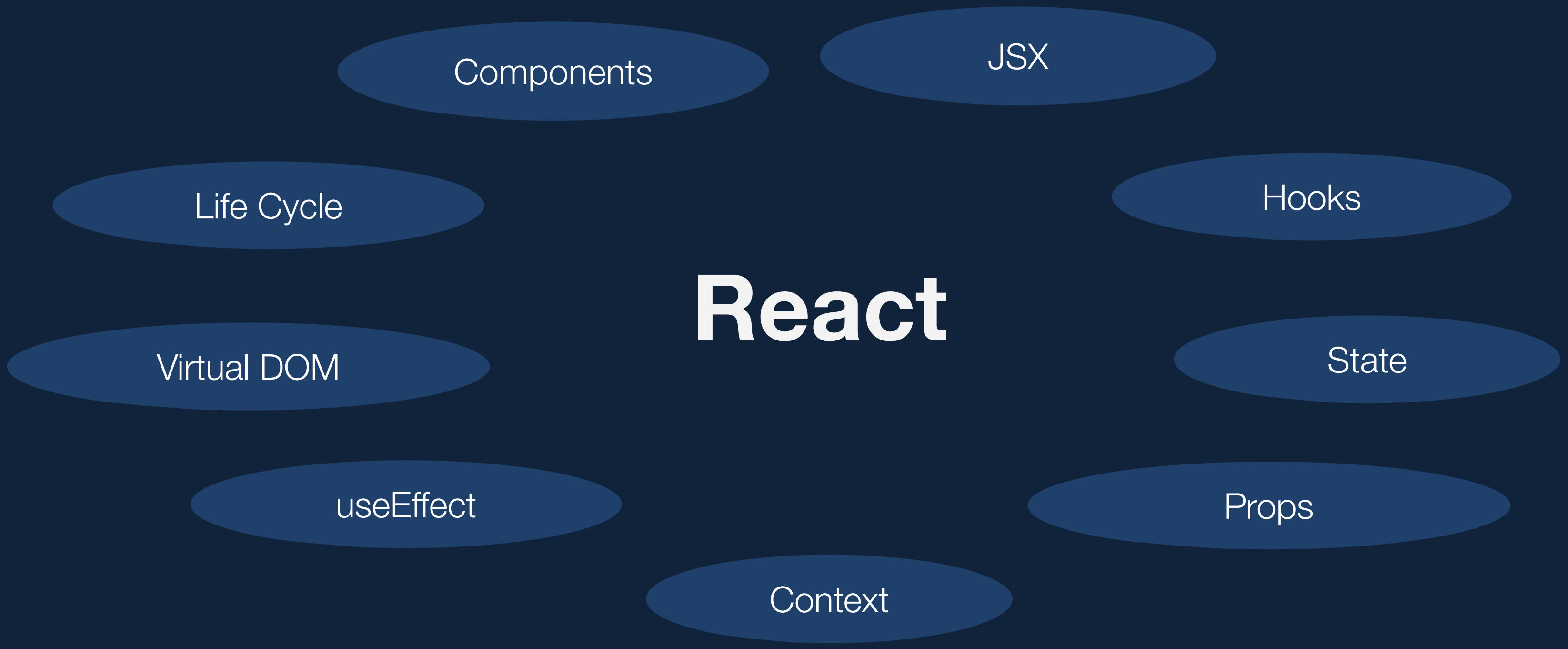
Workshop

React & Typescript

Hint for trainers

- Report each change or addition to the **trainers’** Discord-Channel.
- Tell which Slide is affected, why the change is important and what benefit your change provides.
- Use the code-highlighting-app if you work with code-snippets.
- Use the following slide if you want to repeat certain topics of the workshop.

Task: Test your knowledge



React

Components

JSX

Life Cycle

Hooks

Virtual DOM

State

useEffect

Props

Context

Introduction

A JavaScript library for building user interfaces.

- React Website

Used by

Facebook, Instagram, Whatsapp, Yahoo, AirBnB, Khan Academy, Netflix, ...

Motivation

Why do we need React?

Modern web applications often have complex UIs based on constantly changing state.

React helps us to keep our **UI in sync with** our **state** (/data).

Challenges of web applications

user inputs

live data updates

updates from server

button events

local preferences & offline support

Photo by Kent Pilcher https://unsplash.com/photos/jW8hkB_Qmj8

Challenges of web applications

It is our job…

- to keep data and UI in sync
- to respond to events and update our data and UI

React supports us with building complex UIs.

Photo by Kent Pilcher https://unsplash.com/photos/jW8hkB_Qmj8

Core ideas of React

- React is declarative.
- React is component-based.
- Our UI becomes a function of our state – same input, same output.

Imperative nature of Vanilla JavaScript

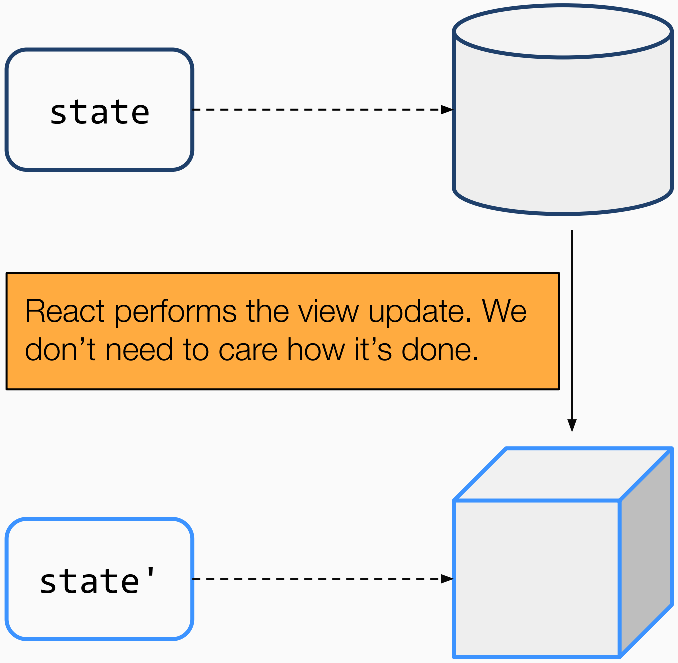
<code>

Vanilla JS: Imperatively updating DOM elements

```javascript
// Reference to DOM node
const wrapper = document.getElementById('wrapper');

// Dynamically adjust style within JavaScript
wrapper.style.backgroundColor = "blue";

// Reference to DOM node
const button = document.getElementById('button');

// Interactivity via event listeners
button.addEventListener('click', () => {
    wrapper.style.backgroundColor = 'red';
})
```

React is declarative

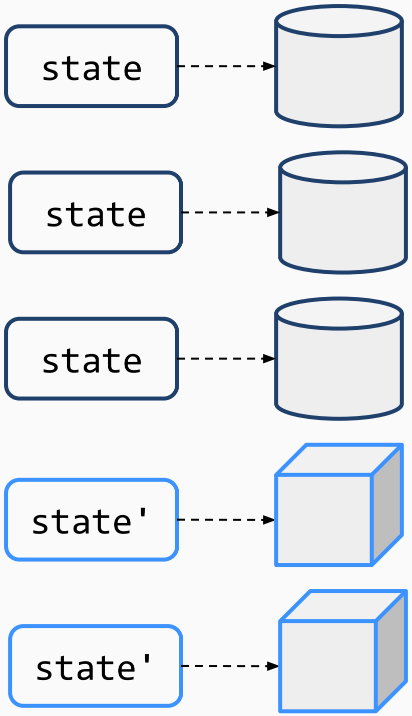
We declare simple views based on our state (= data). React updates them for us when the data changes.


state
React performs the view update. We don’t need to care how it’s done.
state'

Our UI becomes a function of our state

For the same state, a component returns the same output.

When the state changes, it returns a new output based on the new state.



React is component-based

We build encapsulated components managing their own state. We compose multiple components together to complex layouts.

```
<Instruments />
    <FlightData />
    <MotorData />
        <Instrument type="gauge" />
        <Knob />
```

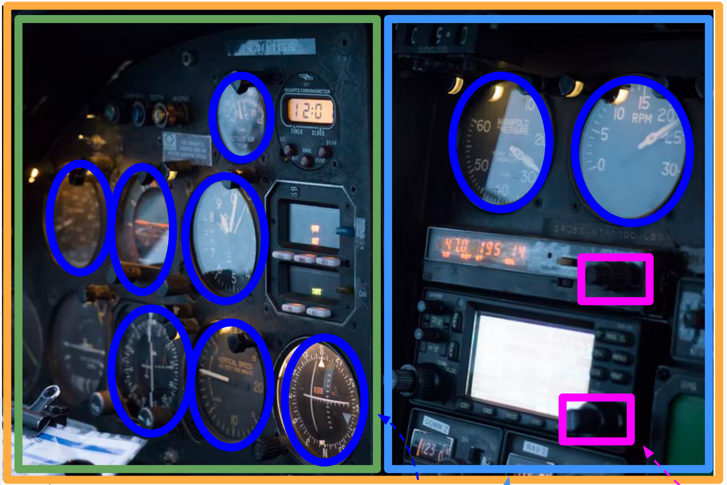


Photo by Kent Pilcher https://unsplash.com/photos/jW8hkB_Qmj8

Project setup with Create React App

Starter kit for React Apps with no build configuration. (> 100.000 ★ on GitHub)

Why / What you'll learn

- Modern Web Apps use more and more complex tooling
  - ES6 Transpiler
  - Bundler
  - CSS Preprocessor

Why / What you’ll learn

- Create React App let’s you start quickly
    - Support for React, JSX, ES2017 and TypeScript
    - A dev server with linter
    - Import CSS and image files directly from TypeScript
    - Autoprefixed CSS
    - A build script **for production** (including sourcemaps)
- No vendor lock-in *(npm run eject)*
- Officially supported by the React Team

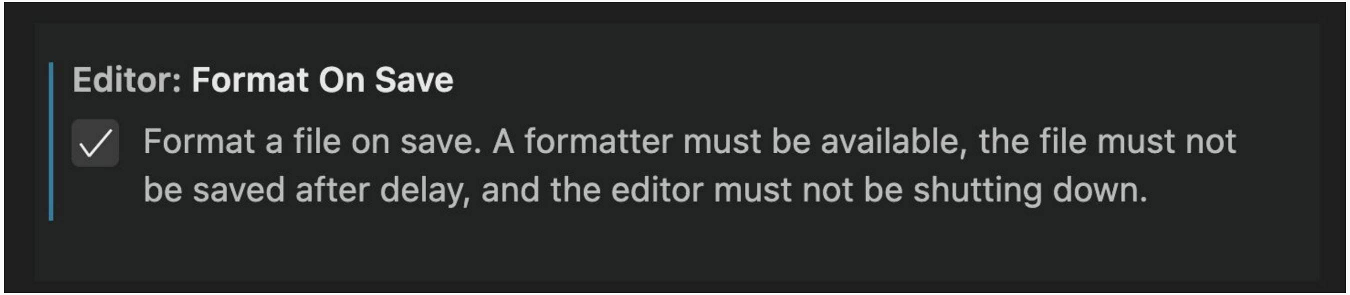
Important commands

```
npx create-react-app react-workshop --typescript
```

- npm start
- npm test
- npm run build
- npm run eject
- npm install react-scripts@latest *// Update your build ENV*

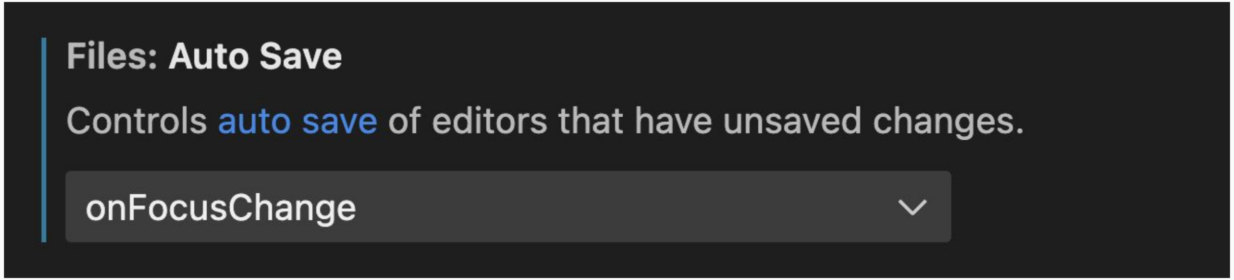
Recommended
VS Code Extensions & Settings

Important VS Code Settings



Editor: Format On Save

☑ Format a file on save. A formatter must be available, the file must not be saved after delay, and the editor must not be shutting down.



Files: Auto Save

Controls auto save of editors that have unsaved changes.

onFocusChange

Recommended VS Code Extensions

- Prettier
- ESLint
- Error Lens
- Auto rename tag

Task

Create your demo app

Part of your course preparation

React Elements

React elements are the building blocks of a React UI and describe what we see on the screen.

Normally, we don’t use them on their own.

Signature of the createElement function

Use the createElement function to create an element

```
import React from 'react';

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

Create an element with React

Example use of the createElement function

```
React.createElement('div',  {id: 'tooltip'},  'Hello World')

// <div id="tooltip">Hello World</div>
```

Create an element with React

<code>

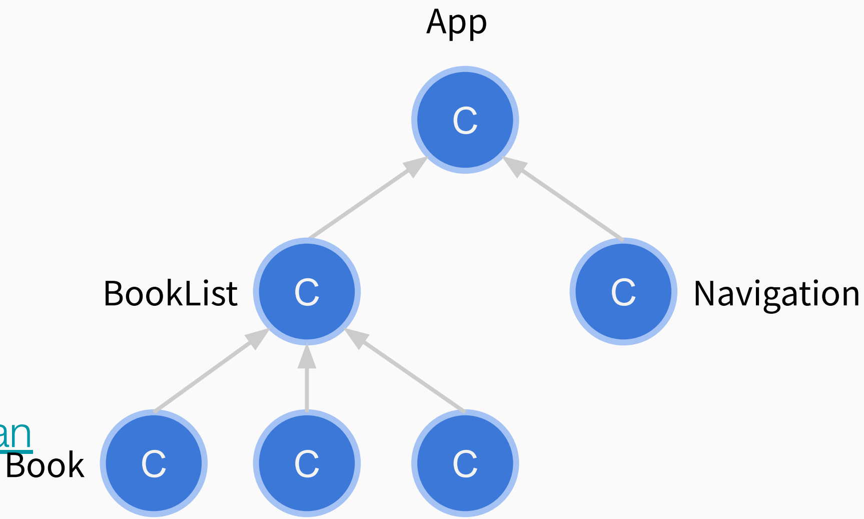
Use object for DOM properties

```
React.createElement('div',  {style: {backgroundColor: 'red'}},  'Hi')

// <div style="background-color: red">Hi</div>
```

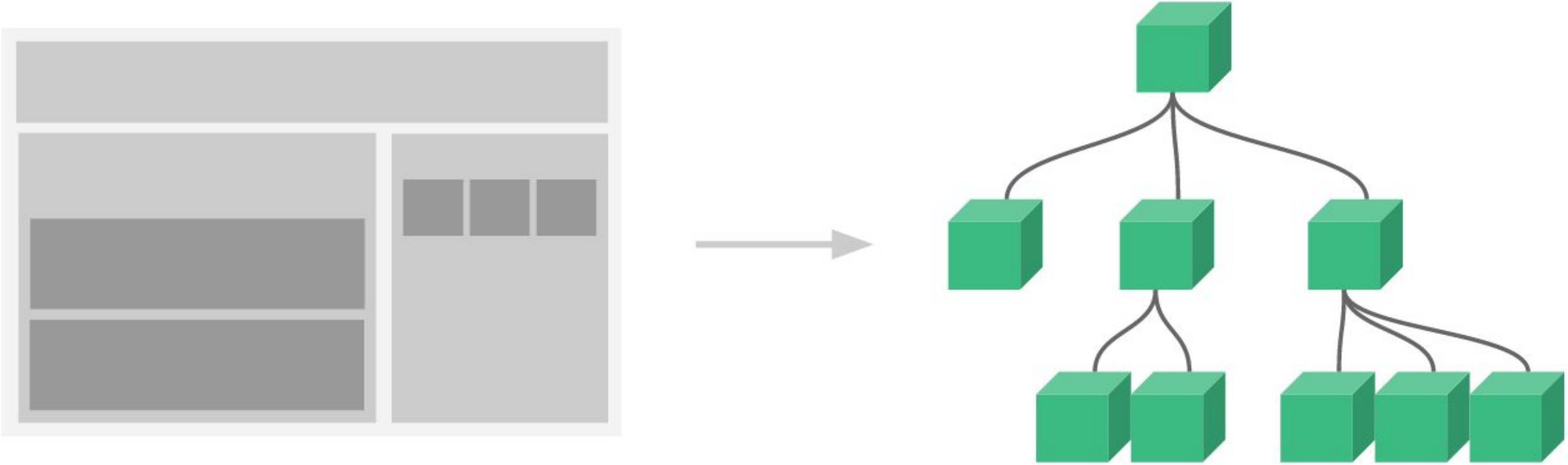
React Components

Components

- Break your application into **small smart & reusable parts**
- They render **React Elements**
- Each in one file
  - Not a must. See this comment of Dan Abramov



Components



A React function component is a function returning a React element.

Create a Simple component

The easiest way to create a React component

```
import React from 'react';

export const HelloComponent = () => {
  return <div>Hello World</div>;
};
```

Create a Simple component

<code>

Under the hood the html-like syntax gets converted to plain old javascript function calls

```
const HelloComponent = () => {
  return <div>Hello World</div>;
};
```

```
const HelloComponent = () => {
  return React.createElement('div',  null,  'Hello World');
};
```

We'll look at this in more detail later on…

Use a component

<code>

Just import your component and use it in other components as if it was an HTML-Tag

```
import { HelloComponent } from './components/HelloComponent';

const App = () => {
    return (
      <div className="App">
        <HelloComponent />
      </div>
    );
}
```

Use a component

<code>

Just import your component and use it in other components as if it was an HTML-Tag

```
import { HelloComponent } from './components/HelloComponent';

const App = () => {
    return (
      <div className="App">
        <HelloComponent />
      </div>
    );
}
```

React Elements can be either plain old HTML tags or React Components

Use a component

<code>

HTML-Attributes are actual DOM properties.

```
import { HelloComponent } from './components/HelloComponent';

const App = () => {
    return (
      <div className="App">
        <HelloComponent />
      </div>
    );
}
```

~~class~~
className

Bootstrapping

Bootstrapping is the process which places your React application in the *real* DOM.

Every React application needs a starting point.

Bootstrapping React

We create an explicit client-side root, which our application can be mounted in.

```
// index.html
<div id="root"></div>

// index.tsx
import ReactDOM from "react-dom/client"

const container = document.getElementById("root") as HTMLElement;
const root = ReactDOM.createRoot(container);
root.render(<App />);
```

Bootstrapping React

<code>

We create an explicit client-side root, which our application can be mounted in.

```
// index.html
<div id="root"></div>

// index.tsx
import ReactDOM from "react-dom/client"

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(<App />);
```

`as` : TypeScript type assertion.

`document.getElementById` can return `null` if no element is found. We’re sure we have the element in the DOM, so we cast the variable to a non-null type which is required by `ReactDOM.createRoot`.

Using `!` would be another option.

Bootstrapping React

<code>

We create an explicit client-side root, which our application can be mounted in.

```
// index.html
<div id="root"></div>

// index.tsx
import ReactDOM from "react-dom/client"

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(<App />);
```

This is our root React Element

React.StrictMode

<code>

Wrap your root element in this built-in React Component, to turn on various safety checks.

```
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Code Conventions:
Component Folder Structure

Convention: Own Folder per component

- Components used throughout the app live in src/components
- Each component in its own folder
- You can group them with isolated styles

```
components/
   AppHeader/
      |- index.ts
      |- AppHeader.tsx
      |- AppHeader.test.tsx
      |- AppHeader.css
```

Convention: Own Folder per component

<code>

```
// components/AppHeader/AppHeader.tsx
import "./AppHeader.css"

export const AppHeader = () => {
  return <div>Hello World</div>;
};

// components/AppHeader/index.ts
export * from "./AppHeader";
```

The index.ts file should re-export all the public definitions in its folder

Using a component

<code>

Typescript automatically looks for an ***index.ts*** when importing a folder

```
import { AppHeader } from "./components/AppHeader";
const App () => {
  return <AppHeader />;
};
```

Task

Add an AppHeader component

JSX / TSX

TSX is a preprocessor step that adds XML syntax to TypeScript

Why / What you'll learn

- Using a lot of (nested) `React.createElement` gets confusing
- HTML is familiar for most developers

Nested `React.createElement`

At first TypeScript did not support JSX 🙈

```
React.createElement("section", {className: "image-gallery"},
  React.createElement("div", {
    onMouseLeave: handleMouseLeave,
    /* other props */ },
    React.createElement("a", {className: "image-gallery-left-nav",
      onClick: slideToIndex(currentIndex - 1)}),
    React.createElement("a", {className: "image-gallery-right-nav",
      onClick: slideToIndex(currentIndex + 1)}),
    React.createElement(Swipeable, {
      onSwipedLeft: slideToIndex(currentIndex + 1),
      onSwipedRight: slideToIndex(currentIndex - 1)},
        React.createElement("div", {className: "image-gallery-slides"},
          slides
        )
    ),
      props.showBullets &&
        React.createElement("div", {className: "image-gallery-bullets"},
          React.createElement("ul", {className: "image-gallery-bullets-container"},
            [
              // further React elements (created with `createElement`)
            ]
          )
        )
  ),
)
```

J(T)SX is syntactic sugar for JS functions

With TSX you write your components in an HTML-like style

```
// JavaScript
React.createElement(Message, { content: 'Hello world!' });

<!-- JSX -->
<Message content="Hello world!" />
```

JSX under the hood

- Technically JSX is transformed into `jsx` calls (since React v17.0), but the signature is very similar to `React.createElement`

```
// JSX
const App = () => {
  return <div>Hello World</div>;
}

// Inserted by a compiler (don't import it yourself!)
import {jsx as _jsx} from 'react/jsx-runtime';

function App() {
  return _jsx('div', { children: 'Hello world' });
}
```

Own Components Must Be Capitalized

- Built-in components start with a lowercase letter

```
<div>, <span>, ...
```

- User-Defined components must be capitalized

```
<Name>,<ToolTip>, ...
```

Embedding Expressions in TSX

You can embed any JavaScript expression in JSX by wrapping it in curly braces.

Examples:

- `{2 + 2}`
- ``{`Template Strings are ${contextVariable}`}``
- `{user.firstName}`
- `{formatName(user)}`

JavaScript Expressions in TSX

<code>

Use any JavaScript/TypeScript expressions with { curly braces }

```tsx
export const MaxComponent = () => {
  const name: string = "Max";

  return <span>Hello {name}</span>;
};
```

TSX
Gotchas

TSX Gotchas

You always need a root element.

```
// Wrong!

() => (
  <span>Hello</span>
  <span>React</span>
);
```

```
// Right

() => (
  <div>
    <span>Hello</span>
    <span>React</span>
  </div>
);
```

TSX Gotchas

Alternative: Fragment (doesn’t render any html)

```
// Wrong!

() => (
  <span>Hello</span>
  <span>React</span>
);
```

```
// Right
import { Fragment }
from 'react';

() => (
  <Fragment>
    <span>Hello</span>
    <span>React</span>
  </Fragment>
);
```

```
// Right (short syntax)

() => (
  <>
    <span>Hello</span>
    <span>React</span>
  </>
);
```

TSX Gotchas

<code>

Set CSS classes via **className not class**

```
// Wrong!

const HelloComponent = () => (
  <div class="hello">
    Hello World
  </div>
);
```

```
// Right

const HelloComponent = () => (
  <div className="hello">
    Hello World
  </div>
);
```

TSX Gotchas

<code>

Set CSS-Styles as an Object **not** as a String

```
// Wrong!

const HelloComponent = () => (
  <div
    style="background-color: red"
  >
    Hello World
  </div>
);
```

```
// Right

const HelloComponent = () => (
  <div
    style={{backgroundColor: 'red'}}
  >
    Hello World
  </div>
);
```

TSX Gotchas

It’s not possible to use if() in TSX but you can use the ternary operator

```
// Wrong!

let isTrue = true;

const HelloComponent = () => (
  <div>
    { if (isTrue) { /*…*/ } }
  </div>
);
```

```
// Right

let isTrue = true;

const HelloComponent = () => (
  <div>
    { isTrue ? 'yes' : 'no' }
  </div>
);
```

Task

Display an example book

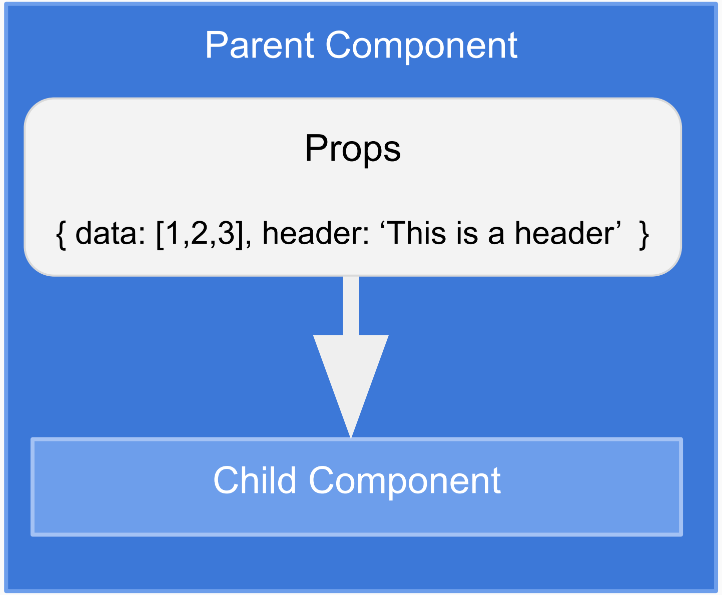
Props

Props allow you to pass data to a component

*Exactly like arguments allow you to pass data to a function

Components with Props

- Pass data into child component(s)
- Configure parameters for sub components
- React passes TSX props to a component as a single object



Props in a component

<code>

Child component receives props as a function argument

```tsx
// Person.tsx (child)
export interface PersonProps {
  name: string;
}

export const Person = (props: PersonProps) => (
  <span>{props.name}</span>
);
```

Always define an interface for your props

Props in a component

<code>

Parent component passes props to child components by setting ‘attributes’ on the TSX element

```
// PersonList.tsx (parent)
const PersonList = () => (
  <ul>
    <li><Person name="Robin"  /></li>
    <li><Person name="Sophie" /></li>
    <li><Person name="Wojtek" /></li>
  </ul>
);
```

Props in a component

In a function component props are a function argument

```
// Person.tsx (child)
export interface PersonProps {
  name: string;
}

export const Person = (props: PersonProps) => (
  <span>{props.name}</span>
);

// PersonList.tsx (parent)
const PersonList = () => (
  <ul>
    <li><Person name="Robin"  /></li>
    <li><Person name="Sophie" /></li>
    <li><Person name="Wojtek" /></li>
  </ul>
);
```

Props in a component

<code>

You normally want to use destructuring for props (ES6 feature)

```
// Without Destructuring
export const Person = (props: PersonProps) => (
  <span>{props.name}</span>
);


// With Destructuring
export const Person = ({name}: PersonProps) => (
  <span>{name}</span>
);
```

Default Props

<code>

You can define optional props with a ? at the interface
and define default values at destructuring

```tsx
// Person.tsx
export interface PersonProps {
  name?: string;
}

export const Person = ({name = 'Wojtek'}: PersonProps) => (
  <span>{name}</span>
);

// PersonList.tsx
... <li><Person /></li> ...
```

Composition over inheritance

Realize **Specialization** with **Composition** instead of **Inheritance**: A more “specific” component (`WelcomeDialog`) renders a more “generic” one (`Dialog`) and configures it with props.

```
const Dialog = ({ title, message }: DialogProps) => (
  <View>
    <Text>{title}</Text>
    <Text>{message}</Text>
  </View>
);

const WelcomeDialog = () => (
  <Dialog title="Welcome" message="Thank you for visiting our spacecraft!" />
);
```

Task

Create a BookListItem component

List Rendering

Rendering a list

<code>

In HTML, (most) elements can have multiple children.

```html
<ul class="shoppingList">
   <li>🍌 bananas</li>
   <li>🍅 tomatoes</li>
   <li>🥑 avocado</li>
</ul>
```

Rendering a list

<code>

This is possible in React as well, a React element can have multiple children.

```
<ul class="shoppingList">
   <ListItem item="🍌 bananas" />
   <ListItem item="🍅 tomatoes" />
   <ListItem item="🥑 avocado" />
</ul>
```

Rendering a list

<code>

In JSX/TSX the children of an element are just an array of elements. So we can generate them programmatically:

```
const shoppingList: string[] = [
  "🍌 bananas",
  "🍅 tomatoes",
  "🥑 avocado"
]

// output/return value of our component
<div>
  {shoppingList.map(
      (item: string) => <ListItem key={item} item={item} />
  )}
</div>
```

This use of .map is extremely common in React

Rendering a list

<code>

In JSX/TSX the children of an element are just an array of elements. So we can generate them programmatically:

```
const shoppingList: string[] = [
  "🍌 bananas",
  "🍅 tomatoes",
  "🥑 avocado"
]

// output/return value of our component
<div>
  {shoppingList.map(
      (item: string) => <ListItem key={item} item={item} />
  )}
</div>
```

React needs a key to identify list elements across rerenders

Task

Display a list of books

Conditional Rendering

Conditional Rendering

- Your components will often need to display different things depending on some conditions.

- In React, you can conditionally render JSX using JavaScript syntax like `if` statements, `&&`, and `? :` operators.

Conditional Rendering

<code>

Using if else

```
if (progress < 100) {
  return <div>Loading... {progress}%</div>;
} else {
  return <StartScreen />;
}
```

Conditional Rendering

<code>

Using the `?` `:` (ternary) operator

```
return (
  <div className="App">
    {progress < 100 ? <div>Loading...</div> : <StartScreen />}
  </div>
);
```

Conditional Rendering

<code>

Using && (no alternative / else branch)

```
return (
  <div className="App">
    {progress < 100 && <div>Loading...</div>}
    {progress === 100 && <StartScreen />}
  </div>
);
```

React doesn’t render falsy expressions like `false, null` or `undefined`

Conditional Rendering

<code>

Reminder: you can also use conditionals to set attributes / props

```
return <div className={darkMode ? "dark" : "light"}></div>
```

Task

Display a 💰 next to free books

Events

Handling events with React

There are some *differences* to handling events on DOM elements:

- React events are named using **camelCase**, rather than lowercase.
- With TSX you **pass a function** as the event handler, rather than a string.

Set up an EventHandler

<code>

`onClick` EventHandler as an anonymous function

```
const SimpleContainer = () => {
  return (
    <button onClick={() => alert('Attention')}>
      Press me!
    </button>
  );
}
```

Set up an EventHandler

Create an `onClick` EventHandler function

```
const SimpleContainer = () => {
  const handleClick = () => {
    alert('Attention');
  }

  return (
    <button onClick={handleClick}>
      Press me!
    </button>
  );
}
```

Dynamic values

How to deal with values which change during the lifetime of a component

```
let count = 0;

export const Counter = () => {
  const increment = () => {
    count = count + 1;
    console.log({ count });
  };

  return (
    <button onClick={increment}>
      {count}
    </button>
  );
};
```

Dynamic values

<code>

`count` value will change, but **component won’t rerender!!**

```
let count = 0;

export const Counter = () => {
  const increment = () => {
    count = count + 1;
    console.log({ count });
  };

  return (
    <button onClick={increment}>
      {count}
    </button>
  );
};
```

Hooks

How to use state and other features in Function Components

*“**Hooks** are functions that let you “hook into” React state and lifecycle features from function components.”*

How to use hooks

<code>

Hooks are functions and start with ‘*use*’.

```
import {
  useState,
  useEffect,
  useContext,
  useRef
} from "react";
```

The useState hook

Fixing our rerender issue

Use state with hooks

Array Destructuring to retrieve current state and setter function

```
const [state, setState] = useState<T>(initialState);

setState(newState);
```

Use state with hooks

When `setCount` is called, component will rerender.

```
const Counter: React.FC = () => {
  const [count, setCount] = useState<number>(0);

  return (
    <div>
      <p>Clicked {count}x!</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

In many cases this type can be inferred correctly by Typescript

Event handler + useState hook

Set up an EventHandler

<code>

`onClick` EventHandler which changes state

```
const SimpleContainer = () => {
  const [mood, setMood] = useState<'super'|'awesome'>('super');

  const handleClick = () => {
    setMood('awesome');
  }

  return (
    <button onClick={handleClick}>
      Today my mood is {mood}!
    </button>
  );
}
```

useState & Props

<code>

The state update function can also be passed to child components

```
const SimpleContainer = () => {
  const [mood, setMood] = useState<'super'|'awesome'>('super');

  return <MoodButton mood={mood} setMood={setMood} />;
};

const MoodButton = ({mood, setMood}) => {
  return (
    <button onClick={() => setMood("awesome")}>
      Today my mood is {mood}!
    </button>
  );
};
```

useState & Props

<code>

Props interface

```
interface MoodButtonProps {
  mood: "super" | "awesome";
  setMood: (mood: "super" | "awesome") => void;
}

const MoodButton = ({ mood,  setMood }: MoodButtonProps) => {
  return (
    <button onClick={() => setMood("awesome")}>
      Today my mood is {mood}!
    </button>
  );
};
```

Task

Add a like counter to the BookListItem component

`children` prop

Components and Props

<code>

Special **children** prop

```
import { ReactNode } from "react";

interface WelcomeTextProps {
  children: ReactNode;
}
const WelcomeText = (props: WelcomeTextProps) => {
  return <h1>Welcome to {props.children}</h1>;
};
```

Components and Props

Special **children** prop

```
const WelcomeText = (props: WelcomeTextProps) => (
  <h1>Welcome to {props.children}</h1>
);

const App = () => {
  return (
    <div className="App">
      <WelcomeText>this new App</WelcomeText>
    </div>
  );
};
```

Task

Display the book’s abstract, but make it hideable

Page (Re-)Rendering

A React component can return completely different element trees each time it’s invoked.

React manages the DOM updates for us.

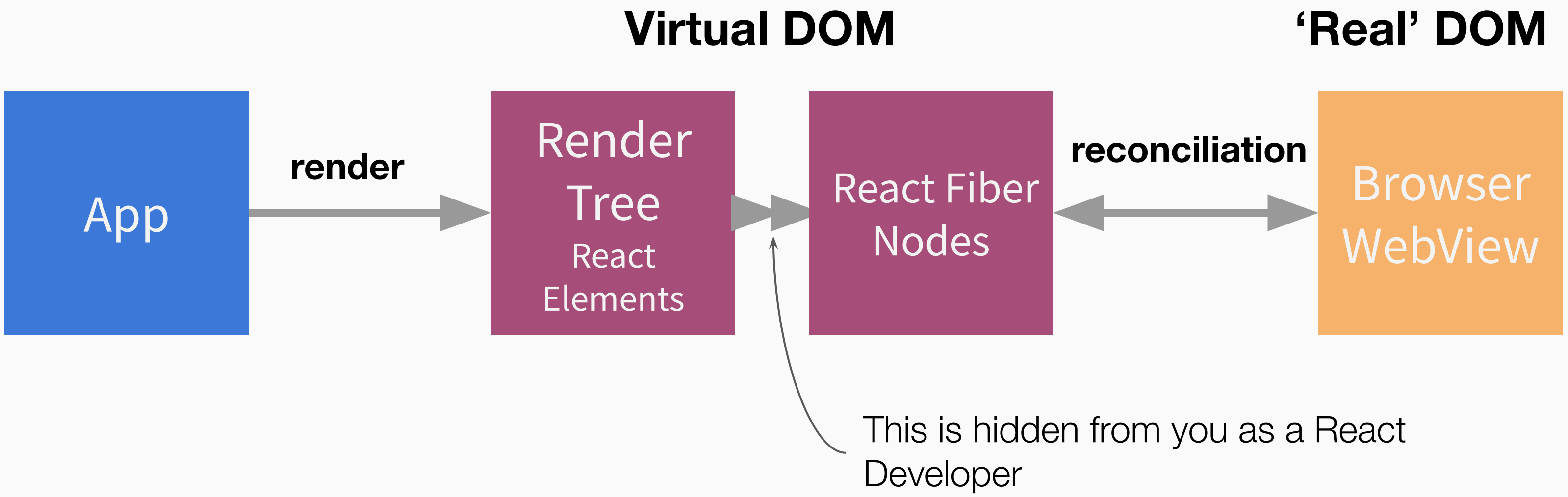
But how?

Little what

The **Virtual DOM** is a data structure made of plain JS objects that **React** uses to represent the state of the browser DOM in JavaScript.

Virtual DOM

Relationship between Virtual and ‘Real’ DOM



Constantly re-render the virtual DOM, but touch the “real” DOM only if necessary.

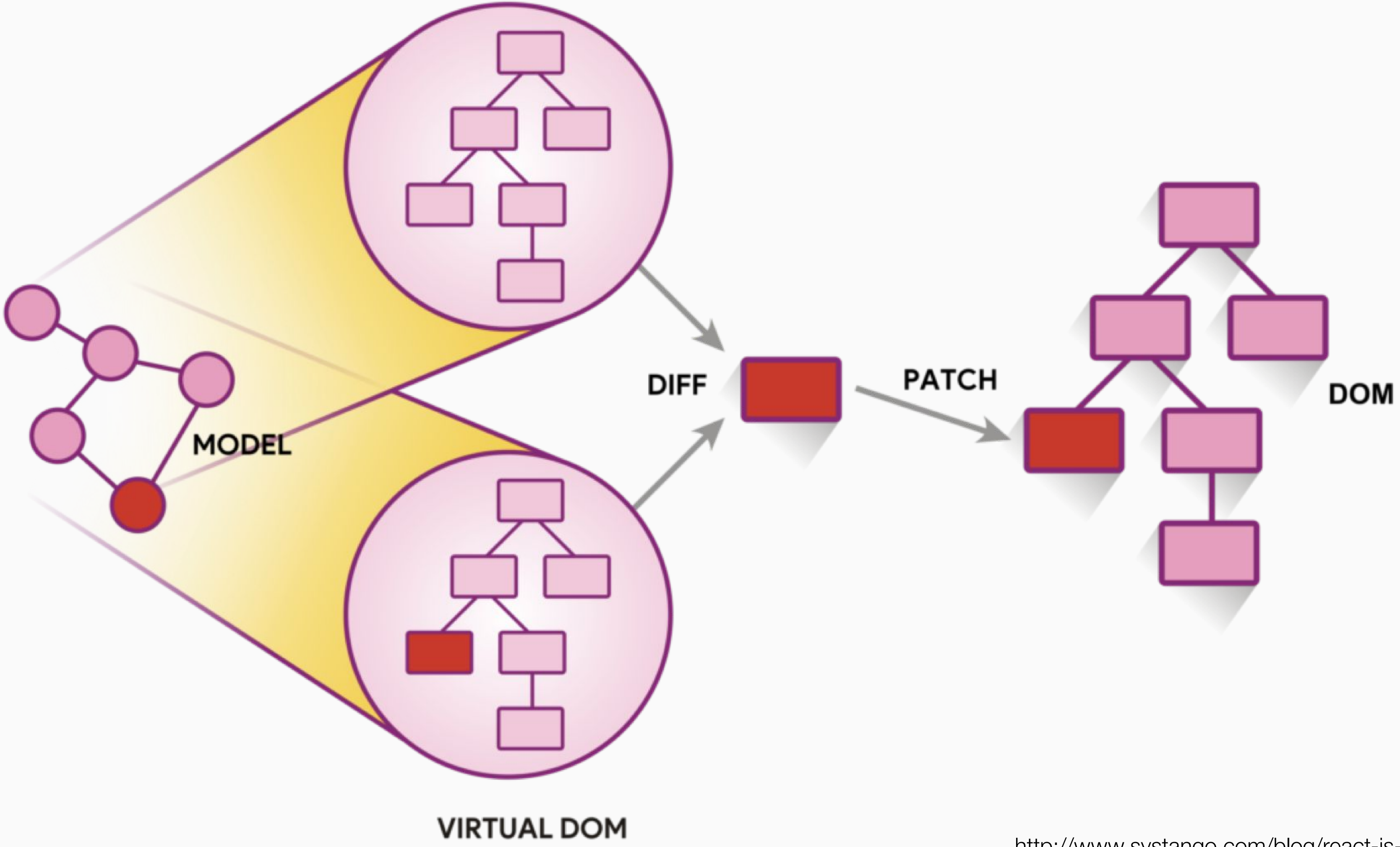
Rendering Cycle

1. React renders components
    a. calls function components / calls render method of class components
    b. re-runs corresponding effects (if not restricted)
2. React compares virtual DOM trees (*reconciliation algorithm*)
3. React updates actual browser DOM (which will trigger a repaint)
4. Browser paints new DOM and CSSOM (UI reflow) **(expensive 💰)**
5. React runs side effects

How React works

Big Idea

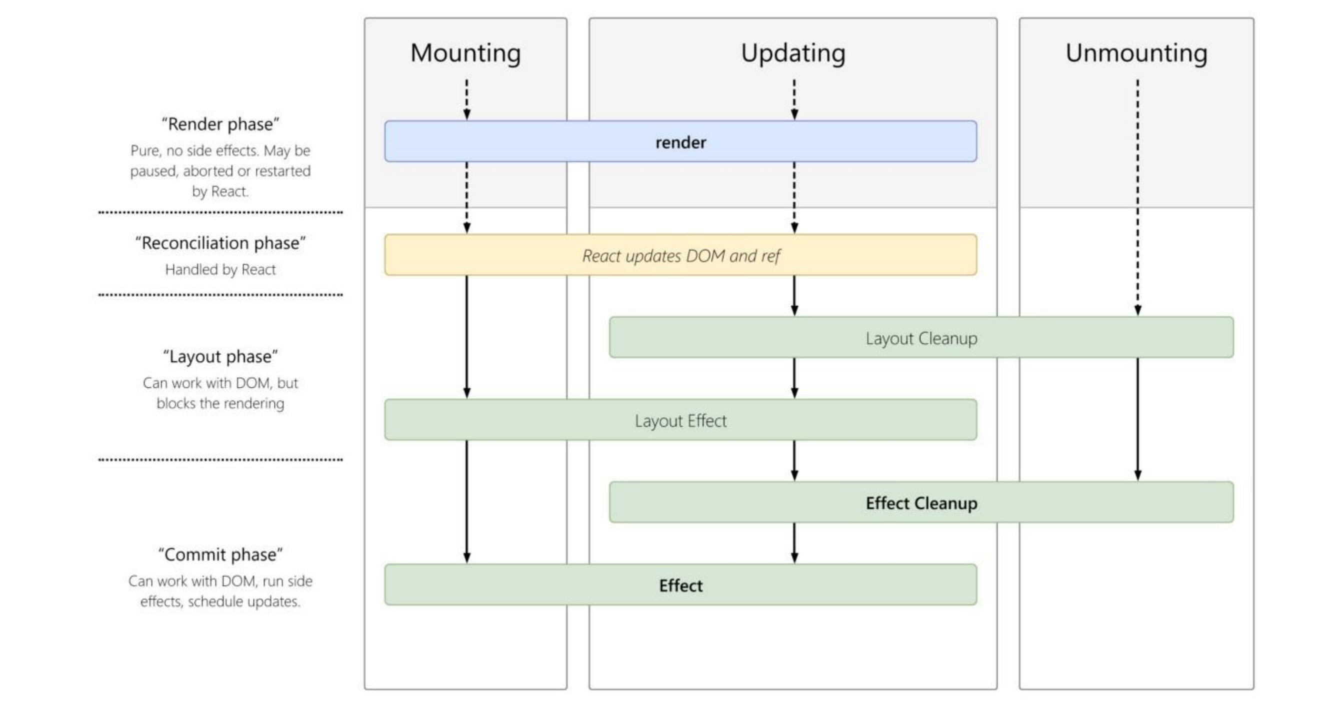
- Always compare Virtual DOM trees.
- Let React calculate the differences between the old and the new one
- Make minimal modifications to the original DOM



http://www.systango.com/blog/react-js-infinity/

Component Life Cycle

React (Hooks) **Life-Cycle**



"Render phase"

Pure, no side effects. May be paused, aborted or restarted by React.

"Reconciliation phase"

Handled by React

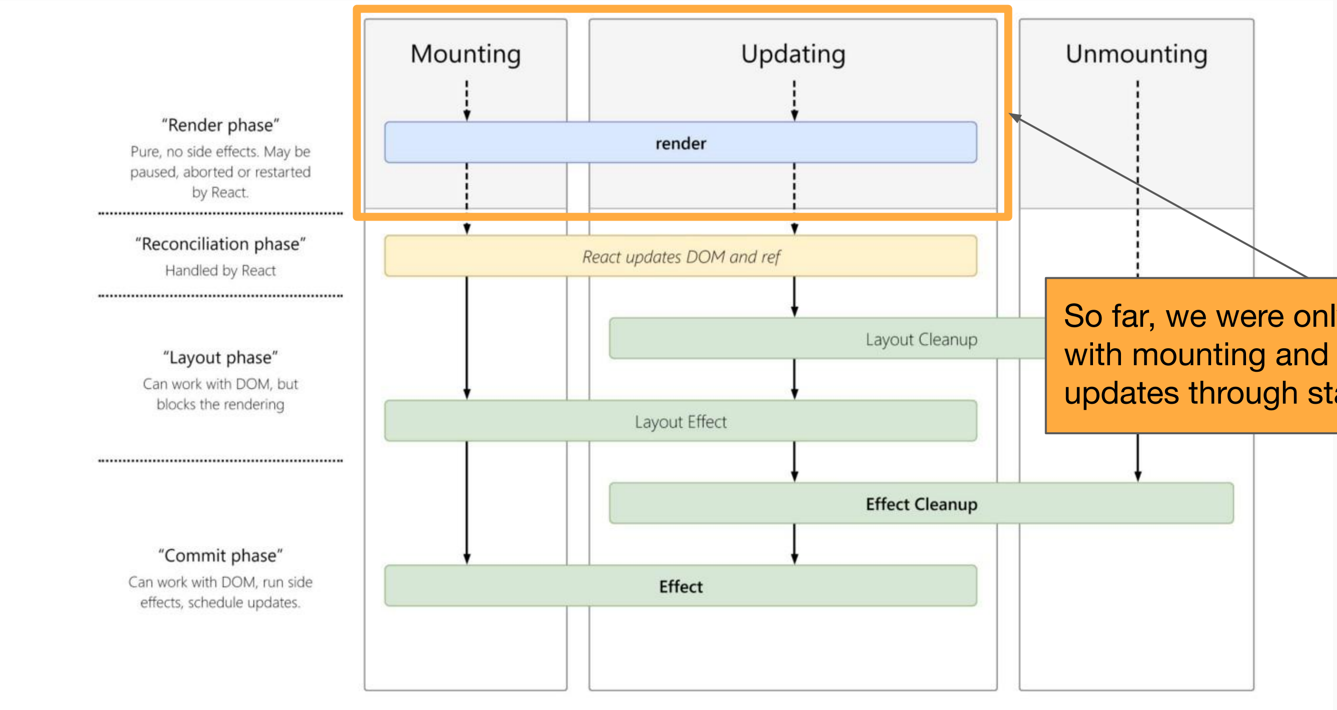
"Layout phase"

Can work with DOM, but blocks the rendering

"Commit phase"

Can work with DOM, run side effects, schedule updates.

React (Hooks) **Life-Cycle**



Mounting

Updating

Unmounting

"Render phase"
Pure, no side effects. May be paused, aborted or restarted by React.

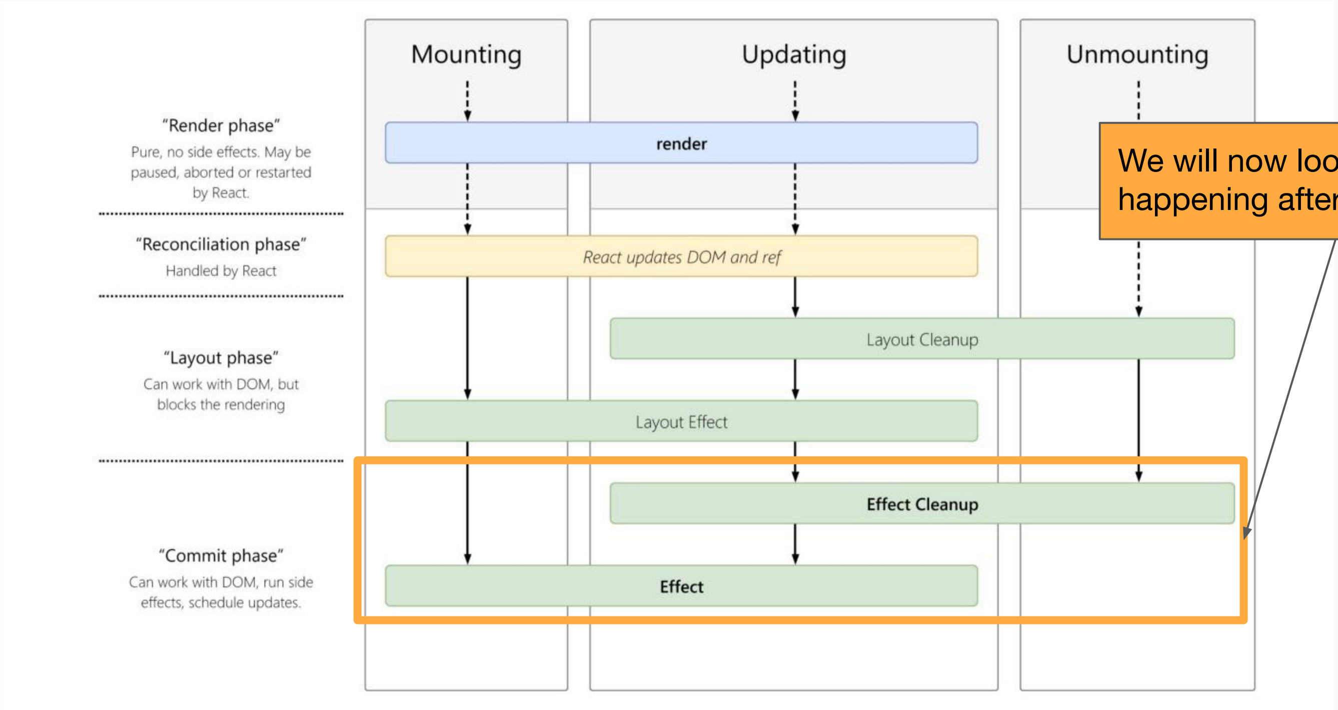
"Reconciliation phase"
Handled by React

"Layout phase"
Can work with DOM, but blocks the rendering

"Commit phase"
Can work with DOM, run side effects, schedule updates.

So far, we were only concerned with mounting and triggering updates through state changes.

React (Hooks) **Life-Cycle**



Mounting

Updating

Unmounting

"Render phase"

Pure, no side effects. May be paused, aborted or restarted by React.

"Reconciliation phase"

Handled by React

"Layout phase"

Can work with DOM, but blocks the rendering

"Commit phase"

Can work with DOM, run side effects, schedule updates.

render

React updates DOM and ref

Layout Cleanup

Layout Effect

Effect Cleanup

Effect

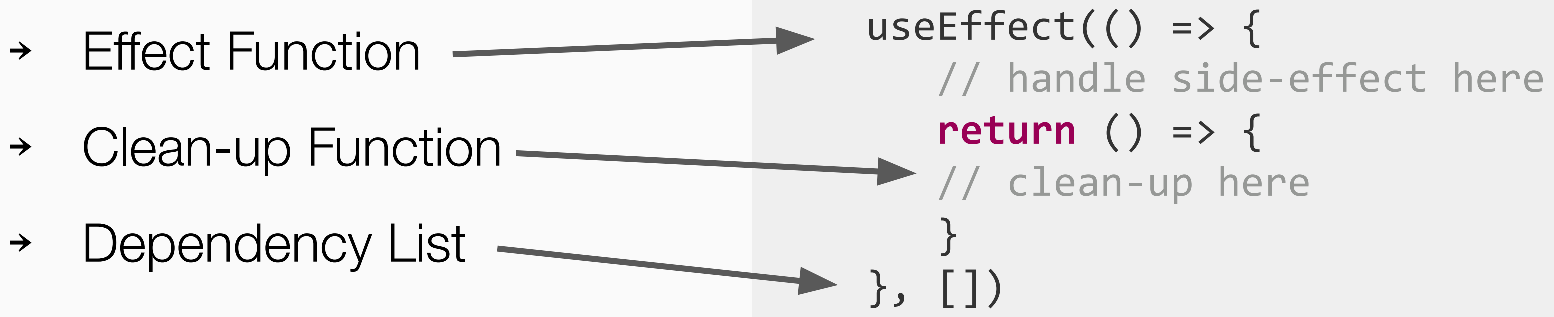
We will now look at effects happening after the rendering.

The useEffect hook

The `useEffect` Hook adds the ability to perform side effects.

It will run after the render is committed to the screen.

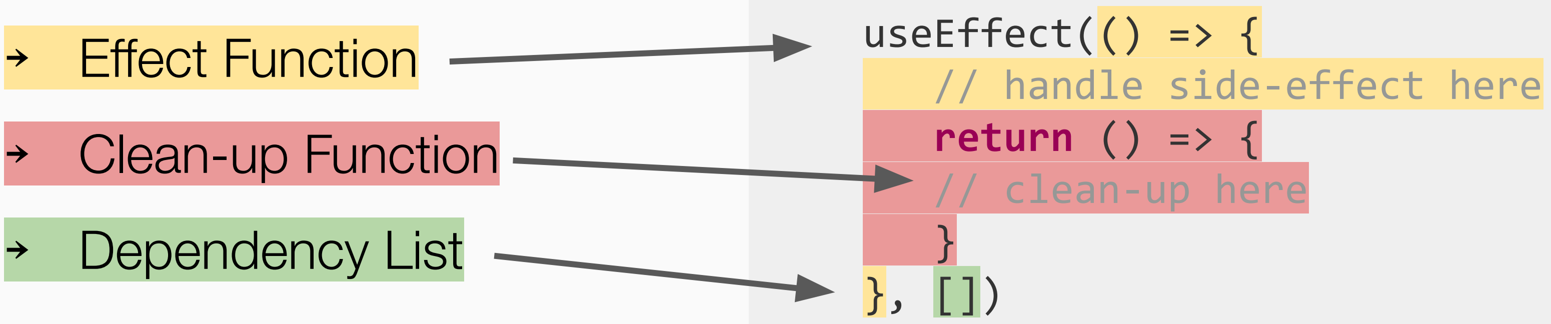
Managing Side Effects



- → Effect Function
- → Clean-up Function
- → Dependency List

```
useEffect(() => {
   // handle side-effect here
   return () => {
   // clean-up here
   }
}, [])
```

Managing Side Effects



- → Effect Function
- → Clean-up Function
- → Dependency List

```
useEffect(() => {
    // handle side-effect here
    return () => {
    // clean-up here
    }
}, [])
```

Effect Function

- Anonymous effect function passed to effect hook.
- Declare effect functions within scope to have access to props and state.

```
const BlogArticle = props => {
   useEffect(() => {
      fetchDataFromHttpApi()
   }, []);

   // ...
 }
```

Clean-up Function

- It’s called **every time before the effect runs (except on mount)** – to clean up from the last run and **when component unmounts**.

```
const BlogArticle = props => {
  useEffect(() => {
    const subscription = props.source.subscribe();

    return () => {
      subscription.unsubscribe();
    };
  }, [props.source]);

  // ...
};
```

Dependency List

- By default effect is run **after every render** is committed to the screen.
- effect runs only when values in dependency array change.

```
const BlogArticle = (props) => {
  useEffect(() => {
    document.title = `Article ${props.id}`;
  }, [props.id]);

  // ...
};
```

Dependency List

- `[]` tells React that your effect doesn't depend on any values from props or state
- effect runs only once, **when the component mounts**.

```
const BlogArticle = (props) => {
  useEffect(() => {
    document.title = "Article 1";
  }, []);

  // ...
};
```

Dependency List

- The dependency List can NOT evaluate changes in complex objects
- Off-the-shelf solution, "use-deep-compare-effect" by Kent C. Dodds

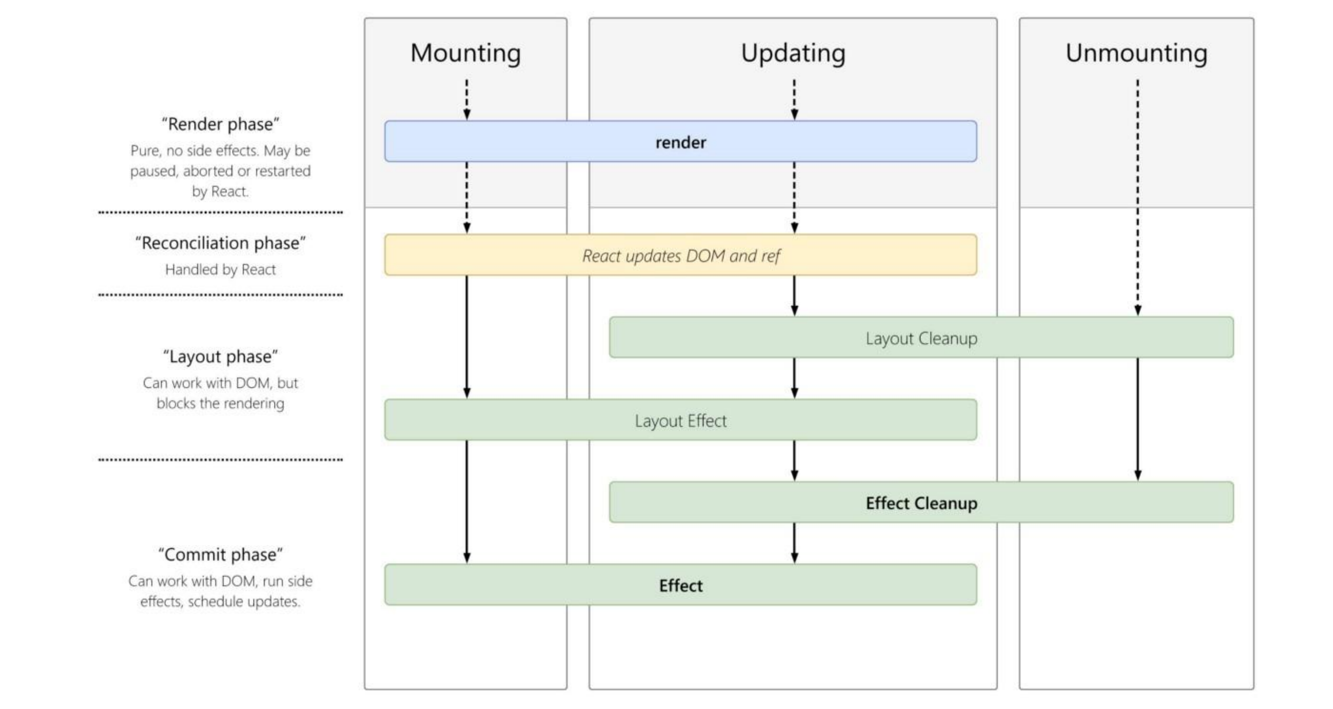
```
function BlogArticle(props) {
  useEffect(() => {
     document.title = `Article ${props.id}`;
  },[props]);
...
}
```

Use Multiple Effects to Separate Concerns

- Hooks let us split the code based on use case rather than lifecycle.
- Hooks are **applied in the order** they were specified

```
const BlogArticle = (props) => {

   useEffect(() => {
      document.title = `Article ${props.id}`;
   },[props.id]);

  useEffect(() => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  }, [props.source]);
 ...
 }
```

React (Hooks) **Life-Cycle**


Mounting
Updating
Unmounting
"Render phase"
Pure, no side effects. May be paused, aborted or restarted by React.
render
"Reconciliation phase"
Handled by React
React updates DOM and ref
"Layout phase"
Can work with DOM, but blocks the rendering
Layout Cleanup
Layout Effect
"Commit phase"
Can work with DOM, run side effects, schedule updates.
Effect Cleanup
Effect

Keep in mind...

- Hook Rules: https://reactjs.org/docs/hooks-rules.html
  - Only call hooks at the top-level and not inside loops or conditions
  - Only call hooks from React functions
- useEffect callback can not be async! Create a local async function you invoke immediately or use .then / .catch to wait for Promises
- Enforce them via an Eslint plugin

Async data fetching

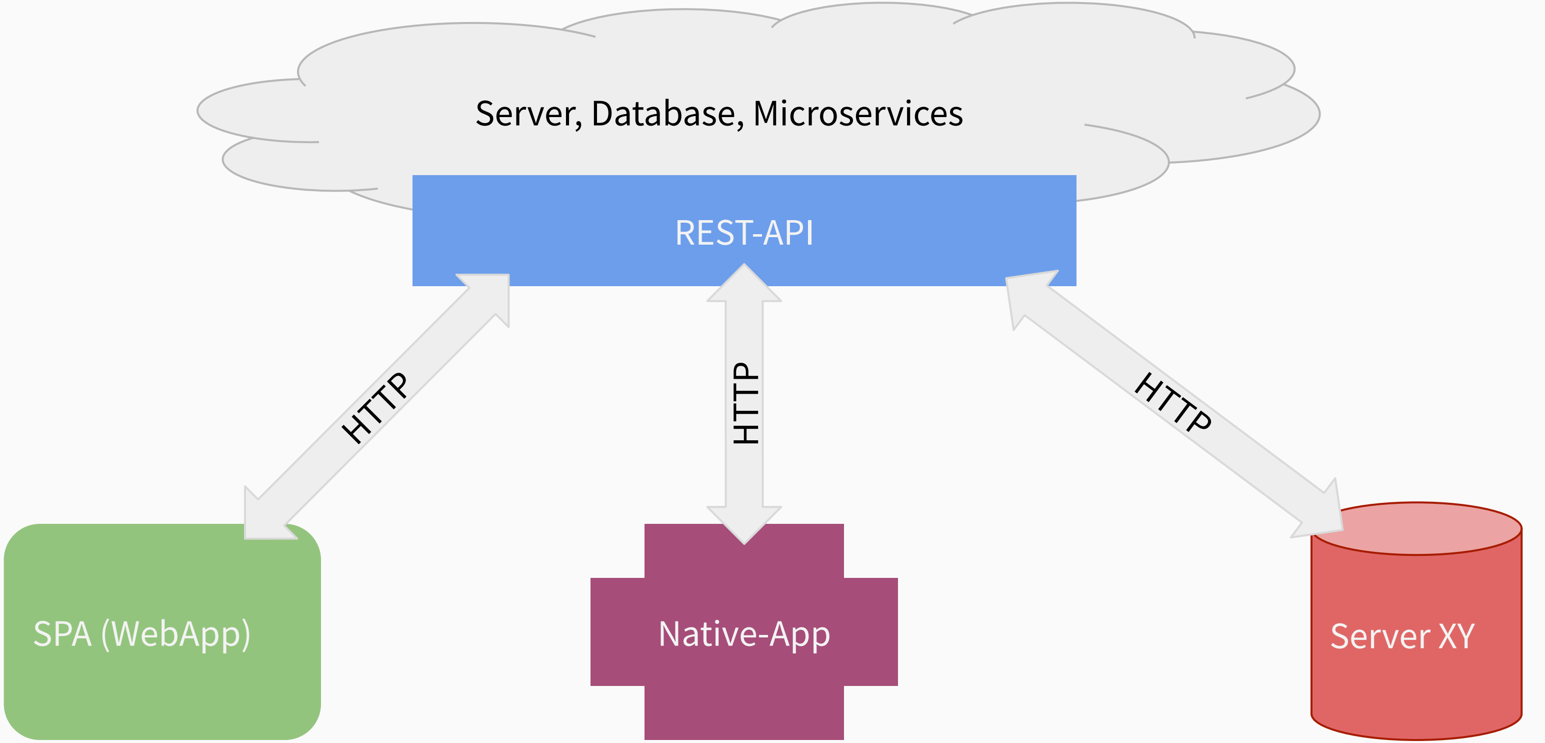
Load data from an API via HTTP

Why / What you’ll learn

- Your data isn't stored locally
  - Multiple clients
  - Huge amount of data

- Communication via HTTP (REST/CRUD)
- Different ways to get data from an API

Using a Rest API



Basic CRUD Service

http://localhost:4730

```
POST   /books        // Create a new book
GET    /books        // Read all books
PUT    /books/:isbn  // Update a book by ISBN
DELETE /books/:isbn  // Delete a book by ISBN
GET    /books/:isbn  // Read a specific book by ISBN
```

Using Fetch

- `fetch` is a modern concept equivalent to `XMLHttpRequest`.
- The fetch API is completely **Promise**-based.
    - JavaScript runtime waits for an asynchronous action to be either **fulfilled** or **rejected** without blocking the UI.

Request an API via Fetch

<code>

One argument as a string results in a **GET** request to this URL

```
return fetch(URL)
  .then(response => response.json())
  .then(result => console.log(result))
```

Using async and wait

Instead of chaining then you also could use async/wait

```
const fetchBooks = async () => {
    const response = await fetch('http://localhost:4730/books')
    const result = await response.json();
    return result;
}
```

Request an API via Fetch

<code>

Request interface allows more detailed control of a resource request

```
const request = new Request(URL, {
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  method : 'PUT',
  body   : JSON.stringify({ /* a JavaScript Object */ })
});

return fetch(request)
  .then(response => response.json())
  .then(result => console.log(result));
```

Task

Fetch and display books from the Bookmonkey API

Custom Hooks

Custom hooks

<code>

On top of the built-in hooks (useState, useEffect, …) you can build new hooks, with a signature that matches your use case.

```
// Restrict state changes to increment and decrement
const useCounter = (initialValue: number) => {
  const [counter, setCounter] = useState(initialValue);

  const increment = () => setCounter(counter + 1);
  const decrement = () => setCounter(counter - 1);

  return { counter, increment, decrement };
};
```

Custom hooks

Using a custom hook looks very similar to using a built-in hook

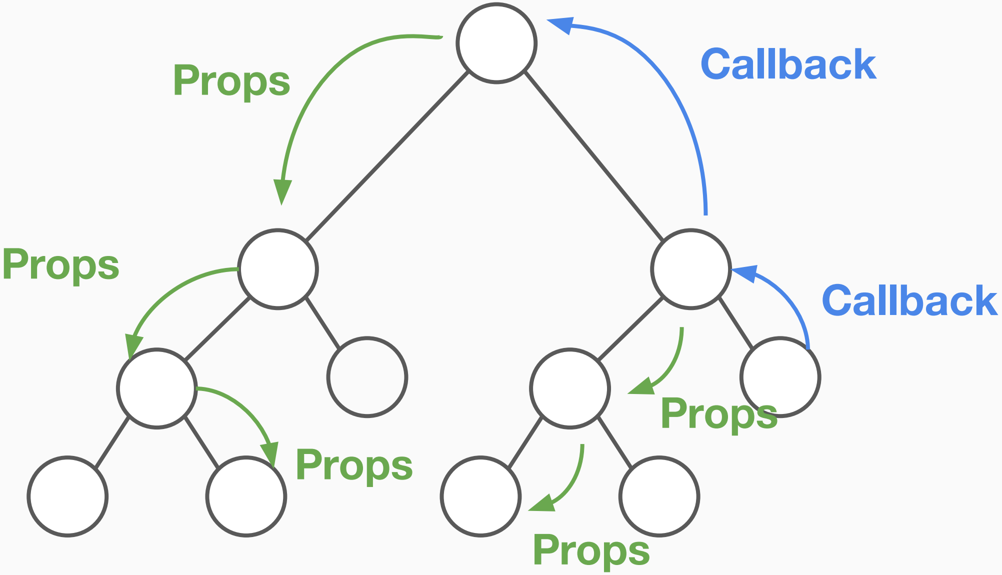
```
const PageSelector = () => {
  const { counter, increment, decrement } = useCounter(1);

  return (
    <div>
      <button onClick={decrement}>-</button>
      <span>{counter}</span>
      <button onClick={increment}>+</button>
    </div>
  );
};
```

Task

Create a custom useBooks hook

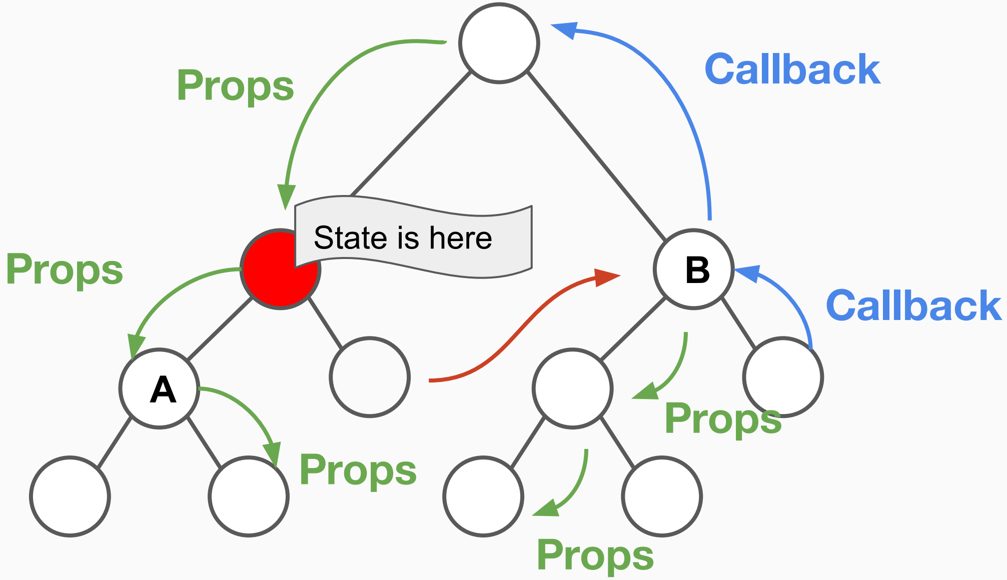
Problem: Props Drilling

How state flows through a React App

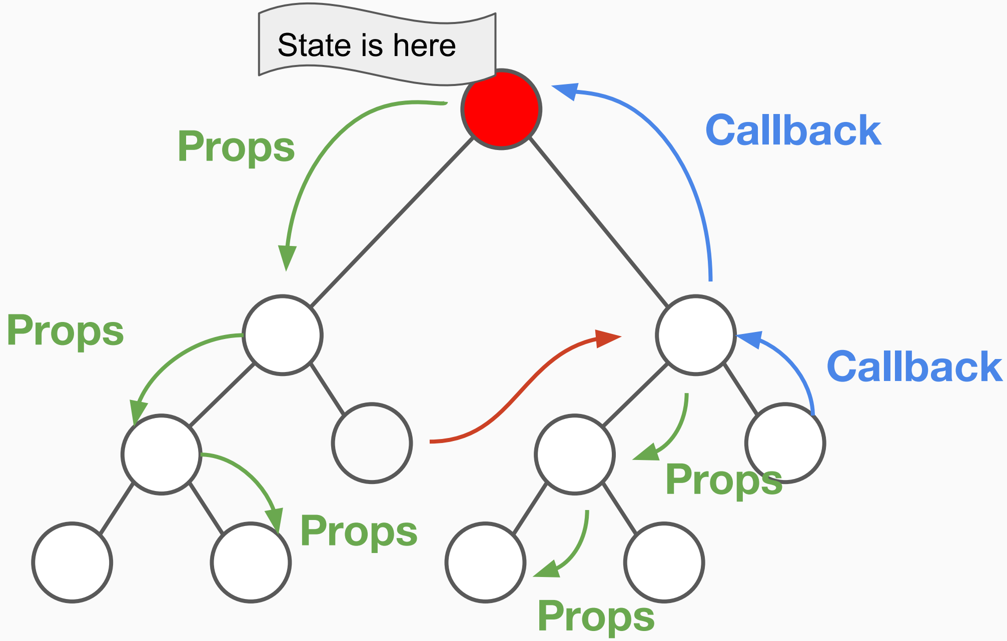


Share state across component sub-trees

Problem: How to trigger *state changes* in component **B** by component **A**?

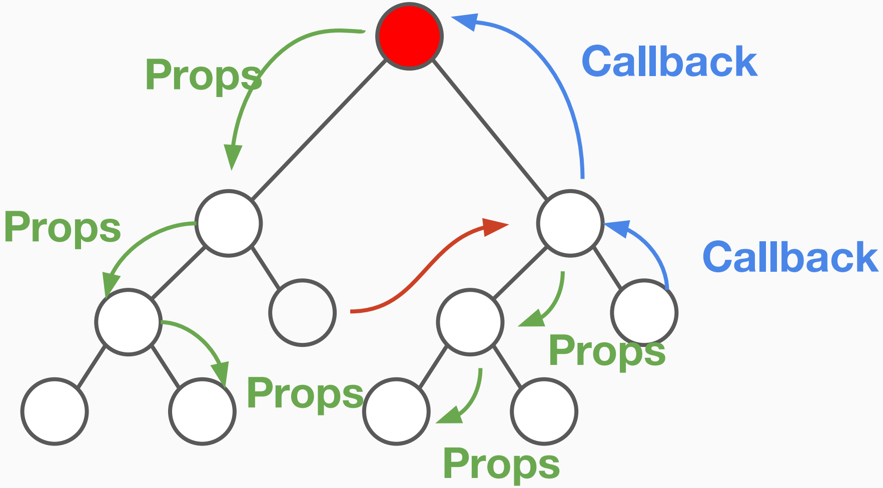


Solution: Lifting state up



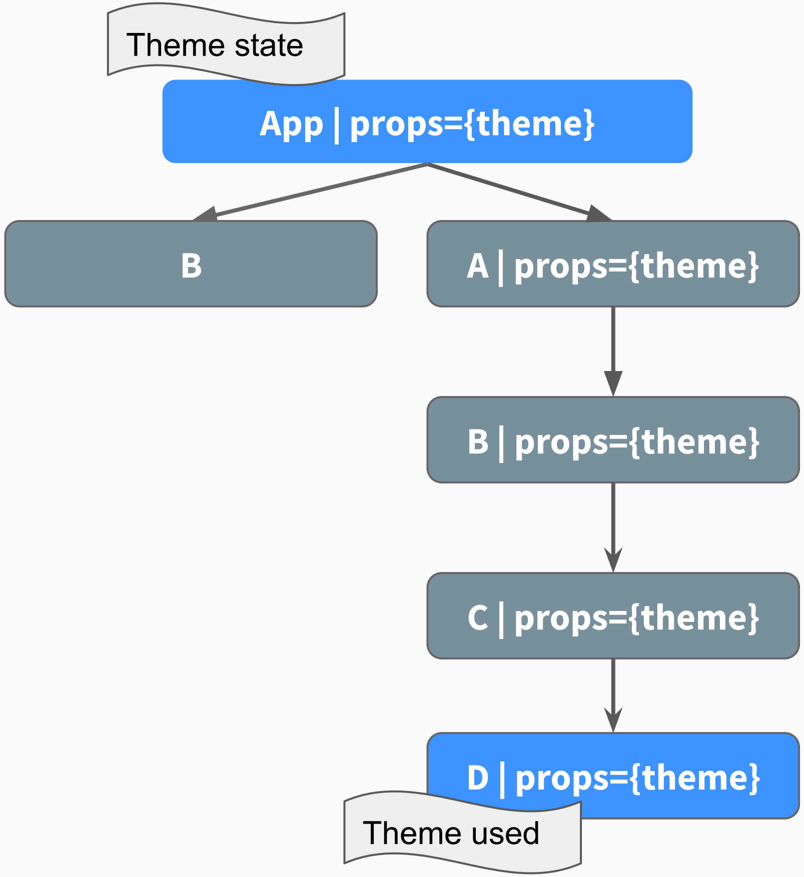
Issues with “Lifting state up”

- Lifting state up leads to big code **refactorings**
- Passing down props over several hierarchy levels creates **tight coupling**
- **Routing** makes it hard to pass props down



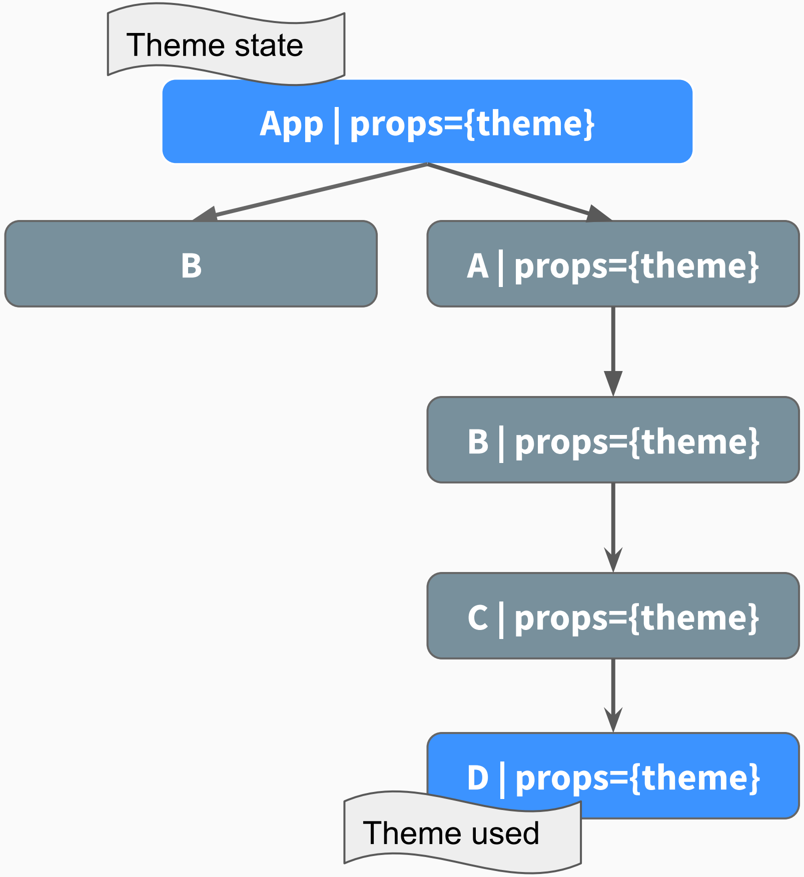
Problem: Props Drilling

- How can we share state across a component subtree?
- Clutters intermediate component with props they don’t use
- Increases coupling between components in a true and reduces reusability



Solution: React Context

Reminder: Props Drilling



Theme state

App | props={theme}

B

A | props={theme}

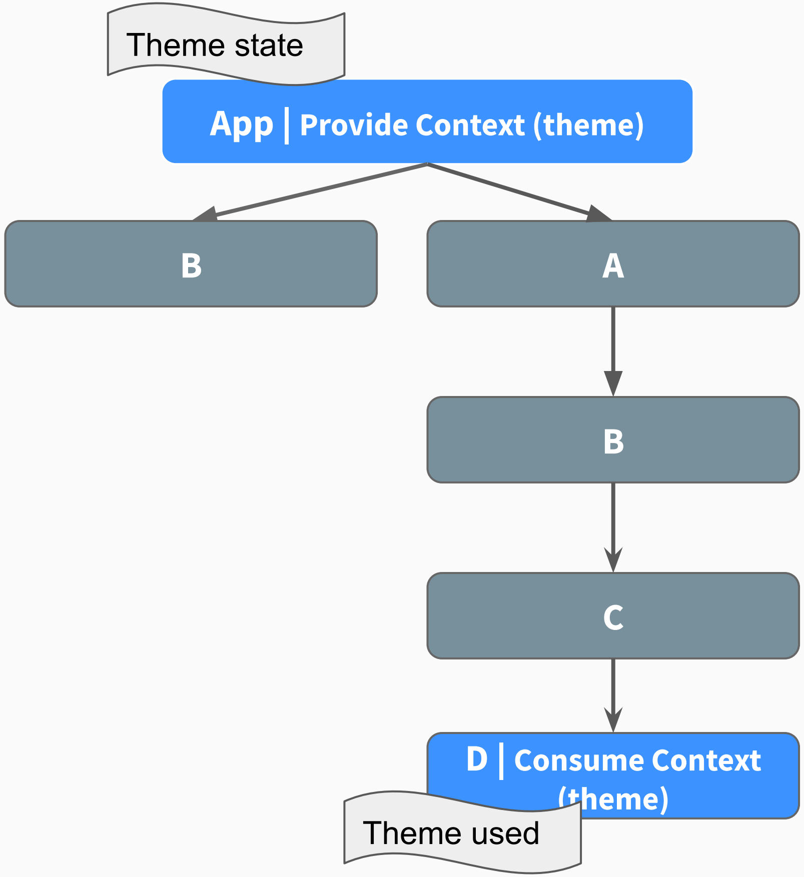
B | props={theme}

C | props={theme}

D | props={theme}

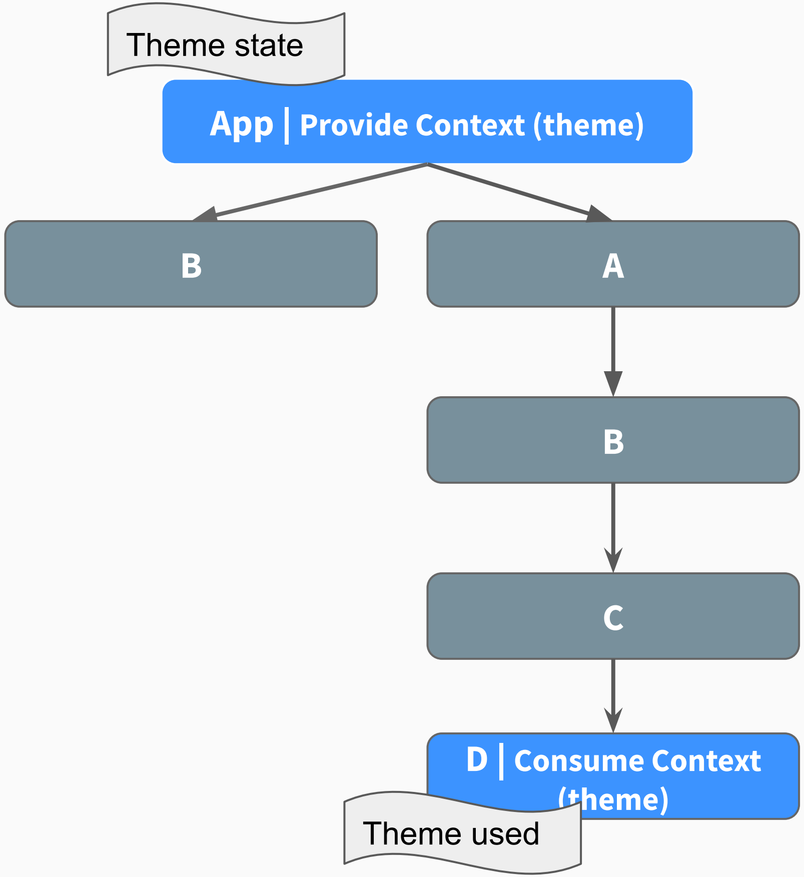
Theme used

React Context



Solution: React Context

- Instead of being passed down, props can be ***tunneled*** through to where they are needed
- **Coupling** is reduced; now only between App and Component D
- Other component can consume props ***on demand*** without requiring changes elsewhere in the tree



Create Context

Wrap subtree in a provider component

```
export const CounterContext = React.createContext({
  count: 0,
  increment: () => {},
  decrement: () => {},
  reset: () => {},
});
```

Consume Context: useContext

Use a hook to retrieve a context value

```
import { useContext } from React;
import ThemeContext from '../context';

export default () => {
  const { theme } = useContext(ThemeContext);
  return (
    <div style={{ backgroundColor: theme.bg }}>
      I am a themed component
    </div>
  );
};
```

Task

Add a ThemeContext to provide a primary color

Problem:

The default value of a React Context is immutable (can not trigger a rerender)

Solution: Context Provider

Provide Context

<code>

Wrap subtree in a provider component

```
import defaultTheme from 'common/themes/default';

const ThemeContext = React.createContext({});

export default () =>
  <ThemeContext.Provider value={{ theme: defaultTheme }}>
    <App />
  </ThemeContext.Provider>
```

Task

Create a ThemeEditor component

We teach.

workshops.de