[**1. Что такое React Native? Общая характеристика, применение, аналоги.**](#_heading=h.s2vu5rburylk) **4**

[**2. Для чего используется React Native?**](#_heading=h.v4x6i6u35lq1) **5**

[**3. Текущие стандарты JavaScript: ECMAScript2015 (он же ES6), ECMAScript2019 (он же ES10)**](#_heading=h.3d20oayuggi7) **6**

[**4. Расширенный набор JavaScript, который добавляет строгую типизацию**](#_heading=h.73j3ls5ppra) **7**

[**5. React Native библиотеки: React-Router, Redux, Redux-Thunk, Redux-Saga, GraphQL**](#_heading=h.sheqebbw6s99) **8**

[**6. Менеджеры пакетов: NPM, Yarn**](#_heading=h.qfnaxinfqsck) **9**

[**7. Инструменты, помогающие поддерживать стиль кода: ESLint, TSLint, Prettier**](#_heading=h.q0fvhyc82139) **10**

[**8. Виды и общая характеристика компонентов в React Native. Отличия, область применения.**](#_heading=h.xcnk8nf4c1ge) **11**

[**9. Повторное использование кода в приложении на react Native. Какие методы вы знаете, чтобы разделять общую логику?**](#_heading=h.qq84uhc3v4fo) **12**

[**10. JSX в React Native. Общая характеристика и примеры использования.**](#_heading=h.1hr04qyy6f2t) **13**

[**11. Установка React Native и создание каркаса приложения.**](#_heading=h.rezb8xfc87nw) **14**

[**12. Redux - общая характеристика, назначение, примеры использования.**](#_heading=h.825bo5rbrd3s) **16**

[**13. Состояние приложения state. Назначение, примеры использования.**](#_heading=h.xewzjllu880w) **17**

[**14. React Native как инструмент создания общей кодовой базы для Android и iOS.**](#_heading=h.6ng3ps6pq115) **18**

[**15. SetState — механизм использования, примеры.**](#_heading=h.j8kzn8uazkec) **19**

[**Передача функции обновления позволяет получить доступ к значению текущего состояния внутри программы обновления. Поскольку вызовы setState являются пакетными, это позволяет вам связывать обновления и гарантировать, что они строятся друг на друге, а не конфликтуют:**](#_heading=h.yxznb6309ls8) **19**

[**16. Кросс-платформенная разработка приложений?**](#_heading=h.an7agf4pgd3u) **21**

[**17. Redux в React Native. Основные компоненты Redux в приложении React Native.**](#_heading=h.4e9veu9w83g3) **22**

[**18. UI-библиотека React Native — назначение, преимущества.**](#_heading=h.1k2g6zrt364u) **23**

[**19. Особенности разработки приложений React Native с использованием IDE.**](#_heading=h.lob524hoqi2k) **24**

[**20. Свойства (props) в приложении React Native. Назначение, примеры.**](#_heading=h.iujdnqt3ei0d) **25**

[**21. Компонент Scrollview. Назначение, примеры.**](#_heading=h.2bjwcvgxfj76) **26**

[**22. Функциональные и классовые компоненты в приложении React Native.**](#_heading=h.90x8mf5cc2ig) **27**

[**23. Методы жизненного цикла компонентов в React Native.**](#_heading=h.q82m43tn85ab) **28**

[**24. Управление состоянием в приложении React Native.**](#_heading=h.wev0mdky2wuk) **32**

[**25. Redux store. Назначение, применение, примеры использования.**](#_heading=h.cdk65orjs8uk) **34**

[**26. Практическое использование методов shouldComponenUpdate()**](#_heading=h.tg77xxmu0w0s) **35**

[**27. Сравнительная характеристика компонентов Scrollview и FlatList.**](#_heading=h.drqfaq4zwuyh) **37**

[**28. Хуки в приложении React Native. Общий механизм работы.**](#_heading=h.v29i6doqymfb) **38**

[**29. Функция onPress в React Native View.**](#_heading=h.swxp71lf8sl9) **39**

[**30. Передача информации между экранами в React Native.**](#_heading=h.om15qrm710sq) **40**

[**31. Какие способы вы знаете для обработки состояния приложения без Redux?**](#_heading=h.pjls8pybbjfu) **41**

[**32. Маршрутизация в приложениях React Native. (с навигацией?)**](#_heading=h.2jcuc11c3v3l) **42**

[**33. Универсальные компоненты React Native.**](#_heading=h.7prmw74yasok) **43**

[**34. Платформенно-специализированные компоненты React Native.**](#_heading=h.u0xqqznoj9gl) **44**

[**35. Сравнительная характеристика props и state в React Native.**](#_heading=h.hqrmyo7cumom) **45**

[**36. Способы создания стилей в React Native.**](#_heading=h.5zylhmtzl1rq) **46**

[**37. Навигация в приложениях React Native**](#_heading=h.qfy5t74an8th) **47**

[**38. Зачем использовать классы в React Native?**](#_heading=h.kvm3ctwxwos0) **48**

[**39. “Умные” и “глупые” компоненты в React Native. Сравнительная характеристика.**](#_heading=h.21d26m8bvp46) **49**

[**40. Как использовать CSS в React Native**](#_heading=h.bva1xxt3h2r0) **50**

[**41. Обработка пользовательского ввода в React Native.**](#_heading=h.u6x1nxipy5gf) **51**

[**42. Основные компоненты в React Native и аналогичные html-объекты.**](#_heading=h.g8a728b8h8ft) **52**

[**43. Компонент ListView и его использование в React Native.**](#_heading=h.3vk5qhv1k9mb) **53**

[**44. Асинхронное хранилище в React Native — общее назначение, области применения.**](#_heading=h.4l0jg2bwpak0) **55**

[**45. Хранение конфиденциальных данных в React. Как можно достичь готовым способом React или как можно это сделать за счет других средств?**](#_heading=h.bv7g8iqjtdqy) **56**

[**46. Анимация в приложении React Native. Общий механизм, обеспечение плавности.**](#_heading=h.lo1ajbtowigc) **57**

[**47. Сходства между реквизитом (props) и состоянием (state).**](#_heading=h.6ozq41fu1wgx) **59**

[**48. Гибкие макеты в React Native. Принципы создания, примеры.**](#_heading=h.u4ebjsyxexi7) **60**

[**49. Синхронные и асинхронные действия в Redux.**](#_heading=h.l3u07zb27y5i) **63**

[**50. Что позволяют сделать свойства стиля borderWidth, borderColor, borderRadius?**](#_heading=h.cxdb1up5i8y) **64**

[**51. Линковка пользовательских шрифтов в React Native.**](#_heading=h.qyobyrjcgxed) **65**

[**52. Задание стиля текста с помощью свойств в React Native.**](#_heading=h.pu8muwmsrtmj) **66**

[**53. Отображение контента React Native поверх Status Bar и System Bar.**](#_heading=h.tyyn3h9nr0ca) **67**

[**54. Свойства стилей React Native, отвечающие за расположение объекта.**](#_heading=h.awtberjedyfr) **68**

[**55. Задание относительных размеров объектов в React Native.**](#_heading=h.5dr8t9trfac0) **69**

[**56. Компиляция кода React Native в платформенно-специфичный код.**](#_heading=h.bj01kg6nqrb) **70**

[**57. Как реализовать анимацию в React Native?**](#_heading=h.mlpm22jx90ll) **71**

[**58. Организация файловой системы в React Native.**](#_heading=h.3xm5fqjaqiua) **72**

[**59. За счет чего можно осуществить передачу данных с одного экрана на другой?**](#_heading=h.km18uqfsu9d4) **73**

[**60. Работа с БД в React Native.**](#_heading=h.d18xpc79ucoz) **74**

[**61. Загрузка данных с сервера в React Native.**](#_heading=h.trdk8igvpo35) **75**

[**62. Component Update. Общий механизм работы.**](#_heading=h.olvjtullqeiv) **76**

[**63. С помощью какого компонента можно идентифицировать платформу, на которой будет запускаться приложение? Для чего это нужно?**](#_heading=h.hvu3nn7dsyze) **77**

[**64. Какое свойство стиля управляет направлением, в котором размещаются дочерние элементы части экрана?**](#_heading=h.lupixsqnz6gb) **78**

[**65. Библиотеки обработки жестов, совместимые с React Native.**](#_heading=h.g3oit69iva8d) **79**

[**66. Какие методы обновления жизненного цикла компонента вы знаете?**](#_heading=h.wq97wal1f79e) **80**

[**67. Зачем нужен UseEffect?**](#_heading=h.m9q0rahydwhx) **81**

## 1. Что такое React Native? Общая характеристика, применение, аналоги.

**Общая характеристика**

React Native это фреймворк для разработки мобильных приложений, который позволяет создавать приложения для Android и iOS используя JavaScript и среду разработки React. Он использует компонентный подход к разработке, который позволяет разработчикам создавать элементы интерфейса пользователя, такие как кнопки и текстовые поля, и использовать их в разных частях приложения. Приложения, созданные с помощью React Native, могут использовать все функции телефона, такие как камера, GPS и датчики, и могут быть опубликованы в App Store и Google Play.

**Применение**

React Native используется для создания кросс-платформенных мобильных приложений, которые могут работать на нескольких платформах без изменения кода. Он также может использоваться для разработки веб-приложений и десктопных приложений.

**Аналоги**

1. Xamarin - фреймворк для создания кросс-платформенных мобильных приложений, использующий C#.
2. Flutter - фреймворк для создания мобильных приложений, использующий Dart, который также позволяет создавать кросс-платформенные мобильные и веб-приложения.
3. Ionic - фреймворк для создания кросс-платформенных мобильных приложений, использующий Angular и Apache Cordova.

## 

## 2. Для чего используется React Native?

**React Native** используется для создания мобильных приложений для Android и iOS. Он предлагает множество преимуществ для разработчиков, включая:

* Возможность разрабатывать приложения для разных платформ одновременно, что снижает время разработки и уменьшает стоимость разработки.
* Использование JavaScript в качестве языка разработки, который является одним из самых популярных языков в мире программирования. Это означает, что многие разработчики уже имеют навыки, необходимые для работы с React Native.
* Использование компонентного подхода к разработке, который позволяет разработчикам создавать элементы интерфейса пользователя, такие как кнопки и текстовые поля, и использовать их в разных частях приложения. Это облегчает поддержку и обновление приложений.
* Инструменты для отладки и профилирования, которые позволяют разработчикам создавать высококачественные приложения с минимальными усилиями.

В общем, React Native используется для создания мобильных приложений, которые работают на разных платформах, с использованием JavaScript и компонентного подхода к разработке.

## 

## 3. Текущие стандарты JavaScript: ECMAScript2015 (он же ES6), ECMAScript2019 (он же ES10)

**ECMAScript2015** (также известный как ES6) является шестой версией стандарта ECMAScript, который был опубликован в 2015 году. Он включает в себя ряд улучшений и новых функций, таких как классы, модули, лямбда-выражения и параметры по умолчанию.

**ECMAScript2019** (также известный как ES10) является десятой версией стандарта ECMAScript, который был опубликован в 2019 году. Он включает в себя ряд улучшений и новых функций, таких как оператор spread для массивов, объектов и строк, а также методы для работы со строками и массивами.

Cтандарт JavaScript ECMAScript2019 (также известный как ES10) включает в себя следующие новые функции:

* Оператор spread: Этот оператор может быть использован для разворачивания массивов, объектов и строк. Он позволяет разворачивать элементы массива в отдельные аргументы или разворачивать свойства объекта в отдельные переменные.
* Методы строк: Новые методы строк включают в себя trimStart(), trimEnd() и matchAll(). Метод trimStart() удаляет все пробелы в начале строки, метод trimEnd() удаляет все пробелы в конце строки, а метод matchAll() возвращает все совпадения регулярного выражения в строке.
* Методы массивов: Новые методы массивов включают в себя flat(), flatMap() и trimEnd(). Метод flat() создает новый массив, содержащий все элементы из исходного массива развернутыми на определенную глубину, а метод flatMap() создает новый массив, содержащий результаты вызова функции обратного вызова для каждого элемента исходного массива, развернутыми на определенную глубину. Метод trimEnd() удаляет все элементы из конца массива, пока функция обратного вызова не вернет false.

## 

## 4. Расширенный набор JavaScript, который добавляет строгую типизацию

TypeScript, расширенный язык JavaScript со строгой типизацией. TypeScript является суперсетом JavaScript, что означает, что все коды написанные на JavaScript также являются корректными TypeScript. TypeScript добавляет типы данных и структуру языка, что позволяет разработчикам писать более структурированный и масштабируемый код.

Строгая типизация в TypeScript означает, что разработчики должны указывать типы данных для переменных и аргументов функций. Это позволяет избежать ошибок, которые могут возникнуть в результате неправильной работы с типами данных. Например, если вы попытаетесь присвоить строку переменной, ожидающей число, это вызовет ошибку во время компиляции, что позволит избежать проблем в процесе работы приложения.

TypeScript также позволяет разработчикам использовать интерфейсы, классы и модификаторы доступа, которые помогают структурировать код и улучшают его читабельность.

TypeScript широко используется в разработке веб-приложений, особенно с использованием фреймворка Angular. Он также может использоваться с React Native.

## 

## 5. React Native библиотеки: React-Router, Redux, Redux-Thunk, Redux-Saga, GraphQL

**React-Router:** это библиотека для управления маршрутизацией в приложениях React. Она позволяет определять маршруты и обрабатывать их в зависимости от URL.

**Redux:** это библиотека для управления состоянием приложения. Она позволяет хранить все состояние приложения в едином хранилище, которое может быть обновлено только с помощью экшенов.

**Redux-Thunk:** это библиотека для написания асинхронных экшенов в Redux. Она позволяет вам отложить диспатч экшена до тех пор, пока не будут выполнены какие-то асинхронные операции.

**Redux-Saga:** это библиотека для написания саг (side-effects) в Redux. Она позволяет вам отделить саги от редьюсеров и управлять ими через генераторы.

**GraphQL:** это язык запросов и схема данных, который позволяет клиентам запрашивать только те данные, которые им нужны. Он также позволяет серверу описывать схему данных и обрабатывать запросы клиентов. Библиотека GraphQL для React Native позволяет использовать GraphQL в мобильных приложениях, создавая и отправляя запросы на сервер и обрабатывая ответы.

## 

## 6. Менеджеры пакетов: NPM, Yarn

**Пакет** — это специальный набор файлов и данных. В пакете содержится:

* сама программа, которую нужно установить;
* файлы с настройкой — куда что копировать;
* информация о том, какие ещё пакеты или библиотеки нужны для установки;
* команды и другие действия, которые нужно будет выполнить во время установки;
* сведения о версии программы.

В обычной программе-установщике всё это тоже есть, но установщик может сам выполнить все необходимые команды и установить программу, а пакет — нет. Для установки и управления пакетами как раз и нужен менеджер пакетов. Кроме установки, у менеджера пакетов много работы, но основная — распутать все зависимости. Зависимость пакетов — это когда для работы одного пакета требуется установка нескольких других, причём определённых версий.Отдельная задача менеджера пакетов — проследить, чтобы при удалении пакета также удалились и все его зависимости (то, что нужно было ему для работы). Но при этом менеджер посмотрит, а не использует ли другой пакет что-то из этих зависимостей. Если использует — менеджер не будет её удалять, а оставит для другой программы.

**npm** (Node Package Manager) – дефолтный пакетный менеджер для JavaScript, работающий на Node.js. Менеджер npm состоит из двух частей:

* CLI (интерфейс командной строки) – средство для размещения и скачивания пакетов,
* онлайн-репозитории, содержащие JS пакеты.

Структуру репозитория npmjs.com можно представить, как центр исполнения заказов, который получает товары (npm-пакеты) от продавцов (авторы пакетов) и распространяет эти товары среди покупателей (пользователи пакетов). В центре исполнения заказов в качестве персональных менеджеров для каждого покупателя работает армия вомбатов (npm CLI).

**Yarn** - это менеджер пакетов, который выполняет функции менеджера проектов:

* Рабочие пространства - Разделите свой проект на подкомпоненты, хранящиеся в одном репозитории.
* Стабильность - Yarn гарантирует, что установка, которая работает сейчас, будет продолжать работать таким же способом и в будущем
* Документация - ей уделяется особое внимание
* Плагины - Yarn не может решить все ваши проблемы, но становится основой для того, чтобы это сделали другие
* Yarn - это независимый проект с открытым исходным кодом

## 

## 7. Инструменты, помогающие поддерживать стиль кода: ESLint, TSLint, Prettier

**Линтинг** — это вид статического анализа кода, который часто используют для нахождения проблемных паттернов проектирования или кода, который не следует определённым руководствам по стилю. JavaScript — интерпретируемый язык, поэтому синтаксические и другие ошибки в коде обычно выявляются только после запуска этого кода. Линтеры, наподобие ESLint, позволяют разработчикам находить проблемы в коде, не запуская его.

**TSLint** - это расширяемый инструмент статического анализа, который проверяет код TypeScript на наличие ошибок в удобочитаемости, ремонтопригодности и функциональности. Он может быть настроен с использованием ваших собственных правил, конфигураций и параметров форматирования. На данный момент TSLint не поддерживается, его функционал переносится в ESLint

**ESLint** статически анализирует ваш код, чтобы быстро найти проблемы. ESLint встроен в большинство текстовых редакторов, и вы можете запускать ESLint как часть конвейера непрерывной интеграции. Многие проблемы, обнаруженные ESLint, могут быть автоматически устранены. Исправления ESLint учитывают синтаксис, поэтому вы не столкнетесь с ошибками, вызванными традиционными алгоритмами поиска и замены. Предварительно обработайте код, используйте пользовательские анализаторы и напишите свои собственные правила, которые работают вместе со встроенными правилами ESLint. Настройте ESLint так, чтобы он работал именно так, как вам нужно для вашего проекта.

[**Prettier**](https://prettier.io/)— это средство для форматирования кода, которое нацелено на использование жёстко заданных правил по оформлению программ. Оно форматирует код автоматически.

* Приведение в порядок существующей кодовой базы. Подобное, с помощью Prettier, можно выполнить буквально одной командой. Ручная обработка больших объёмов кода займёт гораздо больше времени
* Prettier легко внедрить. Prettier использует «усреднённый», наименее спорный подход к стилю при форматировании кода. Так как проект это опенсорсный, многие внесли в него вклад, улучшая его и сглаживая острые углы.
* Prettier позволяет сосредоточиться на написании кода, а не на его форматировании. Многие просто не осознают того, как много времени и сил тратится на форматирование кода. Использование Prettier позволяет не думать о форматировании, а заниматься вместо этого программированием.
* Prettier помогает начинающим. Если вы — начинающий программист, работающий в одной команде с серьёзными профессионалами, и вы хотите достойно смотреться на их фоне, в этом вам поможет Prettier.

## 

## 8. Виды и общая характеристика компонентов в React Native. Отличия, область применения.

Компоненты React Native - Это строительные блоки библиотеки react-native, из которых мы создаём интерфейсы для приложений. Компоненты позволяют разделить UI на независимые, повторно используемые части и работать с каждой из них отдельно. React Native похож на React, но он использует собственные компоненты вместо веб-компонентов в качестве строительных блоков.

* **Компоненты-классы.**

*class Welcome extends React.Component {  
 render() {  
 return <h1>Привет, {this.props.name}</h1>;  
 }  
}*

Компонент-классполучает доступ к методам жизненного цикла React, таким render, а также к функциям состояния / реквизита от родительского элемента. Компоненты класса используются в качестве компонентов контейнера для управления состоянием и переноса дочерних компонентов.

* **Компоненты-функции**. Самый простой способ объявить компонент – это написать JavaScript-функцию:

*function Welcome(props) {  
 return <h1>Привет, {props.name}</h1>;  
}*

Функциональные компоненты проще. Они не управляют своим собственным состоянием и не имеют доступа к методам жизненного цикла, предоставляемым React Native. Они буквально являются простыми старыми функциями JavaScript и иногда называются компонентами без состояния. Функциональные компоненты обычно используются только для отображения - эти компоненты вызывают функции из родительских компонентов для обработки пользовательских взаимодействий или обновлений состояния.

## 

## 9. Повторное использование кода в приложении на react Native. Какие методы вы знаете, чтобы разделять общую логику?

Есть несколько способов добиться повторного использования кода и разделения логики, наиболее популярными из которых являются компоненты более высокого порядка, свойства рендеринга, хуки и контекст.

* **Компонент высшего порядка** в React это паттерн, используемый для того, чтобы делить функционал между компонентами без повторения кода. Такие компоненты, по факту, не совсем являются компонентами, это скорее функции. Такая функция берёт компонент как аргумент и отдаёт уже готовый компонент. Она переделывает заданный компонент в другой компонент и добавляет дополнительные данные, либо функционал.
* **Рендеринг** - позволяет рендерить только то, что необходимо. Также за рендер отдельных компонентов могут отвечать разные классы, и тогда пользователю будет показано только то, что необходимо.
* С помощью **хуков** мы можем извлекать состояние компонента, так чтобы его можно было тестировать и переиспользовать. Хуки позволяют повторно использовать логику состояния без изменения иерархии компонентов. Это облегчает обмен ссылками между многими компонентами или всей системы в целом.
* **Вынесение констант** в пропсы позволяет переиспользовать их в нескольких местах.
* **Разделение бизнес-логики и отображения**
  + Бизнес-логика: Код, который принимает решения и изменяет состояния. Обычно всё внутри компонента до инструкции return.
  + Логика отображения: код, который отображает состояние приложения на экране и взаимодействует с пользователем. Все внутри инструкции return.
* **Разделение на файлы.** Разделение монолита на две части открыло широкий простор для нового рефакторинга. Чтобы программа стала еще более читаемой и годной к переиспользованию, можно разделять код на отдельные изолированные модули. Как говорится, чем изолированнее, тем реюзабельнее.

## 

## 10. JSX в React Native. Общая характеристика и примеры использования.

*const element = <h1>Привет, мир!</h1>;*

Синтаксис этого странного тега не является ни строкой, ни HTML. Он называется JSX.   
**JSX (JavaScript eXtension)** - это синтаксическое расширение для JavaScript, которое позволяет использовать элементы HTML в коде JavaScript. Это расширение было разработано компанией Facebook для фреймворка React, но может использоваться и в других проектах. JSX позволяет создавать компоненты интерфейса и описывать их внешний вид в виде элементов HTML, что упрощает создание и поддержку кода

В следующем примере мы объявляем переменную name и затем используем её внутри JSX, обрамляя фигурными скобками:

*const name = 'Иван-Царевич';  
const element = <h1>Здравствуй, {name}!</h1>;*

JSX допускает использование любых корректных JavaScript-выражений внутри фигурных скобок. Например, 2 + 2, user.firstName и formatName(user)

В примере ниже мы встраиваем результат вызова JavaScript-функции formatName(user) в элемент <h1>:

*function formatName(user) {  
 return user.firstName + ' ' + user.lastName;  
}  
const user = {  
 firstName: 'Марья',  
 lastName: 'Моревна'  
};  
const element = (  
 <h1>  
 Здравствуй, {formatName(user)}! </h1>  
);*

После компиляции каждое JSX-выражение становится обычным вызовом JavaScript-функции, результат которого — объект JavaScript.

Из этого следует, что JSX можно использовать внутри инструкций if и циклов for, присваивать переменным, передавать функции в качестве аргумента и возвращать из функции.

*function getGreeting(user) {  
 if (user) {  
 return <h1>Здравствуй, {formatName(user)}!</h1>; }  
 return <h1>Здравствуй, незнакомец.</h1>;}*

Чтобы использовать строковый литерал в качестве значения атрибута, используются кавычки:

*const element = <a href="https://www.reactjs.org"> link </a>;*

Если же в значении атрибута требуется указать JavaScript-выражение, то на помощь приходят фигурные скобки:

*const element = <img src={user.avatarUrl}></img>;*

## 11. Установка React Native и создание каркаса приложения.

**С помощью react-native cli**

React Native - это фреймворк для создания мобильных приложений, основанный на JavaScript и React.js.

Чтобы установить React Native, вам понадобится Node.js, React Native CLI и Xcode (для iOS) или Android Studio (для Android). После установки этих зависимостей можно использовать следующую команду для создания нового проекта:

*react-native init MyProject*

| **Чтобы запустить проект на iOS**, используйте команду:  *cd MyProject react-native run-ios* | **Для запуска на Android:**  *cd MyProject react-native run-android* |
| --- | --- |

После этого вы сможете открыть проект в своем любимом редакторе кода и начать разрабатывать ваше приложение.

**С помощью expo**

**Expo** - это инструментарий, который позволяет быстро и легко создавать приложения React Native без необходимости установки Xcode или Android Studio. Он предоставляет набор инструментов и сервисов, которые помогают разработчикам в процессе создания и отладки приложений.

Чтобы создать новое приложение с помощью Expo, нужно установить Expo CLI и использовать команду:

*npm install -g expo-cli  
expo init MyProject*

Это создаст новый проект, который будет использовать шаблон проекта Expo. Вы также можете использовать команду create-react-native-app для создания проекта без Expo. После создания проекта можно запустить его на эмуляторе или на реальном устройстве, используя команду:

*cd MyProject  
expo start*

Далее выбирается template: blank, blank (TypeScript), tabs, minimal, minimal (TypeScript)

И создаём каркас приложения в App.js

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

*export default function App() {  
 return(  
 <View style={styles.container}>  
 …  
 </View>  
 );  
}  
const styles = StyleSheet.create({container: {flex: 1}});*

## 12. Redux - общая характеристика, назначение, примеры использования.

**Redux** — это инструмент для управления состоянием данных и пользовательским интерфейсом в приложениях JavaScript с большим количеством сущностей. Представляет собой библиотеку JavaScript. Содержит ряд инструментов, позволяющих значительно упростить передачу данных хранилища через контекст. Это помогает вам писать приложения, которые ведут себя согласованно, выполняются в разных средах (клиентской, серверной и собственной) и просты в тестировании. Кроме того, он обеспечивает отличный опыт разработчика, например, редактирование кода в реальном времени в сочетании с перемещающимся во времени отладчиком. Сам по себе Redux очень похож на то, что мы описали с помощью useContext() + useReducer():

* Состояние всего приложения находится в одном хранилище (store)
* Дочерние компоненты оборачиваются в Provider из [react-redux](https://react-redux.js.org/), которому в виде пропа «store» передается хранилище
* Редукторы (reducers) каждой части состояния объединяются с помощью combineReducers() в один корневой редуктор (root reducer), который передается при создании хранилища в createStore()
* Компоненты подключаются к хранилищу с помощью connect() (+ mapStateToProps(), mapDispatchToProps()) и т.д.

Использовать Redux для управления состоянием приложения стало проще, чем комбинацию useContext() + useReducer(), но он создан в основном для крупных приложений.

React Redux включает в себя <Provider />компонент, который делает хранилище Redux доступным для остальной части вашего приложения:

import React from 'react'

import ReactDOM from 'react-dom/client'

import { Provider } from 'react-redux'

import store from './store'

import App from './App'

// As of React 18

const root = ReactDOM.createRoot(document.getElementById('root'))

root.render(

<Provider store={store}>

<App />

</Provider>

)

React Redux предоставляет несколько пользовательских хуков React, которые позволяют вашим компонентам React взаимодействовать с хранилищем Redux.

useSelector считывает значение из состояния хранилища и подписывается на обновления, в то время useDispatch возвращает метод хранилища dispatch, позволяющий отправлять действия.

import React from 'react'

import { useSelector, useDispatch } from 'react-redux'

import {

decrement,

increment,

incrementByAmount,

incrementAsync,

selectCount,

} from './counterSlice'

import styles from './Counter.module.css'

export function Counter() {

const count = useSelector(selectCount)

const dispatch = useDispatch()

return (

<div>

<div className={styles.row}>

<button

className={styles.button}

aria-label="Increment value"

onClick={() => dispatch(increment())}

>

+

</button>

<span className={styles.value}>{count}</span>

<button

className={styles.button}

aria-label="Decrement value"

onClick={() => dispatch(decrement())}

>

-

</button>

</div>

{/\* omit additional rendering output here \*/}

</div>

)

}

## 

## 13. Состояние приложения state. Назначение, примеры использования.

**Состояние** — это собирательное понятие для любой информации, имеющей отношение к приложению. Это могут быть как данные, используемые в приложении, такие как тот же список задач или список пользователей, так и состояние как таковое, например, состояние загрузки или состояние формы. Условно, состояние можно разделить на локальное и глобальное. Под локальным состоянием, обычно, понимается состояние отдельно взятого компонента, например, состояние формы, как правило, является локальным состоянием соответствующего компонента. В свою очередь, глобальное состояние правильнее именовать распределенным или совместно используемым, подразумевая под этим то, что такое состояние используется более чем одним компонентом. Условность рассматриваемой градации выражается в том, что локальное состояние вполне может использоваться несколькими компонентами (например, состояние, определенное с помощью useState(), может в виде пропов передаваться дочерним компонентам), а глобальное состояние не обязательно используется всеми компонентами приложения

Примеры:

*import React, { useState } from 'react';  
function Example() {  
 // Объявление новой переменной состояния «count»   
 const [count, setCount] = useState(0);  
}*

Когда мы хотим отобразить текущее состояние счётчика в классе, мы обращаемся к this.state.count:

*<p>Вы кликнули {this.state.count} раз(а)</p>*

В функции же мы можем использовать count напрямую:

*<p>Вы кликнули {count} раз(а)</p>*

В классе мы вызываем this.setState(), когда надо обновить состояние count:

*<button onClick={() => this.setState({ count: this.state.count + 1 })}>Нажми на меня</button>*

В функции нам не нужен this, потому что setCount и count уже доступны как переменные:

*<button onClick={() => setCount(count + 1)}>Нажми на меня</button>*

## 

## 14. React Native как инструмент создания общей кодовой базы для Android и iOS.

Это кроссплатформенный фреймворк с открытым исходным кодом для разработки нативных мобильных и настольных приложений на JavaScript и TypeScript, созданный Facebook, Inc. React Native поддерживает такие платформы как Android, Android TV, iOS, macOS, tvOS, Web, Windows и UWP, позволяя разработчикам использовать возможности библиотеки React вне браузера для создания нативных приложений, имеющих полный доступ к системным API платформ.

**React Native** – кроссплатформенный язык, и 90% кода, работающего под iPhone, будет работать и на Андроид-устройствах.

Для создания примитивных приложений на базе React Native можно использовать утилиту Expo cli – это отдельное приложение для iOS и Android, содержащее в себе все React-компоненты, встроенные во фреймворк по умолчанию.

Второй вариант: настраивать полноценное окружение для создания программы на React Native.

**Настройка для iOS**

* Скачиваем зависимые пакеты
* Загружаем и настраиваем Xcode

Естественно, необходима node.js. Без нее не получится вести разработку на JavaScript. Устанавливаем ее командой brew install node

Добавляем пакет Watchmen. Это инструмент для отслеживания изменений в файловой системе, разработанный Facebook. Корпорация обещает, что Watchmen может заметно ускорить производительность всего проекта на базе React-Native.

Следом скачаем CocoaPods. Это менеджер библиотек для Xcode, без которого он нормально не функционирует. Смело вводим команду sudo gem install cocoapods и ждем.

**Настройка для Android**

* Сначала скачиваем менеджер пакетов Chocolatey.
* Нам нужна среда для разработки Android Studio.

После установки зависимостей Facebook рекомендует использовать встроенную утилиту для управления React-Native-приложениями через командную строку. Поэтому для создания нового шаблонного проекта на базе RN введем команду npx react-native init <название проекта>

После запуска приложения при помощи команды npx react-native run-ios (для iOS) перед нами появится интерфейс симулятора. В macOS он связан с Xcode, а в Windows, macOS и Linux при разработке под Android – c Android Studio.

## 

## 15. SetState — механизм использования, примеры.

**React** — это компонентно-ориентированная UI библиотека. В основе каждого компонента лежит функция, которая принимает некоторые свойства и возвращает UI элемент. Компонент может иметь состояние, и в этом случае должен уметь управлять им. В этом случае, компонент обычно объявляется как класс, а его состояние объявляется в его функции-конструкторе. Чтобы управлять состоянием, React предоставляет специальный метод **setState()**. Функциональный **setState** — это изменение состояния компонента отдельно от объявления его класса.

Как работает **setState()**: ему передается объект, содержащий часть информации, которую необходимо обновить. Другими словами, значения в передаваемом объекте будут сначала сравниваться со значениями из состояния компонента и, после, setState() либо обновит, либо добавит новые значения в state.

*changeColor = () => {this.setState({color: "blue"});}*

**setState()** также принимает функцию. Функция в своих аргументах принимает текущий state, а также props(свойства) компонента, которые используются для вычислений следующего состояния.

*this.setState(function(state, props) {  
 return {  
 score: state.score - 1  
 }  
});*

Первым делом React объединяет уже имеющийся state с объектом, передаваемым в setState(). Затем запускает процесс синхронизации с DOM и создает новое дерево элементов виртуального ReactDOM, сравнивает его с предыдущем деревом, окончательно выясняет, что случилось и как это сделать наиболее безболезненно и после происходит финальное обновление DOM в браузере.

Пример использования: *this.setState({comment: 'Hello'});* (единственным местом, где вы можете назначить *this.state* является конструктор.)

Вызовы setState являются асинхронными, когда они находятся внутри обработчиков событий, и когда вы не полагаетесь на this.state, чтобы отобразить новое значение сразу после вызова setState. Передача функции обновления позволяет получить доступ к значению текущего состояния внутри программы обновления. Поскольку вызовы setState являются пакетными, это позволяет вам связывать обновления и гарантировать, что они строятся друг на друге, а не конфликтуют:

*incrementCount() {*

*this.setState((state) => {*

*// Важно: читайте `state` вместо` this.state` при обновлении.*

*return {count: state.count + 1}*

*});*

*}*

*handleSomething() {*

*// Допустим, `this.state.count` начинается с 0.*

*this.incrementCount();*

*this.incrementCount();*

*this.incrementCount();*

*// Если вы сейчас прочитаете `this.state.count`, оно все равно будет равно 0.*

*// Но когда React повторно выполнит рендеринг компонента, это будет 3.*

*}*

## 16. Кросс-платформенная разработка приложений?

**Кроссплатформенная разработка** — это когда мы один раз пишем универсальный код, который потом превращается в приложение и на iOS, и на Android. Реализуется это так:

1. Один раз пишем кроссплатформенный код.
2. При компиляции выбираем, для какой платформы адаптировать код.
3. Кроссплатформенный фреймворк сам меняет свои команды на те, которые приняты на выбранной платформе.
4. Итоговый код пригоден для запуска на выбранной платформе.

Можно поставить сразу две галочки и скомпилировать код сразу под iOS и Android — в этом случае фреймворк по очереди преобразует универсальные команды в нужный код и мы получим сразу два приложения. В мобильной разработке есть два кроссплатформенных фреймворка — React Native и Flutter. Они написаны на разных языках и работают по-разному. React Native — это развитие библиотеки React, которую используют для создания веб-интерфейсов. Сначала React Native использовали для iOS-приложений, но быстро сообразили, что из него можно сделать универсальный инструмент, и добавили поддержку Android. Когда код на React Native компилируется под конкретную платформу, все команды на JavaScript преобразуются во внутренние инструкции операционной системы. За это отвечает движок JavaScriptCore — его использует мобильный браузер Safari.

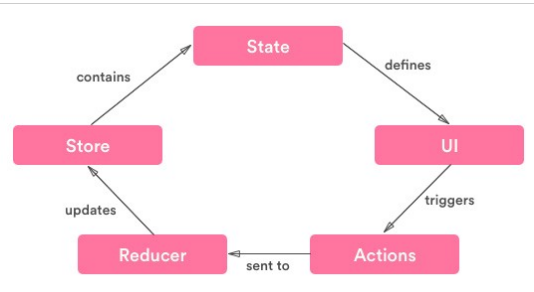
## 

## 17. Redux в React Native. Основные компоненты Redux в приложении React Native.

**Redux** — это инструмент управления состоянием. С помощью Redux все состояние вашего приложения сохраняется и может быть доступно из всех компонентов.

Redux можно разбить на следующие разделы, которые используются в приложении:

* **Действия:** используются для отправки данных (информации о полезной нагрузке) из вашего приложения в хранилище, т. е. если вы вносите какие-либо изменения в данные, которые будут отправлены посредством действия и обновлены в магазине. Также есть только одна точка доступа для хранения, доступ к которой возможен только с помощью действий.
* **Редюсеры:** после действия следующим в очереди являются редукторы, основная роль которых заключается в обновлении необходимых изменений в состояниях и возврате нового состояния с обновленным значением.
* **Магазин:** с помощью редукторов можно создать хранилище, в котором хранится все состояние приложения. Рекомендуется использовать одно хранилище для всего приложения, а не несколько хранилищ, которые нарушат использование избыточности, которая имеет только одно хранилище.
* **Компоненты:** где вы сохранили свой пользовательский интерфейс (UI).
* **Промежуточное ПО:** это функции, которые позволяют нам расширять функциональность Redux в нашем приложении. Промежуточное ПО находится между действием отправки и редюсером, что означает, что мы можем выполнить какую-то другую функцию до того, как диспетчеризация достигнет редюсера. В нашем примере проекта мы используем регистратор в качестве промежуточного программного обеспечения, которое предоставит нам подробную информацию о предыдущем состоянии, триггере действия и обновленном состоянии.



## 

## 18. UI-библиотека React Native — назначение, преимущества.

**React Native** (также известный как RN) - это популярная платформа мобильных приложений на основе [JavaScript](https://ru.wikipedia.org/wiki/JavaScript), которая позволяет создавать мобильные приложения с собственным интерфейсом для iOS и Android. Фреймворк позволяет создавать приложения для различных платформ, используя [одну и ту же кодовую базу.](https://www.netguru.com/blog/react-native-faq)

React Native был впервые выпущен Facebook в качестве проекта с открытым исходным кодом в 2015 году. Всего за пару лет он стал одним из лучших решений, используемых для мобильной разработки. Разработка React Native используется для поддержки некоторых ведущих мобильных приложений в мире, включая Instagram, Facebook и Skype.

React Native был построен на основе React - библиотеки [JavaScript](https://ru.wikipedia.org/wiki/JavaScript), которая уже была чрезвычайно популярна, когда была выпущена мобильная платформа.

Фреймворк позволил разработчикам интерфейсов, которые раньше могли работать только с веб-технологиями, создавать надежные, готовые к работе приложения для мобильных платформ.

**Преимущества:**

* Hot Reloading. Дает возможность вносить изменения в код, которые мгновенно отображаются на девайсе или симуляторе без рекомпиляции измененных файлов.
* Декларативный UI. Наверное, каждый, кто работал с React Native или Swift UI, согласится: это удобный подход к написанию UI. Можно не указывать как создавать элементы или как они связаны между собой. Шагов нет, только результат, которого хотим
* Code push. Этот облачный сервис позволяет развертывать обновления приложений непосредственно на устройствах пользователей без прохождения ревью. Правда, это оправдано только при изменениях в JS. При обновлении нативного кода ревью все же потребуется. Также можно сделать rollback к предыдущей версии.
* Stories & Storybook. Предоставляет возможность создавать библиотеки UI-компонентов (текст-филды, баттоны, свитчеры) и хранить их в любом хранилище. К веб-интерфейсу будут подтягиваться все созданные компоненты. И в нем же сможете интерактивно изменять их параметры. Полезно в случае будущих модификаций в дизайне продукта.
* Большое сообщество пользователей React Native. Сегодня это самый популярный гитхабный репоз — у него 105 000 звездочек! Фактически развитие React Native держится именно на этом сообществе.

## 

## 19. Особенности разработки приложений React Native с использованием IDE.

Выбор правильного инструмента разработки для React Native имеет решающее значение для успешного создания будущих проектов. Можно использовать как IDE, так и инструменты непосредственно предназначенные для React Native.

* [Expo](https://expo.io/) — это один из лучших инструментов разработки приложений для React Native. Он предоставляет набор инструментов и сервисов, которые помогают быстро и эффективно разрабатывать и запускать приложения на React Native. Одна и та же база кода может работать на нескольких платформах, и это очень удобно при разработке кроссплатформенных приложений, включая iOS и Android. Этот инструмент разработки работает по принципу server, share, build & polish (сервер, общий доступ, сборка и отладка), который считается основой разработки программного обеспечения.
* Можно использовать Android Studio. [Android Studio](https://developer.android.com/studio) - это интегрированная Среда Разработки для создания нативных приложений для Android. Android studio также позволяет разрабатывать приложения для React native с помощью “версии Xcode” для Android. Существует множество преимуществ использования этой IDEвместо аналогов для разработки приложений.
* Есть React Native CLI. [React Native CLI](https://github.com/react-native-community/cli) React Native CLI-это официальный инструмент разработки приложений для React Native. Автоматизация заданий, включая связывание, совместное использование и создание приложения, очень просто реализовать в React Native CLI. Этот инструмент разработки ReactNative очень полезен для программистов. Вы можете использовать CLI для установки проектов, предварительно настроенных с помощью определенного пакета.
* Еще хорошим решением можно назвать [Nuclide](https://nuclide.io/docs/platforms/react-native/). [Nuclide](https://nuclide.io/docs/platforms/react-native/) - это полный пакет для фреймворка React Native, интегрированный в редактор кода Atom. Вам нужно установить пакет Nuclide в Atom, потому что Nuclide имеет встроенную поддержку разработки приложений React Native. Он имеет множество встроенных функций, как и любой другой инструмент разработки, для удобства в отладке и поддержки кода.

IDE можно использовать, но лучше использовать Expo, который специально сделан для React Native. Можно выбрать связку [VS Code](https://code.visualstudio.com/) в качестве основной среды для написания javascript кода, [Android Studio](https://developer.android.com/studio/index.html) для написания нативных модулей под Android и [AppCode](https://www.jetbrains.com/objc/download/) для написания нативных модулей под iOS (в качестве альтернативы можно взять XCode от самой Apple).

## 

## 20. Свойства (props) в приложении React Native. Назначение, примеры.

**Props** — это список атрибутов, передаваемые в компоненты React. Свойства передаются компонентам через атрибуты HTML. Компонент функции React получает этот список в качестве своего первого аргумента. Список передается как объект с ключами, представляющими имена атрибутов, и значениями, представляющими присвоенные им значения. Чтобы отправить props в компонент, используйте тот же синтаксис, что и атрибуты HTML.

Самый простой способ определить компонент — написать JavaScript-функцию:

*function Welcome(props) {  
 return <h1>Привет, {props.name}</h1>;  
}*

Данная функция — корректный компонент React, потому что он принимает один аргумент-объект «props» (который обозначает свойства) с данными и возвращает элемент React. Такие компоненты мы называем «функциональными», потому что они являются буквально функциями JavaScript.

Вы также можете использовать [класс из ES6](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Classes) для определения компонента:

*class Welcome extends React.Component {  
 render() {  
 return <h1>Привет, {this.props.name}</h1>;  
 }  
}*

Два вышеуказанных компонента эквивалентны с точки зрения React.

## 

## 21. Компонент Scrollview. Назначение, примеры.

**ScrollView**: Компонент ScrollView - это встроенный компонент react-native, который служит универсальным прокручиваемым контейнером с возможностью прокрутки дочерних компонентов и представлений внутри него.

**Когда использовать ScrollView?**

ScrollView загружает все содержимое, то есть данные, которые будут отображаться на экране одновременно. Это делается сразу после загрузки компонента. Таким образом, весь контент (список данных) монтируется целиком. Теперь, если этот список данных содержит много элементов, это автоматически вызовет проблемы с производительностью. Поэтому не рекомендуется использовать ScrollView, если у вас есть сто или тысяча элементов, которые должны отображаться на экране. Он используется, когда у вас меньше элементов данных в списке ограниченного размера.

**Пример**

*import { ScrollView } from 'react-native';  
const MyComponent = () => {  
 return (  
 <ScrollView>  
 <View>  
 <Text>Content 1</Text>  
 <Text>Content 2</Text>  
 <Text>Content 3</Text>  
 <Text>Content 4</Text>  
 <Text>Content 5</Text>  
 <Text>Content 6</Text>  
 </View>  
 </ScrollView>  
 );  
}*

В приведенном примере внутри ScrollView может быть много контента, который не может быть помещен на экран одновременно. Пользователи могут прокручивать содержимое для просмотра всего контента.

## 

## 22. Функциональные и классовые компоненты в приложении React Native.

Для создания компонентов React не обязательно использовать классы. В тех случаях, когда у компонента нет состояния, гораздо проще использовать функциональные компоненты.

Компоненты, созданные как функции, называются функциональными. Они принимают объект со свойствами как первый аргумент, и также начинаются с большой буквы. Функциональные компоненты нужно использовать, когда компонент не хранит в себе состояние. Другими словами, большинство компонентов в проекте должно быть именно функциональными. В остальном они ведут себя точно так же, как и компоненты на классах.

## 

## 23. Методы жизненного цикла компонентов в React Native.

**Жизненный цикл React 16.3.0**

1. Монтаж

* *constructor()* - Как правило, в React конструкторы используются только для двух целей:
  + Инициализация [local state] путем присвоения объекта this.state
  + Привязка методов обработчика события к экземпляру
* *static getDerivedStateFromProps()* - вызывается непосредственно перед вызовом render(). Он должен вернуть объект для обновления состояния или null, чтобы ничего не обновлять. Этот метод существует для редких случаев использования, где состояние зависит от изменений в свойствах со временем.
* *render()* - единственный обязательный метод в классовом компоненте. При вызове должен проверять this.props и this.state и возвращать один из следующих типов
  + react элемент
  + массивы и фрагменты
  + порталы
  + строки и числа
  + логические значения и null
* *componentDidMount()* - Вызывается сразу после монтирования компонента в DOM-дерево. Вся инициализация, требующая dom-узлов, должна быть прописана в этом методе. Если есть необходимость загружать данные из удаленного endpoint, в этом методе стоит создать экземпляр сетевого запроса

1. Обновление

* *static getDerivedStateFromProps()*
* *shouldComponentUpdate()* - вызывается непосредственно перед тем, как последний отрисованный вывод будет зафиксирован, например, в DOM. Позволяет компоненту захватывать информацию из DOM прежде чем она будет изменена. Любое значение, возвращаемое этим методом, будет передано как параметр componentDidUpdate()
* *render()*
* *getSnapshotBeforeUpdate()* - Вызывается прямо перед изменениями из VDOM, которые должны быть отображены в DOM. Ее можно использовать в основном, если вам нужно прочитать текущее состояние DOM. Например у вас есть приложение, в котором новые сообщения добавляются сверху экрана – если пользователь будет скроллить вниз, и добавится новое сообщение, экран будет «прыгать» и это сделает UI тяжелее в использовании. Добавлением getSnapshotBeforeUpdate вы сможете рассчитать текущее положение скролла и восстанавливать его через апдейт DOM-а.
* *componentDidUpdate()* - Эта функция будет вызываться после render() в каждом цикле перерисовки. Это означает, что вы можете быть уверены, что компонент и все его дочерние компоненты уже перерисовали себя. В связи с этим эта функция является единственной функцией, что гарантировано будет вызвана только раз в каждом цикле перерисовки, поэтому любые сайд-эффекты рекомендуется выполнять именно здесь. Как componentWillUpdate и componentWillRecieveProps в эту функцию передается предыдущие props, состояние (state) и контекст, даже если в этих значениях не было изменений.

1. Демонтаж

* *componentWillUnmount()* - Вызывается непосредственно перед размонтированием и уничтожением компонента. В этом методе лучше сделать всю необходимую очистку, такую как отмена таймеров, сетевых запросов. В этом методе нельзя вызывать setState(), потому что компонент никогда не будет повторно отрисовываться. Как только экземпляр будет размонтирован, его больше никогда не примонтируют

## 

## 24. Управление состоянием в приложении React Native.

**Состояние** — это собирательное понятие для любой информации, имеющей отношение к приложению. Под локальным состоянием, обычно, понимается состояние отдельно взятого компонента, например, состояние формы. В свою очередь, глобальное состояние правильнее именовать распределенным или совместно используемым, подразумевая под этим то, что такое состояние используется более чем одним компонентом. Условность рассматриваемой градации выражается в том, что локальное состояние вполне может использоваться несколькими компонентами (например, состояние, определенное с помощью useState(), может в виде пропов передаваться дочерним компонентам), а глобальное состояние не обязательно используется всеми компонентами приложения

Хук «useState()» предназначен для управления локальным состоянием компонента. Он возвращает массив с двумя элементами: текущим значением состояния и сеттером — функцией для обновления этого значения. Основные проблемы данного подхода к управлению состоянием:

* Необходимость передачи состояния и/или сеттера на каждом уровне вложенности, обусловленная локальным характером состояния
* Логика обновления состояния приложения разбросана по компонентам и смешана с логикой самих компонентов
* Сложность обновления состояния, вытекающая из его иммутабельности
* Однонаправленный поток данных, невозможность свободного обмена данными между компонентами, находящимися на одном уровне вложенности, но в разных поддеревьях виртуального DOM

Первые две проблемы можно решить с помощью комбинации useContext()/ useReducer().

Контекст (context) позволяет передавать значения дочерним компонентам напрямую, минуя их предков. Хук «useContext()» позволяет извлекать значения из контекста в любом компоненте, обернутом в провайдер (provider). Хук «useReducer()» принимает редуктор (reducer) и начальное состояние. Он возвращает значение текущего состояния и функцию для отправки (dispatch) операций (actions), на основе которых осуществляется обновление состояния. Алгоритм обновления состояния выглядит так: компонент отправляет операцию в редуктор, а редуктор на основе типа операции (action.type) и опциональной полезной нагрузки операции (action.payload) определенным образом изменяет состояния. Результатом комбинации useContext() и useReducer() является возможность передачи состояния и диспетчера, возвращаемых useReducer(), любому компоненту, являющемуся потомком провайдера контекста. Сам по себе Redux очень похож на то, что мы описали с помощью useContext() + useReducer():

* Состояние всего приложения находится в одном хранилище (store)
* Дочерние компоненты оборачиваются в Provider из [react-redux](https://react-redux.js.org/), которому в виде пропа «store» передается хранилище
* Редукторы (reducers) каждой части состояния объединяются с помощью combineReducers() в один корневой редуктор (root reducer), который передается при создании хранилища в createStore()
* Компоненты подключаются к хранилищу с помощью connect() (+ mapStateToProps(), mapDispatchToProps()) и т.д.

Использовать Redux для управления состоянием приложения стало проще, чем комбинацию useContext() + useReducer(), но он создан в основном для крупных приложений. Альтернатива для небольших приложений - Recoil. В основе Recoil лежат атомы и селекторы. Атом — это часть состояния, а селектор — часть производного состояния. Атомы создаются с помощью функции «atom()», а селекторы — с помощью функции «selector()». Для извлечение значений из атомов и селекторов используются хуки «useRecoilState()» (для чтения и записи), «useRecoilValue()» (только для чтения), «useSetRecoilState()» (только для записи) и др. Компоненты, использующие состояние Recoil, должны быть обернуты в RecoilRoot.

## 

## 25. Redux store. Назначение, применение, примеры использования.

**Redux** - это предсказуемое хранилище состояния для JavaScript приложений. (Не путайте с WordPress фреймворком – Redux Framework.). Он помогает вам писать приложения, которые ведут себя предсказуемо, работают в разных окружениях (клиентских, серверных и нативные приложения) и которые легко тестировать.

**Redux Store**. Хранилище — это ваш контейнер, в котором живет состояние. Хранилище — объект со своим API, который позволяет взаимодействовать с вашим состоянием: изменять его, запрашивать его значение, и так далее.

*const store = new Store(reducers, initialState);*

Пояснение => **Редюсер**— просто чистая функция, принимающая состояние state нашего приложения (наше внутреннее дерево состояния, переданное хранилищем в редюсер), а вторым аргументом «прилетевшее» действие action.

**Store.dispatch()**

Метод dispatch позволит передать инструкцию для Store, что мы хотим изменить дерево состояния. Это обрабатывается с помощью только что рассмотренного редюсера.

**Store.subscribe()**

Метод subscribe позволит передать в Store функцию-подписчик, которой, когда наше дерево состояния изменится, мы сможем передать эти его новые изменения с помощью аргумента в обратном вызове .subscribe()

**Store.value**

Свойство value создают как геттер и будет возвращать внутреннее дерево состояния (так мы сможем обращаться к свойством)

## 

## 26. Практическое использование методов shouldComponenUpdate()

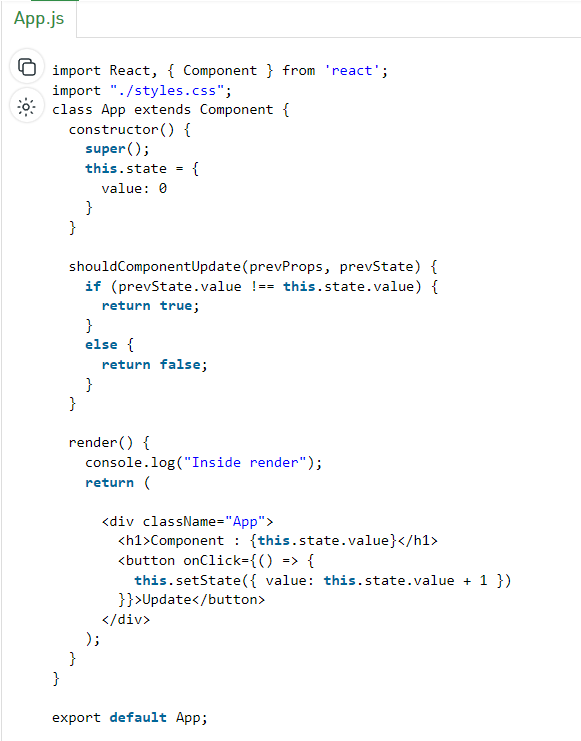
**shouldComponentUpdate** - это метод жизненного цикла в React. Этот метод заставляет компонент повторно отображать только тогда, когда происходит изменение состояния или реквизитов компонента, и это изменение повлияет на результат.

**shouldComponentUpdate()** вызывается перед визуализацией уже смонтированного компонента при получении новых реквизитов или состояний.

**Синтаксис:** shouldComponentUpdate(nextProps, nextState)

**Параметры:** он принимает два параметра с именами *nextProps* и *nextState*. shouldComponent возвращает логическое значение того, возвращать компонент или нет, путем сравнения существующего prop и state с nextProps и nextState .

**Возвращаемое значение:** этот метод по умолчанию возвращает true, что означает, что компонент будет повторно отображаться, и если он возвращает false, то методы render(), componentWillUpdate() и componentDidUpdate() не вызываются.



**Объяснение:** Выше приведен пример, в котором при нажатии кнопки значение состояния изменяется. Внутри функции shouldComponentUpdate() мы проверяем, отличается ли наше текущее значение состояния от предыдущего. Если оно отличается, то функция возвращает true, что означает, что компонент будет повторно отображаться. В консоли мы видим, что при каждом нажатии кнопки появляется сообщение “внутри рендеринга”, поскольку компонент повторно рендерится снова и снова. В случае, если текущее значение состояния такое же, как и раньше, функция shouldComponentUpdate() вернет false и компонент не будет повторно отображаться.

**Важность shouldComponentUpdate():**

* Это помогает проверить, требуется ли повторный рендеринг компонентов или нет. Если повторный рендеринг не требуется, то shouldComponentUpdate не должен отображать компонент. Например, если мы хотим, чтобы наш компонент не отображался при каком-то определенном условии, то shouldComponentUpdate может быть очень полезен.
* Это помогает повысить производительность.
* Это повышает скорость реагирования и оптимизацию веб-сайта

## 

## 27. Сравнительная характеристика компонентов Scrollview и FlatList.

**ScrollView**: Компонент ScrollView - это встроенный компонент react-native, который служит универсальным прокручиваемым контейнером с возможностью прокрутки дочерних компонентов и представлений внутри него.

**Когда использовать ScrollView?**

ScrollView загружает все содержимое, то есть данные, которые будут отображаться на экране одновременно. Это делается сразу после загрузки компонента. Таким образом, весь контент (список данных) монтируется целиком. Теперь, если этот список данных содержит много элементов, это автоматически вызовет проблемы с производительностью. Поэтому не рекомендуется использовать ScrollView, если у вас есть сто или тысяча элементов, которые должны отображаться на экране. Он используется, когда у вас меньше элементов данных в списке ограниченного размера.

**FlatList**: Компонент FlatList - это встроенный компонент react-native, который отображает данные аналогичной структуры в прокручиваемом списке. Он показывает только те элементы, которые в данный момент отображаются на экране

**Когда использовать плоский список?**

В отличие от ScrollView, плоский список отображает только те элементы, которые в данный момент отображаются на экране (по умолчанию: 10 элементов). Таким образом, это не оказывает никакого влияния на производительность приложения. Таким образом, предпочтительно использовать компонент Flat List для отображения большого списка данных.



## 

## 28. Хуки в приложении React Native. Общий механизм работы.

А зачем нам вообще нужны классы в React? Обычно затем, чтобы работать с локальными состояниями и методами жизненного цикла. Хуки позволяют работать с ними сразу — без написания классов.

Например, хук useState отвечает за состояние. Рассмотрим пример его использования.

*const [name, setName] = useState(‘’)*

При монтировании компонента в переменной *name* будет храниться пустая строка, а с помощью функции setName можно изменить это значение. Вызываем *setName(‘Андрей’)* — и переменная *name* становится равна ‘Андрей’.

Хук эффекта useEffect отвечает за методы жизненного цикла. В компоненте их может быть несколько, поэтому можно распределить логику в его разные части. Например, можно вызвать 3 хука, аналогичных методу *componentDidMount()*.



Однако самое важное — **можно создавать собственные хуки**. Это позволяет избежать компонентов высшего порядка. К примеру, вы можете написать хук useAuth: в нём происходит обработка данных и определяется, авторизованы ли вы на сайте. После этого нужно вызвать useAuth внутри компонента, присвоить результат какой-то переменной — и можете дальше спокойно использовать это значение.

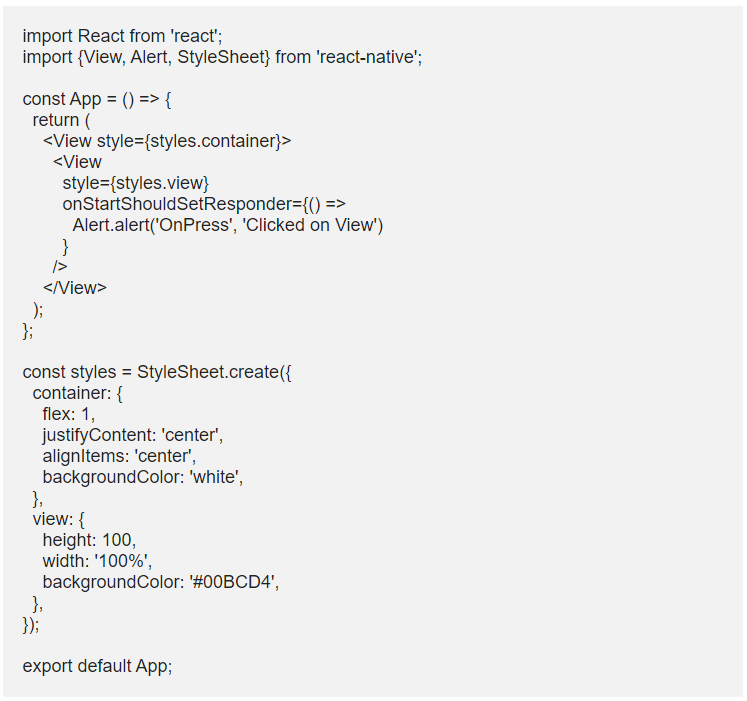
## 

## 29. Функция onPress в React Native View.

В react native мы используем onPress в осязаемых компонентах, чтобы включить функцию нажатия. Но как насчет наличия функции onPress для самого компонента View?

Если вы используете react native версии больше 0.55.3, то вы можете добавить функциональность onPress к любому компоненту представления, используя onStartShouldSetResponder.

В следующем примере react native view onPress компонент view с OnStart, который должен установить Responder, который вызывает функцию оповещения. Это означает, что оповещение появится при нажатии кнопки.



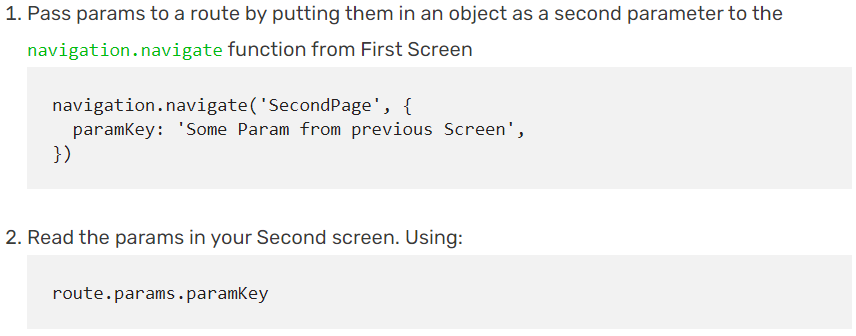
(обрезала styles, так как к теме не относится)

а не надо обрезать, надо копировать

## 

## 30. Передача информации между экранами в React Native.

Передавать данные между различными экранами можно, используя *this.props.navigation.navigate()*, который используется для навигации между различными экранами. С помощью того же метода можно передать значение и получить его с других экранов. В примере ниже мы передадим входные данные, взятые с первого экрана, на второй экран.



## 

## 31. Какие способы вы знаете для обработки состояния приложения без Redux?

Существует несколько способов управления состоянием без Redux. Это можно делать через хуки:

* **useState():** До появления хуков у функциональных компонентов не было возможности задавать локальный стейт. Ситуация изменилась с появлением useState(). Это выглядит примерно так:

*const [state, setState] = useState(0);*

Данный вызов возвращает массив. Первым его элементом является переменная, предоставляющая доступ к значению стейта. Второй элемент — это функция, которая обновляет стейт и перерисовывает компонент для отражения изменений.

* **useEffect():** Классовые компоненты реагируют на сайд-эффекты, используя lifecycle-методы, такие как componentDidMount(). Хук useEffect() позволяет делать то же самое в функциональных компонентах. По умолчанию эффекты запускаются после каждой перерисовки. Но можно сделать так, чтобы они выполнялись только после изменения значений конкретных переменных, передавая их вторым опциональным параметром в виде массива:

*useEffect(() => {  
 console.log('Я вызовусь только при изменении valueA');  
}, [valueA]);*

## 

## 32. Маршрутизация в приложениях React Native. (с навигацией?)

Мобильные приложения редко состоят из одного экрана. Управление представлением нескольких экранов и переходом между ними обычно осуществляется с помощью так называемого навигатора. React Navigation предоставляет простое навигационное решение с возможностью представления общей навигации по стеку и шаблонов навигации с вкладками как на Android, так и на iOS. На обеих платформах это реализуется с помощью библиотеки react-native-navigation.

Переходом между несколькими экранами управляют навигаторы. React Navigation позволяет использовать различные типы навигаторов, такие как стековые навигаторы, навигаторы ящиков, навигаторы вкладок и т.д. Наряду с навигацией между несколькими экранами, он также может использоваться для обмена данными между ними.

Для навигации между экранами можно использовать StackNavigator. StackNavigator работает точно так же, как стек вызовов. Каждый экран, на который переходит пользователь, помещается на самый верх стека. Каждый раз, когда нажимается кнопка назад, экраны удаляются с верхней части стека.

Реализация:

*import { NavigationContainer } from '@react-navigation/native';  
import { createStackNavigator } from '@react-navigation/stack';*

*const Stack = createStackNavigator();*

*class App extends React.Component {  
 render() {  
 return (  
 <NavigationContainer>  
 <Stack.Navigator>  
 <Stack.Screen  
 name="Home"  
 component={HomeScreen}  
 />  
 <Stack.Screen  
 name="Friends"  
 component={FriendsScreen}  
 />  
 </Stack.Navigator>  
 </NavigationContainer>  
 );  
 }  
}*

Пример использования (в компоненте HomeScreen):

*<Button  
 title="Add some friends"  
 onPress={() =>  
 this.props.navigation.navigate('Friends')  
 }  
 />*

## 33. Универсальные компоненты React Native.

При Android-разработке представления пишутся на Kotlin или Java; при разработке для iOS используется Swift или Objective-C. С помощью React Native можно вызывать эти представления с помощью JavaScript, используя компоненты React. Во время выполнения React Native создает соответствующие представления Android и iOS для этих компонентов. Поскольку компоненты React Native поддерживаются теми же представлениями, что Android и iOS, приложения React Native выглядят, ощущаются и работают так же, как и любые другие приложения. Компоненты, поддерживаемые платформой, называются собственными компонентами.

React Native содержит множество основных компонентов для всего - от элементов управления до индикаторов активности. Все их можно найти в документации API. В основном нам приходилось работать со следующими компонентами:

* <View> - это контейнер, который поддерживает компоновку с гибким интерфейсом, стили, некоторые элементы управления касанием и элементы управления доступом.
* <Text> - отображает, стилизует и вкладывает строки текста и даже обрабатывает события касания
* <Image> - отображает различные типы изображений
* <ScrollView> - универсальный контейнер для прокрутки, который может содержать несколько компонентов и представлений
* <TextInput> - позволяет пользователю вводить текст. К него есть некоторые дополнительные свойства, такие как автозамена, текст-заполнитель, автокапитализация и автозаполнение (на Android). Значение, получаемое из TextInput всегда представляют собой строку.
* <FlatList> - компонент, который отображает элементы в прокручиваемом представлении, например список стран, штатов или провинций.
* <ActivityIndicator> - отображение анимированного индикатора загрузки.
* <ImageBackground> - основной способ отображения фонового изображения.

## 

## 34. Платформенно-специализированные компоненты React Native.

При Android-разработке представления пишутся на Kotlin или Java; при разработке для iOS используется Swift или Objective-C. С помощью React Native можно вызывать эти представления с помощью JavaScript, используя компоненты React. Во время выполнения React Native создает соответствующие представления Android и iOS для этих компонентов. Поскольку компоненты React Native поддерживаются теми же представлениями, что Android и iOS, приложения React Native выглядят, ощущаются и работают так же, как и любые другие приложения. Компоненты, поддерживаемые платформой, называются собственными компонентами.

React Native также позволяет создавать собственные нативные компоненты для Android и iOS в соответствии с уникальными потребностями приложения.

Компоненты для Android:

* **BackHandler** - обнаруживает нажатия аппаратных кнопок для обратной навигации, позволяет регистрировать прослушиватели событий для действий системы и позволяет контролировать реакцию приложения.
* **PermissionsAndroid** - предоставляет доступ к новой модели разрешений Android M. Так называемые "обычные" разрешения предоставляются по умолчанию. Однако для "опасных" разрешений требуется диалоговое приглашение. Можно использовать этот модуль для этих разрешений.
* **ToastAndroid** - API ToastAndroid от React Native предоставляет модуль ToastAndroid платформы Android в виде JS-модуля.

Компоненты для IOS:

* **ActionSheetIOS** - Отображает встроенный в iOS компонент ActionSheet.

## 

## 35. Сравнительная характеристика props и state в React Native.

* **Props** - это сокращение от properties, и они используются для передачи данных между компонентами React. Поток данных React между компонентами является однонаправленным. Хотя слово “props” расшифровывается как “свойства”, оно больше похоже на атрибуты в HTML-тегах и работает как аргументы для функций. Используя props, можно передавать данные из родительского представления вниз в дочернее представление.

Пример:

*<TextInputView*

*placeholder="sample@gmail.com"*

*label="Email"*

*value={this.state.email}*

*onChangeText={email => this.setState({ email })}*

*/>*

Этот компонент принимает от юзера введенный логин.

* В React есть еще один специальный встроенный объект, называемый **state**, который позволяет компонентам создавать свои собственные данные и управлять ими. Таким образом, в отличие от props, компоненты не могут передавать данные с состоянием, но они могут создавать и управлять ими. State используется для данных, которые собираются измениться. Его роль очень похожа на свойства класса в ООП. Именно поэтому оно используется для классовых компонентов.
  + state - это объект JS, который содержит данные, относящиеся к компоненту
  + обновление state приводит к повторному рендеру компонента
  + state должно быть инициализировано при создании компонента
  + state может быть обновлено методом setState()

То есть, props и state - это простые объекты JavaScript. Хотя оба содержат информацию, которая влияет на результат рендеринга, они отличаются одним важным способом: props передаются компоненту (аналогично параметрам функции), тогда как state управляется внутри компонента (аналогично переменным, объявленным внутри функции). Основная идея совместного использования “state” и “props” состоит в том, чтобы хранить временные данные в “state” и передавать их своим дочерним элементам через “props”.

## 

## 36. Способы создания стилей в React Native.

Стили в React Native - это не то же самое, что стили в CSS. Для стилизации элементов в React Native используются объекты JavaScript. Каждый основной компонент в React Native принимает prop style, который принимает объект JavaScript, содержащий имена свойств CSS, в качестве ключа. Например, любое свойство CSS, используемое в этих объектах, следует соглашению camel case. backgroundColor используется вместо background-color.

Способы создания:

* Использование встроенных стилей: использование встроенных стилей предпочтительно только для небольших, очень маленьких компонентов. Для крупномасштабных производственных приложений использование встроенных стилей становится громоздким и неэффективным.

*<View  
 style={{  
 height: 550,  
 margin: 20  
 }}></View>*

* Использование таблицы стилей: по мере усложнения компонента использование StyleSheet.create становится намного чище и эффективнее для определения нескольких стилей в одном месте.

*<View style={styles.product}></View>*

*const styles = StyleSheet.create({  
 product: {  
 height: 500,  
 margin: 20  
}})*

* Styled Components — это библиотека для React и React Native для написания и управления CSS. Это решение «CSS-in-JS», то есть вы пишете CSS в файлах Javascript (в частности, в компонентах, которые являются файлами Javascript).

*const Post = styled.View`  
padding: 15px;  
background-color: red;  
`*

## 

## 37. Навигация в приложениях React Native

Мобильные приложения редко состоят из одного экрана. Управление представлением нескольких экранов и переходом между ними обычно осуществляется с помощью так называемого навигатора. React Navigation предоставляет простое навигационное решение с возможностью представления общей навигации по стеку и шаблонов навигации с вкладками как на Android, так и на iOS. На обеих платформах это реализуется с помощью библиотеки react-native-navigation.

Переходом между несколькими экранами управляют Навигаторы. React Navigation позволяет использовать различные типы навигаторов, такие как стековые навигаторы, навигаторы ящиков, навигаторы вкладок и т.д. Наряду с навигацией между несколькими экранами, он также может использоваться для обмена данными между ними.

Для навигации между экранами вы будете использовать StackNavigator. StackNavigator работает точно так же, как стек вызовов. Каждый экран, на который вы переходите, помещается на самый верх стека. Каждый раз, когда вы нажимаете кнопку назад, экраны удаляются с верхней части стека.

Реализация:

*import { NavigationContainer } from '@react-navigation/native';  
import { createStackNavigator } from '@react-navigation/stack';*

*const Stack = createStackNavigator();*

*class App extends React.Component {  
 render() {  
 return (  
 <NavigationContainer>  
 <Stack.Navigator>  
 <Stack.Screen  
 name="Home"  
 component={HomeScreen}  
 />*

*<Stack.Screen  
 name="Friends"  
 component={FriendsScreen}  
 />  
 </Stack.Navigator>  
 </NavigationContainer>  
 );  
 }  
}*

Пример использования (в компоненте HomeScreen):

*<Button  
 title="Add some friends"  
 onPress={() =>  
 this.props.navigation.navigate('Friends')  
 }  
 />*

## 38. Зачем использовать классы в React Native?

Компоненты класса - это классы JavaScript ES6, которые расширены из базового класса под названием React.Компонент.

Пример:

*import React, { Component } from 'react';*

*class LoginForm extends Component {*

*state = { email: '', password: '' };*

*render() {*

*return (*

*<View>*

*<Text>{this.state.email}</Text>*

*<Text>{this.state.password}</Text>*

*</View>*

*);*

*}*

*}*

Компонент-класс получает доступ к методам жизненного цикла React, таким как render, а также к функциям состояния / реквизита от родительского элемента. Компоненты класса используются в качестве компонентов контейнера для управления состоянием и переноса дочерних компонентов.

Особенности классовых компонентов:

* Используются для динамических данных
* Могут обрабатывать динамические данные, такие как выборка данных, вводимые пользователем данные и т.д.
* Знает, когда он будет отображен (другими словами, реализует методы жизненного цикла)

Нужно использовать классы в React, если нам нужно использовать "продвинутые" методы жизненного цикла компонента, такие как shouldComponentUpdate() или подобные.

## 

## 39. “Умные” и “глупые” компоненты в React Native. Сравнительная характеристика.

В разработке приложений умные компоненты (smart components) обычно являются контейнерами, которые содержат логику и менеджмент состояния. Они часто связаны с данными, которые могут быть получены с сервера или храниться в состоянии приложения.

Глупые компоненты (dumb components), также известные как презентационные компоненты, обычно используются для отображения информации. Они не содержат логики или управления состоянием, а просто предоставляют интерфейс для отображения данных, переданных им в качестве пропсов (props).

Обычно рекомендуется разбивать приложение на умные и глупые компоненты, чтобы упростить тестирование и улучшить разбиение обязанностей.

Умные компоненты отвечают за логику и управление состоянием, а глупые компоненты обычно предназначены только для отображения информации, поэтому они часто бывают простыми и легко тестируемыми.

Они также могут быть переиспользованы в разных частях приложения. Например, вы можете создать компонент, который отображает карточку продукта, и использовать его в разных списках продуктов в вашем приложении. В то же время, умный компонент, такой как страница корзины, будет содержать логику для работы с корзиной покупателя, такую как добавление и удаление товаров, и также будет управлять состоянием корзины, выполняя определенные запросы на сервер.

## 

## 40. Как использовать CSS в React Native

React Native использует стили JavaScript вместо CSS, но они работают похожим образом. Чтобы использовать стили в React Native, вам нужно создать объект стилей и применить его к элементу, используя свойство style.

Вот пример того, как это может выглядеть:

*Import { StyleSheet } from 'react-native';*

*const styles = StyleSheet.create({  
 container: {  
 flex: 1,  
 backgroundColor: '#fff',  
 alignItems: 'center',  
 justifyContent: 'center',  
 },  
});*

*return (  
<View style={styles.container}>  
 {/\* остальной код компонента \*/}  
</View>  
);*

В React Native есть свойства padding и margin, которые можно использовать в стилях, чтобы установить отступы элемента. Они работают похожим образом, как в CSS, и могут быть заданы с помощью чисел, обозначающих количество пикселей, или строк, обозначающих относительное значение, такое как 10% или 3em.

Например, чтобы установить отступ сверху и снизу элемента на 20 пикселей, а справа и слева на 10 пикселей, можно использовать данный код:

*const styles = StyleSheet.create({  
 container: {  
 padding: 20,  
 paddingRight: 10,  
 paddingLeft: 10,  
 },  
});*

Чтобы установить отступы со всех сторон элемента, вы можете использовать свойство *paddingVertical* для отступов сверху и снизу и свойство *paddingHorizontal* для отступов справа и слева. Для margin аналогично.

Обратите внимание, что в React Native у свойств *padding* и *margin* нет коротких алиасов (таких как *padding-top* и *margin-left* в CSS), поэтому вам нужно указывать каждое свойство отдельно.

В React Native есть множество других свойств стилей, которые можно использовать, такие как flex, *backgroundColor*, *alignItems* и т.д.

## 

## 41. Обработка пользовательского ввода в React Native.

**TextInput**

Основополагающий компонент для ввода текста в приложение с помощью клавиатуры. Props обеспечивает настраиваемость нескольких функций,таких как автокоррекция,автокапитализация,текст с подставой и различные типы клавиатур,например,цифровая клавиатура.

Самый простой вариант использования — вывести *TextInput* и подписаться на события *onChangeText* для чтения пользовательского ввода. Есть и другие события, такие как *onSubmitEditing* и *onFocus* , на которые можно подписаться. Минимальный пример:

Два метода,выставляемые через "родной" элемент-это *.focus()* и *.blur()*, которые программно сфокусируют или размывают TextInput.

TextInput является основным компонентом , который позволяет пользователю вводить текст. У него есть *onChangeText* параметр, который вызывает функцию при каждом изменении текста, и *onSubmitEditing* параметр, который вызывает функцию при отправке текста.

Есть много других вещей, которые вы можете сделать с текстовым вводом. Например, вы можете проверить текст внутри, пока пользователь печатает.

**Пример кода:**

<TextInput

style={{height: 40}}

placeholder="Enter Your Email"

onChangeText={text => setText(text)}

defaultValue={text}

/>

## 

## 42. Основные компоненты в React Native и аналогичные html-объекты.

React Native поставляется с набором необходимых, готовых к использованию компонентов Native, которые вы можете использовать, чтобы начать создавать свое приложение. Это основные компоненты React Native . React Native также позволяет создавать собственные нативные компоненты для Android и iOS в соответствии с уникальными потребностями вашего приложения.

**Основные компоненты.**

* View - Самый фундаментальный компонент для создания пользовательского интерфейса.
* Text - Компонент для отображения текста.
* Image - Компонент для отображения изображений.
* TextInput - Компонент для ввода текста в приложение с помощью клавиатуры.
* ScrollView - Предоставляет контейнер с прокруткой, который может содержать несколько компонентов и представлений.
* StyleSheet - Предоставляет уровень абстракции, аналогичный таблицам стилей CSS.

React Native имеет множество основных компонентов для всего: от элементов управления до индикаторов активности. Вы можете найти их все документально в разделе API . В основном вы будете работать со следующими основными компонентами:

| **Компонент React Native** | **Android View** | **iOS View** | **Web Analog** | **Description** |
| --- | --- | --- | --- | --- |
| <View> | <ViewGroup> | <UIView> | Непрокручива- ющийся <div> | Контейнер,поддерживающий компоновку с помощью flexbox,стиль,некоторую обработку касаний и элементы управления доступностью. |
| <Text> | <TextView> | <UITextView> | <p> | Отображение,стилизация и вложение строк текста и даже обработка событий касания |
| <Image> | <ImageView> | <UIImageView> | <img> | Отображение различных типов изображений |
| <ScrollView> | <ScrollView> | <UIScrollView> | <div> | Общий контейнер для прокрутки,который может содержать несколько компонентов и представлений |
| <TextInput> | <EditText> | <UITextField> | <input type="text"> | Позволяет пользователю вводить текст |

## 

## 43. Компонент ListView и его использование в React Native.

**ListView** - основной компонент, предназначенный для эффективного отображения вертикально прокручиваемых списков изменяющихся данных. Минимальный API-интерфейс - создать ListView.DataSource , заполнить его простым массивом renderRow объектов данных и создать экземпляр компонента ListView с этим источником данных и обратным вызовом renderRow, который берет большой двоичный объект из массива данных и возвращает визуализируемый компонент.

ListView также поддерживает более продвинутые функции, в том числе разделы с закрепленными заголовками разделов, поддержку верхнего и нижнего колонтитула, обратные вызовы при достижении конца доступных данных ( onEndReached ) и для набора строк, которые видны при изменении области просмотра устройства ( onChangeVisibleRows ), а также несколько оптимизаций производительности.

**Using ListView**

React Native предоставляет набор компонентов для представления списков данных. Как правило, вы захотите использовать либо FlatList, либо SectionList .

Компонент FlatList отображает прокручиваемый список изменяющихся, но аналогично структурированных данных. FlatList хорошо работает для длинных списков данных, где количество элементов может меняться со временем. В отличие от более общего ScrollView, FlatList он отображает только те элементы, которые в данный момент отображаются на экране, а не все элементы сразу.

Компонент FlatList требует два реквизита: data и renderItem. data является источником информации для списка. renderItem берет один элемент из источника и возвращает отформатированный компонент для рендеринга.

В этом примере создается FlatList основа из жестко запрограммированных данных. Каждый элемент data реквизита отображается как Text компонент. Затем FlatListBasics компонент отображает FlatList и все Text компоненты.

const FlatListBasics = () => {

return (

<View style={styles.container}>

<FlatList

data={[ {key: 'Devin'}, {key: 'Dan'}, {key: 'Dominic'}, ]}

renderItem={({item}) => <Text style={styles.item}>{item.key}</Text>}

/>

</View>

);

};

export default FlatListBasics;

Если вы хотите отобразить набор данных, разбитых на логические разделы, возможно, с заголовками разделов, похожими на UITableViews в iOS, то SectionList — это то, что вам нужно.

const SectionListBasics = () => {

return (

<View style={styles.container}>

<SectionList

sections={[

{title: 'D', data: ['Devin', 'Dan', 'Dominic']},

{ title: 'J', data: [ 'Jackson', 'James', 'Jillian' ],},

]}

renderItem={({item}) => <Text style={styles.item}>{item}</Text>}

renderSectionHeader={({section}) => (

<Text style={styles.sectionHeader}>{section.title}</Text>

)}

keyExtractor={item => `basicListEntry-${item}`}

/>

</View>

);

};

export default SectionListBasics;

Одним из наиболее распространенных применений представления списка является отображение данных, получаемых с сервера.

## 

## 44. Асинхронное хранилище в React Native — общее назначение, области применения.

AsyncStorage— это незашифрованная, асинхронная, постоянная система хранения ключей и значений, которая является глобальной для приложения. Его следует использовать вместо LocalStorage. Поскольку asyncStorage данные не зашифрованы, сохраненные данные не преобразуются в код и не шифруются для предотвращения несанкционированного доступа, а это означает, что любой, у кого есть ваше устройство, может легко получить доступ к данным. AsyncStorage также является асинхронным, что означает, что сохраненные данные всегда будут доступны глобально, даже если вы выйдете из системы или перезапустите приложение. В iOS AsyncStorage поддерживается собственный код, который хранит небольшие значения в сериализованном словаре, а большие значения — в отдельных файлах. На Android AsyncStorage будет использоваться либо RocksDB , либо SQLite в зависимости от того, что доступно. Код AsyncStorageJavaScript — это фасад, который предоставляет четкий API JavaScript, реальные Error объекты и немногие функции. Каждый метод в API возвращает Promise объект.

AsyncStorage принимает и хранит только строковые данные, поэтому мы всегда должны сериализовать данные перед их сохранением, если это не строка. Это означает, что мы должны сначала преобразовать его в строковые данные перед сохранением; здесь и ключ, и значение являются строками.

Не рекомендуется хранить конфиденциальную информацию AsyncStorage, так как это небезопасно для конфиденциальной информации. Это связано с тем, что любой, у кого есть доступ к вашему телефону, может прочитать данные, хранящиеся в файловой системе документов вашего телефона.

Мы можем выполнить три основных действия с AsyncStorage: Set, Get и Delete:

* Set устанавливает или сохраняет данные в async хранилище с помощью пар ключ-значение
* Get получает значения данных из async хранилища по ключу
* Delete удаляет определенный фрагмент данных или несколько фрагментов данных с помощью ключа

Эти методы включают в себя:

* setItem()
* getItem()
* mergeItem()
* removeItem()
* multiGet()
* clear()

## 

## 45. Хранение конфиденциальных данных в React. Как можно достичь готовым способом React или как можно это сделать за счет других средств?

Никогда не храните конфиденциальные ключи API в коде вашего приложения. Все, что включено в ваш код, может быть доступно в виде обычного текста любому, кто просматривает комплект приложений. Такие инструменты, *как react-native-dotenv* и *react -native-config*, отлично подходят для добавления переменных среды, таких как конечные точки API, но их не следует путать с переменными среды на стороне сервера, которые часто могут содержать секреты и ключи API.

**Async Storage** — это поддерживаемый модуль для React Native, который обеспечивает асинхронное, незашифрованное хранилище ключей и значений. Асинхронное хранилище не используется совместно приложениями: каждое приложение имеет свою собственную изолированную программную среду и не имеет доступа к данным из других приложений.

**Используйте асинхронное хранилище, когда...**

* Сохранение нечувствительных данных во время работы приложения
* Сохранение состояния Redux
* Сохранение состояния GraphQL
* Хранение глобальных обще прикладных переменных

**Не используйте асинхронное хранилище для...**

* Token storage
* Secrets

React Native не поставляется в комплекте с каким-либо способом хранения конфиденциальных данных.Однако существуют готовые решения для платформ Android и iOS.

**iOS-Keychain Services** позволяет безопасно хранить небольшие фрагменты конфиденциальной информации для пользователя. Это идеальное место для хранения сертификатов, токенов, паролей и любой другой конфиденциальной информации, которой нет в асинхронном хранилище.

**Android-Shared Preferences** — это эквивалент постоянного хранилища данных «ключ-значение» в Android. Данные в Shared Preferences не шифруются по умолчанию , но Encrypted Shared Preferences является оболочкой для класса Shared Preferences для Android и автоматически шифрует ключи и значения.

**Android - Keystore** позволяет хранить криптографические ключи в контейнере, что затрудняет их извлечение с устройства.

Чтобы использовать сервисы iOS Keychain или Android Secure Shared Preferences, вы можете либо написать мост самостоятельно, либо использовать библиотеку, которая обернет их для вас и предоставит унифицированный API. Некоторые библиотеки,которые стоит рассмотреть:

* expo-secure-store
* react-native-encrypted-storage — использует связку ключей на iOS и EncryptedSharedPreferences на Android.
* react-native-keychain
* react-native-sensive-info — безопасно для iOS, но использует общие настройки Android для Android (которые по умолчанию небезопасны). Однако есть ветка , использующая Android Keystore.
* redux-persist-sensitive-storage — обертывает react-native-sensitive-info для Redux.

## 46. Анимация в приложении React Native. Общий механизм, обеспечение плавности.

React Native предоставляет две взаимодополняющие системы анимации: Animated для детального и интерактивного управления конкретными значениями и LayoutAnimation для анимированных транзакций глобального макета.

**Animated**

API Animated предназначен для краткого и очень эффективного выражения большого разнообразия интересных анимационных и интерактивных шаблонов. Animated фокусируется на декларативных отношениях между входными и выходными данными с настраиваемыми преобразованиями между ними и методами start/ stop для управления выполнением анимации на основе времени. Animated экспортирует шесть типов анимируемых компонентов: View, Text, Image, и ScrollView, но вы также можете создавать свои собственные с помощью .FlatListSectionListAnimated.createAnimatedComponent()

***Configuring animations***

В зависимости от типа анимации можно настроить пользовательские и предопределенные функции упрощения,задержки,продолжительности и многое другое.

Animated предоставляет несколько типов анимации, наиболее часто используемый - Animated.timing(). Он поддерживает анимацию значения с течением времени с помощью одной из различных предопределенных функций плавности, или вы можете использовать свою собственную. Функции замедления обычно используются в анимации, чтобы передать постепенное ускорение и замедление объектов.

По умолчанию timing будет использовать кривую easyInOut, которая передает постепенное ускорение до полной скорости и завершается постепенным замедлением до полной остановки. Вы можете указать другую функцию плавности, передав параметр easing. Пользовательская duration или даже delay перед запуском анимации также поддерживается.

Например,если мы хотим создать 2-секундную анимацию объекта,который слегка отступает назад,прежде чем перейти в конечное положение:

Animated.timing(this.state.xPosition, {

toValue: 100,

easing: Easing.back(),

duration: 2000

}).start();

***Composing animations***

Анимации можно комбинировать и воспроизводить последовательно или параллельно. Последовательные анимации могут воспроизводиться сразу после завершения предыдущей анимации или запускаться после указанной задержки. Animated API предоставляет несколько методов, таких как sequence() и delay() , каждый из которых принимает массив анимаций для выполнения и автоматически вызывает start() / stop() по мере необходимости.

Если одна анимация останавливается или прерывается, все остальные анимации в группе также останавливаются. Animated.parallel имеет параметр stopTogether , для которого можно установить значение false , чтобы отключить его.

***Interpolation***

Каждое свойство может быть сначала пропущено через интерполяцию.Интерполяция сопоставляет входные диапазоны с выходными диапазонами,обычно используя линейную интерполяцию,но также поддерживает функции упрощения.По умолчанию она экстраполирует кривую за пределы заданных диапазонов,но вы также можете закрепить выходное значение.

interpolate() также поддерживает произвольные функции замедления, многие из которых уже реализованы в модуле Easing. interpolate() также имеет настраиваемое поведение для экстраполяции outputRange. Вы можете установить экстраполяцию, установив параметры extrapolate, extrapolateLeft или extrapolateRight. Значение по умолчанию - extend но вы можете использовать clamp чтобы выходное значение не превышало outputRange .

## 

## 47. Сходства между реквизитом (props) и состоянием (state).

**Props**

Большинство компонентов можно настроить при их создании с различными параметрами. Эти параметры создания называются props.

Ваши собственные компоненты также могут использовать файлы props. Это позволяет вам создать один компонент, который будет использоваться во многих разных местах вашего приложения, с немного разными свойствами в каждом месте. С помощью props и основных компонентов Text, Image, и View вы можете создавать самые разнообразные статические экраны.

**State**

В отличие от props, которые доступны только для чтения и не должны изменяться, state компоненты React позволяют компонентам изменять свой вывод с течением времени в ответ на действия пользователя, сетевые ответы и что-либо еще.

В компоненте React **props**— это переменные, которые мы передаем от родительского компонента к дочернему компоненту. Точно так же состояние также является переменными, с той разницей, что они не передаются как параметры, а компонент инициализирует их и управляет ими внутри.

## 

## 48. Гибкие макеты в React Native. Принципы создания, примеры.

**Flexbox** — это технология для создания сложных гибких макетов за счёт правильного размещения элементов на странице.

**flex** - определяет, какую часть экрана занимает элемент (целые значение больше 0).

<View>

<<View style={{ flex: 4, backgroundColor: "white" }} />

<<View style={{ flex: 1, backgroundColor: "black" }} />

<<View style={{ flex: 7, backgroundColor: "green" }} />

</View>

**flexDirection** - определяет, в каком направлении размещаются дочерние элементы.

* column выравнивает дочерние элементы сверху вниз.
* row выравнивает дочерние элементы слева направо.
* column-reverse выравнивает дочерние элементы снизу вверх.
* row-reverse выравнивает дочерние элементы справа налево.

<View style={{flex: 1, flexDirection: "row"}}>

<<View style={{ flex: 4, backgroundColor: "white" }} />

<<View style={{ flex: 1, backgroundColor: "black" }} />

<<View style={{ flex: 7, backgroundColor: "green" }} />

</View>

**Layout Direction** - направление макета, в котором должны располагаться дочерние элементы и текст в иерархии. Также влияет на то, к чему относятся начало и конец кромки.

* LTR (значение по умолчанию) Текст и дочерние элементы располагаются слева направо. Поля и отступы, применяемые к началу элемента, применяются к левой стороне.
* RTL Текст и дочерние элементы располагаются справа налево. Поля и отступы, применяемые к началу элемента, применяются с правой стороны.

const DirectionLayout = () => {

const [direction, setDirection] = useState('ltr');

return (

<PreviewLayout

label="direction"

selectedValue={direction}

values={['ltr', 'rtl']}

setSelectedValue={setDirection}>

</PreviewLayout> ); };

**justifyContent** - определяет, как элемент должен отображаться вдоль основной оси.

* flex-start (значение по умолчанию) выравнивает элементы по началу главной оси контейнера.
* flex-end выравнивает элементы контейнера по концу его главной оси.
* center выравнивает элементы по центру главной оси контейнера.
* space-between расстояния между соседними элементами одинаковые, между элементами и краями контейнера отступов нет.
* space-around расстояния между соседними элементами одинаковые, между элементами и краями контейнера есть отступ, равный половине расстояния между соседними элементами.
* space-evenly задает одинаковое расстояние между элементами, а также между элементом и краем контейнера.

<View style={{flex: 1, flexDirection: 'column', justifyContent: 'space-between',}}>

<View style={{width: 50, height: 50, backgroundColor: 'powder blue'}} />

<View style={{width: 50, height: 50, backgroundColor: 'sky blue'}} />

<View style={{width: 50, height: 50, backgroundColor: 'steel blue'}} />

</View>

**alignItems** - определяет, как элемент должен отображаться вдоль вторичной оси.

* stretch (значение по умолчанию) растягивает вдоль оси.
* flex-start элементы выравниваются по верхнему краю или по левому краю.
* flex-end элементы выравниваются по нижнему краю или по правому краю.
* center элементы выравниваются по центру.
* baseline элементы выравниваются в соответствии со своей базовой линией.

<View style={styles.container}>

</View>

const styles = StyleSheet.create({

container: {backgroundColor: "#7CA1B4", flex: 1, justifyContent: "center", alignItems: "center" },

});

**alignSelf** - определяет, как дочерний элемент должен выравниваться и переопределяет alignItems. <View style={styles.container}>

<View style={styles.square} />

<View style={[styles.square, { alignSelf: "flex-end" }]} />

<View style={[styles.square, { alignSelf: "flex-start" }]} />

</View>

**Align Content**[**​**](https://www.jscamp.app/ru/docs/reactnative04/#align-content) - определяет распределение линий вдоль поперечной оси. Это действует только тогда, когда элементы переносятся на несколько строк с помощью flexWrap. Имеет уже описанные выше свойства: flex-start, flex-end, stretch, center, space-between, space-around.

const AlignContentLayout = () => {

const [alignContent, setAlignContent] = useState('flex-start');

return (

<PreviewLayout

label="alignContent"

selectedValue={alignContent}

values={[

'flex-start',

'flex-end',

'stretch',

'center',

'space-between',

'space-around',

]}

setSelectedValue={setAlignContent}>

</PreviewLayout>

);

};

**flexWrap** - задаёт правила вывода flex-элементов — в одну строку или в несколько, с переносом блоков. Если перенос разрешён, то возможно задать направление, в котором выводятся блоки.

* wrap элементы разбиваются на несколько строк
* nowrap мы видим кол-во блоков помещающихся на экран
* wrap-reverse порядок строк меняется на противоположный

<View style={styles.container}> </View>

const styles = StyleSheet.create({

container: { flex: 1, flexDirection:'row', flexWrap: "wrap",}

## 

## 49. Синхронные и асинхронные действия в Redux.

В **Redux** все действия являются синхронными. Они описывают факт изменения состояния приложения, но не указывают, как именно это должно произойти.

Чтобы управлять асинхронными операциями в Redux, вы можете использовать санки. Санки(thunk) — это функции, которые могут выполнять асинхронные операции и отправлять синхронные действия в хранилище.

В React Native, асинхронность часто возникает при работе с сетью или файлами. Чтобы управлять асинхронными операциями в этих случаях, вы можете использовать санки или инструменты типа redux-thunk или redux-saga.

**redux-thunk** — это библиотека для Redux, которая позволяет писать асинхронные экшены. Она работает с помощью так называемых thunk-функций, которые являются функциями, которые возвращают другие функции. Чтобы начать использовать redux-thunk в своем приложении, необходимо установить соответствующую библиотеку.

**redux-saga** — это библиотека для Redux, которая позволяет управлять асинхронными операциями в вашем приложении. Она работает с помощью специальных функций, называемых "сагами", которые могут быть запущены, остановлены или отменены.

Саги могут быть использованы для управления такими вещами, как сетевые запросы, работа с файлами и т. д. Они также могут быть использованы для обработки синхронных действий, отправленных в хранилище, и отправки новых действий в хранилище в ответ.

## 

## 50. Что позволяют сделать свойства стиля borderWidth, borderColor, borderRadius?

**borderWidth**, **borderColor**, **borderRadius** - это свойства стилей, которые можно использовать в React Native для настройки рамки элемента.

* **borderWidth** устанавливает толщину рамки элемента. Это свойство принимает значение числа, которое определяет толщину рамки в пикселях.
* **borderColor** устанавливает цвет рамки элемента. Это свойство принимает значение строки цвета в формате "#RRGGBB" или "rgba(r, g, b, a)".
* **borderRadius** устанавливает радиус скругления углов рамки элемента. Это свойство принимает значение числа, которое определяет радиус скругления углов в пикселях.

Пример:

*<View style={{  
 borderWidth: 1,  
 borderColor: '#fafafa',  
 borderRadius: 5,  
}}>  
 <!-- Контент элемента -->  
</View>*

## 

## 51. Линковка пользовательских шрифтов в React Native.

Для начала необходимо выбрать семейство шрифтов, которые вы хотите использовать в своем приложении react native. Затем необходимо добавить файлы в формате otf в папку assets/fonts. Далее, если версия React Native ≥ 0.60, то необходимо создать файл “react-native.config.js” со следующим кодом:

*module.exports = {  
 project: {  
 ios: {},  
 android: {},  
 },  
 assets: [“./assets/fonts/”]  
};*

Если версия React Native < 0.60, то необходимо добавить в файл package.json следующий код

*“rnpm”: {  
 “assets”: [  
 “./assets/fonts/”  
 ]  
}*

Далее, если **React Native ≥ 0.69**, необходимо написать в терминале: *npx react-native-asset*

Для версии **React Native < 0.69**: *npx react-native link*

Таким образом, можно использовать свои шрифты добавит в styles свойство fontFamily

*instruction: {  
 fontFamily: “fontName”  
}*

## 

## 52. Задание стиля текста с помощью свойств в React Native.

**color** - цвет текста

**fontFamily** - шрифт

**fontSize** - размер текста

**fontStyle** - обычный текст или курсив ('normal', 'italic')

**fontWeight** - жирность текста ('normal', 'bold', '100', '200', '300', '400', '500', '600', '700', '800', '900')

**letterSpacing** - расстояние между символами

**lineHeight** устанавливает величину пространства между строками

**textAlign** - выравнивание текста ('auto', 'left', 'right', 'center', 'justify')

**textShadowColor** - добавляет тени к тексту

**textTransform** - преобразует элементы текста в в заглавные или прописные символы ('none', 'uppercase', 'lowercase', 'capitalize')

## 

## 53. Отображение контента React Native поверх Status Bar и System Bar.

React Native использует компонент SafeAreaView для регулирования отображения контента поверх Status Bar и System Bar.

Пример:

*import { SafeAreaView } from 'react-native';  
<SafeAreaView>  
 {/\* Контент, который будет отображаться поверх Status Bar и System Bar \*/}  
</SafeAreaView>*

Компонент SafeAreaView автоматически подгоняет свою геометрию и стили для соответствия безопасной области экрана на устройствах с различными форм-факторами и разрешениями, такими как iPhone X.

## 54. Свойства стилей React Native, отвечающие за расположение объекта.

**position('absolute', 'relative')** - задает абсолютное или относительное расположение элемента

* **right** - отступ справа от родительского элемента
* **left** - отступ слева от родительского элемента
* **top** - отступ сверху от родительского элемента
* **bottom** - отступ снизу от родительского элемента
* **margin** - задает расстояние от полей элемента до соседних элементов.

## 

## 55. Задание относительных размеров объектов в React Native.

Для этого необходимо установить значение свойств width и height в процентах.

Например:

*<View  
 style={{  
 width: '33%',  
 height: '50%',  
 backgroundColor: 'steelblue',  
 }}  
 />*

Таким образом, ширина элемента будет занимать 33% ширины родительского элемента, а высота 50% от высоты. Это позволяет задавать размер элемента в зависимости от размера экрана.

Также, задать размер элементов относительно друг друга можно с помощью свойства flex.

Пример:

<View style={{flex: 1, backgroundColor: 'red'}} />

<View style={{flex: 2, backgroundColor: 'darkorange'}} />

<View style={{flex: 3, backgroundColor: 'green'}} />

В данном примере указано, что второй View будет занимать в 2 раза больше свободного пространства, чем первый. Аналогично с третьим.

## 

## 56. Компиляция кода React Native в платформенно-специфичный код.

React Native использует JavaScriptCore, чтобы интерпретировать JavaScript код и преобразовывать его в нативный код, который может быть понят и выполнен на мобильных платформах. Это означает, что разработчик может написать код на JavaScript и JSX (расширенный JavaScript, который похож на HTML и используется для описания UI компонентов) и затем использовать React Native для преобразования этого кода в нативный код для iOS и Android.

В процессе трансляции, React Native компилирует JavaScript код в машинный код с помощью **JavaScriptCore**. Затем он использует специальные библиотеки для каждой платформы, такие как **iOS UIKit** и **Android UI**, чтобы обеспечить выполнение кода и отображение UI компонентов.

React Native также использует механизм построенный на принципе **"bridge"** (мост) для обеспечения взаимодействия между JavaScript кодом и нативным кодом. Это позволяет React Native обрабатывать события, которые происходят в нативном коде, и обновлять состояние приложения в JavaScript. В другую сторону, JavaScript код может вызывать методы в нативном коде для обновления UI или для доступа к нативным API платформы.

В целом, алгоритм трансляции React Native обеспечивает возможность использовать JavaScript и JSX для создания мобильных приложений для разных платформ, благодаря интеграции с JavaScriptCore и библиотеками для каждой платформы, а так же механизмом "bridge" для обеспечения взаимодействия между JavaScript кодом и нативным кодом.

Кратко:

1. Разработчик написал JS и JSX код для создания компонентов и логики приложения.
2. React Native использует JavaScriptCore для компиляции JS кода в машинный код.
3. React Native использует библиотеки для каждой платформы, такие как iOS UIKit и Android UI, для обеспечения выполнения кода и отображения UI компонентов.
4. React Native использует "bridge" для установления связи между JS кодом и нативным кодом, для обработки событий и обновления состояния приложения.
5. После трансляции, машинный код может быть скомпилирован и собран в нативное приложение для iOS или Android.

Важно отметить, что трансляция происходит в реальном времени, при запуске приложения. Это означает, что разработчик может мгновенно видеть изменения в приложении после каждой модификации кода и обновления приложения.

## 

## 57. Как реализовать анимацию в React Native?

Анимация представляет собой изменение одного состояния на другое, то есть необходимо определить начальную и конечную точки (состояния) объекта перед тем, как его анимировать. Например, должен ли объект появиться с изменением прозрачности или он просто “въехать”?

В React Native для создания и управления анимацией используется такой инструмент, как Animated. Но прежде всего, есть три «группы вещей» или «блока», о которых нужно знать, чтобы работать с анимацией.

**1: Визуальное состояние Animated.Value**

С логической точки зрения любой компонент может быть либо показан, либо скрыт, то есть визуальное состояние может быть чем-то между истинным состоянием и ложным. Мы можем ввести переменную для представления диапазона визуального состояния. Нужно, чтобы это было число, поскольку независимо от того, какое мы имеем логическое состояние, числа помогают нам представлять промежуточные значения.

*this.\_shown = new Animated.Value(0);*

В то время как логическое состояние может быть двоичным (1 или 0), визуальное состояние – это число с плавающей точкой.

**2: Переходы Animated.timing**

Если компонент скрыт: это значит, что логическое состояние его параметров видимости будет ложным, и визуальное состояние будет также 0.0. Но если мы хотим показать компонент, логическое состояние должно немедленно стать истинным, тогда как визуальное - постепенно переходить от 0.0 до 1.0. Для этого нам нужен способ сообщить визуальному состоянию переход к 1.0.

*Animated.timing(this.\_shown, {  
 toValue: 1,  
 duration: 300,  
}).start();*

Здесь, мы говорим Animated изменять \_shown до 1.0 в промежутке 300 мс.

**3: Пиксели Animated.View и interpolate**

Переходы \_shown ничего не значат, если их нельзя увидеть. Нужно использовать \_shown для установки степени прозрачности дочернего компонента. Для примера возьмем код:

*<View style={{ opacity: this.state.shown ? 1 : 0 }}>  
 <SomeComponent />  
</View>*

Мы устанавливаем прозрачность в 0, когда скрываем компонент и в 1, когда показываем.

Animated.Value можно использовать при работе со стилями. Нужно всего лишь изменить View на Animated.View, и теперь у нас есть следующее:

*const opacity = this.\_shown; // This is an Animated.Value  
<Animated.View style={{ opacity: opacity }}>  
 <SomeComponent />  
</Animated.View>*

## 58. Организация файловой системы в React Native.

React Native использует файловую систему своей платформы, так что нужно использовать специфические модули для работы с файлами в iOS и Android.

**Модули для работы с файловыми системами**

Например, для работы с файлами в iOS можно использовать react-native-fs, а для Android - react-native-fetch-blob. Эти модули позволяют сохранять, читать и удалять файлы и папки на устройстве.

**Модули для выбора файлов из ФС**

Также может понадобиться доступ к файловой системе устройства для загрузки файлов, и для этого можно использовать модули, такие как react-native-document-picker и react-native-image-picker. Они позволяют выбрать и загрузить файлы из файловой системы устройства. Также может понадобиться использовать модули, такие как react-native-fetch-blob и react-native-fs, чтобы загружать и сохранять файлы на сервер.

**Ограничения операционных систем**

Важно иметь в виду, что доступ к файловой системе устройства может быть ограничен и необходимо запрашивать разрешения для доступа к некоторым файлам и папкам

**Различия между файловыми системами IOS и Android**

* Android хранит файлы и папки в корневой папке устройства, которая называется /, и имеет несколько внутренних и внешних хранилищ, которые можно использовать для хранения файлов и папок.
* iOS имеет более строгую структуру файловой системы, которая ограничивает доступ приложений к файлам и папкам. Файлы и папки, которые необходимо сохранять между сеансами, должны сохраняться в каталоге документов или каталоге библиотеки приложения.

## 

## 59. За счет чего можно осуществить передачу данных с одного экрана на другой?

Одним из самых больших недостатков React Native является то, что он не поставляется с какой-либо встроенной системой навигации. Мы должны использовать другие библиотеки для реализации навигации в React Native. Библиотека навигации React широко используется вместе с React Native для реализации различных типов навигационных систем.

Передавать данные между экранами в React Native можно с помощью навигатора стека и хуков. Для начала нужно создать навигатор в **основном компоненте** приложения. У нас будет 2 экрана “First” и “Second”:

*const Stack = createNativeStackNavigator();*

*export default function App() {  
 return (  
 <NavigationContainer>  
 <Stack.Navigator  
 <Stack.Screen name="First" component={First} />  
 <Stack.Screen name="Second" component={Second} />  
 </Stack.Navigator>  
 </NavigationContainer>  
 );  
}*

**В компоненте “First” мы будем отдавать данные:**

*const [message, setMessage] = useState('');* - заводим переменную для хранения данных

*const navigation = useNavigation();*

*const go = () => {* - и метод для перехода между экранами и передачи сообщения *navigation.navigate('Second', {  
 message,  
 });  
 };*

Запоминать данные будем из TextInput

*<TextInput  
 onChangeText={(text) => setMessage(text)}  
 />*

А переходить между экранами с помощью кнопки

*<Button title="Go"   
 onPress={go}/>*

**Считать данные в компоненте Second** можно следующим образом:

*const route = useRoute()  
<Text>{route.params.message}</Text>*

## 

## 60. Работа с БД в React Native.

Разберем работу с БД на примере sqlite. Откроем существующую базу данных с помощью openDatabase()

*const db = SQLite.openDatabase('todos.db');*

Любой запрос оборачивается в db.transaction() - транзакцию. Транзакция - это атомарная операция, она может быть выполнена только целиком, либо не быть выполнена вообще. В качестве параметра передается объект SQLTransaction. Он имеет один единственный метод executeSql(sqlStatement, args, callback, errorCallback).

* sqlStatement - строка, содержащая SQL-запрос к базе данных. Она может содержать “?”-заполнители со значениями для замены, перечисленными в args.
* args (optional) - массив параметров для подстановки в SQL-запрос.
* callback (optional) - вызывается при успешном завершении запроса во время транзакции. Принимает два параметра: сама транзакция и объект ResultSet
* errorCallback (optional) - вызывается, если произошла ошибка при выполнении запроса в транзакции. Принимает два параметра: сама транзакция и объект ошибки.

Таким образом, для любого запроса к базе данных используется следующий синтаксис (на примере вставки данных):

*export const insertData = (data) => {  
 return new Promise((resolve, reject) => {  
 db.transaction(tx => {  
 tx.executeSql(  
 'insert into todos (data, isCompleted) values (?, ?)',  
 [task, 0],  
 (\_, result) => resolve(result),  
 (\_, err) => reject(err),  
 );  
 });  
 });  
};*

## 

## 61. Загрузка данных с сервера в React Native.

**С использованием fetch**

Для загрузки данных с сервера можно использовать fetch API, который похож на XMLHttpRequest или любой другой сетевой API.

Пример запроса к серверу с помощью fetch.

*fetch('https://jsonplaceholder.typicode.com/posts/1')  
 .then((response) => response.json())  
 .then((responseJson) => console.log(responseJson));  
 .catch(error => {// Обработка ошибки});*

В приведенном выше коде мы извлекаем файл JSON https://jsonplaceholder.typicode.com/posts/1 и выводим данные на консоль. Простейший вызов API fetch() принимает один аргумент, то есть путь, и возвращает объект Promise с http-ответом. Чтобы получить содержимое JSONа из ответа, нужно использовать метод json().

Второй параметр fetch API - это объект, который может иметь метод (GET, POST и т.д.), headers, данные, которые нужно отправить, и другое.

При желании можно обработать ошибки в catch().

**С использованием axios**

axios - это библиотека JavaScript, которая используется для отправки и получения HTTP запросов. Она может быть использована в браузере и на стороне сервера с помощью Node.js.

Пример использования 'axios' с кастомными headers:

*import axios from 'axios';  
axios.get('https://your-server.com/api/data', {  
 headers: {  
 'Authorization': 'Bearer ' + token  
 }  
 })  
 .then(response => {// Обработка загруженных данных})  
 .catch(error => {// Обработка ошибки});*

## 

## 62. Component Update. Общий механизм работы.

Он связан с примирением. Конечной целью согласования является наиболее эффективное обновление пользовательского интерфейса в соответствии с новым состоянием. Если мы знаем, что определенная часть пользовательского интерфейса не изменится, то у React нет причин беспокоится о том, следует ли ее отображать. Возвращая false из shouldComponentUpdate, React будет предполагать, что текущий компонент и все его дочерние элементы останутся такими же, как текущий компонент.

## 

## 63. С помощью какого компонента можно идентифицировать платформу, на которой будет запускаться приложение? Для чего это нужно?

**Platform module**

Этот модуль в React Native позволяет определять платформу, на которой запущено приложение. Мы можем использовать логику обнаружения для реализации кода, специфичного для конкретной платформы. Использовать этот параметр рекомендуется в тех случаях, когда только небольшие части компонента зависят от платформы.

## 

## 64. Какое свойство стиля управляет направлением, в котором размещаются дочерние элементы части экрана?

**flexDirection** управляет направлением, в котором располагаются дочерние элементы узла. Это также называется главной осью. Поперечная ось - это ось, перпендикулярная главной оси, или ось, по которой проложены линии обертывания.

* **column** - (значение по умолчанию) Выравнивает дочерние элементы сверху вниз. Если перенос включен, то следующая строка будет начинаться справа от первого элемента в верхней части контейнера.
* **row** - Выравнивает дочерние элементы слева направо. Если перенос включен, то следующая строка будет начинаться под первым элементом слева от контейнера.
* **column-reverse** - Выравнивает дочерние элементы снизу вверх. Если перенос включен, то следующая строка будет начинаться справа от первого элемента в нижней части контейнера.
* **row-reverse** - Выравнивает дочерние элементы справа налево. Если перенос включен, то следующая строка будет начинаться под первым элементом справа от контейнера.

## 

## 65. Библиотеки обработки жестов, совместимые с React Native.

**Pan gesture**

Чтобы реализовать жест панорамирования в React Native с использованием библиотеки react-native-gesture-handler, мы будем использовать PanGestureHandler. PanGestureHandler - это непрерывный обработчик жестов, который генерирует потоки событий жестов, когда пользователь перемещает элемент.

**TapGestureHandler**

С помощью TapGestureHandler мы можем реализовать жесты как с одним, так и с двойным касанием.

**Swipeable gesture**

Благодаря этому жесту можно реализовать возможность пролистывания.

**Hold/long-press gesture**

С помощью встроенного обработчика жестов React реализация жеста длительного нажатия / удержания проста.

## 

## 66. Какие методы обновления жизненного цикла компонента вы знаете?

**static getDerivedStateFromProps()** - вызывается непосредственно перед вызовом метода render(). Должен вернуть объект для обновления состояния или null, чтобы ничего не обновлять.

**shouldComponentUpdate()** - Используйте shouldComponentUpdate(), чтобы сообщить React, не влияет ли текущее изменение состояния или реквизитов на выходные данные компонента.

**render()** - единственный обязательный метод в классовом компоненте. При вызове он должен проверять this.props и this.state и возвращать один из следующих типов: react-элемент, массивы и фрагменты, порталы, строки и числа, логические значения или null.

**getSnapshotBeforeUpdate()** - вызывается перед тем, как последний отрисованный вывод будет зафиксирован, например, в DOM. Он позволяет вашему компоненту захватывать некоторую информацию из DOM до ее потенциального изменения.

**componentDidUpdate()** - вызывается сразу после обновления. Не вызывается при первоначальной отрисовке. Используется для возможности работы с DOM при обновлении компонента.

## 

## 67. Зачем нужен UseEffect?

**UseEffect** позволяет использовать побочные эффекты в функциональных компонентах.

Используя этот hook, React понимает, что компонент должен что-то сделать после рендера. React запоминает переданную функцию и вызовет ее позже после выполнения обновлений DOM. В этом случае мы задаем заголовок документа, но также можно выполнить и выборку данных или вызвать какой-либо другой императивный API.

Размещение внутри компонента позволяет получить доступ к переменной состояния прямо из эффекта. Специальный API в таком случае не нужен.