REACT-NATIVE

**SEC-2: REFORZAMIENTO DE LAS BASES**

**INICIO DE PROYECTO – REACT CON TYPESCRIPT**

Crear el proyecto de react con typescript con el comando:

**npx create-react-app my-app --template typescript**

Y a continuación, en el archivo **public/index.html** importar Bootstrap:

...

<link

      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css"

      rel="stylesheet"

      integrity="sha384-KK94CHFLLe+nY2dmCWGMq91rCGa5gtU4mk92HdvYe+M/SXH301p5ILy+dN9+nJOZ"

      crossorigin="anonymous"

    />

<title>Reforzamiento TS - React</title>

...

**PRIMEROS FUNCTIONALS COMPONENTS EN TS**

Borramos todo lo que no utilicemos como el index.css, etc.

Creamos la carpeta **Typescript** dentro de la carpeta **src**, y creamos el archivo **TiposBasicos.tsx**.

Y dentro del archivo **index.html** le damos la **class=”container”** al body.

<body class="container">

**OBJETOS LITERALES E INTERFACES**

Creamos el archivo **typescript/ObjetosLiterales.tsx:**

interface Persona {

  nombre: string;

  edad: number;

  aísción: Direccion;

}

interface Direccion {

  aís: string;

  casaNo: number;

}

export const ObjetosLiterales = () => {

  const persona: Persona = {

    nombre: “Joaquín”,

    edad: 25,

    aísción: {

      aís: “Argentina”,

      casaNo: 420,

    },

  };

  return (

    <>

      <h3>Objetos Literales</h3>

      <code>

        <pre>{JSON.stringify(persona, null, 2)}</pre>

      </code>

    </>

  );

};

**FUNCIONES, RETORNO Y ARGUMENTOS**

Creamos el archivo **typecript/Funciones.tsx:**

export const Funciones = () => {

  const sumar = (a: number, b: number): number => {

    return a + b;

  };

  return (

    <>

      <h3>Funciones</h3>

      <span>El resultado es {sumar(1, 2)}</span>

    </>

  );

};

**HOOK – useState**

Creamos la carpeta **components** dentro de **src**, con el archivo **Contador.tsx**

import { useState } from "react";

export const Contador = () => {

  const [value, setValue] = useState(0);

  const acumular = (numero: number) => {

    setValue(value + numero);

  };

  return (

    <>

      <h3>

        Contador: <small>{value}</small>

      </h3>

      <button className="btn btn-primary" onClick={() => acumular(1)}>

        +1

      </button>

      &nbsp;

      <button className="btn btn-primary" onClick={() => acumular(-1)}>

        -1

      </button>

    </>

  );

};

**CUSTOM HOOK – useCounter**

Creamos la carpeta **hooks** con el archivo **useCounter.tsx** en donde va a recibir toda la lógica del useState que utilizamos en el componente **Contador.tsx**:

import { useState } from "react";

export const useCounter = (inicial: number = 10) => {

  const [value, setValue] = useState(inicial);

  const acumular = (numero: number) => {

    setValue(value + numero);

  };

  return {

    value,

    acumular,

  };

};

Y ahora en la carpeta **components** creamos el archivo **ContadorConHook.tsx** e importamos este **useCounter.tsx:**

import { useCounter } from "../hooks/useCounter";

export const ContadorConHook = () => {

  const { value, acumular } = useCounter(0);

  return (

    <>

      <h3>

        Contador con hook: <small>{value}</small>

      </h3>

      <button className="btn btn-primary" onClick={() => acumular(1)}>

        +1

      </button>

      &nbsp;

      <button className="btn btn-primary" onClick={() => acumular(-1)}>

        -1

      </button>

    </>

  );

};

**COMPONENTE LOGIN – ANTESALA AL useReducer**

Creamos el componente **components/Login.tsx**:

export const Login = () => {

  return (

    <>

      <h3>Login</h3>

      <div className="alert alert-info">Validando...</div>

      <div className="alert alert-danger">No autenticado</div>

      <div className="alert alert-success">Autenticado</div>

      <button className="btn btn-primary">Login</button>

      <button className="btn btn-danger">Log out</button>

    </>

  );

};

**HOOK – useReducer**

Ahora importamos dentro del componente **Login.tsx** el **useReducer** y creamos la **interface** para el state, el **initialState**, el **type** para el action y la **función** para ejecutar en el useReducer:

import { useReducer } from "react";

interface AuthState {

  validando: boolean;

  token: string | null;

  userName: string;

  name: string;

}

const initialState: AuthState = {

  validando: true,

  token: null,

  userName: "",

  name: "",

};

type AuthAction = {

  type: "logout";

};

const authReducer = (state: AuthState, action: AuthAction): AuthState => {

  switch (action.type) {

    case "logout":

      return { validando: true, token: null, userName: "", name: "" };

    default:

      return state;

  }

};

export const Login = () => {

  const [state, dispatch] = useReducer(authReducer, initialState);

  return (

    <>

      <h3>Login</h3>

      <div className="alert alert-info">Validando...</div>

      <div className="alert alert-danger">No autenticado</div>

      <div className="alert alert-success">Autenticado</div>

      <button className="btn btn-primary">Login</button>

      <button className="btn btn-danger">Log out</button>

    </>

  );

};

**AUTH-REDUCER**

Implementamos la lógica para que dependiendo el valor de **validando** en el state, muestre una u otra cosa mediante un **useEffect** el cual después de 1 segundo y medio ejecuta el **dispatch** del reducer:

import { useEffect, useReducer } from "react";

interface AuthState {

  validando: boolean;

  token: string | null;

  userName: string;

  name: string;

}

const initialState: AuthState = {

  validando: true,

  token: null,

  userName: "",

  name: "",

};

type AuthAction = {

  type: "logout";

};

const authReducer = (state: AuthState, action: AuthAction): AuthState => {

  switch (action.type) {

    case "logout":

      return { validando: false, token: null, userName: "", name: "" };

    default:

      return state;

  }

};

export const Login = () => {

  const [state, dispatch] = useReducer(authReducer, initialState);

  useEffect(() => {

    setTimeout(() => {

      dispatch({ type: "logout" });

    }, 1500);

  }, []);

  if (state.validando) {

    return (

      <>

        <h3>Login</h3>

        <div className="alert alert-info">Validando...</div>

      </>

    );

  }

  return (

    <>

      <h3>Login</h3>

      <div className="alert alert-danger">No autenticado</div>

      <div className="alert alert-success">Autenticado</div>

      <button className="btn btn-primary">Login</button>

      <button className="btn btn-danger">Log out</button>

    </>

  );

};

**LOGIN y LOGOUT**

Creamos las funciones de Login y Logout para pasarle a los button y ejecutamos el payload que tiene un tipado específico de lo que tiene que recibir para logearse:

import { useEffect, useReducer } from "react";

interface AuthState {

  validando: boolean;

  token: string | null;

  userName: string;

  name: string;

}

const initialState: AuthState = {

  validando: true,

  token: null,

  userName: "",

  name: "",

};

type LoginPayload = {

  userName: string;

  name: string;

};

type AuthAction = { type: "logout" } | { type: "login"; payload: LoginPayload };

const authReducer = (state: AuthState, action: AuthAction): AuthState => {

  switch (action.type) {

    case "logout":

      return { validando: false, token: null, userName: "", name: "" };

    case "login":

      const { name, userName } = action.payload;

      return {

        validando: false,

        token: "abc123",

        userName,

        name,

      };

    default:

      return state;

  }

};

export const Login = () => {

  const [state, dispatch] = useReducer(authReducer, initialState);

  useEffect(() => {

    setTimeout(() => {

      dispatch({ type: "logout" });

    }, 1500);

  }, []);

  const login = () => {

    dispatch({

      type: "login",

      payload: { userName: "joaquincaggiano", name: "Joaquín" },

    });

  };

  const logout = () => {

    dispatch({ type: "logout" });

  };

  if (state.validando) {

    return (

      <>

        <h3>Login</h3>

        <div className="alert alert-info">Validando...</div>

      </>

    );

  }

  return (

    <>

      <h3>Login</h3>

      {state.token ? (

        <>

          <div className="alert alert-success">

            Autenticado com: {state.name}

          </div>

          <button className="btn btn-danger" onClick={logout}>

            Log out

          </button>

        </>

      ) : (

        <>

          <div className="alert alert-danger">No autenticado</div>

          <button className="btn btn-primary" onClick={login}>

            Login

          </button>

        </>

      )}

    </>

  );

};

**PETICIONES HHTP – AXIOS**

Creamos el componente **components/Usuarios.tsx** y la carpeta **api/reqRes.tsx** la cual contendrá el llamado a la api que vamos a utilizar para traer usuarios.

**reqRes.tsx:**

import axios from "axios";

export const reqResApi = axios.create({

    baseURL: "https://reqres.in/api"

});

**Usuarios.tsx:**

import { useEffect } from "react";

import { reqResApi } from "../api/reqRes";

export const Usuarios = () => {

  useEffect(() => {

    reqResApi

      .get("/users")

      .then((resp) => {

        console.log(resp.data.data);

      })

      .catch((err) => console.log(err));

  }, []);

  return (

    <>

      <h3>Usuarios</h3>

      <table className="table">

        <thead>

          <tr>

            <th>Avatar</th>

            <th>Nombre</th>

            <th>Email</th>

          </tr>

        </thead>

        <tbody></tbody>

      </table>

    </>

  );

};

**ESTABLECER EL TIPO DE LAS RESPUESTAS HTTP**

Vamos a crear la carpeta **interfaces** con el archivo **reqRes.tsx** en donde va a tener la interface con el tipado especifico de la respuesta http que estamos utilizando en el componente **Usuarios.tsx**.

Para hacer el tipado más rápido vamos a utilizar la página <https://app.quicktype.io/> en la cual debemos copiar y pegar la respuesta JSON y nos hará el tipado automáticamente.

**reqRes.tsx:**

export interface ReqResListado {

    page:        number;

    per\_page:    number;

    total:       number;

    total\_pages: number;

    data:        User[];

    support:     Support;

}

export interface User {

    id:         number;

    email:      string;

    first\_name: string;

    last\_name:  string;

    avatar:     string;

}

export interface Support {

    url:  string;

    text: string;

}

**Usuarios.tsx**: importamos esta interface y le decimos al axios.get que es de este tipo la respuesta. Además creamos un **state** y le decimos que es de tipo **User[]**.

import { useEffect, useState } from "react";

import { reqResApi } from "../api/reqRes";

import { ReqResListado, User } from "../interfaces/reqRes";

export const Usuarios = () => {

    const [users, setUsers] = useState<User[]>([])

  useEffect(() => {

    reqResApi

      .get<ReqResListado>("/users")

      .then((resp) => {

        setUsers(resp.data.data)

      })

      .catch((err) => console.log(err));

  }, []);

  return (

    <>

      <h3>Usuarios</h3>

      <table className="table">

        <thead>

          <tr>

            <th>Avatar</th>

            <th>Nombre</th>

            <th>Email</th>

          </tr>

        </thead>

        <tbody></tbody>

      </table>

    </>

  );

};

**MOSTRAR USUARIOS EN PANTALLA**

Ahora que ya tenemos los users en el state con el tipado especificado podemos mostrarlos en pantalla. Para eso vamos a crear una función que renderice un elemento JSX con los datos de cada usuario:

import { useEffect, useState } from "react";

import { reqResApi } from "../api/reqRes";

import { ReqResListado, User } from "../interfaces/reqRes";

export const Usuarios = () => {

  const [users, setUsers] = useState<User[]>([]);

  useEffect(() => {

    reqResApi

      .get<ReqResListado>("/users")

      .then((resp) => {

        setUsers(resp.data.data);

      })

      .catch((err) => console.log(err));

  }, []);

  const renderItem = (user: User) => {

    return (

      <tr key={user.id.toString()}>

        <th>

          <img

            src={user.avatar}

            alt={user.first\_name}

            style={{ width: "50px", borderRadius: "100px" }}

          />

        </th>

        <th>

          {user.first\_name} {user.last\_name}

        </th>

        <th>{user.email}</th>

      </tr>

    );

  };

  return (

    <>

      <h3>Usuarios</h3>

      <table className="table">

        <thead>

          <tr>

            <th>Avatar</th>

            <th>Nombre</th>

            <th>Email</th>

          </tr>

        </thead>

        <tbody>{users.map((user) => renderItem(user))}</tbody>

      </table>

      <button className="btn btn-primary">Next</button>

    </>

  );

};

**CREAR UNA PEQUEÑA PAGINACIÓN**

Creamos la función **loadUsers** la cual le pasamos como parámetro la **page**, el cual va a tener el valor de un useRef que creamos con el valor de 1.

import { useEffect, useRef, useState } from "react";

import { reqResApi } from "../api/reqRes";

import { ReqResListado, User } from "../interfaces/reqRes";

export const Usuarios = () => {

  const [users, setUsers] = useState<User[]>([]);

  const pageRef = useRef(1)

  useEffect(() => {

    loadUsers();

  }, []);

  const loadUsers = async () => {

    const resp = await reqResApi.get<ReqResListado>("/users", {

        params: {

            page: pageRef.current

        }

    });

    if(resp.data.data.length > 0) {

        setUsers(resp.data.data);

        pageRef.current ++;

    } else {

        alert("No hay más registros")

    }

  };

  const renderItem = (user: User) => {

    return (

      <tr key={user.id.toString()}>

        <th>

          <img

            src={user.avatar}

            alt={user.first\_name}

            style={{ width: "50px", borderRadius: "100px" }}

          />

        </th>

        <th>

          {user.first\_name} {user.last\_name}

        </th>

        <th>{user.email}</th>

      </tr>

    );

  };

  return (

    <>

      <h3>Usuarios</h3>

      <table className="table">

        <thead>

          <tr>

            <th>Avatar</th>

            <th>Nombre</th>

            <th>Email</th>

          </tr>

        </thead>

        <tbody>{users.map((user) => renderItem(user))}</tbody>

      </table>

      <button className="btn btn-primary" onClick={loadUsers}>

        Next

      </button>

    </>

  );

};

**CUSTOM HOOK – useUsers**

Creamos dentro de la carpeta **hooks** el archivo **useUser.tsx**:

import { useEffect, useRef, useState } from "react";

import { reqResApi } from "../api/reqRes";

import { ReqResListado, User } from "../interfaces/reqRes";

export const useUser = () => {

  const [users, setUsers] = useState<User[]>([]);

  const pageRef = useRef(1);

  useEffect(() => {

    loadUsers();

  }, []);

  const loadUsers = async () => {

    const resp = await reqResApi.get<ReqResListado>("/users", {

      params: {

        page: pageRef.current,

      },

    });

    if (resp.data.data.length > 0) {

      setUsers(resp.data.data);

      pageRef.current++;

    } else {

      alert("No hay más registros");

    }

  };

  return {

    users,

    loadUsers,

  };

};

Y ahora en el componente **Usuarios.tsx** lo importamos y desestructuramos la información que necesitamos:

import { useUser } from "../hooks/useUser";

export const Usuarios = () => {

  const { users, loadUsers } = useUser();

...

**SIGUIENTES Y ANTERIORES REGISTROS – PAGINACIÓN**

Ya no vamos a utilizar la función **loadUsers()** en el componente, sino que vamos a crear dos funciones, una para ir a la página siguiente y otra para volver de página.

**useUser.tsx:**

import { useEffect, useRef, useState } from "react";

import { reqResApi } from "../api/reqRes";

import { ReqResListado, User } from "../interfaces/reqRes";

export const useUser = () => {

  const [users, setUsers] = useState<User[]>([]);

  const pageRef = useRef(1);

  useEffect(() => {

    loadUsers();

  }, []);

  const loadUsers = async () => {

    const resp = await reqResApi.get<ReqResListado>("/users", {

      params: {

        page: pageRef.current,

      },

    });

    if (resp.data.data.length > 0) {

      setUsers(resp.data.data);

    } else {

      pageRef.current--;

      alert("No hay más registros");

    }

  };

  const nextPage = () => {

    pageRef.current++;

    loadUsers();

  };

  const prevPage = () => {

    if (pageRef.current > 1) {

      pageRef.current--;

      loadUsers();

    }

  };

  return {

    users,

    nextPage,

    prevPage,

  };

};

**Usuarios.tsx:**

...

const { users, nextPage, prevPage } = useUser();

...

<button className="btn btn-primary" onClick={prevPage}>

  Prev

</button>

&nbsp;

<button className="btn btn-primary" onClick={nextPage}>

  Next

</button>

**FORMULARIOS**

Creamos el componente **Formularios.tsx** el cual va a tener dos inputs donde se manejan sus value con un onChange:

import { useState } from "react";

export const Formularios = () => {

  const [form, setForm] = useState({

    email: "test@gmail.com",

    password: "123456",

  });

  const onChange = (value: string, campo: string) => {

    setForm({

      ...form,

      [campo]: value,

    });

  };

  return (

    <>

      <h3>Formularios</h3>

      <input

        type="text"

        className="form-control"

        placeholder="Email"

        value={form.email}

        onChange={(e) => onChange(e.target.value, "email")}

      />

      <input

        type="password"

        className="form-control my-2"

        placeholder="Password"

        value={form.password}

        onChange={(e) => onChange(e.target.value, "password")}

      />

    </>

  );

};

**CUSTOM HOOK – useForm<Genéricos>**

Creamos dentro de la carpeta **hooks** el archivo **useForm.tsx**, el cual va a recibir un formulario que va a ser un objeto de tipo genérico que va a recibir como parámetro.

import { useState } from "react";

export const useForm = <T extends Object>(formulario: T) => {

  const [state, setState] = useState(formulario);

  const onChange = (value: string, campo: keyof T) => {

    setState({

      ...state,

      [campo]: value,

    });

  };

  return {

    state,

    onChange,

  };

};

Y ahora en el componente **Formulario.tsx** es donde mandamos ese objeto:

import { useForm } from "../hooks/useForm";

export const Formularios = () => {

  const {state, onChange} = useForm({

    email: "test@gmail.com",

    password: "123456",

  });

  return (

    <>

      <h3>Formularios</h3>

      <input

        type="text"

        className="form-control"

        placeholder="Email"

        value={state.email}

        onChange={(e) => onChange(e.target.value, "email")}

      />

      <input

        type="password"

        className="form-control my-2"

        placeholder="Password"

        value={state.password}

        onChange={(e) => onChange(e.target.value, "password")}

      />

    </>

  );

};

**SEC-3: INSTALACIÓN Y CONFIGURACIÓN DE REACT NATIVE CON EMULADORES Y DISPOSITIVOS FÍSICOS**

**DIFERENCIA ENTRE EXPO CLI y REACT NATIVE CLI**

|  |  |
| --- | --- |
| **Expo CLI** | **React Native CLI** |
| Expo se encarga de controlar los builds para IOS y Android. | Genera un proyecto con lo mínimo necesario para correr. |
| Provee muchas funcionalidades nativas incluidas. | Genera los proyectos nativos para IOS y Android de forma independiente. |
| Es excelente para alguien que empieza el desarrollo móvil. | Te permite escribir código nativo por plataforma (en caso de ser necesario). |
| Permite correr y probar la aplicación en IOS aunque no dispongas de una MAC. |  |

**Expo CLI:** usa el **EJECT** en caso de que un módulo nativo no sea soportado por Expo SDK.

NO HAGAS UN EJECT SI:

* Lo que deseas es distribuir en las App stores.
* Estás inconforme con el código nativo.
* Disfrutas de las actualizaciones que vienen con Expo.
* Si usas Expo Push Notificacion Services.
* Dependes de la comunidad de Expo.

Contras:

|  |  |
| --- | --- |
| **Expo CLI** | **React Native CLI** |
| No puedes extender una funcionalidad nativa no soportada por expo (a menos que uses el EJECT) | Necesitas los ambientes de desarrollo por separado. Android Studio (las herramientas al menos) y Xcode. |

Pros comunes en ambos:

* La mayoría de los paquetes soportan ambos ambientes con instalaciones independientes.
* Cambios en caliente (hot reload).
* Desarrollar y ver cambios en desarrollo en dispositivos físicos.
* Utilizar el conocimiento que tienes de React en React Native.
* Desplegar en las App stores.

**WINDOWS: INSTALACIONES NECESARIAS**

Ir a la documentación en:

<https://reactnative.dev/docs/environment-setup?guide=native>

Luego clickear en **Chocolatey** y en la powershell correr los comandos:

* Get-ExecutionPolicy.
* Si retorna **Restricted**, correr el comando:
* **Set-ExecutionPolicy AllSigned**
* Si no dice Restricted, ejecutamos el comando:
* **Set-ExecutionPolicy Bypass -Scope Process**

Una vez finalizado esos comandos, copiamos y pegamos la siguiente línea:

Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))

MIRAR LA DOCUMENTACIÓN ACÁ POR LAS DUDAS: <https://chocolatey.org/install>

Una vez instalado todo, cerramos la powershell y la volvemos a abrir, y pegamos el comando que aparece en la documentación de React Native:

**choco install -y nodejs-lts microsoft-openjdk11**

**……. NO CONTINUE PORQUE NO ME ANDABA BIEN EN WINDOWS**

**MAC OSX: INSTALACIONES NECESARIAS – ANDROID**

Vamos a la documentación oficial en: <https://reactnative.dev/docs/environment-setup?guide=native&platform=android>

Seleccionamos en Development **MacOs** y en **Target** Android.

Luego si no tenemos instalado **Homebrew**, vamos a <https://brew.sh/> y copiamos y pegamos el comando que nos aparece en la consola.

Por ejemplo este:

/bin/bash -c "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh>)"

Una vez terminado la instalación, verificamos que con el comando **brew –version** nos de una respuesta afirmativa, y seguimos.

Ahora corremos el comando **brew install watchman**.

Y una vez finalizado eso corremos los comandos:

* brew tap homebrew/cask-versions
* brew install --cask zulu11
* brew info --cask zulu11

Luego ya debemos abrir **Android Studio** e ir a **SDK Manager** y verificar en la sección SDK Platforms si tenemos descargado (en el video utiliza el Android 10) Android 13.

Luego clickear en **Show package details** (esta debajo a la derecha) y verificar que tengamos descargado **Android SDK platform 33** y **Google APIs ARM 64 v8a System Image**.

Ahora debemos configurar variables de entorno, por lo que abrimos el **Finder** y clickeamos en nuestro usuario. A continuación, apretamos **command + shift + .** para abrir los archivos ocultos y buscamos el archivo **zshrc** o el archivo **zprofile**, y copiamos las siguientes líneas de código:

export ANDROID\_HOME="$HOME/Library/Android/sdk"

export PATH="$PATH:$ANDROID\_HOME/emulator"

export PATH="$PATH:$ANDROID\_HOME/platform-tools"

**MAC OSX: EMULADOR DE ANDROID**

Vamos a **Android Studio** y clickeamos en **Virtual Device Manager>Create device**.

Creamos un device a nuestro placer o elegimos algunos de los que están por defecto.

Y luego ya está, le damos next hasta que abra el emulador.

**INSTALACIONES NECESARIAS PARA IOS**

Vamos a la documentación oficial de React Native:

<https://reactnative.dev/docs/environment-setup?platform=ios>

Y elegimos Macos e IOs, luego instalar node y watchman, como dice la documentación.

A continuación, abrir XCode y vamos en el menú de arriba de todo en la mac a **settings** del XCode.

Luego seleccionamos **Locations** y verificar que command line tools corresponda la versión con la de xcode.

Ahora debemos instalar cocoapods con el comando: **brew install cocoapods**

**CREAR PROYECTO DE REACT NATIVE**

Vamos a crear el proyecto con el comando:

**npx react-native init MiProyecto --template react-native-template-typescript**

Una vez terminado la descarga abrir el emulador de Android y correr el comando dentro de la carpeta del proyecto:

**npx react-native run-android**

Este comando nos va abrir otra terminal la cual se llama **Metro**, si presionamos la tecla **r**, reinicia la aplicación.

Luego abrimos la carpeta del proyecto en VSC y en el **App.tsx** borramos todo y creamos el siguiente componente:

import React from 'react'

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

const App = () => {

return (

<View>

<Text>Hola Mundo</Text>

</View>

)

}

export default App;

**CORRER SIMULADOR DE IOS**

Dentro de la carpeta del Proyecto corremos el comando: **npx react-native run-ios**

Además si queremos correr el proyecto con un emulador determinado agregamos al comando de arriba las siguientes opciones:

* --simulator=”iPhone 8”
* --simulator=”iPhone 8 Plus”
* --simulator=”iPhone 11”
* --simulator=”iPhone 11 Pro”
* --simulator=”iPhone 11 Pro Max”
* --simulator=”iPhone 12”
* --simulator=”iPhone 12 Pro”
* --simulator=”iPhone 12 Pro Max”
* Etc.

El comando para ver la listas de dispositivos es:

**xcrun simctl list devices**

**SEC-4: MI PRIMER APP EN REACT NATIVE – COUNTER APP**

**DESHABILITAR PRETTIER**

Nos dirigimos al archivo **.eslintrc.js** y agregamos la siguiente línea en el **module.exports:**

rules: {

'prettier/prettier': 0

}

**HOLA MUNDO**

En el **App.tsx** borrar todo y escribir lo siguiente:

import React from 'react';

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

const App = () => {

return (

<View

style={{

flex: 1,

justifyContent: 'center',

}}>

<Text style={{

fontSize: 45,

textAlign: 'center'

}}>

Hola Mundo

</Text>

</View>

);

};

export default App;

En React Native no temenos un <div> o un <p>, pero tenemos los componentes **<View>** y **<Text>**.

**CREAR PANTALLAS INDEPENDIENTES**

En la raíz del proyecto creamos la carpeta **src** y dentro de la misma, las carpeta **screens** y **components**.

Ahora dentro de la carpeta **screens** creamos la pantalla **HolaMundoScreen.tsx**, la cual va a contener todo el código que estaba en el **App.tsx**, y luego importamos esta screen en el App.tsx.

**CREAR UN CONTADOR**

En la carpeta **screens** creamos el archivo **ContadorScreen.tsx**, e implementamos la lógica para mostrar dos botones que aumenten o disminuyan el contador:

import React, {useState} from 'react';

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

export const ContadorScreen = () => {

const [counter, setCounter] = useState(0);

return (

<View style={{flex: 1, justifyContent: 'center'}}>

<Text

style={{

textAlign: 'center',

fontSize: 40,

}}>

Contador: {counter}

</Text>

<Button title="Aumentar" onPress={() => setCounter(counter + 1)} />

<Button title="Decrementar" onPress={() => setCounter(counter - 1)} />

</View>

);

};

Acá está la documentación de los componentes que podemos usar de React-Native:

<https://reactnative.dev/docs/components-and-apis>

**TOUCHABLE OPACITY**

La documentación recomienda no usar tanto el componente Button, en cambio, debemos usar el **TouchableOpacity**:

import React, {useState} from 'react';

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

export const ContadorScreen = () => {

const [counter, setCounter] = useState(0);

return (

<View style={{flex: 1, justifyContent: 'center'}}>

<Text

style={{

textAlign: 'center',

fontSize: 40,

}}>

Contador: {counter}

</Text>

<TouchableOpacity onPress={() => setCounter(counter + 1)}>

<View>

<Text>+1</Text>

</View>

</TouchableOpacity>

</View>

);

};

**STYLE-SHEET**

En React Native se acostumbra a importar **StyleSheet** y crear un objeto el cual va a contener los estilos de los componentes que utilicemos:

import React, {useState} from 'react';

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

export const ContadorScreen = () => {

const [counter, setCounter] = useState(0);

return (

<View style={styles.container}>

<Text style={styles.title}>Contador: {counter}</Text>

<TouchableOpacity onPress={() => setCounter(counter + 1)}>

<View>

<Text>+1</Text>

</View>

</TouchableOpacity>

</View>

);

};

const styles = StyleSheet.create({

container: {

flex: 1,

justifyContent: 'center',

},

title: {

textAlign: 'center',

fontSize: 40,

},

});

**BUTTON PERSONALIZADO FLOTANTE**

Vamos hacer que los TouchableOpacity queden debajo de todo, uno a la derecha y el otro a la izquierda:

import React, {useState} from 'react';

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

export const ContadorScreen = () => {

const [counter, setCounter] = useState(0);

return (

<View style={styles.container}>

<Text style={styles.title}>Contador: {counter}</Text>

<TouchableOpacity

style={styles.fabLocationRight}

onPress={() => setCounter(counter + 1)}>

<View style={styles.fab}>

<Text style={styles.fabText}>+1</Text>

</View>

</TouchableOpacity>

<TouchableOpacity

style={styles.fabLocationLeft}

onPress={() => setCounter(counter - 1)}>

<View style={styles.fab}>

<Text style={styles.fabText}>-1</Text>

</View>

</TouchableOpacity>

</View>

);

};

const styles = StyleSheet.create({

container: {

flex: 1,

justifyContent: 'center',

},

title: {

textAlign: 'center',

fontSize: 40,

},

fabLocationRight: {

position: 'absolute',

bottom: 25,

right: 25,

},

fabLocationLeft: {

position: 'absolute',

bottom: 25,

left: 25,

},

fab: {

backgroundColor: '#5856D6',

width: 60,

height: 60,

borderRadius: 100,

justifyContent: 'center',

},

fabText: {

color: 'white',

fontSize: 25,

fontWeight: 'bold',

alignSelf: 'center',

},

});

**COMPONENTE PERSONALIZADO: FAB – FLOATING ACTION BUTTON**

En la carpeta **components** creamos el archivo **Fab.tsx**, el cual va a ser nuestro botón reutilizable:

import React from 'react';

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

interface Props {

title: string;

}

export const Fab = ({title}: Props) => {

return (

<TouchableOpacity

style={styles.fabLocationRight}

onPress={() => console.log('Click')}>

<View style={styles.fab}>

<Text style={styles.fabText}>{title}</Text>

</View>

</TouchableOpacity>

);

};

const styles = StyleSheet.create({

fabLocationRight: {

position: 'absolute',

bottom: 25,

right: 25,

},

fabLocationLeft: {

position: 'absolute',

bottom: 25,

left: 25,

},

fab: {

backgroundColor: '#5856D6',

width: 60,

height: 60,

borderRadius: 100,

justifyContent: 'center',

},

fabText: {

color: 'white',

fontSize: 25,

fontWeight: 'bold',

alignSelf: 'center',

},

});

Como vemos estamos diciendole que va a recibir **props** y además le decimos que tipado tienen esas Props.

En el **screen** ContadorScreen.tsx importamos este componente y le pasamos la prop title:

…

<Fab title='+1' />

…

**ESTILO CONDICIONAL**

Debemos hacer ahora que un botón este a la derecha y el otro la izquierda, por lo que le mandamos en Props la propiedad **position**, y le especificamos en la interface que puede ser **‘bl’** o **‘br’**, y que por defecto es br.

Además ya no usamos el componente TouchableOpacity, en vez de eso usamos **TouchableNativeFeedback**, el cual tiene una propiedad **background** la cual produce el efecto de un shadow cuando clickeamos en el botón (por ahora solo nos va a servir en Android, ver el próximo video para ver cómo se soluciona en IOS).

**Fab.tsx:**

import React from 'react';

import {

StyleSheet,

Text,

TouchableNativeFeedback,

TouchableOpacity,

View,

} from 'react-native';

interface Props {

title: string;

position?: 'br' | 'bl';

onPress: () => void;

}

export const Fab = ({title, onPress, position = 'br'}: Props) => {

return (

<View

style={[

styles.fabLocation,

position === 'br' ? styles.right : styles.left,

]}>

<TouchableNativeFeedback

onPress={onPress}

background={TouchableNativeFeedback.Ripple('#28425B', false, 30)}>

<View style={styles.fab}>

<Text style={styles.fabText}>{title}</Text>

</View>

</TouchableNativeFeedback>

</View>

);

};

const styles = StyleSheet.create({

fabLocation: {

position: 'absolute',

bottom: 25,

},

right: {

right: 25,

},

left: {

left: 25,

},

fab: {

backgroundColor: '#5856D6',

width: 60,

height: 60,

borderRadius: 100,

justifyContent: 'center',

shadowColor: '#000',

shadowOffset: {

width: 0,

height: 4,

},

shadowOpacity: 0.3,

shadowRadius: 4.65,

elevation: 8,

},

fabText: {

color: 'white',

fontSize: 25,

fontWeight: 'bold',

alignSelf: 'center',

},

});

**ContadorScreen.tsx:**

...

<Fab title="+1" onPress={() => setCounter(counter + 1)} />

<Fab title="-1" position="bl" onPress={() => setCounter(counter - 1)} />

...

**CODIGO ESPECIFICO PARA PLATAFORMA**

En el componente **Fab.tsx** importamos **Platform** de react-native y creamos dos funciones, una **ios** y otra **android**, las cuales van a retornar un JSX.Element:

import React from 'react';

import {

StyleSheet,

Text,

TouchableNativeFeedback,

TouchableOpacity,

View,

Platform,

} from 'react-native';

interface Props {

title: string;

position?: 'br' | 'bl';

onPress: () => void;

}

export const Fab = ({title, onPress, position = 'br'}: Props) => {

const ios = () => {

return (

<TouchableOpacity

activeOpacity={0.75}

onPress={onPress}

style={[

styles.fabLocation,

position === 'br' ? styles.right : styles.left,

]}>

<View style={styles.fab}>

<Text style={styles.fabText}>{title}</Text>

</View>

</TouchableOpacity>

);

};

const android = () => {

return (

<View

style={[

styles.fabLocation,

position === 'br' ? styles.right : styles.left,

]}>

<TouchableNativeFeedback

onPress={onPress}

background={TouchableNativeFeedback.Ripple('#28425B', false, 30)}>

<View style={styles.fab}>

<Text style={styles.fabText}>{title}</Text>

</View>

</TouchableNativeFeedback>

</View>

);

};

return Platform.OS === 'ios' ? ios() : android();

};

...

De esta forma en Android tendremos ciertos estilos y en ios otros.