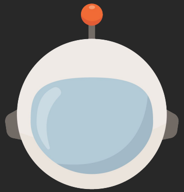
POSTMAN

React Internals



PRESENTED BY

Ankit Muchhala

Software Developer @ Postman

August 18, 2018

Contents

1. React Philosophy
2. Internal Structure
3. Stack Reconciliation
4. Fiber Reconciliation
5. Outro

POSTMAN

1. React Philosophy

$$y = f(d)$$

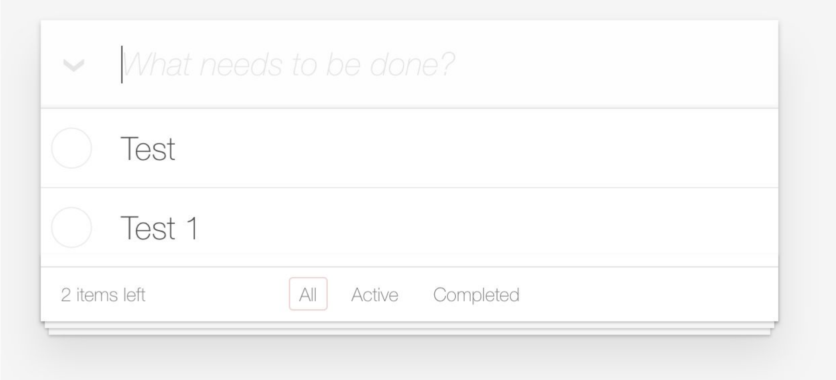
- UI is a projection of some data
- Declarative structure of code
- Component based

POSTMAN

Data

```
{
  items: [{
    name: "Test",
    active: true,
    completed: false
  }, {
    name: "Test 1",
    active: true,
    completed: false
  }]
}
```

User Interface



What needs to be done?

Test

Test 1

2 items left

Declarative

```
function Button (props) {
  return (
    <button
      className=`btn ${props.color}`
      onClick={this.handleChange}
    >
      Sample
    </button>
  );
}
```

Imperative

```
const con = document.getElementById(‘container’);
const btn = document.createElement(‘button’);
btn.className = ‘btn red’;
btn.innerHTML = ‘Sample’;
btn.onclick = function(event) {
  // handle click
};

con.appendChild(btn);
```

Component Based

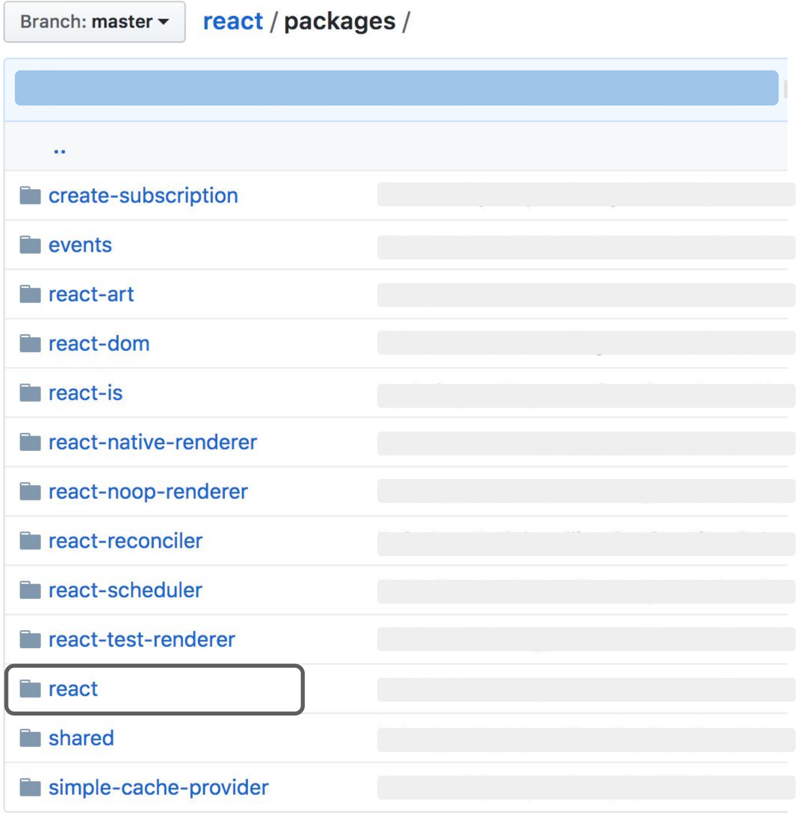
2. Internal Structure

POSTMAN

React

- This is where the **core public API resides**.

- It provides methods to create components and elements.



Branch: master ▾ react / packages /

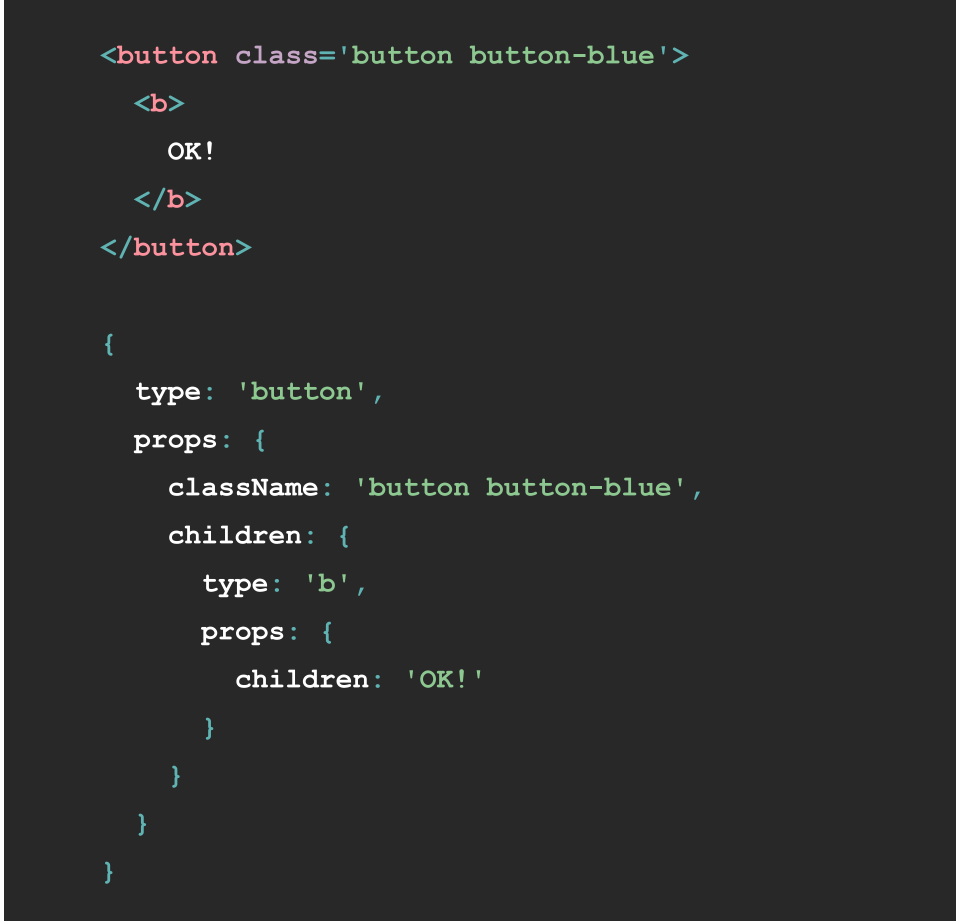
..
create-subscription
events
react-art
react-dom
react-is
react-native-renderer
react-noop-renderer
react-reconciler
react-scheduler
react-test-renderer
react
shared
simple-cache-provider

Elements, Components & Instances

- An element is a **plain object** describing a component instance or DOM node.

- A DOM element will have `string` type and custom components will have a `function` type.

- Instances are never accessed publicly.



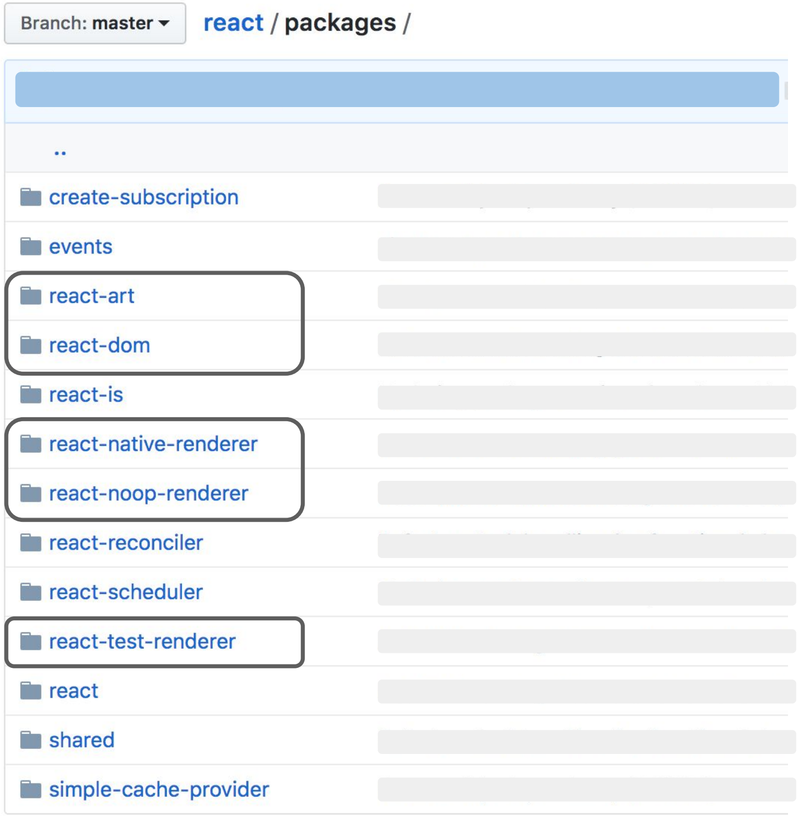
```
<button class='button button-blue'>
  <b>
    OK!
  </b>
</button>

{
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}
```

POSTMAN

Renderers

- They take care of applying the element tree to the host environment.
- It applies the **minimal set of changes** to the host environment to update the UI.
- This decoupling of renderers allows react to be used in multiple environments - VR, mobile, web, etc.



Branch: master ▾ react / packages /

..

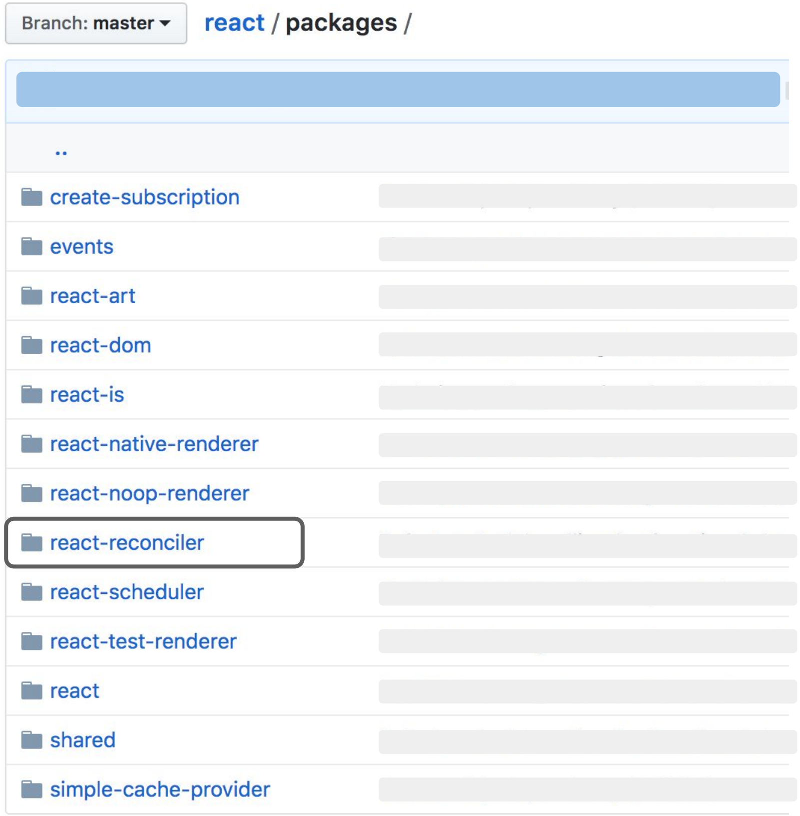
- create-subscription
- events
- react-art
- react-dom
- react-is
- react-native-renderer
- react-noop-renderer
- react-reconciler
- react-scheduler
- react-test-renderer
- react
- shared
- simple-cache-provider

POSTMAN

Reconciler

- Renderers like React DOM use it to update the UI according to the React components.

- Reconciler is responsible for mounting, unmounting and updating the element tree.



Branch: master ▾ react / packages /

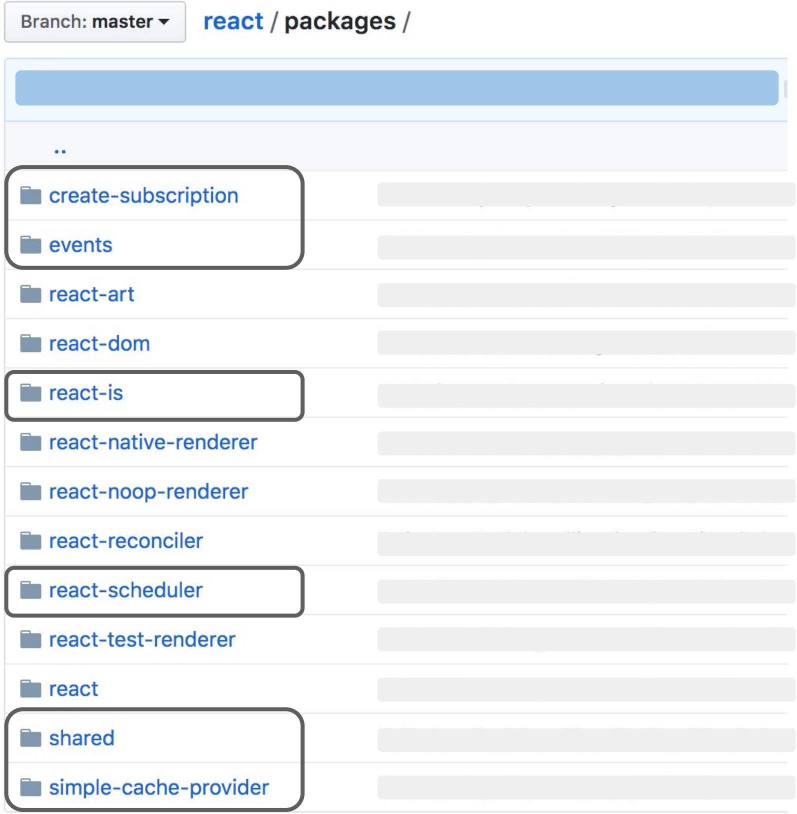
..
create-subscription
events
react-art
react-dom
react-is
react-native-renderer
react-noop-renderer
react-reconciler
react-scheduler
react-test-renderer
react
shared
simple-cache-provider

POSTMAN

Helpers

- These are utilities for internal workings of React and renderers.

- Some of them do have a public API.



Branch: master ▾ react / packages /

..

create-subscription

events

react-art

react-dom

react-is

react-native-renderer

react-noop-renderer

react-reconciler

react-scheduler

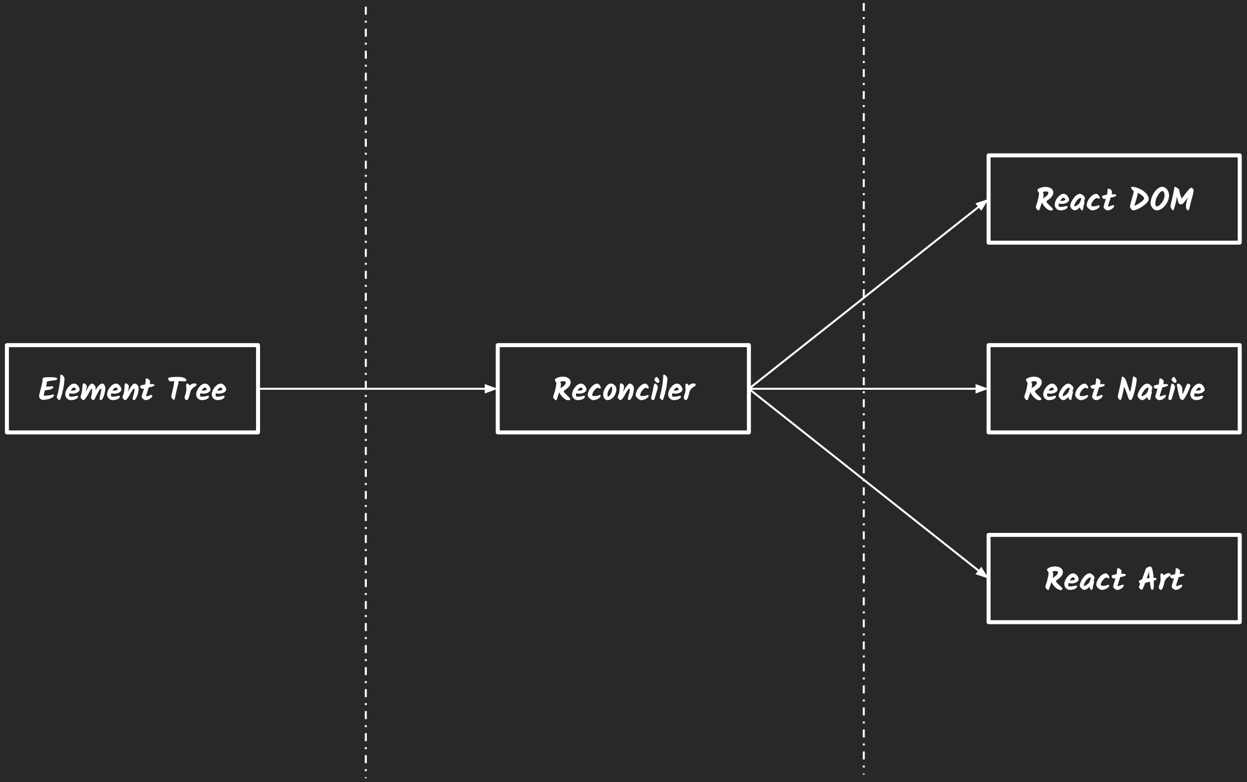
react-test-renderer

react

shared

simple-cache-provider

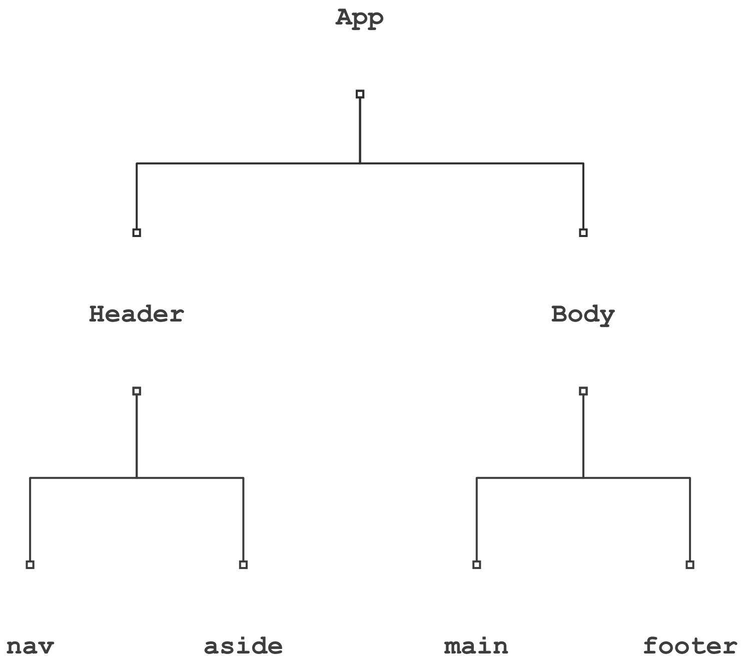
POSTMAN



3. Stack Reconciler

Mounting

- Mounting is the process where the reconciler builds the element tree from all the components.
- It happens when you make a call to `React.render`.
- This process is recursive.
- This recursion goes on until it comes across a leaf host node with no children.



Mounting components

- Custom components can either be of class type or function type.
- React will get the element from the component based on the type.
- It will then recursively mount children.

```
mountComposite (el)

if isClass(el)
    renderedEl = (new el()).render()
else
    renderedEl = el()

mount(renderedEl)
```

Mounting host elements

- If element’s type property is a string, it is a host element.
- When the reconciler encounters a host element, it lets the renderer take care of mounting it.
- If the host element has children, the reconciler recursively mounts them.

```
function mount(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return mountComposite(element);
  }
  else if (typeof type === 'string') {
    // Platform-specific components
    return mountHost(element);
  }
}
```

Internal Instances

- To provide a uniform interface between a DOM element and composite element, **React wraps each element in separate classes with same public methods.**

- Each component is instantiated and an internal reference is maintained.

```
function instantiateComponent(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return new CompositeComponent(element);
  } else if (typeof type === 'string') {
    // Platform-specific components
    return new DOMComponent(element);
  }
}
```

```
class CompositeComponent {
  constructor(element) {
      // ....
  }

  getPublicInstance() {
      // ....
  }

  mount() {
      // ....
    return renderedComponent.mount();
  }
}
```

```
class DOMComponent {
  constructor(element) {
      // ....
  }

  getPublicInstance() {
      // ....
  }

  mount() {
    //.....
    return node;
  }
}
```

Unmounting

- Unmounting is the **process of destroying an element tree.**
- This happens when a component is removed or its type has changed.
- Just like mounting, unmounting is also a recursive process.
- At the end, the `innerHTML` of the parent is set to an empty string.

```
class CompositeComponent {
  unmount() {
    // Unmount the single rendered component
    var renderedComp = this.renderedComponent;
    renderedComp.unmount();
  }
}

class DOMComponent {
  unmount() {
    // Unmount all the children
    var renderedChildren = this.renderedChildren;
    renderedChildren.forEach(child =>
child.unmount());
  }
}
```

How does the UI change?

- ReactDOM.render
- setState
- forceUpdate *

* not recommended

Diffing Algorithm

- State of the art algorithms are $O(n^3)$

- React uses heuristics
  - **Different component types are assumed to generate substantially different trees**. React will not attempt to diff them, but rather replace the old tree completely.
  - **Diffing of lists is performed using keys**. Keys should be "stable, predictable, and unique."

Update Composite

- Heuristic diffing algorithm
- Update props and render if the type remains the same
- Unmount and recreate if type is different

```
receive(nextElement) {
  // …

  if (isClass(type)) {
    nextRenderedEl = publicInstance.render();
  } else if (typeof type === 'function') {
    nextRenderedEl = type(nextProps);
  }

  if (prevRenderedEl.type === nextRenderedEl.type) {
    prevRenderedComp.receive(nextRenderedEl);
    return;
  }

  // Unmount and replace
}
```

Update Host

- Update the attributes of the current node.

- Quantify all the actions needed on the children in terms of `ADD`, `REPLACE`, `REMOVE` or `MOVE` operations.

- All these operations are put in a queue and executed in one go.

```
1.    Update attributes of the current node
2.    Iterate over new children
        a.    Is it ADD or REPLACE operation
        b.    Append operation
3.    Check for children removed
4.    Append REMOVE operations
5.    Flush all operations
```

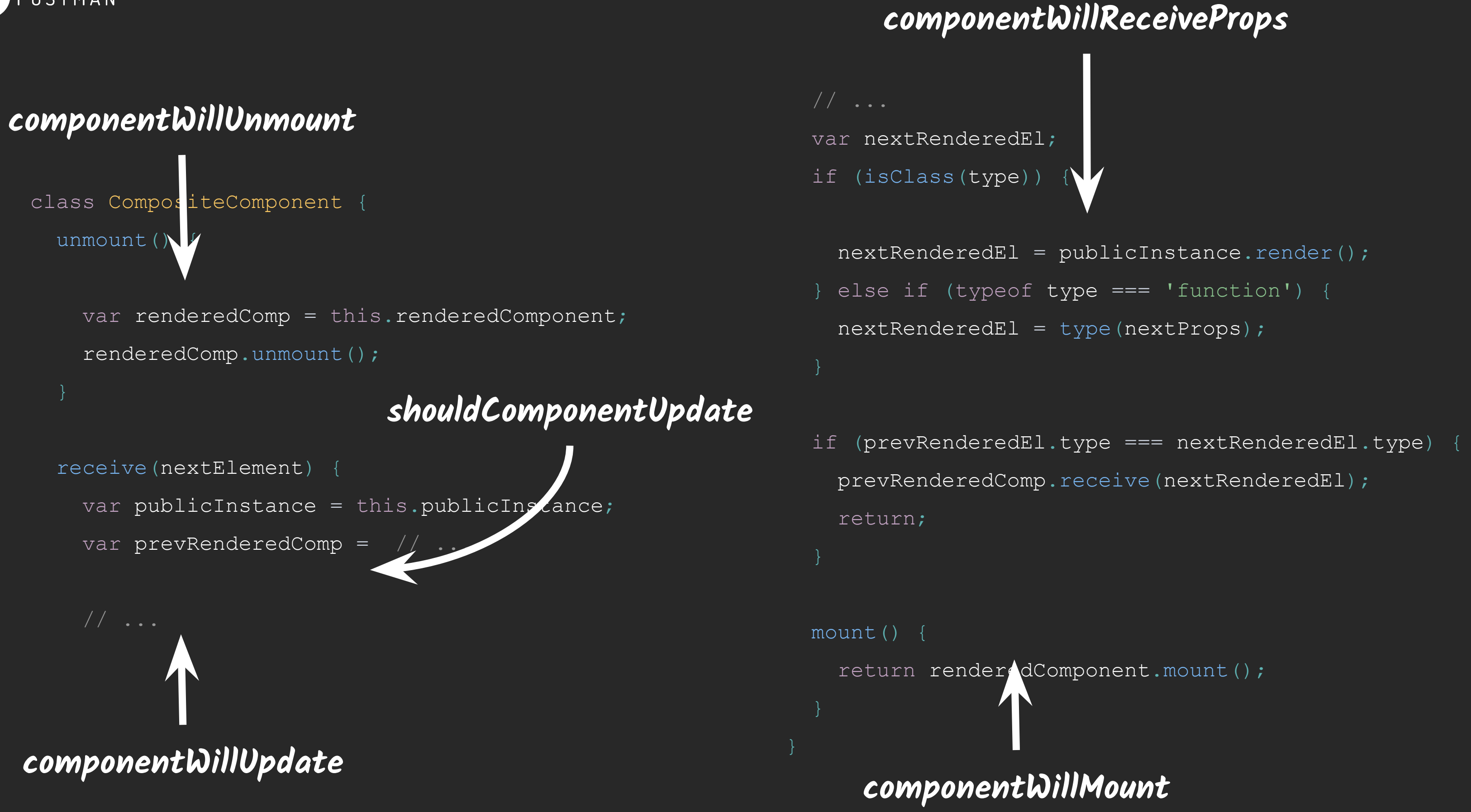
Keys!

- React expects stable keys to identify each component uniquely.
- This allows React to easily differentiate between elements, **especially when the order has changed.**

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

// No problem without keys
<ul>
  <li>Duke</li>
  <li>Villanova</li>
  <li>Connecticut</li>
</ul>

// Inefficient without keys!
<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```



componentWillUnmount

```
class CompositeComponent {
  unmount() {

    var renderedComp = this.renderedComponent;
    renderedComp.unmount();
  }

  receive(nextElement) {
    var publicInstance = this.publicInstance;
    var prevRenderedComp =  // ...

    // ...
```

shouldComponentUpdate

componentWillUpdate

componentWillReceiveProps

```
  // ...
  var nextRenderedEl;
  if (isClass(type)) {

    nextRenderedEl = publicInstance.render();
  } else if (typeof type === 'function') {
    nextRenderedEl = type(nextProps);
  }

  if (prevRenderedEl.type === nextRenderedEl.type) {
    prevRenderedComp.receive(nextRenderedEl);
    return;
  }

  mount() {
    return renderedComponent.mount();
  }
}
```

componentWillMount

Where is VDOM?

- The element tree which React relies on is the **VDOM**!
- Changes to this React element tree are fast as there is no rendering.
- Note: **This is not the same as Shadow DOM**

POSTMAN

Learnings

- Basic algorithmic challenge - diffing trees.
- How context helps reduce complexity.
- Don’t over optimize.

Problems

- Large tree diffing or heavy `render` methods can block the main thread.
- This blocks the main thread and can make the UI unresponsive or janky.

4. Fiber Reconciler

Aim of Fiber

- **60 fps** web applications
- Ability to split **interruptible** work in chunks.
- **Ability to prioritize**, rebase and reuse work in progress.

POSTMAN

Key Ideas

- It is not necessary to update everything immediately
- Different type of updates have different priority
- Pull based approach

Phases

- **Reconciliation / Render phase**: React builds the work in progress tree and finds out the changes it needs to make without flushing them to the renderer. This is **interruptible**.

- **Commit phase:** All the changes are flushed to DOM. This is **uninterruptible**.

What is a fiber?

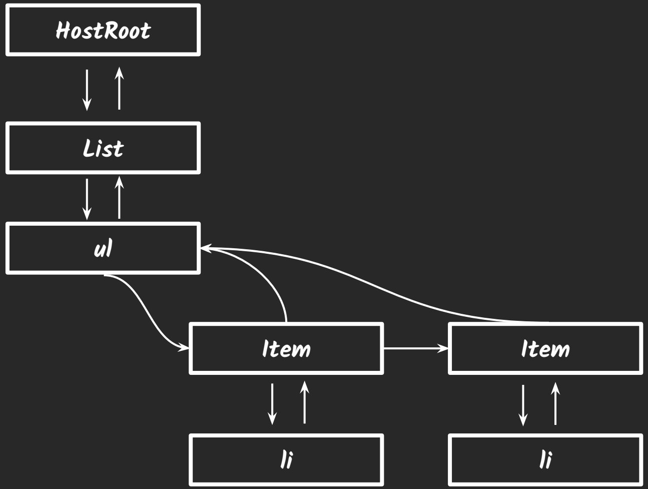
- A fiber represents a **unit of work.**
- A fiber is a **JavaScript object** that contains information about a component, its input, and its output.
- It has a one-to-one relationship with an instance.

```
{
        stateNode,
        child,
        sibling,
        parent
}
```

POSTMAN

Fiber Tree

```
{
  type: List,
  props: {
    children: {
      type: 'ul',
      props: {
        children: [{
          type: Item,
          props: {
            children: {
              type: 'li',
              children: 'list item 1'
            }
          }
        }, {
          type: Item,
          props: {
            children: {
              type: 'li',
              children: 'list item 1'
            }
          }
        }]
      }
    }
  }
}
```



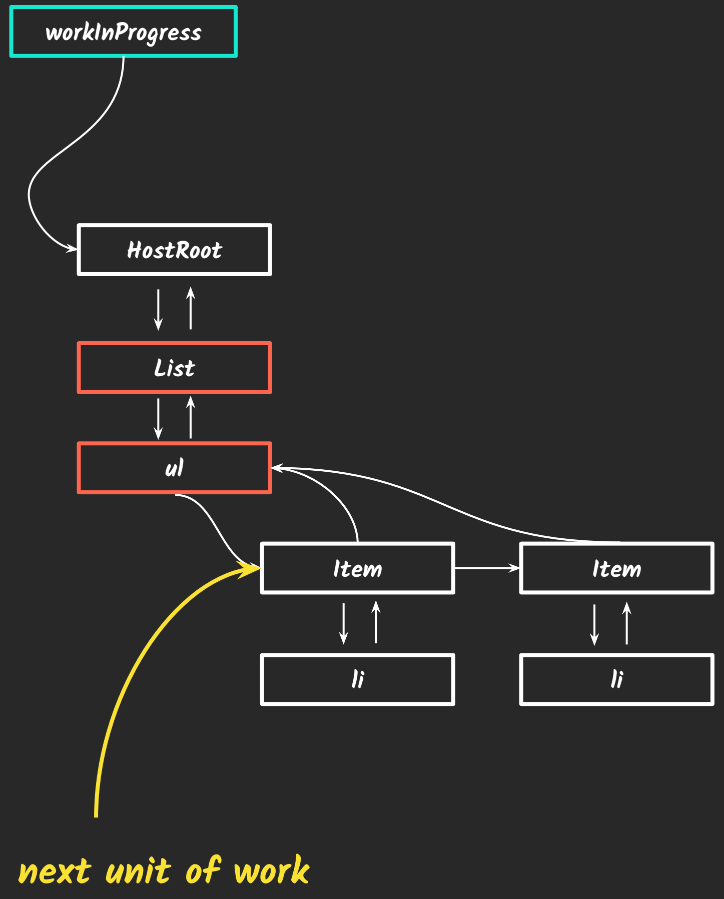
Work Loop

- `requestIdleCallback` - Call when browser is idle with the `timeRemaining`
- `workLoop (timeRemaining, nextUnitOfWork)`
- After time has elapsed, allow main thread to do other work.

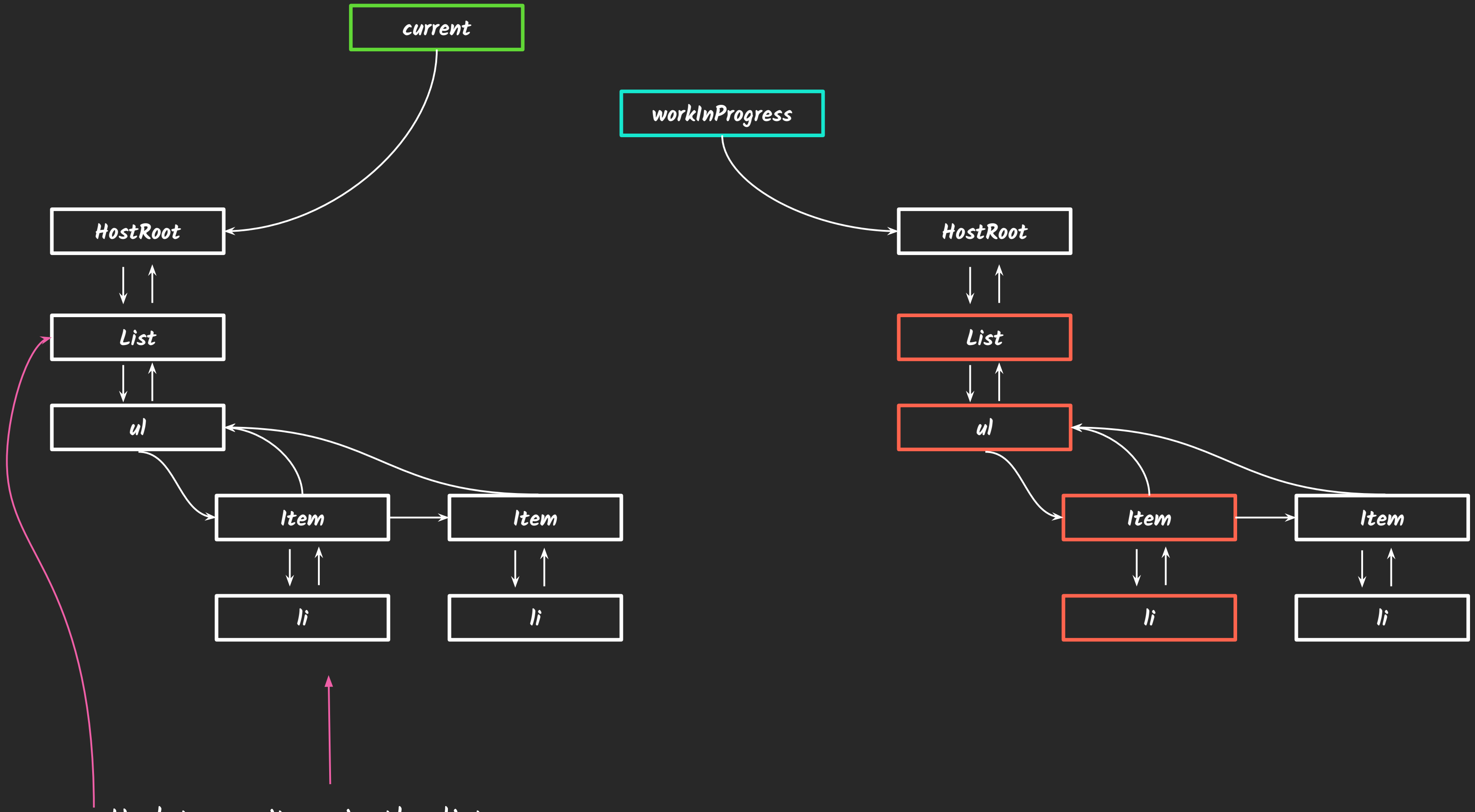
POSTMAN

WIP Tree

- Keeps a track of changes in the current fiber tree.
- Traverses each node, calling *render* lifecycle methods, until leaves reached.
- Not entirely a clone of current tree.



POSTMAN

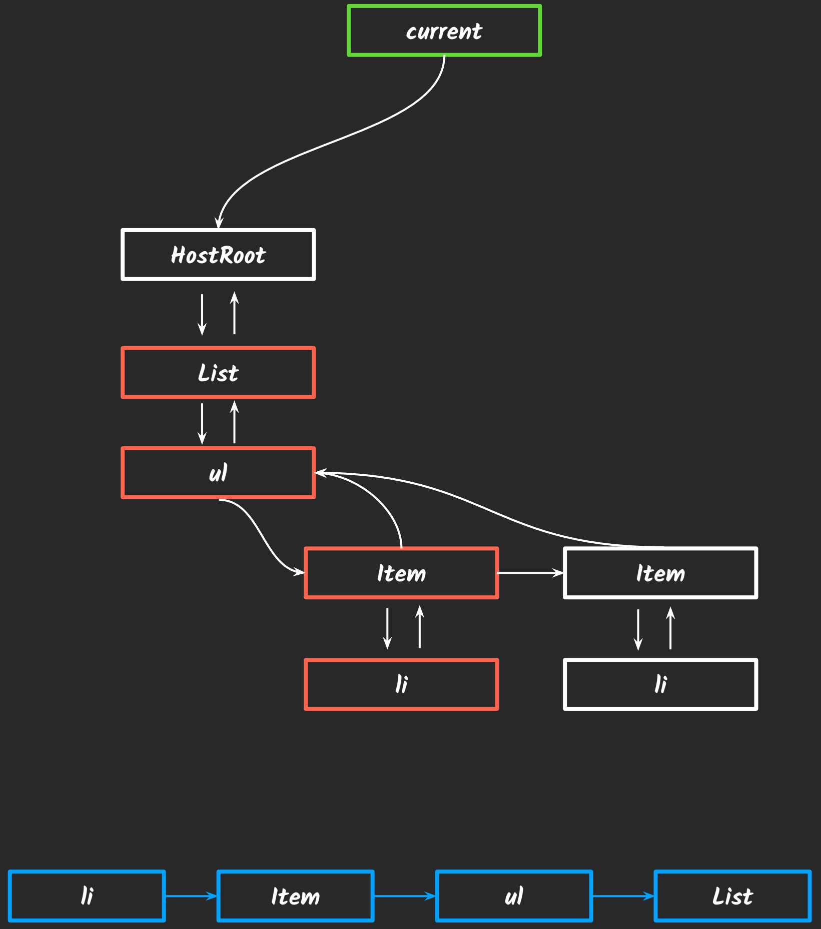


Update an item in the list

POSTMAN

Effect lists

- List of changes to be applied on DOM
- Traverse the list to make changes and call *commit* lifecycle methods.
- After changes are flushed, WIP tree becomes the *current tree*.



“Render Phase”

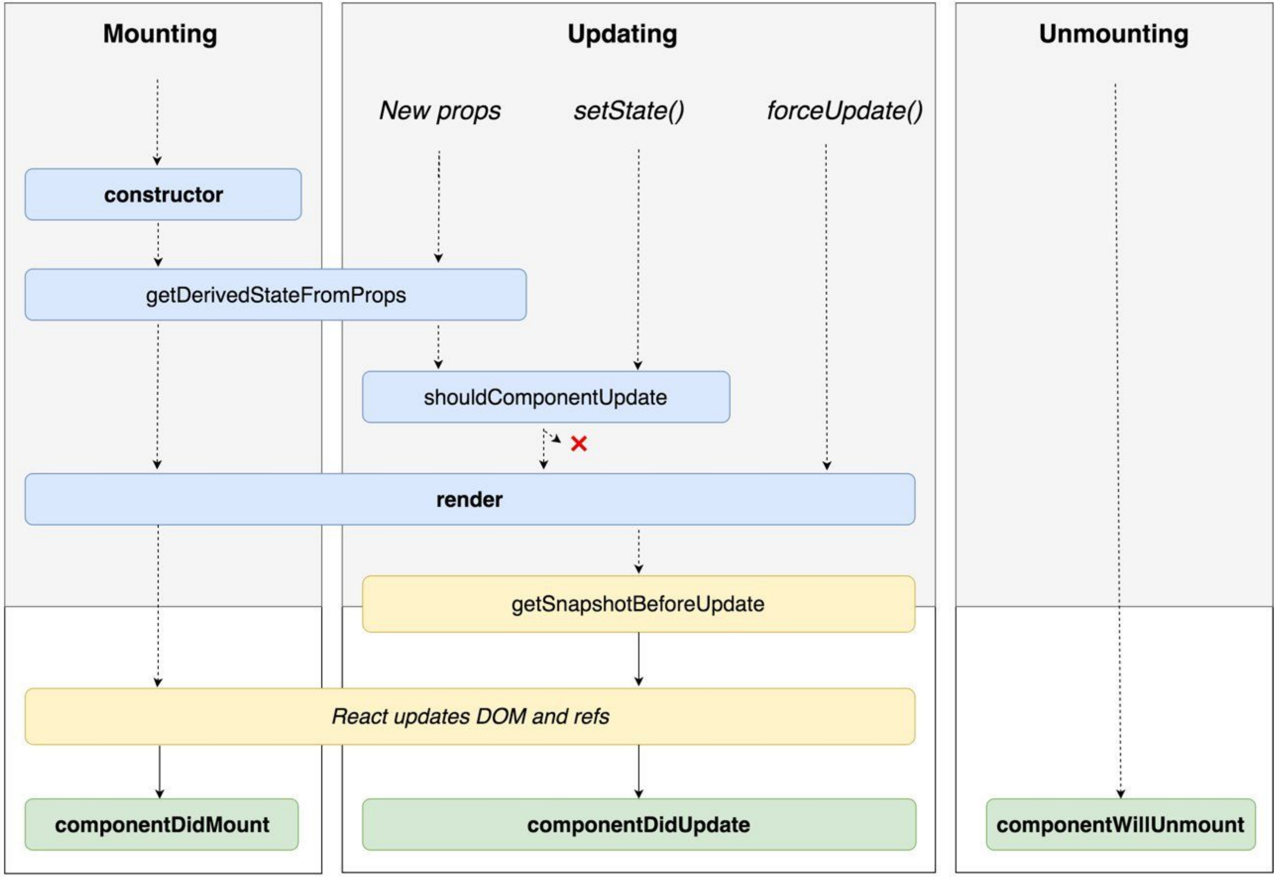
Pure and has no side effects.
May be paused, aborted or
restarted by React.

“Pre-Commit Phase”

Can read the DOM.

“Commit Phase”

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



Mounting

Updating

Unmounting

New props

setState()

forceUpdate()

constructor

getDerivedStateFromProps

shouldComponentUpdate

render

getSnapshotBeforeUpdate

React updates DOM and refs

componentDidMount

componentDidUpdate

componentWillUnmount

https://twitter.com/dan_abramov/status/981712092611989509?lang=en

Priorities

In order to make the UI feel more responsive, React assigns priorities to various changes and schedules them accordingly -

1. **Synchronous** - same as stack reconciler
2. **Task** - before next tick
3. **Animation** - before next frame
4. **High** - pretty soon
5. **Low** - delay is okay
6. **Offscreen** - prepare for scroll

POSTMAN

Learnings

- Concepts from RTOS
- Push vs Pull model
- Using the correct datastructure to suit your needs.

6. Outro

How does this help me?

- Write better code
- Understand errors
- Understand why
- Debug and improve performance

Do you really need React?

- Changes in the DOM do not arise from a change in underlying data.
- Is your project complex enough to justify a **100kB** bundle size jump?
- Does your project involve heavy JavaScript animation?
- Do you need complete control of DOM updates?

What makes it tick?

- Evolutionary method of solving problems.
- Application of computer science concepts?
- **Understanding the wheel you don’t reinvent.**

Thank You

References

- http://www.mattgreer.org/articles/react-internals-part-one-basic-rendering/
- https://bogdan-lyashenko.github.io/Under-the-hood-ReactJS/
- https://www.youtube.com/watch?v=aV1271hd9ew&feature=youtu.be (React Fiber)
- https://www.youtube.com/watch?v=ZCuYPiUIONs (Cartoon intro to fiber)
- https://www.youtube.com/watch?v=crM1iRVGpGQ (Dan Abramov explains Fiber)
- https://reactjs.org/docs/implementation-notes.html (Implementation Notes)
- React Documentation

POSTMAN

Questions