React

A library for creating user interfaces.

Press Space for next page →

1. What is React.js ?
2. React.js motivations?
3. JSX / TSX
4. VirtualDOM
5. React main concepts
    1. Components
    2. Props
    3. State
6. Component Lifecycle
7. Component development models
    1. Class components
    2. Function components
8. Component hierarchy
9. Conclusion on React

What is React.js ?

What is React.js

- React is a JavaScript library created by a collaboration of Facebook and Instagram
- It’s not MVC, only the "V" in "MVC"
- Big Companies use React
  - Facebook, Instagram, Yahoo!, Airbnb, Sony, Netflix

Why was React.js made?

- React isn’t an MVC framework
- React does not use templates
- React updates are dead simple (Reactive pattern)
- HTML is just the beginning

How is React.js used?

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.6.1/react.js"> </script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.6.1/react-dom.js"> </script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.38/browser.min.js"> </script>
<script src="app.js" type="text/babel"> </script>

<div id="root"></div>
```

Root DOM node

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Render React element into root DOM node

How is React.js used?

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.6.1/react.js"> </script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.6.1/react-dom.js"> </script>

<div id="root"></div>
```

Root DOM node

```
const element = React.createElement('h1', null, 'Hello react');
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Render React element into root DOM node

Update the Rendered Element

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}
setInterval(tick, 1000);
```

Web components

> A React component is a reusable, self-contained piece of code that defines how a portion of the user interface (UI) should appear and behave. React components are the building blocks of a React application, and they can be as simple as a button or as complex as an entire page layout.

It could be seen a simple function that produces a piece of html

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}
```

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

React.js motivations?

Rule 1: Build components, not templates

We all like separation of concerns, right?

Separation of concerns:

> Reduce **coupling**, increase **cohesion**.

Coupling is:

> The degree to which each program module relies on each of the other modules.

wikipedia (Coupling)

Cohesion is:

> "The degree to which elements of a module belong together." wikipedia (Cohesion)

Templates encourage a poor separation of concerns.

So are Angular-style directives.

"View model" tightly couples template to display logic

[{"price": "7.99", "product": "Back scratcher", "tableRowColor": "rgba(0, 0, 0, 0.5)"}]

Display logic and markup are inevitably tightly coupled.

How do you find DOM nodes?

Display logic and markup are highly cohesive.

They both show the UI.

Templates separate technologies, not concerns.

And they do it by being deliberately underpowered.

Symptoms that your front-end technology is underpowered:

Reliance on primitive abstractions like

```
{{ > }}
{{  #each }}
```

Symptoms that your front-end technology is underpowered:

Inventing lots of new concepts (that already exist in JavaScript).

From the Angular directives docs:

> However isolated scope creates a new problem: if a transcluded DOM is a child of the widget isolated scope then it will not be able to bind to anything. For this reason the transcluded scope is a child of the original scope, before the widget created an isolated scope for its local variables. This makes the transcluded and widget isolated scope siblings.

The framework cannot know how to separate your concerns for you.

It should only provide **powerful**, **expressive tools** for the user to do it correctly.

This tool is a React component

A **highly cohesive** building block for UIs **loosely coupled** with other components.

As an architect, use components to separate your concerns

With the full power of JavaScript, not a crippled templating language.

Components are **reusable**

Components are **composable**

Components are **unit testable**

They are, after all, units.

What about spaghetti code?

Just don’t write spaghetti code

Keep your components small.

Just don’t write spaghetti code

Only put *display logic* in your components.

Just don’t write spaghetti code.

> With great power comes great responsibility – Uncle Ben in Spiderman

What about **working with designers**?

JSX / TSX

JSX is an *optional* preprocessor to let you use HTML-like syntax

```
<a href="http://olivier.barais.fr">
    my personal web site
</a>
```

JSX /TSX

JSX is an *optional* preprocessor to let you use HTML-like syntax

```
React.DOM.a(
  {href: 'http://olivier.barais.fr'},
  'my personal web site'
)
```

With JSX, it’s easy for designers to contribute code

The accessibility of templates and the power of JavaScript

VirtualDOM

Rule 2. Re-render the whole app on every update

The **key design decision** that makes React awesome.

Building UIs is hard because there is so much state

Lots of UI elements · Design iteration · Crazy environments · Mutable DOM · User input · etc etc

Data changing over time is the root of all evil

> “Our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible” – Dijkstra

In the 90s it was easier

Just refresh the page when the data changes.

Design decision: When the data changes, **React re-renders the entire component**

That is, **React components are basically just idempotent functions**

They describe your UI at any point in time, just like a server-rendered app.

Re-rendering on every change makes things simple

- Every place data is displayed is guaranteed to be up-to-date.
- No magical data binding.
- No model dirty checking.
- No more explicit DOM operations – **everything is declarative**

"Re-render on every change? That seems expensive."

> “And doesn’t it mess up form fields and scroll position?”

Rule 3. Virtual DOM

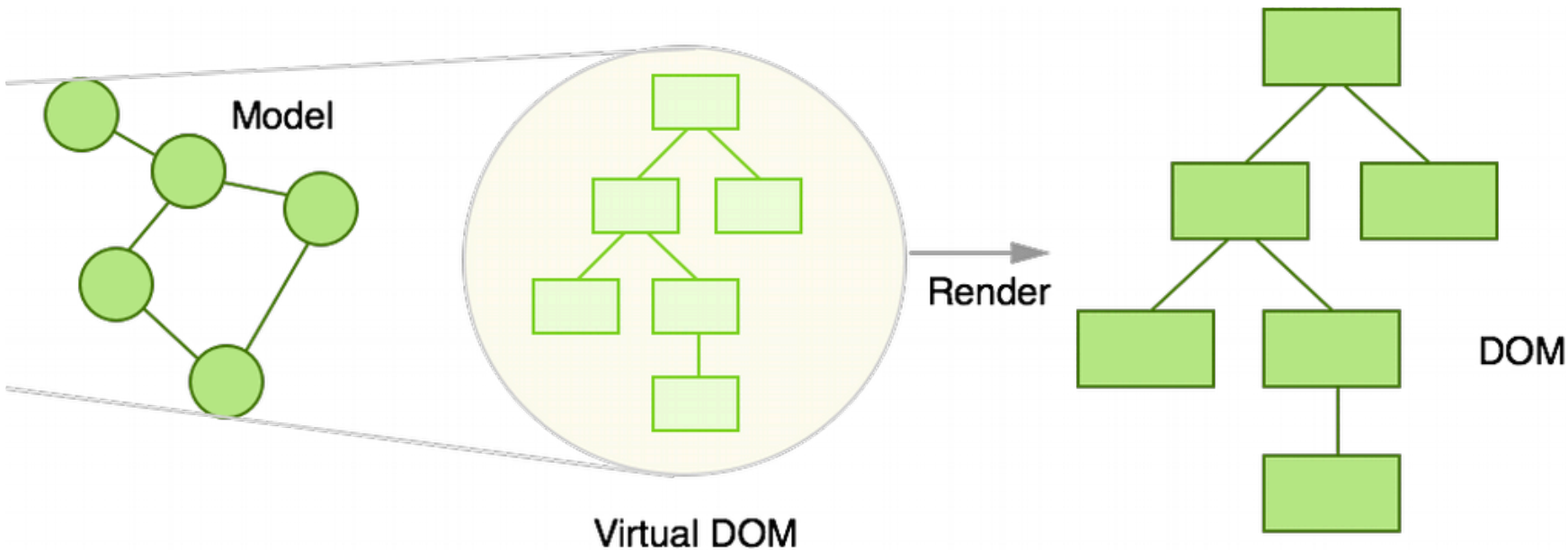
Makes re-rendering on every change fast.

You can’t just throw out the DOM and rebuild it on each update

It’s too slow and you’ll lose form state and scroll position.

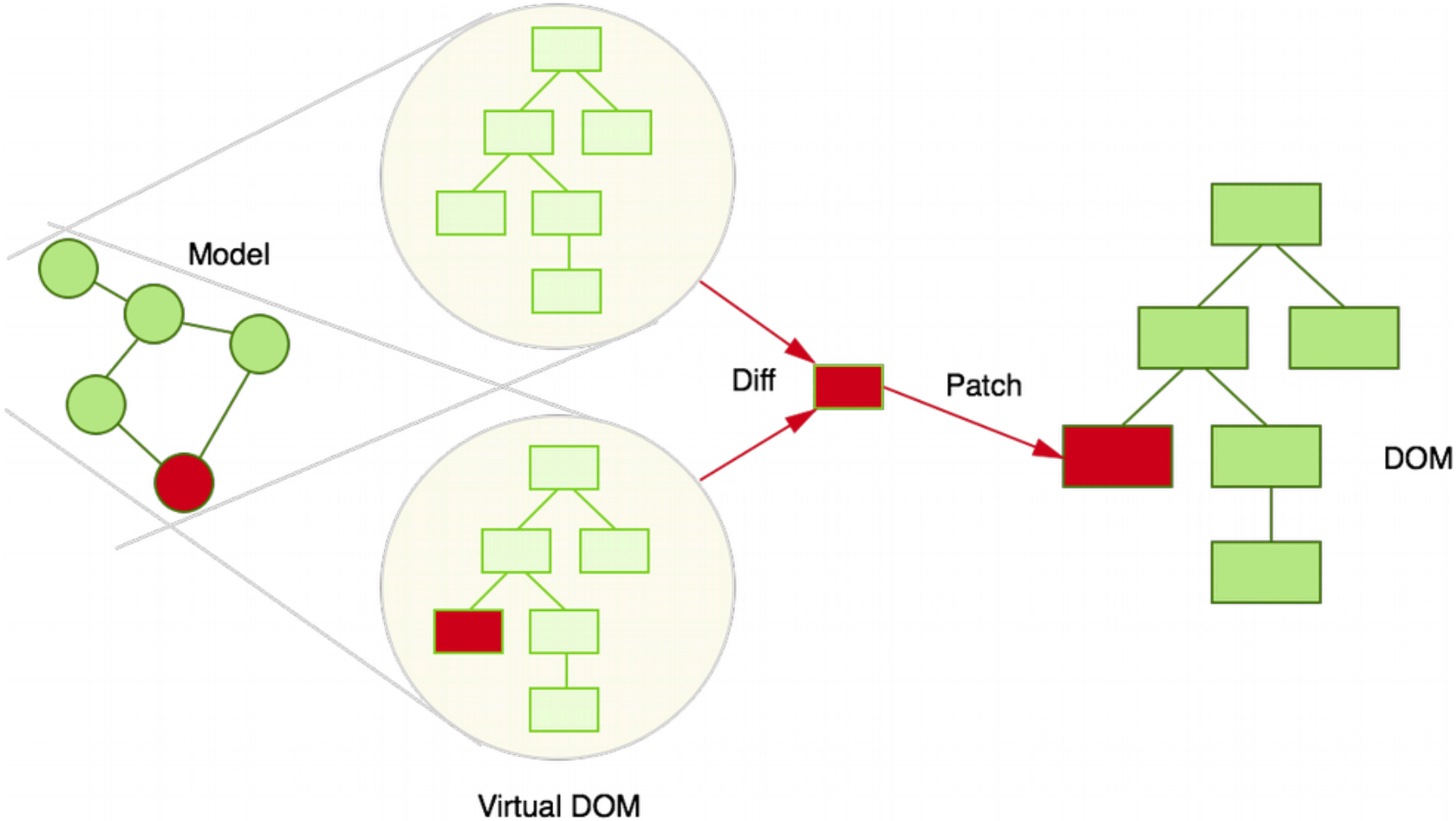
So React built a **virtual DOM** (and events system)

Optimized for performance and memory footprint



On every update…

- React builds a new virtual DOM subtree
- … diffs it with the old one
- … computes the minimal set of DOM mutations and puts them in a queue
- … and batch executes all updates


Model
Diff
Patch
DOM
Virtual DOM

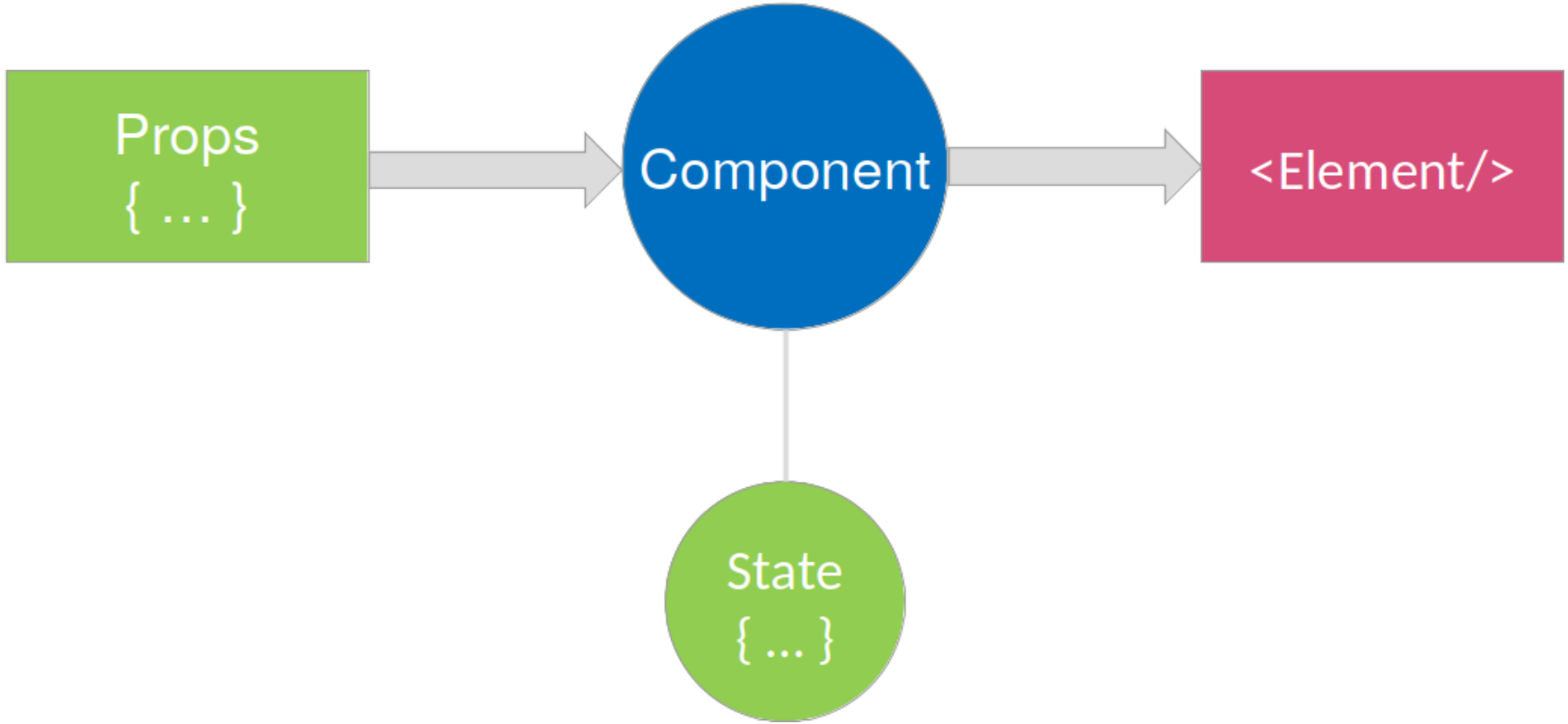
It’s **fast**!

Because the DOM is slow!

The virtual DOM lets us do fun things

- **Testability** for free
- **SVG**, **VML** and **canvas** support
- Running the whole app in a Web Worker

React main concepts


Props
{ … }
Component
<Element/>
State
{ … }

Components

> A React component is a reusable, self-contained piece of code that defines how a portion of the user interface (UI) should appear and behave. React components are the building blocks of a React application, and they can be as simple as a button or as complex as an entire page layout.

It could be seen a simple function that produces a piece of html

```
import { useState } from 'react';

export default function Counter(props) {
  const [count, setCount] = useState(props.counter);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      You pressed me {count} times
    </button>
  );
}
```

Props

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="Olivier" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

State

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {
    this.timerID = setInterval(() => this.tick(),1000);
  }
  componentWillUnmount() { clearInterval(this.timerID);}
  tick() {
    this.setState({ date: new Date()});
  }
  render() {
    return
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
  }
}
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Component Lifecycle

- Mounting
- Updating
- Unmounting

Mounting

1. constructor()
2. componentWillMount()
3. render()
4. componentDidMount()

Updating

1. componentWillReceiveProps()
2. shouldComponentUpdate()
3. componentWillUpdate()
4. render()
5. componentDidUpdate()

Unmounting

1. componentWillUnmount()

Component development models

- Class components
- Function components

Class components

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

a simple class with a method render

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

Class components with props

```
class Car extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <h2>I am a {this.props.model}!</h2>;
  }
}
}
```

a simple class with a method render

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car model="Mustang"/>);
```

Components in Components

```
class Car extends React.Component {
  render() {
    return <h2>I am a Car!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my Garage?</h1>
      <Car />
      </div>
    );
  }
}
```

a simple class with a method render

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

React Class Component State

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
    };
  }
  changeColor = () => { this.setState({color: "blue"});  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>
      </div>
    ); }
}
```

React Class Component lifecycle methods

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

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

At first my favorite color is red, but give me a second, and it is yellow instead:

Function components

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}
```

a simple function that produce a piece of html

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

Function components with Props

```
function Car(props) {
  return <h2>I am a {props.color} Car!</h2>;
}
```

a simple function with parameter

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red"/>);
```

Components in Components

```
function Car() {
  return <h2>I am a Car!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}
```

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

Limitations with functional dev model

Access to *lifecycle methods*, *changing state*, ... are a bit painfull

=> Hooks to the rescue

What are Hooks ?

(Ques.) What is Hook in General terms ?

> (Ans.) a curved piece of metal, plastic, etc. that is used for hanging or catching something

What are Hooks in React ?

> Hooks are a new feature addition in React version 16.8 which allow you to use React features without having to write a class.

> *Ex: State of a Component*

Motivations behind Hooks

Why Hooks ?

Reason 1: Classes confuse bath people and machines

- Understand how this keyword works in JavaScript
- Remember to bind event handlers in class components
- Classes don’t minify very well and make hot reloading very unreliable

> Better in Typescript ;)

Why Hooks ?

Reason 2: It's hard to reuse stateful logic between components

- There is no particular way to reuse stateful component logic
- HOC (Higher Order Component) and render props patterns do address this problem but have to do some restructuring of components
- Makes the code harder to follow
- There is need a to share stateful logic in a better way

Why Hooks ?

Reason 3: Complex components become hard to understand

- Create components for complex scenarios such as data fetching and subscribing to events
- Related code is not organized in one place
- Ex: Data fetching - In *componentDidMount* and *componentDidUpdate*
- Ex:Event Listeners - In *componentDidMount* and *componentWillUnmount*
- Because of stateful logic - Cannot break components into smaller ones

Types and Rules of Using Hooks

Types

Built-In Hooks :

- useState Hooks
- useEffect Hook
- useRef Hook
- useCallback Hook
- useMemo Hook
- useContext Hook
- useReducer Hook

Custom Hooks :

> You can create your own custom hooks if you have stateful logic that is needed by multiple components in your application.

Rules

Hooks are normal JavaScript functions,but they Impose some additional rules :

- Hooks are called only at the toplevel of a component.
- Do not call Hooks inside loops, conditions, or nested functions.
- Hooks are called only **from React functional Components**.
- Do not call Hooks from regular JavaScript functions.

There is one other way to call Hooks i.e. in your own custom Hooks

Let us use your first Hooks

useState(initialState)

Call useState at the top level of your component to declare a state variable.

```
import { useState } from 'react';

export default function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      You pressed me {count} times
    </button>
  );
}
```

https://react.dev/reference/react/hooks

Component hierarchy

Component hierarchy

```
function Car() {
  return <h2>I am a Car!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}
```

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

Rule 4: a flow is unidirectional

All data flows down the component hierarchy

Communication between Components

- props and context
- 8 no-Flux strategies for React component communication

Communication between Components

- Use Flux or Redux

And it works with Typescript

- demo

```
npx create-react-app my-app --template typescript
cd my-app
npm start
```

Conclusion on React

- Components, not templates.
- Three main concepts:
  - Components
  - Props
  - State
- Re-render, don’t mutate.
- Virtual DOM is **simple** and **fast**
- https://facebook.github.io/react/