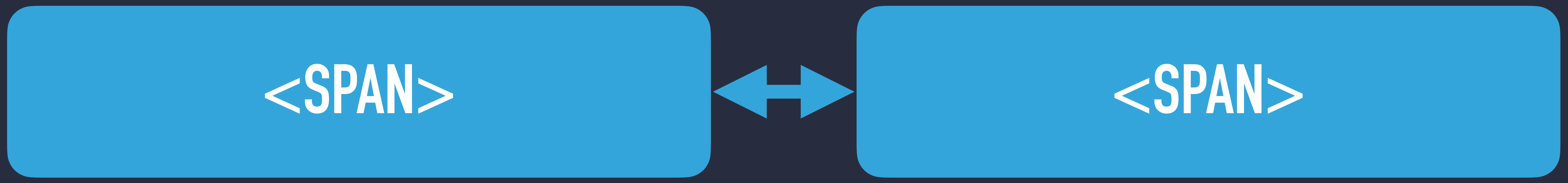
NORMAL ELEMENTS & FRAGMENTS



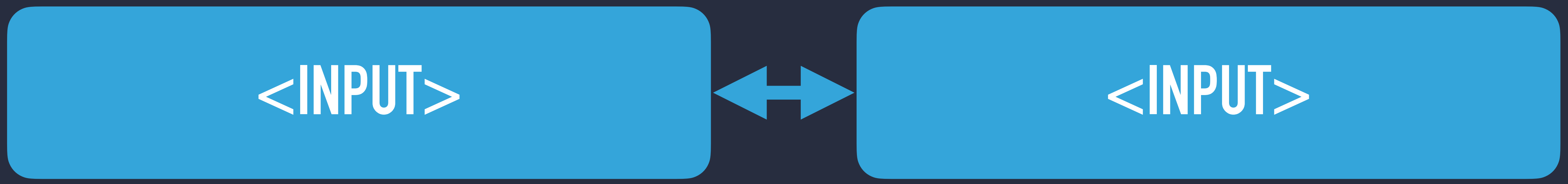
matches? --> yes!

```
html.div([], [
  html.label([], [
    html.span([], [text("Your name:")]),
    html.input([
      attribute.value(model.name),
      attribute.on_input(UserChangedName)
    ])
  ])
])
```

NORMAL ELEMENTS & FRAGMENTS



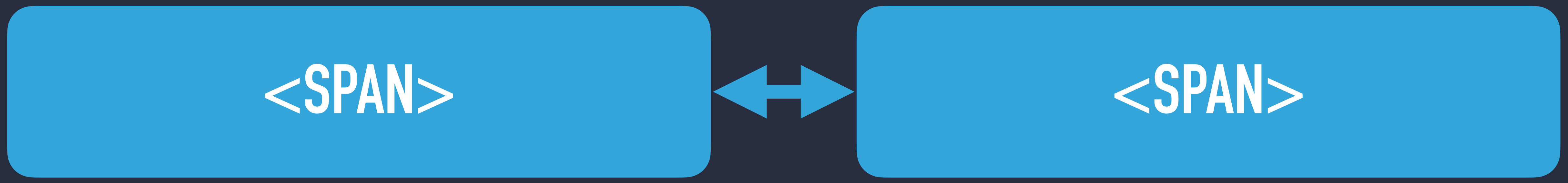
matches? --> yes!



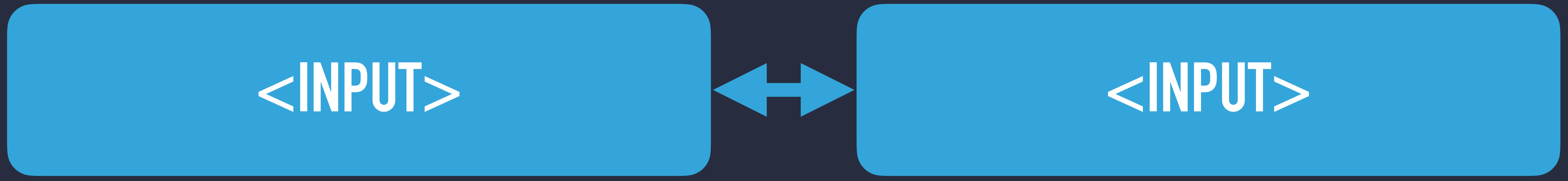
matches? --> different value!

```
html.div([], [
  html.label([], [
    html.span([], [text("Your name:")]),
    html.input([
      attribute.value(model.name),
      attribute.on_input(UserChangedName)
    ])
  ])
])
```

NORMAL ELEMENTS & FRAGMENTS



matches? --> yes!



matches? --> different value!

<label> has 1 different child

<div> has 1 different child

```
html.div([], [
  html.label([], [
    html.span([], [text("Your name:")]),
    html.input([
      attribute.value(model.name),
      attribute.on_input(UserChangedName)
    ])
  ])
])
```

NORMAL ELEMENTS & FRAGMENTS

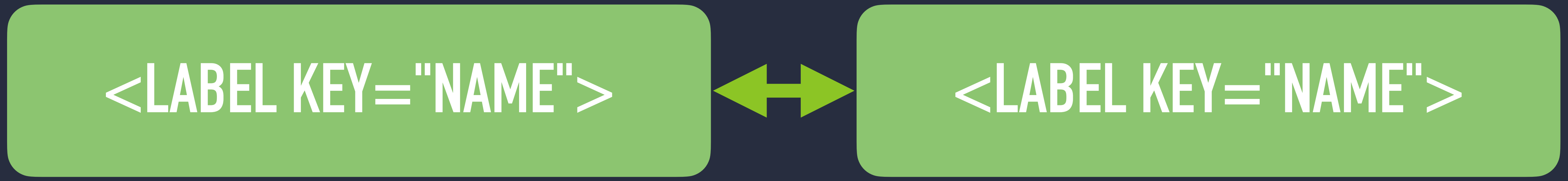
- Elements can be diffed by walking both child lists in parallel
- Extra children at the end are inserted or removed
- Children with different tags or type get replaced
- Fragments work exactly like normal elements
- Comparing them is extremely fast!

KEYED ELEMENTS & FRAGMENTS

Keys!

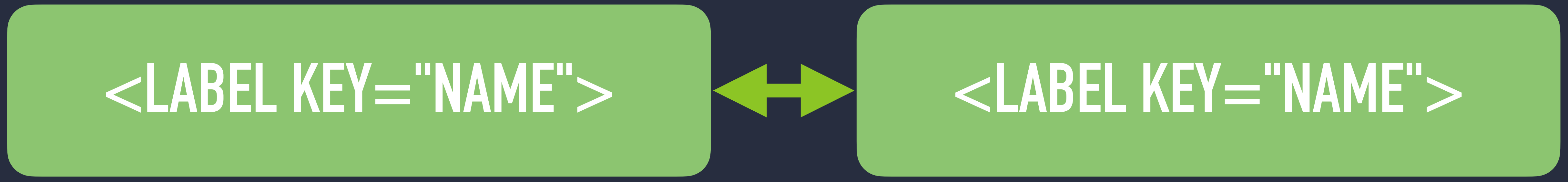
```
keyed.div([], [
  #("name", html.label([], [
    todo
  ])),
  #("birthday", html.label([], [
    todo
  ]))
])
```

KEYED ELEMENTS & FRAGMENTS



```
keyed.div([], [
  #("name", html.label([], [
   todo
  ])),
  #("birthday", html.label([], [
   todo
  ]))
])
```

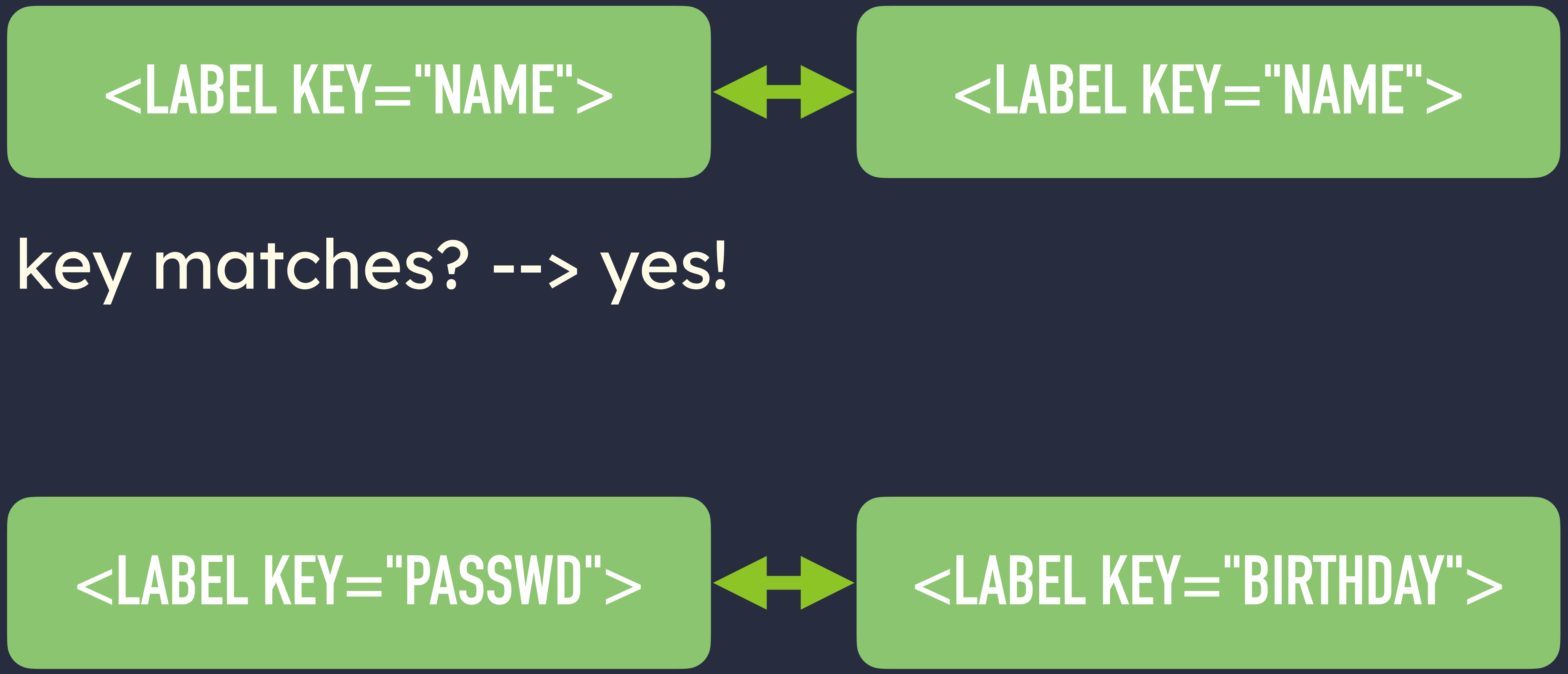
KEYED ELEMENTS & FRAGMENTS



key matches? --> yes!

```
keyed.div([], [
  #("name", html.label([], [
   todo
  ])),
  #("birthday", html.label([], [
   todo
  ]))
])
```

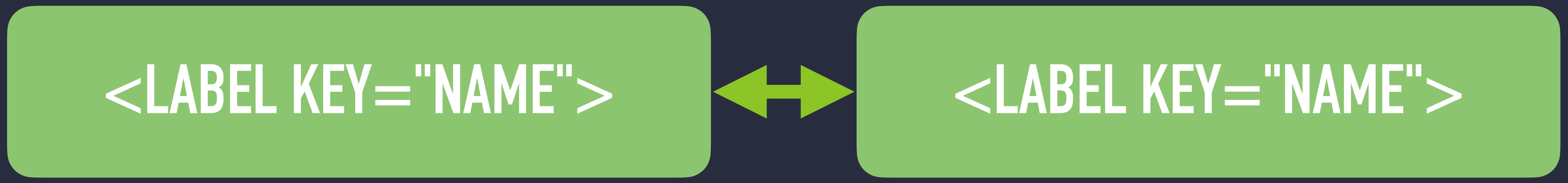
KEYED ELEMENTS & FRAGMENTS



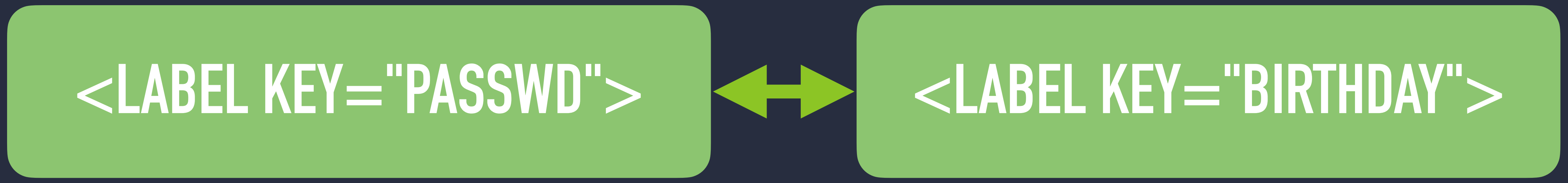
key matches? --> yes!

```
keyed.div([], [
  #("name", html.label([], [
   todo
  ])),
  #("birthday", html.label([], [
   todo
  ]))
])
```

KEYED ELEMENTS & FRAGMENTS



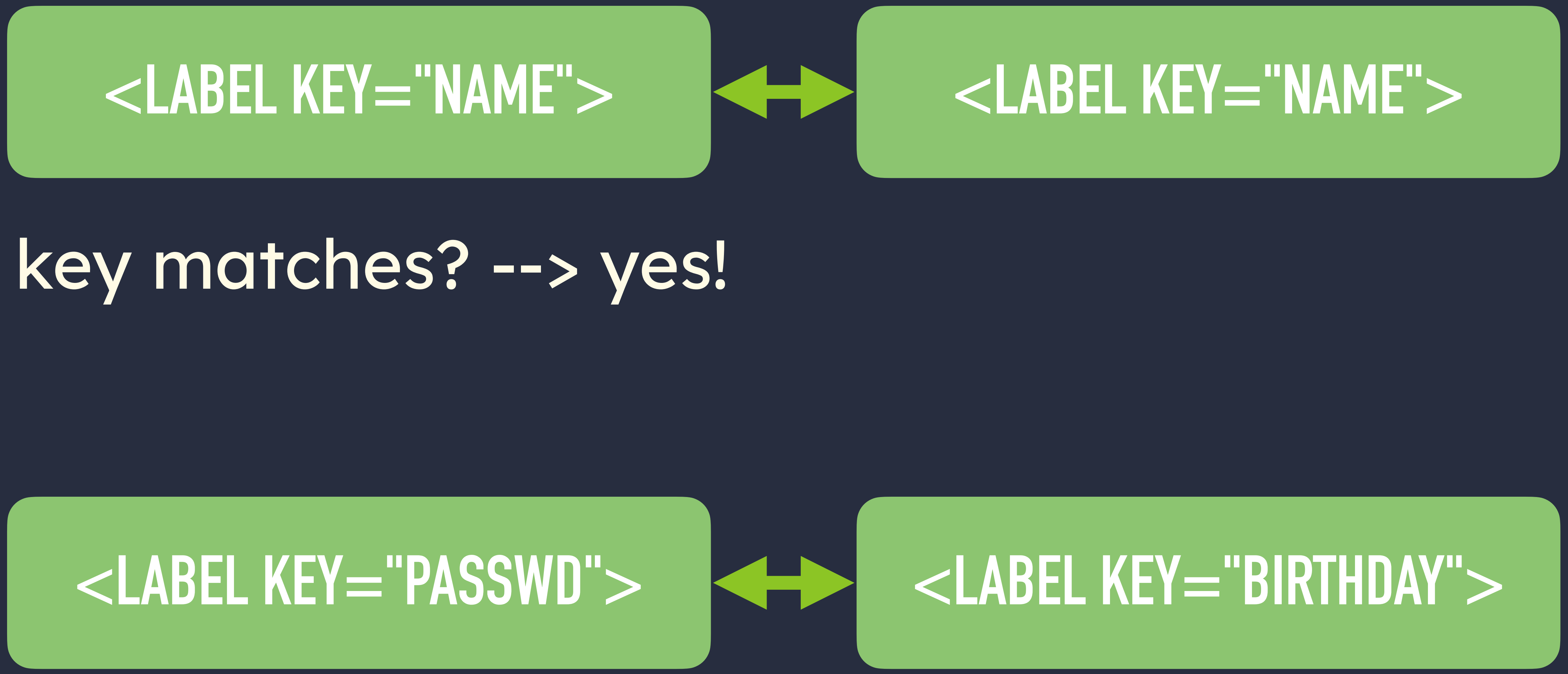
key matches? --> yes!



key matches? --> different key!

```
keyed.div([], [
  #("name", html.label([], [
   todo
  ])),
  #("birthday", html.label([], [
   todo
  ]))
])
```

KEYED ELEMENTS & FRAGMENTS



key matches? --> yes!

key matches? --> different key!

replace entire element

```
keyed.div([], [
  #("name", html.label([], [
   todo
  ])),
  #("birthday", html.label([], [
   todo
  ]))
])
```

KEYED ELEMENTS & FRAGMENTS

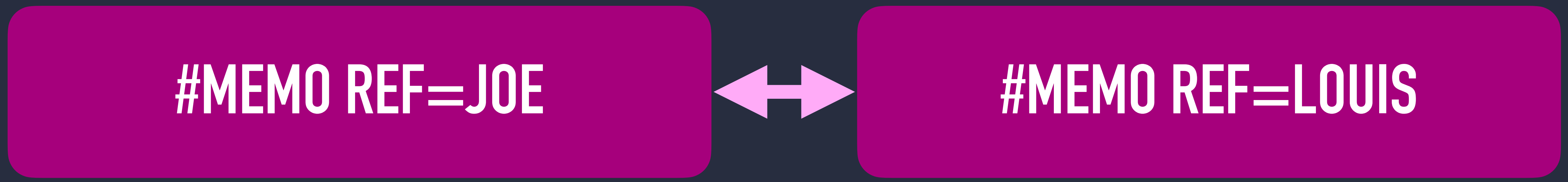
- Elements are matched up by key not by index
- If the same key exists at a different index, the child is moved
- Extra keys are removed or inserted
- Preserves state & identity*
- Building and comparing dictionaries has some overhead

* except in Safari :-)

MEMO NODES

```
html.div([], [
  element.memo([ref(model.name)], fn() {
    html.label([], [
      todo
    ])
  }),
 ])
```

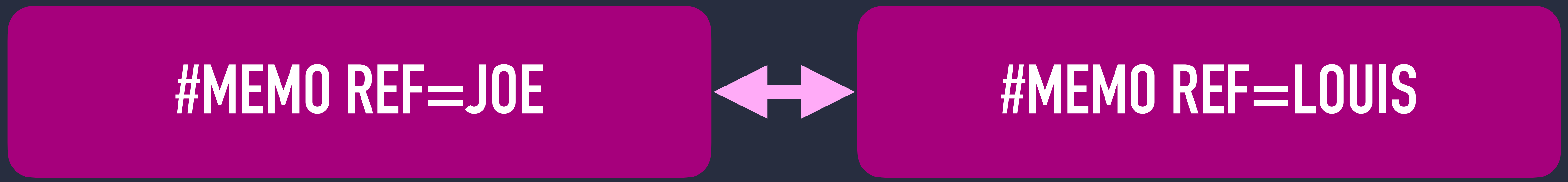
MEMO NODES



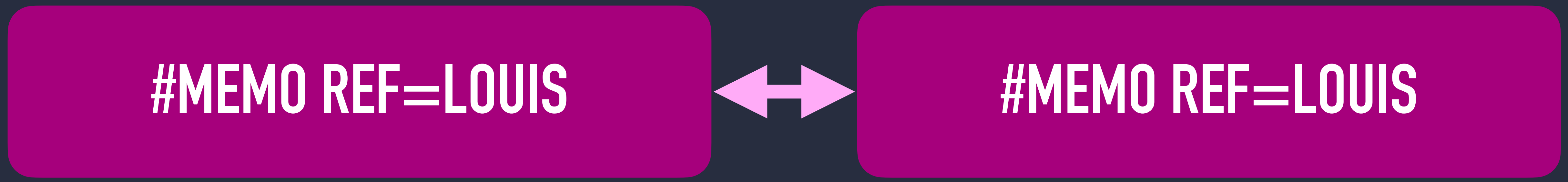
refs don't match? --> diff children!

```
html.div([], [
  element.memo([ref(model.name)], fn() {
    html.label([], [
      todo
    ])
  }),
])
```

MEMO NODES



refs don't match? --> diff children!



refs match -> everything can be skipped!

```
html.div([], [
  element.memo([ref(model.name)], fn() {
    html.label([], [
      todo
    ])
  }),
])
```

MEMO (AND MAP) NODES

- Memo elements are a boundary for state
- Map elements are a boundary for event listeners
- if the ref hasn't changed, view, diff and reconcile can be skipped entirely

MEMO (AND MAP) NODES

- Memo elements are a boundary for state
- Map elements are a boundary for event listeners
- if the ref hasn't changed, view, diff and reconcile can be skipped entirely

- Remember how I said dictionaries cause the most overhead?
  keyed elements, attributes, event listeners
- But: Memo nodes have to also be stored in… a dictionary.

WHEN TO USE WHAT

NORMAL ELEMENTS
FRAGMENTS

KEYED ELEMENTS
KEYED FRAGMENTS

MAP NODES

MEMO NODES

WHEN TO USE WHAT

Use them by default!
Fragments are good!

NORMAL ELEMENTS
FRAGMENTS

KEYED ELEMENTS
KEYED FRAGMENTS

MAP NODES

MEMO NODES

WHEN TO USE WHAT

Lists that are freq. shuffled
Preserving identity,
optimising page swaps

NORMAL ELEMENTS
FRAGMENTS

KEYED ELEMENTS
KEYED FRAGMENTS

MAP NODES

MEMO NODES

WHEN TO USE WHAT

NORMAL ELEMENTS
FRAGMENTS

KEYED ELEMENTS
KEYED FRAGMENTS

MAP NODES

MEMO NODES

wrap nested MVU "components"

WHEN TO USE WHAT

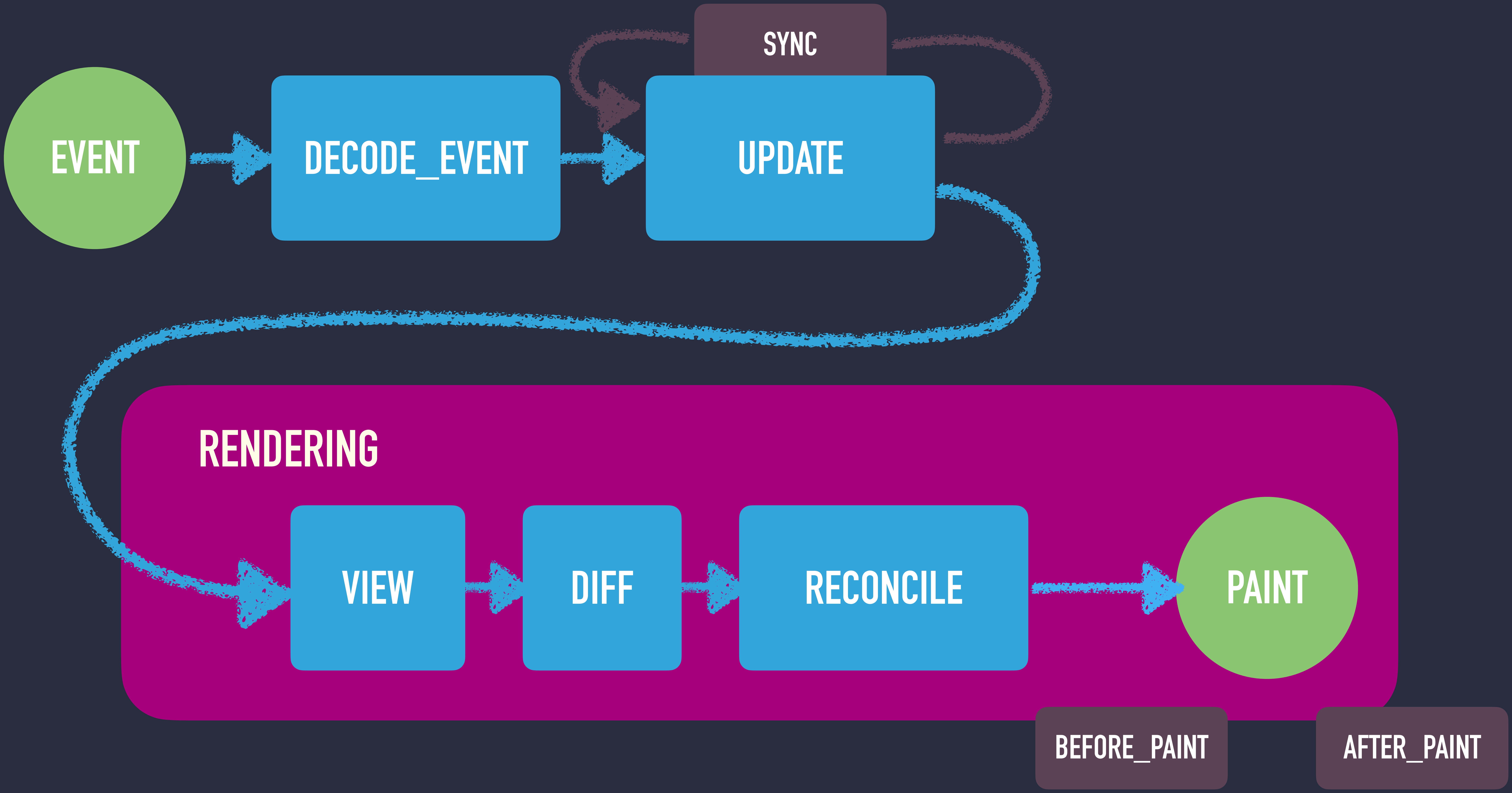
NORMAL ELEMENTS
FRAGMENTS

KEYED ELEMENTS
KEYED FRAGMENTS

MAP NODES

MEMO NODES

cache **large** chunks of parts
that change much more infrequently


SYNC
EVENT
DECODE_EVENT
UPDATE
RENDERING
VIEW
DIFF
RECONCILE
PAINT
BEFORE_PAINT
AFTER_PAINT

😔

I was promised trees!

THANK YOU ~ 💜