Introduction to ReactJS Basics

This blog covers ReactJS from its inception to advanced use case.

It is assumed that the reader is familiar with the working of a web browser.

If not please familiarise yourself on [how browsers work](https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/) before proceeding.

I shall divide this blog in the following manner:

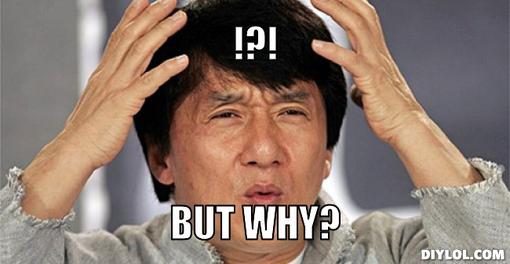
The need for ReactJS

Thinking in components

Writing your first React component

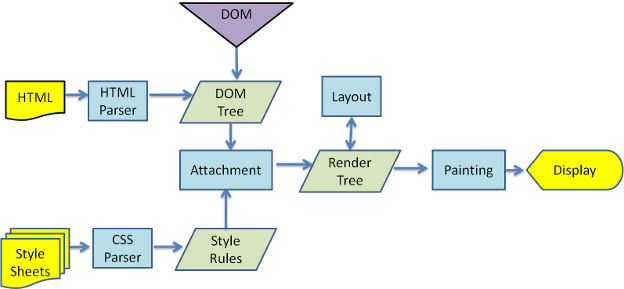
ReactJS Pros and Cons

**The Need for ReactJS**



A Web Application is a program that fetches data, displays data and provides user a way to interact with this data and this program is run on Browsers. While fetching and interacting processes are simple enough, it is the process of displaying the data that takes up a bulk of resources on the Browser.

The DOM Rendering, Layouting and Painting are costly processes when it comes to manipulating the DOM on a large scale.



Traditional MVC approaches would have a model attached to both the business logic and the DOM and would update the DOM when the data attached is changed. These models may be attached to multiple DOM nodes, and on change, these multiple DOM nodes will be updated which goes through the process of Render, Layout and Painting. These DOM node changes are not done in one batch and might end up queueing for resources.

<div ng-bind="boundModel">

<header>

{{boundModel}}

</header>

'

'

</div>

<span ng-bind="boundModel">

'

'

'

'

</span>

<p>

{{boundModel}}

</p>

Fig 1) angularjs models attached to DOM

Let’s take angularjs MVC approach for example. From Fig 1) we observe that the model “boundModel” is attached to a div, span and a p tags of HTML.

When the value of “boundModel” changes in the controller, Angular internally updates each of these DOM nodes with the new value and this may happen over multiple digest cycles.

Moreover the DOM nodes that need updating might be related and redundant.

Now, these changes arrive individually to the DOM tree and are processed individually. Once the DOM tree is updated with the latest change, the DOM node where the change has occurred will call its Layout () method (To update the Dimension and Position of this DOM node on the Browser). This Layout method is recursive. It determines its own dimensions and position and then calls the Layout methods of its children to do the same. After Layout, the DOM node calls its Paint () method (to update the UI). This Paint method is also recursive.

Any subsequent changes to the DOM tree has to wait until these processes are complete.



**Solution:**

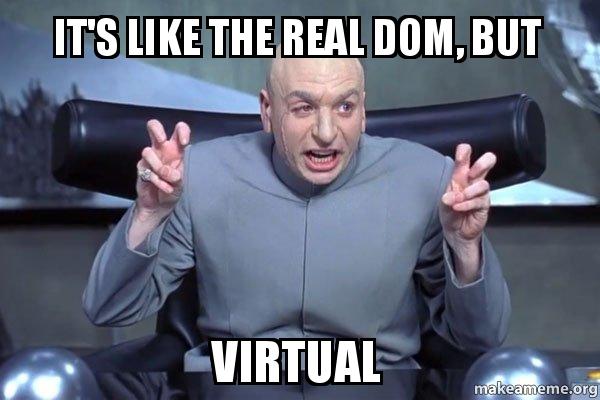
For any change in boundModel, consolidate all the DOM changes that needs to be done into one batch, and do it in one full sweep.

This is achieved thanks to a combo of techniques using Virtual-DOM, Diffing and Patching.

(Author note: This is not the only solution, there is also Incremental DOM and Ember’s Glimmer. Refer [here](https://auth0.com/blog/face-off-virtual-dom-vs-incremental-dom-vs-glimmer/) for more info.)

**Virtual-DOM**

Virtual-DOM is a representation of DOM as a JavaScript object. It holds the tree structure and the node data structure of an actual DOM in-memory.



Diffing

Diffing is a process of comparing two objects and getting their difference.

Patching

Patching is a process of applying a patch onto an object.

How does react fit into this?

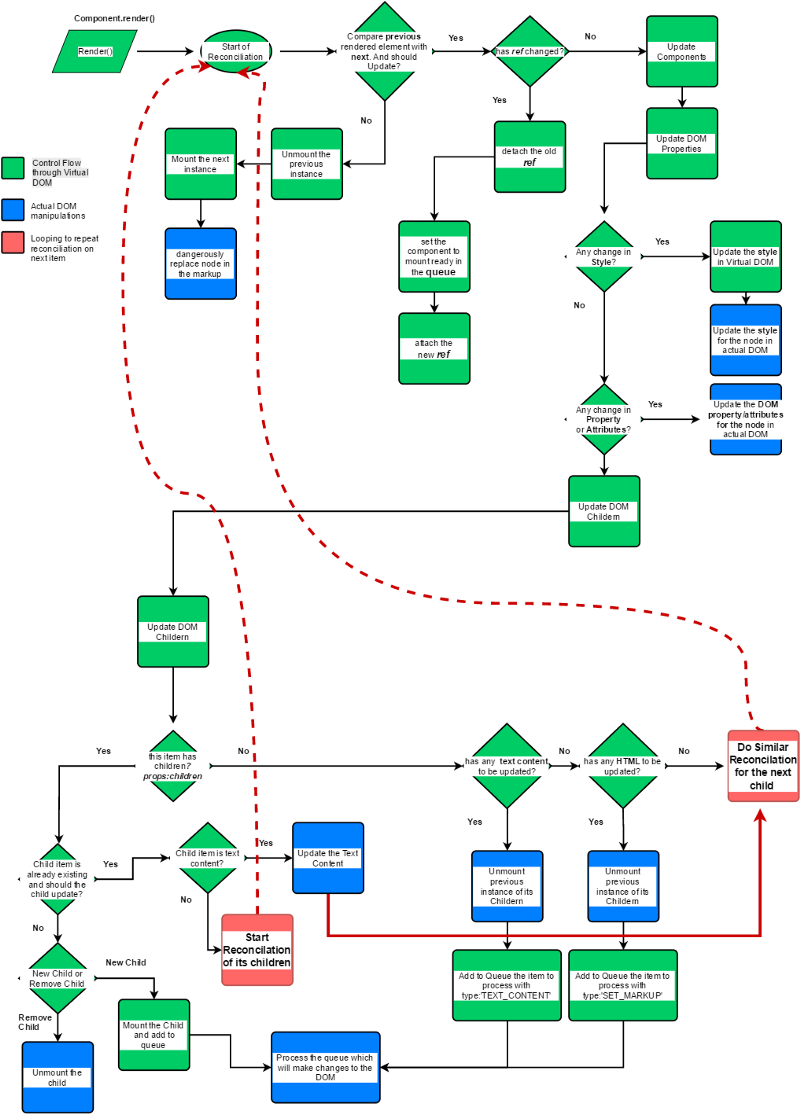
React enforces users to follow Component based architecture. This allows react to create **Observable** models (Angular uses dirty checking). Whenever a model(or state, as it is called in react) is updated, React marks those components that have been affected by this updation as dirty and when it’s time to render, react gets the difference between the virtual-DOM and real DOM and updates this difference to the real DOM in one single batch.

What was once a multiple call to update DOM, can now be done in one.

Isn’t this complex and time consuming?

If you are using react to create a “Hello World” app, then **YES** otherwise **NO**, for a complex app, JavaScript Virtual-DOM operations are faster than direct DOM manipulations.

The process of diffing and patching takes an order of **O(n3)**, but React has reduced that down to **~ O(n)**, thanks to reference keys on DOM nodes.



**Thinking in Components**

A Component is an identifiable part of a larger program. It provides a particular function or group of related functions.



You need to break down your apps into components beforehand. [Here](https://reactjs.org/docs/thinking-in-react.html) is a great article on how to break down your apps into individual components.

Aside from breaking apps into components, you might also need to look into the design pattern of having [Presentational and Container Components](https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0).

**Writing your first React component**

There is no better place to learn the secrets of React coding than the [ReactJS website](https://reactjs.org/docs/getting-started.html).

I have included sample applications in [this repo](https://github.com/maheshkrsna/React_Tutorial/tree/master/Example_Code/React_Todo/react_todo).



**React JS Pros and Cons**

**Pros:**

Componentization.

Virtual DOM

Batched Updates thanks to Virtual DOM

Strong Community Support

Existing products in the market

Good Portfolio!

**Cons:**

Steep learning curve

Poor support for legacy or third party library

Component enforcement

Lack of proper framework (This can be a pro as well)