React Native Components and Styles

CS571 – Mobile Application Development

Maharishi University of Management

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

React Native

A framework that relies on React core.

Allows us build mobile apps using only JavaScript.

Supports iOS and Android.

Learn once, write anywhere

React Native

React Native works a lot like React, implementing JSX, state, and props. Of course, React Native is built on **Native components**, instead of HTML elements.

Unlike React for the web, React Native requires you to import each component in your project - after all, each component is setup to work **both in Android and iOS**.

How does React Native work?

JavaScript is bundled, transpiled and minified.

Unlike the browser, where we run JS and UI in a single thread. A React Native app runs in many threads, so if the JS thread is blocked, the UI thread keeps working and UI is responsive.

The threads communicate **asynchronously** through a **bridge.**

The async communication makes it easy for the JS thread to ask the UI thread to render many items and not wait, when UI finishes or has an event, it is sent back to JS thread in async way.

Threads in React Native App

There are 4 threads in the React Native App:

1. **JavaScript Thread:** is where the logic will run. React App <Button>
2. **Shadow Thread:** This thread is the background thread used by React Native to calculate your layout created using React library. (How the Button is translated into Native)
3. **UI Thread**: Also known as **Main Thread**. This is used for native android or iOS UI rendering. (Native button)
4. **Native Modules Thread**: Sometimes an app needs access to platform API, and this happens as part of native module thread. (A thread for each service: Camera, Geolocation, Sensor.. etc)

React Native Rendering Process

When we first start the app, the main UI thread starts, JS thread starts, and JS bundles are pushed to JS thread.

The UI thread will not suffer at any time as JS thread is doing heavy calculations.

When React starts rendering, it generates a new virtual DOM (layout) it sends changes to Shadow thread, to generates shadow nodes.

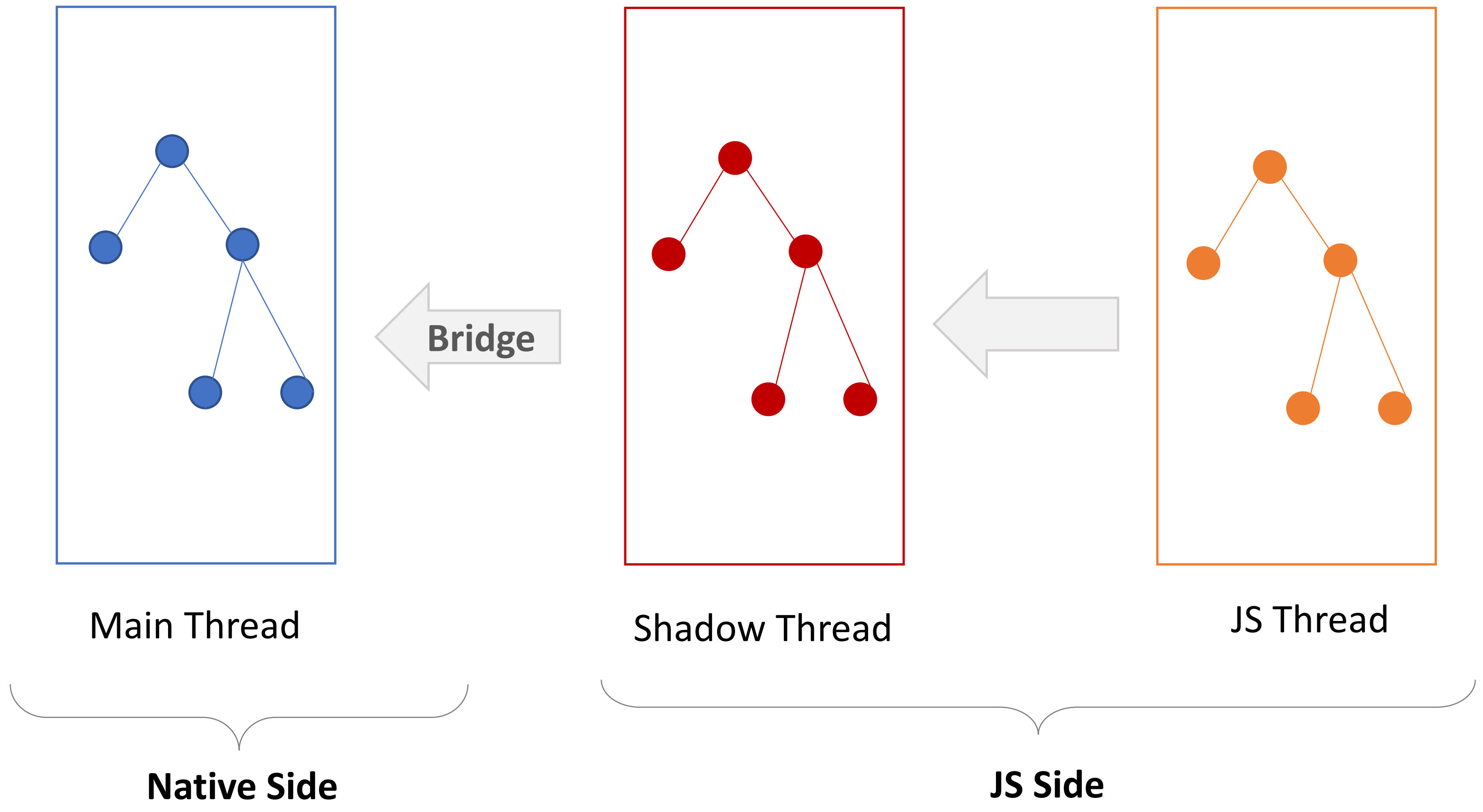
Shadow thread sends generated layout to the main UI thread, and UI renders.

Separation of React Native

Generally, we can separate React Native into 3 parts:

- React Native — JS side
- React Native — Bridge
- React Native — Native side

React Native Threads



Differences between RN and React for Web

All base components are different

The way we style elements (there is no **`className`** in react-native)

No browser APIs, most have been polyfilled (fetch, timers, console..)

Navigation is different (tab, scroll, gestures, transitions.. Etc)

Expo

The fastest way to build an app. Suite of tools to accelerate the React Native development process.

- **Snack** - runs React Native in the browser
- **CLI** - a command-line interface to serve, share, and publish projects
- **Client** - runs your projects on your phone while developing
- **SDK** - bundles and exposes cross-platform libraries and APIs

https://expo.io/tools

React Native Components

Not globally in scope like React web components and we must import them fom **`'react-native'`**

- **`div`** → **`View`**
- **`span`** → **`Text`**

All text must be wrapped by a **`<Text />`** tag

- **`button`** → **`Button`**
- **`ScrollView`**

Button don’t have style, and rendered differently between Android and iOS, instead we will use touchables

Import Components

To import each component, we will simply add it to our imported object:

```
import { Text, View } from "react-native";
```

`<Text></Text>` is a wrapper for any text in your page, it is similar to a **`<p>`** tag in HTML.

`<View></View>` acts very similar to **`<div>`**, it is a container perfect for dividing up and styling your page.

React Native page

```
import React from "react";
import { View, Text } from "react-native";

export default class App extends React.Component {
  render() {
    return (
      <View>
        <Text> Hello MSD! </Text>
      </View>
    );
  }
}
```

Although View does work similarly to a div element, you can’t wrap any text in <View></View>, you must use the <Text> component.

`<Text>` and `<View>`

When a parent **`<Text>`** is wrapping a child **`<Text>`**, the nested text will come out on the same line, assuming there is enough space.

If two **`<Text>`** components were siblings wrapped by a **`<View>`**, they would appear on separate lines.

```
<Text style={{ color: 'red' }}>
  <Text style={{ fontSize: 14 }}>Hello </Text>
  <Text style={{ fontSize: 14 }}>MSD.</Text>
</Text>
```

```
Hello MSD.
```

Notice style inheritance, and discuss the changes of replacing the parent <Text> with a <View>

Style

React Native uses JS objects for styling, object keys are based on CSS properties in camelCase.

Lengths are in **unitless numbers**, abstracted as the app will render in different phones with different pixel density.

`style` attribute may take an array of styles.

`StyleSheet.create()` the same as creating objects for style but with additional optimization, as it only sends IDs over the bridge.

Values are **number** or ‘**text**’. No '10%' is allowed, unlike React for web which accepts percentage.

Everytime we re-render a component <View style={{}} /> a style object is recreated and passed through the bridge to UI thread.

Flexbox

All elements wrapped with **`display: 'flex'`** by default.

Unlike the web, the default **`flexDirection`** is set to **`column`**.

An element with **`flex: 1`** will occupy all space on the phone.

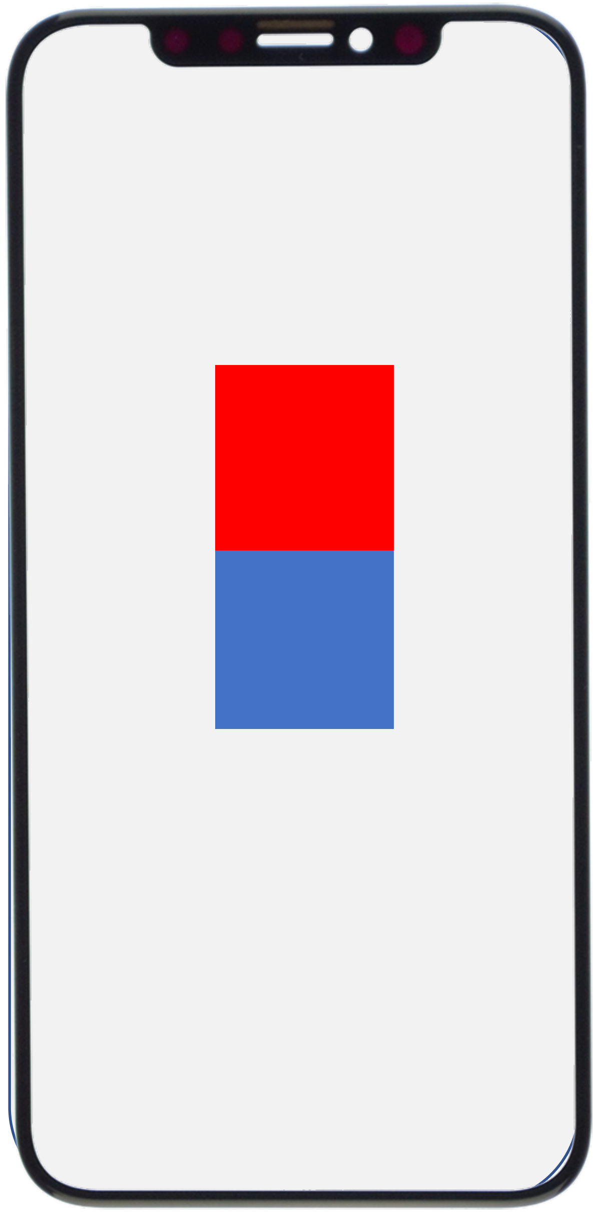
Flexbox offers two main properties **`justifyContent`** and **`alignItems`**.

`justifyContent`	`'center', 'flex-start', 'flex-end', 'space-around', 'space-between'`	How should elements be distributed inside the container. (vertical)
`alignItems`	`'center', 'flex-start', 'flex-end', 'stretched'`	How should elements be distributed inside the container along the secondary axis (opposite of flexDirection, horizontal)

https://reactnative.dev/docs/flexbox

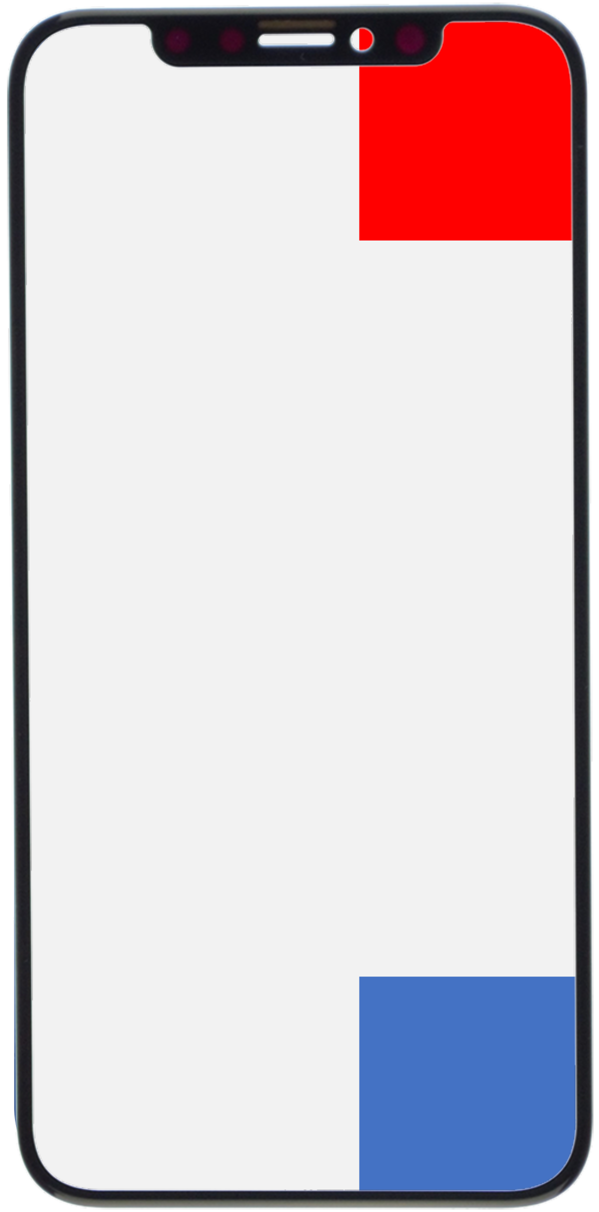
Example

```
container: {
      flexDirection: 'column',
      justifyContent: 'center',
      alignItems: 'center',
      backgroundColor: 'grey',
      flex: 1
},
redbox: {
      width: 100,
      height: 100,
      backgroundColor: 'red'
},
bluebox: {
      width: 100,
      height: 100,
      backgroundColor: 'blue'
}
```



Example

```
container: {
      flexDirection: 'column',
      justifyContent: 'space-between',
      alignItems: 'flex-end',
      backgroundColor: 'grey',
      flex: 1
},
redbox: {
      width: 100,
      height: 100,
      backgroundColor: 'red'
},
bluebox: {
      width: 100,
      height: 100,
      backgroundColor: 'blue'
}
```



```
export default class App extends React.Component {
  constructor(props) {
    super(props)
    this.state = { showCounter: true }
  }

  toggleCounter = () => this.setState({showCounter: !this.state.showCounter})

  render() {
    return (
      <View style={styles.appContainer}>
        <StatusBar barStyle='default' />
        <Button title="toggle" onPress={this.toggleCounter} />
        {this.state.showCounter && <Counter />}
      </View>
    )
  }
}
```

```
class Counter extends React.Component {
  constructor() {
    super()
    this.state = { count: 0 }
  }

  componentDidMount() {
    this.interval = setInterval(this.inc, 1000)
  }

  componentWillUnmount() {
    clearInterval(this.interval)
  }

  inc = () => this.setState({count: count + 1});

  render() {
    return (
      <Text style={styles.count}>{this.state.count} </Text>
    )
  }
}
```

Event Handling

Unlike web, not every component has every interaction.

Only a few touchable components:

- `Button`
- `Touchables*`

Web handlers will receive the event as an argument, but React Native handlers often receive different arguments. (No event obj, no defaultBehaviour)

Read the docs

<View /> does not have events
<Text /> does not have events

`<Button />`

The React Native **`<Button />`** has an **`onPress()`** prop, as opposed to anything **`click`**-related.

It also has a **`title`** prop for the text which goes inside it.

It doesn’t have a **`style`** prop, but only a **`color`** prop.

On the iOS platform, it only shows up as the text in the title - with no background. Therefore, the **`color`** prop will only change the text color, though on Android it will change the background color of the button.

The Touchables

```
<TouchableHighlight onPress={this.pressHandle}>
   <Text>Click</Text>
</TouchableHighlight>
```

The **`TouchableHighlight`** acts like a container for other components with a function when pressed, including a built-in animation. So you could just add **`View`** and **`Text`** components within a Touchable component to act just like a **`<Button />`**, but with customizable style.

Touchable Components

- **`TouchableHighlight`**: when pressed, darkens the background.
- **`TouchableOpacity`**: when pressed, dims the opacity of the button.
- **`TouchableNativeFeedback`**: Android-only ripple effect.
- **`TouchableWithoutFeedback`**: a press without any feedback/effect.

Button Example

```
<Button
  onPress={() => Alert.alert("button pressed!")}
  title="alert button"
  color="blue"
/>
```

We used **`Alert`** which is imported from the React Native core. This will show an alert on top of our screen, similar to a web **`alert()`**.

React Native Component Types

Return a node (something that can be rendered).

Represent a discrete piece of the UI.

- **Stateless** Functional Component (SFC), Pure Functional Component
- **Stateful** React.Component or Functional Component with Hooks.

Stateless Functional Component (SFC)

Simplest component: use when you don’t need state.

A function that takes props and returns a node.

Should be pure (it should not have any side effects like setting values, updating arrays, etc.).

Any change in props will cause the function to be re-invoked.

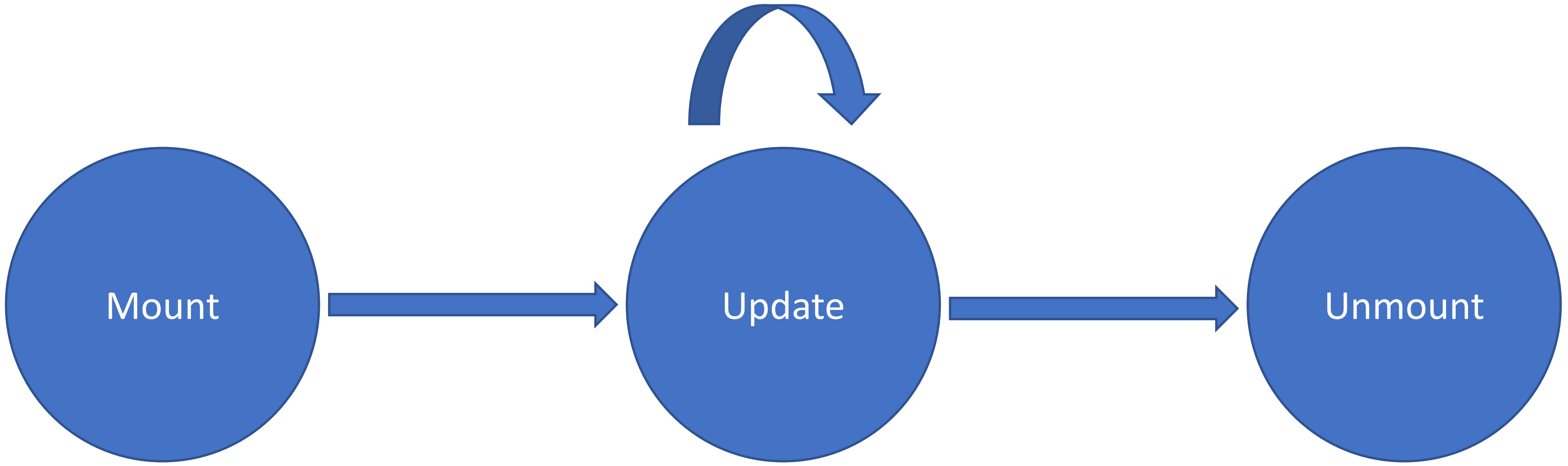
Stateful React Components

These have additional features that SFCs don’t.

Maintain their own state.

Have lifecycle methods (similar to hooks or event handlers) that are automatically invoked.

Component Lifecycle



Mount

1. **`constructor(props)`**: Initialize state or other class properties
2. **`render()`**: Return a node
3. **Send to UI Thread**
4. **`componentDidMount()`**
   - Do anything that isn’t needed for UI (async actions, timers, etc.)
   - Setting state here will cause a re-render before updating the UI

Update

1. **`shouldComponentUpdate(nextProps, nextState):`** Compare changed values, return true if the component should rerender. If returned false, the update cycle terminates.
2. **`render()`**
3. **`componentDidUpdate(prevProps, prevState):`** Do anything that isn’t needed for UI (network requests, etc.)

Unmount

`componentWillUnmount()`

- Clean up
- Remove event listeners
- Invalidate network requests
- Clear timeouts/intervals

Expo CLI

```
npm i expo-cli -g // expo cli tool
expo init projectName // will add expo SDK in node_modules
expo start
```

https://expo.io

Read the Docs

Have a goal in mind

See what the library/framework/API offers

Find something that solves your problem

Configure using the exposed API