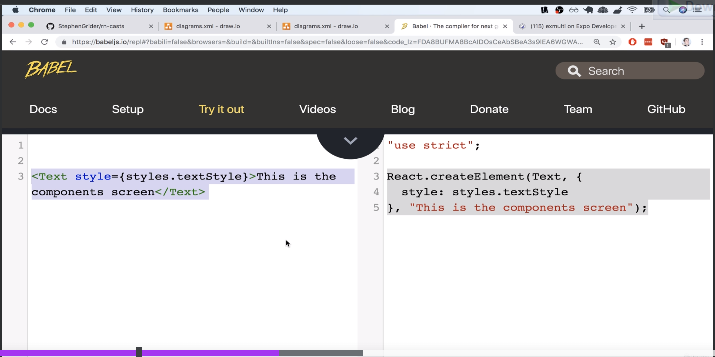
**JSX:**

JSX is converted into JS internally using Babel



We cannot directly print a js object

Arrays can be printed

const FirstComponent = () => {

    const obj = {“name”:”joel”}

    return (

        <View>

            <Text style={style.textStyle}>

                This is a first Reac Native Component

            </Text>

            <Text>Hello {obj}</Text>

        </View>

    );

}

**It will show error**

A jsx element can be assigned and used as well.

const FirstComponent = () => {

    const name = "Lifafa"

    const jsxElement = <Text>This is a const</Text>

    return (

        <View>

            <Text style={style.textStyle}>

                This is a first Reac Native Component

            </Text>

            <Text>Hello {name}</Text>

            {jsxElement}

        </View>

    );

}

This would work just fine.

**FlatList:**

It is better optimized than map

Takes in two properties data, renderItem.

Data – the data that needs to be listed

renderItem – the function to render the items

note that we dereference the “item” directly.

If we don’t then the console.log() for

    return (

        <View>

            <Text>Hello</Text>

            <FlatList data={data}

                renderItem={(item) => {

                    console.log(item)

                }}>

            </FlatList>

        </View>

    );

Would be :

LOG {"index": 0, "item": {"color": "red", "value": "#f00"}, "separators": {"highlight": [Function highlight], "unhighlight": [Function unhighlight], "updateProps": [Function updateProps]}}

LOG {"index": 1, "item": {"color": "green", "value": "#0f0"}, "separators": {"highlight": [Function highlight], "unhighlight": [Function unhighlight], "updateProps": [Function updateProps]}}

LOG {"index": 2, "item": {"color": "blue", "value": "#00f"}, "separators": {"highlight": [Function highlight], "unhighlight": [Function unhighlight], "updateProps": [Function updateProps]}}

LOG {"index": 3, "item": {"color": "cyan", "value": "#0ff"}, "separators": {"highlight": [Function highlight], "unhighlight": [Function unhighlight], "updateProps": [Function updateProps]}}

LOG {"index": 4, "item": {"color": "magenta", "value": "#f0f"}, "separators": {"highlight": [Function highlight], "unhighlight": [Function unhighlight], "updateProps": [Function updateProps]}}

LOG {"index": 5, "item": {"color": "yellow", "value": "#ff0"}, "separators": {"highlight": [Function highlight], "unhighlight": [Function unhighlight], "updateProps": [Function updateProps]}}

LOG {"index": 6, "item": {"color": "black", "value": "#000"}, "separators": {"highlight": [Function highlight], "unhighlight": [Function unhighlight], "updateProps": [Function updateProps]}}

We can do <Text>{data.item.color}</Text> in this case

Or else dereference it as ”item”

It is by default item itself

import React from 'react'

import { Text, StyleSheet, View, FlatList } from 'react-native'

const ListScreen = () => {

    const data = [

        {

            color: "red",

            value: "#f00"

        },

        {

            color: "green",

            value: "#0f0"

        },

        {

            color: "blue",

            value: "#00f"

        },

        {

            color: "cyan",

            value: "#0ff"

        },

        {

            color: "magenta",

            value: "#f0f"

        },

        {

            color: "yellow",

            value: "#ff0"

        },

        {

            color: "black",

            value: "#000"

        }

    ]

    // return (

    //     data.map((d) => {

    //         return (

    //             <View>

    //                 <Text>{d.color}</Text>

    //                 <Text>{d.value}</Text>

    //             </View>

    //         )

    //     })

    // )

    return (

        <View>

            <FlatList

                keyExtractor={(item) => item.value}

                data={data}

                renderItem={({ item }) => {

                    return (<Text>{item.color}</Text>)

                }}></FlatList>

        </View >

    );

}

const style = StyleSheet.create({

})

export default ListScreen;

by using horizontal={true} prop, we can scroll horizontally from left to right in the screen, by default it is vertical.

By default when scrolling we have a scrollbar as a thin white line enaled. To remove this showsHorizontalScrollIndicator={false} prop

**Use of key prop:**

Note that we have to use keyExtrator to give key values.

If we don’t use key, when we delete one obj from an array, it basically recrates the whole array of object to display. When using key it would delete and move the ones below it. Keys are not props and cannot be accessed like props. Keys have to be unique. It helps in optimization.

**Buttons and TouchableOpacity in React-Native:**

We have the primitive <Button> or the advanced <TouchableOpacity> which can detect press on any elements and components.

Button only has onPress

TouchableOpacity has onPress,onBlur,onFocus,onLongPress

**Navigation:**

Note that since we have predefined code from the teacher we have the navigation property in props and it is only because of App.js that has this createStackNavigator

import { createAppContainer } from "react-navigation";

import { createStackNavigator } from "react-navigation-stack";

import HomeScreen from "./src/screens/HomeScreen";

import FirstComponent from "./src/screens/FirstComponent"

import ListScreen from "./src/screens/ListScreen";

const navigator = createStackNavigator(

  {

    Home: HomeScreen,

    First: FirstComponent,

    List: ListScreen

  },

  {

    initialRouteName: "Home",

    defaultNavigationOptions: {

      title: "App",

    },

  }

);

export default createAppContainer(navigator);

and now because of that we are able to use props.navigation.navigate("First")

“First” here refers to the name given in App.js under createStackNavigator.

Use withNavigator(Component) while exporting to use navigation without passing through parent to child and can be used directly.

To pass data use prop.navigation.naviage(“ComponentNameFromApp.js”,{“Id”:1,”name”:”name”}).

Basically as a second parameter.

To get it use props.navigation.getParam(“id”);

const HomeScreen = (props) => {

  console.log();

  return (

    <>

      <Text style={styles.text}>Hello</Text>

      <Button title="Go to First Component"

        onPress={() => props.navigation.navigate("First")} />

      <TouchableOpacity

        onPress={() => props.navigation.navigate("List")}>

        <Text>Go To List Demo</Text>

      </TouchableOpacity>

    </>

  );

};

**Image:**

 <Image source={require(“localpath”)} alt='no img'></Image>

 <Image source={{uri=”from web”}} alt='no img'></Image>

Image screen for multiple images:

import React from 'react'

import ImageView from '../components/ImageView'

import { FlatList } from 'react-native'

const ImageScreen = () => {

    return (

        <>

            <ImageView text="demo1" url={require("../../assets/images/beach.jpg")}></ImageView>

            <ImageView text="demo2" url={require("../../assets/images/forest.jpg")}></ImageView>

        </>

    )

}

export default ImageScreen

Image View as a single reusable component, note that require() is in ImageScreen component

import React from 'react'

import { Image, Text } from 'react-native'

const ImageView = (props) => {

    console.log(props)

    return (

        <>

            <Image source={props.url} alt='no img'></Image>

            <Text>{props.text}</Text>

        </>

    )

}

export default ImageView

**States:**

States are data that on change overtime and causes rerender’s .

Props are data’s that can be passed on to children.

Note that in useState()

Const[name,setName] = useState(“”); is basically Destructuring array directly



Counter app:

Imagine using a variable let counter = 0, and then incrementing and decrementing. This would potentially change the value of the variable but would still display 0 as it does not trigger rerender like a state.

import React, { useState } from 'react'

import { Button, Text } from 'react-native'

const Counter = () => {

    const [value, setValue] = useState(0);

    return (

        <>

            <Text>Count: {value}</Text>

            <Button title="Increase" onPress={() => setValue(value + 1)}></Button>

            <Button title='Decrease' onPress={() => setValue(value - 1)}></Button>

        </>

    )

}

export default Counter

There are two more small apps,

ColorScreen.js

SingleColor.js

**useReduce:**

go through Counter.js, SingleColorScreen.js in the rn-starter/src/screens

**Form:**

Note that we use <TextInput> and also we use **onChangeText** not onChange, we should also note that we directly setName(**e**).