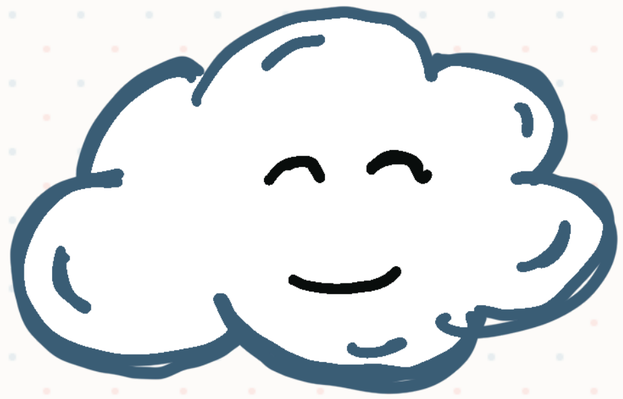
React Performance

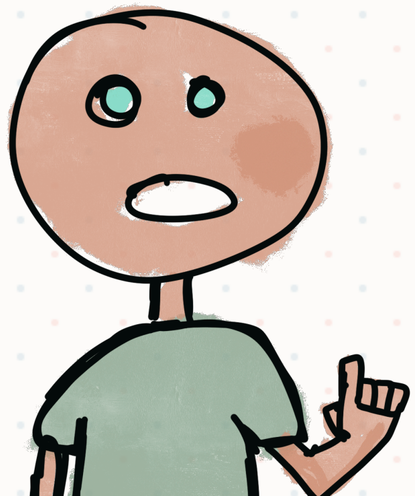
New && Improved

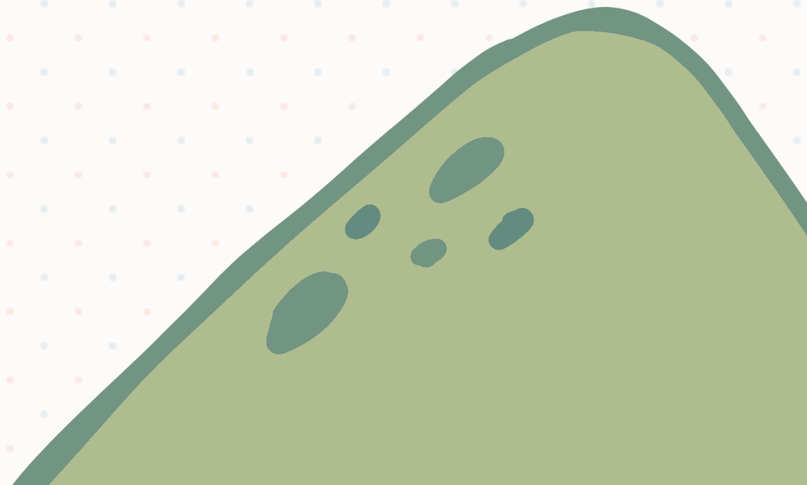
Who is this for?

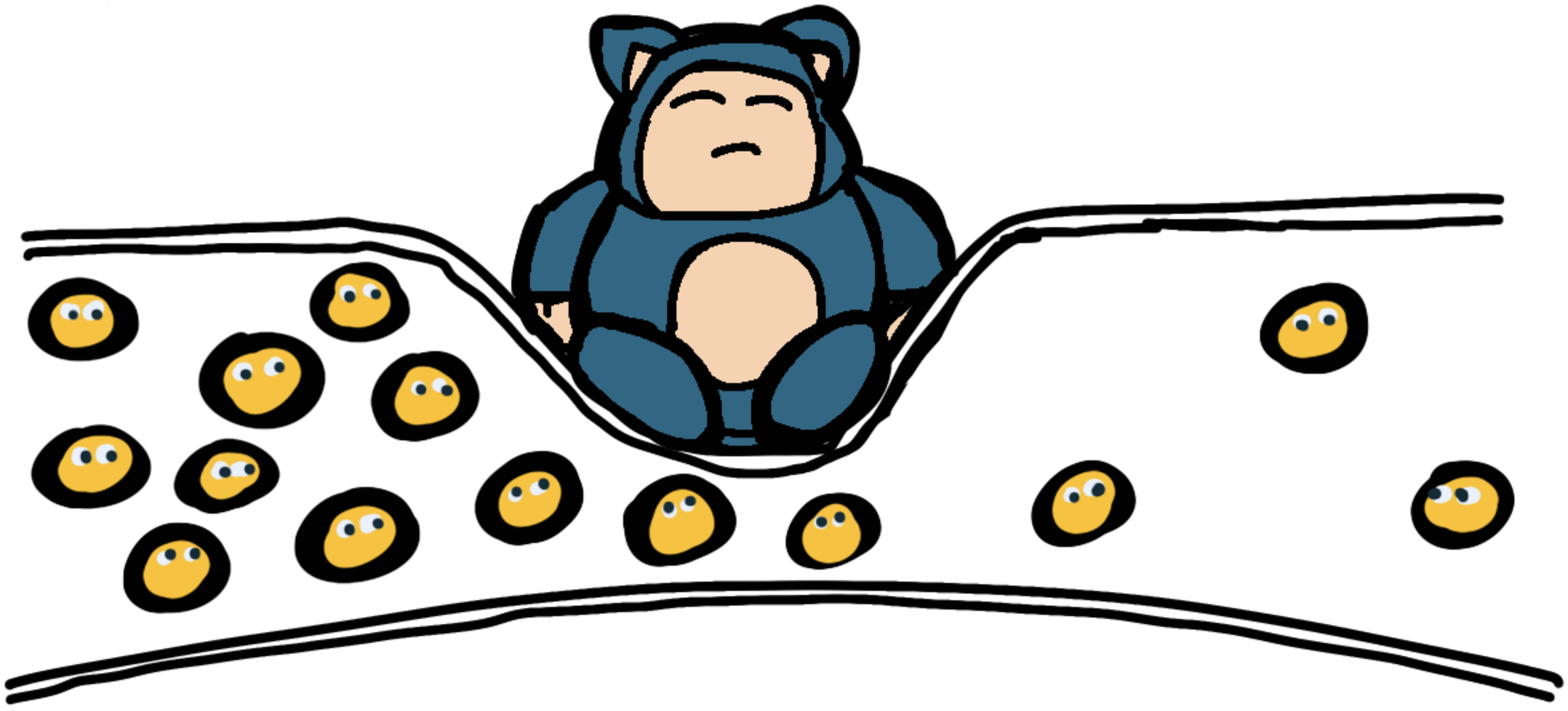


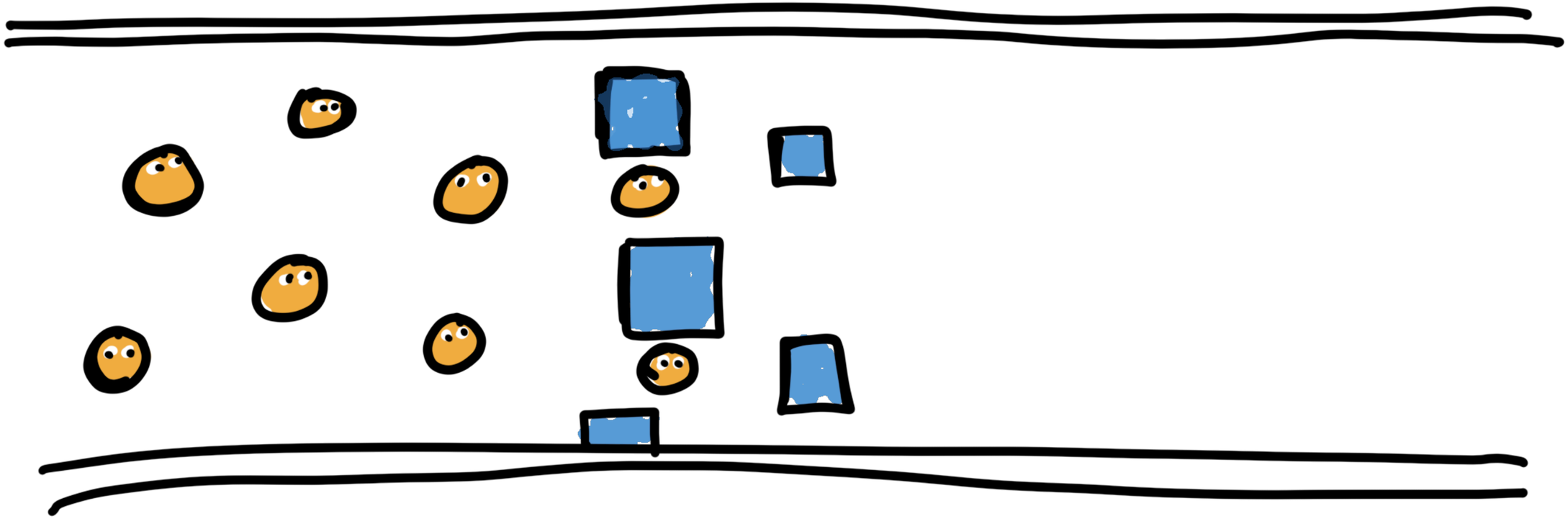
It’s for you, of course.

- You're looking for more insight into *how* React works.
- You want to build a skillset and/or toolset for diagnosing performance issues in your React application.
- You want to learn some ✨**best practices**✨ when building React applications in hopes of avoiding some performance issues down the road.



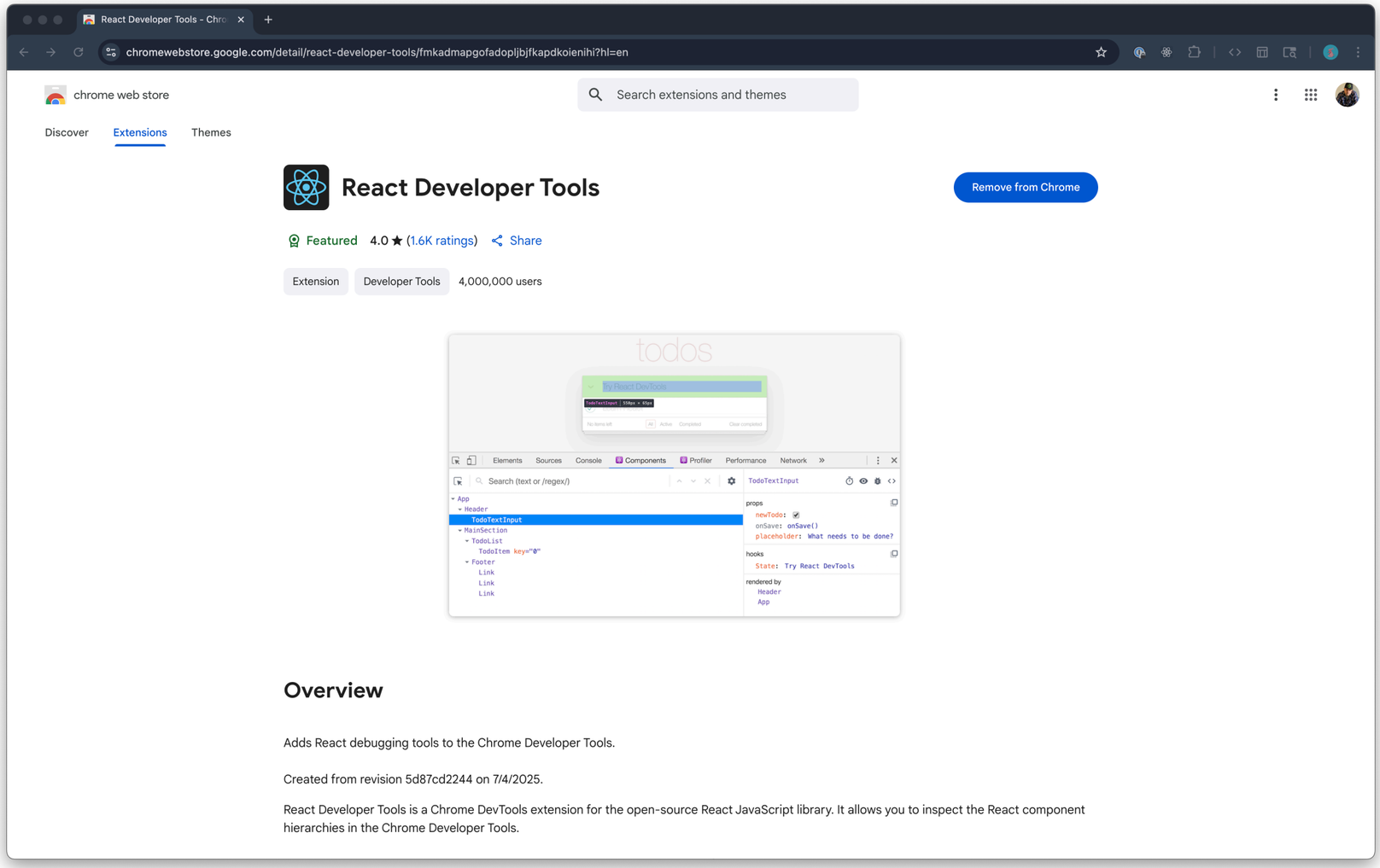






The React Developer Tools

Okay, then go get yourself the React Developer Tools.

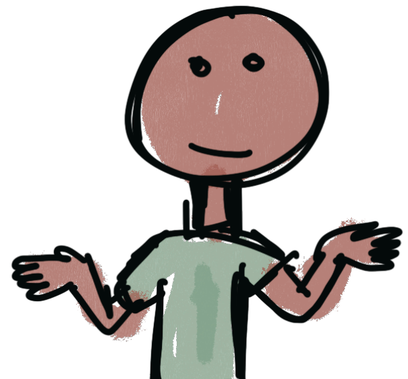


Overview

Adds React debugging tools to the Chrome Developer Tools.

Created from revision 5d87cd2244 on 7/4/2025.

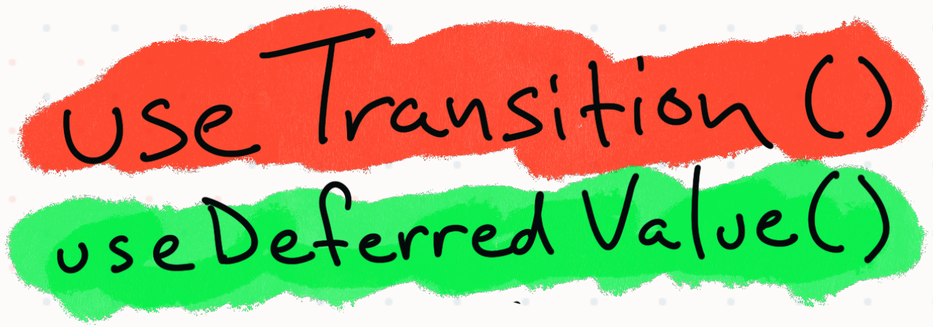
React Developer Tools is a Chrome DevTools extension for the open-source React JavaScript library. It allows you to inspect the React component hierarchies in the Chrome Developer Tools.



At a high level

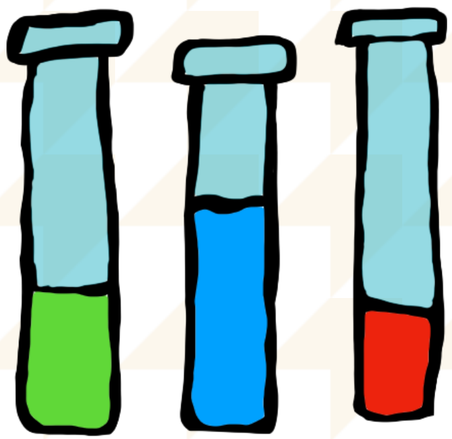
What are we going to cover?

- React has a bunch of tools for **caching and memoizing components**.
- But, you can also get a lot of benefits for free just in the ways you **structure** your component hierarchy and application state.
- Lastly, React has a bunch of super cool **concurrency features**.



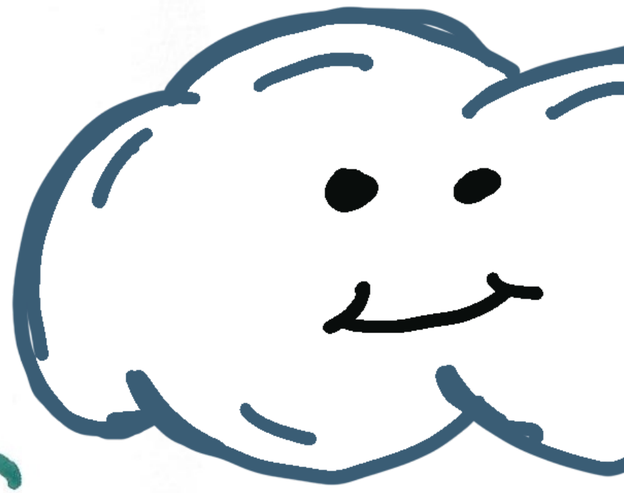
The Labs

Throughout our time together, we're going to work through a series of labs that are hyper-focused on one or two *specific* topics.



Steve's Rules for Performance

Steve's Golden Rule of Performance:



Not doing stuff is way faster than doing stuff.

Steve's Other Rule of Performance:

Feeling fast is pretty much as good as actually being fast.

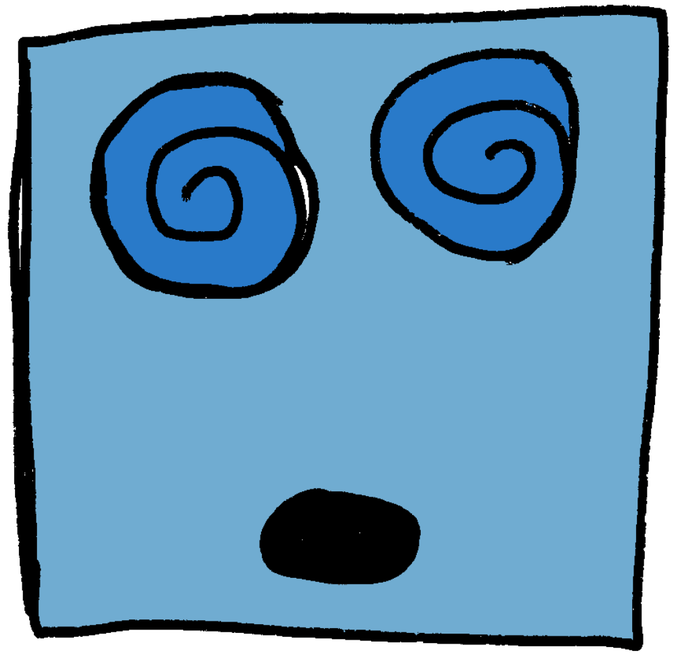
I am going to show you this at the end.

This will *also* be the last slide.

- If you can solve a problem with how you shape your *component hierarchy* or *state*—do that *first*.
- *Memoization* is a solid strategy *only* if the cost of checking pays for itself with the time you save rendering.
- Using the **Suspense API** to progressively load your application is a *good idea™*. And, more good stuff will come soon.
- The *Transition API* is there for you when you're *really* in a pickle.

How does React do what it does?

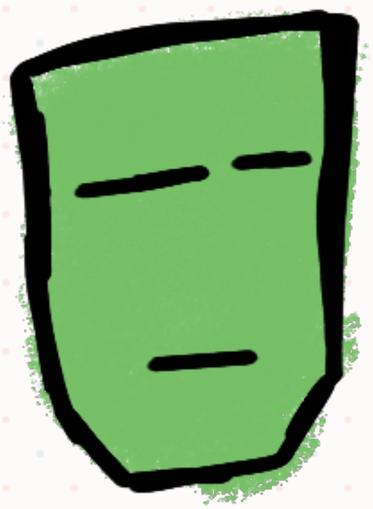
If our application never re-rendered, we wouldn't have any performance issues, right?

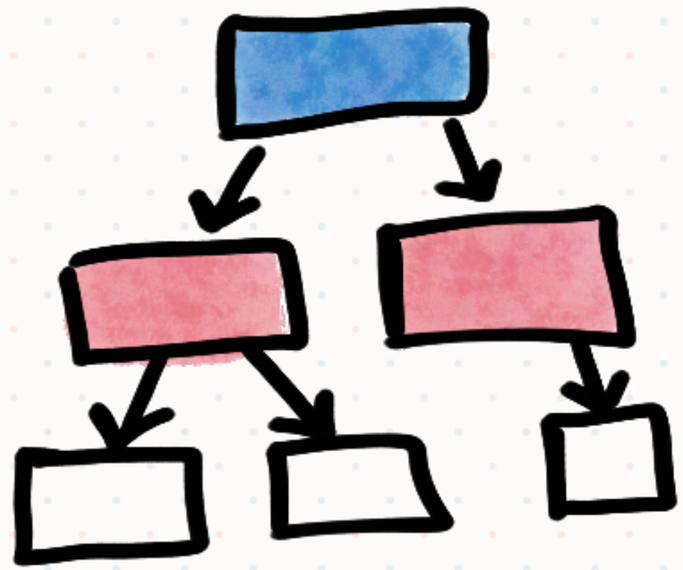


Anatomy of a Re-Render

State typically changes for one of three reasons.

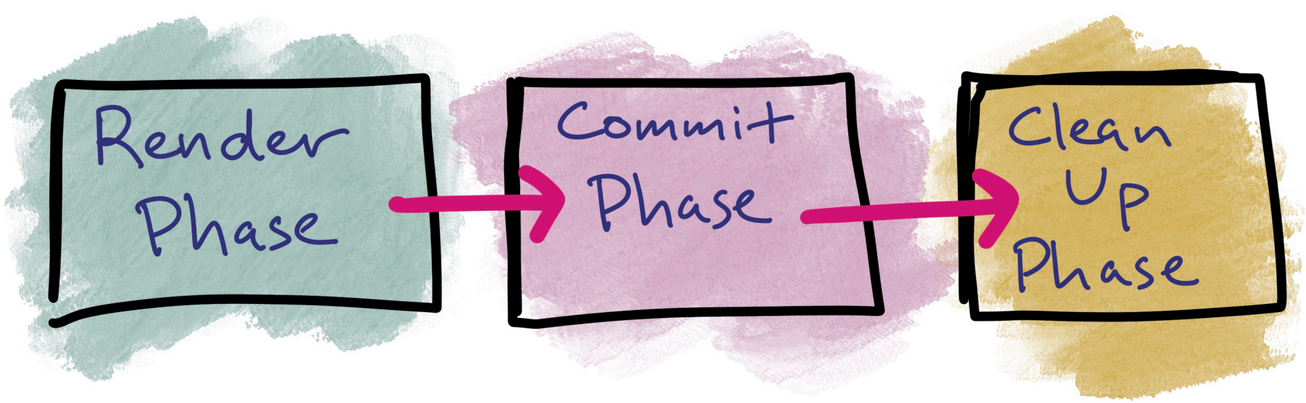
- Its state changed.
- The context changed.
- Its parent changed.

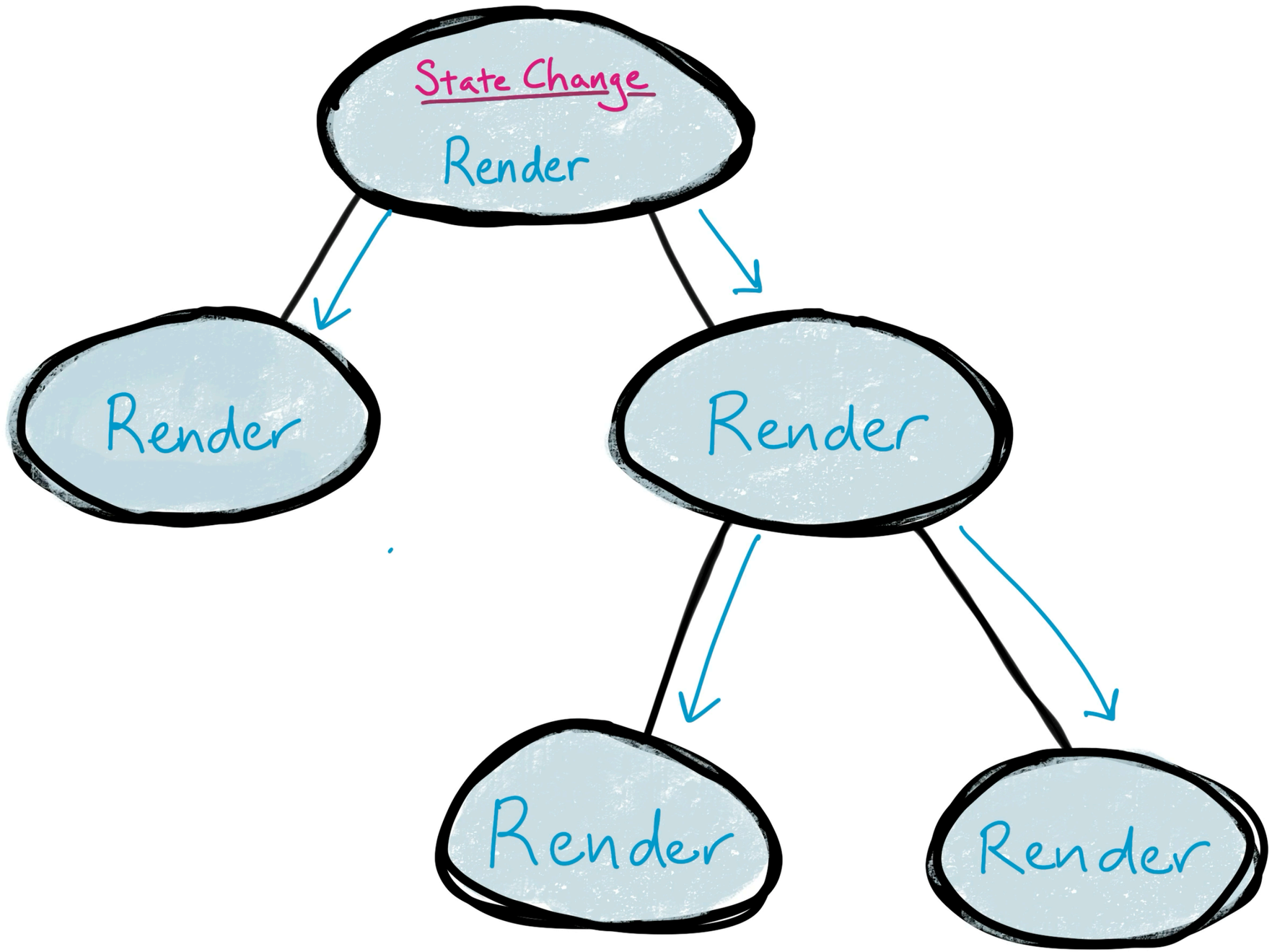




React's Rendering Cycle

How everything goes down.

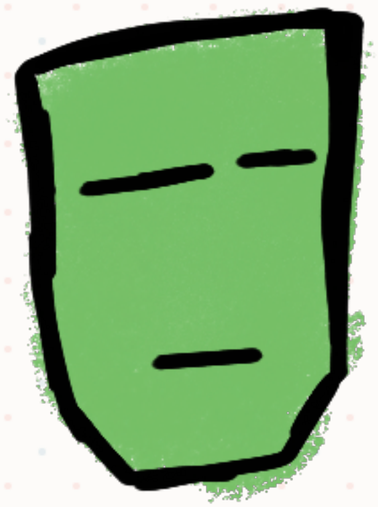



State Change
Render
Render
Render
Render
Render
Render

Anatomy of a Re-Render

State typically changes for one of three reasons.

- Its state changed.
- The context changed.
- ~~Its parent changed.~~
- Its *props* changed.



There are **really** only two types of state changes: *Necessary* and *unnecessary*.

I lied.

Necessary re-renders come in two flavors: **Urgent** and **non-urgent**.

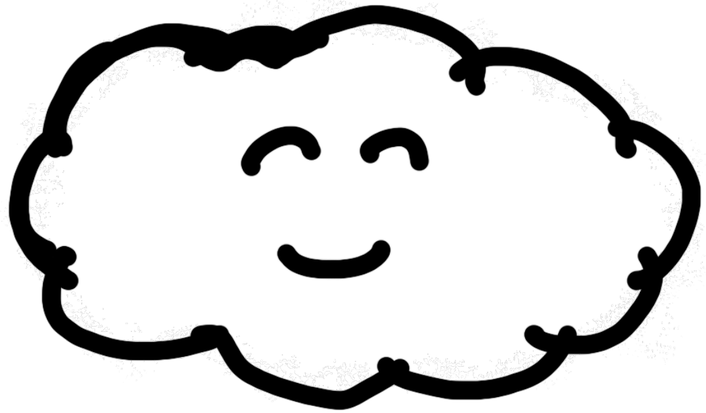
Necessary re-renders come in two flavors: *Urgent* and *non-urgent*.

So, then the name of the game is *eliminating* the unnecessary and *prioritizing* the urgent.

Some themes.

for web performance... and life.

- Not doing stuff is *way faster* than doing stuff. (Component hierarchy & state management)
- Seeing if you can skip doing stuff is *sometimes* less work than doing stuff. (Memoization + Compiler)
- You can *put off* doing stuff. (Suspense + Transitions)
- Load as much as you need and as little as you can get away with. (Lazy loading + bundle optimization)



But, first…

React Fiber

React Fiber is a cooperatively-scheduled rendering engine. Duh.

React Fiber

What does that even mean?

- Effectively, we're not necessarily reducing the time it takes to render the UI, we're just trying to be a bit smarter about it.

- React Fiber introduces this idea of priority as opposed to simply, dealing with requests in the order that they were received.

It's basically a bunch of plain functions, a linked-list tree, and a small scheduler in a trench coat.

React Fiber

What does that even mean?

- Effectively, we're not necessarily reducing the time it takes to render the UI, we're just trying to be a bit smarter about it.

- **React Fiber** introduces this idea of priority as opposed to simply, dealing with requests in the order that they were received.

- The key feature is that it can stop what it's doing and throw away an in-progress render whenever it feels like it.

Basically, if something more important comes along, it can turn its focus and put the less important stuff on pause or toss it out for now and start over later.

Inside of each render,
there are two trees.

The Rendering Phase

One thing at a time.

- Take the next Fiber.
- Run `beginWork(fiber, renderLanes)`. Basically, call the component function, derive the children.
- If there are children, descend. If not, bubble up via `completeWork` to finalize the node, update the DOM, and collect any effects.
- But, along the way—ask *Should I yield?* If yes, pause for a moment and let the browser have the wheel back for a second.
- Pick up where you left off.

Two Kinds of Updates

This is an over simplification, but go with it.

- Urgent Updates: User input and stuff like that.

- Transition Updates: Stuff explicitly marked as non-urgent by *you*.

Should I yield?

How does React decide when to yield?

- Every so often. Approximately after 5ms.
- Are there any higher priority tasks waiting?

Life in Fast Lane

React tries to categorize work and put it into lanes.

If something comes in at a *higher* priority lane, React goes ahead and turns it attention to the more important thing.

```
const SyncLane = 0b0000000000000000000000000000001; // Bit 1
const InputContinuousLane = 0b0000000000000000000000000000100; // Bit 3
const DefaultLane = 0b0000000000000000000000000010000; // Bit 5
const TransitionLane1 = 0b0000000000000000000000001000000; // Bit 7
```

Lanes

A hand-wavy look at priorities.

- SyncLane: Immediate, blocking updates with (e.g. `flushSync`)

- User actions (InputContinuousLane): Clicks, keypresses, dragging, scrolling.

- DefaultLane: The normal course of business.

- TransitionLanes: Stuff explicitly put into a lower priority lane.

- RetryLanes: Failed stuff that we're waiting to retry.

- IdleLane: The lowest priority work.

Lane Assignment

React puts things in their place.

- Assigns the update to the appropriate lane.
- Groups updates in the same lanes together.
- Processes the lanes from the highest priority to the lowest.

The Commit Phase™

Commits are never interrupted. We have a full tree of what the DOM should look like. Let’s make it happen.

That's because we're actively changing the DOM at this point and stopping in the middle could leave us in an inconsistent state.

Take a Picture

The Snapshot Phase

- React calls any lifecycle methods or hooks that need to read the DOM before React mutates it.
- No DOM writes yet, just reads.

Mutation

Let’s change the DOM.

- React applies all the changes it decided on during the render phase.
- Creates, updates, or deletes DOM nodes.
- Clean up any refs that no longer have nodes.
- Runs passive effect cleanups scheduled for this commit.

Layout Phase

Changed, but not painted.

- Perfect for measuring DOM layout or adjusting scroll positions
- If you ever used `useLayoutEffect`, this is where that goes down.

Passive Effect

All done—for now. Almost.

- After the browser has painted, React schedules *useEffect* callbacks.
- These run asynchronously, so they don't block the paint.
- Great for data fetching, subscriptions, logging, timers, etc.

Managing State

Checking to see if you can't just not do stuff.

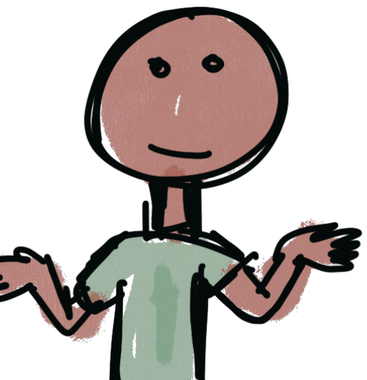
All things are not equal...

...in JavaScript.

```
{ foo: 1, bar: 2 } ≠ { foo: 1, bar: 2 }
[1,2,3] ≠ [1,2,3]
() ⇒ {} ≠ () ⇒ {}
```

Let’s not get carried away with memoization

Checking to see if you need to do stuff is *technically* doing stuff.



Some more tools

React has hooks to help.

- `useMemo()`: If it was expensive to get this value or it could trigger a render, but it's really no different than last time—then just use the value we had last time,.
- `useCallback()`: Actually don't whip up a new function if nothing has changed.

Now, let's use the Context API!

On normalizing your state.

posts.json — hottest-takes

application.jsx | comment.jsx | post.jsx | tailwind.config.js | index.js | add-comment.jsx | posts.json | users.ts | add-post.jsx | add-user.jsx

src > api > posts.json > {} 0

```
[
  {
    "body": "Qui magnam fugit necessitatibus aut tenetur, at beatae blanditiis at est qui consectetur maiores quae
    "title": "Vel sequi qui error sed voluptatem omnr.",
    "id": "1",
    "user": {
      "lastName": "Hasegawa",
      "firstName": "Hiromi",
      "id": "1",
      "username": "Hiromi_Hasegawa",
      "comments": []
    },
    "comments": [
      {
        "text": "Nemo sed sapiente enim aliquid omnis exercitationem, consectetur quaerat consequatur at aliquid h
        "id": "1",
        "user": {
          "lastName": "Rumbelow",
          "firstName": "Thawi",
          "id": "2",
          "username": "Thawi.Rumbelow37",
          "comments": [
            {
              "text": "Nemo sed sapiente enim aliquid omnis exercitationem, consectetur quaerat consequatur at ali
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Filter (e.g. text, **/*.ts, !**/node_modules/**)

No problems have been detected in the workspace.

main 0↓ 5↑ ⊗ 0 ⚠ 0 #340b0b | Blame Steve Kinney (7 hours ago) | Ln 5, Col 15 | Spaces: 2 | UTF-8 | LF | JSON | Prettier

normalized.json — hottest-takes

EXPLORER

HOTTEST-TAKES

Type to search

node_modules
public
android-chrome-192x192.png
android-chrome-512x512.png
apple-touch-icon.png
favicon-16x16.png
favicon-32x32.png
favicon.ico
index.html
logo192.png
logo512.png
manifest.json
robots.txt
src
api
index.js
normalized.json
posts.json
serializer.js
users.json
components
add-comment.jsx
add-post.jsx
add-user.jsx
application.jsx
comment.jsx
post.jsx
posts.jsx
user.jsx
users.jsx
features
posts.ts
users.ts
OUTLINE
TIMELINE
NPM SCRIPTS
HIDDEN ITEMS

application.jsx | comment.jsx | post.jsx | tailwind.config.js | index.js | add-comment.jsx | normalized.json | users.ts

src > api > normalized.json > {} users > {} entities > {} 1 > [ ] commentIds

```
{
  "users": {
    "ids": ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"],
    "entities": {
      "1": {
        "commentIds": [
          "6",
          "7",
          "8",
          "9",
          "10",
          "66",
          "67",
          "68",
          "69",
          "70",
          "81",
          "82",
          "83",
          "84",
          "85"
        ],
        "postIds": ["1", "2"],
        "lastName": "Mahato",
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

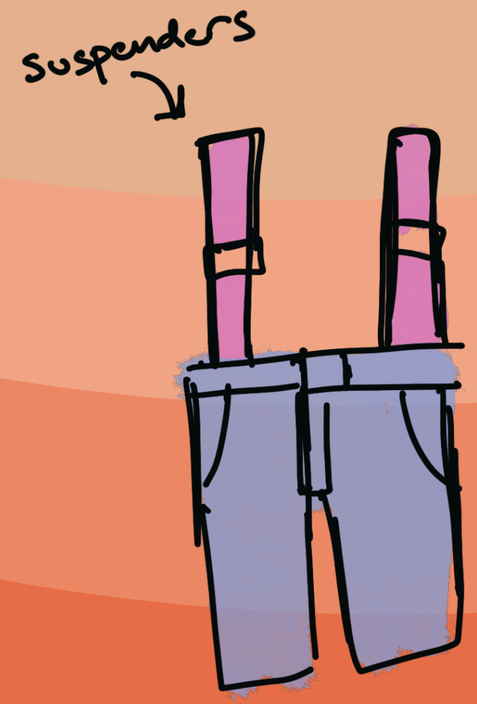
Filter (e.g. text, **/*.ts, !**/node_modules/**)

No problems have been detected in the workspace.

main 0↓ 5↑ ⊗ 0 ⚠ 0 #340b0b

Blame Steve Kinney (7 hours ago) Ln 14, Col 16 Spaces: 2 UTF-8 LF {} JSON Prettier

Keeping you in Suspense.



Suspense

`Suspense` is a React component that displays a fallback until its children have finished loading.

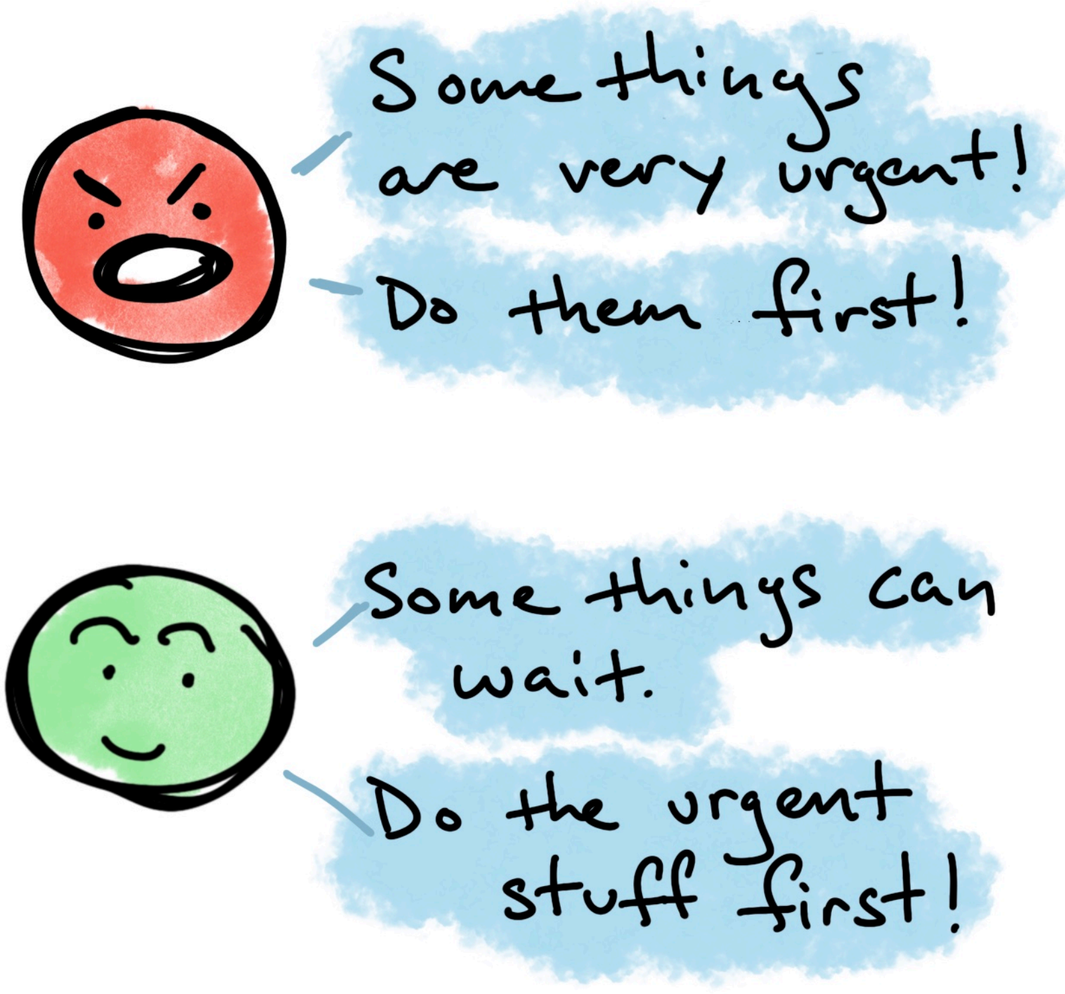
```
<Suspense fallback={<Loading />}>
  <SomeComponent />
</Suspense>
```

Usage

Displaying a fallback while something is loading

Transitioning to another way to do stuff later.


Everything is urgent!
Do it all!


Some things are very urgent!
Do them first!
Some things can wait.
Do the urgent stuff first!

Two more hooks

If I may quote from the documentation

- `startTransition()` is used when triggering an update (i.e. setState) in an event handler.

- `useDeferredValue()` is used when receiving new data from a parent component (or an earlier hook in the same component).

In conclusion

The actual last slide.

- If you can solve a problem with how you shape your *component hierarchy* or *state*—do that *first*.
- *Memoization* is a solid strategy *only* if the cost of checking pays for itself with the time you save rendering.
- Using the **Suspense API** to progressively load your application is a *good idea™*. And, more good stuff will come soon.
- The *Transition API* is there for you when you're *really* in a pickle.