React Native

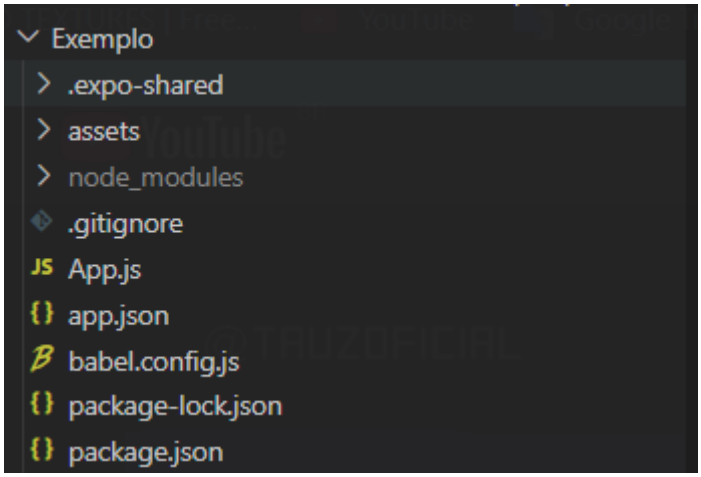
Entendendo React:

React Native é uma biblioteca do React utilizada no desenvolvimento mobile multiplataforma com foco em JavaScript.

Para criarmos um projeto vamos usar o Expo Go, basta baixarmos o aplicativo no smartphone, e executarmos o seguinte comando no terminal:

npx create-expo-app NomeDoProjeto

Teremos uma estrutura semelhante a isso:



A primeira pasta é **.expo**, é nela que estão contidos os arquivos de informação e configuração interna do Expo.

A segunda pasta é a **assets** aqui é onde estará os arquivos de mídia como imagens por exemplo.

A terceira pasta é a **node_modules** aqui é onde ficara as dependências do projeto.

O arquivo **.gitignore** é onde são informados os arquivos a serem ignorados no versionamento.

O **App.js** é o arquivo principal do projeto e onde ficara toda a visualização, o React Native é um sistema de Single Page Application (SPA), ou seja, um sistema de página única.

Em seguida o arquivo que teremos é o **babel.config.js**, esse arquivo é responsável pela configuração do transpilador babel, responsável por traduzir o JSX.

Para executarmos esse projeto basta usar o comando:

npm start

com isso irá aparecer um código Qr para abrirmos o projeto no aparelho telefone.

JSX:

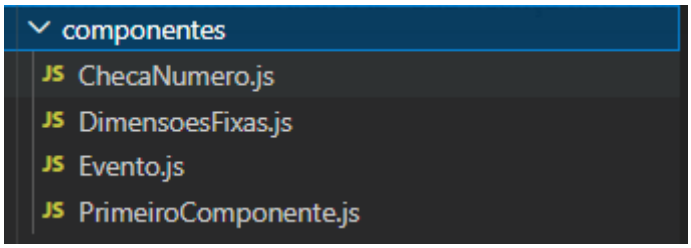
O React utiliza-se do conceito de JSX misturando o HTML, CSS e JavaScript no mesmo código:

```
export default function App() {
  return (
    <View style={styles.container}>
      <Text>Open up App.js to start working on your app!</Text>
      <StatusBar style="auto" />
    </View>
  );
}
```

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

Componentes:

Vamos criar uma pasta chamada componentes para guardarmos nossos componentes:



Existem duas formas de se criar um componente:

Componente Funcional:

```
import React from "react";
import { Text, View } from "react-native";
export default function(props){
    return(

        <Text >{props.nome}</Text>

    )
}
```

Esse tipo de componente utiliza de funções e não tem suporte a propriedade **estado**

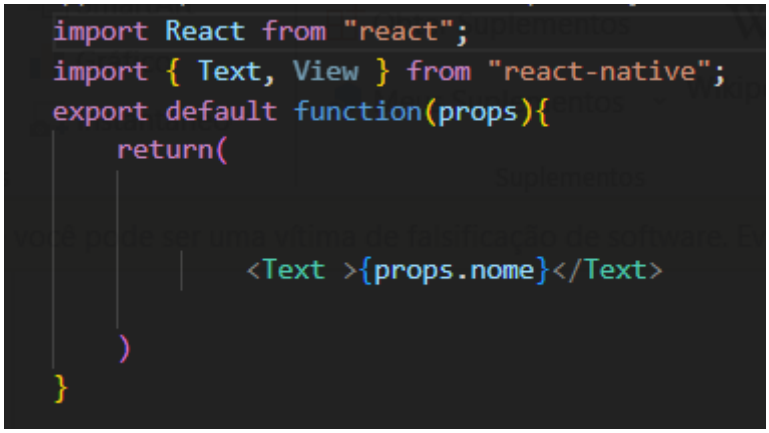
Componente de Classe:

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

class DimensoesFixas extends React.Component{
    render(){
        return(
            <View style={{width:'100%', height:'100%', flexDirection:'row'}}>
                <View style={{width:50, height:50, backgroundColor:'blue'}} />
                <View style={{width:100, height:100, backgroundColor:'red'}} />
                <View style={{width:150, height:150, backgroundColor:'black'}} />
            </View>
        );
    }
};

export default DimensoesFixas;
```

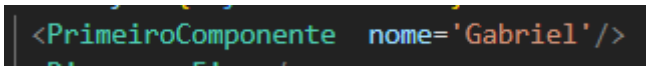
Esse tipo de componente é mais recomendado por utilizar a propriedade de **estado**:

Alterando propriedades do componente:

Podemos modificar propriedades do componente na aplicação:



```
import React from "react";
import { Text, View } from "react-native";
export default function(props){
    return(

            <Text >{props.nome}</Text>

    )
}
```



```
<PrimeiroComponente  nome='Gabriel'/>
```

Fazemos isso através do atributo **props**.

Estado->state:

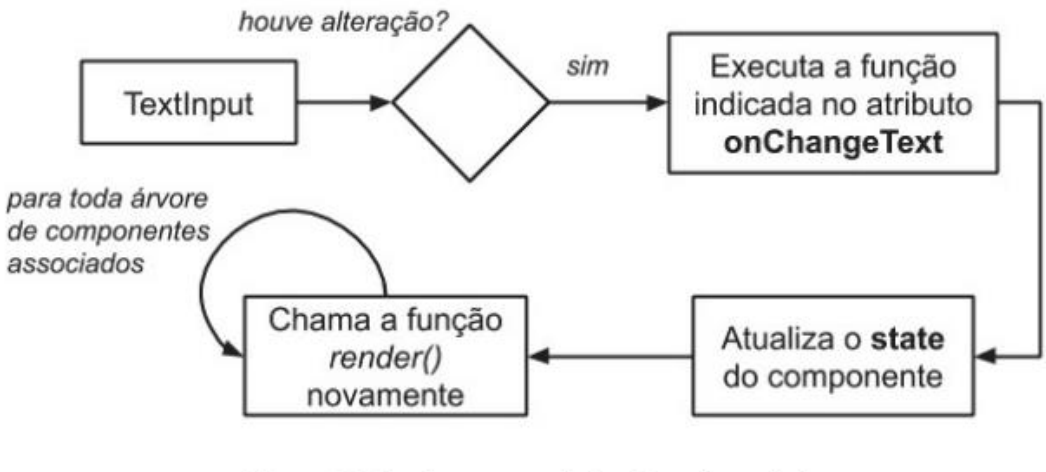
O estado é uma propriedade das classes que basicamente são um “print” do estado atual de um componente:

```
import React from "react";
import { View, Text, TextInput, StyleSheet } from "react-native";

class Evento extends React.Component{
    state = {
        input:'Baby do baby'
    }
    render(){
        return(
            <View>
                <Text>{this.state.input}</Text>
                <TextInput value={this.state.input} onChangeText={(input)=>this.setState({input})}></TextInput>


                </View>
        )
    }
}

export default Evento;
```

Dessa forma conseguimos criar os **Componentes controlados**, esse tipo de componente se baseia em componentes capazes de se auto monitorar, sem a necessidade de outro componente fazer isso.

O atributo **onChangeText** é um atributo fornecido pelo React que fornece uma forma de passar uma função toda vez que houver uma alteração no estado.

Ciclo de vida dos estados:



Renderização Condicional:

A renderização condicional se baseia na utilização de estruturas condicionais para executar ou não um componente:

```
export default props =>
<View>
    {validaParOuImpar(props.numero)}
</View>

function validaParOuImpar(numero){
    return numero %2 ==0
    ? <Text >O numero é par</Text>
    : <Text>O numero é impar</Text>
}
```

Podemos utilizar de **if** e **else**, ou utilizar dos operadores lógicos fornecidos pelo JavaScript:

condição ? expr1 : expr2

CSS:

Existem três formas de utilizarmos de CSS no JSX:

Primeira: Na declaração de atributos do JSX:

```
<View style={{width:'100%', height:'100%', flexDirection:'row'}}>
```

Segunda: Separado em uma variável da classe:

```
const styles = StyleSheet.create({
container:{
  flex:1,
  backgroundColor:'white',
  alignItems:'center',
  justifyContent:'center'
}

})
```

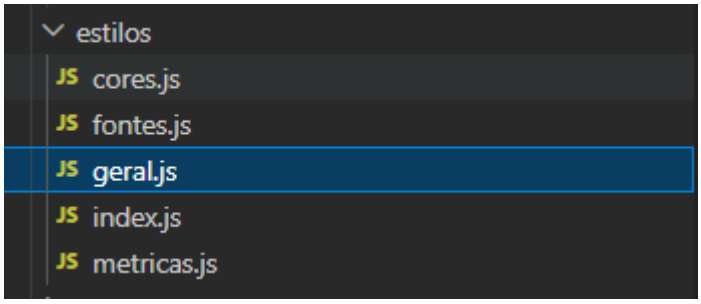
```
<View style={styles.container}>
```

Terceira: Através de um componente JS criado única e exclusivamente para conter regras do CSS:

```
import metricas from "./metricas";
import cores from "./cores";
import fontes from "./fontes";

const geral = {
    container:{
        flex:1,
        backgroundColor: cores.background,
    },
    section:{
        margin: metricas.doubleBaseMargin,
    },
    sectionTitle:{
        color: cores.text,
        fontWeight: 'bold',
        fontSize: fontes.regular,
        alignSelf: 'center',
        marginBottom: metricas.doubleBaseMargin,
    },
};

export default geral;
```

CSS separado em arquivos:

Pelas boas práticas é interessante utilizarmos de arquivos separados contendo nossas regras CSS, Criaremos a pasta **estilos** e vamos acrescentar uma separação muito comum:



Cores: Aqui são armazenadas cores recorrentes no projeto:

```
const cores ={
    header: '#333333',
    primario: '#069',
};
export default cores;
```

Fontes: Aqui são armazenados tamanhos padrões de fonte para a aplicação:

```
const fontes ={
    input:16,
    regular:14,
    medium:12,
    small:11,
    tiny:10,
};

export default fontes;
```

Metricas: Aqui são armazenadas as métricas de tamanhos da aplicação:

```
import { Dimensions, Platform } from "react-native";

const {width, height} = document.get('window');

const metricas = {
    smallMargin: 5,
    baseMargin: 10,
    doubleBaseMargin:20,
    screenWidth: width <height ? width : height,
    screenHeight: width <height ? height : width,
    tabBarHeight:54,
    navBarHeight: (Platform.OS === 'ios') ? 64:54,
    statusBarHeight: (Platform.OS === 'ios') ? 20:0,
    baseRadius:3,
};

export default metricas;
```

Geral: Aqui são armazenados alguns layouts padrões da aplicação:

```
import metricas from "./metricas";
import cores from "./cores";
import fontes from "./fontes";

const geral = {
    container:{
        flex:1,
        backgroundColor: cores.background,
    },
    section:{
        margin: metricas.doubleBaseMargin,
    },
    sectionTitle:{
        color: cores.text,
        fontWeight: 'bold',
        fontSize: fontes.regular,
        alignSelf: 'center',
        marginBottom: metricas.doubleBaseMargin,
    },
};

export default geral;
```

Index: Aqui basicamente temos apenas uma classe que armazenara todos os outros estilos:

```
import cores from './cores';
import fontes from './fontes';
import metricas from './metricas';
import geral from './geral';

export { cores, fontes, metricas, geral};
```

Flexbox:

Vale lembrar que a **width** e **height** no React Native não utiliza de pixels, mas sim de dimensões **untitless**, isso significa que na pratica eles representam pixels independentes da densidade, podemos dizer que seu tamanho continuara o mesmo independente da tela.

```
<View style={{width:'100%', height:'100%', flexDirection:'row'}}>
    <View style={{width:50, height:50, backgroundColor:'blue'}} />
    <View style={{width:100, height:100, backgroundColor:'red'}} />
    <View style={{width:150, height:150, backgroundColor:'black'}} />
</View>
```

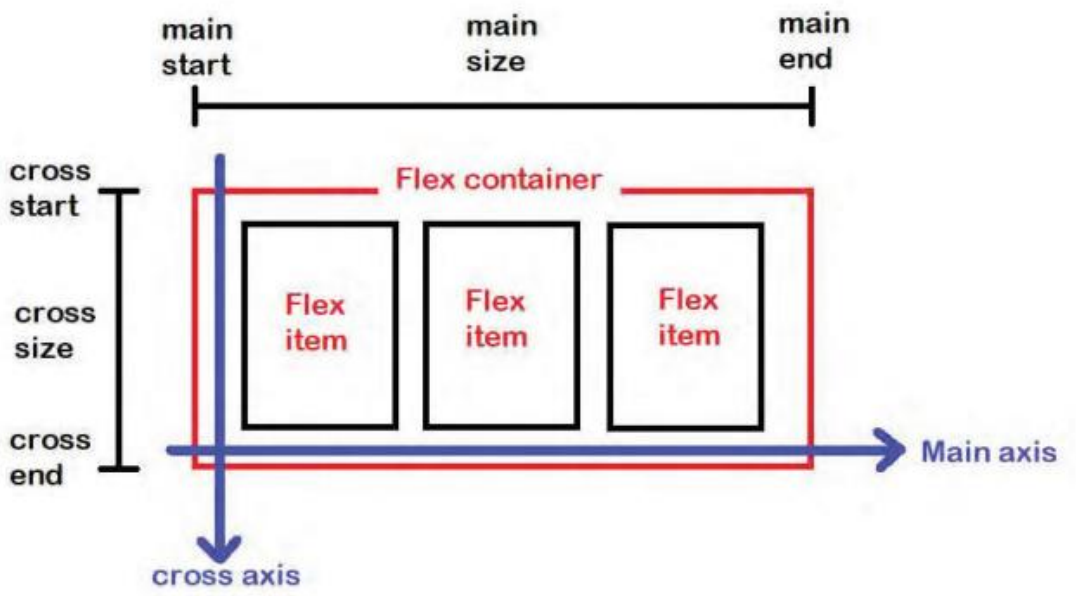
A ideia de utilizar de Flexbox é ter um layout responsivo sem ter que utilizar de frameworks complexos como Botstrap.

Flex-container e Flex-items:

Basicamente o container é onde serão armazenados os **Flex-items**, de forma que eles se auto organizem a depender das propriedades e das dimensões do container.

Para usarmos um **Flex-Container** basta anotarmos a propriedade **display: flex**, com isso estamos dizendo que o componente é **Flex-Container** e seus filhos **Flex-Items**.

Todo elemento dentro do **Flex-Container** é orientado através dos eixos: **a main-axis** (eixo principal) e **a cross-axis** (eixo transversal), por padrão o eixo principal é orientado horizontalmente e o transversal verticalmente:



Flex Direction:

Define a direção do eixo principal e secundário, por padrão o valor é **row** (linha), isso indica que o eixo principal ira se orientar horizontalmente e o eixo principal verticalmente, porem temos outros propriedades:

Row: valor padrão se orienta horizontalmente.

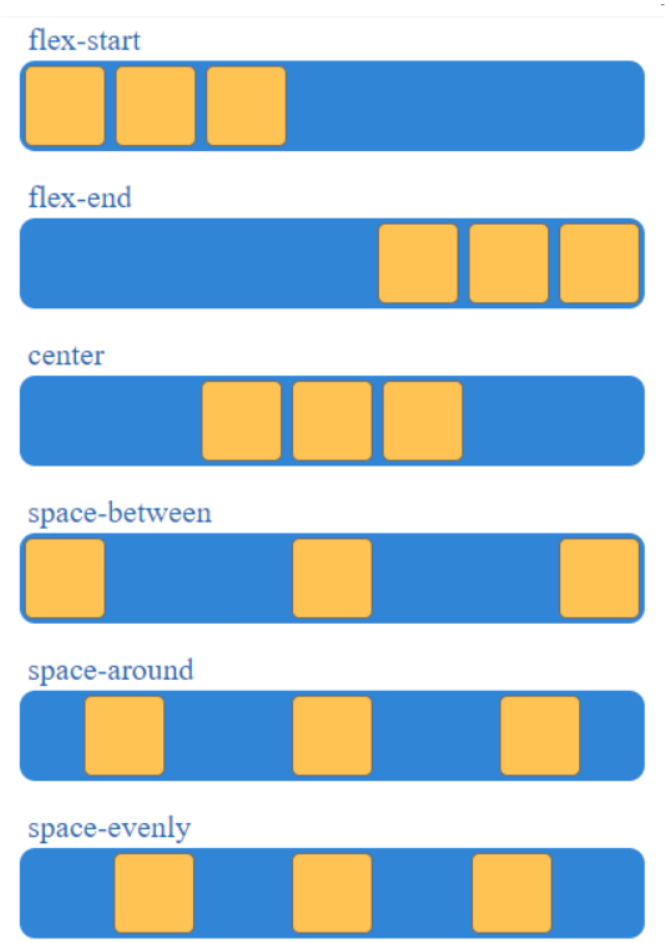
Row-reverse: Se orienta horizontalmente começando pelo final.

Column: Se orienta verticalmente.

Column-Reverse: Se oriente verticalmente começando do final.

Justify Content:

Determina a distribuição dos flex-items ao eixo principal pode ser feita das seguintes maneiras:



Flex-start: Alinha os itens no início do contêiner.

Flex-end: Alinha os itens no final do contêiner.

Center: Alinha os itens no centro do contêiner.

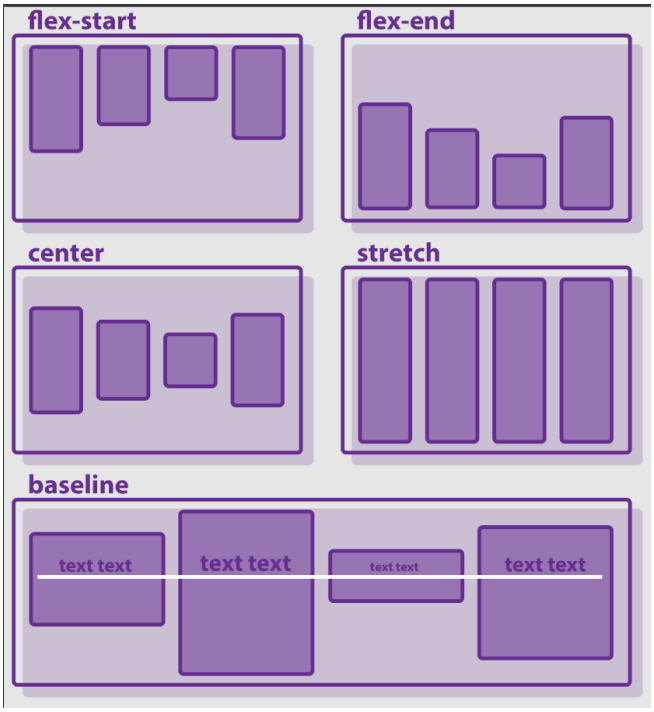
Space-between: Alinha os itens de forma que o espaço entre eles sejam o mesmo e o ultimo e o primeiro fiquem no final e no início respectivamente.

Space-around: Alinha os itens de forma que o espaço entre eles sejam o dobro do espaço entre o primeiro elemento e o inicio e o ultimo elemento e o final.

Space-evenly: Alinha os itens de forma que o espaço entre eles, o final e o inicio sejam do mesmo tamanho.

Align Items:

Determina o alinhamento dos **flex-items** ao longo do eixo secundário, possui os seguintes valores:



Flex-start: Alinha os itens no início do eixo secundário.

Flex-end: Alinha os itens no final do eixo secundário.

Center: Centraliza os itens em relação ao eixo secundário.

Stretch: valor padrão, aumenta o tamanho dos itens em relação ao eixo secundário de forma que na soma de todos os itens, todo o espaço disponível seja ocupado.

Baseline: Alinha os itens de acordo com a linha base da tipografia.

Flex-Wrap:

Define se os itens devem quebrar ou não a linha. Por padrão os itens não iram quebrar a linha, isso faz com que os **Flex-items** sejam compactados além do limite do conteúdo.

Pode assumir três valores:

Nowrap: valor padrão, não permite a quebra de linha.

Wrap: Quebra a linha assim que um dos flex-items não puder mais ser compactado.

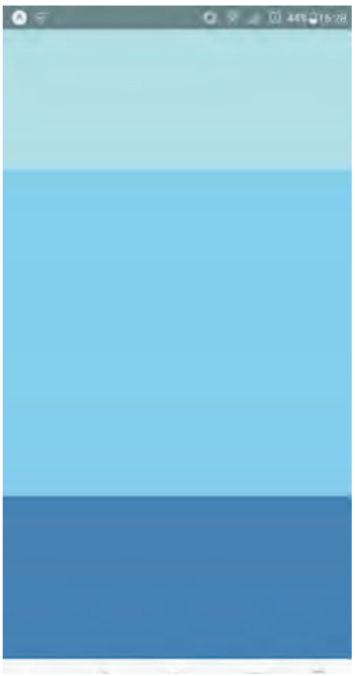
Wrap-reverse: Quebra a linha assim que um dos flex-items não puder mais ser compactado. Porém essa quebra é na direção contrária, ou seja, para a linha acima.

Flex-Grow:

Propriedade exclusiva dos **Flex-Items**, basicamente define a capacidade de um item tem de crescer.

Por padrão o seu valor é zero, por isso os **Flex-Items** irão ocupar um tamanho máximo relacionado ao conteúdo interno deles.

Se definirmos o tamanho 1, eles irão ter um tamanho igual e ocupar 100%, porém se aumentarmos o tamanho de algum deles para 2 por exemplo ele terá o dobro do tamanho dos outros com a propriedade 1.



Flex-shrink:

Propriedade exclusiva dos **Flex-Items**, basicamente define a capacidade de um item tem de diminuir.

Idêntico e inversamente a propriedade **Flex-grow**.

Flex-Basis:

Propriedade exclusiva dos **Flex-Items**, basicamente define o tamanho inicial do item antes da distribuição do espaço, por padrão vem com o valor **auto** significando que seu tamanho será proporcional ao conteúdo presente nele.

Requisições AJAX e APIs

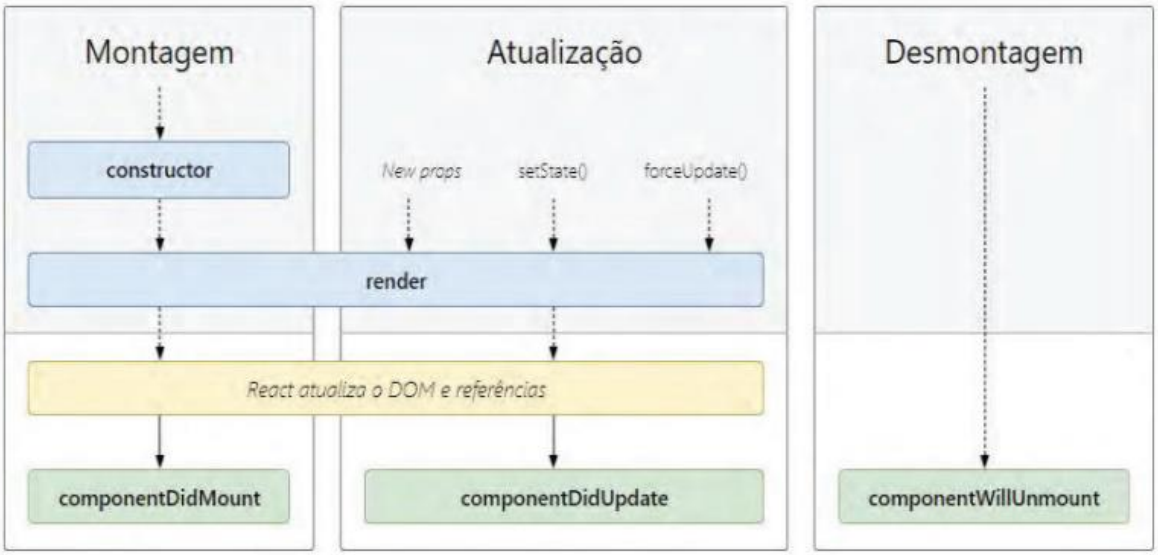
Ciclo de vida dos componentes:

Os métodos de ciclo de vida dos componentes do React são funcionalidades customizadas que são executadas durante as diferentes fases de um componente.

Um componente possui 4 fases no seu ciclo de vida:

1. Montagem (Mounting)
2. Atualização (Updating)
3. Desmontagem (Unmounting)
4. Erros (Error Handling)

Cada uma destas fases possui alguns métodos associados que podem ser sobrescritos nas classes dos componentes:



Os métodos em azul fazem parte do que chamamos de render, ela é pura e sem efeitos colaterais, pode ser pausada, abortada ou reiniciada pelo React.

As que estão abaixo fazem parte da “Fase commit”: Podem operar o dom, executar efeitos colaterais e agendar atualizações.

Montagem

Esses métodos são chamados na seguinte ordem quando uma instância de um componente está sendo criada e inserida no DOM:

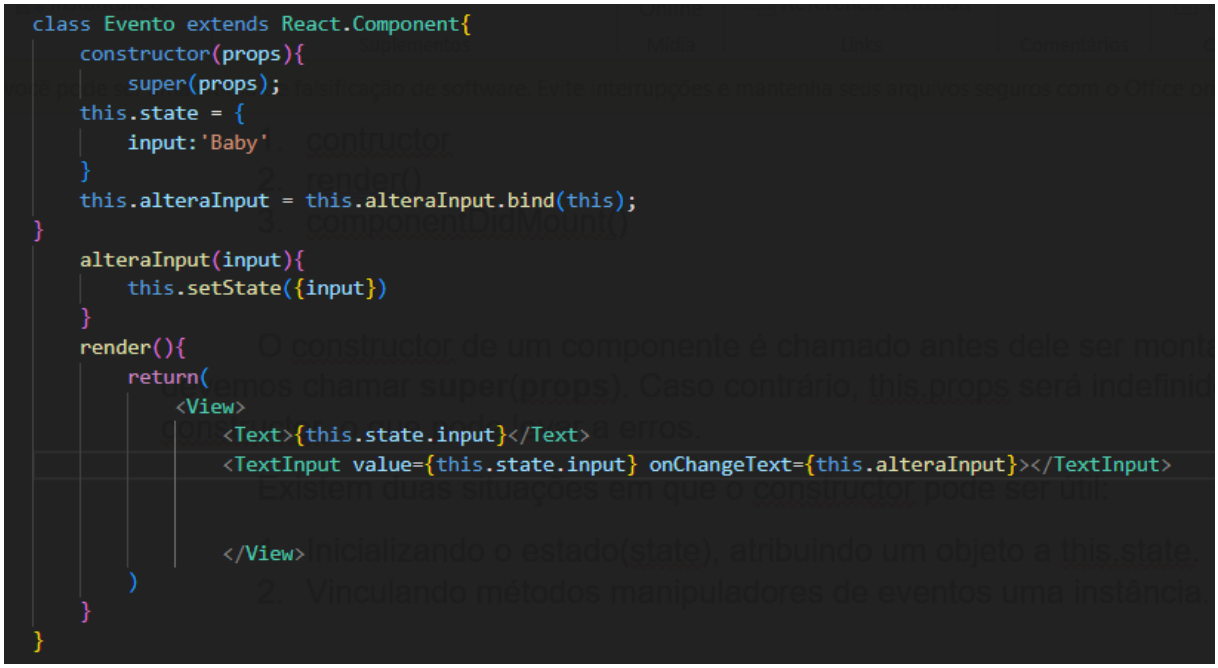
1. contructor
2. render()
3. componentDidMount()

O constructor de um componente é chamado antes dele ser montado e devemos chamar **super**(**props**). Caso contrário, this.props será indefinido no constructor, o que pode levar a erros.

Existem duas situações em que o constructor pode ser útil:

1. Inicializando o estado(state), atribuindo um objeto a this.state.
2. Vinculando métodos manipuladores de eventos uma instância.

Refatoração do evento com o construtor:



```
class Evento extends React.Component{
    constructor(props){
        super(props);
    this.state = {
        input:'Baby'
    }
    this.alteraInput = this.alteraInput.bind(this);
}
    alteraInput(input){
        this.setState({input})
    }
    render(){
        return(
            <View>
                <Text>{this.state.input}</Text>
                <TextInput value={this.state.input} onChangeText={this.alteraInput}></TextInput>


                </View>
        )
    }
}
```

Logo após isso o método que será chamado é o **componentDidMount()** ele é a ultima parte do ciclo de montagem e é perfeito para se ter as **APIs**.

Atualização

Uma atualização pode acontecer por mudanças em seu estado e/ou propriedades. Temos 3 métodos na atualização:

1. **setState()**
2. **forceUpdate()**
3. **componentDidUpdate()**

SetState() é o responsável por enviar mudanças de estado para o componente, nesse método passamos apenas aquelas propriedades de estado que devem ser atualizadas.

O **forceUpdate()** é útil quando precisamos renderizar novamente nosso componente, mas ele não depende exclusivamente das propriedades e dos estados, porém é perigoso para aplicação abusar desse método.

O **componentDidUpdate()** é executado imediatamente quando o componente termina de ser atualizado.

Desmontagem

Os componentes são desmontados no seu final, nesse momento podemos atuar através do método **componentWillUnmount**. Ele é invocado imediatamente antes de um componente ser desmontado e destruído.

Métodos exóticos

Existem alguns métodos extras:

1. **getDerivedStateFromProps**
2. **shouldComponentUpdate**
3. **getSnapshotBeforeUpdate**

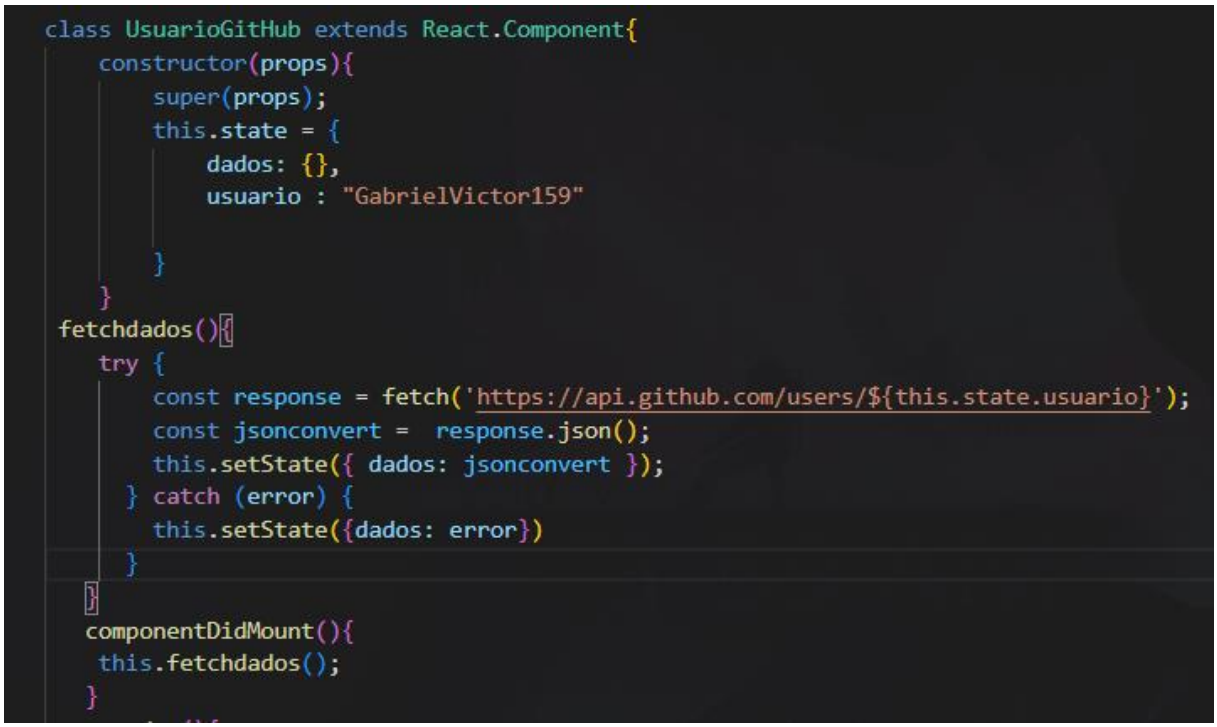
O **getDerivedStateFromProps** é invocado imediatamente antes de chamar o método de render, tanto na montagem quanto na atualização, deve retornar um objeto para atualizar o estado ou nulo para não atualizar nada.

O **shouldComponentUpdate** é utilizado em casos onde queremos que o React saiba se a saída de um componente não é afetada pela alteração atual no estado ou propriedades, esse método é chamado logo antes do render.

O **getSnapshotBeforeUpdate** é invocado imediatamente antes da saída processada mais recentemente ser confirmada para o DOM, permite que nosso componente capture algumas informações do DOM.

AJAX:

Usando o AJAX e os métodos do ciclo de vida dos componentes conseguimos fazer requisições para APIs:



```
class UsuarioGitHub extends React.Component{
    constructor(props){
        super(props);
        this.state = {
            dados: {},
            usuario : "GabrielVictor159"

        }
    }
fetchdados(){
    try {
        const response = fetch('https://api.github.com/users/${this.state.usuario}');
        const jsonconvert =  response.json();
        this.setState({ dados: jsonconvert });
      } catch (error) {
        this.setState({dados: error})
      }
  }
  componentDidMount(){
   this.fetchdados();
  }
```

```
getAllCursos= async() =>{
      fetch(this.state.url)
      .then(response => response.json())
      .then(responseJson=>{
          this.setState({dados: responseJson})
          return responseJson;
      })
      .catch((error) =>{
          console.error(error);
      })
     };


     getOneCurso = async() =>{
      fetch(this.state.url+id)
      .then(response => response.json())
      .then(responseJson=>{
          this.setState({dados: responseJson})
          return responseJson;
      })
      .catch((error) =>{
          console.error(error);
      });
     }
   deleteOneCurso = async() =>{
      fetch(this.state.url+id, {
          method: "DELETE"
      })
      .then(response => response.json())
      .then(responseJson=>{
          this.setState({dados: responseJson})
          return responseJson;
      })
      .catch((error) =>{
          console.error(error);
      });
     }
     postOneCurso = async() =>{
      const requestOptions = {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ nomeDoCurso: nomeDoCurso, valorDoCurso: valorDoCurso})
      };
      fetch(this.state.url+id, {
        requestOptions
```

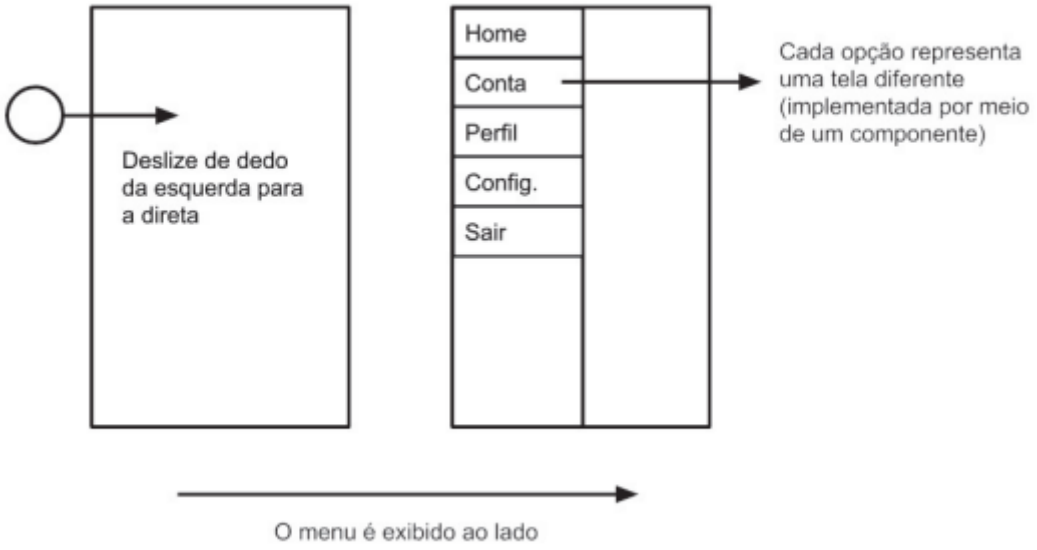
Fazemos isso através do método **fetch**:

Navegação

A navegação entre paginas no React Native não existe, por ser um aplicativo **SGP (Single Page Application)**, o que temos é uma simulação de navegação entre páginas, onde navegamos entre os componentes porem não alteramos de pagina efetivamente.

Navegação lateral:

A navegação lateral é uma técnica que podemos usar para navegar entre páginas, ela se baseia em um menu que aparece quando arrastamos para a esquerda a tela:



Primeiramente precisamos adicionar a biblioteca @**react-navigation/native** para o projeto:

npm install @react-navigation/native

Após isso precisamos adicionar a biblioteca do método **drawer** que é o método de navegação que estamos vendo:

npm install @react-navigation/drawer

Em seguida temos que adicionar algumas bibliotecas secundarias:

expo install react-native-gesture-handler react-native-reanimated react-native-screens react-native-safe-area-context

@react-native-community/masked-view

Agora pode ser que a biblioteca **react-native-reanimated** cause conflitos no código, para resolvermos esses problemas precisamos adicionar uma anotação no **babel.config.js**:

```
module.exports = function(api) {
  api.cache(true);
  return {
    presets: ['babel-preset-expo'],
  };
};
module.exports = function(api) {
  api.cache(true);
  return {
    presets: ['babel-preset-expo'],
    plugins: ['react-native-reanimated/plugin'],
  };
};
```

Agora vamos aplicar nosso método no App.js, primeiro precisamos importar o **NavigationContainer** que será onde será armazenado o método usado para navegação:

```
import { NavigationContainer } from '@react-navigation/native';
```

Em seguida importamos o **createDrawerNavigator** para o App.js:

```
import { createDrawerNavigator } from '@react-navigation/drawer';
```

Esse método será responsável por criar o menu lateral:

```
const Drawer = createDrawerNavigator();
```

Através dessa variável vamos acessar as propriedades do menu:

```
function MyDrawer() {
  return (
    <Drawer.Navigator
    useLegacyImplementation
      screenOptions={{
        drawerStyle: {
          hideStatusBar: true,
          backgroundColor: 'black',
          overlayColor: '#6b52ae',
          contentOptions: {
            activeTintColor: '#fff',
            activeBackgroundColor: '#6b52ae',
          }
        },
      }}
    >
      <Drawer.Screen name="Feed" component={Feed} />
      <Drawer.Screen name="Article" component={Article} />
    </Drawer.Navigator>
  );
}
```

Vamos analizar essa função:

Primeiro temos alguns estilos CSS sendo aplicados no menu através da propriedade **screenOptions**.

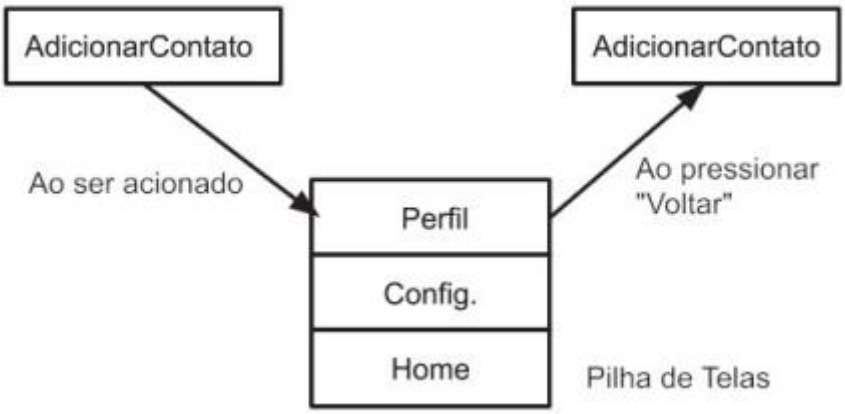
Em seguida temos os atributos **.Screen** responsável por darem um nome para o componente no menu e o segundo para dizer qual é o componente, vale lembrar que ele está sendo acessado através da variável **Drawer** que criamos.

Em seguida vamos utilizar do **NavigationContainer** :

```
export default function App() {
  return (
    <NavigationContainer>
      <MyDrawer />

    </NavigationContainer>

  );
}
```

Navegação por Links:

Na navegação por links vamos usar de botões e links para fazer a navegação, mas também temos que utilizar de um **StackNavigator** esse método serve para voltarmos para páginas anteriores:



Precisamos importar o método **createStackNavigator** para o App.js:

```
import { createStackNavigator } from '@react-navigation/stack';
```

Para isso devemos adicionar essa biblioteca no projeto:

npm install @react-navigation/stack

Agora precisamos criar o método do stack e listar os componentes que fazem parte da pilha:

```
const Stack = createStackNavigator();

function MyStack() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Home" component={Home} />
      <Stack.Screen name="Feed" component={Feed} />
      <Stack.Screen name="Article" component={Article} />
    </Stack.Navigator>
  );
}
```

Em seguida adicionamos esse método no NavigationContainer:

```
export default function App() {
  return (
    <NavigationContainer>
      <MyStack />

    </NavigationContainer>

  );
}
```

Personalizando o cabeçalho:

Existem três propriedades principais a serem usadas ao personalizar o estilo do seu cabeçalho: **headerStyle, headerTintColore headerTitleStyle**.

- **headerStyle:** um objeto de estilo que será aplicado ao Viewque envolve o cabeçalho. Se você definir backgroundColor, essa será a cor do seu cabeçalho.
- **headerTintColor:** o botão voltar e o título usam essa propriedade como sua cor. No exemplo abaixo, definimos a cor da tonalidade para branco (#fff) para que o botão Voltar e o título do cabeçalho sejam brancos.
- **headerTitleStyle**: se quisermos personalizar o fontFamily, fontWeighte outras Textpropriedades de estilo para o título, podemos usar isso para fazer isso.

```
  return (
    <Stack.Navigator>
      <Stack.Screen
        name="Home"
        component={HomeScreen}
        options={{
          title: 'My home',
          headerStyle: {
            backgroundColor: '#f4511e',
          },
          headerTintColor: '#fff',
          headerTitleStyle: {
            fontWeight: 'bold',
          },
        }}
      />
    </Stack.Navigator>
  );
}
```

Agora precisamos criar os links de navegação, basicamente vamos usar de botões que irão nos direcionar para os componentes:

```
<Button
  title="Go to Details"
  onPress={() => navigation.navigate('Details')}
/>
```

Temos que nos atentar no método **navigate** ele é apenas um dos métodos do navigation e cada um deles possui uma função especifica:

Navigation()->Basicamente navegamos entre os componentes através das rotas nomeadas no stack.

Push()->Basicamente adicionamos outra rota independentemente do histórico de navegação existente.

GoBack()->Volta para a página anterior do histórico.

PopToTop()->Volta para a primeira página do histórico.

Route Params:

Podemos passar parâmetros para as rotas através do atributo **route.params** para isso basta usarmos de **{}** :

Exemplos:

```
navigation.navigate('Profile', {
  user: {
    id: 'jane',
    firstName: 'Jane',
    lastName: 'Done',
    age: 25,
  },
});
```

```
navigation.navigate('Profile', { userId: 'jane' });
```

```
onPress={() => {
  // Pass and merge params back to home screen
  navigation.navigate({
    name: 'Home',
    params: { post: postText },
    merge: true,
  });
```

Navegação por abas:

Nesse formato, um rodapé é criado e as guias disponíveis são disponibilizadas para o usuário:

Para isso precisamos adicionar a biblioteca **@react-navigation/bottom-tabs** no projeto:

npm install @react-navigation/bottom-tabs

Em seguida adicionamos o método **createBottomTabNavigator** no App.js:

```
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
```

Após isso é extremamente simples criar uma barra de navegação:

```
const TabNavigator = createBottomTabNavigator({
Home: Home,
Feed: Feed,
Article: Article
});
```

Após isso basta adicionarmos ao **NavigationContainer**:

```
<NavigationContainer>
  <TabNavigator />

</NavigationContainer>
```

OBS: se tentarmos usar mais uma técnica de navegação como o **stack** o cabeçalho que tínhamos anteriormente simplesmente não existira mais, pois o react entende que é melhor o usuário ter apenas uma forma de menu de navegação.

Personalizando o rodapé de navegação:

```
<Tab.Navigator
  screenOptions={({ route }) => ({
    tabBarIcon: ({ focused, color, size }) => {
      let iconName;

      if (route.name === 'Home') {
        iconName = focused
          ? 'ios-information-circle'
          : 'ios-information-circle-outline';
      } else if (route.name === 'Settings') {
        iconName = focused ? 'ios-list-box' : 'ios-list';
      }

      // You can return any component that you like here!
      return <Ionicons name={iconName} size={size} color={color} />;
    },
    tabBarActiveTintColor: 'tomato',
    tabBarInactiveTintColor: 'gray',
  })}
>
  <Tab.Screen name="Home" component={HomeScreen} />
  <Tab.Screen name="Settings" component={SettingsScreen} />
</Tab.Navigator>
```

- **tabBarIcon** é uma opção suportada no navegador da guia inferior. Portanto, sabemos que podemos usá-lo em nossos componentes de tela no **options** prop, mas neste caso optamos por colocá-lo no **screenOptionsprop Tab.Navigator** para centralizar a configuração do ícone por conveniência.
- **tabBarIcon** é uma função que recebe o **focused** estado, **color**, e **size** params. Se você der uma olhada mais abaixo na configuração, verá **tabBarActiveTintColor** e **tabBarInactiveTintColor**. Esses padrões são os padrões da plataforma iOS, mas você pode alterá-los aqui. O **color** que é passado para o **tabBarIcon** é o ativo ou inativo, dependendo do **focused** estado (focado é ativo). O **size** é o tamanho do ícone esperado pela barra de guias.
- Leia a referência completa da API para obter mais informações sobre as **createBottomTabNavigator** opções de configuração.

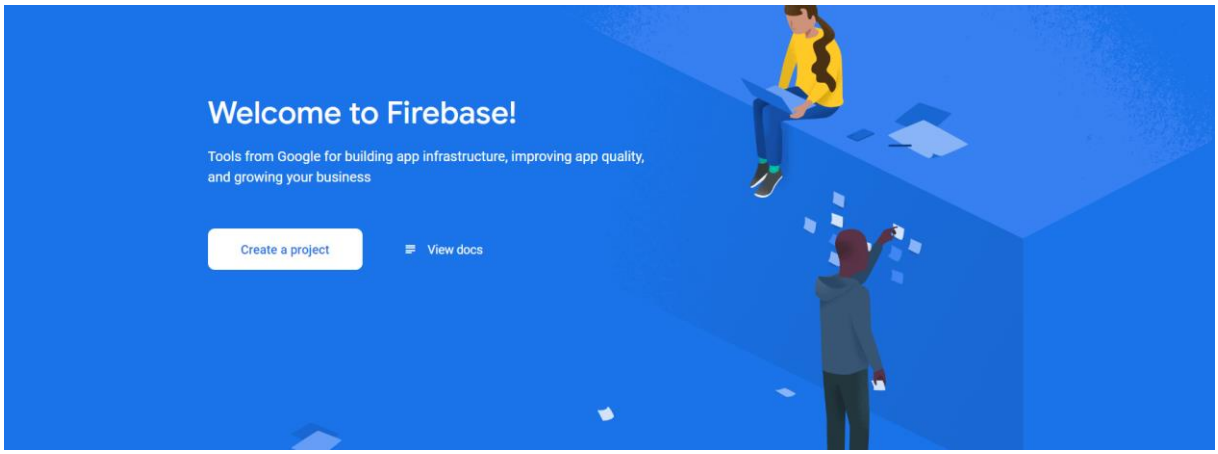
Integração com Firebase e banco de dados

O Firebase é o que chamamos de BaaS (Backend as a Service) para aplicações web e mobile, desenvolvido e mantido pelo google.

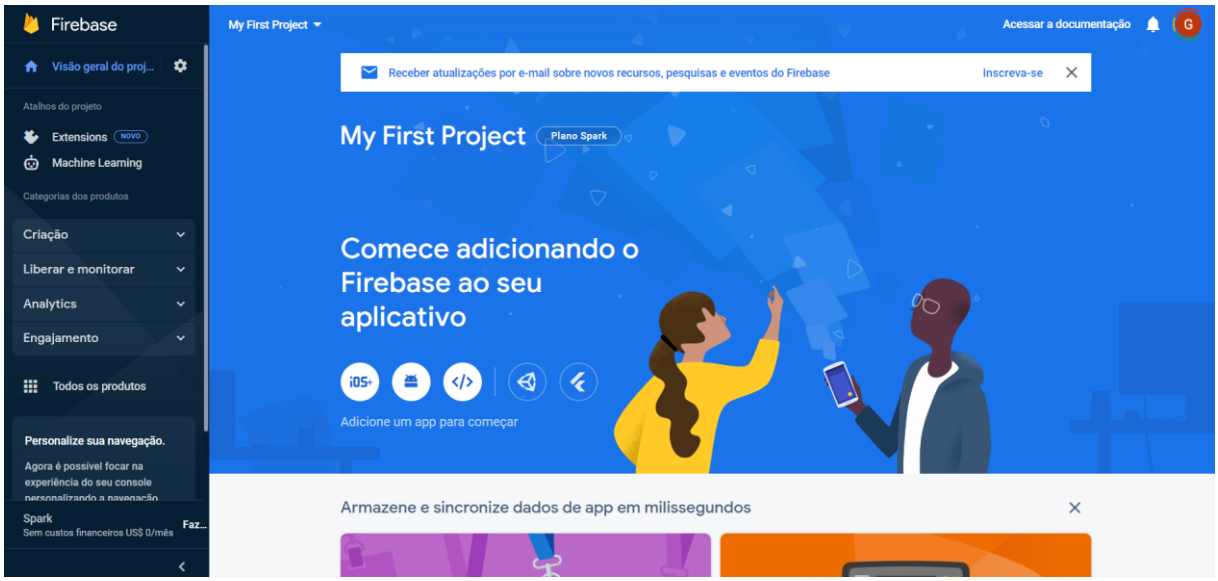
Configuração:

O banco de dados em tempo real do Firebase permite armazenamento e sincronismo dos dados entre usuários e dispositivos em tempo real com um banco de dados NoSQL hospedado em nuvem, o Firebase em sua versão gratuita possui algumas limitações nos recursos.

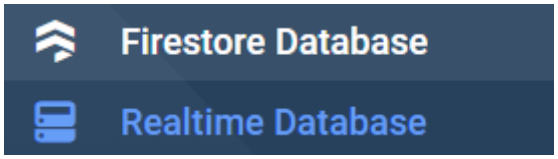
Para usarmos primeiro acessamos o site https://firebase.google.com/ depois entramos na nossa conta google, em seguida clique na opção “ir para o console”:



Em seguida clique em create a Project, após isso configure seu projeto e o crie:



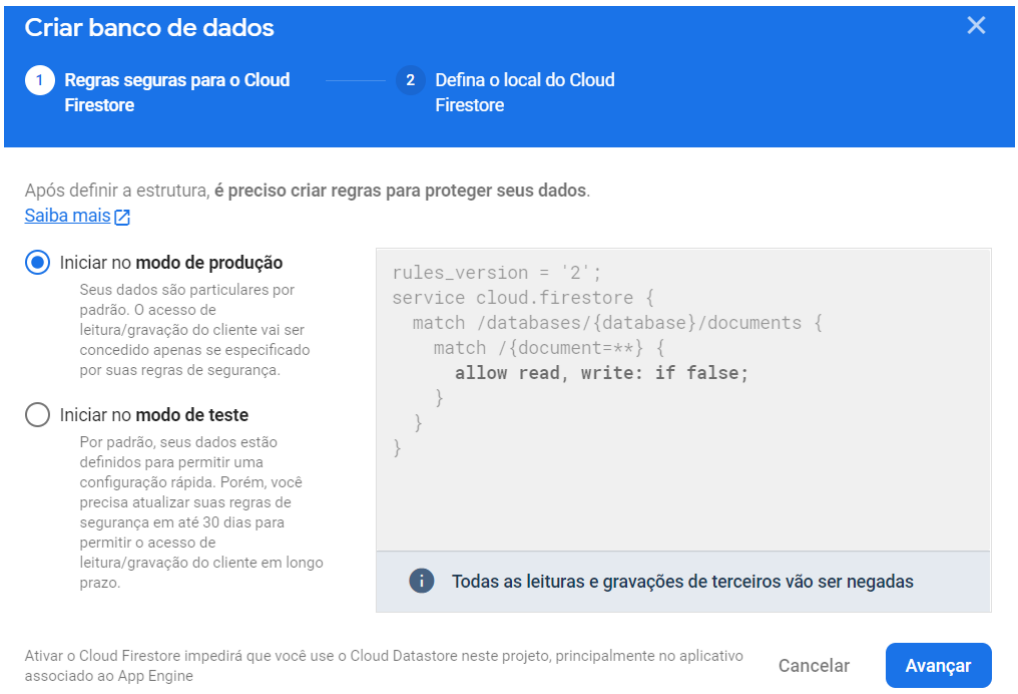
Como podemos ver o Firebase oferece diversas funcionalidades em CLOUD para usarmos no nosso projeto, a que vamos usar aqui é a **database**:



Temos essas duas opções:

- O **Cloud Firestore** é o mais novo banco de dados do Firebase para o desenvolvimento de apps para dispositivos móveis. Ele se baseia nos resultados do Realtime Database com um novo modelo de dados mais intuitivo. O Cloud Firestore também tem consultas mais avançadas e rápidas, além de melhor escalonamento que o Realtime Database.
- O **Realtime Database** é o banco de dados original do Firebase. Ele é uma solução eficiente e de baixa latência para aplicativos móveis que exigem estados sincronizados entre clientes em tempo real.

Vamos usar o **Cloud Firestore** primeiro clique em criar banco de dados:

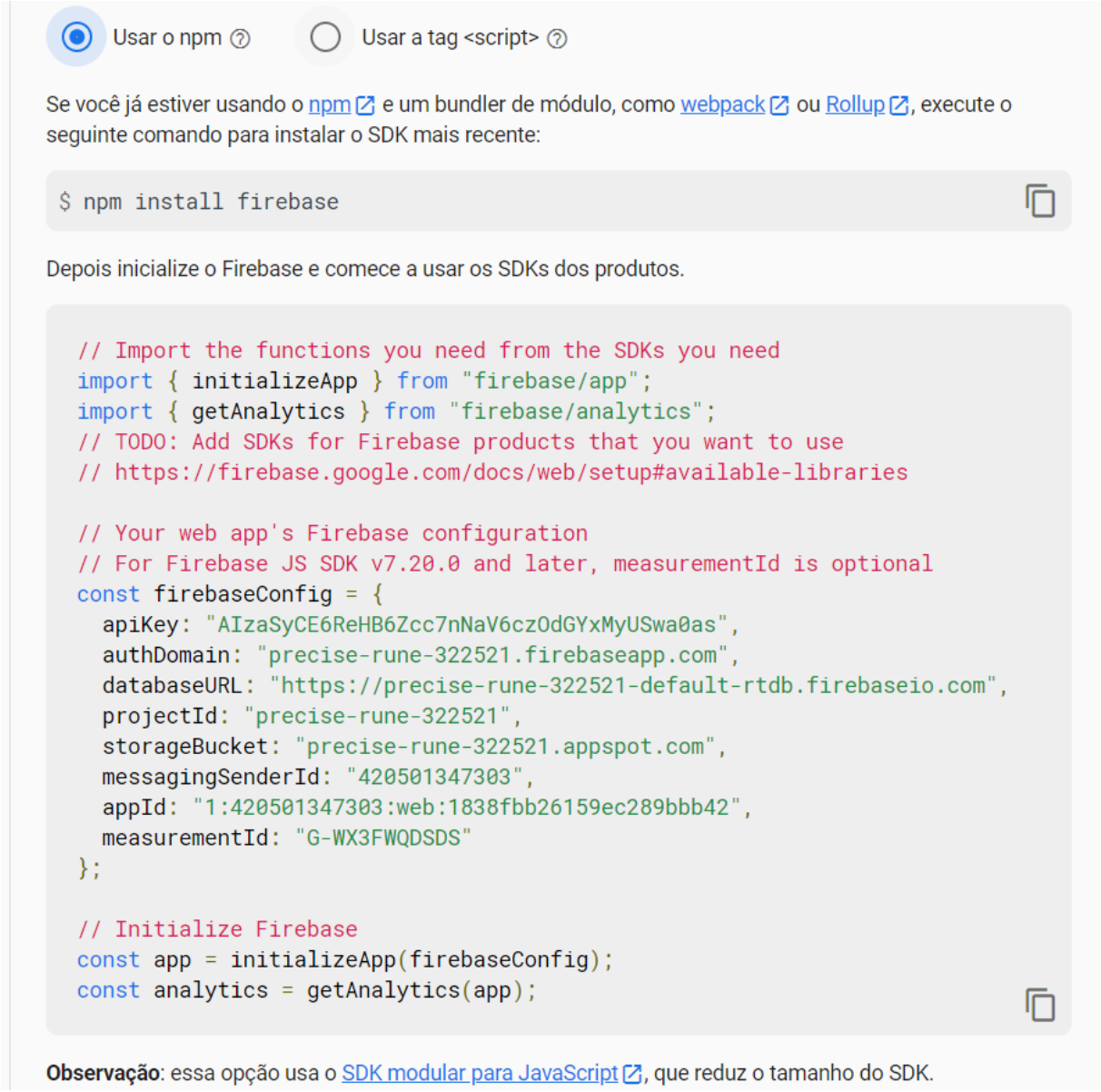


Escolha o modo que melhor te agrada.

Escolha um local do CLOUD, recomendo o mais próximo possível da sua região.

Após criarmos os bancos de dados vamos ate a pagina inicial do firebase e vamos atrás da opção para adicionarmos o firebase no app:

Após isso vamos registrar o App e após isso teremos essa tela:



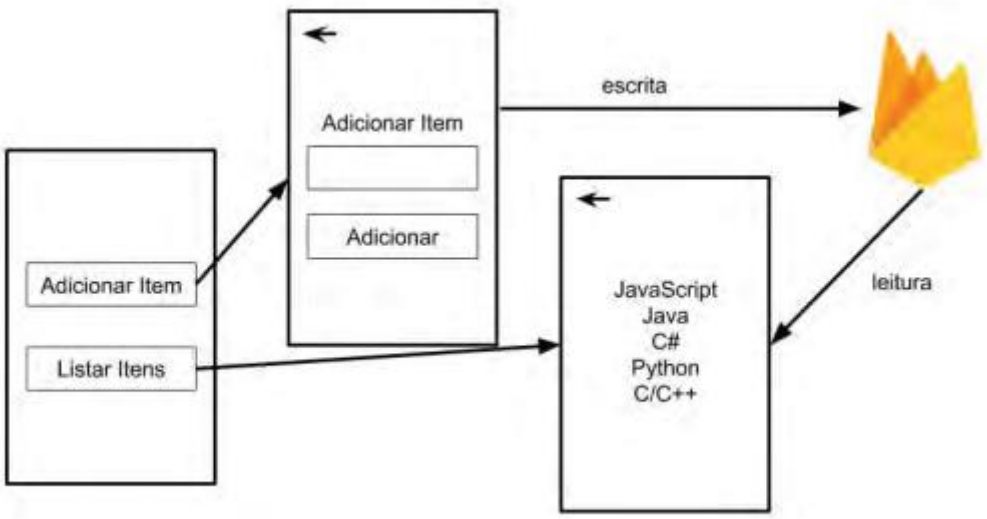
Podemos ver que teremos que adicionar uma dependência no nosso projeto:

npm install firebase

E temos alguns códigos de configuração do firebase no projeto.

Aplicação:

Vamos criar um exemplo onde teremos uma tela com dois botões adicionar item e listar itens:



Integração:

Como vamos ter que adicionar uma configuração ao projeto, vamos criar a pasta config e dentro o arquivo config.js.

Dentro do config.js, vamos adicionar os códigos referentes a configuração do firebase:

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
import { getAnalytics } from "firebase/analytics";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
const firebaseConfig = {
  apiKey: "AIzaSyCE6ReHB6Zcc7nNaV6czOdGYxMyUSwa0as",
  authDomain: "precise-rune-322521.firebaseapp.com",
  databaseURL: "https://precise-rune-322521-default-rtdb.firebaseio.com",
  projectId: "precise-rune-322521",
  storageBucket: "precise-rune-322521.appspot.com",
  messagingSenderId: "420501347303",
  appId: "1:420501347303:web:1838fbb26159ec289bbb42",
  measurementId: "G-WX3FWQDSDS"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
const analytics = getAnalytics(app);
```

Vamos exportar essas configurações como **DB.**

Após isso vamos na pasta componentes e iremos criar os seguintes componentes:

1.Inicial.js

2.AdicionarItens.js

3.Itens.js

4.ListaItens.js

1.

```
import React from "react";
import { View, Text, Button, StyleSheet } from "react-native";

export default class Inicial extends React.Component{
    render(){
        return(
            <View style={styles.conteudoBtn}>
                <Button
                title="Adicionar Item"
                color="blue"
                onPress={()=>
                    this.props.navigation.navigate('AdicionarItens')
                }
                />
                <Button
                title="Listar Itens"
                color="blue"
                onPress={()=>
                    this.props.navigation.navigate('ListarItens')
                }
                />
            </View>
        )
    }
}
const styles= StyleSheet.create({
        conteudoBtn:{
            flex:1,
            flexDirection:'column',
            justifyContent:'center'
        },
});
```

2.

roApp > componentes > AdicionarItens.js > AdicionarItens > render

```
import React from "react";
import { View, Text, Button, TouchableHighlight, Alert , StyleSheet } from "react-native";
import { TextInput } from "react-native-gesture-handler";
import DB from "../config/config";


export default class AdicionarItens extends React.Component{
    state = {
        item:''
    };
    salvaItem = () =>{
        DB.ref('/itens').push({
            item: this.state.item
        });
        Alert.alert('Item salvo!');
    };
    render(){
        return(
            <View>
                <Text>
                    Adicionar Item
                </Text>
                <TextInput
                onChangeText={
                    item =>{ this.setState({item}) }
                }
                />
                <TouchableHighlight
                underlayColor='white'
                onPress = {this.salvaItem}
                >
                <Text>
                    Adicionar
                </Text>
                </TouchableHighlight>
            </View>
        );
    }
}
```

Perceba o que fizemos aqui: Criamos um método para salvar os itens utilizando-se do **config.js** aqui nomeado como DB e criamos uma interação para que o item no state seja atualizado com o parâmetro que o usuário irá escrever.

Hooks

O que são?

Hooks são uma nova adição no React 16.8. Eles permitem que você use o estado e outros recursos do React sem escrever uma classe.

State Hook:

Este exemplo renderiza um contador. Ao clicar no botão, ele incrementa o valor:

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Nós o chamamos dentro de um componente de função para adicionar algum estado local a ele. O React preservará esse estado entre as re-renderizações. useStateretorna um par: o valor do estado *atual* e uma função que permite atualizá-lo.