dex

Обучающий курс

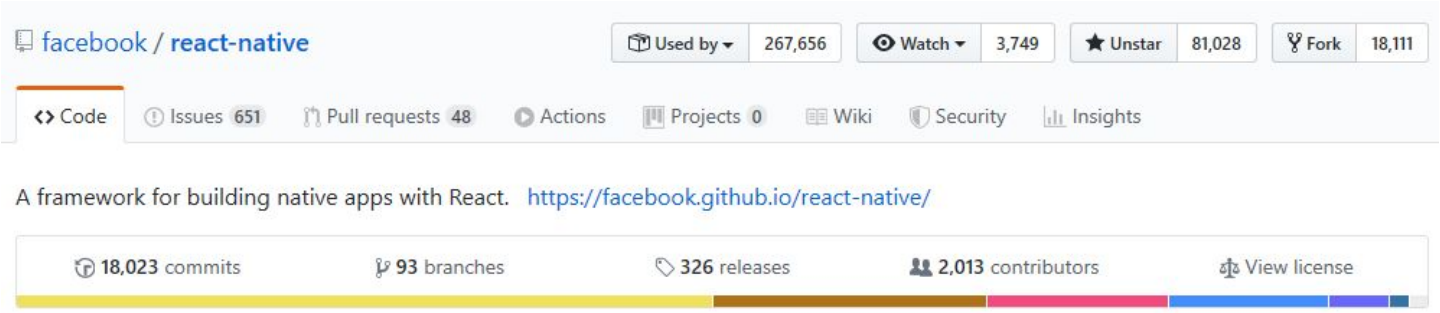
Мобильная разработка на **React Native**

(первый уровень)

https://t.me/UpRNDex – группа для вопросов

Вступление

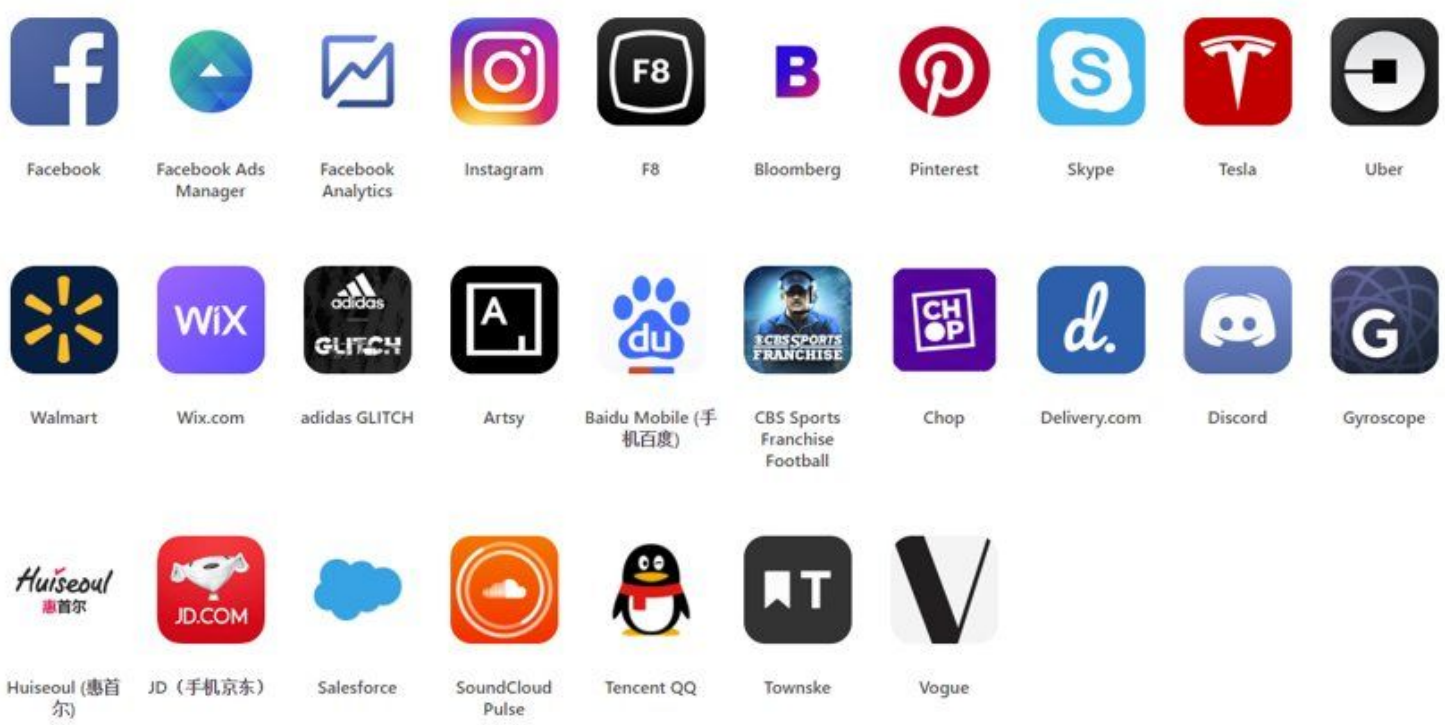
Данный курс поможет вам написать кроссплатформенное приложение используя React Native.
React Native это фреймворк разработанный корпорацией Facebook с целью упрощения написания мобильных приложений с использованием ранее созданного фреймворка React. На сегодняшний день это один из самых популярных репозиториев GitHub.



При написании кода используется JavaScript или TypeScript (будет рассмотрен в данном курсе). С развитием фреймворка и появлением энтузиастов использовать React Native стало возможно за пределами мобильных платформ: Windows, MacOS, Web. Для этих систем появились свои версии React Native которые приносят поддержку данного фреймворка.

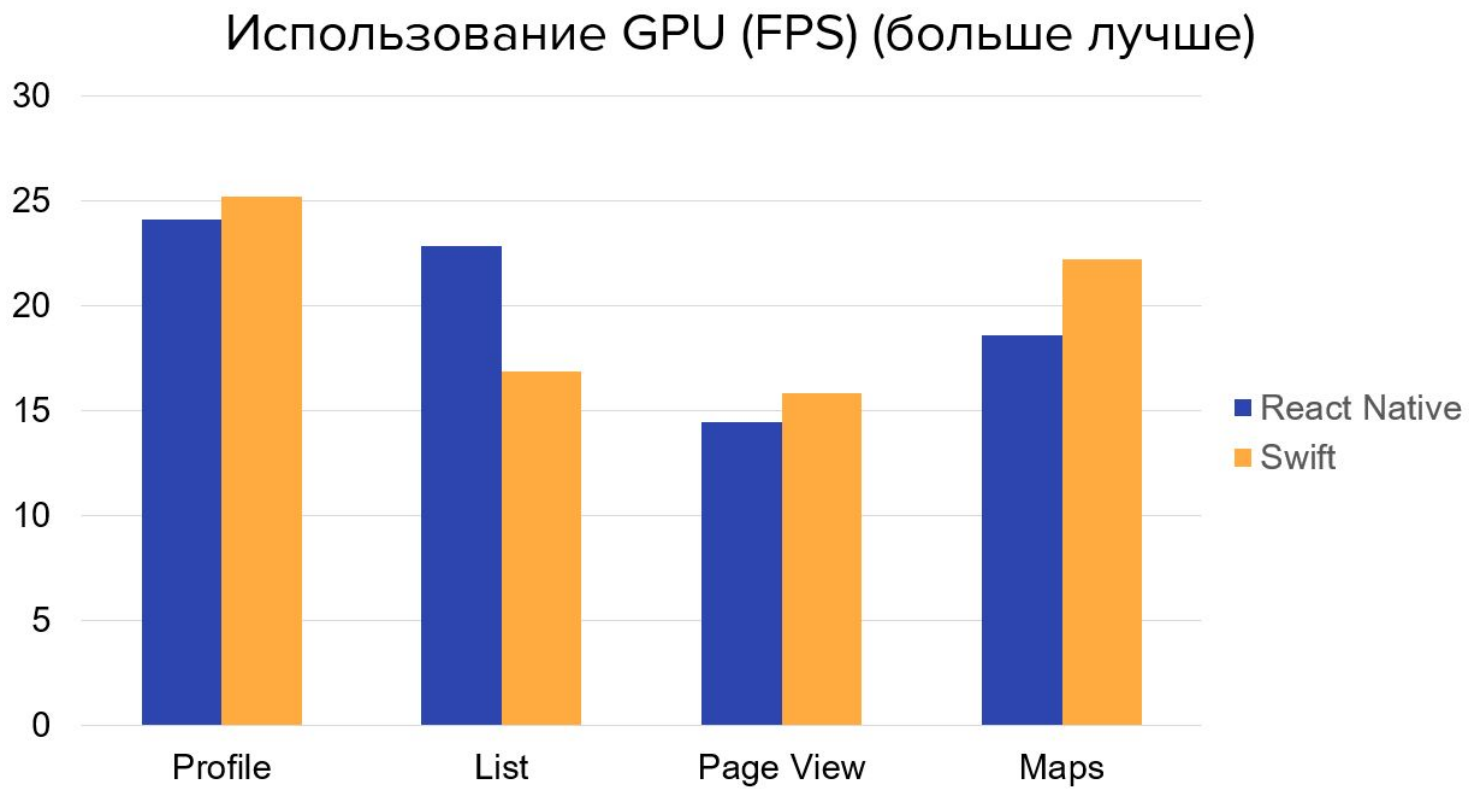
Возможно у вас возникает вопрос почему именно React Native, когда на слуху часто можно услышать другие фреймворки? Ответом на данный вопрос будет в основном зрелость данной технологии и количество людей которые поддерживают её прямо (развивая сам фреймворк) или косвенно (создавая компоненты), ни один другой фреймворк не может похвастаться такими плюсами, ближайшим к таким показателям может стать Flutter от корпорации Google, но на данный момент он ещё слишком молод чтобы на него можно было положиться.

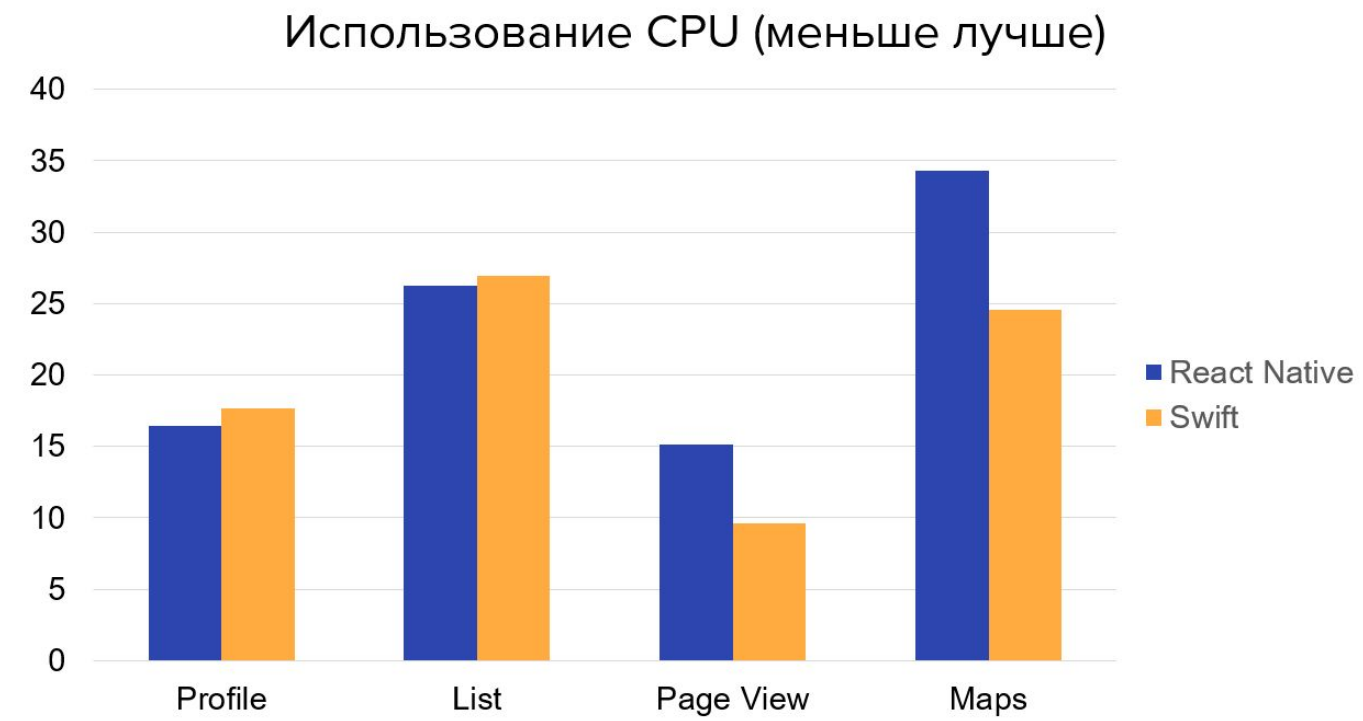
Чтобы подпитать ранее написанный текст фактами из реальной жизни, достаточно увидеть список приложений которые написаны при использовании React Native:

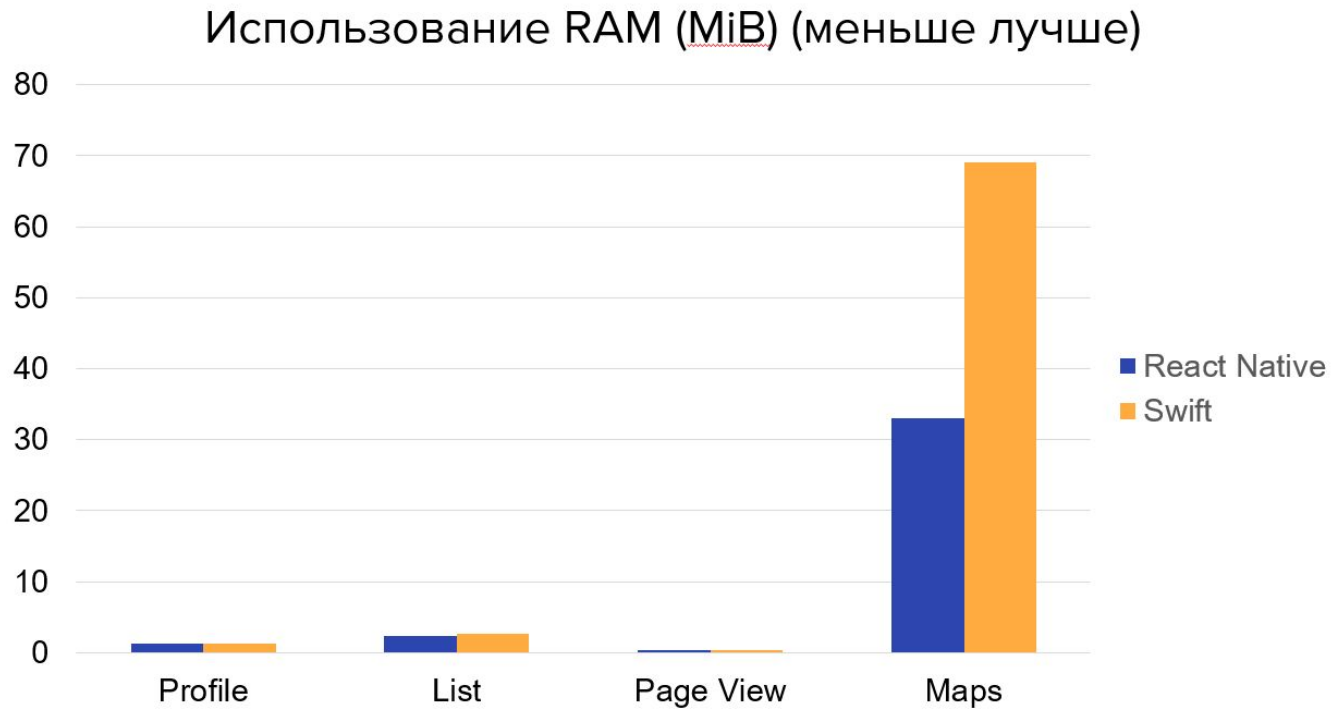


Отдельно вы можете изучить содержимое [следующего репозитория](), в котором в виде списка представлены приложения написанные различными разработчиками.

Если же вы считаете в целом что кроссплатформенная разработка не может быть соизмерима и быть полноценной заменой нативной, то следующие цифры должны изменить ваше мировоззрение касательно данного вопроса:


Использование GPU (FPS) (больше лучше)
30
25
20
15
10
5
0
Profile
List
Page View
Maps
React Native
Swift





Если у вас ещё остались сомнения или же вы считаете что есть технологии которые способны заменить React Native и при этом увеличить количество положительных моментов, то напишите нам, мы рассмотрим любое ваше предложение (серьёзно, пишите), в ином случае начнем процесс с того с чего начинается любой другой курс, установки инструментария для работы.

Примечание

Документ затрагивает различные аспекты разработки при использовании React Native. По мере прочтения документа вы будете встречаться с различными ссылками которые дадут вам детальное объяснение работы модуля или функционала.

На момент написания документа последняя версия React Native была 0.61.1. К моменту прочтения версия может отличаться от той что используется в документе и различные требования и зависимости могут изменится.

Установка инструментария

Зависимости

Android

Для стабильной работы React Native на вашей машине потребуется установить следующие зависимости:

- Java SE Development Kit (JDK) - комплект разработчика приложений на языке Java, включающий в себя компилятор Java (javac), стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему Java (JRE);
- Node JS - программная платформа, основанная на движке V8 (транслирующем JavaScript в машинный код), превращающая JavaScript из узкоспециализированного языка в язык общего назначения;
- Watchman - утилита которая “следит” за изменениями файлов;
- Google Chrome - пригодиться для удаленной отладки приложений;
- Yarn - менеджер пакетов
- Android Studio - IDE для разработки приложений под Android;

Вы могли заметить что вместе с Node JS был установлен менеджер пакетов npm который позволяет нам устанавливать внешние зависимости и компоненты, выполнять различные команды при помощи CLI и многое другое.

Для того чтобы мы могли выполнять команды связанные с React Native необходимо установить React Native Command Line Interface, чтобы сделать это достаточно выполнить следующую команду в командной строке:

npm install -g react-native-cli

Даже если у вас есть устройство под управлением Android, вам нужно будет установить эмулятор для проверки работы вашего приложения на различных версиях данной ОС и различных размерах и форм факторах девайсов, чтобы убедиться что ваше приложение будет выглядеть и работать отлично на большинстве устройстве.

Более детальную информацию по установке вы можете прочитать на сайте React Native

iOS

Если вы работаете на Windows или Linux то вам необходимо будет либо купить устройство которое поддерживает Mac OS или установить виртуальную машину (в этом случае вы сможете работать на вашей операционной системе, а работа связанная с установкой приложений на устройства под управлением iOS или работа с нативным кодом будет происходить в виртуальной машине)

Более детальную информацию по установке вы можете прочитать на сайте React Native

IDE

Для непосредственного написания кода нам нужно установить IDE, здесь у нас есть несколько альтернатив которые можно будет использовать в зависимости от ваших предпочтений:

- WebStorm - IDE от JetBrains распространяемая на основе подписки для работы с JavaScript
- Visual Studio Code - Бесплатно распространяемая, open source IDE от корпорации Microsoft

Для большего удобства при работе с кодом рекомендуется поставить следующие расширения:

WebStorm

- Key Promoter X - отображает горячие клавиши для действий которые вы совершаете при помощи мыши.
- Code Glance - показывает полное содержимое открытого файла и позволяет передвигаться по нему.
- Ace Jump - навигация по коду при использовании различных сочетаний клавиш.
- UUID Generator - генератор универсальных уникальных идентификаторов

Visual Studio Code

- git-autoconfig - позволяет выбирать git пользователя перед работой с проектом
- GitLens - даёт больше контроля и возможностей для работы с git

JavaScript

Перед тем чтобы продолжить изучение данного курса, желательно ознакомится с основами JavaScript и TypeScript, для этого вам помогут следующие учебники:

- Современный учебник JavaScript
- Учебник TypeScript - TypeScript лишь расширяет возможности JavaScript, поэтому большинство что вы прочтете в этом учебнике будет вам знакомо после прочтения первой ссылки.

Коротко опишем основы касаемые JavaScript:

JavaScript - это язык интерактивности на веб-страницах. Без него сейчас не обходится ни один сайт в интернете. И даже больше! Используя различные фреймворки, JavaScript заполоняет интернет: серверная сторона (**Node.js**), клиентская часть(**React**, **Angular**, **Vue**) мобильные приложения (**React Native**, **Ionic**), виртуальная реальность (**React VR**) и так далее. Поэтому, если вы хотите пойти по одному из этих путей - нативный JS станет для вас просто необходимой базой.

Типы данных, переменные, основные операторы

Очень важный аспект любого языка программирования — это его система типов и типы данных в нем. Для строго типизированных языков программирования, например для таких как Java, переменные определяются конкретными типами, которые в свою очередь ограничивают значения переменных.

Несмотря на то, что JavaScript — это динамически типизированный язык программирования, существуют расширения над языком, которые поддерживают строгую типизацию, например TypeScript.

В JavaScript есть 7 основных типов.

- number для любых чисел: целочисленных или чисел с плавающей точкой.
- string для строк. Строка может содержать один или больше символов, нет отдельного символьного типа.

- boolean – булевый тип может принимать только два значения: true (истина) и false (ложь).
- null для неизвестных значений – отдельный тип, имеющий одно значение null.
- undefined для не присвоенных значений – отдельный тип, имеющий одно значение undefined.
- object для более сложных структур данных.
- symbol для уникальных идентификаторов.

Вы уже наверно начали задаваться вопросом, а как же массивы, функции, регулярные выражения и прочие вещи?

Все это специальные виды объектов.

JavaScript содержит несколько конструкторов для создания и других различных объектов, например, таких как:

- Date — для создания объектов даты и времени
- RegExp — для создания регулярных выражений
- Error — для создания JavaScript ошибок

В JavaScript есть следующие типы операторов:

- Операторы присваивания
- Операторы сравнения
- Арифметические операторы
- Бинарные операторы
- Логические операторы
- Строковые операторы
- Условный (тернарный) оператор
- Оператор запятая
- Унарные операторы
- Операторы отношения
- Приоритет операторов

Массивы, методы map, filter деструктуризация массивов

Массив – структура данных, хранящая набор значений (элементов массива), идентифицируемых по индексу или набору индексов, принимающих целые (или приводимые к целым) значения из некоторого заданного непрерывного диапазона. Например, массивы понадобится нам для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

- Методы массивов — map, filter, reduce
    - Array.prototype.map() принимает массив, каким-нибудь образом преобразует его элементы и возвращает новый массив трансформированных элементов.
    - Array.prototype.filter() принимает массив, просматривает каждый элемент и решает, убрать его или оставить. Возвращает массив оставшихся значений.
    - Array.prototype.reduce() принимает массив и вычисляет на основе его элементов какое-то единое значение, которое и возвращает.

Деструктуризация это разбивка сложной структуры на простые части. В JavaScript, таковая сложная структура обычно является объектом(о них поговорим в следующей главе) или массивом.

Объекты, деструктуризация объектов, классы

Как мы знаем из первой главы, в JavaScript существует семь типов данных. Шесть из них называются «примитивными», так как содержат только одно значение (будь то строка, число или что-то другое). Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка.

В этой статье подробно рассказывается про объекты, их свойствах, методы объектов, а также про деструктуризацию объектов можно почитать здесь.

Метод Object.keys() возвращает массив из собственных перечисляемых свойств переданного объекта, в том же порядке, в котором они бы обходились циклом for...in (разница между циклом и методом в том, что цикл перечисляет свойства и из цепочки прототипов). Примеры использования.

В объектно-ориентированном программировании класс – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

На практике нам часто надо создавать много объектов одного вида, например пользователей, товары или что-то ещё. С этим нам могут помочь конструкторы, создание объектов через "new", а именно new function.
Но в современном JavaScript есть и более продвинутая конструкция «class», которая предоставляет новые возможности, полезные для объектно-ориентированного программирования.
Иногда говорят, что class – это просто «синтаксический сахар» в JavaScript (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что мы можем сделать всё то же самое без конструкции class, однако, есть важные отличия.
Как и в литеральных объектах, в классах можно объявлять вычисляемые свойства, геттеры/сеттеры и т.д.

Подробнее про классы, их наследование, методы и т.д. можно почитать здесь и здесь.

Функции, контекст выполнения

Каждая функция в JavaScript — это объект Function. О свойствах и методах объектов Function можно прочитать в статье Function.

Функции — это не процедуры. Функция всегда возвращает значение, а процедура может возвращать, а может не возвращать.

Чтобы вернуть значение, отличное от значения по умолчанию, в функции должна быть инструкция return, которая указывает, что именно нужно вернуть. Функция без него вернёт значение по умолчанию. В случае конструктора, вызванного с ключевым словом new, значение по умолчанию — это значение

его параметра this. Для остальных функций значением по умолчанию будет undefined.
Подробнее про функции можно почитать здесь и здесь.

Создание проекта

Описание основных минимальных знаний для работы с React Native лучше всего отображают документы главной странички React Native.

Документы небольшие и на простых примерах отображают работу различных компонентов.

Теперь у нас есть все чтобы можно было перейти к изучению самого React Native.

Создадим наше первое приложение, в качестве примера разработаем с нуля наверное самый часто используемый тип приложения - магазин, который позволит нам также использовать большинство самых частых операций и компонентов (напр. авторизация, регистрация, работа со списками и т.д.) которые можно переиспользовать для других сценариев.

Для создания проекта достаточно выполнить следующую команду:

react-native init myapp --template react-native-template-dex

Рассмотрим данную команду детальнее

- ***react-native init*** - создает стандартное приложение для работы с React Native добавляя минимально требуемые зависимости для его работы. Все зависимости имеют последнюю или рекомендуемую версию для работы.
- ***myapp*** - название вашего приложения, а также часть его Bundle ID
- ***--template react-native-template-dex*** - шаблон компании Dex с предустановленными компонентами, собственной структурой проекта, утилитами и зависимостями.

Название приложения желательно писать на английском языке, без пробелов и в нижнем регистре для избежания предупреждений и возможных проблем с последующей работой.

В момент выполнения команды, в командной строке вы можете увидеть какие шаги происходят для создания проекта.

В результате у вас должна появится папка с пустым проектом, пройдемся по его содержимому чтобы понять что где находиться и за что отвечает.

Пример приложения реализованного при помощи шаблона:
https://github.com/dex-it/react-native-template-dex-example

Описание проекта

Начнём описание с корня нашего проекта делая акцента на каждой папке и файле.
Некоторые файлы будут пропущены т.к. напрямую не используются и не изменяются при разработке.

- **__tests__** - содержит файлы с тестами (Jest) для нашего проекта
- **android / ios** - содержит файлы связанные с нативной частью для Android / iOS
- **node_modules** - набор внешних зависимостей и их зависимостей, которые были установлены на основе зависимостей из package.json
- **resources** - содержит настройки, шрифты и изображения используемые в проекте
- **src** - папка с кодом нашего приложения
- **tools** - набор JavaScript скриптов для изменений внешних зависимостей и выполнению до или после установки внешних зависимостей
- **.gitignore** - список файлов и папок которые должны быть игнорированы git
- **index.js** - точка входа приложения
- **package.json** - список зависимостей, скриптов и дополнительных описаний проекта.
- **tsconfig.json** - параметры TypeScript для компилятора
- **tslint.json** - расширяемый инструмент статического анализа, который проверяет код TypeScript на наличие ошибок читаемости, удобства обслуживания и функциональности.
- **yarn.lock** - файл который фиксирует версии внешних зависимостей (создается и обновляется автоматически)

- **resources**
  - **fonts** - набор шрифтов которые будут добавлены в нативные проекты
  - **images** - все изображения которые используются в проекте
  - **settings** - настройки проекта для различных сред исполнения (development, test, staging, production)
- **src**
  - **common** - набор часто используемых компонентов, функций и т.п.

- **core** - основные компоненты, файлы конфигураций и т.п. минимальные для корректной работы приложения
- **modules** - набор модулей (страницы, специфичные компоненты, работа с redux)
- **navigation** - описание навигации и компоненты связанные с ней
- **types** - описание типов внешних зависимостей которые не имеют встроенной типизации

Стоит уделить особое внимание компонентам которые содержаться в *common/components*, т.к. некоторые из них заменяют некоторые уже существующие компоненты дополняя их определенным функционалом (*напр*. *FlatListWrapper, SectionListWrapper*).

Дополнительно необходимо ознакомиться с файлами из папки *core/theme,* где вы можете задавать цвета вашей схемы (*colors.ts*), описывать шрифты (*fonts.ts*) и часто используемые стили (*commonStyles.ts*), а также использовать различные полезные функции и константы (*common.ts*)

В целом рекомендуется ознакомиться с каждым файлом приложения, чтобы понимать что доступно и заранее реализовано.

Package.json, зависимости, скрипты

Здесь содержиться вся основная информация о ваших зависимостях, скриптах и дополнительных описаниях.

Содержимое

В нашем template вы можете увидеть минимально необходимый набор скриптов для работы с React Native:

- **preinstall** - общий скрипт который выполняется перед тем как вы производите команду yarn install. В нашем случае здесь происходит проверка на то, что вы используете yarn, а не npm. Это сделано с целью избежания проблем с версионностью, т.к. yarn и npm создают свои lock файлы
- **postinstall** - общий скрипт который выполняется после установки зависимостей. В нашем случае мы выполняем набор различных функций которые производят замены в зависимостях с целью устранения неполадок, изменения работы определенного функционала и т.п. из файла postinstallFixes.js. Если у вас мало опыта, то советуем не вносить свои изменения в этот файл.
- **ts** - выполняет компиляцию вашего кода. Сейчас из-за того, что мы напрямую работаем с JS кодом, в основном используется для проверки кода на наличие ошибок TypeScript, также удобен для работы с CI/CD.
- **start** - запуск packager’a
- **start-reset-cache** - запуск с packager’a с очисткой кэша. Используется в случаях если при работе с компонентами возникают трудности или он ведет себя неадекватно, либо при обновлении компонента у вас до сих пор остаётся его старая версия.
- **android:run** - установка вашего приложения на устройство
- **android:run+start** - установка вашего приложения на устройство с последующим запуском packager’a в одном и том же терминале
- **android:build-release-apk** - сборка релизного билда вашего приложения, по итогу которой ваш билд окажется в одной из следующих папок (Не забудьте, что для сборки релизной версии вашего приложения вам понадобиться добавить для него подпись):

    - Windows: "C:/tmp/${rootProject.name}/${project.name}"
    - Linux/macOS: "/tmp/${rootProject.name}/${project.name}"

- **android:clean** - очистка нативного содержимого вашего проекта, включая созданные APK из папок что описаны пунктом выше
- **android:bundle** - создание бандла для релизной или дебаг конфигураций (можно дописывать в самом скрипте)
- **ios:run** - установка вашего приложения на устройство (предпочтительно не пользоваться этим скриптом, а производить установку напрямую из XCode, чтобы видеть все ошибки, выбирать устройство для установки и т.п.)
- **check-dependencies-updates** - проверка ваших текущих зависимостей на наличие новых версий с целью последующего обновления
- **jest:run-tests** - запуск jest тестов из папки __tests__
- **jest:tests-watch** - запуск всех тестов с возможностью фокусировке на определенном наборе тестов
- **lint** - проверка на наличие ошибок линтера
- **config-setup** - установка environment, версии, номера билда и директории проекта в нативном коде и файлах настроек.
- **ResourcesGenerator** - производит генерацию путей к изображениям которые будут добавлены в папку /resources/images, после чего эти ресурсы будут доступны для обращения в коде при помощи файла ImageResources.g.ts

Зависимости

Здесь расположены, по нашему мнению минимальные зависимости которые достаточны для разработки любого проекта.

Для того чтобы добавить внешнюю зависимость вам нужно будет прочесть документацию той зависимости которую вы хотите подключить и следовать по всем пунктам инструкции как это сделать.

Если зависимость содержит работу с нативным кодом (требует линковки (linki) или просто вносит изменения в файлы нативного кода вашего проект, то для того чтобы эту зависимость можно было использовать, вам необходимо будет пересобрать проект на ваше устройство после установки и линковки

компонента. При работе с зависимостями с нативным кодом для iOS будет использоваться CocoaPods

Если зависимость не требует линковки и не затрагивает нативный код, то вы в режиме реального времени можете её установить и выполнить команду Reload на вашем устройстве, после чего компонент будет доступен для работы (в редких случаях потребуется перезапустить packager)

Вы могли заметить что в файле есть dependencies и devDependencies. Разница между этими двумя видами зависимостей состоит в том что dependencies это модули которые требуются в момент выполнения (runtime) вашего приложения, в то время как вторые нужны лишь во время разработки.

Устанавливаются такие модули также как и обычные модули, только добавляется дополнительный параметр --dev.

Существует также ещё несколько видов зависимостей, о них вы детально сможете прочесть в документе от npm. В ежедневной разработке они используются редко.

Если вы обнаружили ошибку во внешней зависимости и хотите исправить её, или же вы хотите подстроить зависимость под ваш компонент, то вам необходимо будет делать fork этого компонента и вносить изменения в вашем fork'е и ссылаться на него в зависимостях. Вместо версии напр. "1.0.0" вам нужно будет использовать ссылку которая будет выглядеть следующим образом "git+http://{ссылка на ваш компонент}.git#ветка". Если вы вносите изменение в вашем репозитории и хотите видеть эти изменения, то вам нужно будет вести версионность в данном компоненте.

Полезные ссылки:

- Установка зависимостей и параметры установки
- Работа с зависимостями
- CocoaPods

Компоненты

Концептуально, компоненты похожи на JavaScript-функции. Они принимают произвольные данные (называемые props) и возвращают React-элементы, которые описывают то, что должно появиться на экране.

Компоненты бывают “умными” (подключенные к Redux) и “тупыми” (не подключенные к Redux).

“Тупые” компоненты - обычные компоненты в которые принимают описанные пользователем данные, а также могут иметь свой локальный state.

Пример “тупого” компонента:

```
import React, {PureComponent} from "react";
import {Text, TextStyle, View, ViewStyle} from "react-native";
import {styleSheetCreate, styleSheetFlatten} from "../utils";
import {Colors, Fonts, isIos} from "../../core/theme";

// Данные которые поступают в компонент из
 родительского компонента
interface IProps {
    value: string;
    style?: ViewStyle;
}

export class Indicator extends PureComponent<IProps> {
    render(): JSX.Element {
        const {value, style} = this.props;

        return (
            <View style={styleSheetFlatten([styles.container, style])}>
                <Text style={styles.text} numberOfLines={1}>{value}</Text>
            </View>
        );
    }
}

// Стили компонента
const styles = styleSheetCreate({
    container: {
        paddingTop: isIos ? 4 : 2,
        paddingBottom: 4,
        paddingHorizontal: 7.5,
        alignContent: "center",
        justifyContent: "center",
```

```
        backgroundColor: Colors.greenish,
        borderRadius: 12,
    } as ViewStyle,
    text: {
        fontSize: 12,
        fontFamily: Fonts.medium,
        color: Colors.white,
        letterSpacing: 0,
    } as TextStyle,
});
```

“Умные” компоненты как правило являются страницами, либо отдельными уникальными элементами, которые подключены к Redux и получают определенную информацию из хранилища. Она также как и “тупые” компоненты могут иметь свой локальный state и иметь props’ы. Про подключение к Redux и работу с данными из хранилища подробно можно будет ознакомится в главе связанной с Redux.

Полезные ссылки:

- Часто используемые компоненты и API
- Жизненные циклы детально
- Когда использовать Component и PureComponent
- Компоненты и свойства в React
- Props
- State

JSX + FlexBox

JSX - это надстройка на JavaScript, которая позволяет использовать XML-подобный синтаксис в JavaScript. JSX рекомендуется использовать при написании компонентов, поскольку с помощью него проще представить DOM-модель, в коде, написанном на JSX, легко разобраться.

Основные функции JSX:

- Встраивание выражений в JSX
- Использовать JSX-выражения внутри операторов if и циклов for, присвоить его переменной, принимать в качестве аргумента и возвращать его из функции
- Определение атрибутов с JSX
- Определение дочерних модулей с JSX
- JSX предотвращает хакерские атаки

Официальная документация детально покрывает раздел работы с FlexBox с примерами и визуальной демонстрацией изменений.

Полезные ссылки:

- Официальная документация
- Описание JSX для React
- TypeScript и JSX

Redux

Redux — библиотека управления состоянием для приложений, написанных на JavaScript.

Она помогает писать приложения, которые ведут себя стабильно/предсказуемо, работают на разных окружениях (клиент/сервер/нативный код) и легко тестируемые.

Рекомендуется прочитать небольшую статью которая коротко и в картинках отражает всю суть Redux. Далее рассмотрим как происходит работа с Redux в нашем приложении:

Структура модуля

При создании модулей (во главе стоит страница) должна быть следующая структура файлов:

- modules
  - home
    - Home.tsx
    - homeActions.ts
    - homeReducer.ts
    - homeState.ts

Описание состояния приложения

Первым делом необходимо решить, какие данные будут храниться в состоянии приложения. Описать состояние каждого модуля в виде интерфейса и добавить поля в интерфейс состояния приложения.

```
// src/core/store/appState.ts
import {IAuthState, AuthInitialState} from "../modules/auth/authState";
import {IFriendsState, FriendsInitialState} from "../modules/friends/friendsState";
import {IHomeState, HomeInitialState} from "../modules/home/homeState";

export interface IAppState {
    auth: IAuthState;
    friends: IFriendsState
```

```
    home: IHomeState;
    ...
}

export function getAppInitialState(): IAppState {
    return {
        auth: AuthInitialState,
        friends: FriendsInitialState,
        home: HomeInitialState,
        ...
    };
}
```

Названия полей в объекте состояния потом будут использованы при создании корневого редьюсера.

Описание состояния модуля

Состояние каждого модуля изолированно в отдельном поле объекта состояния приложения. Необходимо описать интерфейс состояния модуля и создать объект начального состояния:

```
// описание состояния модуля
export interface IHomeState {
 counter: number;
 isLoading: boolean;
 message: string | null;
}

// значение начального состояния модуля
export const HomeStateInitial: IHomeState = {
 counter: 0,
 isLoading: false,
 message: null
};
```

Actions и Reducer модуля

Actions - это структуры, которые передают данные из вашего приложения в хранилище. Они являются единственными источниками информации для хранилища. Вы отправляете их в хранилище, используя метод *dispatch()*

Действия (Actions) описывают тот факт, что что-то совершилось, но не определяют, как в ответ изменяется состояние (state) приложения. Это задача для **редюсеров (reducers)**.

Далее нужно описать экшены и редьюсер модуля. Чтобы облегчить эту задачу используем typescript-fsa и typescript-fsa-reducers

Например:

Создание action'ов:

```
import {actionCreator} from "../../core/store";
import {IGrowArgs} from "../../core/api/generated/dto/home/GrowArgs";
import {IGrowResult} from "../../core/api/generated/dto/home/GrowResult";

export class HomeActions {
   // Экшен для асинхронных операций:
Home/GROW_COUNTER_STARTED, Home/GROW_COUNTER_DONE, Home/GROW_COUNTER_FAILED
   static growCounter = actionCreator.async<IGrowArgs, IGrowResult,
Error>("Home/GROW_COUNTER");
   // Простой экшен для атомарного действия
   static growCounterProgress =
actionCreator<number>("Home/GROW_COUNTER_PROGRESS");
}
```

Создание reducer'a:

```
import {Failure, Success} from "typescript-fsa";
import {reducerWithInitialState} from "typescript-fsa-reducers";
import {HomeStateInitial, IHomeState} from "./homeState";
import {IAppState} from "../../core/store/appState";
import {CoreActions} from "../../core/store";

// обработчик экшена rehydrate - происходит на старте
 приложения, после успешного получения данных из
 хранилища
function rehydrateHandler(state: IHomeState, rehydratedState: IAppState):
IHomeState {
   const nState = rehydratedState.home || state;

   return newState(nState, {isLoading: false, counter: 0});
}

// обработчик экшена Home/GROW_COUNTER_STARTED - начало
 асинхронной операции
function startedHandler(state: IHomeState, args: IGrowArgs): IHomeState {
   return newState(state, {isLoading: true});
}
```

```
// обработчик экшена Home/GROW_COUNTER_DONE - успешное
 окончание асинхронной операции
function doneHandler(state: IHomeState, success: Success<IGrowArgs, IGrowResult>):
IHomeState {
   return newState(state, {isLoading: false, counter:
success.result.resultCounter});
}
// обработчик экшена Home/GROW_COUNTER_FAILED - ошибка
 выполнения асинхронной операции
function failedHandler(state: IHomeState, failure: Failure<IGrowArgs, Error>):
IHomeState {
   return newState(state, {isLoading: false, message: failure.error.message});
}
// обработчик экшена Home/GROW_COUNTER_PROGRESS -
function growCounterProgressHandler(state: IHomeState, value: number): IHomeState {
   return newState(state, {counter: value});
}
// создание редьюсера из описаний экшеном и их
 обработчиков
export const homeReducer = reducerWithInitialState(HomeStateInitial)
   .case(CoreActions.rehydrate, rehydrateHandler)
   .case(HomeActions.growCounter.started, startedHandler)
   .case(HomeActions.growCounter.done, doneHandler)
   .case(HomeActions.growCounter.failed, failedHandler)
   .case(HomeActions.growCounterProgress, growCounterProgressHandler);
```

Корневой reducer

Корневой reducer комбинирует все reducer’ы которые используются в приложении

```
import {Reducer} from "redux";
import {IAppState} from "./appState";
import {Reducers} from "./Reducers";
import {authReducer} from "../modules/auth/authActions";
import {friendsReducer} from "../modules/friends/friendsActions";
import {homeReducer} from "../modules/home/homeActions";

// структура передаваемого объекта
 аналогична структуре корневого
 объекта состояния приложения appState.ts
// каждый редьюсер модуля отвечает
 только за часть состояния приложения
```

```
// при этом каждый экшен может быть обработан в любом редьюсере
export function createMainReducer(combineMethod: (reducers: any) =>
Reducer<IAppState>): Reducer<IAppState> {
    const reducers: Reducers<IAppState> = {
        auth: authReducer,
        friends: friendsReducer,
        home: homeReducer
    };

    return combineMethod(reducers);
}
```

Отображение данных состояния приложения

Любой компонент приложения может напрямую подключиться к redux хранилищу при помощи декоратора **connectAdv**.

Чтобы изменить состояние приложения необходимо задиспатчить экшен в стор состояния:

```
// Описание получаемых данных из хранилища
interface IStateProps {
    counter: number;
}

// Описание действий
interface IDispatchProps {
    growCounterProgress(value: number): void;
}

// Подключение к хранилищу состояния, подписка на изменение состояния
@connectAdv(
    // Выборка данныхе для компонента из хранилища
```

```
    (state: IAppState): IStateProps => ({
        counter: state.home.counter,
    }),
    (dispatch: Dispatch): IDispatchProps => ({
        growCounterProgress: (value: number): void => {
            dispatch(HomeActions.growCounterProgress(value));
        }
    })
)
export class HomePage extends BaseReduxComponent<IStateProps, IDispatchProps> {
    render(): JSX.Element {
        // Данные counter из хранилища
        const {counter} = this.stateProps;

        return (
            <TouchableOpacity onPress={this.growCounterProgress}>
                <Text>
                    {counter}
                </Text>
            </TouchableOpacity>
        );
    }

    private growCounterProgress = (): void => {
        // Вызов действия для изменения
 состояния
        this.dispatchProps.growCounterProgress(this.stateProps.counter + 1);
    };
}
```

Работа с бизнес-логикой (API)

Работа с бизнес-логикой содержит в себе ранее знакомые операции, добавляется лишь только запрос на внешнее API:

```
export class HomeActionsAsync {
    static growRemoteCounter(currentValue: number): SimpleThunk {
        return async (dispatch: Dispatch): Promise<void> => {
            const params: GrowRequest = {value: currentValue};

            try {
                dispatch(HomeActions.growCounter.started(params));
                // Запрос внешнего API с ответом
```

```
                const result = await requestsRepository.homeApiRequest.grow(params);
                dispatch(HomeActions.growCounter.done({params, result}));
            } catch (error) {
                dispatch(HomeActions.growCounter.failed({params, error}));
            }
        };
    }
}
```

Детально о работе с Redux вы сможете прочесть или просмотреть информацию по следующим ссылкам:

- Redux (русский)
- Видео курс от создателя Redux Дэна Абрамова
- Почему Reducer должен быть чистой функцией в Redux
- Ещё одно объяснение Redux

Навигация

Основным компонентом для работы навигации внутри приложения является react-navigation.

Существует четыре типа навигации:

- **StackNavigator** - переход вперед от экрана экрана к экрану и обратно
- **StackNavigator**(mode: "modal") - отображение экранов в модальном режиме (поверх текущего окна)
- **DrawerNavigator** - боковое выезжающее меню (hamburger menu)
- **TabNavigator** - переключение между экранов при помощи табов в произвольном порядке

Настройки элементов навигации не следует смешивать с кодом компонентов экранов.

В идеале в коде экранов не должно быть информации о том каким образом он появляется в структуре навигации приложения.

Поэтому элементы навигации должны быть расположены в отдельной папке:

- /navigation
    - **components** - набор часто используемых компонентов для навигации
    - **config** - описание создания и подключение навигации + начальный конфиг навигации
    - **configurations** - описание каждого типа навигации с его страницами, reducer'ом и т.д.
    - **actions.ts** - набор действий для переходов
    - **pages.ts** - описание страниц

В первую очередь перейдем в файл pages.ts и опишем наши страницы:

```
export class Pages {
    login = "login";
```

```
    main = "main";

    playground = "playground";
    inDeveloping = "inDeveloping";
}
```

Далее добавим новые страницы во внутрь навигатора (*rootNavigationConfiguration.ts*):

```
export const RootNavigator = createStackNavigator({
    [NavigationPages.login]: {screen: AuthPage},
    [NavigationPages.main]: {screen: MainPage},
    [NavigationPages.playground]: {screen: Playground},
    [NavigationPages.inDeveloping]: {screen: InDeveloping},
}, {
    headerMode: "screen",
    cardStyle: {
        backgroundColor: isIos ? Colors.white : Colors.transparent
    },
});
```

Опишем переходы на эти страницы (*actions.ts*):

```
 export class Actions {
     toggleDrawer = toggleDrawer();
     closeMenu = closeMenu();

     navigateToInDevelopment = simpleToRoute(NavigationPages.inDeveloping);
     navigateToPlayground = simpleToRoute(NavigationPages.playground);
     navigateToMain = simpleToRoute(NavigationPages.main);
     navigateToAuth = simpleToRoute(NavigationPages.login);

     navigateToBack = (): SimpleThunk => {
         return async (dispatch, getState): Promise<void> => {
             const backAction = getBackAction(getState().navigation);

             if (backAction != null) {
                 dispatch(backAction);
             }
         };
     };

     internal = {
         backInRoot: actionCreator("AppNavigation/BACK_IN_ROOT"),
     };
 }
```

Теперь на страницах вы можете вызывать переход при помощи описанного action’a:

```
interface IDispatchProps {
    navigateToMain(): void;
}


@connectAdv(
    (dispatch: Dispatch): IDispatchProps => ({
        navigateToMain: (): void => {
            dispatch(NavigationActions.navigateToMain());
        },
    }))
export class Login extends BaseReduxComponent<IStateProps, IDispatchProps> {
```

Работа с запросами

Для работы с запросами, в нашем template выделена отдельная папка **core/api** внутри которой содержится информация о запросах, описание интерфейсов, различные функции, а также файл-обработчик, отвечающий за формирование запроса и обработку ответа.

Пройдем весь путь формирования запросов, в качестве примера рассмотрим два запроса на авторизацию и логаут:

1. Добавим файл (*MobileUserApiRequest.ts*) в папку *core/api/generated/dto* который будет непосредственно содержать наши запросы:

```
/*tslint:disable*/
import {BaseRequest} from "../BaseRequest";
import {LoginRequest} from "./dto/user/LoginRequest.g";
import {LogoutRequest} from "./dto/user/LogoutRequest.g";
import {LogoutResponse} from "./dto/user/LogoutResponse.g";
import {UserProfile} from "./dto/user/UserProfile.g";

export class MobileUserApiRequest extends BaseRequest {
    constructor(protected baseurl: string) {
        super();
        this.login = this.login.bind(this);
        this.logout = this.logout.bind(this);
    }
    login(request: LoginRequest, config?: Object): Promise<UserProfile> {
        return this.fetch(
            `/user/login`,
            Object.assign({
                method : "POST",
                body:  JSON.stringify(request)
            }, config))
            .then((response) => response.json())
            .catch(BaseRequest.handleError);
    }
    logout(request: LogoutRequest, config?: Object): Promise<LogoutResponse> {
        return this.fetch(

`/user/logout?lang=${request.lang}&city_id=${request.city_id}&access_token=${req
uest.access_token}`,
            Object.assign({
                method : "DELETE",
            }, config))
            .then((response) => response.json())
            .catch(BaseRequest.handleError);
```

```
        }
    }
```

2. Добавим ранее созданный класс в общий “репозиторий запросов” (*core/api/generated/RequestsRepository.g.ts*):

```
import {MobileUserApiRequest} from "./MobileUserApiRequest.g";

export class RequestsRepository {
  constructor(private baseurl: string) {
  }

  mobileUserApiRequest = new MobileUserApiRequest(this.baseurl);
}
```

3. Опишем интерфейсы которые используются в запросах (*LoginRequest, UserProfile, LogoutResponse, LogoutRequest*), внутри папки *core/api/generated/dto* создадим папку *user* внутри которой создадим для каждого интерфейса свои файлы, для примера создадим только один файл LoginRequest.g.ts:

```
/*tslint:disable*/

export interface LoginRequest {
    phone: string;
    password: string;
}
```

4. Теперь мы можем использовать этот запрос в наших асинхронных action’ах:

```
export class AuthAsyncActions {
    static login(phone: string, password: string): SimpleThunk {
        return async function (dispatch: Dispatch): Promise<void> {
            const params: LoginRequest = {phone, password};

            try {
                dispatch(AuthActions.login.started(params));
                const result = await
requestsRepository.mobileUserApiRequest.login(params);
                dispatch(AuthActions.login.done({params, result}));

            } catch (error) {
                dispatch(AuthActions.login.failed({params, error}));
            }
        };
    }
}
```

За различную обработку ответов запросов и формирование самих запросов отвечает файл *core/api/BaseRequest.ts*, который может совершать определенные действия связанные с ответом.

Настройки и среда разработки (dev, test, stage, prod)

Для работы с различными настройками и переключениями между средами где будет работать ваше приложение, были созданы несколько конфигурационных файлов позволяющих вам самостоятельно заполнять свои параметры или изменять уже существующие параметры.

Файлы расположены в папке: */resources/settings*

Корневым файлом является *mobileSettings.json.*

Внутри этого файла задаются все значения для всех параметров типы которых описаны в отдельном файле (*/src/core/settings/appSettings.ts)*. Эти параметры впоследствие объединяются с файлами той конфигурации которая выбрана сейчас (*environment*).

Для удобства разработки, также есть файл *localSettings.json*, в котором указываются значения которые будут использоваться на момент разработки, т.е. при сборке билдов эти настройки не будут учитываться.

Пройдемся по каждой настройке, которая предоставляется из коробки:

- **appName** - название пакета приложения
- **environment** - среда разработки (для каждой среды, в коде уже предусмотрены различные сценарии работы в отдельных компонентах)
- **serverUrl** - ссылка сервера для работы с API
- identity
- **bugReportApiKey** - ключ от сервиса для сбора ошибок Bugsnag
- **useBugReporter** - должен ли использовать сервис сбора ошибок или нет
- **version** - версия приложения, используется для внутреннего отображения на экранах приложения
- **build** - номер билда
- **showVersion** - должна ли отображаться версия или нет
- **supportUrl** - ссылка для поддержки
- **devOptions** - параметры которые помогают при разработки приложения

- **startScreen** - стартовый экран для работы с приложением (чтобы при reload'е приложения без fast reload'а отображался тот экран который вам требуется)
- **useTestCase** - позволяет внутри кода отталкиваться от этого значения с целью создания определенных тестовых случаев для различных сценариев
- **reduxLogger** - параметры для redux logger'а который добавляет записи в консоль разработчика в Google Chrome при удаленном дебагге
- **reduxLoggerWhiteList** - white list значений которые должны попадать в logger
- **purgeStateOnStart** - очистка (purge) redux store при обновлении (reload) приложения
- **showAllComponentsOnStart** - отображение страницы Playground на момент запуска приложения
- **disableReduxLogger** - отключение работы redux logger'а

Для создания собственных параметров, вам следует перейти в файл с описаниями интерфейсов */src/core/settings/appSettings.ts* добавить те параметры которые вы хотите, и добавить начальное значение в *mobileSettings.json*

Для использования текущих параметров, вы можете использовать константу ***appSettingsProvider***, например:

```
 if (appSettingsProvider.settings.environment == "Production") {
            return null;
        } else {
            return (
                <View style={styles.container}>
                    <TouchableWithoutFeedback onPress={this.showMenu}>
                        <View style={styles.buttonStyle}>
                            <PopupMenu actions={this.testUsersActions}
ref={this.menu.handler}/>
                            {this.props.children}
                        </View>
                    </TouchableWithoutFeedback>
                </View>
            );
        }
```

Отладка

Для работы с отладкой вашего приложения вы можете использовать следующие средства и возможности:

- Удаленная отладка в Google Chrome
- React DevTools
- Отладка нативного кода в XCode и Android Studio

Также начиная с React Native 0.61 у вас есть возможность видеть вывод различных методов класса console в терминале packager'а.

В нашем template, учтено логирование состояний хранилища Redux после совершения action'ов, т.е. вы сможете видеть предыдущее, текущие изменения и изменённое состояние хранилища после того как выполните dispatch action'а

ВАЖНО! При удаленной отладке работа приложения может отличаться!

На ваших девайсах, без удаленной отладки используются JavaScript движки от платформ (JavaScriptCore на iOS и Hermes на Android), но при удаленной отладке используется движок от Google Chrome - V8

Обращайте особое внимание на работу с датами, они будут отличаться в первую очередь.

Для вывода информации в консоль используется console.
Основные методы которые пригодятся при отладке:

- log - делает обычный вывод в консоль
- warn - делает вывод в консоль и дополнительно отображает информацию в самом приложении.

Полезные ссылки:

- Официальная документация
- Детальное рассмотрение средств для отладки
- Полезная статья связанная с отладкой

Производительность

Для устранения проблем с производительностью связанных с определенными компонентами рекомендуется детально читать документацию компонента, в большинстве случаев она содержит множество параметров которые исправляют эти проблемы.

Хорошим примером в этом случае является **FlatList** который наследуется от **VirtualizedList** где вы можете встретить множество параметров которые улучшают работу списка, например:

- getItemLayout
- removeClippedSubviews
- keyExtractor
- windowSize

Для общих проблем связанных с общей работой, не завязанной на определенные компоненты, Facebook предоставляет множество советов и возможностей в плане производительности.

Общие советы связанные с производительностью:

- Используйте как можно меньше компонентов в render
- Используйте PureComponent
- Не делайте анонимные вызовы внутри параметров, callback'ов и т.п.
- Для работы со стилями используйте styleSheetCreate, styleSheetFlatten
- При работе с анимацией, используйте useNativeDriver
- Избегайте вычислений в render
- Используйте key для элементов списков и keyExtractor
- Не забывайте использовать React.Fragment

Правила написания кода

Следуя следующим простым правилам ваш код будет лаконичным и понятным к восприятию.
Эти правила не являются обязательными, но следуя им вы будете уверены в вашем коде:

1. Отсутсвие ошибок TypeScript'а и TSLint'а
2. Отсутсвие Warning'ов (только зависящие от разработчика)
3. Форматирование кода
4. Названия переменных, методов и т.д. должны быть короткие и отображать то что они выполняют
5. Отсутствие копирования компонентов
6. Код не должен приводить к падениям (проверка написанного кода)
7. Общие стили должны находиться в файле CommonStyles
8. Практически все не redux компоненты должны быть PureComponent'ами
9. Reducer'ы должны быть чистыми функциями
10. Selector'ы должны выбирать только используемые используемые параметры из state'а, а не весь state
11. State должен быть уникальным (отсутсвие копирования state'а)
12. State должен содержать только логически связанные с ним элементы
13. Количество строк в методах и функциях не должно превышать 30 строк
14. Отсутствие закомментированного кода
15. Использовать комментарии только в случае если алгоритм тяжелый для понимания
16. Использовать //TODO для описания незавершенности компонента, метода и т.д.
17. Корректное типизирование кода (уменьшение использования any)
18. Полное отсутствие мутирования redux state'а
19. Копирование массивов должно осуществляться следующим образом: [...your_array]
20. Отсутствие неиспользуемого кода
21. Функция должна выполнять только одну операцию
22. Версия компонентов в package.json должна быть стабильная и неизменяющейся
23. Стараться ежедневно следить за обновлениями компонентов
24. Писать код таким образом, чтобы его можно было тестировать

Обновление React Native

Обновление React Native, может происходить как вручную, так и автоматически.

Обновления желательно производить в отдельных ветках, чтобы не навредить текущему состоянию проекта и чтобы при любой возможности можно было откатить изменения.

При установке обновлений, старайтесь делать удаление ресурсов которые могут кэшировать текущее состояние, например:

- Удаляйте node_modules
- Удаляйте pods
- Выполняйте скрипт android:clean

Мы советуем производить обновления вручную, чтобы самостоятельно можно было убедиться, что всё прошло без проблем, а также узнать какие именно изменения вносятся в нативную сторону проекта.

Для ознакомления с обновлениями используются два репозитория:

- react-native
- react-native-releases

Если после обновления у вас возникают ошибки, то решение может быть найдено среди issue обоих репозиториев. В частности, react-native-releases создает issue при каждом релизе где ведется дискуссия касательно текущей и следующей версии React Native.

Тестовое задание

Для проверки и закрепления материала, предлагаем выполнить тестовое задание:

Требуется реализовать приложение для кофейни.

Требования:
Дизайн приложения должен быть максимально схож с представленным ниже.
Авторизация и данные для отображения должны получатся от сервера.

Приложение должно работать на следующих платформах:

- Android (начиная с версии 5.0)
- iOS (начиная с версии 9.0)

Необходимо предусмотреть уникальные для платформы жесты, компоненты и т.п. (напр. нажатие твердотельной (solid-state) кнопки назад на Android).

При реализации стараться использовать последние версии компонентов.

Ссылки:

- Дизайн
- API
- Swagger для проверки запросов и ответов от сервера

Полезные ссылки

- Redux-Saga аналог Thunk’ов с большими возможностями
- Scoping и Hoisting в JavaScript
- MDN JavaScript
- You Don’t Know JS Yet (серия книг)
- Native Directory - курируемый список библиотек React Native
- Примеры анимаций в React Native
- React Native CheatSheet
- JavaScript CheatSheet
- Набор готовых компонентов
- AirBnB JavaScript Style Guide
- Google JavaScript Style Guide
- Idiomatic JavaScript Style Guide
- Standard JavaScript Style Guide
- Описание работы с анимацией
- TypeScript Deep Dive
- Курируемый список компонентов

Теперь Вы готовы ко второму этапу!

(углубленная практика)

office@dex-it.ru

777 783 35