Core Components

In Expo and React Native, components are translated to native components for the platform it's running on. To help you get started, React Native provides a set of ready-to-use components for your app. These components are called "core components" and most of them have built-in Android and iOS implementations. Lets take a dive into some of the core components in React Native.

View Component

With View, you can create responsive layouts using flexbox or add basic styling to nested components.

The component is best comparable with a div HTML element. Just like div, the View component is not visible unless styling is applied. We can apply this styling through the style property.

```
<View style={{ width: 250, backgroundColor: 'yellow' }}>
  ...
</View>
```

Note that Expo and React Native don't use CSS. This method of styling is not available on native platforms. Instead of using CSS, we can write our styling using plain JavaScript objects.

Text Component

Because Expo and React Native translate components to their native counterpart, they aren't exactly similar to plain React. With web, you can render text anywhere in the document without adding parent elements. On native platforms like Android and iOS, that's not possible.

To render text on Android and iOS, the string needs to be wrapped in a Text component. With this component, you can render and style text. It can also inherit styling from a parent Text component, perfect for emphasizing certain words.

```
<Text
 style={
  {
    fontSize: 16
  }
 }
> The <Text style={
          {
            fontWeight: 'bold'
          }
    }> quick brown fox
    </Text>
 jumps over the lazy dog
</Text>
```

Image Component

Images in React Native can be rendered using the Image component. It is similar to the <img> component in HTML, but is much more powerful in functionality. Once such pwoerful feature is it's ability to to load images from different sources. These can be publically acceessible https:// links, or local file:// references, or even impored images through the require() function.

```
<Image style = {
   {
     width: 100,
     height: 100
   }
 }

 source = {
   {
     uri: "https://picsum.photos/100/100"
   }
 }
/>
```

ScrollView Component

View components aren’t scrollable in Expo and React Native. Rendering scrollable content requires additional calculations which could hurt performance when applied to all View components. Expo and React Native have different scrollable components that we can use, like ScrollView. ScrollView allows us to fully manage and customize how the content should be scrolled.

```
<ScrollView horizontal>
   <View style={{ width: 300, height: 300, backgroundColor: 'red' }} />
   <View style={{ width: 300, height: 300, backgroundColor: 'green' }} />
   <View style={{ width: 300, height: 300, backgroundColor: 'blue' }} />
</ScrollView>
```

Button Component

Although the Button is very limited in styling customization, it’s perfect for learning or prototyping purposes. To capture the user clicking the button, we need to use the onPress event handler. This is called onPress because there usually is no mouse available on a mobile device

Lets consider an example where we want a button to update and maintain the number of times it has been clicked, and that it should not be pressed more thatn 3 times.

```
<Button
   title='Press me'
   disabled={pressedCount >= 3}
   onPress={() => {
     setPressedCount(pressedCount + 1)
     }
   }
/>
```

TextInput Component

Comparable to the HTML <Input> component, the TextInput component in React Native is used to capture textual input. To listen for changes on this component, we may use the onChangeText event handler, which receives the input as a string.

NOTE: we also use the TextInput component to accept passwords by using the secureTextEntry property.

```
<TextInput
   style={
    {
      padding: 8,
      backgroundColor: '#f5f5f5'
    }
   }
   onChangeText={text => setName(text)}
/>
```

Combining Core Components

When creating components in ReactNative, the design needs to be broken down into smaller reusable components, however bear in mind that this design process in subjective, and there is no "one size fits all" solution.

Let us consider an example of designing a box component. This box should be serve as a template, and the colow contained the box will be customizable property. To achieve such functionality, we may use the props to pass the color property. One possible design of such a coomponent would be as below.

```
export const Box = (props) => (
  <View style={
    {
      width: 100,
      height: 100,
      backgroundColor: props.color
    }
  }
/>
);
```

As in React, this component may be imported and rendered wherever required. See example below

```
import { Box } from "../components/box";


const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <Box color="red" />
    <Box color="green" />
    <Box color="blue" />
  </View>
);

export const App;
```