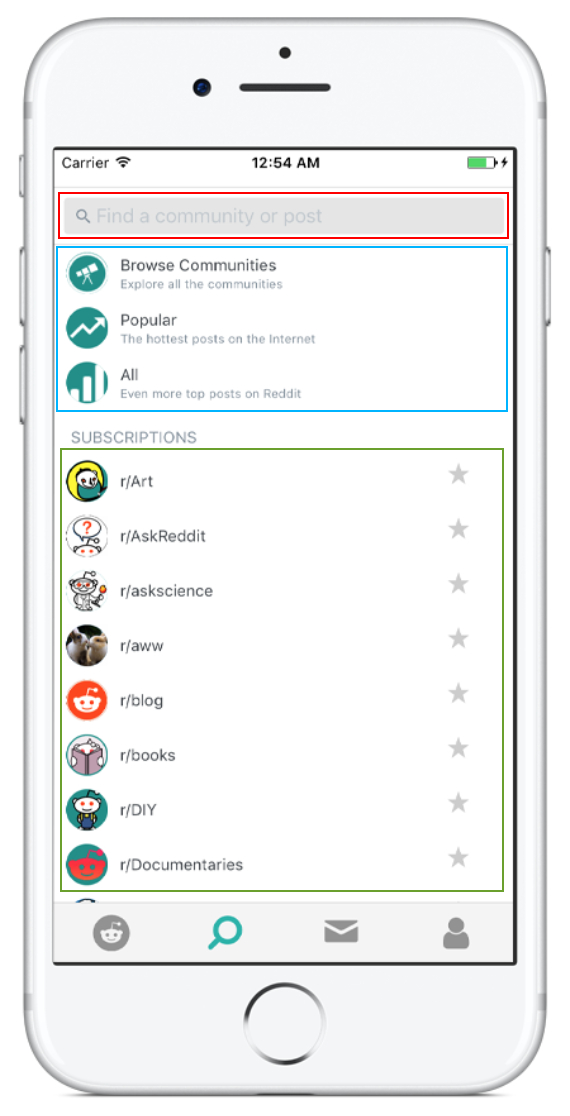
**Component**

Concept cơ bản đằng sau React và React Native đó là việc chia nhỏ app của chúng ta thành các **Component**. Hãy cùng xem giao diện của một màn hình dưới đây (<https://github.com/finetimi/react-native-reddit>)



Có thể nói tất những gì bạn thấy trong ảnh đều là **Component**. Ví dụ phần khoanh vùng màu đỏ là 1 component thể hiện search box. Vùng khoanh màu xanh da trời là tập hợp của 3 component giống nhau, tương tự với phần khoanh màu xanh lá. Thậm chí nếu chúng ta đi sâu hơn nữa thì mỗi component (mỗi dòng) trong vùng khoanh màu xanh lại là tập hợp của những component khác như Image (hiển thị hình ảnh), Text (hiển thị chữ),...

**Component** chính là 1 trong những lí do khiến ứng dụng viết bằng React/React Native có khả năng tái sử dụng và tăng khả năng mở rộng. Đây là điểm mà tôi rất thích ở React, vì nó buộc bạn phải nghĩ đến việc thiết kế và lọc ra những thành phần chung trong một ứng dụng thay vì ném mọi thứ vào 1 cái container view.

Không chỉ có vậy, điểm mạnh thứ 2 liên quan đến **Component** chính là việc React Native cung cấp cho chúng ta rất nhiều các component mặc định, ví dụ như Text. Khi chúng ta build app trên iOS, React Native sẽ đảm nhận việc chuyển component này thành UIView chứa đoạn text, còn khi build trên Android thì sẽ thành TextView native.

Vậy thì chúng ta sẽ xử lý việc khởi tạo và update dữ liệu cho các Component bằng cách nào?

**State Management**

Trong React, chúng ta điều khiển component bằng cách sử dụng props và state.

**Props**

props là viết tắt của Properties. Một điều mà bạn cần phải nhớ khi sử dụng props đó là không bao giờ nên thay đổi giá trị của nó, hay nói cách khác, đây là một dữ liệu **immutable**.

Các component nhận props từ component cha. Bạn không được thay đổi giá trị của props trong các component này mà chỉ được phép đọc giá trị ra thôi. Trong React thì dữ liệu sẽ đi theo một chiều, có nghĩa là từ component cha => các component con.

Bạn có thể tạo ra component sử dụng props. Ý tưởng của props đó là việc trừu tượng hoá các component để có thể sử dụng được ở nhiều chỗ khác nhau trong app. Hãy xem một ví dụ đơn giản sau:

// Parent

export default class ScreenOne extends React.Component {

render () {

return (

<View>

<Heading message={'Custom Heading for Screen One'}/>

</View>

)

}

}

// Child component

export default class Heading extends React.Component {

render () {

return (

<View>

<Text>{this.props.message}</Text>

</View>

)

}

}

Heading.propTypes = {

message: PropTypes.string

}

Heading.defaultProps = {

message: 'Heading One'

}

Trong ví dụ trên thì chúng ta có component Heading, với props là message. Class cha ScreenOne sẽ quyết định giá trị của message, và trong trường hợp trên thì là "Custom Heading for Screen One". Sở dĩ nói component Heading có thể được tái sử dụng ở nhiều nơi là do chúng ta có thể tạo ra một class khác, ScreenTwo chẳng hạn, và thay đổi giá trị của message để dùng lại được component Heading. Các class cha cũng hoàn toàn không cần thay đổi giá trị này nếu không cần thiết, vì bạn có thể set giá trị mặc định cho message (mặc định là "Heading One" trong đoạn code trên).

**State**

state thì hoạt động khác với props. state là dữ liệu nội bộ của một Component, trong khi props là dữ liệu được truyền cho Component. Chính vì vậy chúng ta hoàn toàn có thể thay đổi state, và coi nó là một kiểu dữ liệu **mutable**.

Tuy vậy, hãy nhớ rằng đừng bao giờ thay đổi trực tiếp biến this.state. Thay vào đó hãy dùng hàm setState để cập nhật giá trị. Sở dĩ chúng ta cần dùng hàm này là do nó sẽ kích hoạt việc render lại component và tất cả component con nằm trong nó, còn thay đổi this.state thì không. Còn một vấn đề nữa, đó là setState chạy bất đồng bộ, vậy nên nếu bạn tiến hành đọc ra giá trị state ngay sau khi setState thì chưa chắc giá trị sẽ được update lên mới nhất đâu.

Vậy thì khi nào nên sử dụng state? Bất cứ khi nào dữ liệu trong một component có thể thay đổi (bởi chính nó).

Người dùng tương tác với component là một ví dụ về cách mà state hoạt động. Ấn vào một button, checkbox, thêm text vào form,... là những trường hợp mà chúng ta có thể sử dụng state.

Ví dụ nếu bạn cần phải điền một cái form với nhiều text input, thì mỗi text input sẽ giữ đoạn text mà bạn gõ vào trong state của nó. Nếu bạn gõ một đoạn text khác vào, state của text input đó sẽ thay đổi và kích hoạt việc render lại component, và text mới của bạn sẽ được vẽ lên.

class Form extends React.Component {

constructor (props) {

super(props)

this.state = {

input: ''

}

}

handleChangeInput = (text) => {

this.setState({ input: text })

}

render () {

const { input } = this.state

return (

<View>

<TextInput style={{height: 40, borderColor: 'gray', borderWidth: 1}}

onChangeText={this.handleChangeInput}

value={input}

/>

</View>

)

}

}

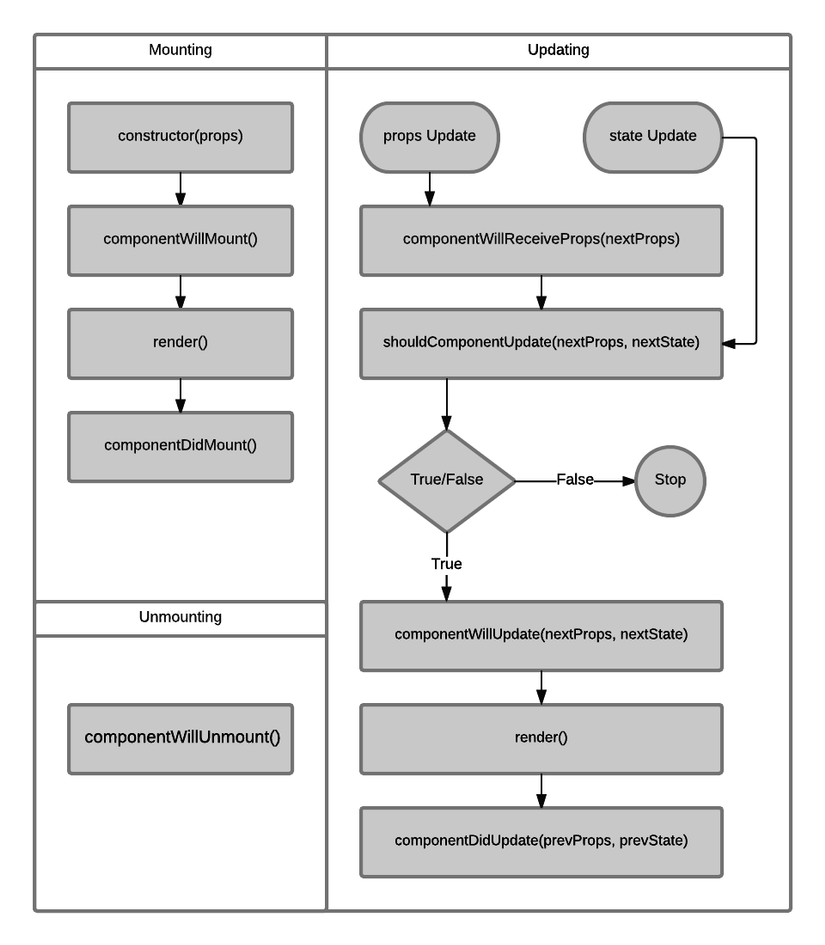
Trong đoạn code trên thì bạn có thể thấy một class Form với state input. Nó có nhiệm vụ render ra một text input để nhận vào thông tin nhập từ user. Một khi người dùng bắt đầu gõ chữ vào, onChangeText sẽ được gọi đến và sau đó nó gọi setState đối với state input.

setState sẽ kích hoạt việc render lại component, và UI sẽ được update với text mới nhất gõ vào bởi người dùng.

Vậy nếu bạn không muốn component render lại khi state thay đổi thì có cách nào không? Đây là lúc chúng ta cần biết về vòng đời của một component.

**Component lifecycle**

Khi tạo ra một **Component** mới, một trong những concept mà chúng ta cần biết đó là vòng đời của nó. Nếu các bạn đã từng code Android hay iOS thì chắc cũng ko lạ gì với vòng đời của Activity hay UIViewController. Những hàm thuộc vòng đời sẽ được system tự động gọi vào thời điểm thích hợp, và chúng ta có thể dựa vào đó để khởi chạy những tiến trình cần thiết. React xử lý việc update DOM thông qua sự thay đổi của state và props. Một component khi bắt đầu được khởi tạo cho đến lúc bị xoá khỏi DOM sẽ trải qua 3 giai đoạn, đó là mounting (gắn), updating (cập nhật) và unmounting (gỡ bỏ). Mounting diễn tả quá trình một component được khởi tạo và vẽ lên trong DOM. Unmounting là khi một component bị xoá khỏi DOM. Giữa mount và unmount thì thường chúng ta sẽ cần cập nhật dữ liệu mới, đó là giai đoạn updating. React cung cấp cho chúng ta một danh sách các hàm tương ứng với từng giai đoạn, các bạn có thể xem ở hình dưới đây:



**Mounting:**

constructor(props)

Hàm khởi tạo này là hàm đầu tiên được gọi khi một Component được tạo ra. Tại đây thường thì chúng ta sẽ khởi tạo các giá trị ban đầu của state, đọc ra giá trị props được truyền vào hoặc bind các hàm của class.

componentWillMount()

Chú ý: Hàm này đã deprecated và chúng ta không nên dùng nó nữa.

Khi hàm này được gọi đến thì component của chúng ta đã được khởi tạo thông qua constructor rồi. Thường thì trong hàm này chúng ta ko nên thay đổi state hãy khởi tạo state trong constructor.

render()

Đây là một hàm bắt buộc phải có trong một component class. Hàm này trả về các React element (thường thì tạo bằng JSX). Tại đây thì nó sẽ phân tích state và props của Component này để vẽ lên DOM. Chúng ta ko nên thay đổi state trong hàm này mà hãy làm ở những chỗ khác.

componentDidMount()

Hàm này được gọi sau khi component đã được khởi tạo và vẽ lên DOM. Nếu bạn cần lấy dữ liệu từ server thì đây là nơi thích hợp để request API.

**Updating**

componentWillReceiveProps(nextProps)

Chú ý: Hàm này sẽ bị deprecated trong tương lai.

Hàm này sẽ được gọi đến khi một component (đã qua giai đoạn mounting) nhận được props mới. Nếu bạn cần update state tương ứng với thay đổi của props thì bạn có thể làm tại đây, bằng việc so sánh this.props (props hiện tại) và nextProps (props mới). Chú ý là nếu bạn gọi setState thì hàm này sẽ không được gọi đâu, nhìn tên hàm là biết.

shouldComponentUpdate()

Hàm này khá đặc biệt bởi nó quyết định việc các hàm khác của vòng đời có được chạy hay không. Mặc định nó sẽ trả về true, và nếu trả về false thì những hàm như componentWillUpdate, render, componentDidUpdate sẽ không được gọi. Về cơ bản, chúng ta không nên thực hiện những tiến trình so sánh quá phức tạp trong này, mà thay vào đó hãy sử dụng PureComponent (tự động perform shallow equal với state và props). Trong tương lai, React sẽ chỉ coi hàm này là một gợi ý cho việc có nên update lại component hay không, nghĩa là kể cả bạn có trả về false nhưng sẽ không đảm bảo là component không được update.

componentWillUpdate(nextProps, nextState)

Hàm này được gọi ngay trước khi render khi nhận được props hoặc state mới. Tuy chúng ta có cả state và props mới tại đây, chúng ta không thể gọi setState trong hàm này, hoặc làm bất cứ cái gì có thể gây ra update cho component. Trong hầu hết trường hợp thì chúng ta có thể thay hàm này bằng componentDidUpdate().

render()

Chúng ta chỉ nhìn thấy thay đổi trong DOM khi hàm này được gọi trong một component. Như đã nói ở trên, một trong những cách để hàm này được gọi đó là sử dụng setState. Render trong giai đoạn này chỉ đc gọi khi shouldComponentUpdate() trả về true, và nếu state hoặc props thay đổi.

componentDidUpdate(prevProps, prevState)

Hàm này được gọi ngay khi DOM được update. Bạn có thể truy cập state và props cũ tại đây, và có thể thực hiện so sánh với state và props hiện tại để tiến hành request API hay làm gì đó khi có thay đổi.

**Unmounting**

componentWillUnmount()

Hàm này sẽ được gọi mỗi khi một component bị xoá khỏi DOM. Đây là lúc để bạn có thể thực hiện những công việc như clear dữ liệu, tắt các connection. Ví dụ nếu bạn có một component chat real time, bạn có thể khởi tạo connection trong componentWillMount và close connection trong componentWillUnmount. Nếu bạn không đóng các connection thì nó sẽ vẫn tồn tại ở đấy và làm chậm ứng dụng.