Authors: Samuel Larsson – samla949, William Uddmyr - wilud321

Getting started with React Native

React native is a hybrid approach to building native apps for Android and iOS. This is achieved by combining the JavaScript library React together with native bindings for Android and iOS. A React Native application UI consists of native Android or iOS components, ensuring the best possible performance for the UI. Logic (such as data fetching) is handled by a JavaScript runtime in the application. With React Native it is possible to write the code once and run on both Android and iOS.

Prerequisites

A new React Native project can be created by using the recommended CLI-tools **Expo CLI** or **React Native CLI**, how to use these tools can be found on the official React Native website: https://reactnative.dev/docs/environment-setup

Expo CLI is recommended for new users since it sets up an easy-to-use development environment.

If the project is created with Expo the only requirements is an update version of NodeJS and a physical devices or emulator. React Native CLI also (beyond the requirements for Expo) requires Android-Studio (Windows, Mac, Linux) or XCode (Mac only) to be installed. To build a final release version for iOS the developer needs access to a Mac computer, while for Android this can be achieved using Windows, Mac or Linux. If the application need access to functionality such as the camera or push notifications it is recommended to use a physical device instead of the emulator.

React Basics

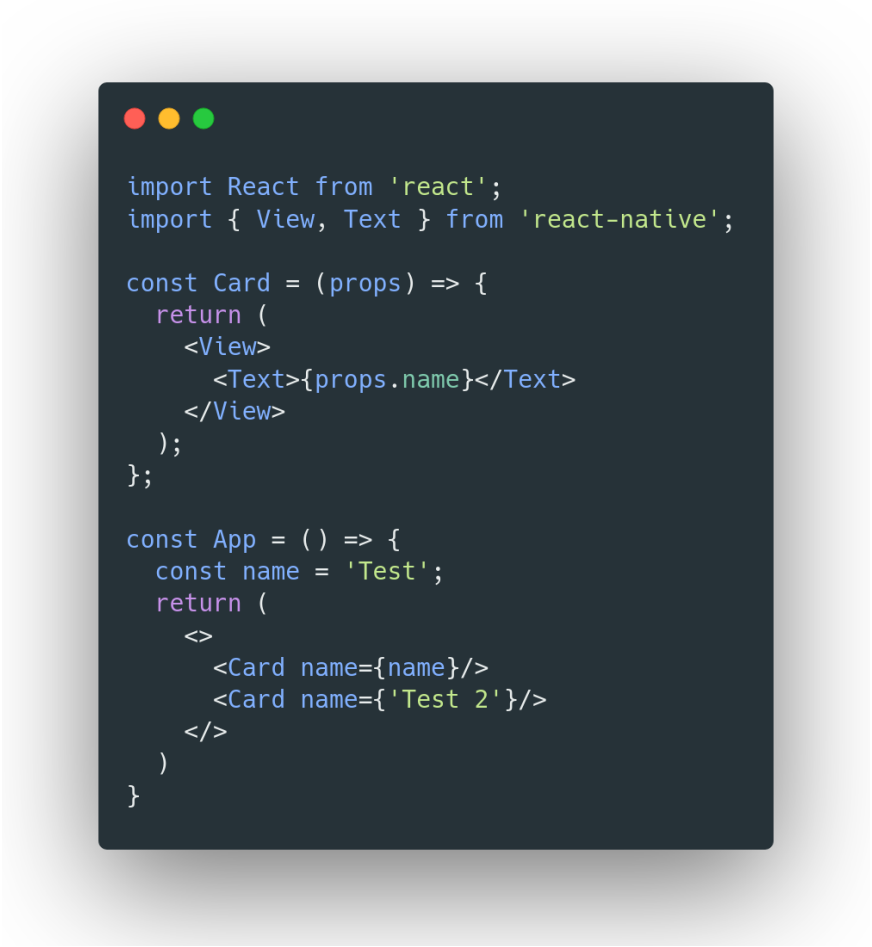
With react the developer builds components; a component could be anything from a button to a whole screen in the application. In modern React there are two ways of building components, class- or functional components. This guide will focus on functional components, these components are also called Hooks.

A hook is a JavaScript function where the return statement contains the content (the components) to be rendered to the screen. In the example below the component *App* is built-up by the two components *Text* and *View*.

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

const App = () => {
  return (
    <View>
      <Text>Test App</Text>
    </View>
  );
};
```

Communication between two components can be achieved with so called *props*, a component sends values to a child component. The example below displays how the component *App* sends a prop called *name* to the *Card* component. This gets messy quite fast if a component needs to send props to a child of a child of a child (or similar), for these cases it is recommended to use more advanced features such as the Context API (https://reactjs.org/docs/context.html).



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

const Card = (props) => {
  return (
    <View>
      <Text>{props.name}</Text>
    </View>
  );
};

const App = () => {
  const name = 'Test';
  return (
    <>
      <Card name={name}/>
      <Card name={'Test 2'}/>
    </>
  )
}
```

An important concept in React is state, what should happen in the application based on certain events. The example below demonstrates the two fundamental React functions *useState* and *useEffect.* The *useState* function returns two values, the current state (*click*) and a function which can be used to set a value (*setClick*). When the *setClick* function is called the value of *click* will be update (the value sent to *setClick*) and this will trigger a new render of the component. This example also illustrates the usage of *useEffect*, a way to react to state changes. The function passed as the first parameter to the *useEffect* function will run every time the values passed in the second parameter are changed. In this case the component will send the value of *click* to an external source every time the value of *click* is changed.

```
import React, { useState, useEffect } from 'react';
import { View, Text, Button } from 'react-native';

const App = () => {
  const [click, setClick] = useState(0);

  useEffect(() => {
    fetch('http://myserver.net/click', {
      method: 'POST',
      body: click
    });
  }, [click]);

  return (
    <View>
      <Text>You have clicked {click} times</Text>
      <Button onPress={() => setClick(click + 1)} title="increment click counter"/>
    </View>
  );
}
```

UI Core Components

As mention earlier React Native translates the standard components to native components in Android and iOS. These standard components are the building blocks of React Native, in the end all components in the application will be composited by these standard components. Below follow a brief description of the most important components and their usage.

View

The simplest building block of them all, used to group other elements and for layout. Very similar to the div-element in HTML5.

Text

Displays text with different formatting on the screen.

Button

A clickable button. If a function is passed to the prop *onPress* it will be executed when the user clicks the button.

TouchableOpacity

A clickable area, same functionality as the Button component but with more styling features. A Button offers very limited styling option, with TouchableOpacity a button can be composited by other components (such as Text or View).

TextInput

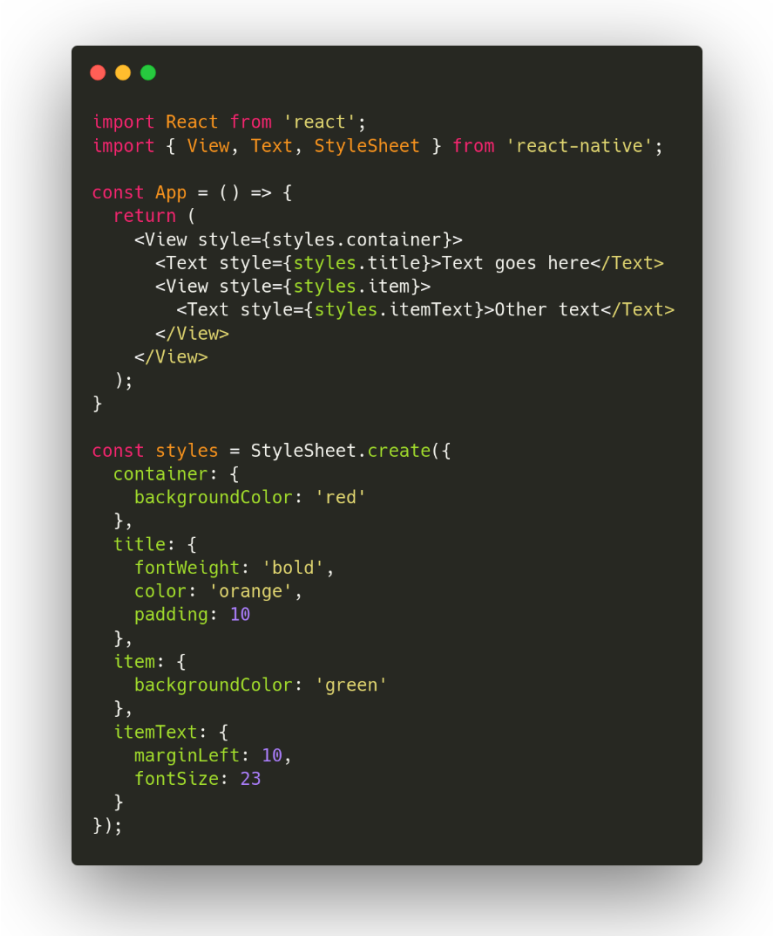
Text input field, when clicked this component will trigger the keyboard and show the inputted text on the screen. The input can be passed to the parent component by using the different provided props.

ScrollView

Wrapper around other elements (like the view), but if the wrapped components overflow the designated area the view will become scrollable.

Layout and Styling

All standard components have a style prop, to this prop values describing the look of a component can be passed. For example, you can change the background color, border radius or the size of the component. The values that can be change is named very similar to CSS (background-color, padding, margin, and so on) and they do almost the same thing, with some minor exceptions. Below is an example on how styling can be applied to the standard components. Styles can be created and applied using the StyleSheet component (see below) or added inline as an object to the style prop.



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

const App = () => {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Text goes here</Text>
      <View style={styles.item}>
        <Text style={styles.itemText}>Other text</Text>
      </View>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    backgroundColor: 'red'
  },
  title: {
    fontWeight: 'bold',
    color: 'orange',
    padding: 10
  },
  item: {
    backgroundColor: 'green'
  },
  itemText: {
    marginLeft: 10,
    fontSize: 23
  }
});
```

Flexbox

In React Native the layout follows the Flexible Box Model (simply called flexbox). Flexbox is used to decide the position, direction, and size of flex-items in a flex-container. A few examples of usable flex properties are further described in the following subsections.

Flex

In React Native, you must define how much of the available space the items will fill. In the figure below, the blue header has flex: 1, green content flex: 10 and yellow footer flex: 0.5. 1 + 10 + 0.5 = 11.5, which means that for example, the content takes up 10/11.5 of the available space.

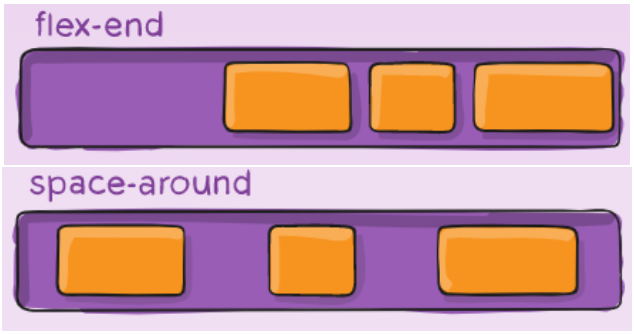
```
header: {
    flex: 1,
    backgroundColor: 'blue'
},
content: {
    flex: 10,
    backgroundColor: 'green'
},
footer: {
    flex: .5,
    backgroundColor: 'yellow'
}
```

Flex-Direction

Flexbox is a single direction layout concept, which means that you can choose the direction to be either row or column (column is default). Note that row-reverse and column-reverse also can be used, in order to define the directions reversed.

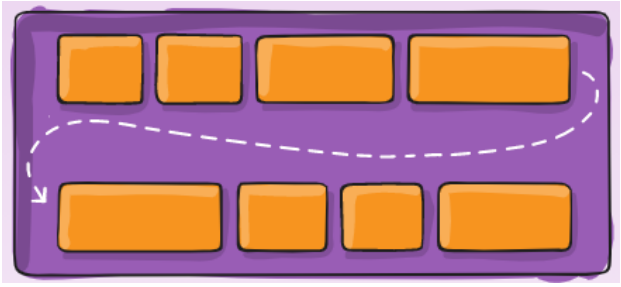
Justify-Content

In order to align content of the container along the main axis, justify-content is used. Content can be aligned in a lot of different ways. For example, to pack items towards the end of the direction or have some space between centered items, flex-end or space-around is used.



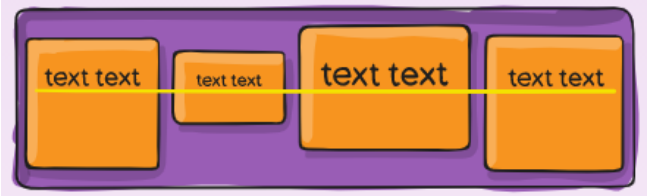
Flex-Wrap

To wrap items inside a container with a fixed height or width, flex-wrap: wrap (nowrap is default) is able to force items to be placed onto multiple rows as in the figure below. The direction of items is depending on the flex-direction.



Align-Items

The align-items property controls the alignment of the items on the cross axis and can be used in a lot of different ways as justify-content. The align-items property is even able to align items at the baseline of the text in items, as shown in the figure below.


text text
text text
text text
text text