Styling in React Native

Visual design is more than simpty using nice colors and grid layouts. A good design can draw the attention of the user to the important information on the page, and form a key element of complex user interactions.

One main difference between Web development and React Native in particular, is that styling on websites is achieved through the use of CSS. Sadly, CSS is not be used, while developing on React Naitve with Expo.

React Native and expo uses a separate JavaScript thread in order to use React code and run that in a Native Environment. As a result, the styling of components is defined in JavaScript too.

The style property we have used so far, is akin to writing inline styles in CSS.

Let us take a deeper dive into styling fo componets in React Native.

StyleSheets

Similar to writing a .css file for styling purposes, React Native with Expo provides a StyleSheet API for writing all our styles separately, in order to maintain the modularity in code.

Consider the example below.

Say we design a Box component as below.

```
const Box = () => (
  <View style={{ width: 100, height: 100, backgroundColor: 'red' }} />
);
```

With the help of the StyleSheet API, we can use the functionality of .create() to create reusable styles for the Box component, thereby making it more modular and readable.

```
const styles = StyleSheet.create({
    box: {
      width: 100,
      height: 100,
      backgroundColor: "red"
    }
  }
);

const Box = () => (
  <View styles={styles.box}/>
)
```

Combining Styles through Styling Arrays

The style property is more powerful than it seems. Till now, we have only been performing static styling i.e. there is no way for styles to change based on user interaction.

Luckily for us, the styles property can accept an array of inline style objects or styleshet references which can help us achieve dynamic styling. This can be though of as analogous to applying multiple CSS classes to an HTML element i.e.

Consider the example below. We want to change the background color to purple when isActive prop is set to true.

```
const box = (props) => {
  return (
    <View style = {
      [styles.box, props.isActive && { backgroundColor: "purple" }]
    }/>
  );
};

const styles = StyleSheet.create({
  box: {
    width: 100,
    height: 100,
    backgroundColor: "red"
  }
});
```

Units of Measurement

While the use of units like pixels or px is quite common in web developemnt, this is not the case while using React Naitive with Expo.

A different unit called Density Independent Pixels or dp is used since it also taked the precision of screen into account. This is important since screens with lower precision (measures in pixels per inch) may make the content in each pixel display larger than it should, if specifying measurements in pixels. This may be thought of as using a relative scale of measurement, as opposed to an absolute scale of precision.

Another commonly used measurement is percentages. This is more frequently used with stlyling properties like widhth and height.

FlexBox

Responsive styling is a very important part of mobile application development since screen sizes may change based on the orientation of the device. As a result flex or flexbox serves as a very important tools for making components responsive. Good news!! All components in React Native are flex-boxes. We do not have to explicilty set display: flex.

By adding the flex property with the factor number, we distribute the avalable space by the factor provided.

Conside the example below. The height of each of the View is calculates by dividing the avaiable space by the sunm of the factors used i.e. height / (1 + 1 + 1) * factor.

```
export const styles = StyleSheet.create({
  layout: {
    flex: 1,
    backgroundColor: '#e5e5e5',
  },
  box: {
    flex: 1,
    backgroundColor: 'black',
  },
});

const App = () => (
  <View style={styles.layout}>
    <View style={[styles.box, { backgroundColor: 'red' }]} />
    <View style={[styles.box, { backgroundColor: 'green' }]} />
    <View style={[styles.box, { backgroundColor: 'blue' }]} />
  </View>
);

export default App;
```

As in CSS, the flex direction may be specified as any of the following

- row renders child from left to right
- row-reverse renders child from right to left
- column renders child from top to bottom
- column-reverse renders child from bottom to top

This may be specified in the flexDirection property of the style object.

In the same example as above, we can set the flex direction as shown below.

```
export const styles = StyleSheet.create({
  layout: {
    flex: 1,
    flexDirection: 'row',
    backgroundColor: '#e5e5e5',
  },
  box: {
    flex: 1,
    backgroundColor: 'black',
  },
});
```

We can also control how the child elements are positioned in the parent flexbox using the justifyContent property. It can take the following values.

- center renders child within center of parent flexbox
- flex-start renders child at the start of the parent flexbox
- flex-end renders child at the end of the parent flexbox
- space-around renders child elements with remaining space around the child elements
- space-between renders child elements with remaining space between the child elements
- space-evenly render children elements with the remaining space evenly divided including space at

the start and at the end.

NOTE: that start and end above art defined with respect to the direction of flex specified in flexDirection.

Considering the same example, say we want space to be evenly divided between the children elements in the flex-box, we can set the "justifyContent" property to "space-around" i.e.

```
export const styles = StyleSheet.create({
  layout: {
    flex: 1,
    flexDirection: 'row',
    backgroundColor: '#e5e5e5',
    justifyContent: "space-around",
  },
  box: {
    flex: 1,
    backgroundColor: 'black',
    justifyContent: "space-around",
  },
});
```