React JS

(v 16.x.onwards)

Created by :

Sangeeta Joshi

Agenda

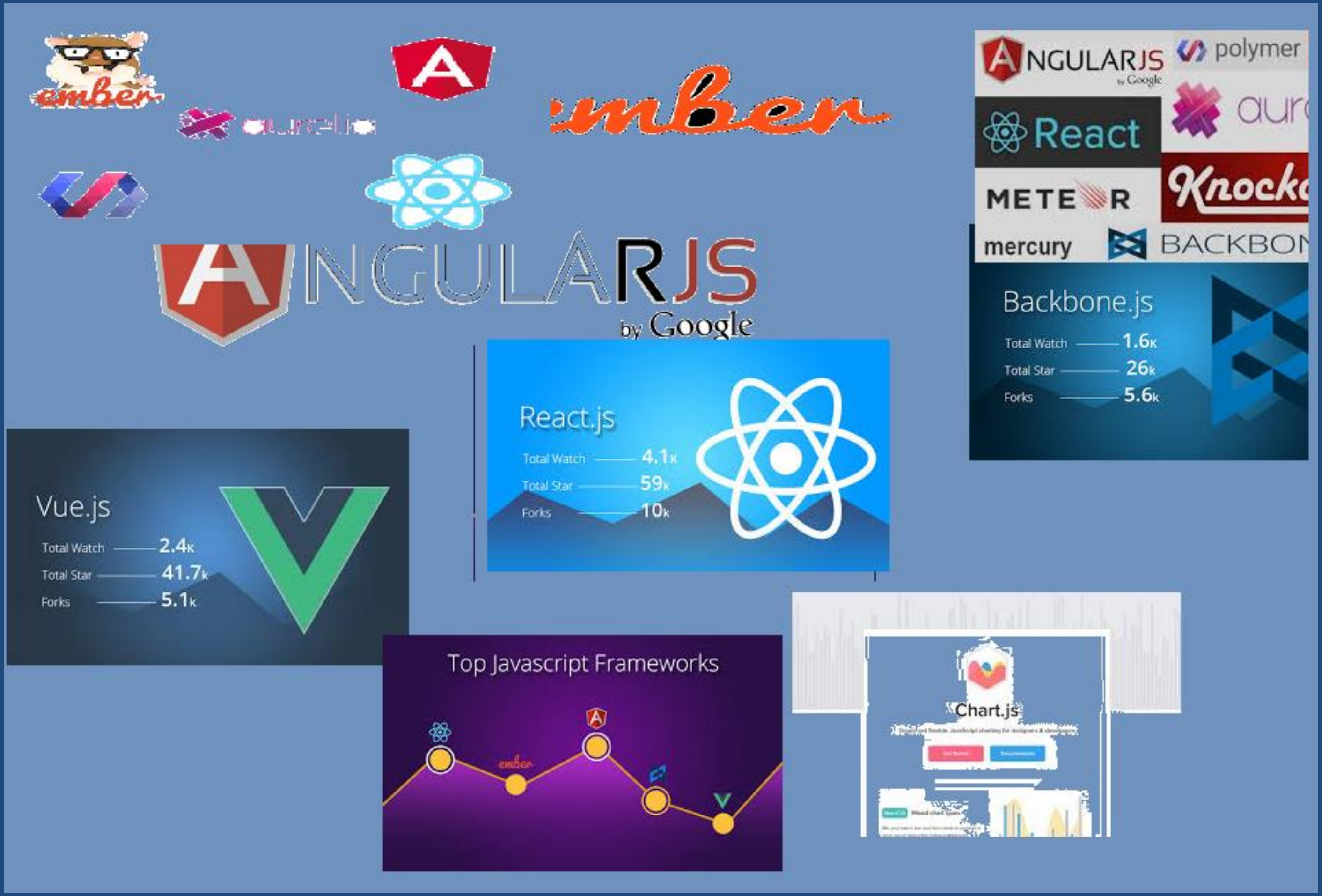
- React Overview & Component Architecture
- Starting with React
- Hello World
- Introduction to JSX
- Elements
- Components & Props
- Functional Components
- Components created using class
- State
- Component Life Cycle

What is React?

A JS Library for building User Interfaces

Target : Great User Experience

JavaScript Frameworks



Architectures

MVC

MVVM

MVW

What is React

- React is not a framework
- It’s a J S Library
- It’s a view rendering engine or a component model.
- React focuses on view/UI layer i.e rendering

Components

Components : Declarative approach

- Reusable APIs
- Encapsulate Behavior (CSS,JS etc)
- Hides implementation details
- Something like pick a comp & drop it in page where you need it

Declarative Components

Components :

- More Template building
- Invokes some function
- No Explicit data binding

(As in angular : changes in data causes view updates & vice a versa)

Components & Props

- Entire view/UI is split into multiple components where component is:
- Component :

  independent

  reusable pieces

  where each piece can be considered in isolation.

- Conceptually, components are like JavaScript functions.

Starting with React

React is the entry point to the React library.

- If you load React from a <script> tag, these top-level APIs are available on the React global.
- If you use ES6 with npm, you can write import React from 'react'.
- If you use ES5 with npm, you can write

  var React = require('react').

Diving in

Libraries to import:

React

ReactDOM

ReactDOM :

- glue between React and the DOM.
- Often, you will only use it for one single thing: mounting with *ReactDOM.render().*
- Another useful feature of ReactDOM is *ReactDOM.findDOMNode()* which you can use to gain direct access to a DOM element.
- If your app is "isomorphic", you would also use *ReactDOM.renderToString()* in your back-end code.

React:

for everything else, there's React. You use React to define and create your elements, for lifecycle hooks, etc

Elements

- smallest building blocks of React Apps
- describes what you want to see on the screen

Example:

```
    const element=<h1> Hello World</h1>;
```

Rendering element:

```
ReactDOM.render(element,
document.getElementIdBy(‘root’));
```

Hello World

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

http://codepen.io/sangeetaj/pen/KmNEax

Components & Props

- Conceptually, components are like JavaScript functions.

- They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Components

Always start component names with a Capital letter.

For example,

- <div /> represents a DOM tag,
- <Welcome /> represents a component and requires Welcome to be in scope.

JSX

```
const element = <h1>Hello, world!</h1>
```

- This funny tag syntax is *neither a string nor HTML*
- It is JSX
- *It is a syntax extension to JavaScript*.
- Use it with React to describe : ***what the UI should look like***.
- JSX looks like a template language, but it comes with the full power of JavaScript
- JSX produces React "elements".

JSX

- Embedding Expressions in JSX : { }

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

For example :

- 2 + 2
- user.firstName
- formatName(user)

JSX is an Expression

- After compilation, JSX expressions become regular ***JavaScript objects***.
  - Can use JSX inside of if statements and for loops,
  - assign it to variables,
  - accept it as arguments,
  - return it from functions

JSX is an Expression

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

- Specifying Attributes with JSX

```
const element = <div tabIndex="0"></div>;
```

- use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

JSX

- Fundamentally, JSX just provides syntactic sugar for :

React.createElement(component, props, ...children) function.

JSX

- The JSX code:

```
<MyButton
 color="blue" shadowSize={2}> Click Me
</MyButton>
```

compiles into:

```
React.createElement( MyButton,
 {color: 'blue', shadowSize: 2}, 'Click Me' )
```

Choosing the Type at Runtime

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Wrong! JSX type can't be an expression.
  return <components[props.storyType] story={props.story} />;
}
```

You cannot use a general expression as the React element type

Choosing the Type at Runtime

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Correct! JSX type can be a capitalized variable.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

we will assign the type to a capitalized variable first:

Components

Always start component names with a Capital letter.

For example,

- <div /> represents a DOM tag,
- <Welcome /> represents a component and requires Welcome to be in scope.

Functional Components

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

- It’s a valid functional component as
  - it accepts a single "props" object with data
  - returns a React element.
- We call such components "functional" because they are literally JavaScript functions.

Demo - http://codepen.io/sangeetaj/pen/OmWgPW

props

- When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "*props*".
- const element = <Welcome name="Sara" />;

Components & Props

```
function WelcomeComp(props)
{
  return <h1>Hello,{props.name}</h1>
};
const element = <WelcomeComp name="Sangeeta" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Props are Read Only

- Whether a component is a function or a class, it must never modify its own props.
- React is pretty flexible but it has a single strict rule:
- **All React components must act like pure functions with respect to their props**

Composing Components

- Components can refer to other components in their output.
- This lets us use the same component abstraction for any level of detail.

https://codepen.io/sangeetaj/pen/QvvmOg?editors=1111

Extracting Components

- split components into smaller components
- having a palette of reusable components pays off in larger apps.
- *A good rule of thumb : if a part of UI is used several times (Button, Panel etc),*

  or

  is complex enough on its own (App, FeedStory, Comment),

it is a good candidate to be a reusable component.

Class Components

- We can use an ES6 class to define component:

```
class Welcome extends React.Component {
render() {
return <h1>Hello, {this.props.name}</h1>;
   }
 }
```

- Classes have some additional features

http://codepen.io/sangeetaj/pen/mmmxQP

State

State:

- application UIs are dynamic and change over time.
- State allows React components to change their output over time in response to :

    - user actions

    - network responses

    - and anything else

without violating “Props” rule.

State

components defined as classes have some additional features.

State:

- State is similar to props, but it is private and fully controlled by the component
- Local state : a feature available only to classes.

Using State Correctly

- Do Not Modify State Directly:

```
// Wrong
this.state.comment = 'Hello';
```

- Instead, use setState():

```
// Correct
this.setState({comment: 'Hello'});
```

- ***The only place where you can assign this.state is the constructor.***

PropsType

Inheritance vs composition

To reuse code between components:

composition is recommended

over

inheritance .

Inheritance vs composition

Scenarios :

- a WelcomeDialog is a special case of Dialog.
- a more “specific” component renders a more “generic” one and configures it with props:

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

Inheritance vs composition

- Some components don’t know their children ahead of time.
- This is especially common for components like Sidebar or Dialog that represent generic “boxes”.
- such components use special *children prop* to pass children elements directly into their output:

Inheritance vs composition

```
function FancyBorder(props)
        {
            return (
     <div className={'FancyBorder FancyBorder-' + props.color}>
        {props.children}
      </div> );


    }
```

Inheritance vs composition

```
function WelcomeDialog()
{ return
  ( <FancyBorder color="blue">
     <h1 className="Dialog-title"> Welcome </h1>
     <p className="Dialog-message">
      Thank you for visiting our spacecraft! </p>
     </FancyBorder> );
}
```

Inheritance vs composition

this.props.children :

it is used to display whatever we include

between the opening and closing tags

when ***invoking*** a component.

Virtual DOM

- Dynamic View : Manipulation of DOM

  Steps involed

  - The browser parses the HTML to find node with this id.
  - It removes the child element of this specific element.
  - Updates the element(DOM) with the ‘updated value’.
  - Recalculates the CSS for the parent and child nodes.
  - Update the layout.
  - Finally, traverse the tree and paint it on the screen(browser) display.

Virtual DOM

- So updating the DOM
  - not only involves changing the content, it has a lot more attached to it.
  - Also recalculating the CSS and changing the layouts involves complex algorithms and they do affect the performance.
  - So React has a different approach of dealing with this, as it makes use of something known as *Virtual DOM*.

Virtual DOM

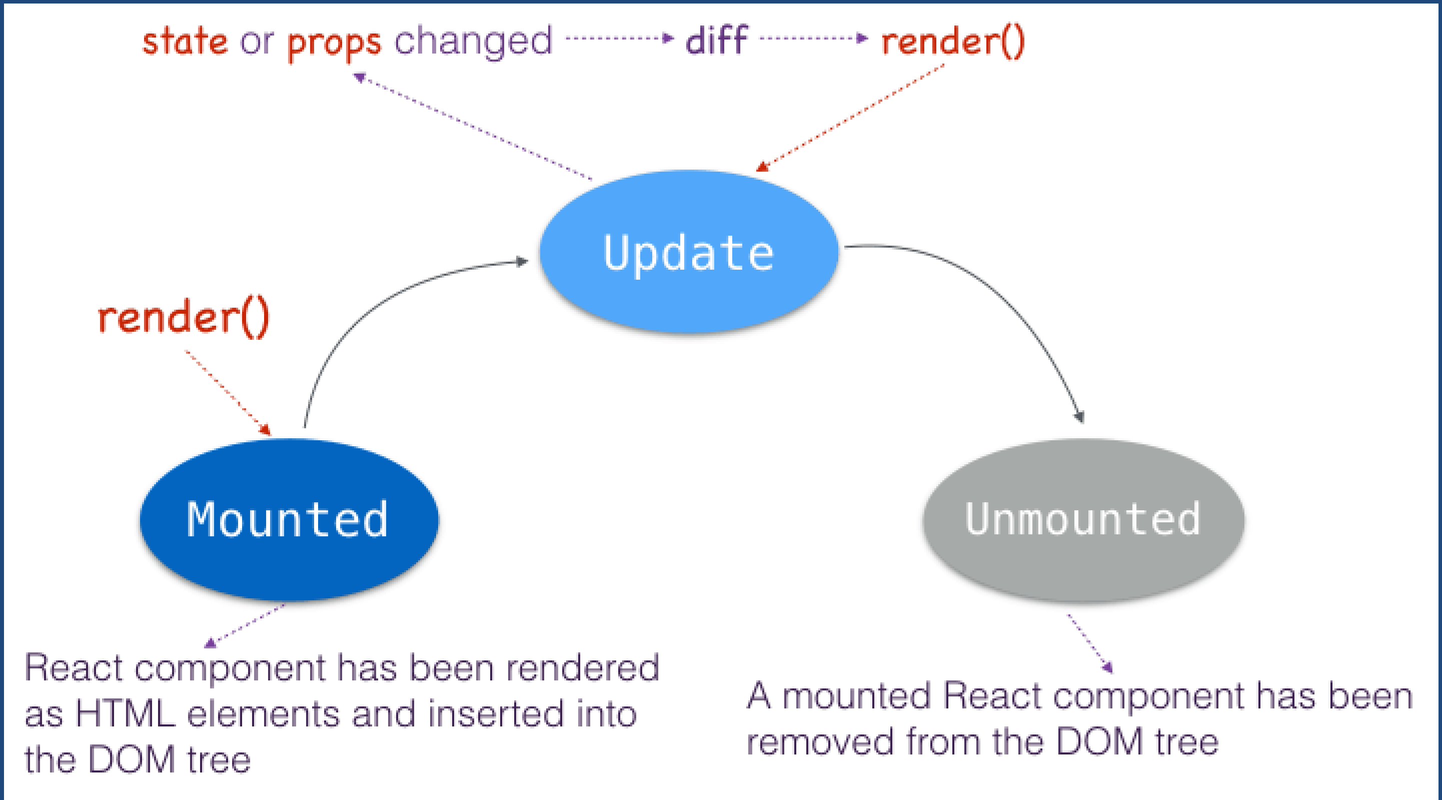
- **Virtual DOM:**
  - Virtual DOM is like *a lightweight copy* of the actual DOM.
  - So for every object that exists in the original DOM there is an object for that in React Virtual DOM.
  - It is exactly the same, but *without power to directly change the layout* of the document.
  - Manipulating DOM is slow, but *manipulating Virtual DOM is fast* as nothing gets drawn on the screen.

Virtual DOM

- **How Virtual DOM helps React?**
  - Each time we change something in JSX, all the objects in the virtual DOM get updated.
  - React maintains two Virtual DOM every time,
    - one contains the updated Virtual DOM
    - and one which is just the pre-update version of the Virtual DOM.
    - It compares pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM. This process is known as ‘diffing’.
    - Once React finds out what exactly has changed then it updates those objects only, on real DOM.
    - This significantly improves the performance

Life Cycle

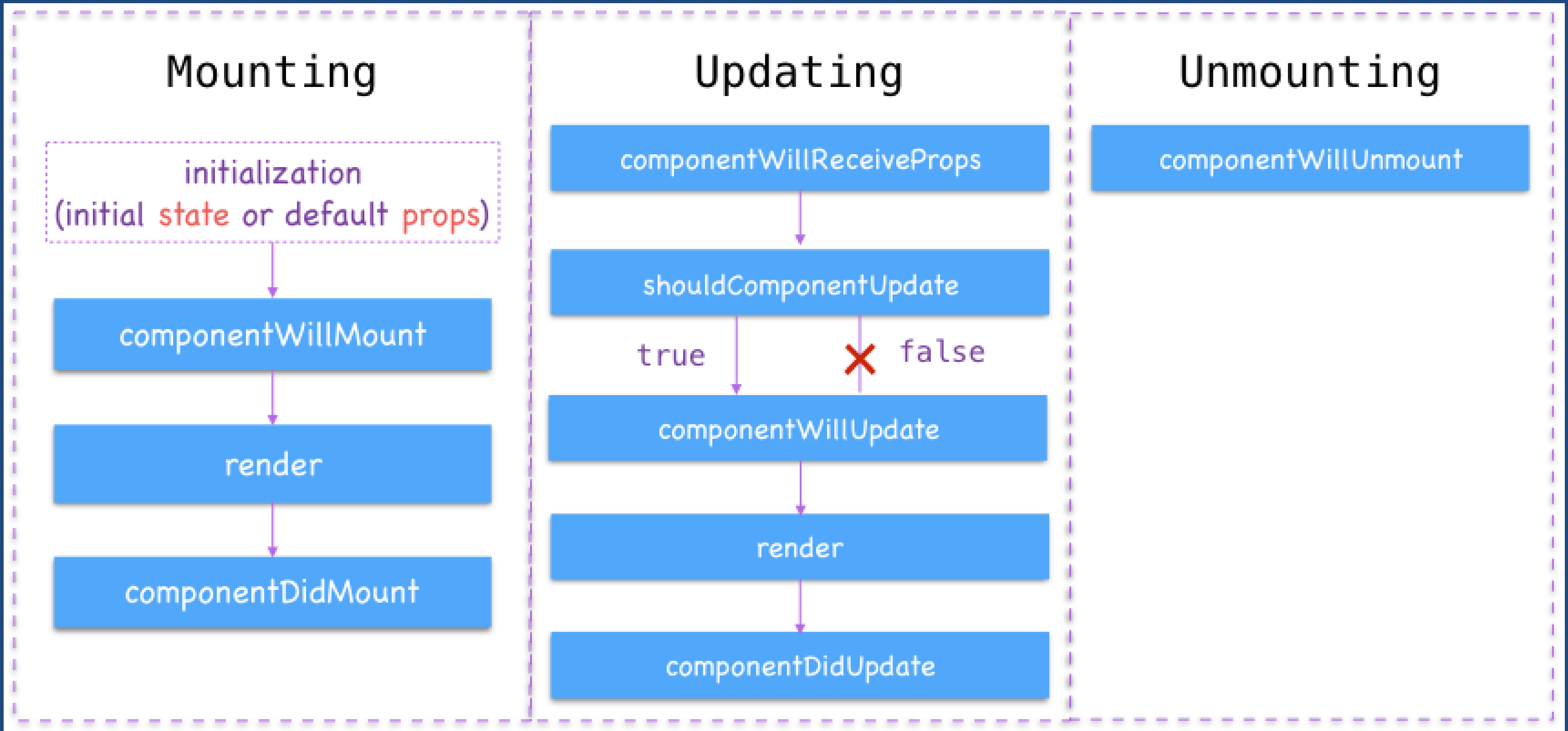
Three Stages :   mounted ,update ,unmounted



React component has been rendered as HTML elements and inserted into the DOM tree

A mounted React component has been removed from the DOM tree

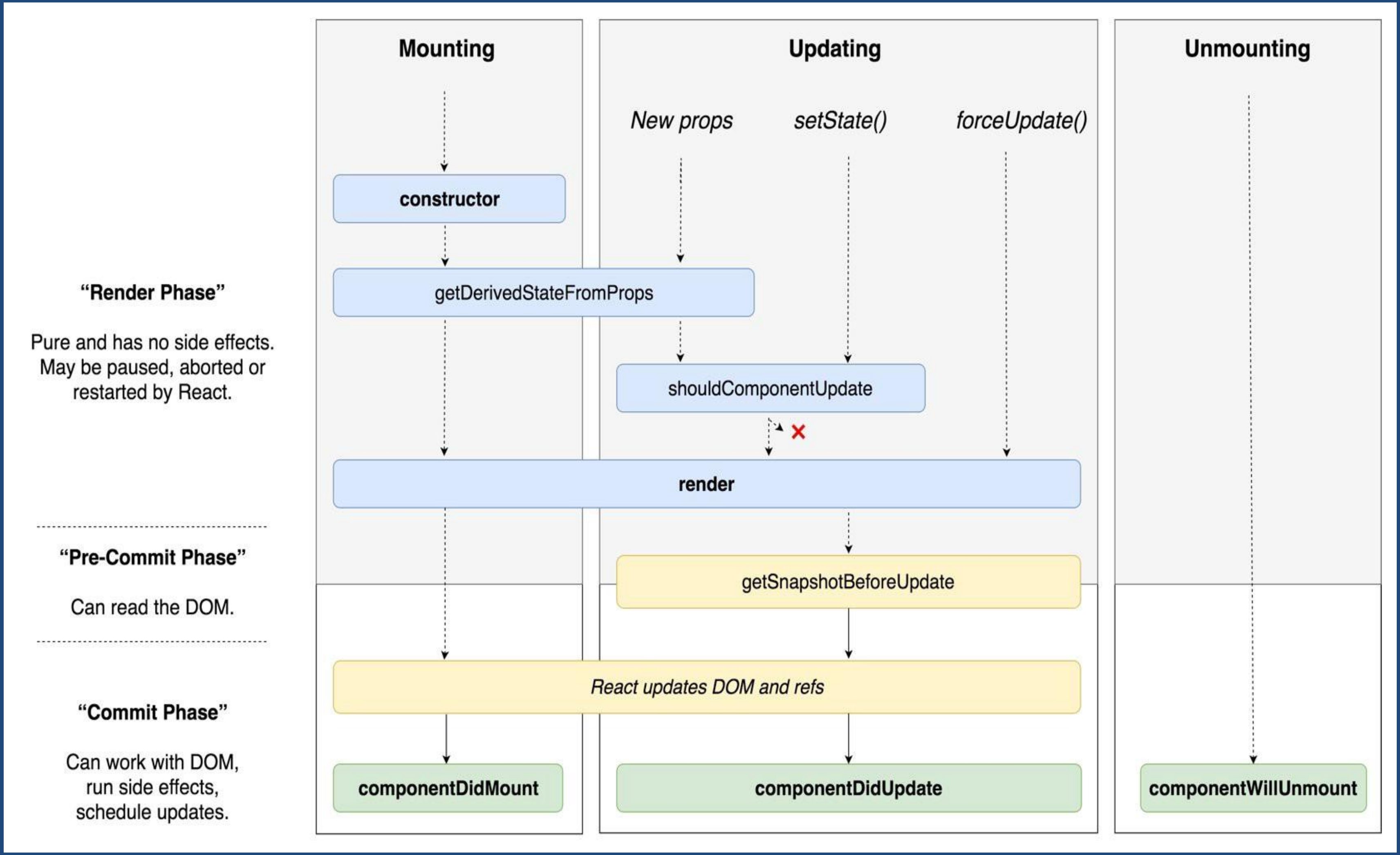
Life Cycle –Refer revised one



Life Cycle –revised (after 16.3V)

- Version-16.3 introduced new life-cycle:
  - Replaces some existing methods to provide better support for *new asynchronous nature* of React.

Life Cycle –revised (after 16.3V)



1.New Life Cycle Methods

1.Constructor:

- perfect for setting up our Component —
  - create any fields (variables starting with this.)
  - initialize state based on props received.
- This is also the only place where you are expected to change / set the state by directly overwriting the this.state fields.

2.New Life Cycle methods

2. **static getDerivedStateFromProps(nextProps, prevState)**
   - main responsibility - ensuring state and props are in *sync* for when it is required.
   - is a static function and has no access to 'this'
   - used when a component is updated & also when it is mounted, right after the constructor was called

3.New Life Cycle methods

- render()

4.New Life Cycle methods

- 4. getSnapshotBeforeUpdate(prevProps, prevState)
  - invoked in “pre-commit phase”, right before the changes from VDOM are to be reflected in the DOM.
  - It is usable mostly if you need to read the current DOM state
  - Even though the function is not static, it is recommended to return the value, not update the component.
  - The returned value will be passed to componentDidUpdate as the 3rd parameter.

Life Cycle Method:Unmounting Phase

- componentWillUnmount()
  - Invoked immediately before a component is unmounted or removed from the DOM.
  - Use this as an opportunity to perform cleanup operations. For example, unbind event listeners here to avoid memory leaking.

```
 class ScoreBoard extends React.Component {
        componentWillUnmount() {
          window.removeEventListener('scroll', this._handleScroll);
      }
}
```

VDOM /DOM

Rendering

Two important things:

1. How initially things are rendered
2. How updates happen

A. Initial Rendering

Step 1 : Initial Rendering

(Unlike to other frameworks who work on creating DOM & wiring up events)

- render(){….} (just one function)

- a) Describe how your component looks at any point in time
- b) Does not return a string but returns:

“Representation of your view”

It means we generate markups as an Element & inject it into document

A. Initial Rendering

Step 1 : Initial Rendering ....continued

Two Pass Rendering :

1. Generate Mark up & inject it into DOM
2. Attach Events

Because of this separation, rendering can happen on Server or Client

B. Reconcile

~~Update~~ Reconcile

If data changes updates view

call render () ………get a representation

if something happens,again

call render () ……..get another representation

Compare two representations ,compute minimum differences and do minimum updates

React Hooks (16.3 V)

React Fragment