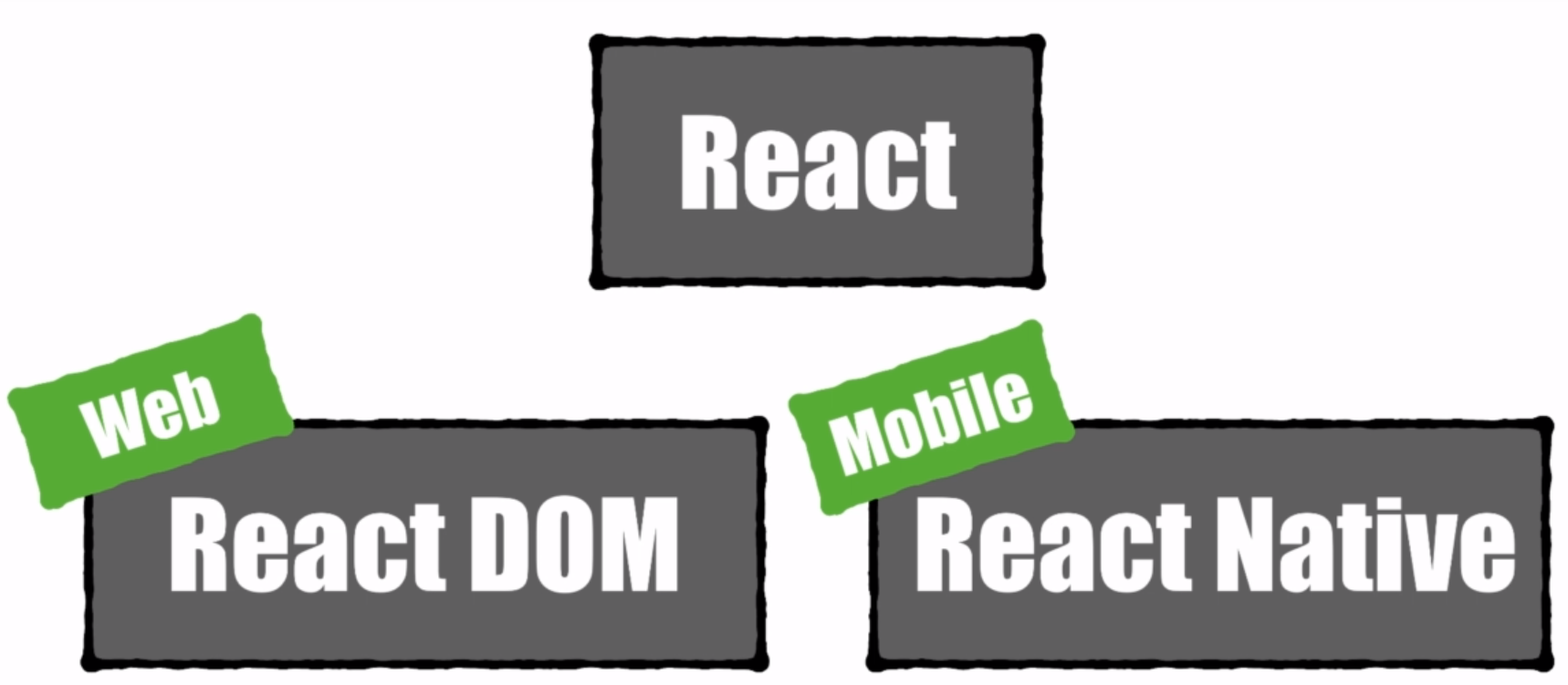
***REACT NATIVE***

*JS – ECS 6*

# Introdução

## O que é?

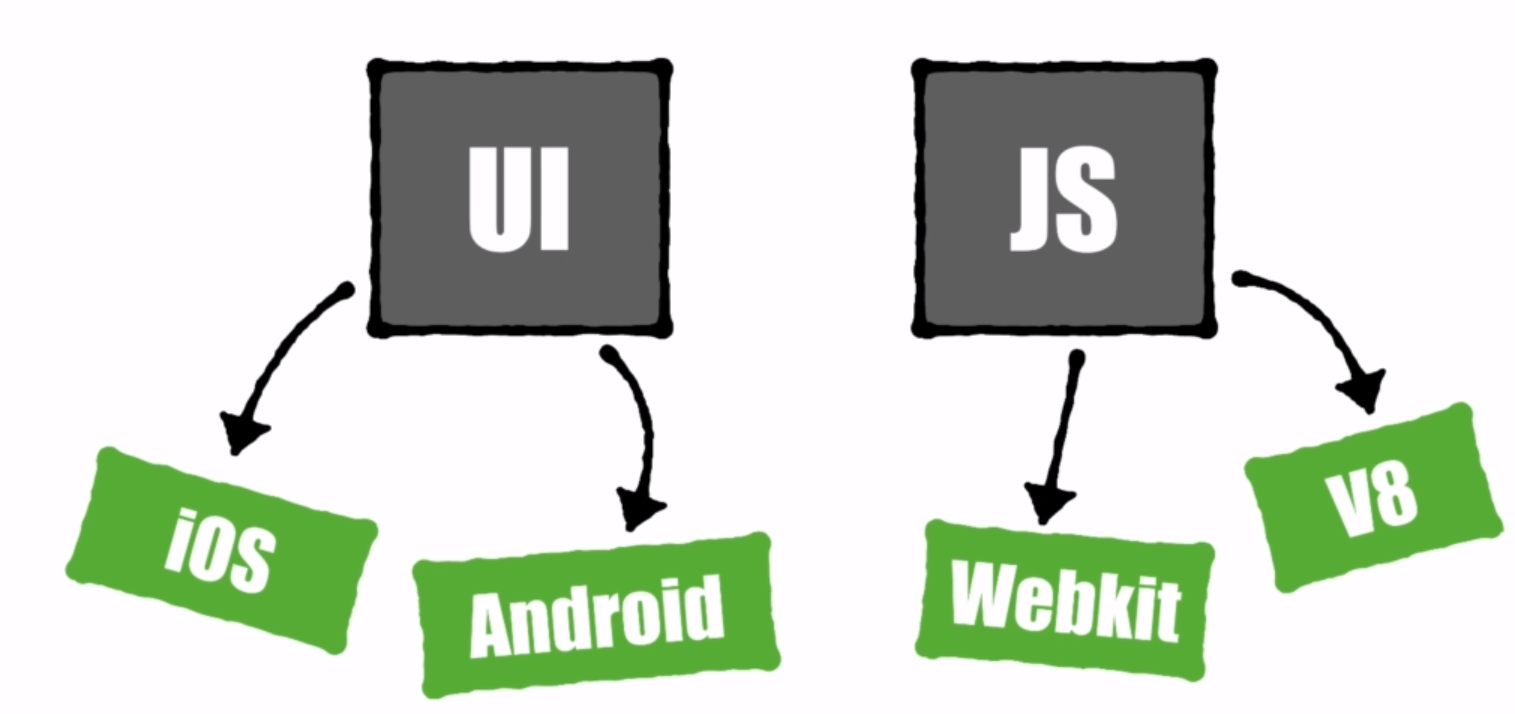


React (core): parte central, cuida da componentização.

Serve tanto para aplicações web (React DOM), como para mobile (React Native).

Curso com foco no core da biblioteca que serve tanto para web e mobile.

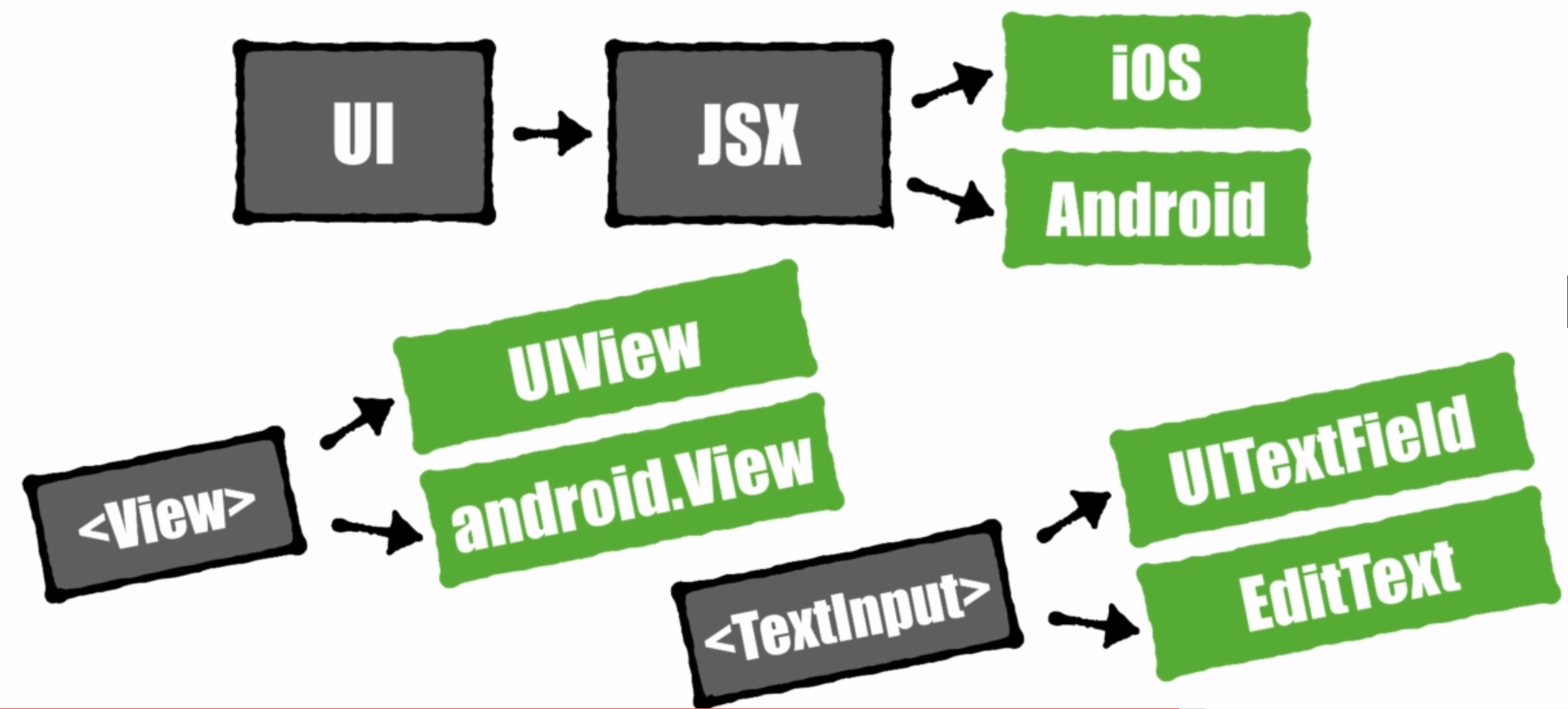
## Como funciona?



Interface Gráfica (UI – User Interface): desenvolvida com JSX (o mesmo para web).

JavaScript – JS: lógica: NÃO É CONVERTIDO PARA A LINGUAGEM DO SO – android e swift. Os runtimes são chamados em cada dispositivo para executar o código JS e a comunicação com os recursos nativos serão acessados via websocket.

JSX: o JSX criado vai gerar componentes nativos tanto para IOS quanto para Android.



Os componentes visuais são componentes nativos, você escreve uma única vez as tags.

A interface gráfica é definida utilizando JSX. O JSX é convertido para os componentes visuais de cada uma das plataformas (IOS e android).

Uma View em uma aplicação React Native é convertido para UIView (ios) e android.View (android).

Se fosse web, a mesma view seria uma div, o TextInput é convertido para UITextField

# Revisão JS

JS NÃO É NECESSÁRIO USAR ; NÃO MUDA EM NADA, FUNCIONA PERFEITAMENTE

## Fundamentos

### Var, Let e Const

Utilizar let é o mais adequado.

O let não pode ser redeclarado, mas o var sim.

É altamente recomendável que as variáveis sejam const, sempre que a variável for mudar, utilize o let.

### Função VS Objeto

O tipo dos objetos é função.

Typeof new Object -> é objeto, pois há uma instanciação da função gerando um objeto.

Classes são funções.

Funções podem ser instanciadas.

### Par Nome/Valor - Escopos

Escopo/Contexto: você pode definir uma variável em um contexto maior, mas também pode definir outra variável com o mesmo nome em um contexto menor, ele sempre vai pegar do contexto/escopo em questão e se não achar procura em contextos mais abrangentes.

### Operador Destructure

Operador de desestruturação: tira da estrutura (ex: objeto – forma de extrair do objeto seus atributos **<{}>**, array – tirar elementos do array **<[]>**) alguma coisa.

* Objeto {}:

const pessoa = {

nome: 'Carla',

Dessa forma você consegue extrair vários atributos do objeto em uma única linha.

É possível extrair também dados de objetos em objetos.

idade: 19,

endereco: {

rua: 'Rua',

bairro: 'Bairro'

}

}

const {nome:n, idade:i} = pessoa

const {nome:n, idade:i, endereco} = pessoa

const {endereco : {nome:n\_end, rua:r\_end, bairro:b\_end}} = pessoa

console.log(n\_end, r\_end, b\_end)

const { sexo, estado\_civil = 'solteira' } = pessoa

Se você tentar acessar um atributo que não existe, irá retornar undefined, ou você pode atribuir um valor padrão.

console.log(sexo, estado\_civil)

Para dados aninhados, o caminho completo deve existir se não dá erro.

* Array []:

const a = [1,2,3]

const [b, , d, e=0, f] = a

console.log(b, d, e, f) //1,3,0,undefined

const [, [, nota]] = [[, 8, 8] , [9, 6, 8]]

console.log(nota) //6

* Funções
  + Objetos

function rand( {min = 0, max = 1000} = {}){ //desestruturação

    const valor = Math.random() \* (max - min) \* min

    return Math.floor(valor)

}

const obj = { max: 50, min: 40 }

console.log(rand(obj))

console.log(rand({max: 955}))

console.log(rand({}))

console.log(rand())

* + Arrays

function rand( [min = 0, max = 1000] = []){ //desestruturação

    if(min > max) [min, max] = [max, min]  //muda os valores da variável utilizando desestruturação a partir da criação de um array com max, min

    const valor = Math.random() \* (max - min) + min

    return Math.floor(valor)

}

console.log(rand([50, 40]))

console.log(rand([992]))

console.log(rand([, 10]))

console.log(rand([]))

console.log(rand())

## Funções

### This

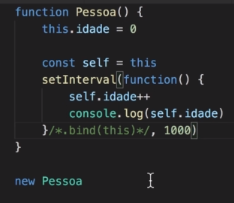
Em funções normais (function) sempre é o objeto que está sendo referenciado no contexto de execução, dono da função que está sendo executada naquele contexto, depende, é variável. Varia de acordo com a forma que você chama a função.

O this não varia nunca em arrow function, ele é definido no momento que a função é definida (this léxico – contexto/escopo em que a função foi escrita). É o dono, arquivo em que foi definido.

### Bind

É o método responsável por amarrar um determinado objeto para ele ser o dono da execução daquele método sempre que ele for chamado. Passa um objeto no qual você deseja que seja resolvido o this, pra quem você irá direcionar o this. Sempre que você for utilizar o this, ele será o objeto que você amarrou passando por parâmetro no bind.





Exemplo utilizando o bind e o artifício do self(menos adequado).

O .bind é utilizado pois funções em JS possuem um conjunto de funções atreladas em que o bind é uma delas.

### Arrow Functions

* **Sintaxe mais reduzida, gera funções pequenas**
* **This associado ao contexto em que a função foi escrita** (maior previsibilidade do código). A chamada da função de locais diferentes não influencia no this.
  + Retira a palavra reservada function e substitui por =>.
  + É uma função anônima.
  + Se você quiser chamar a função depois, você deve armazená-la em uma variável ou constante.
  + Pode tirar os parênteses do parâmetro quando ele é apenas um.
  + Quando o corpo da função é de apenas uma linha, pode tirar as chaves e o return, o return é implícito.
  + Quando a função não possuir parâmetro basta colocar (). É possível utilizar o \_, porém ele cria um parâmetro.

let dobro = function (a) {

    return a \* 2

}

dobro = (a) => {

    return a \* 2

}

dobro = a => a \* 2

console.log(dobro(2))

let ola = () => 'Olá, tudo bem?'

console.log(ola())

* exemplo this:

function Pessoa() {

    this.idade = 0

    setInterval(() => {

        this.idade++

        console.log(this.idade)

    }, 1000)

}

new Pessoa

**Não é possível mudar a referência do this no arrow function utilizando o bind. Sempre será do contexto léxico em questão.**

### Funções Anônimas

São funções sem nome.

Você joga a função em uma variável e chama a partir dela.

const soma = function (x, y) {

    return x + y

}

const imprimirResultado = function (a, b, operacao = soma) {

    console.log(operacao (a, b))

}

imprimirResultado(3,4)

Funções arrow são funções anônimas

imprimirResultado(3, 4, soma)

imprimirResultado(3, 4, function (x, y) {

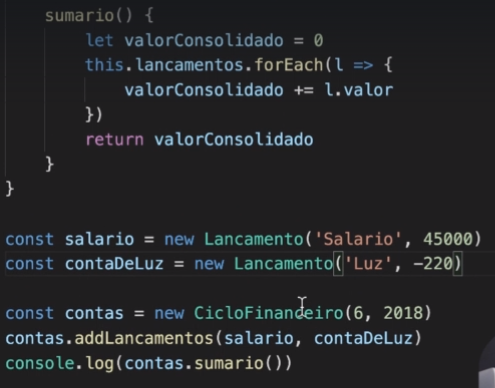
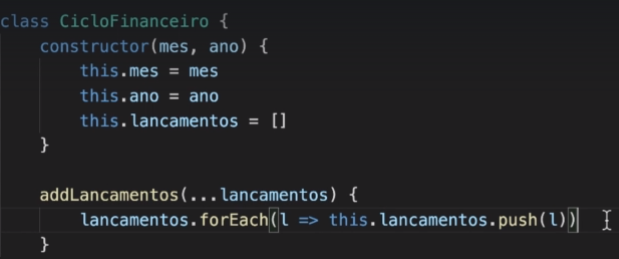
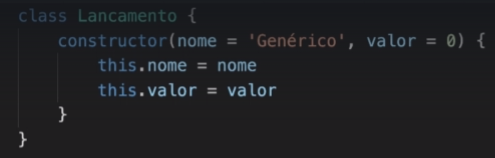
    return x - y

})

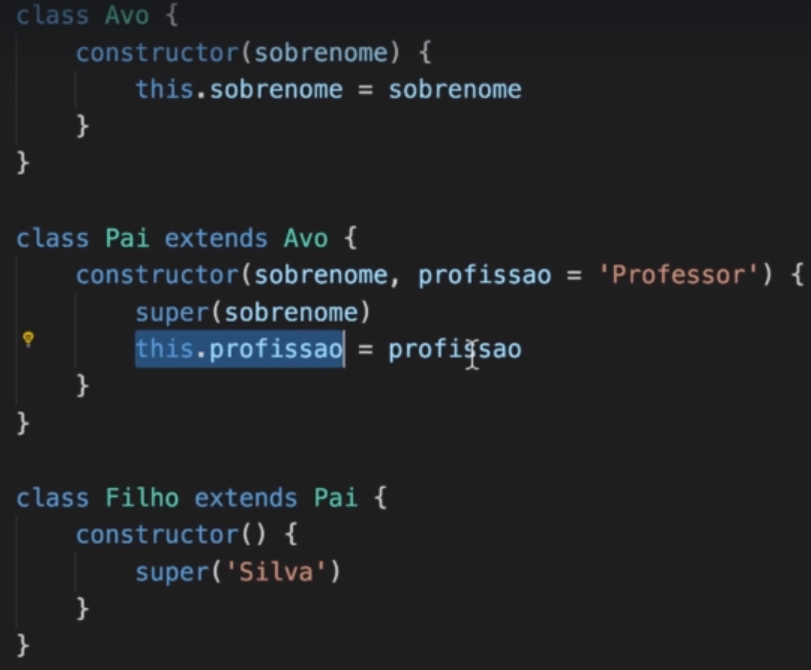
imprimirResultado(3, 4, (x, y) => x \* y)

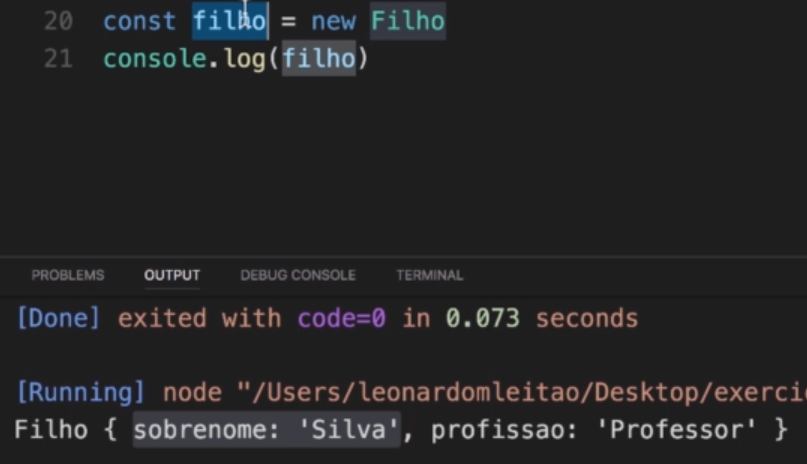
## Classes

Classes são convertidas para funções.



A herança funciona via protótipo de funções.





## Arrays

Arrays são objetos específicos. Diferente dos objetos {chave, valor}, os arrays são indexados, índices começando de 0.

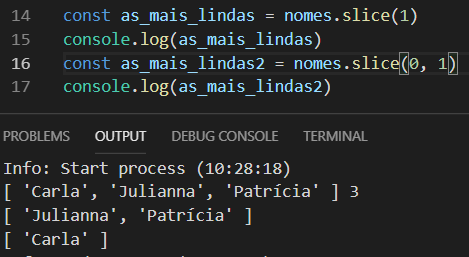
* É uma estrutura dinâmica, cresce e diminui dinamicamente.
* É uma estrutura heterogênea, os dados aceitos são variados, não é tipado, podendo aceitar inclusive outros arrays. Porém a boa prática indica que se deve trabalhar com dados homogêneos em arrays para facilitar a manipulação/compreensão...
* Arrays constantes podem ser modificados (adicionar/remover elementos) eles não podem apontar para um novo endereço de memória (instanciar novamente).
* Adicionar uma função ao array: Array.prototype.nomeFuncao = ...

### Funções

* Push() -> adiciona um elemento no final do array.
* Sort() -> ordena o próprio array (não gera um novo) em ordem crescente.
* Delete veotr[x] -> deleta o elemento na posição informada deixando-a undefined (sem fazer a reoganização).
* Splice -> incluir/excluir, incluir e excluir elementos em um determinado índice. Parâmetros: índice, quantidade de elementos que você deseja excluir a partir do índice, quantos elementos você deseja adicionar.



* Pop() -> remove o último elemento do array.
* Shift() -> remove o primeiro da lista, sem deixar undefined.
* Unshift(valor) -> adiciona no início da lista.
* Slice(indice) <pedaço> -> retorna um novo vetor a partir do índice informado.



Slice() pega a partir do índice informado ou em um intervalo de n, n-1 (não pega o último informado).

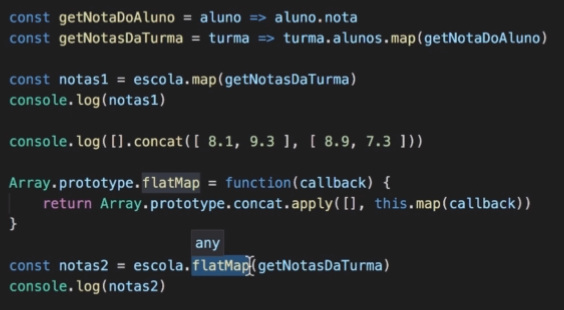
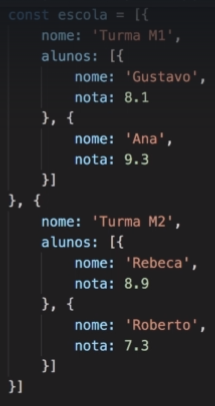
* Concat(valores) -> concatena vários arrays/outros elementos em um único array. Os arrays originais são mantidos gerando um novo array que é o resultado da concatenação.

const array = ['a', 'b'].concat([1, 2], [3, 4], 5, [[6, 7], [8, 9]], 'c')

console.log(array, array.length)



* FlatMap() -> função que não existe, é criada, para pegar uma matriz e achatar em uma única dimensão.





### Formas de percorrer o array

#### Foreach

Percorre todos os elementos do array.

const nomes = ['Ana', 'Marcela', 'Júlia', 'Josefina']

console.log(nomes)

const imprimir = (nome, indice, array) => console.log(nome, indice, array)

nomes.forEach(imprimir)

A função call-back chamada no forEach possui apenas os seguintes atributos (3):

Nome: valor do índice

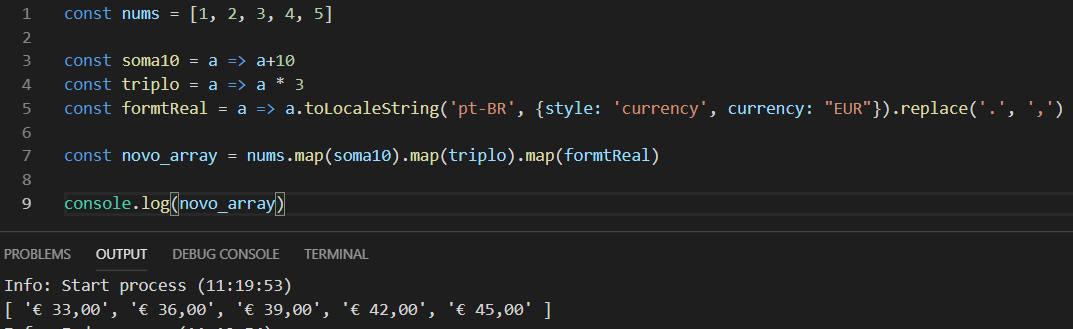
Índice: índice associado

Array: array completo

#### Map

É um for com propósito.

Executa uma transformação nos dados, mapeia o array para um novo array do mesmo tamanho. Pode ser chamado sequencialmente.

Constrói um novo vetor a partir de um dado selecionado.

#### Filter

Filtra um array para gerar um sub-array baseado em um critério de filtro.

O retorno é um vetor de tamanho variável que pega os valores que retornam true na expressão lógica.

const produtos = [

    { nome: "Notebook", preco: 2499, fragil: true},

    { nome: "iPad Pro", preco: 4199, fragil: false},

    { nome: "Copo de Vidro", preco: 12.49, fragil: true},

    { nome: "Copo de Plástico", preco: 18.99, fragil: false}

]

console.log(produtos.filter( a => a.fragil && a.preco >= 500 ))

const caro = a => a.preco >= 500

const fragil = a => a.fragil

console.log(produtos.filter(caro).filter(fragil))



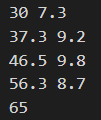
#### Reduce

Há a presença de um parâmetro (acumulador) que vai sendo acumulado de chamada em chamada, o resultado da função reduce é passado sempre para a próxima iteração, em que você tem o elemento atual e o acumulador.

A primeira iteração você tem os primeiros dois elementos do array (podendo passar, ou não, um valor inicial).

É uma forma de agregar/reduzir valores.

const alunos = [



    { nome: 'João', nota: 7.3, bolsista: false },

    { nome: 'Maria', nota: 9.2, bolsista: true },

    { nome: 'Pedro', nota: 9.8, bolsista: false },

    { nome: 'Ana', nota: 8.7, bolsista: true }

]

resultado = alunos.map(a => a.nota).reduce(function (acumulador, atual){

    console.log(acumulador, atual)

    return acumulador + atual

Realiza a soma de todas as notas dos alunos.

Passagem do valor 30 como valor inicial.

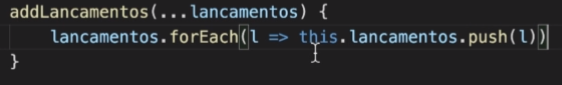
}, 30)

console.log(resultado)

O resultado pode ser aquilo que você desejar: uma String, vetor, objeto...

# Curiosidades

* Vários Parâmetros em uma função (parâmetros variáveis): recebe um ou mais <lançamentos> que você passe como parâmetro, no final é concatenado todos os parâmetros em um array.



* O operador new em uma função JS gera um objeto.
* O JS não existe erros de NullPointerException, erro de falta da passada de parâmetros em uma função, ele apenas retorna undefined.

## Declarativo x Imperativo

A abordagem imperativa está preocupada em **como** realizar uma determinada tarefa. Informar o passo a passo completo.

Enquanto que a abordagem declarativo está preocupada em **o que** deve ser feito. Não se preocupa em como deve ser feito, passo a passo, apenas informar o que deve ser feito.

Abordagens declarativas são mais interessantes pois elas permitem uma reusabilidade do código, separar as funcionalidades por funções pequenas.

# Operações Interessantes

## Convertendo JSON

const carrinho = [                              //JSON

    '{ "nome": "Borracha", "preco": 3.45 }',

    '{ "nome": "Caderno", "preco": 13.90 }',

    '{ "nome": "Kit de Lapis", "preco": 41.22 }',

    '{ "nome": "Caneta" , "preco": 7.50 }'

]

const paraObj = json => JSON.parse(json)

const somentePreco = produto => produto.nome

const resultado = carrinho.map(paraObj).map(somentePreco)

console.log(resultado)



## Copiar vários valores de uma vez só

Quando você deseja colocar um mesmo valor de uma vez só para várias linhas, use o comando: ‘Ctrl + D’ – para pegar a quantidade de linhas desejadas, caminhe utilizando as setas para atingir a posição desejada, e digite o valor desejado para todas as linhas.

## Função Eval

A função eval avalia a expressão passada e retorna o seu resultado: funciona para todas as operações

eval(´${values[0]} ${operation} ${values[1]} ´) = > 23 + 2 => 25

## Função Fill

Preenche os valores de um array com um determinado valor passado:

Array(15).fill(0) -> preenche todos os índices do array com 0

Array(15).fill(0).map( (\_, i) => i + 1 ) -> preenche todos os índices do array de forma crescente

Possui o underline para ignorar o elemento, i pega o índice.

## Utilizando bibliotecas/fontes externas

Para utilizar bibliotecas e fontes siga os passos:

1. Crie um arquivo na raiz do projeto com o nome: react-native.config.js
2. Adicione:

module.exports = {

project: {

ios: {},

android: {},

},

assets: ['./assets/fonts']

};

1. Rode o comando na pasta do projeto: react-native link
2. Logo em seguida rode: react-native run-android ou react-native run-ios

O link serve para linkar bibliotecas de terceiros e fontes

## Renderização do componente

Função utilizada quando o componente é montado:

componentDidMount = () => {

}

# Componentes/Bibliotecas

## FlatList

<FlatList data={this.state.tasks}

                        keyExtractor={item => `${item.id}`}

                        renderItem={({ item }) => <Task {...item} />} />

//keyextractor: sempre em um array vc precisa definir qual sera a chave q será gerada

    //propriedade para renderizar o item

## TouchableWithoutFeedback

Área clicável sem nenhum retorno visual.

## Datas

Dependência para trabalhar com datas

Rode o comando: npm i -s moment

<https://momentjs.com/>

### Diferenciando datas no IOS e Android

//manipula as mudanças de data

    handleDateAndroidChanged = () => {

        //passa a data

        //a função retorna uma promise

        //o then recebe o evento que aconteceu a partir da seleção da data

        //só entra no if se o usuário não mudou a data (ignorou e deixou a marcada)

        //o o toDate transforma uma data do moment para uma data pura do JS para setar no estado

        DatePickerAndroid.open({

            date: this.state.date

        }).then(e => {

            if (e.action !== DatePickerAndroid.dismissedAction){

                const momentDate = moment(this.state.date)

                momentDate.date(e.day)

                momentDate.month(e.month)

                momentDate.year(e.year)

                this.setState({ date: momentDate.toDate() })

            }

        })

    }

let datePicker = null

        if (Platform.OS === 'ios') {

            datePicker = <DatePickerIOS mode='date' date={this.state.date}

            onDateChange={date => this.setState({ date })} />

        } else {

            datePicker = (

                <TouchableOpacity onPress={this.handleDateAndroidChanged}>

                    <Text style={styles.date}>

                        {moment(this.state.date).format('ddd, D [de] MMMM [de] YYYY')}

                    </Text>

                </TouchableOpacity>

            )

        }

Chamando no return:

{datePicker}

DatePickerAndroid has been merged with DatePickerIOS and TimePickerAndroid into a single component called [DateTimePicker](https://github.com/react-native-community/react-native-datetimepicker" \l "react-native-datetimepicker) and will be removed in a future release.

## Ícones

Dependência para utilizar ícones. Site: oblador.github.io/react-native-vector-icons/

Rode o comando na pasta do projeto: npm i -s react-native-vector-icons

Linke a biblioteca: react-native link

Faça o deploy novamente: react-native run-android ou react-native run-ios

## Botão de Ação

Botão de ação vermelho que fica no canto inferior direito.

Rode o comando na pasta raiz do projeto: npm i -s react-native-action-button

## Swipe

Ação de arrastar um elemento lateralmente para realizar alguma ação, como por exemplo, excluí-lo.

Rode o comando: npm i -s react-native-swipeable

Exemplo de uso:

//define o que vai aparecer do lado esquerdo e direito do swipe

    //conteúdo do lado esquerdo

    const leftContent = (

        <View style={styles.exclude}>

            <Icon name='trash' size={20} color={'#FFF'} />

            <Text style={styles.excludeText}>Excluir</Text>

        </View>

    )

    //vc pode colocar um array de opções de ações

    const rightContent = [

        <TouchableOpacity style={[styles.exclude, { justifyContent: 'flex-start', paddingLeft: 20  }]}

            onPress={() => props.onDelete(props.id)}>

            <Icon name='trash' size={20} color='#FFF' />

        </TouchableOpacity>,

    ]

//a task inteira é envolvida para habilitar o swipe em qualquer lugar da task

    //leftActionActivationDistance: quanto de distância irá percorrer até disparar o evento

<Swipeable leftActionActivationDistance={200}

            onLeftActionActivate={() => props.onDelete(props.id)}

            leftContent={leftContent} rightButtons={rightContent}>

//ÁREA DO SWIPE

</Swipeable>

# Documentação

## JS

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>

# Execução em App

Instale o expo.

Rode o comando: create-react-native-app nome\_app.

Na pasta do projeto rode: expo start --send-to <phone number ex: +553198573-8347>

Leia o QRCode q abrir no navegador, se não der certo mude para Tunnel.

# Projeto

## Organização das Pastas

* Index.js : porta de entrada da aplicação, que carrega apenas 1 componente: App.js.
* Android: projeto do Android Studio
* Ios: projeto do XCode.
* Crie a pasta src para separar bem o que é efetivamente o seu código.
  + Crie uma sub pasta : componentes, para guardar todos os componentes que for criar.

## Recriando o arquivo APP

Importações: sistemas de módulos.

* Import React: importação da biblioteca do React (muito importante)
* Component: está entre chaves pois está em um operador de desestruturação. O componente faz parte da biblioteca React.

import React, { Component } from 'react'

* View, Text: dois componentes do react-native. São os componentes mais básicos.

import { View, Text } from 'react-native'

* Export default: precisa exportar o componente, pois ele está sendo usando (importado) em index.js. Existem dois tipos de componentes: baseados em classe e em função. Não é preciso criar o componente necessariamente por classe, pode utilizar outras estratégias - função. Classe:
* Render(): função responsável por renderizar a tela que o componente se propõe a renderizar. Retorna um trecho de código em JSX (parece uma linguagem de marcação, tags. No react native ele é convertido para os componentes nativos de acordo com a plataforma).
* View: como se fosse uma div. É um container de outros componentes.

export default class App extends Component{ //a classe herda de Component

    render(){

        return (

            <View>

                <Text>

                    App!

                </Text>

            </View>

        )

    }

}

* Utilizando função (componente funcional):

export default function(){

    return(

        <View style={styles.container}>

            <Text style={styles.f20}>

                Hello World!

            </Text>

        </View>

    )

}

* Estilos: fortemente baseados no flexbox (difere na forma de escrever). Importa o stylesheet do react-native. Passa um objeto (formado por chave e valor – chave: nome do estilo que você quer colocar, valor: objeto com várias propriedades setadas) como parâmetro que será todos os estilos.

const styles = StyleSheet.create({

    container: {

        flex: 1, //flexbox do container

        justifyContent: 'center', //justifica o conteúdo para ser central na vertical

        alignItems: 'center' //justifica o conteúdo para ser central na horizontal

    },

    f20: {

        fontSize: 40 //coloca o tamanho da fonte para 40

    }

})

## Recriando de forma funcional

* Criação do componente na pasta componentes dentro de src:

import React from 'react' //sempre que possuir código JSX o React deve estar presente no componente

import { Text } from 'react-native'

import { tsPropertySignature } from '@babel/types'

//construção do primeiro componente que será baseado em função - componente funcional

//é mais simples e possui menos coisas: não precisa extender de componente

//em que o retorno da função é um trecho de JSX (componentes que serão renderizados na tela da sua aplicação)

export default function(props) { //não recebe nenhum parâmetro

    return <Text>{props.texto}</Text>

}

Para mudar tem que mudar no componente, passar um parâmetro (convencional é o nome props, de propriedade), acessar as propriedades de um componente.

Como o nome passado é texto, dentro de props terá um atributo chamado texto e acessa-o por esse nome.

Coloca em chaves para interpolar a variável (interpretar o valor e substituí-lo).

* + Usando arrow function:

export default props => <Text>Arrow: {props.texto}</Text>

//para retornar mais de um elemento: coloque em um container maior (view), e retorne a view

//ou passe um array, em que cada posição é um elemento diferente

* App.js:

import React, { Component } from 'react'

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

import Simples from './components/Simples' //pode importar seus próprios componentes

export default class App extends Component{ //a classe herda de Component

    render(){

        return (

            <View style={styles.container}>

                <Simples texto='Flexível!'/>

<Simples /> -- forma amarrada

            </View> //pode ter tanto a tag de abertura e fechamento para simples, mas pode fechar nela mesma

        )

    }

Você pode passar o texto que será impresso dentro do componente, para não ficar amarrado.

Você passa uma propriedade para o componente. Passa a propriedade chamada texto (nome que desejar) e o seu valor (‘Flexível’).

Para mudar tem que mudar no componente, passar um parâmetro (convencional é o nome props)

}

## Criando estilo

import { StyleSheet } from 'react-native'

export default StyleSheet.create({

    ex: { //usado para os exercicios

        paddingHorizontal: 15, //eixo horizontal padding de 15 em cada lado

        marginVertical: 5, //margem para eles nao ficarem grudados no eixo vertical

        borderRadius: 10, //raio da borda

        borderWidth: 2, //largura da borda

        borderColor: 'gray', //cor da borda de cinza

        fontSize: 24, //tamanho da fonte

        fontWeight: 'bold' //peso da fonte

    }

})

import Padrao from '../style/Padrao' //.. para sair da pasta e entrar na outra

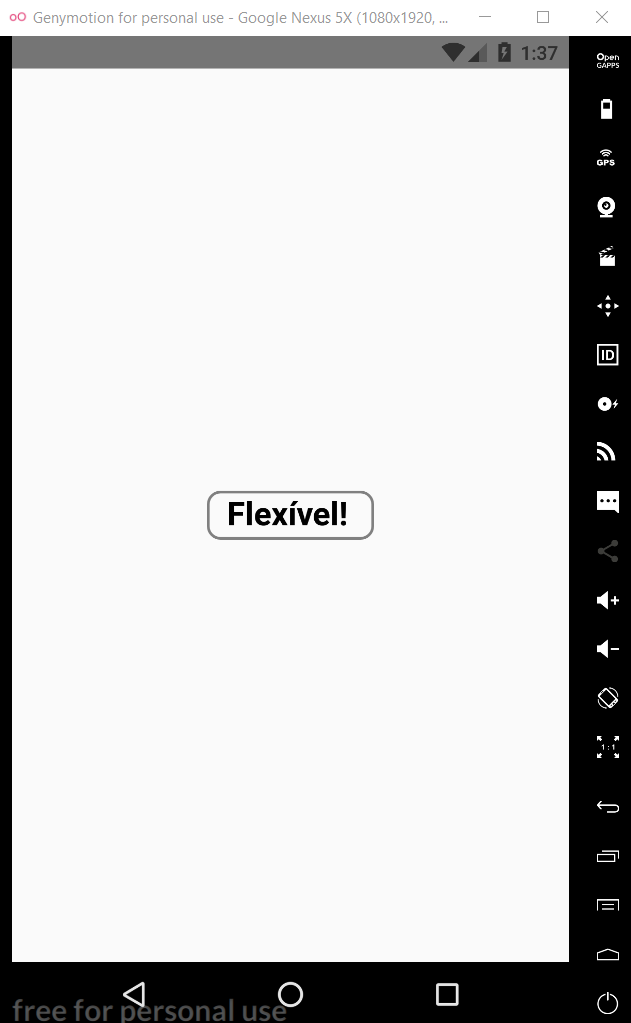
export default props => <Text style={[Padrao.ex]}>{props.texto}</Text>

//para retornar mais de um elemento: coloque em um container maior (view), e retorne a view

//ou passe um array, em que cada posição é um elemento diferente

//no estilo colocando entre colchetes pode-se adicionar mais de um estilo com vírgula para o mesmo elemento

Resultado:



## Renderização Condicional

* Possibilidade 1:
  + Arquivo ParImpar

import React from 'react'

import { View, Text } from 'react-native'

import Padrao from '../style/Padrao'

Mostra o text de acordo com o número informado.

Chaves para interpolar.

export default props =>

    <View>

        {

            props.num % 2 == 0

            ? <Text style={Padrao.ex}>Par</Text>

            : <Text style={Padrao.ex}>Ímpar</Text>

        }

    </View>

* + Arquivo App

import React, { Component } from 'react'

import { View, StyleSheet } from 'react-native'

import Simples from './components/Simples' //pode importar seus próprios componentes

import ParImpar from './components/ParImpar'

export default class App extends Component{ //a classe herda de Component

    render(){

        return (

            <View style={styles.container}>

                <Simples texto='Flexível!'/>

                <ParImpar num={10}/>

            </View> //pode ter tanto a tag de abertura e fechamento para simples, mas pode fechar nela mesma

        )

    }

}

* + Com função:

export default props =>

    <View>

        {parOuImpar(props.num)}

    </View>

function parOuImpar (num) {

    const r = num % 2 == 0 ? 'Par' : 'Ímpar'

    return <Text style={Padrao.ex}>{r}</Text>

}

## Componente com 2 exports

* No export se não tiver o default você é obrigado a colocar nome.
* Você pode exportar funções, objetos não necessariamente apenas componentes. Trabalhar com módulos é muito importante.
* O ideal é colocar um componente por arquivo.
* Quando você exporta por padrão você importa sem as chaves, quando você apenas exporta (deseja exportar outras coisas), você importa utilizando as chaves e separando por vírgula quando possui mais de uma coisa sendo importada.
* Apenas uma única coisa é exportada por padrão.

## Navegação

Estratégias de navegação

### React Navigation

Primeiro instala a dependência

* No arquivo package.json: adicione a dependência react-navigation:

"dependencies": {

    "react": "16.9.0",

    "react-native": "0.61.0",

    "react-navigation": "2.7.0"

  },

* Abra o terminal e rode: npm i – para instalar a dependência.
* Exemplo:

import React from 'react'

import { createDrawerNavigator } from 'react-navigation' //método

//import dos ex para o menu

import Simples from './components/Simples'

import ParImpar from './components/ParImpar'

import Inverter, { MegaSena } from './components/Multi'

//exporta o resultado da chamada deste método (importado)

//passagem de dois parametros

// 1 - lista dos menus e qual será o componente que será carregado quando o usuário clicar no menu (pode confg qual tela será aberta por padrão)

// 2 - define a largura do drawer, o menu que aparece a direita quando o usuário desliza o dedo

export default createDrawerNavigator({

    MegaSena: {

        //screen: qual é o componente que será carregado quando o usuário clicar no item do menu

        //utiliza arrow function para poder passar parâmetros, se n n daria

        screen: () => <MegaSena numeros = {8} />,

        //navigationOptions: muda o label do botão, o título, se não informar o título usa a chave de identificação passada: MegaSena

        navigationOptions: { title: 'Mega Sena' }

    },

    Inverter: {

        screen: () => <Inverter texto='React Nativo!' />

    },

    ParImpar: {

        screen: () => <ParImpar num={40}/>,

        navigationOptions: { title: 'Par & Ímpar' }

    },

    Simples: {

        screen: () => <Simples texto='Flexível!!!' />

    }

}, { drawerWidht: 300, drawerPosition: "left" })

Dúvida: fica aparecendo um warning do drawer position:

Setting DrawerLayoutAndroid drawerPosition using `DrawerLayoutAndroid.positions` is deprecated. Instead pass the string value "left" or "right"

**Não consegui resolver!**

## Componentes de Classe e Estado

Para suportar o estado (argumentos que variam) o componente tem que ser de classe.

O construtor de uma classe recebe o props.

As propriedades de props são somente para leitura, você não consegue manipular o valor, para isso, é necessário o estado.

Para modificar o valor de estado, utilize a função: setState.

state = {

        numero: this.props.numInicial

    }

    maisUm = () => {

        this.setState({ numero : this.state.numero + 1 })

    }

    limpar = () => {

        this.setState({ numero : this.props.numInicial })

    }

    //em componentes baseados em classe, temos q ter pelo menos um método: render() - renderiza o componente

    render() {

        return(

            <View>

                <Text style={Padrao.ex}>{this.state.numero}</Text>

                <TouchableHighlight

                onPress={this.maisUm} onLongPress={this.limpar}>

                    <Text>Incrementar/Zerar </Text>

                </TouchableHighlight>

            </View>

        )

    }

/a função setState é uma função assíncrona, dessa forma, se vc colocar o comando pra

        //chamar a função na próxima linha, ela poderá ser chamada antes de mudar o estado

        //assim, o certo é colocar chamando a função como segundo parâmetro no setState

        //dessa forma, ele muda o estado e, em seguida, chama a função

### TouchableHighlight

Área que será clicada. Duas formas diferentes de clicar: clique mais curto e clique mais longo.

Em funções, basta adicionar o this para acessar um função da própria classe.

* Clique curto: é customizável, ex: incrementa.
* Clique longo: é customizável, ex: apaga o conteúdo.

## Validação de propriedade

import React from 'react'

import PropTypes from 'prop-types' //utiliza para fazer as validações

import { Text } from 'react-native'

const ValidarProps = props =>

    <Text style={{ fontSize: 35 }}>

        {props.label}

        {props.ano + 2000}

    </Text>

//atribuição dos valores default para as propriedades caso não seja passado nenhum valor

ValidarProps.defaultProps = {

    label: 'Ano: '

}

//definir que a propriedade ano precisa ser numérico (number) e é um valor requerido

 ValidarProps.propTypes = {

    ano: PropTypes.number.isRequired

 }

 export default ValidarProps

## Componentes Controlados/Não Controlados e Eventos

* Componente Controlado: padrão, tipo de estratégia utilizado na maior parte do tempo. O valor do componente reflete exatamente a mesma coisa que está no estado do componente – o estado muda -> automaticamente o componente visual muda. Inicialização com ‘’, precisa ter uma função de tratamento de evento atrelada - só consegue modificar se o estado for modificado.
* Componente Não Controlado: não há o sincronismo entre o estado e o componente visual. Inicialização com null, não precisa ter uma função atrelada para tratamento do evento.

## Comunicação entre componentes

A sua aplicação gera uma árvore de componentes. Esses componentes precisam comunicar entre si e temos os tipos de comunicação:

* Direta: props. Pai -> filho

import React from 'react'

import { View, Text } from 'react-native'

//cricação do objeto fonte que recebe um objeto que vai ter style

//forma de passar propriedades para um determinado componente: operador spread

const fonte = { style: { fontSize: 30 } }

function filhosComProps(props){

    return React.Children.map(props.children, c => React.cloneElement(c, {...props, ...c.props}))

}

//três níveis de comunicação: avô - pai - filho

export const Filho = props =>

    <View>

        <Text {...fonte}>Filho: {props.nome} {props.sobrenome}</Text>

    </View>

//...fonte: recebe todas as propriedades do objeto. Utiliza o operador spread (espalha um objeto) - todos os atributos do objeto

//será passado como parâmetro/propriedade

//props.children: consegue acessar todos os filhos

//react.cloneElement(): 1° param: qual elemento vc deseja clonar, 2° param: quais propriedades vc deseja acrescentar no elemento. Funciona apenas para 1 filho

//, mais de 1 componente da erro

//React.Children.map(props.children): percorre o array de filhos tranformando-o em outro array de filhos só que adiciona com as propriedades do pai

//

export const Pai = props =>

    <View>

        <Text {...fonte}>Pai: {props.nome} {props.sobrenome}</Text>

        {/\*props.children\*/}

        {/\*React.cloneElement(props.children, {...props, ...props.children.props})\*/}

        {filhosComProps(props)}

    </View>

//sobrenome: utiliza o mesmo sobrenome que recebeu na propriedade para passar para o pai

//pai {...props}: pega todas as propriedades do pai e sobrescreve logo em seguida

export const Avo = props =>

    <View>

        <Text {...fonte}>Avô: {props.nome} {props.sobrenome}</Text>

        <Pai nome='André' sobrenome={props.sobrenome}>

            <Filho nome='Ana' />

            <Filho nome='Gui' />

            <Filho nome='Davi' />

        </Pai>

        <Pai {...props} nome='Pedro'>

            <Filho nome='Rebeca' />

            <Filho nome='Renato' />

        </Pai>

    </View>

export default Avo

* Indireta via callback: quando você quer receber algo do filho: passa uma função e essa função é chamada sempre que acontecer algo e o componente pai é notificado de forma indireta.

import React, { Component } from 'react'

import { View, Text, TextInput } from 'react-native'

import Padrao from '../style/Padrao'

import { thisExpression } from '@babel/types'

//componente que possui o campo de texto - componente filho

export const Entrada = props =>

    <View>

        <TextInput value={props.texto} style={Padrao.input} onChangeText={props.chamarQuandoMudar} />

    </View>

export default class TextoSincronizado extends Component {

    state = {

        texto: ''

    }

    alterarTexto = texto => {

        this.setState({ texto })

    }

    //O textInput não está dentro do componente, está fora

    render() {

        return (

            <View>

                <Text style={Padrao.fonte40}>{this.state.texto}</Text>

                <Entrada texto={this.state.texto} chamarQuandoMudar={this.alterarTexto} />

            </View>

        )

    }

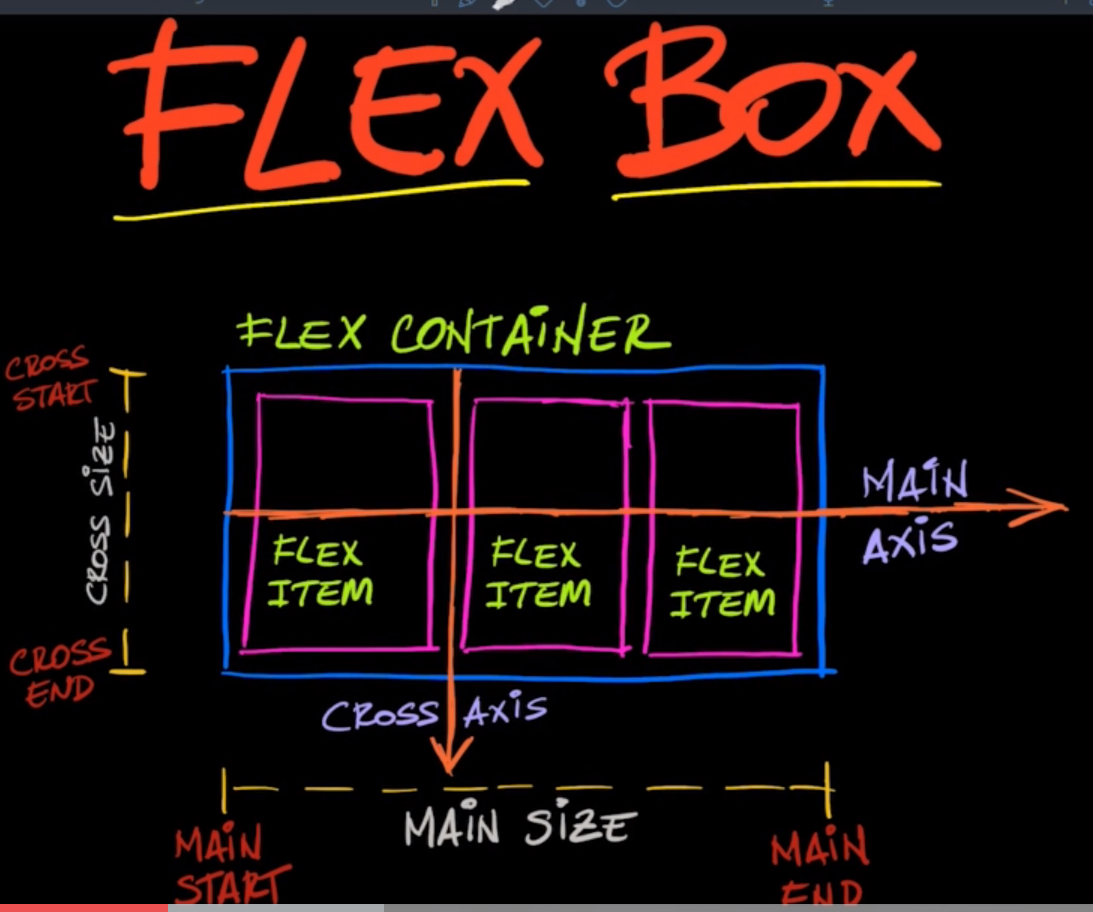
}

## Flexbox

Basta marcar o elemento com displayFlex.

Itens: flex item (tudo que você colocar dentro do flex container). Pode ter flex container dentro do flex item.

Main assix (eixo principal) e cross axis (eixo que corta): você pode definir para ser tanto o eixo vertical quanto o horizontal. Ambos possuem início, fim e tamanho.



### Exemplo:

import React from 'react'

import { View, StyleSheet } from 'react-native'

const styles = StyleSheet.create({

    container: {

        height: '100%', //tamanho da tela com 100%

    },

    //três áreas da tela:

    norte: {

        flex: 1, //crescimento da área, adapta o crescimento dos elementos de acordo com o tamanho restante da tela que não está sendo utilizado

        backgroundColor: '#bdf9ed',

        alignItems: 'center',

        justifyContent: 'center'

    },

    centro: {

        flex: 2, //cresce com maior pioridade do que o norte

        flexDirection: 'row', //muda a diração para ser linha

        backgroundColor: '#f2f9bd',

        alignItems: 'center',

        //justifyContent: 'center',

        justifyContent: 'space-around'

    },

    sul: {

        flex: 1,

        backgroundColor: '#bdf9c4',

        alignItems: 'center',

        justifyContent: 'center'

    },

    circulo: {

        width: 100, //largura

        height: 100, //altura

        backgroundColor: '#f47f61',

        borderRadius: 50, //raio da borda, circulo perfeito

    }

})

const Circulo = props => <View style={styles.circulo}></View>

//componente que possui várias views dentro da outra, para criar a região norte, centro e sul, e posicionando os círculos

export default props => {

    return (

        <View style={styles.container}>

            <View style={styles.norte}>

                <Circulo />

            </View>

            <View style={styles.centro}>

                <Circulo />

                <Circulo />

                <Circulo />

            </View>

            <View style={styles.sul}>

                <Circulo />

            </View>

        </View>

    )

}

# Diferenciações entre Android e IOS

Verificar qual plataforma que é:

import React from 'react'

import { Button, Alert, ToastAndroid, Platform } from 'react-native'

export default props => {

    const notificar = msg => {

        if(Platform.OS === 'android'){ // pode testar para ios também

            ToastAndroid.show(msg, ToastAndroid.LONG)

        }else{

            Alert.alert('Informação', msg)

        }

    }

    return (

        <Button title='Plataforma?'

            onPress={() => notificar('Parabéns!')}/>

    )

}





# Backend

O node será utilizado para construir a API, em que, o express será utilizado p

## Módulo

Cada arquivo é um módulo no Node, vc separa por módulos pra manter o encapsulamento entre todos os arquivos. Dentro do módulo, tudo q vc faz fica encapsulado e para acessar em outro arquivo vc precisa exportar.

## Instalação de Bibliotecas

Crie uma nova pasta duplicando o nome-backend

Mkdir tasks-backend

Crie o arquivo package.json: npm init -y (y: responde as perguntas no padrão)

- Instalação das dependências do projeto

Abre o terminal no vscode

### Web, global, compartilhamento de mensagens

* Express: Framework Web q ajuda a construir os web services:
* Body-Parser: middleware que interpreta o conteúdo de um body de uma requisição (parser do body da requisição e interpretando oq tem lá dentro (lá dentro possui um valor textual, e assim, converte para, por exemplo, JSON))
* Cors: deixar a api acessível de outras origens
* Consign: biblioteca q ajuda a compartilhar os dados entre os vários módulos da aplicação
* npm i -s express body-parser cors consign (o s salva no package.json)

Segurança

Frameworks de autenticação (facebook, google)

* npm i -s bcrypt-nodejs jwt-simple passport passport-jwt

Acesso ao banco de dados: pg (postgres)

* npm i -s knex pg

### Moment:

* npm i -s moment

Inicia a aplicação: starta a aplicação, pega as modificações e atualiza, se der algum erro não deixa a aplicação subir

### Nodemon (Launcher):

Voltada para quando a aplicação está em desenvolvimento. É um disparador. Vantagem: monitora os arquivos, assim, qualquer mudança feita ele automaticamente reinicia a aplicação e lê a versão mais nova dos arquivos.

Se não tiver o nodemon, sempre q vc realizar uma modificação em um arquivo vc deverá reiniciar manualmente a aplicação.

* npm i -s –save-dev nodemon
* "scripts": {
* "start": "nodemon"
* },

Coloque o start, assim, qnd rodar: npm start, irá chamar o nodemon, q irá startar a aplicação chamando o index.js, pois, ele está configurado no “main”: “index.js”

(pode colocar vários scripts de acordo com o desejado: ex: production, test. Quando for chamar, basta colocar npm run nome)



~: Atualiza apenas no fix

Major:minor:fix

(biblioteca: releases: bugs)

^: atualiza o minor e fix

Qnd vc instala usando o -E e colocando o número da versão: [express@4.16.3](mailto:express@4.16.3) ele instala exatamente a versão e não coloca ^ ou ~

O .gitignore garente que o que os nomes que você colocar nesse arquivo não serão persistidos no Git.

### PM2 (Launcher)

Gerenciador de processos do node: PM2 – voltada para quando estiver em produção (controla microserviços, monitoramento de memória e processador, qnts vezes a aplicação caiu e teve q reiniciá-la)

<https://pm2.keymetrics.io/>

Instalação: npm i –save -g pm2

Insira um novo script q será utilizado na produção:

"production": "pm2 start index.js --name appx-backend"

Comando para:

Name: nome que vai aparecer quando for fazer o monitoriamento

* monitorar: pm2 monit. A informação do restart é importante, pois, com ele vc verifica a quantidade de vezes que a aplicação deu erro e teve q reiniciar.
* Status: pm2 status
* Detalhes da aplicação X: pm2 show 0 (id)
* Reestartar a aplicação X: pm2 restart 0 (id) OU pm2 restart appx-backend (nome)
* Parar a aplicação X: pm2 stop 0
* Matar a instância do pm2: pm2 kill

### Instalar uma biblioteca apenas no modo de desenvolvimento, não vai para o ambiente produtivo

Comando: npm –save-dev nome\_biblioteca

## Express

### Importação

//versão mais simples de uma 'api'

//require: função que importa alguma coisa - módulo express

//está como caminho absoluto pq instalou na pasta node\_modules

const express = require('express')

### Instânciação

//instancia o express e retorna a partir da função construtora express()

//retorna o que seria a aplicação

//aplicação: onde vc cria as rotas, a resposta das requisições

const app = express()

### Definição da ocupação da porta

//carrega o express

//porta: processo no seu computador

//é possível ter apenas 1 processo em cada porta

//coloca uma callback para mostrar uma mensagem quando der certo a conexão

app.listen(3000, () => {

    console.log('Backend executando...')

})

### Primeira Requisição

//função use q recebe uma callback com requisição e resposta

//o use é aplicado para qualquer tipo de requisição (GET, POST..)

//protocolo HTTP: requisição e resposta

//assim quando a função é chamada, vc deseja enviar uma resposta

//na resposta vc pode colocar tags HTML

//define a url específica /opa - se n definir, todas as requisições cairão na básica

app.use('/opa', (req, res) => {

    res.send('Estou <b>bem</b>!')

})

//atender a todos: all

app.all('/opa', (req, res) => {

    res.send('Estou <b>bem</b>!')

})

## Tipos de Requisições

Requisição: vc envia algo, processa essa requisição pesquisando no banco de dados, tratando as regras de negócio, e retorna o processamento em formato JSON.

A ordem das funções influencia na resposta.

### GET

Os parâmetros são passados pela URL

//amarrar o tipo de requisição para GET ou POST

app.get('/opa', (req, res) => {

    res.send('Estou <b>bem</b>!')

})

### POST

Os parâmetros são passados no corpo da requisição

app.post('/opa', (req, res) => {

    res.send('Estou <b>bem</b>!')

})\*/

### OPTION

É utilizado para verificar se vc tem acesso a uma API e quais são os métodos disponíveis no endpoint (URL associada a API)(pre flight request – pré voo).

## Tipos de Respostas

Tipos: XML, textual, separado por vírgula, HTML, binário (streaming de dados).

### HTML

app.use('/opa', (req, res) => {

    res.send('Estou <b>bem</b>!')

})

### JSON

Formato mais utilizado

//coloca um objeto JS q será convertido para JSON

app.use('/opa', (req, res) => {

    res.json({

        name: 'iPad 32GB',

        price: 1899.00,

        discount: 0.12

    })

})

Array de objetos:

//coloca um array JS q será convertido para JSON

app.use('/opa', (req, res) => {

    res.json([

        { id: 7, name: 'Ana', position: 1 },

        { id: 34, name: 'Bia', position: 2 },

        { id: 73, name: 'Carlos', position: 3 }

    ])

})

Informações da Requisição

Retorna um objeto com informações da requisição

* Data: conteúdo da resposta
* Count: quantos registros possui na base considerando a consulta
* Skip: paginação (quantas páginas você já pulou, se for a primeira página fica como 0),
* Limit: quantos elementos possui na página
* Status: situação da requisição

//retorna informações adicionais da consulta

app.use('/opa', (req, res) => {

    res.json({

        data: [

            { id: 7, name: 'Ana', position: 1 },

            { id: 34, name: 'Bia', position: 2 },

            { id: 73, name: 'Carlos', position: 3 }

        ],

        count: 30,

        skip: 0,

        limit: 3,

        status: 200

    })

})

Ex: limitou a consulta em 3, possui 10 páginas, está no primeiro registro, e pegou apenas os 3 primeiros.

## Middleware

Função middleware é uma função que recebe req, res, next

O express trabalha com o conceito de CHAIN OF RESPONSABILITY (cadeia de responsabilidade)

Função next: chama a próxima função na cadeia

Importante qnd trabalha com outras bibliotecas, ent vc tem um processamento inicial, dps vai pra uma função sua, e dps vai pro processamento final.

//função middleware que recebe req e res

app.use((req, res, next) => {

    console.log('Antes...')

    next()

})

//retorna informações adicionais da consulta

app.get('/opa', (req, res, next) => {

    console.log('Durante...')

    res.json({

        data: [

            { id: 7, name: 'Ana', position: 1 },

            { id: 34, name: 'Bia', position: 2 },

            { id: 73, name: 'Carlos', position: 3 }

        ],

        count: 30,

        skip: 0,

        limit: 3,

        status: 200

    })

    next()

})

app.use((req, res) => {

    console.log('Depois...')

})

### Simulando Middleware de terceiros

//apenas ignora os parâmetro de requisição e resposta

//pois o JS é bastante flexível, mas qnd ela é chamada, é tratada como função middleware

//para usar outro atributos como parâmetro, por ex o nome, vc faz a chamada para uma função e o seu retorno é uma função middleware

function saudacao(nome) {

    return function (req, res, next) {

        //executa sempre quando uma nova requisição é chamada

        console.log(`Seja bem vindo ${nome}.`)

        next()

    }

}

module.exports = saudacao

index.js

const saudacao = require('./saudacaoMid')

//entende q saudação é uma função middleware

//passa uma função e o retorno dela é a função middleware

app.use(saudacao('Guilherme'))

### Ler dados da requisição

//passa o parâmetro através da query

//mais típico de ser utilizado

//essa função tem q estar antes de cliente/:id pq ela é mais específica e a outra é mais genérica

//com isso, se n mudar a ordem, a função mias genérica será atendida e será executada

app.get('/clientes/relatorio', (req, res) => {

    res.send(`Cliente relatório: completo = ${req.query.completo} ano = ${req.query.ano}`)

})

//parte da URL q pode ser mudada

//os : significa que o id é o parâmetro. Acessa através de params da req

app.get('/clientes/:id/:idade', (req, res) => {

    res.send(`Cliente ${req.params.id} selecionado com ${req.params.idade} anos`)

})

Chamada1: localhost:3000/clientes/relatorio?completo=true&ano=2019

Chamada2: localhost:3000/clientes/Carla Amaral/100

Passa a interrogação para informar que será iniciado uma query string na URL e o & para passar mais de um parâmetro

#### Ler dados do body

Forma mais difícil, mais usual usar o body-parser

//receber os parâmetros no body

app.post('/corpo', (req, res) => {

    let corpo = ''

    //quando tiver chegando dados a partir do corpo da requisição chama a função

    //parte: qual e a parte que esta chegando: String

    req.on('data', function(parte) {

        //vai concatenando pois pode chegar aos poucos

        corpo += parte

    })

    //qnd terminar a requisição

    req.on('end', function(){

        //manda de volta tudo q recebeu

        res.send(corpo)

    })

})

Form-urlencoded: formato padrão de qnd vc envia formulários no front end

#### Body-Parse

Cria uma propriedade dentro de request chamada body q possui o resultado do body interpretado por ele. Facilita o processo de pegar o body. Interpreta o body da requisição.

Instalação: npm i –save body-parser

const bodyParser = require('body-parser')

//chama uma função q existe em bodyParser e o resultado da chamada retorna uma função middleware

//parses do body

//o use é aplicado a todas as requisições

app.use(bodyParser.text())

app.use(bodyParser.json())

//extended: tipos de dados a mais q ele irá interpretar

app.use(bodyParser.urlencoded({extended: true}))

//receber os parâmetros no body

app.post('/corpo', (req, res) => {

    /\*let corpo = ''

    //quando tiver chegando dados a partir do corpo da requisição chama a função

    //parte: qual e a parte que esta chegando: String

    req.on('data', function(parte) {

        //vai concatenando pois pode chegar aos poucos

        corpo += parte

    })

Observe a diferença no código

O código previamente utilizado foi substituído por apenas 1 linha: res.send(req.body)

    //qnd terminar a requisição

    req.on('end', function(){

        //manda de volta tudo q recebeu

        res.send(corpo)

    })\*/

    //objeto q foi interpretado pelo body parser, da pra pegar os atributos por ele

    //res.send(req.body)

    //res.send(req.body.name)

    res.send(JSON.stringify(req.body))

})

## Comunicação entre módulos

Cria um módulo em que os dados/funções estão encapsulados/escondidos/visíveis apenas dentro do módulo, vc exporta oq vc quer e acessa em outro arquivo.

Crie uma pasta chamada api/usuário:

//função middleware

function salvar(req, res) {

    //momento em q acontece o acesso ao banco de dados e todo o processamento

    res.send('Usuario > salvar')

}

function obter(req, res){

    res.send('Usuario > obter')

}

//envia um objeto

module.exports = { salvar, obter }

index.js:

const usuarioApi = require('./api/usuario')

//acessa as funções do módulo

app.post('/usuario', usuarioApi.salvar)

app.get('/usuario', usuarioApi.obter)

### Passagem de parâmetros para um módulo

//simulando uma api

//retorna uma função que recebe parâmetros

//recebe a instância do express

module.exports = (app, text) => {

    function salvar(req, res) {

        res.send('Produto > salvar > ' + text)

    }

    function obter(req, res) {

        res.send('Produto > obter > ' + text)

    }

    //como recebe o app, pode registrar as chamadas da url dentro do módulo

    app.post('/produto', salvar)

    app.get('/produto', obter)

    //retornar as funções para utilizar em outro contexto

    return { salvar, obter }

}

Chamada:

//carrega o módulo e passa os parâmetros

require('./api/produto')(app, 'com param!')

//outra forma:

//const produtoApi = require('./api/produto')

//produtoApi(app, 'com param!')

## Banco de Dados

Postgres: