Разработка мобильных приложений

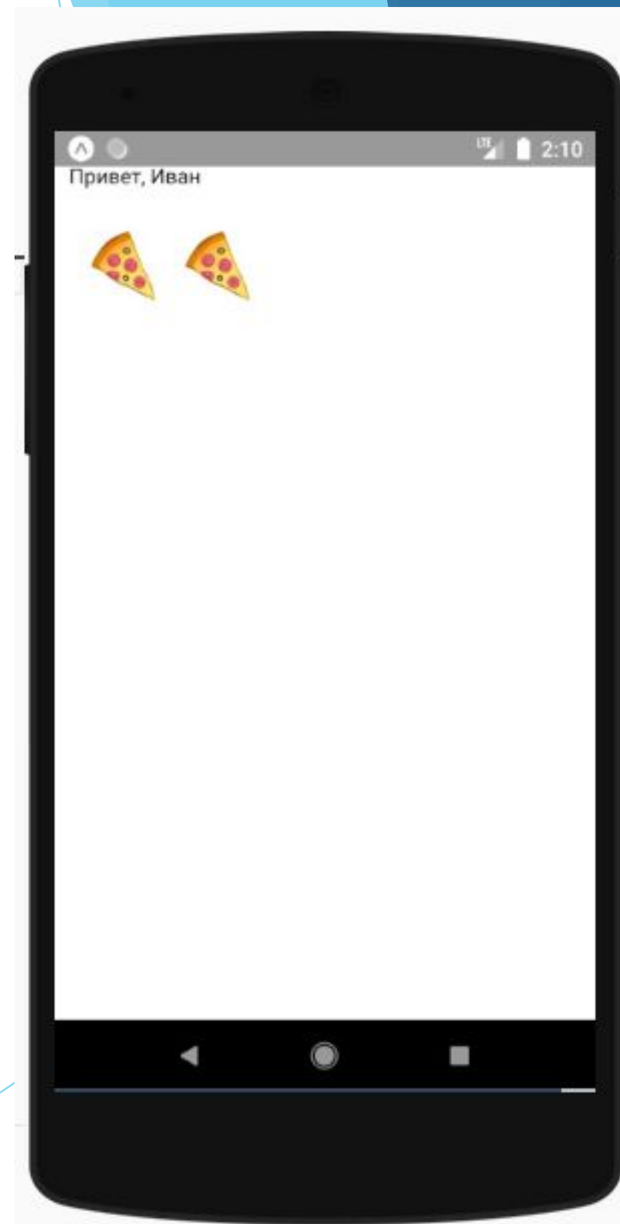
Лекция 11

Обработка ввода текста

- TextInput один из основных компонентов, который позволяет пользователю вводить текст. Он имеет свойство onChangeText, который принимает функцию, вызываемую при каждом изменении текста, и свойство onSubmitEditing, который принимает функцию, вызываемую при отправке текста.

  Например, предположим, что когда пользователь печатает, мы переводим его слова на другой язык. На этом новом языке каждое слово пишется одинаково: 🍕. Таким образом, предложение «Привет, Иван» будет переведено как «🍕🍕».

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

export default class PizzaTranslator extends Component {
  constructor(props) {
    super(props);
    this.state = {text: ''};
  }

  render() {
    return (
      <View style={{padding: 10}}>
        <TextInput
          style={{height: 40}}
          placeholder="Напишите сюда то, что хотите перевести!"
          onChangeText={(text) => this.setState({text})}
          value={this.state.text}
        />
        <Text style={{padding: 10, fontSize: 42}}>
          {this.state.text.split(' ').map((word) => word && '🍕').join(' ')}
        </Text>
      </View>
    );
  }
}
```



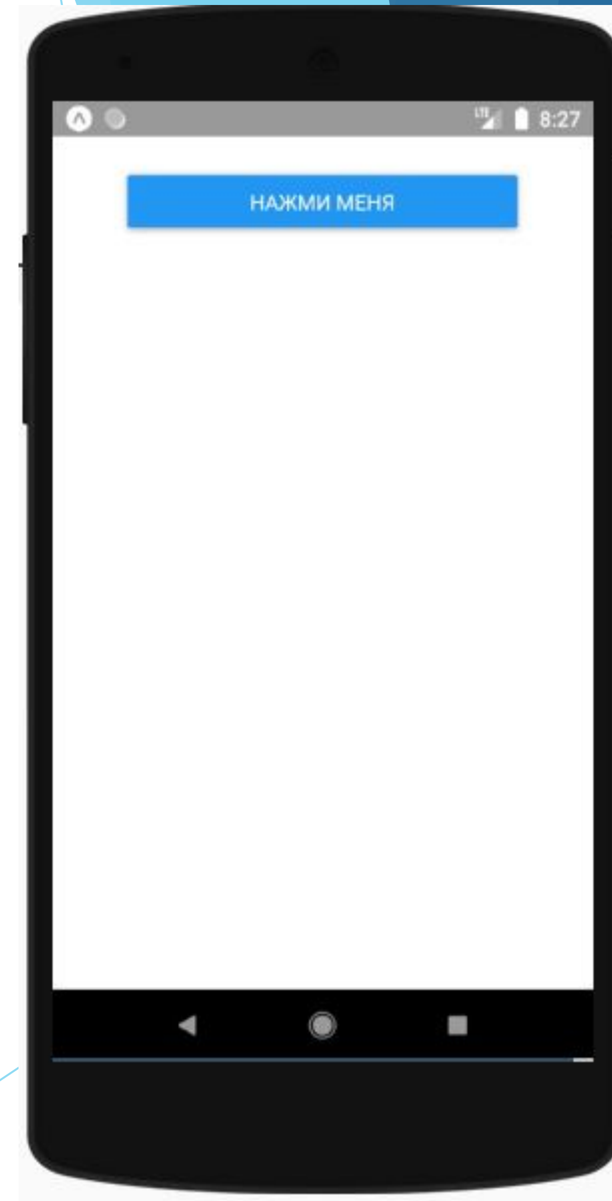
Обработка касаний

- Пользователи взаимодействуют с мобильными приложениями в основном с помощью касания. Они могут использовать комбинацию жестов, таких как нажатие кнопки, прокрутка списка или масштабирование карты. React Native предоставляет компоненты для обработки всевозможных общих жестов, а также комплексную систему реагирования на жесты, обеспечивающую более расширенное распознавание жестов. Самым базовым компонентом обработки касаний является кнопка (Button).

Отображение кнопки

- Кнопка обеспечивает базовый компонент кнопки, который нативно отображается на всех платформах.
- Свойства:
- **onPress** – обработчик, который вызывается при нажатии на кнопку
- **title** – текст, который отображается в кнопке
- **color** – цвет текста (для iOS), или цвет фона кнопки (Android)
- **disabled** – если истина, кнопка будет отключенной

```
export default class ViewWithButton extends React.Comp
onent {
    render() {
      return (
        <View style={{padding: 50}}>
          <Button
            onPress={() => {
              alert('Вы нажали кнопку!');
            }}
            title="Нажми меня"
          />
        </View>
      );
    }
  }
```

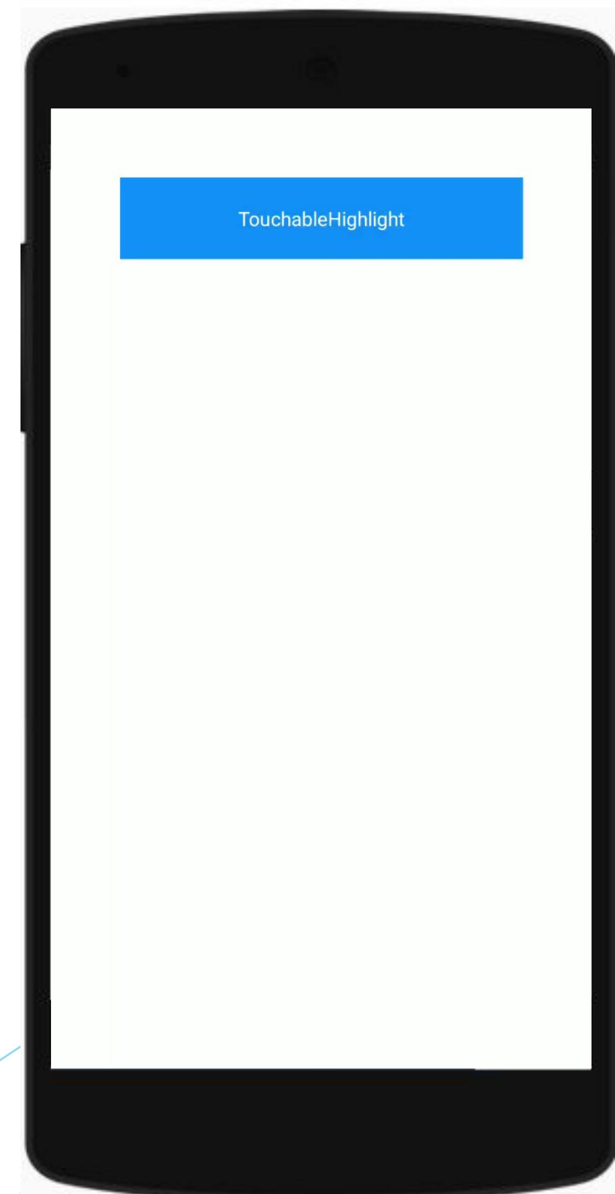


Touchables

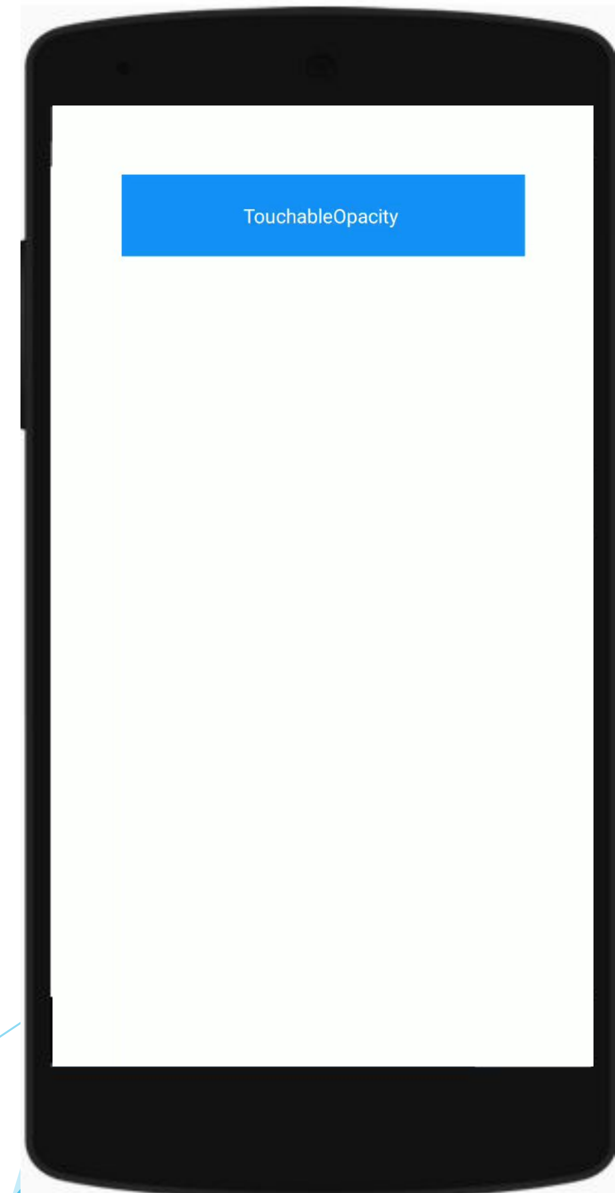
- Если базовая кнопка не подходит для вашего приложения, вы можете создать свою собственную кнопку, используя любой из «Touchable» компонентов, предоставляемых React Native. Компоненты «Touchable» обеспечивают возможность обработки касаний и могут отображать обратную связь при распознавании жеста. Однако эти компоненты не предоставляют стилей по умолчанию, поэтому вам нужно будет немного поработать, чтобы они хорошо выглядели в вашем приложении.

- Какой компонент Touchable вы используете, будет зависеть от того, какую обратную связь вы хотите предоставить:
- **TouchableHighlight** – как правило можно использовать везде: в качестве кнопки, ссылки. Фон будет замнён когда пользователь нажмёт на представление.
- **TouchableNativeFeedback** – можно использовать на Android, для отображения ряби на поверхности представления после прикосновения пользователя.
- **TouchableOpacity** – может использоваться для обеспечения обратной связи, увеличивая прозрачность кнопки, позволяя видеть фон, пока пользователь осуществляет нажатие.
- **TouchableWithoutFeedback** – используется для обработки касаний без какой-либо обратной связи.

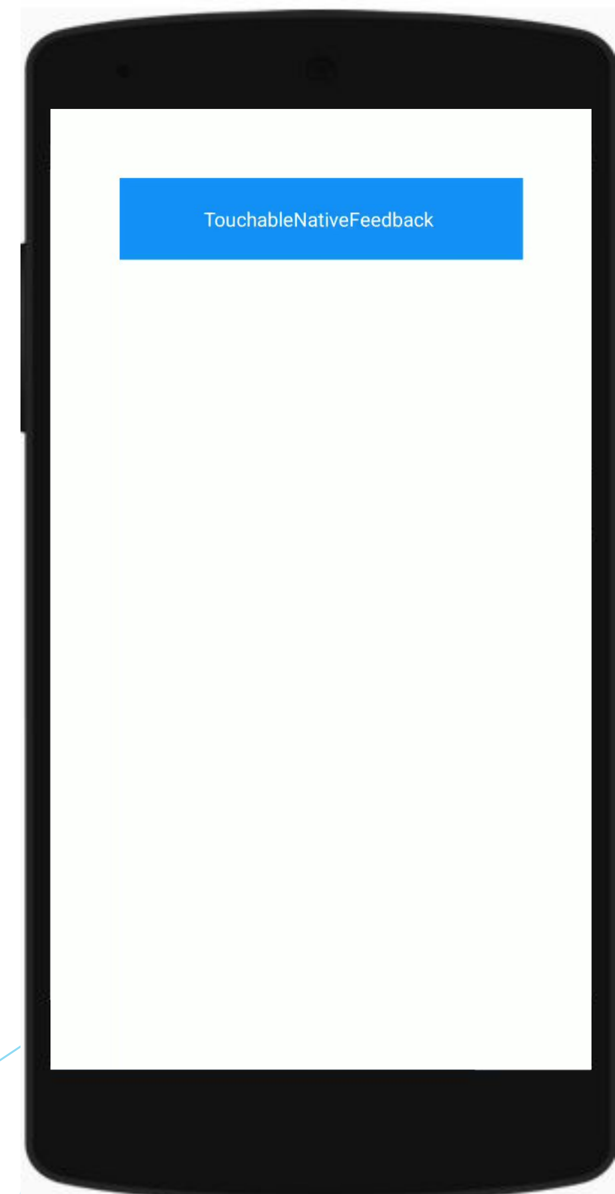
```
export default class ViewWithButton extends React.Component {
    _onPressButton() {
      alert('Вы нажали кнопку!')
    }

    render() {
      return (
        <View style={{padding: 50}}>
          <TouchableHighlight onPress={this._onPressButton} underlayColor="white
">
            <View style={{alignItems: 'center', backgroundColor: '#2196F3'}}>
              <Text style={{padding: 20, color: 'white'}}>TouchableHighlight</Te
xt>
            </View>
          </TouchableHighlight>
        </View>
      );
    }
  }
```



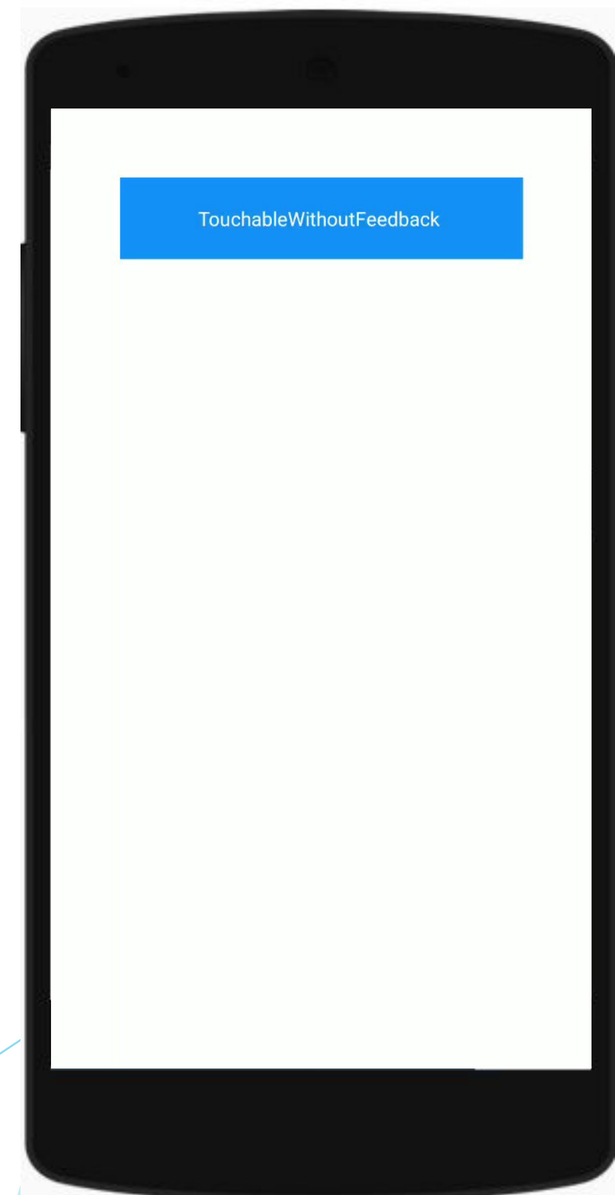
```
export default class ViewWithButton extends React.Component {
    _onPressButton() {
      alert('Вы нажали кнопку!')
    }

    render() {
      return (
        <View style={{padding: 50}}>
          <TouchableOpacity onPress={this._onPressButton} underlayColor
="white">
            <View style={{alignItems: 'center', backgroundColor: '#2196
F3'}}>
              <Text style={{padding: 20, color: 'white'}}>TouchableOpac
ity </Text>
            </View>
          </TouchableOpacity>
        </View>
      );
    }
  }
```



```
export default class ViewWithButton extends React.Component {
    _onPressButton() {
      alert('Вы нажали кнопку!')
    }

    render() {
      return (
        <View style={{padding: 50}}>
          <TouchableNativeFeedback onPress={this._onPressButton} underlayColor="white">
            <View style={{alignItems: 'center', backgroundColor: '#2196F3'}}>
              <Text style={{padding: 20, color: 'white'}}>TouchableNativeFeedback </Text>
            </View>
          </TouchableNativeFeedback>
        </View>
      );
    }
  }
```



```
export default class ViewWithButton extends React.Component {
    _onPressButton() {
      alert('Вы нажали кнопку!')
    }

    render() {
      return (
        <View style={{padding: 50}}>
          <TouchableWithoutFeedback onPress={this._onPressButton} under
layColor="white">
            <View style={{alignItems: 'center', backgroundColor: '#2196
F3'}}>
              <Text style={{padding: 20, color: 'white'}}>TouchableWith
outFeedback </Text>
            </View>
          </TouchableWithoutFeedback>
        </View>
      );
    }
  }
```



Работа с сетью

- Многие мобильные приложения должны загружать ресурсы с удаленного URL. Возможно, вы захотите сделать запрос POST к API REST или вам может понадобиться получить кусок статического контента с другого сервера.

Использование Fetch

Создание запросов

- Чтобы получить содержимое с произвольного URL, вы можете передать URL для получения:

```
fetch('https://mywebsite.com/mydata.json');
```

- Fetch также принимает необязательный второй аргумент, который позволяет настроить HTTP-запрос. Вы можете указать дополнительные заголовки или сделать запрос POST:

```
fetch('https://mywebsite.com/endpoint/', {
  method: 'POST',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    firstParam: 'вашеЗначение',
    secondParam: 'вашеВтороеЗначение',
  }),
});
```

Обработка ответа

- Работа в сети - это асинхронная операция. Методы fetch возвращают Promise, что упрощает написание кода, который работает асинхронно.

```
function getMoviesFromApiAsync() {
    return fetch('https://reactnative.dev/movies.json')
      .then((response) => response.json())
      .then((responseJson) => {
        return responseJson.movies;
      })
      .catch((error) => {
        console.error(error);
      });
  }
```

- Вы также можете использовать предложенный синтаксис ES2017 - async / await в приложении React Native

```
async function getMoviesFromApi() {
    try {
      let response = await fetch('https://reactnative.dev/movies.json');
      let responseJson = await response.json();
      return responseJson.movies;
    } catch (error) {
      console.error(error);
    }
  }
```

- По умолчанию iOS заблокирует любой запрос, который не зашифрован с использованием SSL. Если вам нужно извлечь из открытого текста URL-адрес (тот, который начинается с http), вам сначала нужно добавить исключение App Transport Security. Если вы заранее знаете, к каким доменам вам потребуется доступ, более безопасно добавлять исключения только для этих доменов; если домены не известны до времени выполнения, вы можете полностью отключить ATS. Тем не менее, обратите внимание, что с января 2017 года проверка Apple App Store потребует разумного обоснования для отключения ATS

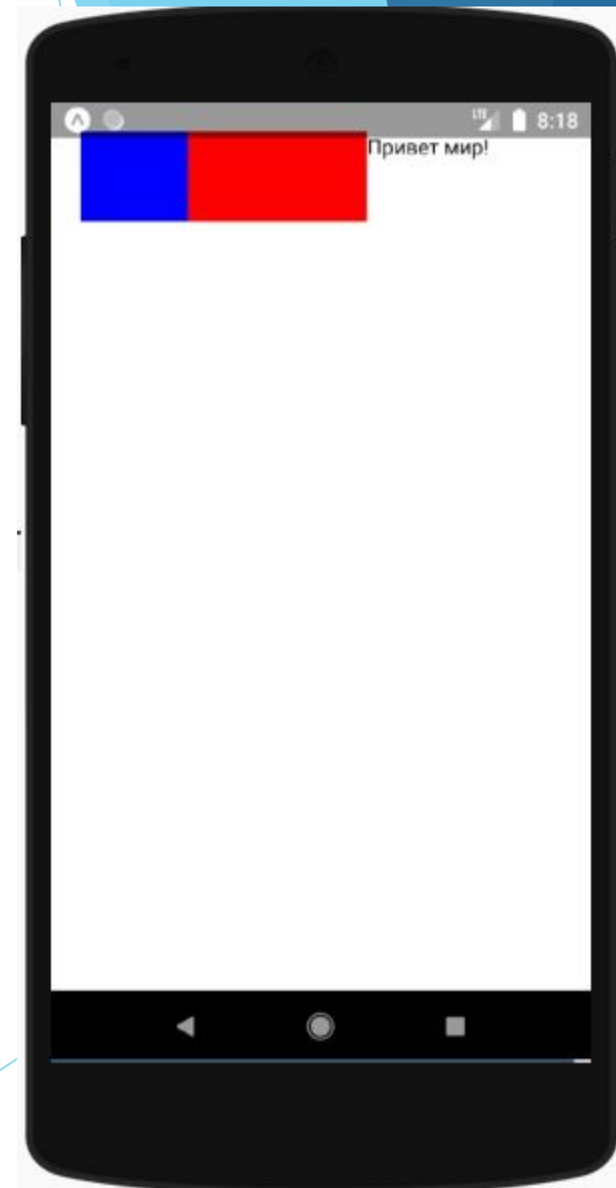
```
componentDidMount(){
  return fetch('https://reactnative.dev/movies.json')
    .then((response) => response.json())
    .then((responseJson) => {

      this.setState({
        isLoading: false,
        dataSource: responseJson.movies,
      }, function(){

      });

    })
    .catch((error) =>{
      console.error(error);
    });
}
```

```
render(){

  if(this.state.isLoading){
    return(
      <View style={{flex: 1, padding: 20}}>
        <ActivityIndicator/>
      </View>
    )
  }

  return(
    <View style={{flex: 1, paddingTop:20}}>
      <FlatList
        data={this.state.dataSource}
        renderItem={({item}) => <Text>{item.title}, {item.releaseYear}</Text>}
        keyExtractor={({id}, index) => id}
      />
    </View>
  );
}
```

View

- View – это фундаментальный компонент для создания пользовательского интерфейса. View – это контейнер, который поддерживает макет с flexbox, стилем, обработкой касаний.
- View служит для того, чтобы в нём размещались дочерние компоненты и может иметь от 0 до скольких угодно дочерних компонентов любого типа.

```
export default class ViewColoredBoxesWithText extends React.Component {
  render() {
    return (
      <View
        style={{
          flexDirection: 'row',
          height: 100,
          padding: 20,
        }}>
        <View style={{backgroundColor: 'blue', flex: 0.3}} />
        <View style={{backgroundColor: 'red', flex: 0.5}} />
        <Text>Привет мир!</Text>
      </View>
    );
  }
}
```



Параметры View

- **hitSlop** – определяет, как далеко будет срабатывать событие тапа по View. Типичные рекомендации по интерфейсу – 30-40 точек/пикселей, не зависящих от плотности. Например, если сенсорное View высотой 20, сенсорная высота может быть увеличена до 40.

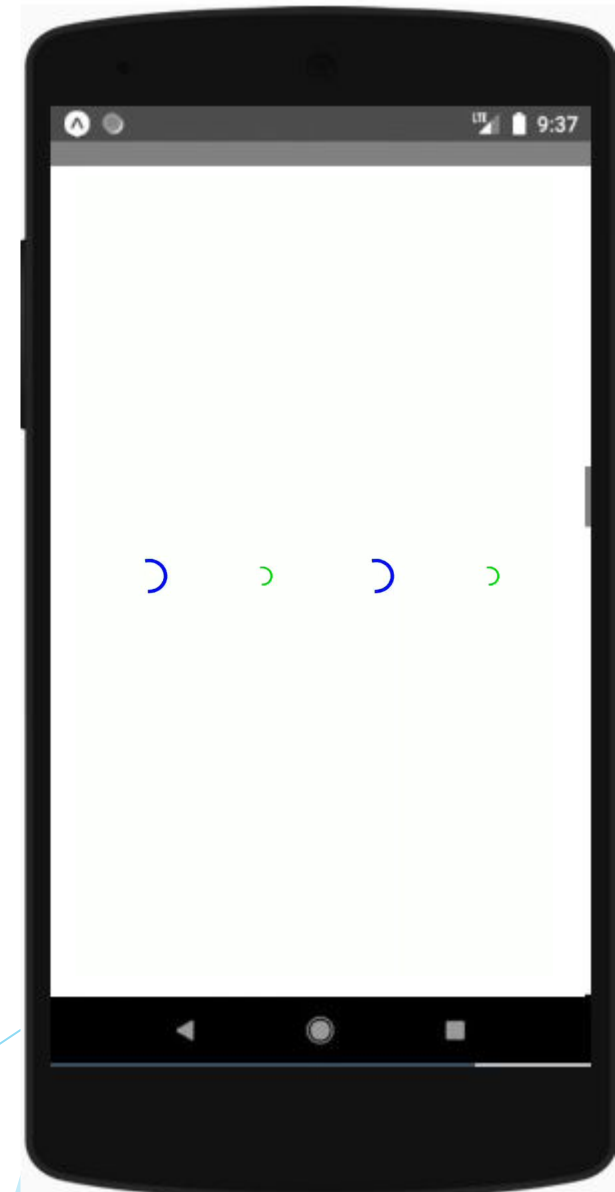
```
hitSlop={{top: 10, bottom: 10, left: 0, right: 0}}
```

- **Collapsable** – представления, которые используются только для разметки своих дочерних элементов, а иначе ничего не отрисовывают, могут автоматически удаляться из собственной иерархии в качестве оптимизации. Установите для этого свойства значение false, чтобы отключить эту оптимизацию.

ActivityIndicator

- Отображает круговой индикатор загрузки.
- **Свойства:**
- **animating** - показывать ли индикатор (true, по умолчанию) или скрывать его (false)
- **color** - цвет переднего плана индикатора (по умолчанию серый на iOS и темно-голубой на Android)
- **hidesWhenStopped** - должен ли индикатор скрываться при отсутствии анимации (по умолчанию true)
- **size** - размер индикатора (по умолчанию «маленький»). Передача числа в размер поддерживается только на Android.

```
export default class App extends Component {
    render() {
      return (
        <View style={{flex: 1, justifyContent:'space-around', flexDirection: 'row', padding: 10}}>
          <ActivityIndicator size="large" color="#0000ff" />
          <ActivityIndicator size="small" color="#00ff00" />
          <ActivityIndicator size="large" color="#0000ff" />
          <ActivityIndicator size="small" color="#00ff00" />
        </View>
      )
    }
  }
```

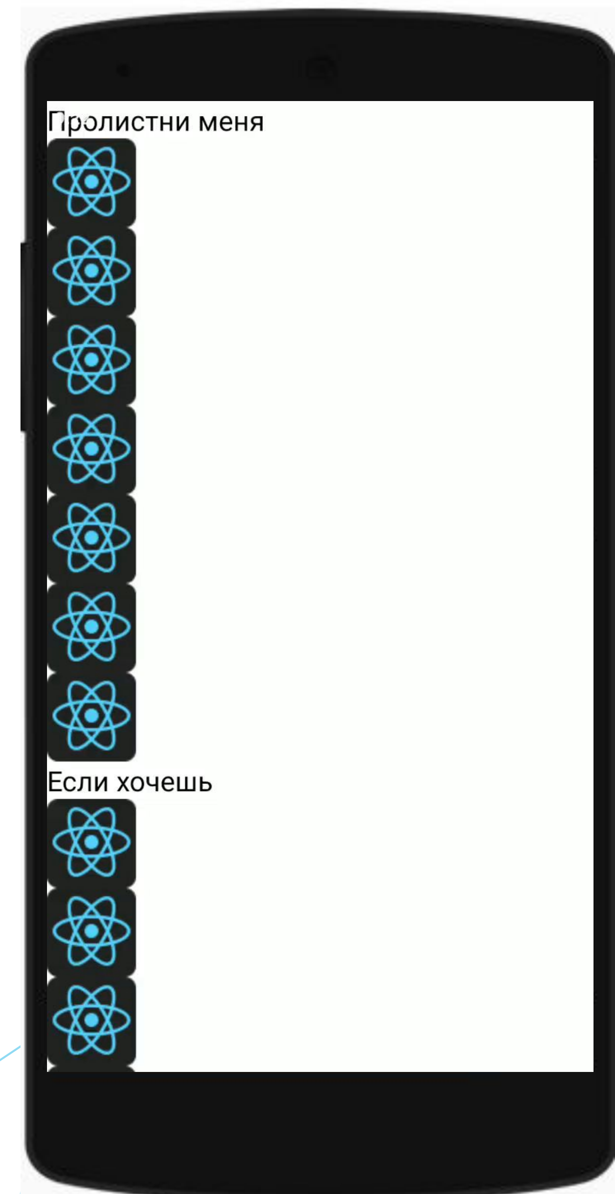


Использование ScrollView

- ScrollView - это универсальный контейнер с прокруткой, который может содержать несколько компонентов и представлений. Прокручиваемые элементы не обязательно должны быть однородными, и вы можете прокручивать как вертикально, так и горизонтально (установив свойство horizontal).
- ScrollView работает лучше всего, чтобы представить небольшое количество вещей ограниченного размера. Все элементы и представления ScrollView отображаются, даже если они в данный момент не отображаются на экране.

- ScrollViews можно настроить, чтобы разрешить пролистывание представлений с помощью жестов смахивания с помощью свойства pagingEnabled. Скольжение по горизонтали между представлениями также может быть реализовано на Android с помощью компонента ViewPager.
- В iOS можно использовать ScrollView с одним элементом, чтобы позволить пользователю масштабировать контент. Установите свойства MaximumZoomScale и MinimZoomScale, и ваш пользователь сможет использовать жесты увеличения и уменьшения для увеличения и уменьшения масштаба.

```
export default class ScrolledDownAndWhatHappenedNextShockedMe extends React.Component {
  render() {
    var records = [];
    for (let i = 0; i < 7; i++) {
      records.push(
        <Image source={{uri: "https://reactnative.dev/img/tiny_logo.png", width: 64, height: 64}} />
      );
    }
    return (
      <ScrollView>
        <Text style={{fontSize:20}}>Пролистни меня</Text>
        {records}
        <Text style={{fontSize:20}}>Если хочешь</Text>
        {records}
        <Text style={{fontSize:20}}>Листай ниже</Text>
        {records}
        <Text style={{fontSize:20}}>Какой фреймворк</Text>
        {records}
        <Text style={{fontSize:20}}>Лучший?</Text>
        {records}
        <Text style={{fontSize:20}}>React Native</Text>
      </ScrollView>
    );
  }
}
```



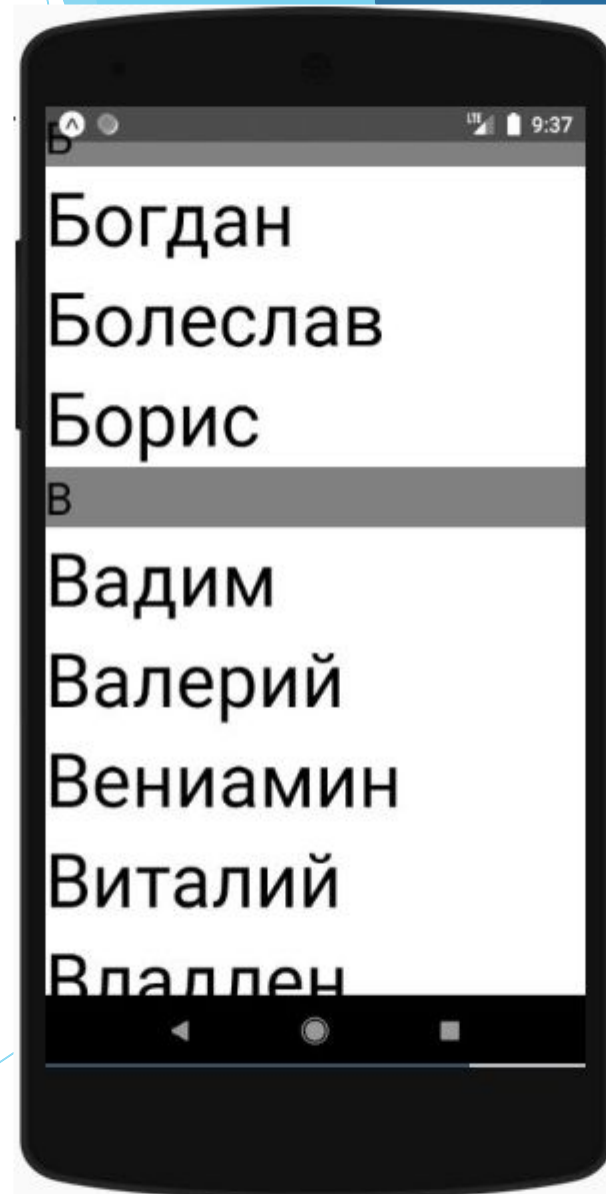
Использование ListView

- React Native предоставляет набор компонентов для представления списков данных. Как правило для этого используется FlatList или SectionList.
- Компонент FlatList отображает прокручиваемый список изменяющихся, но схожих по структуре данных. FlatList хорошо работает для длинных списков данных, где количество элементов может со временем меняться. В отличие от более общего ScrollView, FlatList отображает только элементы, которые в данный момент отображаются на экране, а не все элементы одновременно.

- Компонент FlatList требует двух реквизитов: data и renderItem. Данные являются источником информации для списка. renderItem берет один элемент из источника и возвращает отформатированный компонент для рендеринга.
- Если вы хотите визуализировать набор данных, разбитых на логические разделы, может быть, даже, с заголовками разделов, аналогично UITableViews на iOS, тогда используйте SectionList.
- Одним из наиболее распространенных применений для представления списка является отображение данных, которые вы получаете с сервера.

```
export default class FlatListBasics extends React.Component {
  render() {
    return (
      <View style={{flex: 1}}>
        <FlatList
          data={[
            {key: 'Богдан'},
            {key: 'Болеслав'},
            {key: 'Борис'},
            {key: 'Вадим'},
            {key: 'Валерий'},
            {key: 'Вениамин'},
            {key: 'Виталий'},
            {key: 'Владлен'},
            {key: 'Глеб'},
            {key: 'Григорий'},
          ]}
          renderItem={({item}) => <Text style={{fontSize: 50}}>{item.key}</Text>}
        />
      </View>
    );
  }
}
```

```
export default class FlatListBasics extends React.Component {
  render() {
    return (
      <View style={{flex: 1}}>
        <SectionList
          sections={[
            {title: 'Б', data: ['Богдан', 'Болеслав', 'Борис']},
            {title: 'В', data: ['Вадим', 'Валерий', 'Вениамин', 'Виталий', '
Владлен']},
            {title: 'Г', data: ['Глеб', 'Григорий']},
          ]}
          renderSectionHeader={({section}) => <Text style={{fontSize: 30, back
groundColor: 'gray'}}>{section.title}</Text>}
          renderItem={({item}) => <Text style={{fontSize: 50}}>{item}</Text>}
          keyExtractor={(item, index) => index}
        />
      </View>
    );
  }
}
```



Image

- Компонент React для отображения различных типов изображений, включая сетевые изображения, статические ресурсы, временные локальные изображения и изображения с локального диска, например, с камеры.
- Обратите внимание, что для сетевых изображений и изображений вам нужно будет вручную указать размеры вашего изображения!

Свойства изображения

- **blurRadius** - радиус размытия для фильтра размытия, добавленного к изображению
- **onLoad** - вызывается, когда загрузка успешно завершена
- **onLoadEnd** – вызывается, когда загрузка завершена (успешно или нет)
- **onLoadStart** - вызывается при запуске загрузки
- **resizeMode** - Определяет, как изменить размер изображения, когда рамка не соответствует необработанным размерам изображения. По умолчанию – cover.

- **cover** - равномерно масштабирует изображение (сохраняя пропорции изображения), чтобы оба размера (ширина и высота) изображения были равны или превышали соответствующий размер View (без отступов).
- **contain** - равномерно масштабирует изображение (сохранение соотношения сторон изображения), чтобы оба размера (ширина и высота) изображения были равны или меньше соответствующего размера View (за вычетом отступов).
- **stretch** - масштабирует ширину и высоту независимо, это может изменить соотношение сторон.
- **repeat** - повторяет изображение, чтобы закрыть кадр. Изображение сохранит свой размер и соотношение сторон, если оно не больше View, и в этом случае оно будет равномерно уменьшено, чтобы оно содержалось во View.

- **source** - Источник изображения (удаленный URL или локальный файловый ресурс).
- Этот объект также может содержать несколько удаленных URL-адресов, указанных вместе с их шириной и высотой и, возможно, с помощью аргументов scale. Свойство кэша может быть добавлено для управления взаимодействием сетевого запроса с локальным кэшем.
- В настоящее время поддерживаются следующие форматы: png, jpg, jpeg, bmp, gif, webp (только для Android), psd (только для iOS). Кроме того, iOS поддерживает несколько форматов изображений RAW.

- **loadingIndicatorSource** - подобно **source**, это свойство представляет ресурс, используемый для визуализации индикатора загрузки изображения, отображаемого до тех пор, пока изображение не будет готово к отображению, обычно после его загрузки из сети.
- **onError** – вызывается при ошибки загрузки изображения.
- **resizeMethod** - механизм, который должен использоваться для изменения размера изображения, когда размеры изображения отличаются от размеров View. По умолчанию авто.

- **auto** - используйте эвристику для выбора размера и масштаба.
- **resize** - программная операция, которая изменяет закодированное изображение в памяти, прежде чем оно будет декодировано. Это следует использовать вместо масштаба, когда изображение намного больше, чем вид.
- **scale** - изображение рисуется уменьшенным или увеличенным. По сравнению с resize масштабирование происходит быстрее (обычно с аппаратным ускорением) и позволяет получать изображения более высокого качества. Это следует использовать, если изображение меньше, чем вид. Его также следует использовать, если изображение немного больше, чем изображение.

- **defaultSource** - статическое изображение для отображения при загрузке источника изображения.
- **onPartialLoad** - вызывается, когда частичная загрузка изображения завершена. Определение того, что составляет «частичную загрузку», зависит от конкретного загрузчика, хотя это предназначено для прогрессивной загрузки JPEG.
- **progressiveRenderingEnabled** - Только для Android. Когда true, включает прогрессивную потоковую передачу JPEG.

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

export default class DisplayAnImage extends Component {
  render() {
    return (
      <View>
        <Image
          style={{width: 50, height: 50}}
          source={{uri: 'https://reactnative.dev/img/tiny_logo.png'}}
        />
      </View>
    );
  }
}
```

ImageBackground

- По поведению похож на **background-image** в web.
- Свойства у ImageBackground такие же, как и у Image.

```
return (
    <ImageBackground source={...} style={{width: '
100%', height: '100%'}}>
      <Text>Внутри</Text>
    </ImageBackground>
  );
```

KeyboardAvoidingView

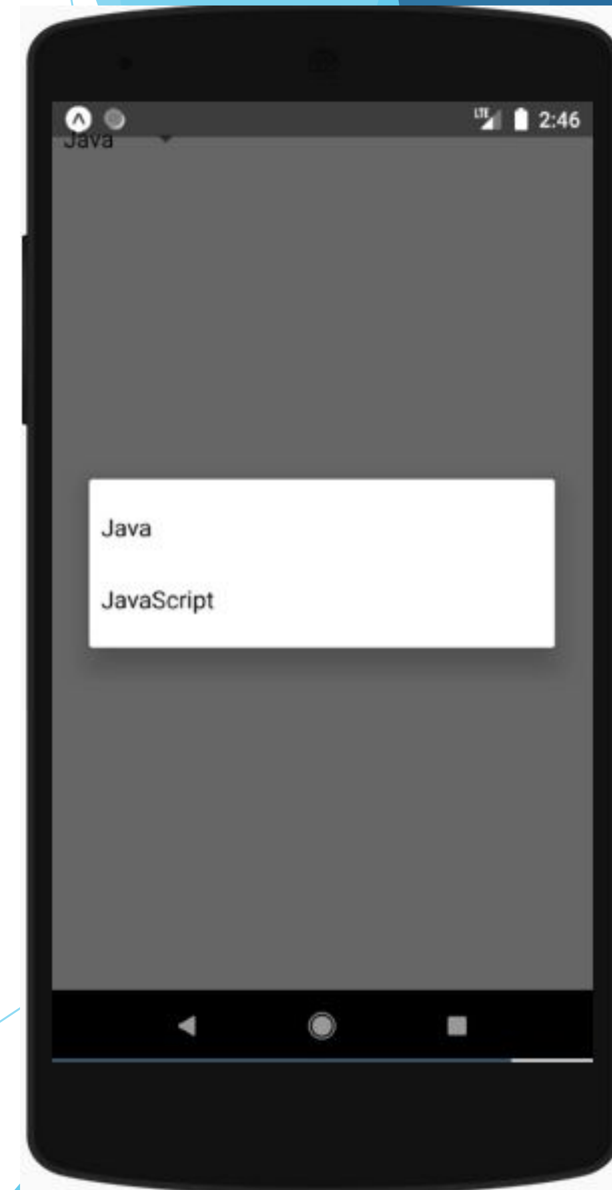
- Это компонент для решения общей проблемы представлений, которым необходимо отодвинуться от виртуальной клавиатуры. Он может автоматически регулировать высоту, положение или нижнее заполнение в зависимости от положения клавиатуры

```
<KeyboardAvoidingView style={styles.container} behavior
="padding" enabled>
  ... ваш интерфейс ...
</KeyboardAvoidingView>
```

Picker

- Отображает нативный компонент выбора на Android и iOS

```
<Picker
  selectedValue={this.state.language}
  style={{height: 50, width: 100}}
  onValueChange={(itemValue, itemIndex) =>
    this.setState({language: itemValue})
  }>
  <Picker.Item label="Java" value="java" />
  <Picker.Item label="JavaScript" value="js" />
</Picker>
```



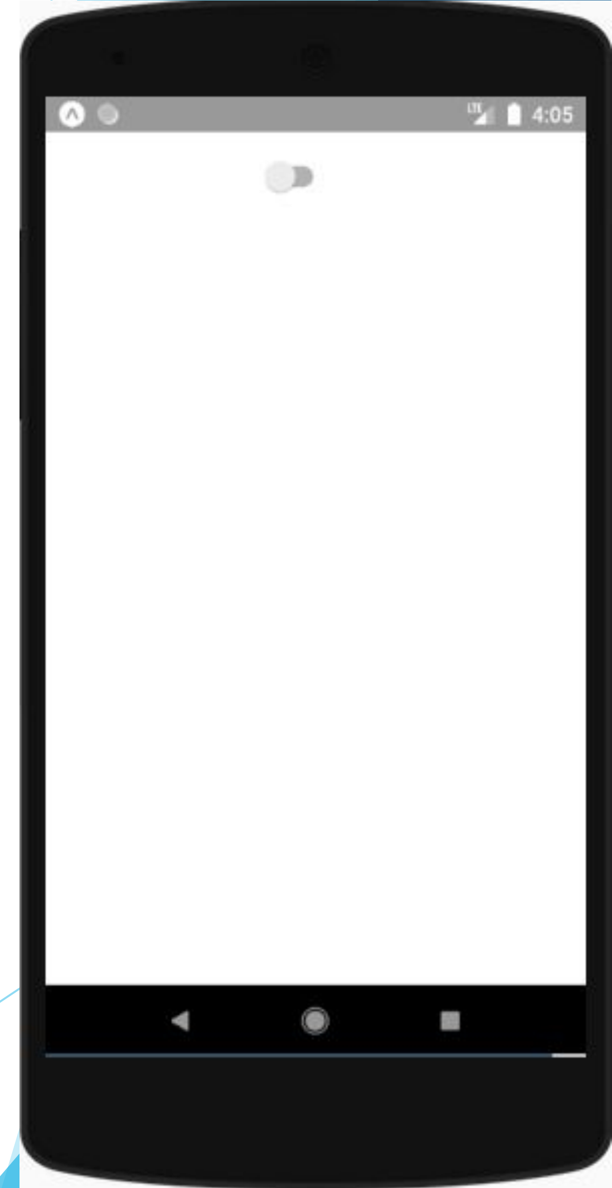
Свойства Picker

- **onValueChange** – вызывается со следующими параметрами когда элемент выбран:
  - **itemValue** – значение свойства выбранного элемента
  - **itemPosition** – индекс выбранного элемента
- **selectedValue** - значение, совпадающее со значением одного из элементов. Может быть строкой или целым числом.
- **enabled** – если false, то компонент выбора будет отключён
- **mode** - на Android указывает, как отображать элементы выбора, когда пользователь нажимает на средство выбора:
  - '**dialog**': показать модальное диалоговое окно (по умолчанию)
  - '**dropdown**': показывает раскрывающийся список, привязанный к представлению выбора

Switch

- Отображает логический ввод.
- Это контролируемый компонент, для которого требуется обратный вызов onValueChange, который обновляет значение prop, чтобы компонент отражал действия пользователя. Если значение prop не обновлено, компонент продолжит отображать предоставленное значение prop вместо ожидаемого результата каких-либо действий пользователя.

```
export default class DisplaySwitch extends Component {
  render() {
    return (
      <View  style={{padding: 10}}>
        <Switch />
      </View>
    );
  }
}
```

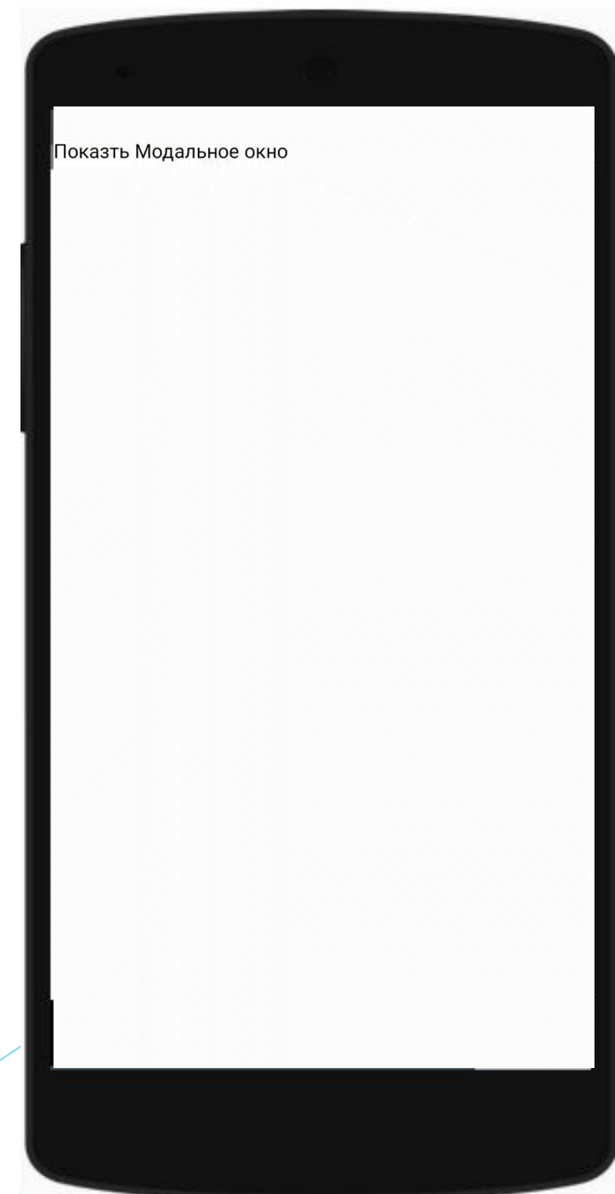


Модальное окно (Modal)

- Модальный компонент - это основной способ представления контента над вложенным представлением.
- **Свойства модального компонента:**
- **visible** – следует ли отображать модальный компонент
- **onRequestClose** – колбэк, вызывается когда пользователь нажимает кнопку аппаратного возврата на Android или кнопку меню на Apple TV
- **onShow** – колбэк, вызывается когда происходит отображение модального компонента
- **transparent** – определяет, будет ли модальный компонент прозрачным

- **animationType** – контролирует, как будет появляться модальное окно
  - **slide** – выскальзывает снизу
  - **fade** – проявляется на экране
  - **none** – появляется без анимации

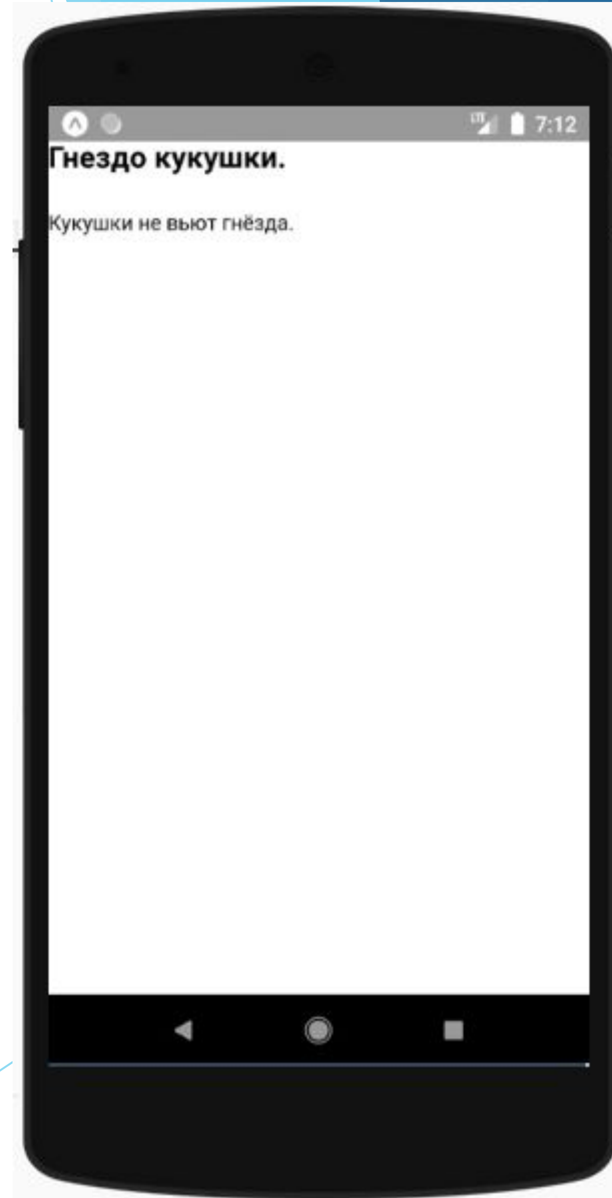
```
export default class ModalExample extends Component {
  state = {
    modalVisible: false,
  };

  render() {
    return (
      <View style={{marginTop: 22}}>
        <Modal
          animationType="slide"  transparent={false}  visible={this.state.modalVi
sible}    onRequestClose={() => {
            Alert.alert('Модальное окно закрыто.');  }}>
          <View style={{marginTop: 22}}>
            <View>
              <Text>Привет мир!</Text>
              <TouchableHighlight
                onPress={() => {  this.setState({modalVisible: false});}}>
                <Text>Скрыть окно</Text>
              </TouchableHighlight>
            </View>
          </View>
        </Modal>
        <TouchableHighlight
          onPress={() => {
            this.setState({modalVisible: true});
          }}>
          <Text>Показть Модальное окно</Text>
        </TouchableHighlight>
      </View>
    );
  }
}
```



Текст (Text)

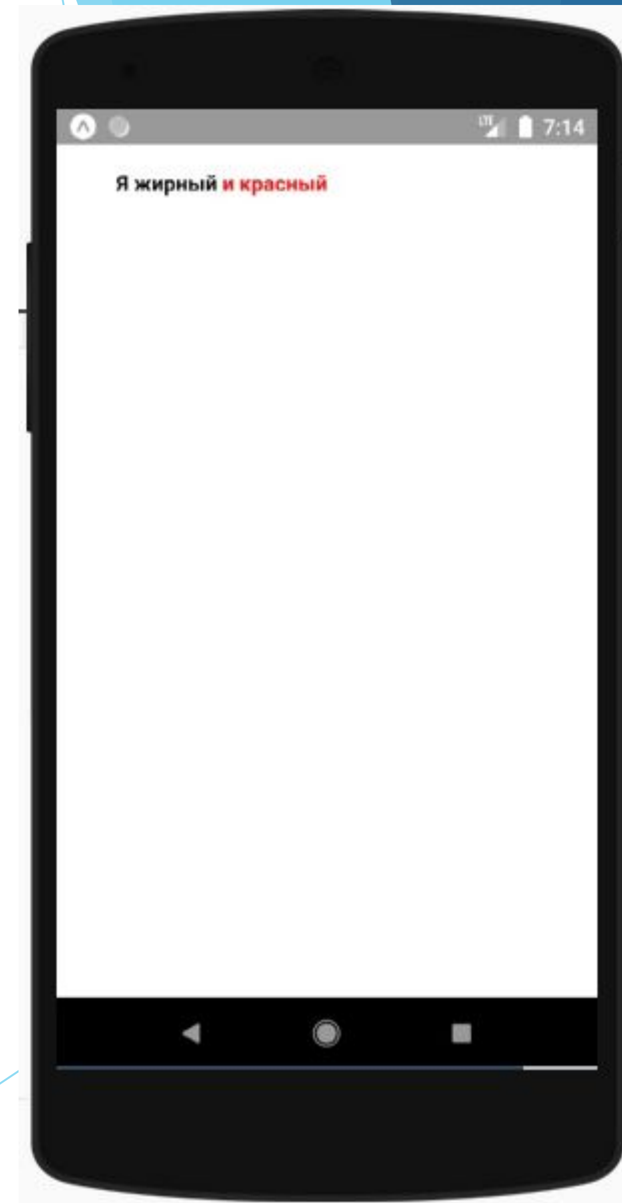
- Компонент React для отображения текста.
- Текст поддерживает вложение, стилизацию и обработку касаний.
- **Вложенный текст** - и Android, и iOS позволяют отображать форматированный текст, помечая диапазоны строк определенным форматированием, например, жирным или цветным текстом (NSAttributedString в iOS, SpannableString в Android). На практике это очень утомительно. В React Native для этого используется веб-парадигма, где вы можете вкладывать текст для достижения того же эффекта.

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

export default class TextInANest extends Component {
  constructor(props) {
    super(props);
    this.state = {
      titleText: "Гнездо кукушки.",
      bodyText: 'Кукушки не вьют гнёзда.'
    };
  }

  render() {
    return (
      <Text style={styles.baseText}>
        <Text style={styles.titleText} onPress={this.onPressTitl
e}>
          {this.state.titleText}{'\n'}{'\n'}
        </Text>
        <Text numberOfLines={5}>
          {this.state.bodyText}
        </Text>
      </Text>
    );
  }
}
```



Вложенный текст

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

export default class BoldAndBeautiful extends Component {
  render() {
    return (
      <Text style={{fontWeight: 'bold', padding: 40}}>
        Я жирный
        <Text style={{color: 'red'}}>
          и красный
        </Text>
      </Text>
    );
  }
}
```



- За кулисами React Native преобразует вложенность в обычный NSAttributedString или SpannableString, который содержит следующую информацию:

```
"я жирный и красный"
0-8: bold
8-18: bold, red
```

Контейнеры

- Элемент <Text> уникален: все внутри больше не использует макет flexbox, а использует текстовый макет. Это означает, что элементы внутри <Text> больше не являются прямоугольниками, а переносятся, когда видят конец строки.

```
<Text>
  <Text>Первая часть и </Text>
  <Text>вторая часть</Text>
</Text>
// Текстовый контейнер: текст будет встроен, если это п
озволяет пространство
// |Первая часть и вторая часть|

// Если пространство не позволяет
// |Первая часть |
// |и вторая     |
// |часть        |
```

```
<View>
  <Text>Первая часть и </Text>
  <Text>вторая часть</Text>
</View>
// View контейнер: каждый Text - это блок
// |Первая часть и|
// |вторая часть  |

// Или Текст может перенести свой блок
// |Первая часть |
// |и            |
// |вторая часть |
```

Свойства

- **dataDetectorType** - Определяет типы данных, преобразованных в интерактивные URL-адреса в текстовом элементе. По умолчанию типы данных не обнаруживаются. Можно выбрать только один тип. Возможные значения для dataDetectorType:
  - 'phoneNumber'
  - 'link'
  - 'email'
  - 'none'
  - 'all’
- **onPress** – вызывается при нажатии
- **selectable** – разрешить пользователю выделять текст, для дальнейшего нативного копирования

ProgressBarAndroid

- Компонент React только для Android, используемый для указания того, что приложение загружается или в нем есть какая-то активность.
- **Свойства**
- **animating** - показывать ли ProgressBar (true, по умолчанию) или скрывать его (false)
- **color** – цвет
- **indeterminate** – следует ли показывать неопределённый прогресс
- **progress** – прогресс выполнения (от 0 до 1)

- **styleAttr** – стиль ProgressBar:
  - Horizontal
  - Normal (по умолчанию)
  - Small
  - Large
  - Inverse
  - SmallInverse
  - LargeInverse

```
export default class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <ProgressBarAndroid />
        <ProgressBarAndroid styleAttr="Horizontal" />
        <ProgressBarAndroid styleAttr="Horizontal" color="#2196F3" />
        <ProgressBarAndroid
          styleAttr="Horizontal"
          indeterminate={false}
          progress={0.5}
        />
      </View>
    );
  }
}
```