Aplikacje mobilne

Komponenty

Mateusz Pawełkiewicz

1) ActivityIndicator

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

const App = () => (
  <View style={[styles.container, styles.horizontal]}>
    <ActivityIndicator />
    <ActivityIndicator size="large" />
    <ActivityIndicator size="small" color="#0000ff" />
    <ActivityIndicator size="large" color="#00ff00" />
  </View>
);

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
  },
  horizontal: {
    flexDirection: 'row',
    justifyContent: 'space-around',
    padding: 10,
  },
});

export default App;
```

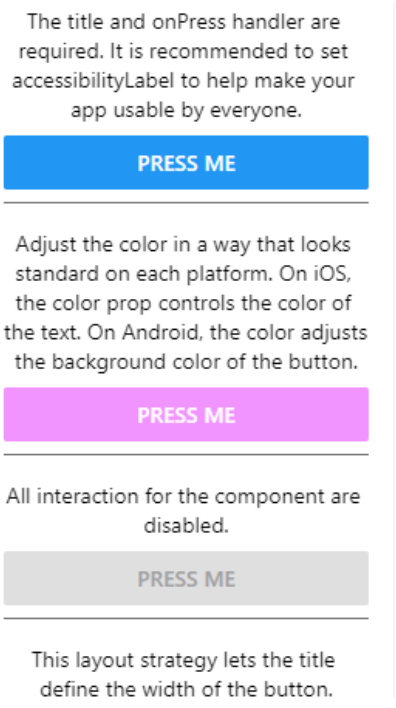
2) Button

Komponent podstawowego przycisku, który powinien ładnie wyglądać na dowolnej platformie. Obsługuje minimalny poziom dostosowywania.

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

const Separator = () => <View style={styles.separator} />;

const App = () => (
  <SafeAreaView style={styles.container}>
    <View>
      <Text style={styles.title}>
        The title and onPress handler are required. It is recommended to
set
        accessibilityLabel to help make your app usable by everyone.
      </Text>
      <Button
        title="Press me"
        onPress={() => Alert.alert('Simple Button pressed')}
      />
    </View>
    <Separator />
```



3) FlatList

Efektywne interfejsy do renderowania podstawowych, płaskich list, obsługujące najbardziej przydatne funkcje:

- W pełni wieloplatformowe.

- Opcjonalny tryb poziomy.

- Konfigurowalne wywołania zwrotna dotyczące widoczności.

- Obsługa nagłówka.

- Obsługa stopki.

- Obsługa separacji (rozdzielających elementów).

- Funkcja "Pull to Refresh" (ciągnij, aby odświeżyć).

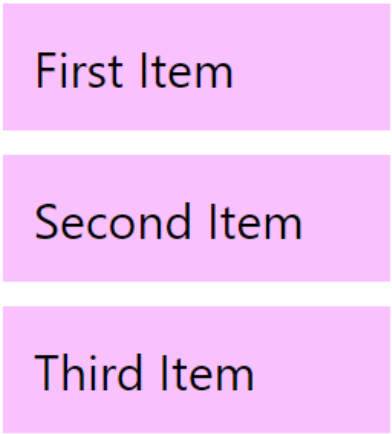
- Ładowanie przy przewijaniu.

- Obsługa ScrollToIndex (przewijanie do indeksu).

- Obsługa wielu kolumn.

```
const App = () => {
  return (
    <SafeAreaView style={styles.container}>
      <FlatList
        data={DATA}
        renderItem={({item}) => <Item title={item.title} />}
        keyExtractor={item => item.id}
      />
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: StatusBar.currentHeight || 0,
  },
  item: {
    backgroundColor: '#f9c2ff',
    padding: 20,
    marginVertical: 8,
    marginHorizontal: 16,
  },
  title: {
    fontSize: 32,
  },
```



4) Image

Komponent React do wyświetlania różnych rodzajów obrazów, w tym obrazów z sieci, zasobów statycznych, tymczasowych obrazów lokalnych i obrazów z lokalnego dysku, takich jak rolka aparatu.

Ten przykład pokazuje pobieranie i wyświetlanie obrazu z lokalnego magazynu, a także obrazu z sieci oraz z danych dostarczonych w schemacie URI 'data:'.

```
const DisplayAnImage = () => {
  return (
    <View style={styles.container}>
      <Image
        style={styles.tinyLogo}
        source={require('@expo/snack-static/react-native-logo.png')}
      />
      <Image
        style={styles.tinyLogo}
        source={{
          uri: 'https://reactnative.dev/img/tiny_logo.png',
        }}
      />
      <Image
        style={styles.logo}
        source={{
          uri:
'
FWHRTb2Z0d2FyZQBwbmdjcnVzaEB1SfMAAABQSURBVGje7dSxCQBACARB+2/ab8WEeQNhFi6WS
YzYLYudDQYGBgYGBgYGBgYGBgZmcvDqYGBgmhivGQYGBgYGBgYGBgYGBgbmQw+P/eM
rC5UTVAAAAABJRU5ErkJggg==',
        }}
      />
    </View>
  );
};
```

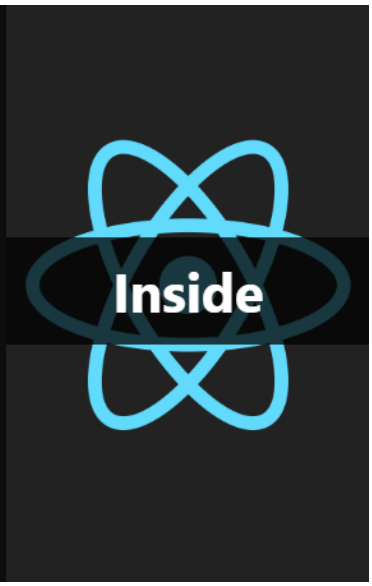
5) ImageBackground

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

const image = {uri: 'https://legacy.reactjs.org/logo-og.png'};

const App = () => (
  <View style={styles.container}>
    <ImageBackground source={image} resizeMode="cover" style=
{styles.image}>
      <Text style={styles.text}>Inside</Text>
    </ImageBackground>
  </View>
);

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  image: {
    flex: 1,
    justifyContent: 'center',
  },
  text: {
    color: 'white',
    fontSize: 42,
    lineHeight: 84,
```



6) Pressable

Komponent "Pressable" jest opakowaniem dla podstawowego komponentu (Core Component), które może wykrywać różne etapy interakcji dotyczącej naciśnięć na dowolne zdefiniowane dzieci komponentu.

```
<Pressable onPress={onPressFunction}>
  <Text>I'm pressable!</Text>
</Pressable>
```

Jak działa komponent "Pressable":

Na elemencie zawiniętym w komponent "Pressable":

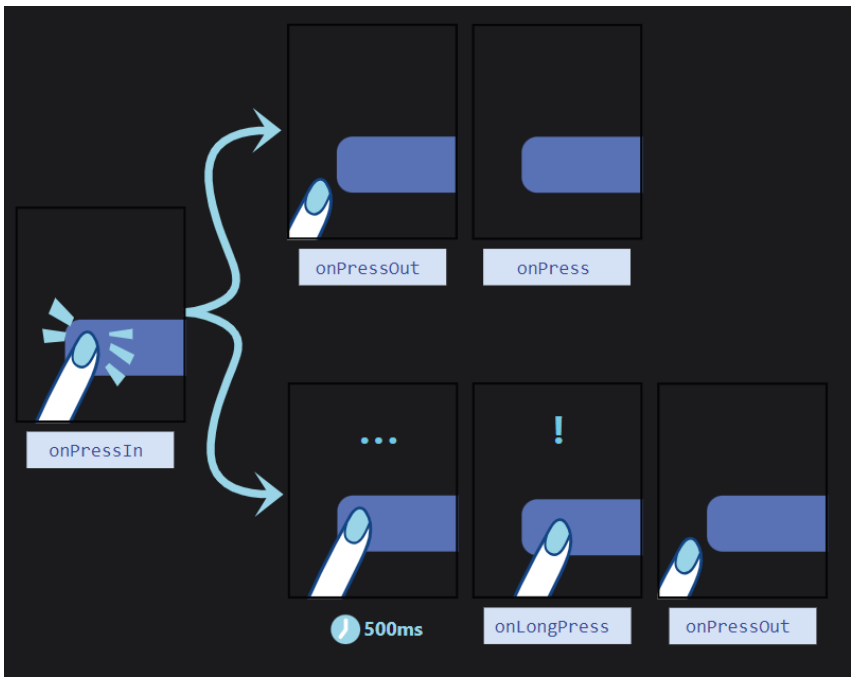
- onPressIn jest wywoływane, gdy naciśnięcie zostaje aktywowane.

- onPressOut jest wywoływane, gdy gest naciśnięcia zostaje dezaktywowany.

Po naciśnięciu onPressIn, może wystąpić jedno z dwóch zdarzeń:

1. Jeśli osoba zdejmie palec z kontrolki, wówczas spowoduje to wywołanie onPressOut, a następnie onPress.

2. Jeśli osoba pozostawi palec dłużej niż 500 milisekund przed jego zdjęciem, zostanie wywołane zdarzenie onLongPress. (onPressOut nadal zostanie wywołane po zdjęciu palca.)



7) ScrollView

Ten komponent owija platformowy ScrollView i zapewnia integrację z systemem blokowania dotykowego "responder".

Pamiętaj, że ScrollView musi mieć ograniczoną wysokość, aby działać, ponieważ zawiera dzieci o nieograniczonej wysokości w ograniczonym kontenerze (za pośrednictwem interakcji przewijania). Aby ograniczyć wysokość ScrollView, albo ustaw wysokość widoku bezpośrednio (co jest odradzane), albo upewnij się, że wszystkie widoki nadrzędne mają ograniczoną wysokość. Zapomnienie o przekazywaniu {flex: 1} w dół stosu widoków może prowadzić tutaj do błędów, które można szybko zdiagnozować za pomocą inspektora elementów.

Ten komponent jeszcze nie obsługuje blokowania innych komponentów responders, które mogą blokować ScrollView przed zostaniem responderem.

<ScrollView> vs. <FlatList> - który z nich używać?

ScrollView renderuje wszystkie swoje komponenty potomne na raz, ale ma to negatywny wpływ na wydajność.

Wyobraź sobie, że masz bardzo długą listę elementów, które chcesz wyświetlić, być może na kilka ekranów treści. Tworzenie komponentów JS i widoków natywnych dla wszystkiego na raz, z czego wiele może nawet nie być widoczne, przyczyni się do wolnego renderowania i zwiększonego zużycia pamięci.

Właśnie wtedy przydaje się FlatList. FlatList renderuje elementy leniwie, gdy są one właśnie odkrywane, i usuwa elementy, które przewijają się daleko poza ekran, aby oszczędzić pamięć i czas przetwarzania.

FlatList jest również przydatny, jeśli chcesz renderować separatory między elementami, wiele kolumn, ładowanie nieskończone przy przewijaniu lub dowolną inną funkcję, którą obsługuje out-of-the-box.

8) Switch

To jest kontrolowany komponent, który wymaga funkcji zwrotnej `onValueChange`, która aktualizuje właściwość `value`, aby komponent odzwierciedlał działania użytkownika. Jeśli właściwość `value` nie zostanie zaktualizowana, komponent będzie nadal renderować dostarczoną właściwość `value`, zamiast oczekiwanego wyniku działania użytkownika.

9) Text

Komponent React do wyświetlania tekstu.

Text obsługuje zagnieżdżanie, stylizację i obsługę dotyku.

W poniższym przykładzie zagnieżdżony tytuł i tekst będą dziedziczyć właściwość fontFamily ze stylu styles.baseText, ale tytuł dostarcza swoje dodatkowe style. Tytuł i tekst zostaną ułożone jeden na drugim ze względu na dosłowne znaki nowej linii:

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

const TextInANest = () => {
  const [titleText, setTitleText] = useState("Bird's Nest");
  const bodyText = 'This is not really a bird nest.';

  const onPressTitle = () => {
    setTitleText("Bird's Nest [pressed]");
  };

  return (
    <Text style={styles.baseText}>
      <Text style={styles.titleText} onPress={onPressTitle}>
        {titleText}
        {'\n'}
        {'\n'}
      </Text>
      <Text numberOfLines={5}>{bodyText}</Text>
    </Text>
  );
};

const styles = StyleSheet.create({
  baseText: {
    fontFamily: 'Cochin',
```

Bird's Nest

This is not really a bird nest.

Zagnieżdżony tekst

Zarówno Android, jak i iOS pozwalają na wyświetlanie sformatowanego tekstu poprzez oznaczanie zakresów ciągu znaków określonym formatowaniem, takim jak tekst pogrubiony lub kolorowy (NSAttributedString na iOS, SpannableString na Androidzie). W praktyce jest to bardzo żmudne. W przypadku React Native zdecydowaliśmy się na użycie paradygmatu stosowanego w sieci, w którym można zagnieżdżać tekst, aby osiągnąć ten sam efekt.

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

const BoldAndBeautiful = () => {
  return (
    <Text style={styles.baseText}>
      I am bold
      <Text style={styles.innerText}> and red</Text>
    </Text>
  );
};

const styles = StyleSheet.create({
  baseText: {
    fontWeight: 'bold',
  },
  innerText: {
    color: 'red',
  },
});

export default BoldAndBeautiful;
```

I am bold and red

10) TextInput

Podstawowy komponent do wprowadzania tekstu do aplikacji za pomocą klawiatury. Właściwości (props) umożliwiają konfigurację wielu funkcji, takich jak automatyczna korekta, automatyczna kapitalizacja, tekst zastępczy (placeholder) i różne rodzaje klawiatury, takie jak klawiatura numeryczna.

Najbardziej podstawowym przypadkiem użycia jest umieszczenie komponentu TextInput i zasubskrybowanie zdarzenia onChangeText, aby odczytać wprowadzany przez użytkownika tekst. Istnieją również inne zdarzenia, takie jak onSubmitEditing i onFocus, do których można się zasubskrybować. Oto minimalny przykład:

```jsx
import React from 'react';
import {SafeAreaView, StyleSheet, TextInput} from 'react-native';

const TextInputExample = () => {
  const [text, onChangeText] = React.useState('Useless Text');
  const [number, onChangeNumber] = React.useState('');

  return (
    <SafeAreaView>
      <TextInput
        style={styles.input}
        onChangeText={onChangeText}
        value={text}
      />
      <TextInput
        style={styles.input}
        onChangeText={onChangeNumber}
        value={number}
        placeholder="useless placeholder"
        keyboardType="numeric"
      />
    </SafeAreaView>
  );
};
```

Useless Text

useless placeholder

11) View

Najbardziej fundamentalny komponent do budowy interfejsu użytkownika, View, jest kontenerem obsługującym układ z użyciem flexbox, stylizację, pewne obsługi dotyku i kontrole dostępności. View jest mapowany bezpośrednio na odpowiednik natywnego widoku na platformie, na której działa React Native, czy to będzie UIView, , android.view itp.

View jest przeznaczony do zagnieżdżania w innych widokach i może mieć od 0 do wielu dzieci dowolnego typu.

Poniższy przykład tworzy widok (View), który zawiera dwie skrzynki z kolorami i komponent tekstowy w układzie rzędu (row) z marginesem (padding):

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

const ViewBoxesWithColorAndText = () => {
  return (
    <View
      style={{
        flexDirection: 'row',
        height: 100,
        padding: 20,
      }}>
      <View style={{backgroundColor: 'blue', flex: 0.3}} />
      <View style={{backgroundColor: 'red', flex: 0.5}} />
      <Text>Hello World!</Text>
    </View>
  );
};

export default ViewBoxesWithColorAndText;
```

12) VirtualizedList

To jest podstawowa implementacja bardziej wygodnych komponentów <FlatList> i <SectionList>, które są również lepiej udokumentowane. Ogólnie rzecz biorąc, ta podstawowa implementacja powinna być używana tylko wtedy, gdy potrzebujesz większej elastyczności niż oferuje FlatList, na przykład w przypadku korzystania z danych niemutowalnych zamiast prostych tablic.

Wirtualizacja znacznie poprawia zużycie pamięci i wydajność dużych list, utrzymując skończony obszar renderowania aktywnych elementów i zastępując wszystkie elementy poza tym obszarem odpowiednio wymiarowanymi pustymi przestrzeniami. Obszar ten dostosowuje się do zachowania przewijania, a elementy są renderowane stopniowo z niskim priorytetem (po zakończeniu wszelkich bieżących interakcji), jeśli są daleko od obszaru widocznego, lub z wysokim priorytetem w przeciwnym razie, aby zminimalizować możliwość pojawienia się pustych przestrzeni.