How to build React

(only a simplified one)

Disclaimer

The stuff presented in this DevSesh does not come from React core team.

Sam doesn’t guarantee that this DevSesh presents how React actually works.

> Instead, it presents how Sam’s simplified version of React works.
>
> Hopefully they are similar enough.

The world without React 1

```
<input id="name" />
<button onclick="syncName()">Sync</button>
<div id="display" />
<script>
function syncName() {
  document.getElementById('display').innerText =
      document.getElementById('name').value;
}
</script>
```

The world without React 2

```
function updateDOMWithLatestData(data) {
  let html = ‘’;
  for (const card of data) {
    html +=
      `<div><a href=”mailto:${card.email}”>name: ${card.name}</a></div>`;
  }
  document.getElementById(‘main-data-container’).innerHTML = html;
}
```

The world without React 3

```
function updateDOMWithLatestDataWithInteractions(data) {
  let html = '';
  for (const card of data) {
    html += `<div id="card-${card.id}">${card.name}</div>
    <input id="input-${card.id}" onchange="update('${card.id}')" />`;
  }
  document.getElementById('main-data-container').innerHTML = html;
}
function update(id) {
  document.getElementById('card-' + id).innerText =
        document.getElementById('input-' + id).value;
}
```

React Example

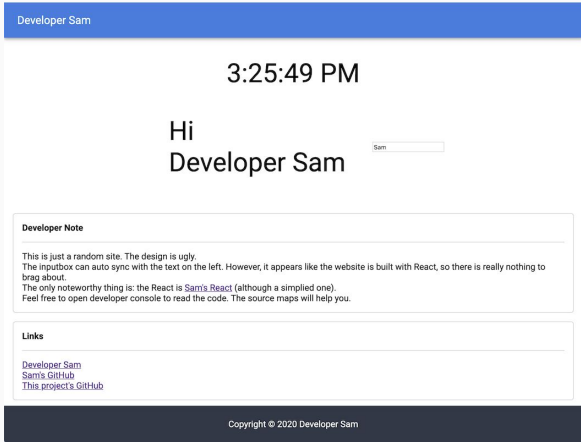
```tsx
const HiDiv = ({ name }: { readonly name: string }) => (
  <div className="hi">
    <span>Hi</span>
    <div>
      <span children="Developer " />
      <span>{name}</span>
    </div>
  </div>
);

const MyInput = ({ name, onChange }: { readonly name: string; readonly onChange: (name: string) => void }) => {
  const onInputChange = (event: Event): void => onChange((event.currentTarget as HTMLInputElement).value);
  return <input value={name} onChange={onInputChange} />;
};

const timeToString = (date: Date): string => `${date.getHours()}:${date.getMinutes()}:${date.getSeconds()}:${Math.floor(date.getMilliseconds() / 100) * 100}`;

const MyTime = () => {
  const [time, setTime] = useState(timeToString(new Date()));

  useEffect(() => {
    setTimeout(() => setTime(timeToString(new Date())), 200);
  });

  return (
    <div className="center">
      <span>{time}</span>
    </div>
  );
};

const App = () => {
  const [name, setName] = useState('Sam');

  useEffect(() => { document.title = `Hello, ${name}`;  });

  if (name.startsWith('Developer')) {
    return <span children="'Developer' is already in the template! Why Repeat?!" />;
  }

  return (
    <div>
      <MyTime />
      <div className="center">
        <HiDiv name={name} />
        <MyInput name={name} onChange={setName} />
      </div>
    </div>
  );
};
```

HiDiv ⇒ Presentational Component

MyInput ⇒ Pattern of Lifting State Up

MyTime ⇒ useEffect Example

App ⇒ main container, single source of truth



Wait? Wat?

- How does React sync your data with real DOM?
- How does useState know which component it’s in?

Also,

- Why props should be immutable?
- Why can’t I just push to an array directly?

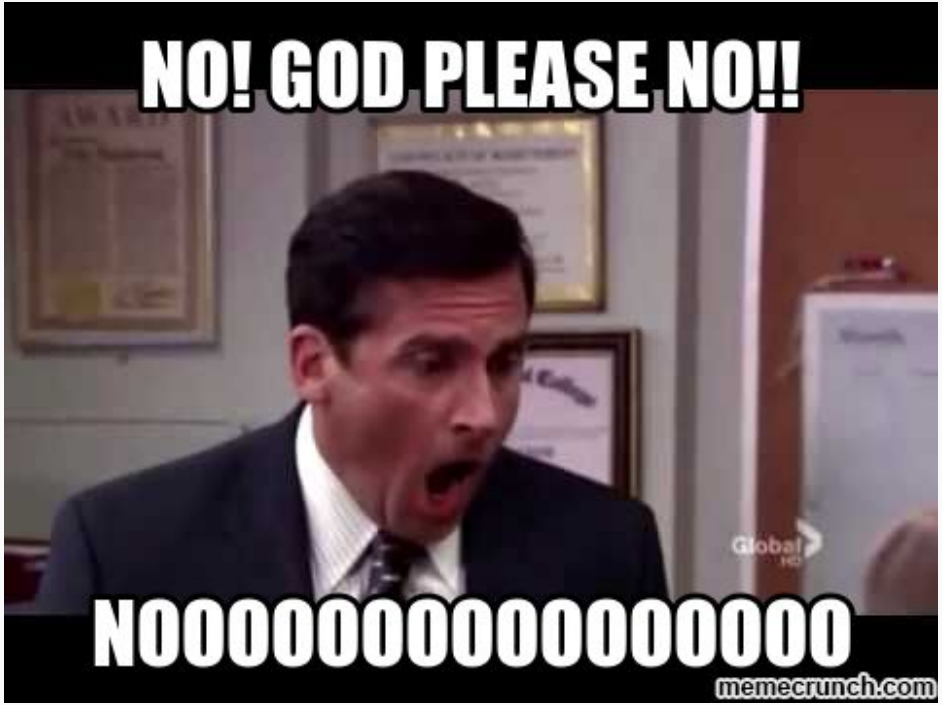
Hopefully you will know the answer by the end of this DevSesh!

Part 1: The Virtual DOM

What is JSX?

- Is it just HTML code?
- Is it just a prettified version of string template?
- Does React simply call the function to dump all HTML and do something like
  - `document.getElementById('root').innerHTML = dumpedHTML`

What is JSX? Answers



Why not HTML

Fact:

Manipulating DOM is orders of magnitude more expensive than in-memory operations.

Setting `innerHTML` may destroy DOM node that doesn't change at all.

In addition, what about those event listeners?

This is the real JSX

`createElement()`

```
React.createElement(
  type,
  [props],
  [...children]
)
```

Create and return a new React element of the given type. The type argument can be either a tag name string (such as `'div'` or `'span'`), a React component type (a class or a function), or a React fragment type.

Code written with JSX will be converted to use `React.createElement()`. You will not typically invoke `React.createElement()` directly if you are using JSX. See React Without JSX to learn more.

https://reactjs.org/docs/react-api.html#createelement

JSX are plain objects

```
console.log(
  <ReactReduxProvider store={store}>
    <MaterialThemedApp
      title="Developer Sam"
      appBarPosition="fixed"
      styles={appStyles}
      buttons={buttons}
    >
      <FirstPage />
      <Suspense fallback={null}>
        <ProjectsSection />
      </Suspense>
      <Suspense fallback={null}>
        <TechTalkSection />
      </Suspense>
      <Suspense fallback={null}>
        <TimelineSection />
      </Suspense>
      <Suspense fallback={null}>
        <WebTerminal />
      </Suspense>
    </MaterialThemedApp>
  </ReactReduxProvider>
);
```

```
▼ {$$typeof: Symbol(react.element), key: null, ref: null, props: {…}, type: ƒ, …}
    $$typeof: Symbol(react.element)
    key: null
  ▼ props:
    ▼ children:
        $$typeof: Symbol(react.element)
        key: null
      ▶ props: {title: "Developer Sam", appBarPosition: "fixed", styles: {…}, buttons: {…}, children: Array(5)}
        ref: null
      ▶ type: ({ title, styles, appBarPosition = 'static', buttons, children }) => {…}
        _owner: null
      ▶ _store: {validated: true}
        _self: null
      ▶ _source: {fileName: "/Users/sam/Desktop/workplace/website/packages/www/src/App.tsx", lineNumber: 34, columnNumber: 5}
      ▶ __proto__: Object
    ▶ store: {dispatch: ƒ, subscribe: ƒ, getState: ƒ, replaceReducer: ƒ, Symbol(observable): ƒ}
    ▶ __proto__: Object
    ref: null
  ▶ type: ƒ Provider(_ref)
    _owner: null
  ▶ _store: {validated: false}
    _self: null
  ▶ _source: {fileName: "/Users/sam/Desktop/workplace/website/packages/www/src/App.tsx", lineNumber: 33, columnNumber: 3}
```

Characteristics of JSX

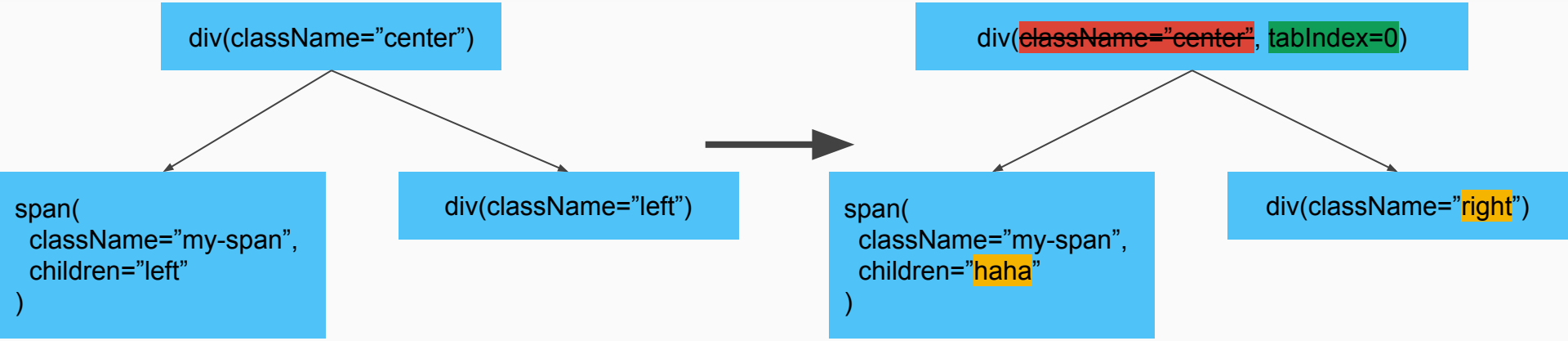
They are plain objects.

They are never converted to actual HTML (except in server side rendering)

They describe what the UI should look like.

JSX alone does nothing. It’s up to some engine and runtime to turn that into actual DOM elements.

The picture of virtual DOM diffing



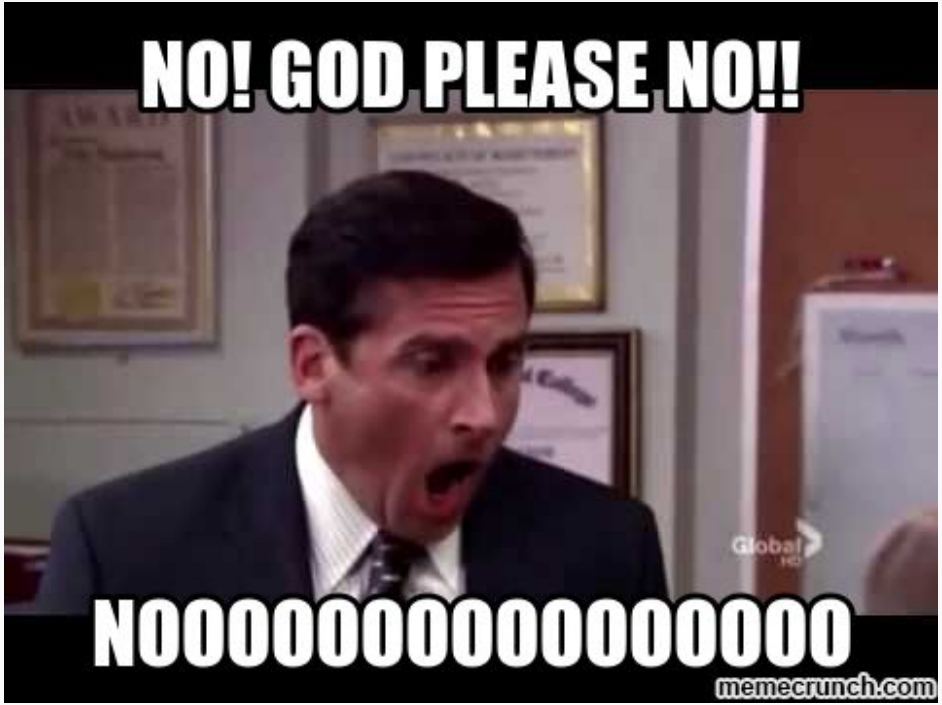
React can then do (assuming we track real DOM somewhere):

```
rootDiv.className = ‘’; rootDiv.tabIndex = 0;
span.innerText = ‘haha’;
childDiv.className = ‘right’;
```

Just ensure you are following

Is JSX Virtual DOM?

Answer



JSX is not virtual DOM!

JSX:

```
<MyContainer>
  <span>Hi</span>
</MyContainer>
```

Virtual DOM:

```
<div className=”pretty-container”>
  <span>Hi</span>
</div>
```

Fact: JSX is lazily evaluated to virtual DOM!

Part 2:
The Virtual DOM and Lifecycles

Surprise!

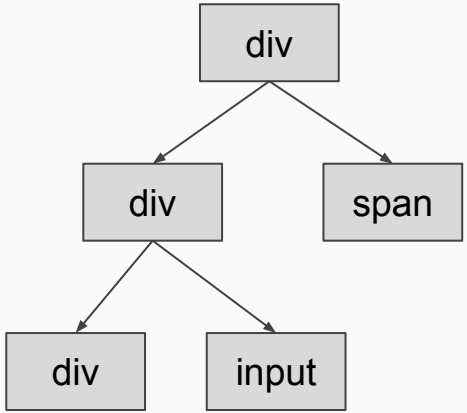
Think you can get away with reasoning about lifecycle methods with Hooks?

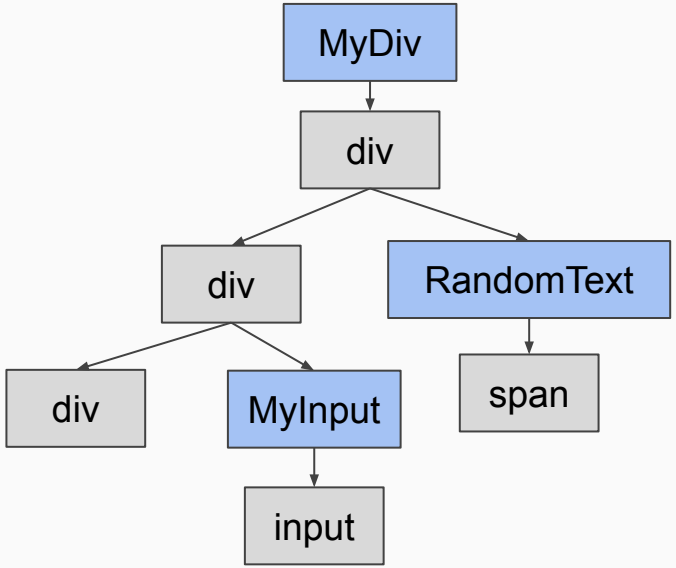
Not possible when you dive into implementation!

What is virtual DOM?

Directly answering this question might be hard, let’s start by choosing.

Which one is virtual DOM?





What happens during state changes?

Suppose setState is run inside MyInput, what should happen?

Some obvious steps:

- Rerender
- Update virtual DOM

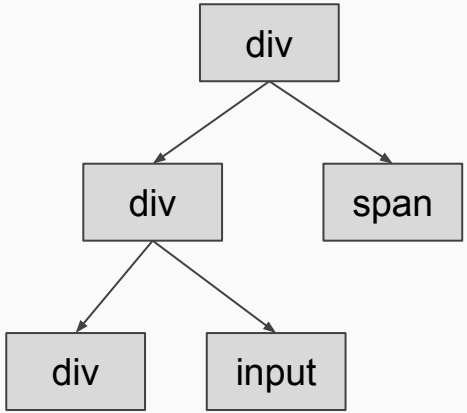
What happens during state changes?

Rerender is easy:

```
const newElement = MyInput(oldProps);
```

What about updating virtual DOM?

Virtual DOM - Alternative 1

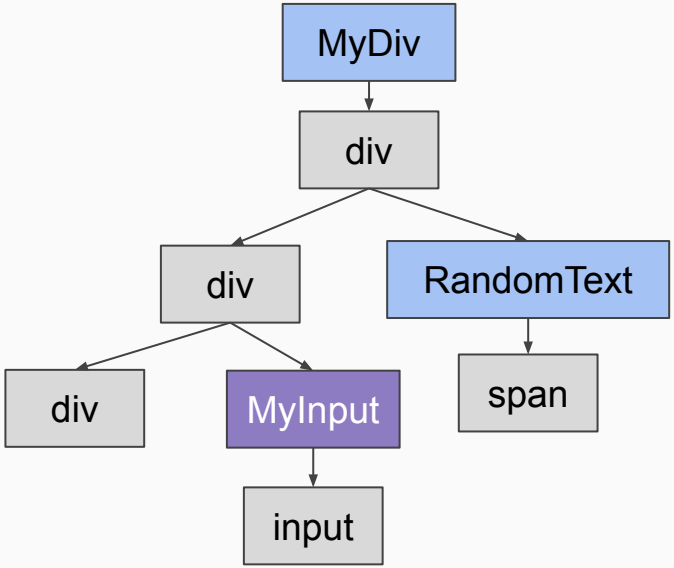


Which node should I update once I got newElement?

YOU DON'T KNOW

That's why this is not virtual DOM

Virtual DOM - Alternative 2




```
const newElement =
  MyInputNode.component(
    MyInputNode.oldProps
  );
// ...
MyInputNode.updateMyVirtualDOM(...);
```

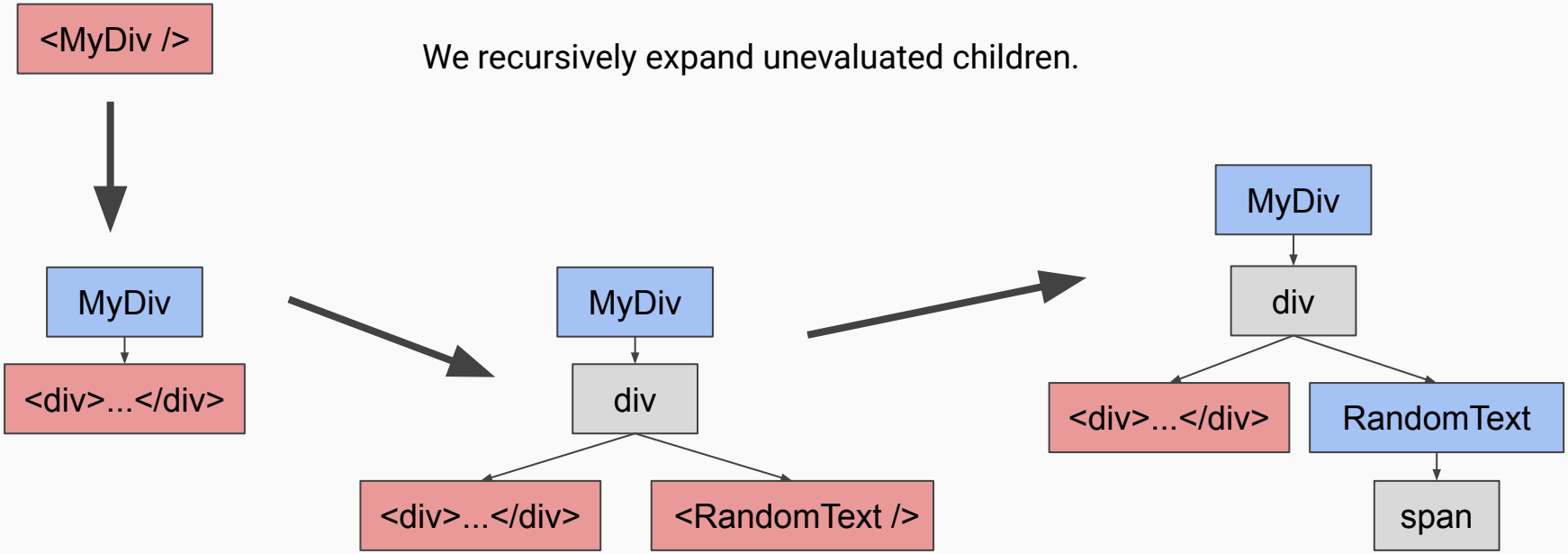
JSX ⇒ Virtual DOM

We already know

- JSX != Virtual DOM
- JSX is lazy

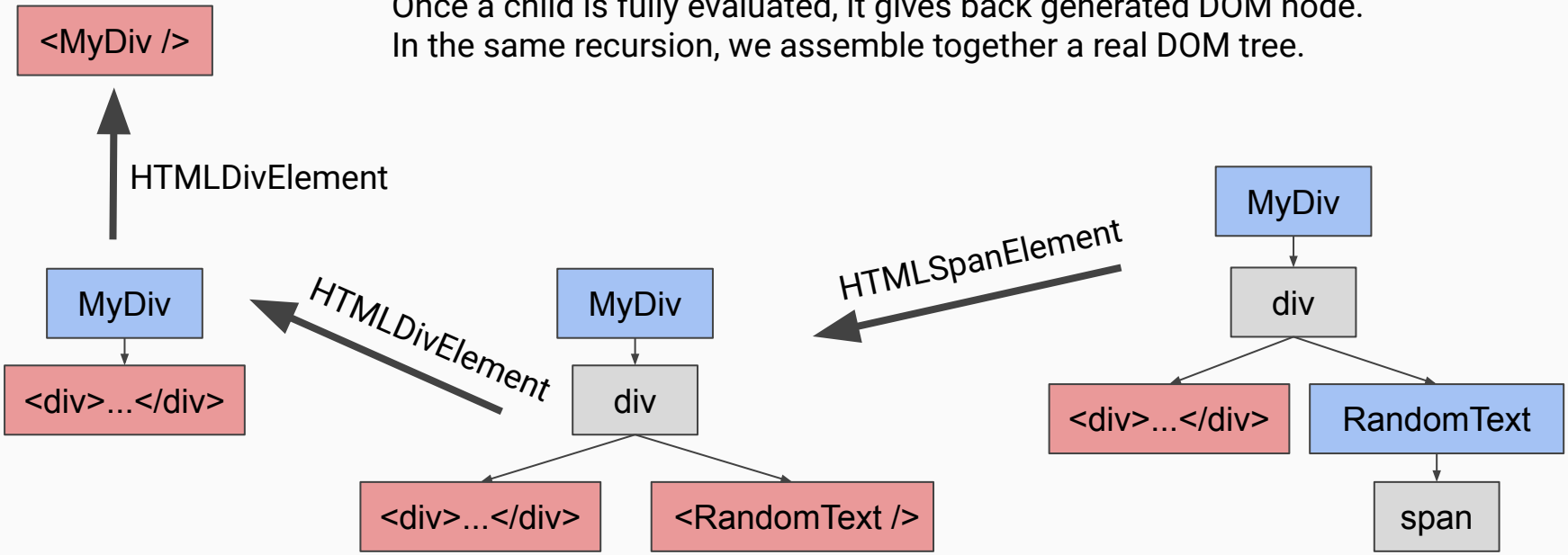
How do we “unlazify” JSX?

JSX ⇒ Virtual DOM (Mounting)



JSX ⇒ Virtual DOM (Mounting)

Once a child is fully evaluated, it gives back generated DOM node.
In the same recursion, we assemble together a real DOM tree.



Virtual DOM Update & Reconciliation

Goal:

The new virtual DOM and real DOM should be indistinguishable from the virtual DOM and real DOM generated from a clean mount with previous state.

Naive Update Strategy

1. Unmount
2. Mount

Why is it bad?

- Slow
- Incorrect: all states are lost.

When is it safe to unmount?

i.e. It’s OK to lose state

Answer: When the component type changes.

- From `div` to `span`
- From `div` to `MaterialCard`
- From `MaterialCard` to `div`
- From `MaterialCard` to `BootstrapCard`

What about children?

The real React does some clever bookkeeping to avoid unmounting children even when order changes.

For the purpose of this DevSesh:

- When old and new children have different length, nuke everything.
- Assume children don’t reorder themselves.
- No support for key.

User-defined component update

Q: What triggers an update? A: JSX Element.

Signature: `update(component: Component, element: JSX): void`

Strategy: recurse down!

```
const newElement = component.renderFunction(oldProps);

update(component.childComponent, newElement);
```

Intrinsic component update

Compare new and old props, only update changed props in real DOM

Recursively call `update` on children components

Part 3: The Hooks Runtime

useEffect

Type Signature: `useEffect(effect: () => void): void`

For simplicity, we don't include cleanup function and dependency array.

How to implement:

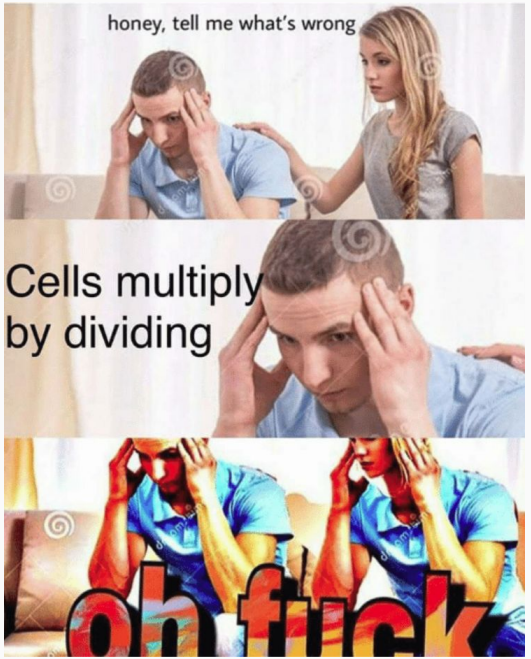
When called, put the effect into a global effect queue.

After rendering changes to the DOM, run and clear all effects in the queue.

useState

Dilemma:

- React component functions should be stateless
- useState introduces state
- 🤯



The runtime to the rescue!

React can inject runtime before it calls your component function!

React can run some cleanup code after it calls your component function!

That’s why JSX is lazily evaluated!

The environment of a render

```
// ...

const hooksRuntime = findExistingRuntimeOrCreateNew();

_HOOKS_RUNTIME = hooksRuntime;

const jsx = YourComponent(yourComponentProps);

storeHooksRuntimeSomewhere(hooksRuntime);

_HOOKS_RUNTIME = null;

// ...
```

Hooks runtime for `useState`

- An array of [state, setState]
- Current index

function useState(defaultValue) {

```
  if (0 <= _HOOKS_RUNTIME.currentIndex < _HOOKS_RUNTIME.stateArray.length) {
    const slot = _HOOKS_RUNTIME.stateArray[_HOOKS_RUNTIME.currentIndex];
    _HOOKS_RUNTIME.currentIndex++;
    return slot;
  }
  // You will see implementation of setStateFunction in later parts
  const setStateFunction = (newState) => { ... };
  const newStateSlot = [defaultValue, setStateFunction];
  _HOOKS_RUNTIME.stateArray[_HOOKS_RUNTIME.currentIndex] = newStateSlot;
  _HOOKS_RUNTIME.currentIndex++;
  return newStateSlot;

}
```

What should `setState` do?

1. Compare the new value against the old one. If it’s the same, do nothing.
2. Update the state in the correct slot.
3. Tells React that an rerender is necessary.

Why does useState depend on call order?

Why can’t we do:

```
useState(‘defaultValue’, ‘key for the state’);
```

Then it’s not compositional!

Why does useState depend on call order?

Suppose we have two custom hooks, which contains

```
useState('defaultValue 1', 'key');
```

```
useState('defaultValue 2', 'key');
```

respectively.

They are both fine in isolation, but when you combine them, there is a collision.

Then, reasoning about correctness of combining hooks requires you to read the code for both.

https://overreacted.io/why-do-hooks-rely-on-call-order/

Now you should know all the answers!

- How does React sync your data with real DOM?
- How does useState know which component it’s in?
- Why props should be immutable?
- Why can’t I just push to an array directly?

Part 4: The Implementation

We don’t have to write our own JSX parser!

JSX is an open standard.

A lot of tools already implemented JSX transpilation for us.

We can reuse the entire create-react-app toolchain, only replacing the react core implementation.

We need to understand our tools first to understand the sentences above.

Understand the toolchain - Babel

- When it sees
  - It generates `React.createElement('div', { foo: 'bar' })`
- When it sees <MyComponent foo=”bar” />
  - It generates `React.createElement(MyComponent, { foo: 'bar' })`
- It doesn’t even care whether you properly imported React!
  - Therefore, you can easily configure Babel to write JSX in Vue

Understand the toolchain - Webpack

It inspects your imports.

If it imports css, it will add <link href=”blabla.css” /> to the head.

It it imports JS/TS, it will call babel to transpile the code and add <script src=”blabla.js” /> to the body.

It can perform minimization if you want.

That’s why we can still import css!

Understand the toolchain - TypeScript

JSX is part of the TypeScript language.

At the language level, it doesn’t contain any React specific stuff.

You can configure it to recognize React specific stuff.

Repo Organization

Repository is setup using Yarn workspace.

If you don’t know what is Yarn workspace, check the last DevSesh.

Let’s jump into the code!

https://github.com/SamChou19815/mini-react

Part 5: Final Thoughts

What’s Missing

Key

Refs

Class components

More hooks

Full DOM support

Optimization

DevTools support

Error boundary

Context

Concurrent Mode

Fast Refresh

Fiber Reconciler

dangerouslySetInnerHTML

Alternative renderers

... (This list is incomplete)

You should always strive to learn more

Build react from scratch: https://www.youtube.com/watch?v=_MAD4Oly9yg

Dan Abramov's Blog: http://overreacted.io/

React Conf: https://conf.reactjs.org/

Why this DevSesh

Of course, some fancy React internals. A lot of new frameworks (Flutter, SwiftUI) are inspired by React, that’s also a plus.

But more importantly, thinking process of developing a framework.

You learn how to develop good abstractions, which is helpful for your subteam work.

Feedback Link

https://forms.gle/jtgLJn16CGegav5o8