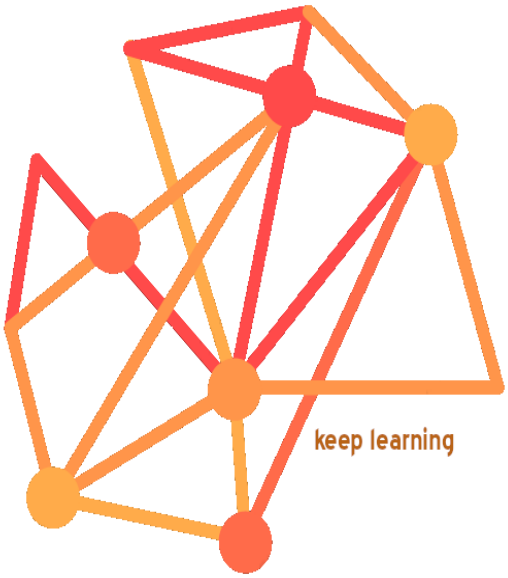
# React Native




Juan Carlos Pérez Rodríguez

# Sumario

# Introducción

React Native es un framework UI (interfaz gráfica ) JavaScript open-source creado por Meta ( Facebook ) para crear aplicaciones reales nativas para iOS y Android, basado en la librería de JavaScript React para la creación de componentes visuales.

En lugar de orientarse a la web, como React accede a plataformas móviles nativas, en este caso iOS y Android. A diferencia de otras herramientas **no** es un webview ( el renderizado de una web ) como una aplicación ios/android. Los componentes se renderizan a objetos UI nativos de android/ios. Es decir, en lugar de desarrollar una aplicación web híbrida o en HTML5, lo que se obtiene es una aplicación con UI nativa, indistinguible gráficamente de la que se puede desarrollar en Objective-C o Java/Kotlin.

React Native usa los mismos componentes visuales con los que interactúa el usuario, que las aplicaciones nativas reales de Android e iOS, pero gestiona la interacción de estos componentes utilizando las capacidades de JavaScript/Typescript y React. A nivel de rendimiento será menor que una app nativa, pero tiene un nivel de rendimiento alto, y superior a plataformas basadas en web híbrida

# Instalaciones necesarias y opcionales

- Android Studio
- Node
- Extensiones de vscode:
  - material icon theme
  - bracket pair colorized 2
  - ES7+ React/Redux/React-Native snippets
  - Simple React Snippets
  - Auto Close Tag
  - Paste JSON as Code
  - TypeScript importer
  - TypeScript React code snippets

La página de react native especifica como hacer la instalación:

https://reactnative.dev/docs/environment-setup

Para linux puede valer ( lo mejor es validar en el enlace anterior que cosas son precisas)
- instalar jdk 17 ( que quede establecido JAVA_HOME a la ruta del jdk 17 )
- Android Studio: Tener en cuenta al instalar el SDK que incluya: Android 15 (VanillaIceCream) y que esté marcado:
  Android SDK Platform 35
  Intel x86 Atom_64 System Image o Google APIs Intel x86 Atom System Image

Se debe establecer las variables de entorno ( por ejemplo en: ~/.profile ) ( Cuidado con la ubicación de ANDROID_HOME. Poner la variable a donde haya quedado instalado el Sdk

```
export ANDROID_HOME=$HOME/Android/Sdk
export PATH=$PATH:$ANDROID_HOME/tools
export PATH=$PATH:$ANDROID_HOME/tools/bin
export PATH=$PATH:$ANDROID_HOME/emulator
export PATH=$PATH:$ANDROID_HOME/platform-tools
```

**Nota**: React Native es multiplataforma, sin embargo las instalaciones de este dossier se centrarán en Android. Hay algunas especificidades de IOS que normalmente nos saltaremos. Adicionalmente es habitual tener en nuestro código cosas como: `Platform.OS === 'ios'` que permiten ejecutar código condicional a la plataforma, que se dejarán para autoformación

# Primeros pasos

Una alternativa muy interesante es usar Expo ( no precisas de la instalación y configuración de tanto software ) Que la mayor parte de las ocasiones iría muy bien. Pero para evitar las complicaciones del Eject de Expo ( “liberar” de Expo para poder introducir módulos que no están disponibles en Expo ) vamos a evitar el uso de expo cli.

```
npx @react-native-community/cli@latest init NombreApp

cd NombreApp

npm start
```

Para editar el código abrimos la carpeta con Visual Studio Code

**Nota**: En algunas versiones hay que desactivar prettier que pone muchos mensajes de error cuando ponemos espacios innecesarios. Para eso vamos a: .eslintrc.js y agregamos en el: module.exports la siguiente regla: rules: { ‘prettier/prettier’: 0, } **Ojo**! Ver también que el extends sea de: @react-native-community

Nos quedaría:

```
module.exports = {
        root: true,
        extends: '@react-native-community',
        rules: {
                'prettier/prettier': 0,
        },
};
```

**Práctica 1**: Crearemos un: “hola mundo!. Soy nombrealumno”
Para ello vamos a App.tsx. Dejaremos únicamente un componente View ( que es el contenedor habitual en react native ) y pondremos dentro un componente Text ( que es donde escribiremos ) Practica con tus conocimientos de CSS
Tener en cuenta que la mayoría son válidos, únicamente cambiando la separación con guiones por camel case, y pasando un objeto JSON a style. Así por ejemplo, si queremos hacer que el view ocupe todo en color rosa y ubique su contenido en el centro de la pantalla:`<View style={{ backgroundColor: "lightpink", flex: 1, justifyContent: 'center' }}>`

# Componentes básicos y layout

En Android nativo todo va dentro de “Views” en una composición jerárquica. Aquí tenemos algo similar. Nuestro renderizado estará dentro de un componente: <View>

## View, Text, Button

View, Text y Button pertencen al core Estos son componentes básicos que se renderizan a las UI nativas

Según la documentación oficial: **View** es el componente fundamental para construir interfaces de usuario. View es un contenedor que soporta hacer layouts con flexbox, estilos con etiqueta style, manejador de eventos y controles de accesibilidad. React Native transforma el componente View en el correspondiente según la plataforma: android.view ( para android) ,  (para web), etc

**Text** es el componente que se utiliza para visualizar texto. NO podemos poner texto en la aplicación sin ningún contenedor. Muchas veces usaremos Text

**Button** es un componente que se usa con autocierre y tiene muy pocas propiedades: color, title ( para el texto que muestre ), onPress ( para la acción al pulsar ) y accessibilityLabel ( texto para accesibilidad )

```
type Props = {}

function App({}: Props) {
  const [contador, setcontador] = useState(0)
  return (
    <View style={{
      flex: 1,
      borderWidth: 3,
      borderColor: "black",
      backgroundColor: "lightgray",
      margin: 1
    }}>

      <Text> ejercicio básico. Contador: {contador}</Text>
      <Button title='púlsame' onPress={()=>setcontador(contador+1)} />
    </View>
  )
}
export default App;
```

**Práctica 2**: Reproducir el código anterior. Cambia el color del borde, redondea un poco el borde ( busca la propiedad, es muy parecida a CSS ) y pon otro color de fondo. Modifica también para que aparezcan dos botones. Uno será para incrementar ( el botón dirá: "incrementar" ) y otro botón para decrementar.

**Nota**: con los snippets podemos crear un componente react native mediante la macro: **tsrnf**

En el código anterior se ha mezclado tanto los estilos como las pantallas en el componente inicial ( App.tsx )

Vamos a empezar con buenas prácticas. Crearemos una carpeta: src y dentro pondremos una carpeta: **src/screens** y otra **src/components**. En la carpeta screens se pondrán los componentes que representen el layout completo de una pantalla de la aplicación. En components aparecerán los que son más pequeños ( por ejemplo, un botón personalizado, etc )

Por otro lado los estilos los vamos a separar poniéndolos en una constante llamada styles.

Para generar ese tipo de constante usaremos el objeto: **react-native.StyleSheet**

**Nota**: habitualmente tendremos una carpeta: **src/theme** donde pondremos los estilos

Así por ejemplo, para aplicar a una View el estilo de un contenedor circular gris sería:

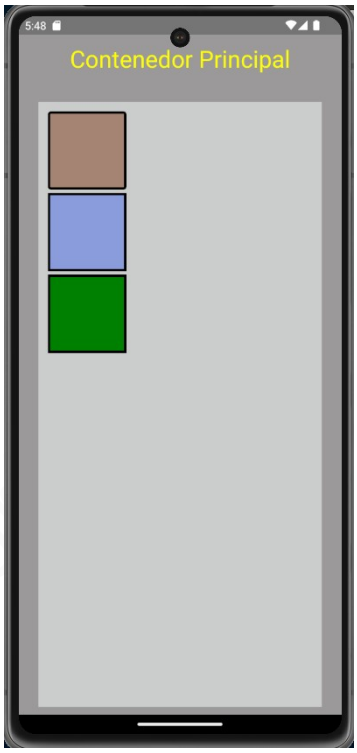
```
function ContenedorCircular(props: Props){
   return(
      <View style={styles.contenedorCircular}>
         <Text>un texto</Text>
      </View>
   );
}

const styles = StyleSheet.create({
  contenedorCircular: {
    borderRadius: 100,
    backgroundColor: "rgba(191, 186, 221, 0.54)",
  },
});
```

Observar que: `StyleSheet.create()` nos permite crear los estilos. Dentro le vamos poniendo los diferentes json para luego pasarlos al atributo style: `<View style={styles.contenedorCircular}>` . Esa será la forma para separar los estilos. Es una buena práctica escribir los estilos en orden alfabético. Así el estilo: botonModal estará antes que: contenedorCircular

**Práctica 3**: Crear la estructura de carpetas que hemos nombrado y un componente: InicioScreen que será el que contenga la pantalla con los botones y el contador. En App.tsx se hará la llamada a: <InicioScreen /> Los estilos deberán separarse usando StyleSheet.create()

**Nota**: con snippets creamos un componente con estilos mediante la macro: **tsrnfs**

# Layout con Flexbox




React native usa para sus layouts el estilo de cajas de Flexbox. Si tenemos práctica usando flex en CSS veremos que es muy similar. Hay pequeñas diferencias, como por ejemplo, por defecto flex-direction es en columnas en lugar de filas. También veremos que NO hay que especificar el equivalente a un display: flex  Cada contenedor tiene ya el comportamiento flexbox para con sus hijos. Por otro lado la sentencia: flex  que en CSS permite pasarle varios parámetros, aquí únicamente permite un número

Vamos a ver un poco su funcionamiento, practicando:

Este renderizado, corresponde a una pantalla inicial ( componente PrincipalScreen.tsx ) donde hay un <Text>Contenedor Principal</Text> y luego un view que hará de "escenario" para nuestros objetos flex.  Dentro de ese view, habrá 3 componentes: Caja  esos componentes tienen igual width que height y reciben mediante props el color que van a renderizar. Por defecto ( si no se pasa por props ) el fondo de la caja es verde. Observar el siguiente estilo:

```
<View style={
  {
    ...styles.caja,
    backgroundColor: bg,
  }
}>
```

En el anterior código, vemos que desestructuramos el json: styles.caja y luego "reemplazamos" la propiedad backgroundColor, estableciéndola a lo que está almacenado en la variable: bg. Observar que esa sería una forma para tener el estilo CSS establecido para: Caja.tsx y modificar el color de fondo con lo recibido por props

**Práctica 4**:  Realizar la pantalla anterior siguiendo lo descrito ( poner un width y un height de 100 en el style del componente Caja.tsx ) . Para que el View "escenario" ocupe el total, poner en sus estilos la propiedad: flex: 1

¿ Cómo funciona la propiedad flex ? Si ponemos: flex: 1 le estamos diciendo que queremos que el objeto coja toda el espacio disponible a una velocidad: 1. Eso implica que si ningún otro objeto trata de tomar todo el espacio disponible lo hará este objeto. Si dos objetos tienen puesto flex: 1 implicará que se reparten el espacio disponible entre los dos. Si el componente A tiene: flex: 1 y el componente B tiene: flex: 2 entonces el componente B va el doble de rápido que A al repartirse el espacio ( se parece al atributo weight en android nativo )

Si está puesto: flex: 0 entonces no tiene efecto flex y se queda con el espacio que le correspondería de forma automática ( si hay puesto un height se queda con ese tamaño )

**Práctica 5**: Si en el componente: Caja.tsx ponemos 2 botones, uno para incrementar el valor de flex y otro para decrementarlo, podremos ir viendo el efecto de modificar dicho atributo en cada una de las tres cajas. Agregar los dos botones y que el dato que se muestra: flex: 0 se actualice según las pulsaciones de esos botones, implicando cambios en el atributo flex correspondiente


Al hacer el anterior ejercicio, podemos observar que tenemos muy limitados los estilos que podemos aplicarle a: <Button> ( de hecho no admite: style lo que se puede es cambiar el color ) Para poder personalizar nuestros botones, y en general para poder poner manejador de evento: Touch, tenemos que usar nuestros propios objetos view envueltos por un: Touchable, ya sea: TouchableHighlight o TouchableOpacity

# TouchableOpacity /Highlight

Estos wrapper ( envoltorios ) de views/text están pensados para poder poner un listener onPress a cualquier view que contenga y establecerle estilos personalizados

La diferencia entre uno y otro es que TouchableOpacity cambia la opacidad del objeto cuando lo mantiene pulsado, haciéndolo más transparente, mientras que TouchableHighlight lo destaca más

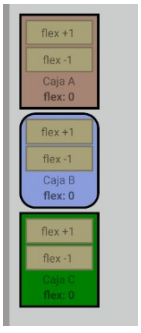
Ej.

```
<TouchableHighlight onPress={()=>Alert.alert("pulsado") }>
  <View style={styles.button}>
    <Text>Púlsame! </Text>
  </View>
</TouchableHighlight>
```

En el ejemplo anterior, se muestra una alerta ( un popup ) con el mensaje: “pulsado” Eso lo conseguimos mediante el objeto Alert: Alert.alert(“mensaje”)

**Práctica 6**: Sustituir los botones por un objeto TouchableHighlight para el botón: flex+1 que cuando se puse abra una alerta e incremente la propiedad flex, y un objeto TouchableOpacity para el botón: flex-1 que decremente la propiedad flex
Personalizar los estilos de esos dos objetos


Observar el siguiente renderizado:




Lo que se pretende es enviar por props el nombre de la Caja ( por eso aparece Caja A, Caja B, Caja C ) y poder enviar por props también estilos personalizados para cada una de las cajas. Para conseguir que se reemplace un estilo que envías por props al que está previamente definido se propone:

```
const styles = StyleSheet.create({
        porDefecto: {
                padding: 4,
                borderWidth: 3,
                margin: 1
        },
}
```

Cuando llamamos al componente: `<Caja style={{borderRadius: 20, borderWidth: 7 } }/>`

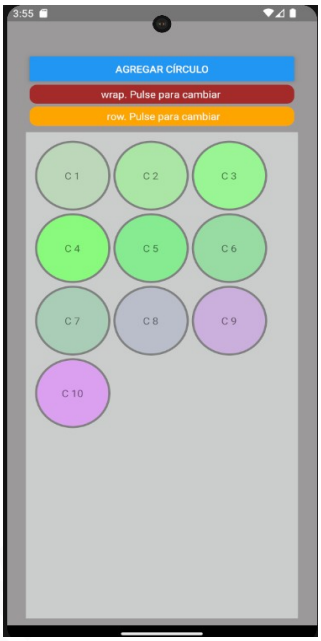
Y ahora en la declaración del componente Caja.tsx algo así:

```
<View style={
        {
        ...styles.porDefecto,
        ...props.style
        }
}>
```

De esa forma primero se establecen los estilos por defecto, reemplazando y agregando con los que se pasen mediante props

**Práctica 7**: Obtener el renderizado anterior, consiguiendo enviar mediante props, tanto el nombre de la Caja.tsx como estilos personalizados que sobreescriban/agreguen a los que estén por defecto (observar que Caja B tiene bordes redondeados )

Ahora vamos a trabajar con la propiedad flexWrap: wrap, nowrap



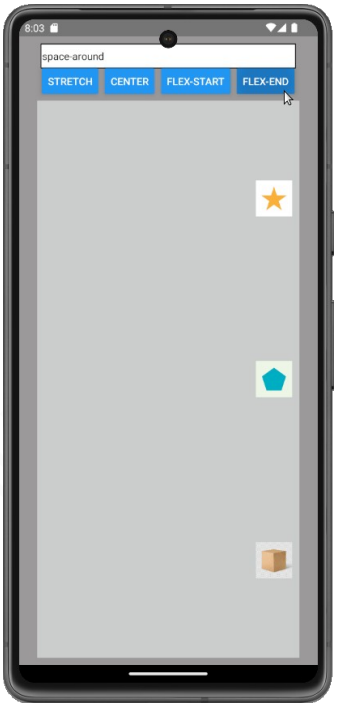

Esa propiedad flexWrap, lo que hace es que si los objetos no caben, saltan a la siguiente fila o columna ( según lo que esté establecido en flexDirection ) Por defecto está establecido a: nowrap y por lo tanto los objetos no saltan de fila/columna sino que hacen un desbordamiento del layout.

La propiedad flexDirection toma los valores: column, row, column-reverse, row-reverse Y permite que los elementos se distribuyan de arriba abajo ( al revés si reverse ) o de izquierda a derecha ( al revés si reverse )

**Práctica 8**: Obtener el renderizado anterior, empezando por no tener ningún círculo. A medida que se pulsa el botón: Agregar Círculo, irán apareciendo. Observar que los colores van variando progresivamente. Se empieza con un: rgb(red,green,blue) y según el número de id que corresponda va variando un poco más ( tener en cuenta que los colores van de 0 a 255 no se puede superar ) Los dos botones cambian entre: wrap /nowrap y entre: column/row. El renderizado que se muestra tiene: flexDirection: 'row', flexWrap: 'wrap'. Observar que dos de los botones tienen estilos, así que no usaremos Button con ellos. Se propone TouchableOpacity

# Image, TextInput, justifyContent, alignItems, SaveAreaView

Vamos a ver el renderizado que queremos alcanzar primero:




Primero aparece un TextInput para introducir el alineado que queremos para justifyContent: space-around, center, etc

La propiedad j**ustifyContent dice como distribuimos los elementos en la dirección definida por flexDirection**. En este caso, como estamos en la situación de flexDirection: column ocurre que justifyContent establece a qué altura se ubican los elementos en la pantalla. En la captura está el caso de: “space-around” y se puede ver que hay la misma distancia entre la estrella y el pentágono, que entre el pentágono y el cubo. Si dividiéramos el espacio total en tres alturas generando así tres regiones, cada figura queda en el centro de su región

**TextInput** es un componente del core de react-native y nos permite introducir texto. Admite varios atributos, entre otros: **placeholder** ( dando una ayuda respecto a qué escribir en el input ), y **value**, que almacena el texto dentro del input. Permite personalizar los estilos con atributo **style** y dispone del listener: **onChangeText** que dispara una acción cada vez que se modifica el texto del input.

En el renderizado tenemos un TextInput que espera recibir un texto que se valida con el evento onChangeText y si coincide con alguno de los valores admitidos por justifyContent entonces se establece

Antes dijimos que el renderizado de la pantalla se ha hecho con el flexDirection por defecto: column, pero observamos que justo debajo del TextInput aparecen 4 botones distribuidos en horizontal. ¿ Cómo hacer eso ? Observar el siguiente código:

```
<View style={{flexDirection: 'column'}}>
        <Text> Hola mundo! </Text>
        <TextInput />
        <View style={{flexDirection: 'row'}}>
                <Text> Uno </Text>
                <Text> Dos </Text>
                <Text> Tres </Text>
        </View>
</View>
```

Los textos: Uno, Dos, Tres se mostrarán distribuidos en horizontal, ya que su contenedor tiene esa dirección dada por  flexDirection: “row”

Por último, vemos que se muestran tres imágenes que corresponden con las url:

https://img.freepik.com/vector-premium/logotipo-estrella-simple-moderno_535345-2471.jpg?w=740

https://img.freepik.com/vector-gratis/vector-forma-geometrica-pentagono-azul_53876-175075.jpg?w=740&t=st=1692203592~exp=1692204192~hmac=b6f63c07a79e2ff41719578b178e004f851718b3b467bf1976878d8b800b9201

https://img.freepik.com/psd-gratis/caja-carton-aislada_125540-1169.jpg?w=1060&t=st=1692203785~exp=1692204385~hmac=ce41256e62eb2af8e8e813b44cbfb4bed697c698b2902ffdb0cb8f502df86d1d

React native tiene otro componente en su core que permite mostrar las imágenes, llamado: <**Image**>  Los atributos que soporta, son: **source** ( equivalente a src de la imágenes html ), **style**

Veamos un ejemplo que tienen en la documentación oficial:

```
<Image
  source={{
    uri: 'https://reactjs.org/logo-og.png',
    method: 'POST',
    headers: {
      Pragma: 'no-cache',
    },
    body: 'Your Body goes here',
  }}
  style={{width: 400, height: 400}}
/>
```

Observamos que el atributo source recibe un json muy completo, que permite hacer una consulta personalizada http. De esos atributos, habitualmente únicamente usaremos: **uri** para especificar la ruta

También podemos cargar una imagen que tengamos guardada en nuestro proyecto. Tener cuidado, la documentación oficial avisa respecto a problemas al usar rutas guardadas en variables para cargar la imagen:

```
// GOOD
<Image source={require('./my-icon.png')} />;

// BAD
const icon = this.props.active
  ? 'my-icon-active'
  : 'my-icon-inactive';
<Image source={require('./' + icon + '.png')} />;

// GOOD
const icon = this.props.active
  ? require('./my-icon-active.png')
  : require('./my-icon-inactive.png');
<Image source={icon} />;
```

Observar que cargamos la imagen mediante: require() y no debemos pasarle la ruta como una variable a require().

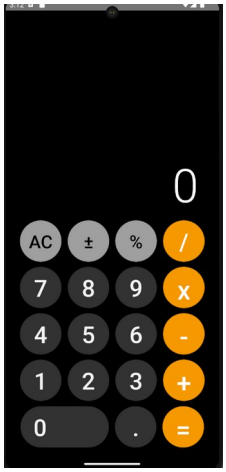
**Práctica 9**: Obtener renderizado parecido al anterior. Las imágenes pueden ser otras. Debe funcionar el onChangeText para cambiar justifyContext y los botones para alignContext. Una de las imágenes se guardará en una carpeta: img y se cargará con: require().
Cuidado! NO poner wrap. Debemos establecer nowrap, o al hacer aignItems se ubicará según el tamaño del objeto más grande de la columna, en lugar de respecto a su contenedor

En las aplicaciones que hemos hecho hasta ahora, podemos notar un comportamiento diferente si hemos tomado un emulador u otro. No es lo mismo un emulador con “notch” que los que no lo tienen. Cuando estamos con una imagen se suele querer aprovechar todo el espacio disponible ( inclusive el notch ) pero si es para cuestiones de texto se prefiere que aparezca escrito debajo de ese elemento de la pantalla. Para conseguir la separación del notch tenemos un componente específico: <**SafeAreaView**> (para IOS >= 11 )

Cuidado al usarlo!, al ser un componente contenedor debe tomar toda la pantalla:

```
function HomeScreen() {
return (
        <SafeAreaView style={{flex: 1}}>
                <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
                        <Text>Home Screen </Text>
                </View>
        </SafeAreaView>
);
}
```

Hay más características asociadas a Flex que pueden verse en la página oficial de react native. Pero en general, si se tienen los conocimientos de Flex en CSS ya lo tenemos. Nombrar únicamente que existe: **alignContent** para hacer parecido a alignItems pero cuando se ha establecido wrap en lugar de nowrap. Las separaciones “gap” que podemos poner entre elementos. Los gap son: **rowGap** ( separación entre filas ), **columnGap** ( separación entre columnas ), **gap** ( tanto para columnas como filas ) Su funcionamiento es similar al del gap de grid en CSS. Y finalmente el atributo: **position** que es parecido al position de CSS con la característica que aquí únicamente tenemos: relative, absolute Donde por defecto no se está en static ( no existe ) sino que todos los elementos vienen por defecto en **postion: ‘relative’**




El renderizado de esta calculadora se puede hacer de varias formas. Se propone “empujar” todos los objetos con flex-end, hasta el final de la pantalla. Luego hacer 5 views para poner los botones de cada fila en su correspondiente view. Observar que los botones pueden realizarse como un componente con TouchableOpacity y pasarles propiedades para personalizarlo. Podemos hacer que el texto se vaya ajustando al tamaño del número introducido por teclado con:

**adjustsfontSizeToFit:**

```
<Text adjustsFontSizeToFit={true} numberOfLines={1} > texto </Text>
```

**Práctica 10**: hacer la calculadora descrita

# Estructura de carpetas

Se propone la siguiente estructura de carpetas:

```
src
    api
    components
    screens
    hooks
    navigation
    components
    themes
    context
```

En general, para la separación de nuestra Vista de su código asociado, usaremos hooks personalizados

> **Práctica 11**: hacer un hook personalizado para la calculadora que tenga todo el código que está fuera del return del componente

Los estilos estarán en themes, los componentes más pequeños que una pantalla en components. Las pantallas en screens, en navigation pondremos las diferentes navegaciones entre pantallas . En api serán las llamadas a las apis

# Formularios, Switch, Alert/Prompt

## Alert y Prompts

Ya hemos usado Alert, pero no hemos visto todas las opciones que tiene. Es habitual que cuando se muestra una alerta el usuario pueda “aceptar” o “rechazar”. Incluso podemos poner más botones en la alerta, darle estilos ( atributo style ) etc

```
const [colorpreferido, setcolorpreferido] = useState("green");

function mostrarAlerta(){
  Alert.alert('Cambio de color', 'Si acepta cambiará el color a rojo', [
    //El array representa a cada botón.
    {
      text: 'Cancel', //texto mostrado
      onPress: () => console.log('No se cambia el color'),
      //style: 'cancel',
    },
    {text: 'OK', onPress: () => setcolorpreferido("rojo")},
  ]);
}

return (
  <View>
    <Button title="cambiar a rojo" onPress={mostrarAlerta} />
    <Text>{colorpreferido}</Text>
```

**Práctica 12**: Reproducir el ejemplo anterior, generando una pantalla específica, pero le haremos algunas diferencias. El botón dirá: “cambiar de color” entonces se genera un color aleatorio a mostrar ( puede ser de un array previamente generado de colores: “gray”, “blue”,... ) y se le muestra una alerta al usuario diciendo cuál es el color que se va a poner de background de la pantalla. Si pulsa ok se cambia el fondo, si pulsa cancel se mantiene el previo

Con lo anterior tenermos un popup con algo de interacción con el usuario. Y, de hecho, las Alert en IOS permiten poner un prompt ( no así en android ) Por ese motivo, vamos a ver un componente de terceros que nos permite hacer un prompt tanto en ios como android

Página: https://www.npmjs.com/package/react-native-prompt-android

```
npm i react-native-prompt-android
```

Ejemplo de uso:

```
return (
  <View style={{}}>

    <Button title="cambiar color" onPress={ ()=>
      prompt(
          'Aquí título',
          'aquí descripción del prompt',
          [
          {text: 'Cancel', onPress: () => console.log('se ha cancelado')},
          {text: 'OK', onPress: password => console.log('OK nueva password: ' + password)},
          ],
          {
              type: 'secure-text', // muestra o no muestra el texto escrito
              cancelable: false, //impide cerrar la ventana pulsando fuera de ella
              //defaultValue: 'test',  //se puede poner valor por defecto
              placeholder: 'password'
          }
      )
    } />
```

**Práctica 13**: Usar el prompt descrito para recrear el ejercicio anterior del color de fondo. Ahora en lugar de una alerta se mostrará un prompt para que el usuario introduzca un color de fondo en lugar de uno aleatorio.
Nota: es posible que de error si se instala con la app ya lanzada. En ese caso parar metro y volver a lanzar

# Switch

Este es otro componente de react-native que se realiza con autocierre y pasando atributos. Los atributos que soporta están relacionados principalmente con los estilos:

```
const [activo, setactivo] = useState(false);

return (
  <View style={{}}>
    <Text>Ejemplo switch:</Text>
    <Switch
      trackColor={{false:"blue", true: "red"}}
      thumbColor={ activo ? "pink":"green"}
      onValueChange={()=>setactivo(!activo)}
      value={activo}

    />
```

**Práctica 14**: Reproducir el ejemplo anterior, poniendo en el pulsador ( la bola ) los colores si falso: brown, si verdadero: orange No aplicar los otros dos colores del ejemplo El switch debe valer para poner un borde a toda la pantalla de color rojo o quitarle el borde

# Formularios

No tenemos un equivalente a form de html ( sí lo hay si nos vamos a componentes de terceros ) Pero lo vamos a reproducir mediante onChange()

Veamos un ejemplo y explicamos:

```
type FormData = {
  jubilado: boolean,
  casado: boolean,
  edad: number,
  nombre: string
}

const IntroducirDatosPersonaScreen = (props: Props) => {
  const [formdata, setformdata] = useState<FormData>({} as FormData);

  function fillFormData(value: boolean|number|string, field: keyof FormData){
    setformdata(
      {
        ...formdata,
        [field]: value
      }
    );
  }
  return (
    <View>
      <Switch
        onValueChange={()=>fillFormData(!formdata.jubilado,"jubilado")}
        value={formdata.jubilado}
      />
      <TextInput placeholder='nombre' onChangeText={(texto)=>fillFormData(texto,"nombre")}/>
      <TextInput placeholder='edad' onChangeText={(texto)=>fillFormData(parseInt(texto),"edad")}/>
      <Text>
        {JSON.stringify(formdata)}
      </Text>
    </View>
  )
}
```

El ejemplo reproduce un posible formulario de datos de una persona ( nombre, edad, casado, jubilado ) Tenemos un switch para establecer is está o no jubilado, un textinput para la string de texto y otro textinput para el number de la edad. A diferencia de un formulario tradicional en html no ocurre que sea al pulsar en el submit el momento en el que se guarden los datos en el state. Observar que ponemos múltiples llamadas al método fillFormData() con: onChangeText(),

onValueChange(). Cada una de estas llamadas se encarga de modificar la parte del state del formulario que le corresponde. Veamos mejor el trozo de código más relevante:

```
function fillFormData(value: boolean|number|string, field: keyof FormData){
        setformdata(
                {
                        ...formdata,
                        [field]: value
                }
        );
}
```

Para llamar a la función: fillFormData(!formdata.jubilado,"jubilado")} Vemos que le pasamos el nombre del campo que queremos modificar: “jubilado” y le decimos cuál es el valor deseado: !formdata.jubilado Analizando la función vemos que al hacer: field: keyof FormData conseguimos obtener en: field el nombre del campo ( **keyof FormData está diciendo que el tipo de la variable field es una clave: keyof del tipo: FormData** ) y luego al hacer: [field]: value le estamos diciendo que a ese campo: field le asignemos el valor deseado ( observar que field lo escribimos entre corchetes: [] )

Finalmente, para delimitar mejor el valor de: value ( siempre podemos poner: value: any como tipo de datos ) aparece: value: boolean|number|string queriendo decir que es uno de esos tres tipos

**Práctica 15**: Reproducir el ejemplo anterior, completando lo que falta (poniendo el switch de: casado )

# Listas

En el core disponemos de: FlatList, SectionList y ScrollView Vamos a prescindir de la última porque ocupa más recursos (renderiza todos los objetos a la vez, quepan o no en la pantalla, con el consiguiente gasto de recursos ). La característica de todas ellas es que nos permiten hacer scroll e ir mostrando todos los objetos en pantalla

# Flatlist

Es un componente muy sencillo. Se le pasa la lista en el atributo: **data** y una función en: **renderItem** para decirle como mostrar el elemento ( igual que en array.map() ). Precisa de un id único que se establece con: **keyExtractor**. Luego como atributos opcionales tenemos entre otros una cabecera: **ListHeaderComponent**, y un separador entre elementos: **ItemSeparatorComponent.** Ej:

```
type Pers={ nombre: string, edad: number}

const array: Pers[] = [
  {nombre: "ana", edad: 21},
  {nombre: "pedro", edad: 12},
  {nombre: "marta", edad: 32},
  {nombre: "lito", edad: 23},
  {nombre: "silvia", edad: 28},
  {nombre: "lisa", edad: 68},
  {nombre: "evaristo", edad: 52},
  {nombre: "arminda", edad: 16},
  {nombre: "felo", edad: 14},
]

return (
  <View style={{flex:1}}>

    <FlatList
            data={array}
            renderItem={(p) => {
              return(
              <View style={{borderWidth: 2, margin: 2}}>
                <Text>{JSON.stringify(p,null,3)}</Text>
              </View>
              )
            }}
            keyExtractor={ (item,index) => item.nombre + index }
            ListHeaderComponent={ () => <Text>Lista Personas</Text>}
            ItemSeparatorComponent={() => <Text> ********** </Text>}

    />
```

**Práctica 16**: Reproducir el ejemplo anterior, reemplazando el array de personas por uno de productos donde se quiere ver el nombre del producto, el precio y el stock. No debe haber separador. Dale un estilo personalizado a los items ( observa que en cada item viene un atributo index ) y muestra un color de fondo para los items impares y otro distinto para los pares. Pon centrado y más grande la cabecera y que diga: Lista Productos

**Práctica 17**: Basándose en el ejemplo de formulario que hemos hecho, vamos a realizar ejercicio de cálculo de imc. Estará el componente: CardPersona.tsx que contendrá el formulario para imc (id, nombre, altura, peso ) y el componente padre que mostrará la lista de cards una debajo de otra. Habrá un botón con title: "+" para crear cada nuevo CardPersona. Habrá en cada card un botón para poder borrar la persona de la lista
Nota: observar que en el padre únicamente se precisa el tamaño del array ( que se obtiene al pulsar el botón y además genera el id que se pasa por props). Así que modificar el state de la card no tiene por qué modificar el state de la lista de cards. Otra cosa es el botón eliminar, que sí debe afectar al state del apadre.

# Iconos

Vamos a usar react-native-vector-icons. Para hacer una instalación actualizada podemos ver la página oficial: https://github.com/oblador/react-native-vector-icons

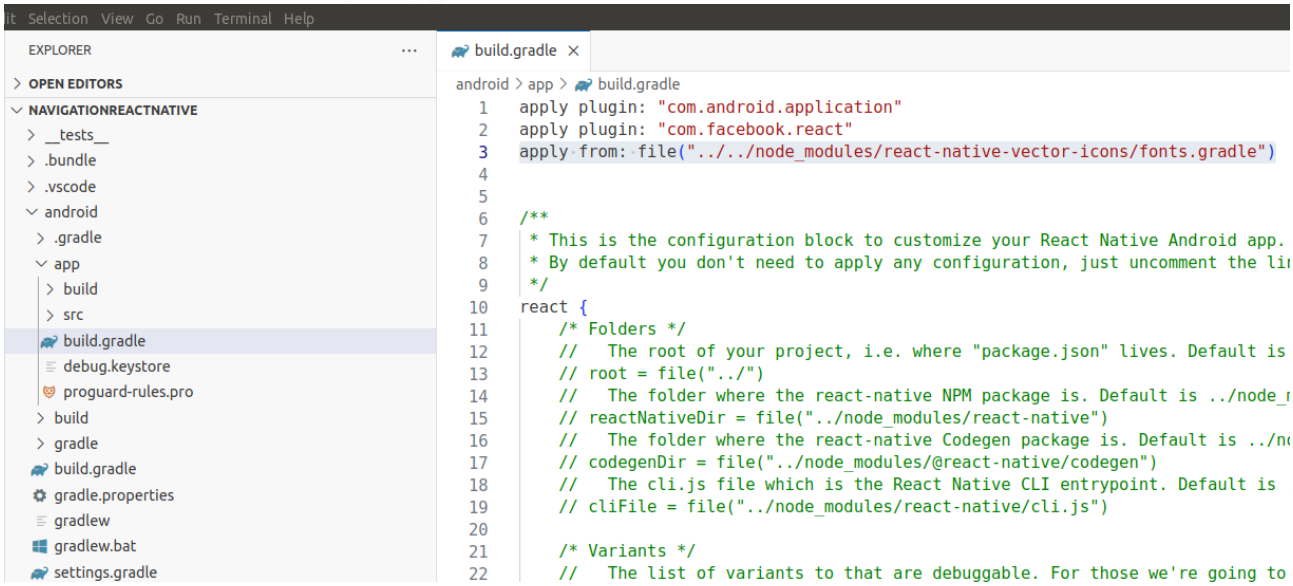
Allí documentan como instalar para IOS y Android, aquí vamos a documentar únicamente android

Para usar iconos vamos a instalar dos paquetes ( el segundo es para evitar errores con typescript):

```
npm install --save-dev react-native-vector-icons
npm install @types/react-native-vector-icons
```

Para instalar todos los iconos ( es demasiado, mejor tomar únicamente los que se precisen ) Agregamos en el fichero: **android/app/build.gradle** ( cuidado!! no confundir con android/build.gradle ) la línea:

```
apply from: file("../../node_modules/react-native-vector-icons/fonts.gradle")
```



Si en lugar de todo el conjunto queremos un subgrupo ( que es lo recomendable ) , agregamos encima de la línea anterior, los que queremos:

```
project.ext.vectoricons = [
        iconFontNames: ['MaterialCommunityIcons.ttf', 'Ionicons.ttf' ]
]

apply from: file("../../node_modules/react-native-vector-icons/fonts.gradle")
```

El anterior ejemplo instala los iconos de ionic y los de material community

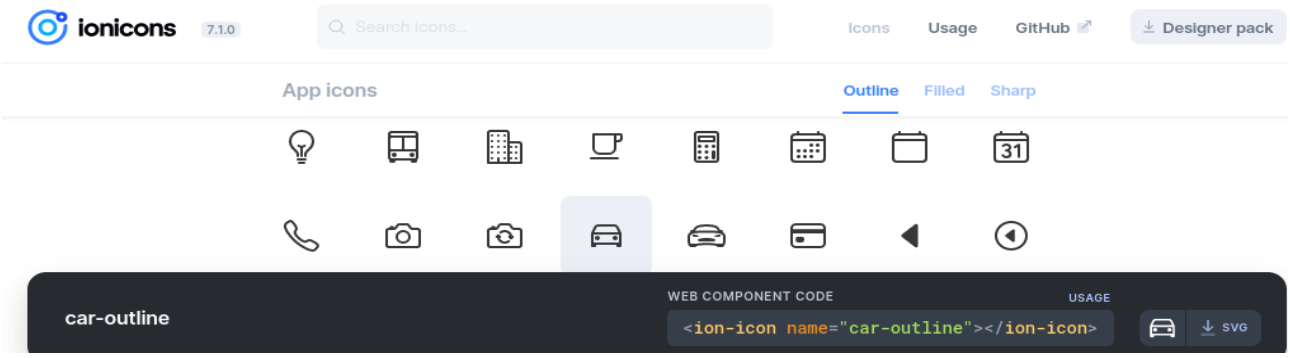
Ahora para usarlos, lo mejor es ver los disponibles desde la propia web ( hay enlaces directos a cada conjunto de iconos )

## Bundled Icon Sets

Browse all.

- `AntDesign` by AntFinance (**298** icons)
- `Entypo` by Daniel Bruce (v1.0.1 **411** icons)
- `EvilIcons` by Alexander Madyankin & Roman Shamin (v1.10.1, **70** icons)
- `Feather` by Cole Bemis & Contributors (v4.28.0, **286** icons)
- `FontAwesome` by Dave Gandy (v4.7.0, **675** icons)
- `FontAwesome 5` by Fonticons, Inc. (v5.15.3, 1598 (free) **7848** (pro) icons)
- `FontAwesome 6` by Fonticons, Inc. (v6.1.2, 2016 (free) **16150** (pro) icons)
- `Fontisto` by Kenan Gündoğan (v3.0.4, **615** icons)
- `Foundation` by ZURB, Inc. (v3.0, **283** icons)
- `Ionicons` by Ionic (v7.1.0, **1338** icons)
- `MaterialIcons` by Google, Inc. (v4.0.0, **2189** icons)
- `MaterialCommunityIcons` by MaterialDesignIcons.com (v6.5.95, **6596** icons)
- `Octicons` by Github, Inc. (v16.3.1, **250** icons)
- `Zocial` by Sam Collins (v1.4.0, **100** icons)
- `SimpleLineIcons` by Sabbir & Contributors (v2.5.5, **189** icons)

Por ejemplo, para usar los de Ionic, ya conocemos el nombre del paquete: Ionicicons y luego entrando en el enlace vemos:



Así, para usar el: “car-outline” haremos el import:

```
import Icon from 'react-native-vector-icons/Ionicons';
```

y luego usaremos el icono en nuestro JSX/TSX así:

```
<Icon name='car-outline' size={50} color={"blue"} />
```

Vemos que podemos establecerle un tamaño: size, y un color al icono: color

**Práctica 18**: Crear una pantalla con un TextiInput un button y un view para hacer una especie de diario.
Cuando el usuario escribe un texto y pulsa en el botón aparece en el view el texto introducido con la fecha ( incluye segundos ) y un smile que describa las emociones escritas en el texto. Así si el usuario ha incluido entre su texto: “:-)” entonces aparecerá una imagen ( icono ) de sonrisa al lado del texto escrito del usuario. Si escribe: “:-(“ lo mismo pero triste

**Práctica 19**: Conversor de iconos smiles a texto. Esta otra pantalla tiene en la parte superior varios iconos ( uno para sonrisa, otro para tristeza, etc ) El usuario va escribiendo en un TextInput un texto y quiere agregar en formato texto el smile. Para que no tenga que conocerlo, basta que en ese momento pulse en el icono correspondiente y le aparecerá agregado al final de su texto, el texto smile correspondiente

# Navegación en React Native

Hay varios tipos de navegación: mediante tabs ( típicamente están en la parte inferior y pulsamos en una u otra para cambiar ). Drawer: que es un menú lateral escondido con las rutas a las que queremos desplazarnos. Y Stack, que es la más habitual, inspirada en la navegación web

En un navegador web, el historial de navegación genera una pila de URLs. Para navegar podemos usar la etiqueta (<a>) en HTML. Cuando se hace clic sobre el enlace, la URL en la que nos encontrábamos se envía a la pila del historial del navegador y navegamos a la página solicitada. Al usar una pila, si se presiona el botón Atrás, se irá al elemento de la parte superior de la pila del historial, así que la página mostrada pasará a ser la página de la que veníamos

Para la instalación del Navigator ( ésta es una fase común para los 3 navigator que hemos nombrado ) lo mejor es seguir las instrucciones de la documentación oficial por si hay actualizaciones: react navigation: https://reactnavigation.org/docs/getting-started/

Pero en resumen es:

Para instalar la navegación:

```
npm install @react-navigation/native
npm install react-native-screens react-native-safe-area-context
```

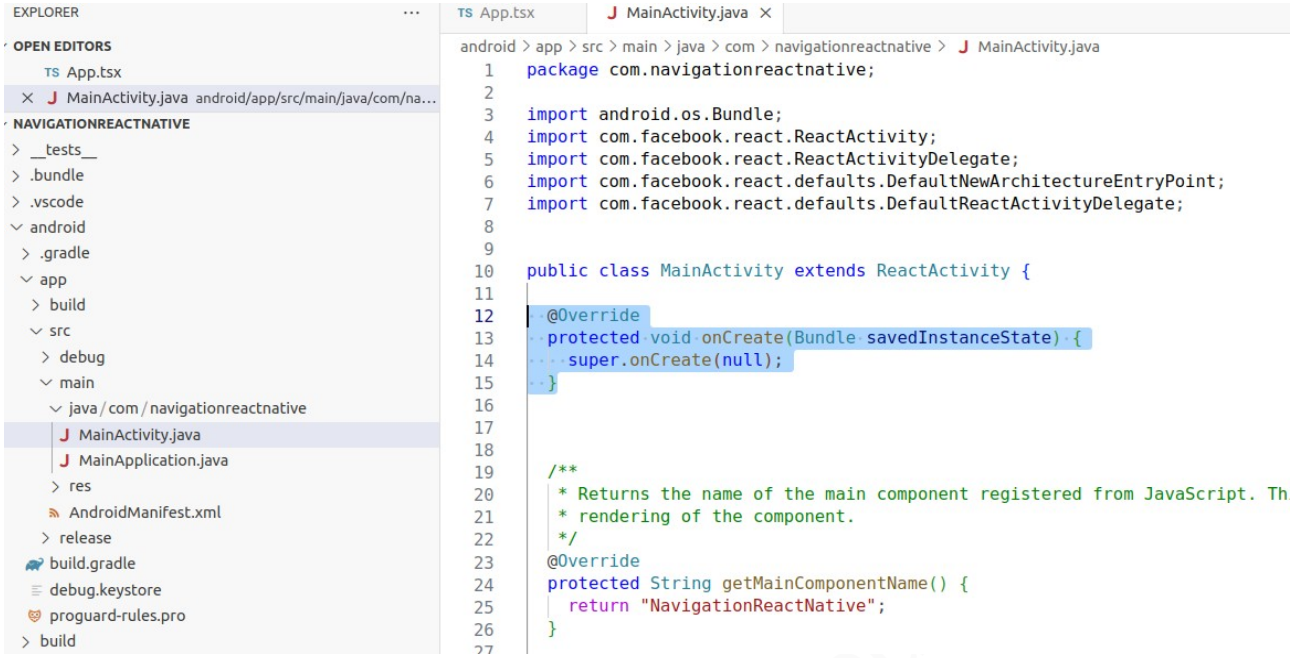
Editamos `android/app/src/main/java/nuestropackage/MainActivity.java`.

Pondremos un import:

```
import android.os.Bundle;
```

Agregamos un onCreate() a MainActivity:

```
public class MainActivity extends ReactActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(null);
  }

}
```



# Stack Navigation

El **Stack Navigator** de React Navigation nos provee de un mecanismo para navegar entre las distintas vistas de nuestra aplicación y gestionar un historial de navegación.

La aplicación con Stack navigator entonces se asemeja a la del navegador web: la aplicación guarda o quita elementos de la pila de navegación, según la interacción del usuario con la aplicación, obteniendo así distintas vistas.

Adicionalmente el stack navigator de React Navigation está provisto de gestos y/o animaciones propias de una app Android/IOS nativa, al desplazarse de una pantalla a otra.

Ahora, para la navegación más habitual ( Stack ) hacemos:

```
npm install @react-navigation/native-stack
```

**NavigationContainer** es un componente que administra nuestro árbol de navegación y contiene el estado de navegación. Este componente debe envolver la estructura de todos los navigators. Por lo general, renderizamos este componente en la raíz de nuestra aplicación.

```
export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
            {/* se ponen todas las screen que queramos el navigation */}
            <Stack.Screen name="unascreen" component{UnaScreen} />
            <Stack.Screen name="otrascreen" component{OtraScreen} />

      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Un ejemplo sencillo basado en la documentación oficial:

App.tsx  MainActivity.java

App.tsx > App

```
18    useColorScheme,
19    View,
20  } from 'react-native';
21
22  import { SafeAreaView } from 'react-native-safe-area-context';
23
24
25  function HomeScreen() {
26    return (
27      <SafeAreaView style={{flex: 1}}>
28        <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
29          <Text>Home Screen </Text>
30        </View>
31      </SafeAreaView>
32    );
33  }
34
35  const Stack = createNativeStackNavigator();
36
37  function App(): JSX.Element {
38
39    return (
40      <NavigationContainer>
41        <Stack.Navigator>
42          <Stack.Screen name="Home" component={HomeScreen} />
43        </Stack.Navigator>
44      </NavigationContainer>
45    );
46  }
47  export default App;
48
```

**Nota**: observar que **SafeAreaView** debe ser importado de: **react-native-safe-area-context**

**Práctica 20**: Reproducir el anterior ejemplo de Stack.Navigator con HomeScreen.

¿ Qué características tiene la navegación Stack ?

- Las pantallas que se cargan NO se “desmontan” quedan en RAM aunque hayamos pasado a otro componente. Eso implica que un useEffect() que se lanza únicamente en la carga inicial del componente ( componentdidmount() ) NO vuelve a lanzarse si cambiamos de pantalla y regresamos a la pantalla

- Tiene un comportamiento similar al historial de un navegador web, pero incorpora los efectos del cambio de pantalla nativo de IOS/Android

- Podemos definir que parámetros puede recibir mediante props una pantalla al ser cargada

Esto último, tiene que ver con la forma de navegar que tiene el usuario. Para poder navegar pulsando un botón, haremos uso de navigate(), que se recibe mediante props. Vamos a verlo:

## Cambiar de pantalla mediante: Navigator.navigate()

Evitamos warnings y podemos aprovechar todo el potencial de typescript para que nos ayude cuándo estamos escribiendo si definimos los parámetros que pueden ser recibidos en las pantallas. Así, en el fichero donde hemos puesto nuestro Stack.Navigator escribiremos parecido a:

```
type RootStackParamList = {
  Primera: undefined;
  Segunda: undefined;
  Tercera: {userName: string};
};

const Stack = createNativeStackNavigator<RootStackParamList>();

function App(): JSX.Element {

  return (
    <NavigationContainer>
      <Stack.Navigator
        screenOptions={{ //headerShown: false,  contentStyle: { backgroundColor: 'pink' }
        }}
      >
      <Stack.Screen name="Primera" component={PrimeraScreen} />
        <Stack.Screen name="Segunda" component={SegundaScreen} />
        <Stack.Screen name="Tercera" component={TerceraScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
export default App;
```

Al crear el stack: `const Stack = createNativeStackNavigator<RootStackParamList>();` le estamos pasando el tipo que incluye todos los parámetros de nuestras pantallas. Ese tipo de datos, que aquí hemos llamado: RootStackParamList (**cuidado**! Hay que exportar este tipo de datos porque lo usaremos en más ficheros ) le estamos diciendo que la primera pantalla no recibe parámetros: `Primera: undefined;` Lo mismo ocurre con la segunda pantalla, y la tercera sí recibe un parámetro llamado userName: `Tercera: {userName: string};`

Bien, ahora en el fichero de la screen desde la que queramos navegar hacia otra haremos algo así:

```
function PrimeraScreen({ navigation }: Props) {
  return (
    <SafeAreaView style={{flex: 1}}>
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
```

```
        <Text>Primera screen---------</Text>
        <Button title="cambiar a tercera" onPress={()=>navigation.navigate('Tercera', {userName:
"Aquilino"} )} />
         <Button title="cambiar a segunda" onPress={()=>navigation.navigate('Segunda' )} />
      </View>
    </SafeAreaView>
  );
}
```

Vemos que hay dos botones. Uno ejecuta: `navigation.navigate('Segunda' )` . La función navigate() significa: “vete a” y en este caso se le está diciendo que abra el componente que definimos como: “Segunda” en el StackNavigator. Veamos de nuevo donde definimos eso antes:

```
<Stack.Screen name="Segunda" component={SegundaScreen} />
```

En la línea anterior se le ha puesto en el StackNavigator el nombre: “Segunda” al componente: SegundaScreen Así que cuando le decimos: `navigation.navigate('Segunda' )` le estamos diciendo que vaya a SegundaScreen

También hay un botón que permite ir a TerceraScreen, en ese caso vemos que le pasa un parámetro: **userName: “Aquilino”** ¿ cómo recibe ese parámetro TerceraScreen ? Vamos a verlo:

```
type Props = NativeStackScreenProps<RootStackParamList, 'Tercera'>;
function TerceraScreen({ navigation, route }: Props) {

  return (
    <SafeAreaView style={{flex: 1}}>
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
        <Text>TERCERA. Nombre recibido: {route.params.userName}</Text>
      </View>

    </SafeAreaView>
  );
}
```

La sentencia importante aquí es:

```
type Props = NativeStackScreenProps<RootStackParamList, 'Tercera'>;
```

Con esa sentencia estamos diciendo que nuestras props vienen dadas por StackNavigation. Donde el primer genérico es el tipo de datos: RootStackParamList qu es el que creamos antes para definir los parámetros posibles en cada pantalla. El segundo parámetro hace referencia al nombre que le dimos a nuestro actual screen en el Stack.Navigator: 'Tercera'

Finalmente vemos que tomamos de las props: `{ navigation, route }: Props` Hemos tomado navigation que ese el que sabemos que nos permite ir a otro componente: navigation.navigate(), y adicionalmente: "route" Mediante route podemos recibir los parámetros: **router.params** hace referencia a los parámetros que hemos recibido mediante nuestro Navigation. En este caso sabemos que recibíamos un parámetro que hemos llamado: userName

**Práctica 21**: Reproducir el anterior ejemplo con tres Screen en el navigator. Poner un TextInput en PrimeraScreen para que el usuario introduzca un nombre. Cuando el usuario pulse el botón que lo lleva a TeceraScreen, el texto introducido en el TextInput se envía como parámetro a TerceraScreen y allí dira: "Saludos nombreusuario. Bienvenido!" Como sabemos, intentamos dejar nuestro componente App.tsx bastante limpio. Crear un nuevo componente: src/navigation/StackNavigation.tsx y trasladar todo el código de la navegación. De tal forma que nuestro App.tsx quede únicamente:

```
function App(): JSX.Element {
  return (    <StackNavigation />  );
}
export default App;
```

## Opciones: screenOptions para StackNavigator

Hay múltiples opciones. Se pueden encontrar todas en:
https://reactnavigation.org/docs/native-stack-navigator/#options

Ahora veremos sólo unas pocas:

```
<NavigationContainer>
  <Stack.Navigator
    screenOptions={{
        //headerShown: false,
        //contentStyle: { backgroundColor: 'pink' },
        headerStyle: { backgroundColor: 'rgb(186, 200, 198)' },
        headerTitleStyle: {color: "gray"},
        headerBackVisible: false,
        headerRight: ()=> <Button title="saludo" onPress={()=>Alert.alert("un saludo a todos!")} /> ,
        headerTitleAlign: 'center',
        title: "MiApp"
    }}
  >
```

headerShown establece si se va o no a ver un header.

headerStyle  permite poner el color de fondo del header

headerTitleStyle: permite varias cosas con el título: (fontSize,..) entre otras ponerle un color al texto: color

headerBackVisible: permite que se muestre o no el botón predeterminado de la cabecera para regresar ( hacer el goBack() ) a la pantalla previa

headerRight: permite una función que retorne un componente React ( en el ejemplo devuelve un botón que permite mostrar un saludo ) en la parte derecha del header

headerTitleAlign: permite ubicar el título: left, center, right

title: establece el título. Observar que si se pone como una screenOptions en Stack.Navigator, quedará establecido para todas las screen. Este mismo atributo se puede poner para cada screen en concreto

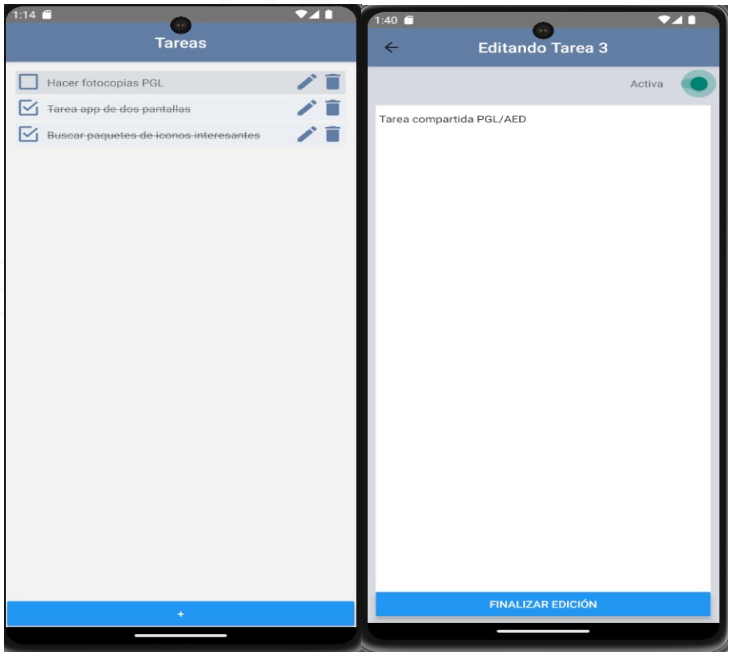
**Práctica 22**: Reproducir el anterior ejemplo con dos Screen en el navigator. Cambiar el color de fondo del header ( a uno de tu elección). Poner en blanco el color del título. Que no se muestre el headerBack ( establecerlo a false.Nota: según el emulador se podrá aún regresar arrastrando desde el extremo de la derecha hacia la izquierda. En otros emuladores quizás no se pueda regresar). Alinear el título en el centro y pon tu nombre en el título

# Contexto con react native y navegación

El manejo del contexto no es diferente a una aplicación normal de React. Lo único a tener en cuenta es la ubicación del provider para que el contexto sea común a toda la navegación. Así lo recomendable es que sea el primer hijo de <NavigatorContainer>

```
return (
        <NavigationContainer>
                <AppContextProvider>
                        <Stack.Navigator
```

**Práctica 23**: Vamos a hacer el clásico: todo list

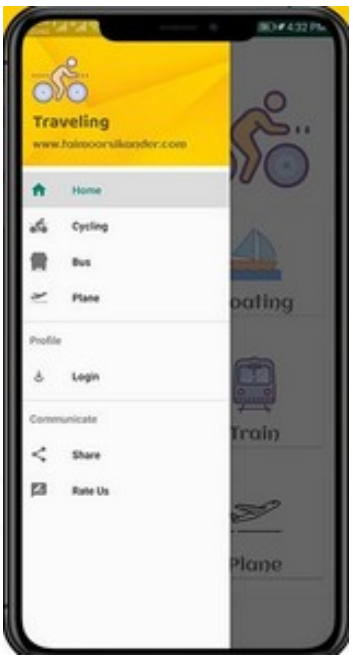



Aparecerán 2 pantallas, una con la lista de tareas. En cada fila del flatlist se puede pulsar sobre el check y que quede marcada como realizada o no ( observar que si está finalizada el texto queda tachado y cambia el icono ) al hacer click sobre el icono de editar nos lleva a la segunda pantalla y modificamos el asunto y si está o no realizada. Si se pulsa sobre el icono de papelera borra la tarea. El botón más: "+" nos lleva a la segunda pantalla para crear una nueva tarea ( los ids se generan automáticamente y el usuario no los puede editar )

Buscar por: "options" en Statck.Screen allí podemos poner opciones muy similar a: screenoptions de Stack.Navigator, pero adicionalmente puede tratar con parámetros de la ruta y así mostrar el id de la tarea en el título

Se debe generar un contexto donde se guardará el state de la lista de ToDos y su correspondiente setter.

# Drawer Navigation




Este tipo de navegación, usualmente sirve para mostrar el usuario activo y tener una lista de enlaces a los que acceder.

Para la instalación, debemos recordar que primero debe estar realizada la instalación del navigation: Navegación en react-native

En pocas palabras eran los dos comandos npm:

```
npm install @react-navigation/native
npm install react-native-screens react-native-safe-area-context
```

Y luego modificar: el fichero: **android/app/src/main/java/nuestropackage/MainActivity.java.**

```
import android.os.Bundle;

public class MainActivity extends ReactActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(null);
  }
}
```

Para instalar Drawer ahora ejecutamos:

```
npm install @react-navigation/drawer
npm install react-native-gesture-handler react-native-reanimated
```

Adicionalmente se debe hacer un import **DEBE SER LA PRIMERA LINEA DEL FICHERO** de la navegación (NavigationContainer). Habitualmente hablamos de: App.tsx

```
import 'react-native-gesture-handler';
```

Ej:

App.tsx > ...

```
1   import 'react-native-gesture-handler';
2   import { NavigationContainer } from '@react-navigation/native';
3   import React from 'react';
4
5   function App(): JSX.Element {
6     return (
7       <NavigationContainer>
```

Vamos a crear nuestro Drawer y lo llamaremos desde el NavigationContainer

Pongamos que lo llamamos: src/navigator/SideMenu.tsx

```
const Drawer = createDrawerNavigator();

const SideMenu = () => {
        return (
                <Drawer.Navigator>
                        <Drawer.Screen name="Pantalla1" component={Pantalla1Screen} />
                        <Drawer.Screen name="Pantalla2" component={Pantalla2Screen} />
                </Drawer.Navigator>
        )
}
export default SideMenu
```

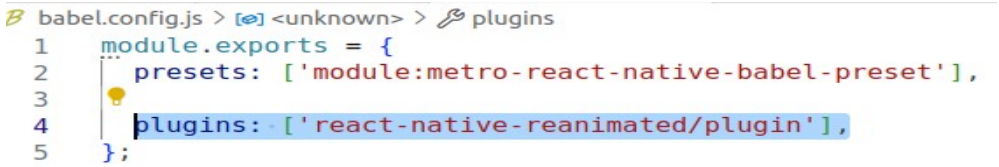
Lo relevante aquí es createDrawerNavigator(), por lo demás vemos que se asemeja a como definíamos los enlaces a las pantallas en Stack

App.tsx queda por tanto así:

```
function App(): JSX.Element {
        return (
                <NavigationContainer>
                        <SideMenu />
                </NavigationContainer>
        );
}
```

Por supuesto, debemos recordar que si pusiéramos un contexto en nuestras pantallas enlazadas por Drawer, pondríamos el provider del contexto justo como hijo de NavigationContainer y dentro del provider iría SideMenu

**Nota**: Si falla siguiendo lo establecido, modificar el fichero: babel.config.js para agregarle al final de modules.exports la línea: `plugins: ['react-native-reanimated/plugin'],` Ej:




y luego lanzar la aplicación borrando la caché:

```
npx react-native start --reset-cache
```

Otra opción es borrar la versión con npm del reanimated e instalar una inferior (1.4 por ejemplo)

**Práctica 24**: Crear una navegación Drawer. La primera Drawer.screen será: StackNavigation.tsx (así se cargará por defecto el Stack ) y la otra: AboutScreen.tsx. Poner en AboutScreen.tsx información tuya ( tú nombre, curso, etc ) En StackNavigation.tsx tendremos un Stack Navigator que apunte al menos a dos pantallas, una será una pantalla de enlaces a otras actividades del Stack, La segunda pantalla será la tarea de la calculadora ( que deberá estar enlazada en la página de enlaces ) Así desde el Drawer podemos acceer directamente a About y luego si desde el Drawer se elige el Stack podemos acceder a las otras pantallas del Stack.
Pon en la carpeta navigation tanto MenuLateral.tsx como StackNavigation.tsx

## Opciones de DrawerNavigation

Tiene múltiples opciones y lo mejor es ir a la documentación oficial. Únicamente vamos a ver algunas. Ej

```
function drawerPersonalizado(props: DrawerContentComponentProps){
    return(
    <DrawerContentScrollView>
      <View > {/* en este view ponemos libremente */}
        <Text>Mascotas</Text>
      </View>
      {/* <DrawerItemList {...props} />  */}
      <DrawerItem
        // icon={()=><Icon name='miicono'/>}
        label="Help"
        onPress={() => props.navigation.navigate('Pantalla2')}
      />

    </DrawerContentScrollView>
    );
}

return (
  <Drawer.Navigator
      screenOptions={{
          drawerStyle: {
              //backgroundColor: '#c6cbef',
              //width: 140,
          },
          drawerType: dimensions.width >= 768 ? "permanent":"front" ,
      }}
      drawerContent={(props)=>drawerPersonalizado(props)}
  >
    <Drawer.Screen name="Pantalla1" options={ {title: "Hola mundo" }} component={Pantalla1Screen} />
    <Drawer.Screen name="Pantalla2" options={ {title: "Segunda pantalla" }} component={Pantalla2Scre
    <Drawer.Screen name="StackNavigation" options={ {title: "Home" }} component={StackNavigation} />
  </Drawer.Navigator>
)
```

Vemos en Drawer.Navigator screenOptions y drawerContent. Las screenOptions permiten por ejemplo darle estilos al menú lateral ( drawerStyle: color de fondo, ancho del menú, etc ). También especificar si mostrarlo siempre o no en pantalla: drawerType: “permanent”→ menú lateral fijo, “front”→ este es el habitual

drawerContent permite personalizar totalmente el contenido de nuestro menú lateral. Se le debe pasar una función que devuelva un JSX. Hemos destacado las props: **DrawerContentComponentProps** Ya que ahí es donde recibimos el navigation (similar al de stack ). Algo importante ( para que permita el scroll y mantenga todas sus características de drawer) es que el JSX que devolvamos esté dentro de: **DrawerContentScrollView**

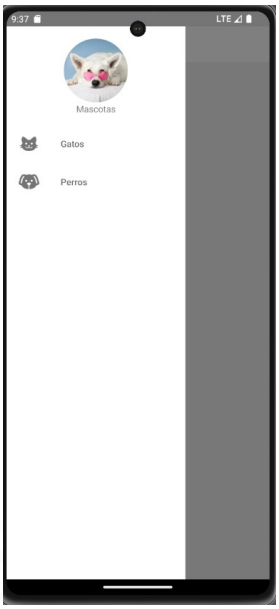
Tenemos entre comentarios: `<DrawerItemList {...props} />` Con este elemento se genera una lista de items de navegación hacia todas las pantallas declaradas en el drawer navigation

Justo debajo tenemos: `<DrawerItem>` que es la forma de ir agregando elementos a la lista ( incluso podemos prescindir de DrawerItemList y hacer todo con DrawerItem ) Observar que la navegación se hace al estilo de stack navigation, mediante props.navigation.navigate() y pasando el: **name** que hayamos declarado en el Drawer.Screen correspondiente.

A DrawerItem podemos pasarle iconos ( como aparece entre comentarios ) y más ( ver la documentación oficial )

También vemos que Drawer.Screen admite “options” por ejemplo le podemos pasar un JSON con un título personalizado, que será el que se mostrará al usuario

**Práctica 25**: Reproducir el ejemplo anterior. Girar la pantalla ¿ quéda el menú visible ? Cambiar el color de fondo del menú lateral y poner que tenga un ancho de 200. Prueba a comentar y descomentar la línea del drawerContent ¿ Cuándo lo activas siguen mostrándose los enlaces a las pantallas ?
Vamos a personalizar el menú.




Imaginemos una app de mascotas. El Drawer nos muestra la opción de ir a las categorías principales. Una vez se entra se muestra un lista de razas y se puede pulsar sobre cada una de ellas, abriendo otra pantalla que informe sobre la raza en concreto (debe mostrar una foto). Para disponer de la información usaremos un contexto ( el provider justo dentro del NavigationContainer y dentro del provider el DrawerNavigation ) Recordar que el contexto admite unos valores iniciales:

< AppContext.Provider value={valoresiniciales} >

En esos valoresiniciales estableceremos nuestros datos para la app

Instalar los iconos para usarlos en el Drawer

Para hacer la navegación únicamente con Drawer debemos incluir las 4 pantallas en Drawer.Navigation y luego para ir a las pantallas que precisan parámetros ( la de la raza de perro individual y la de la raza de gato ), cuándo hagamos botones ( por ej. con touchableopacity ) pondríamos cosas similares a:

```
onPress={()=>props.navigation.navigate("Cat",{raza: "persa" }) }
```

Con lo anterior luego recogemos en la pantalla el parámetro mediante:

```
props.route.params.raza
```

Que no es diferente a como haríamos en StackNavigation ( también podemos tener un Drawer y un Stack a la vez, una dentro de la otra. Por ejemplo, una de las Screen declaradas en Drawer.Navigation puede ser el StackNavigation )

**Práctica 26**: Realizar la app de mascotas descrita

# Tab Navigation

Como todas las navegaciones que hemos visto, hay que hacer la configuración inicial de navigator: React Native navigation Para tener tabs adicionalmente:

```
npm install @react-navigation/bottom-tabs
```

Ahora de forma similar a las anteriores, crearemos un componente que pondremos dentro de la carpeta navigation para poner la navegación por tabs. Por ejemplo: src/navigation/Tabs.tsx

```
const Tab = createBottomTabNavigator();

const Tabs = (props: any) => {
  return (
    <Tab.Navigator screenOptions={{ headerShown: false }} >
        <Tab.Screen name="Viajes" component={StackNavigation}
            options={{tabBarIcon: ({focused})=> <Icon name={(focused)?'car':'car-outline'} size={30}/>}}
        />
        <Tab.Screen name="About" component={AboutScreen} />
    </Tab.Navigator>

  )
}

export default Tabs
```

Observar que en el trozo de código anterior aparece un StackNavigation como una de las tabs. Es habitual que dentro de cada Tab haya que hacer subnavegaciones. Por ejemplo, en viajes separar entre viajes organizados y no organizados, etc.

Vemos también que aparece options en Tab.Screen ( de forma análoga a con las otras navegaciones ) Dentro de estas options tenemos: tabaBarIcon que nos permite personalizar el icono. Observar que se le pasa una función que recibe parámetros. En este caso en concreto, recibe: focused que nos informa si es o no la Tab activa. Si es activa cargamos un icono y si no cargamos uno diferente

Finalmente también vemos que existe screenOptions en TabNavigator. En este caso se usa para que no aparezca la cabecera ( las tabs ya son bastante descriptivas y deja más espacio para que haya una cabecera para los StackNavigation anidados )

**Práctica 27**: Crear una app con dos tabs. En una de ellas aparecerá un icono relacionado con operaciones matemáticas y al acceder cargará un StackNavigation que en su primera pantalla tendrá enlaces a tareas realizadas ( una de ellas será la calculadora

La otra tab tendrá el About.tsx que mostrará tu nombre, curso, aficiones

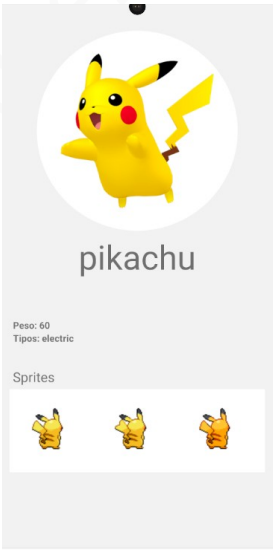
TabNavigation tiene opciones muy parecidas a StackNavigation, no nos vamos a detener más en ella. Para mayor profundización acudir a la documentación oficial

# Accediendo a APIs

No hay nada diferente en la forma que se accede a una api con Native de como se hace en React. ( podemos usar fetch, pero habitualmente instalaremos axios para las peticiones ) La diferencia será en la forma de persistir/cachear los datos obtenidos. En React se dispone de LocalStorage, aquí está AsyncStorage e incluso se puede guardar y recuperar de base de datos local

**Práctica 28**: Aplicación Pokemon.

Haremos una aplicación para ver datos de pokemon. Habrá dos tabs y en cada tab tendremos un StackNavigation. La primera tab tiene un Stack que será para visualizar la lista de pokemon ( usaremos un flatlist ) en la primera pantalla del Stack y cuando se pulse sobre un pokemon se abrirá otra Screen del Stack para ver todos los datos del pokemon ( con sus diferentes sprites )
La segunda tab tiene otro Stack. En la primera de las pantallas del Stack hay un cuadro de texto para buscar pokemon por subnombre ( aparecerán los pokemon que en su nombre incluyan el texto introducido ). Cuando pulsamos sobre alguno de la lista obtenida nos lleva a la Screen con todos los datos del pokemon.



# Permisos

Vamos a instalar: react-native-permissions Podemos ver la instalación actualizada en: https://github.com/zoontek/react-native-permissions

Básicamente lo que haremos es:

```
npm install --save react-native-permissions
```

Ahora hay que ir a: android/app/src/main/AndroidManifest.xml y agregar los permisos que se quieran tener ( verlos en el enlace dado arriba )

Con la instalación por defecto de un proyecto react-native ya viene puesto el permiso de acceso a Internet ( que si eliminamos la línea dejaría de tenerlo ) Vamos a agregar los permisos vinculados a geolocalización ( podemos copiarlos de la web dada antes )

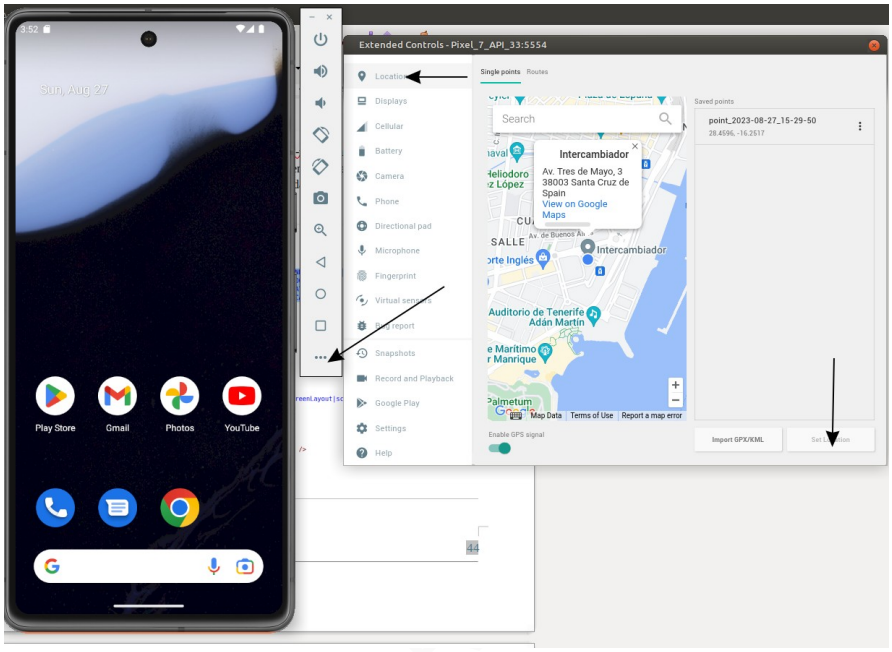
android > app > src > main > AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.INTERNET" />

    <uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_MEDIA_LOCATION" />

    <application
      android:name=".MainApplication"
      android:label="@string/app_name"
      android:icon="@mipmap/ic_launcher"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:allowBackup="false"
      android:theme="@style/AppTheme">
      <activity
        android:name=".MainActivity"
        android:label="@string/app_name"
        android:configChanges="keyboard|keyboardHidden|orientation|screenLayout|screenSize|smallestScreenSize|uiMode"
        android:launchMode="singleTask"
        android:windowSoftInputMode="adjustResize"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
    </application>
</manifest>
```

Vamos a probarlo. Primero vamos a poner una ubicación en el emulador:

Se ha señalado con flechas donde hay que pulsar. Primero en los tres puntos para abrir las opciones. Después se elige el primer elemento que aparece: localización

Después ubicarse en el mapa y finalmente pulsar en: set location




Se pueden generar rutas y que cambie la posición con el tiempo, etc.

Nota: Es posible que falle al tomar datos al principio. Si el servicio de geolocalización no cambia las cifras al cambiar en el emulador la ubicación. Lanzar google maps en el emulador

Ahora necesitamos que la app pueda obtener la localización que hemos establecido en el teléfono. Para ello instalamos el paquete:

```
npm install react-native-geolocation-service --save
```

El siguiente código permite hacer log de la ubicación:

```
import { PERMISSIONS, PermissionStatus, check, request } from 'react-native-permissions'
const ViajesScreen = (props: Props) => {
    async function verPosicion(){
        let ps: PermissionStatus;

        ps = await check(PERMISSIONS.ANDROID.ACCESS_FINE_LOCATION); //precisión de gps
        if( ps != 'granted'){
            ps = await request(PERMISSIONS.ANDROID.ACCESS_FINE_LOCATION);
        }

        if( ps == 'granted'){
            Geolocation.getCurrentPosition(info=>console.log(info));
        }else{
            console.log("no hay permisos ");
        }
    }

    return (
        <View>
        <Text>ViajesScreen</Text>
        <Button title="permiso gps" onPress={verPosicion}></Button>
        </View>
    )
}
```

**PermissionStatus** tiene 5 opciones, entre otras: “granted” ( permiso concedido) “denied” ( no concedido) . Se ha marcado el import porque puede equivocarse. Deben hacerse de: react-native-permissions.

Hay dos funciones asíncronas: **check**() ( que obtiene el status del permiso ) y **request**() (solicita permiso al usuario ) . Geolocation.getCurrentPosition() es asíncrona y recibe una función callback

**Práctica 29**: Reproducir el ejemplo anterior. Pero en lugar de console.log aparecerá la localización en un <Text> mediante json.stringify(). En el caso de que no se tenga permisos se mostrará que no los hay. Probar a conceder o no el permiso, luego eliminarlo, etc

# Async Storage, Webview

En react native tenemos una solución sencilla para tener persistencia. Se asemeja al local storage de React. Guardamos siempre clave valor y únicamente se guardan string. Así que los objetos tienen que pasar por JSON.stringify(). Únicamente se precisan dos órdenes muy sencillas.

Para instalarlo:

```
npm install @react-native-async-storage/async-storage
```

Recuperando información:

```
const jsonValue = await AsyncStorage.getItem('my-key'); //se recupera de storage
const data = jsonValue != null ? JSON.parse(jsonValue) : null;
```

Guardando información:

```
const storeData = async (value) => {
  try {
    const jsonValue = JSON.stringify(value);
    await AsyncStorage.setItem('my-key', jsonValue);
  } catch (e) {
    // saving error
  }
};
```

Sus limitaciones: por defecto tamaño reducido (6MB ) aunque cambiable. No tiene cifrado. Así que los datos que guarda no están a salvo si el teléfono cae en manos indebidas, generando problema de seguridad si datos sensibles

**Práctica 30**: Vamos a crear un historial de ubicaciones, en la actividad de permisos de geolocalización. Vamos a modificarla de tal forma que el botón lo que haga es guardar en asyncstorage la latitud, longitud y timestamp ( se debe mantener que si no hay permisos se soliciten ) Al pulsar en el botón se informará en un <Text> de que se ha guardado y la posición que ha guardado. Habrá otro botón que se llamará “historial” que al pulsarlo lee del asyncstorage la lista de posiciones guardadas y las muestra en un <Text>

Async Storage nos puede ayudar a tener una caché y no depender tanto de la red. Así cuando haya conexión y tomemos con éxito información de una api lo podemos almacenar en nuestro storage. Así en situaciones que no haya conectividad la app puede seguir mostrando los datos que previamente hemos descargado aunque hayamos cerrado la aplicación y la volvamos a abrir.

Un ejemplo sencillo de lo anterior:

```
async function getCache(uri:string){
        try{
                const response = await axios.get(uri);
                const data = response.data ;
                AsyncStorage.setItem(uri,JSON.stringify(data))
                return data;
        }catch( error){
                const dat = await AsyncStorage.getItem(uri);
                if( dat){
                        const data = JSON.parse(dat);
                }
        }
}
```

Vamos a hacer un lector de noticias de tecnología. Tomaremos el agregador:

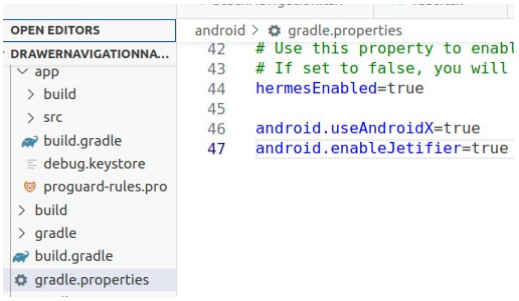
https://www.xataka.com/feedburner.xml

En la primera página lo que se mostrará es los títulos de las noticias: <title> Al pulsar encima de un título va a otra pantalla y muestra la noticia. Para poder ver la noticia haremos uso de **webview**:

```
npm install --save react-native-webview
```

Hay que agregar dos propiedades en android/gradle.properties:

```
android.useAndroidX=true
android.enableJetifier=true
```




```
<WebView
  source={{
    html: `<html>
            <body>
              <h1>hola mundo!</h1>
            </body>
          </html>`
  }}
  style={{ width:"100%", maxHeight: "70%" }}
/>
```

En el **source** recibe un json donde usa un atributo: **html** si le damos directamente el código html o un atributo **uri** si queremos renderizar una página web externa

Si el usuario ya ha hecho click en la noticia se modificará el color . Al cargar la primer pantalla se guardará en AsynStorage el contenido de las páginas enlazadas.

Precisaremos convertir el xml a json para ello instalaremos un parser xml:

```
npm install react-native-rss-parser

npm i --save-dev @types/react-native-rss-parser
```

Para importarlo: `import * as rssParser from 'react-native-rss-parser';`

Un useState para las noticias: `const [articulos, setarticulos] = useState<rssParser.Feed>();`

Usando el parser y convirtiendo a JSON:

```
const response = await axios.get(urlactual)
const responseData = await rssParser.parse(response.data);
```

**Práctica 31**: Hacer la actividad descrita. Cuando arranque la aplicación y se muestre la primera pantalla poner modo avión cerrar la aplicación y volver abrir. Hacer click en alguna de las noticias. Si todo está bien cargará la noticia del async storage y la mostrará

# Bases de datos: Sqlite y TypeORM

Vamos a instalar TypeORM que es un ORM para múltiples DDBB y tiene soporte para react-native.

Nota: Se va a utilizar el sqlite que viene incorporado con android. El paquete para sqlite que vamos a usar: react-native-sqlite-storage permite poner su propio sqlite ( indicaciones en la página del proyecto )  Para ios hay que seguir las instrucciones dadas en la web del proyecto

```
npm install typeorm --save

npm install reflect-metadata --save

npm install @types/node --save-dev

npm install --save react-native-sqlite-storage

npm install --save-dev @babel/plugin-proposal-decorators
```

Luego hay que modificar tsconfig.json. Quedando el fichero así:

```
{
  "extends": "@tsconfig/react-native/tsconfig.json",
  "compilerOptions": {
    "strict": false,
    "allowSyntheticDefaultImports": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "skipLibCheck": true
  }
}
```

También hay que modificar: babel.config.js agregando el plugin de proposal decorators:

```
module.exports = {
   presets: ['module:metro-react-native-babel-preset'],
   plugins: [['@babel/plugin-proposal-decorators', {legacy: true}]],
};
```

# Usando TypeORM

Los ORM ( Object Relational Mapping ) permiten transformar nuestros objetos de programación en tablas de base de datos relacional.

TypeORM tiene muchas opciones y aquí únicamente veremos un subconjunto que permita trabajar mínimamente con Sqlite**. Se recomienda ir a la documentación oficial para ampliar**

Cuando trabajamos con bases de datos compartidas, es probable que nuestra aplicación tome ya una base de datos hecha y en ese caso es más discutible crear las tablas mediante una herramienta ORM. Pero cuando estamos hablando de programación en dispositivo móvil, la base de datos sqlite es local y propia de nuestra aplicación. Así que el procedimento será crear todo desde el ORM.

Vamos a inicializar la base de datos mediante TypeORM desde el punto superior de nuestra aplicación. Para conseguir esto lanzaremos un useEffect() en App.tsx. Así en el momento en el que se monte la aplicación se disparará la inicialización de la base de datos

```
function App(): JSX.Element {
  useEffect(() => {
    async function iniciarDDBB(){
      await dataSource.initialize();
    }
    iniciarDDBB();
  }, [])
```

El objeto **dataSource** que vemos, se crea en la carpeta data: **/src/data** y se ubiera una carpeta api la pondríamos dentro de data, para tener todos los accesos a datos en un mismo sitio. Lo crearemos en un fichero llamado: **src/data/Database.ts**

src > data > TS Database.ts > ...

```
1
2   import {DataSource} from 'typeorm';
3   import { Persona } from './entity/Persona';
4   import { Ejercicio } from './entity/Ejercicio';
5
6
7   export const dataSource = new DataSource({
8     database: 'personasdb.db',
9     entities: [Persona, Ejercicio],
10    location: 'default',
11    logging: [],
12    //logging: ['error', 'query', 'schema'],
13    synchronize: true,
14    type: 'react-native',
15
16  });
17
18  export const PersonaRepository = dataSource.getRepository(Persona);
19  export const EjercicioRepository = dataSource.getRepository(Ejercicio);
20
```

En el código anterior se ve como se genera todo lo necesario. En: **database** ponemos el nombre que tendrá la base de datos. En: **entities** el nombre de las clases que queremos llevar a base de datos. En: **location** decimos la ubicación de la base de datos ( parámetro de react-native-sqlite-storage ) . En: **logging** determinamos que queremos que se muestre de información, si le pasamos un array vacío no informa. En **synchronize** informas de si se debe crear/actualizar el esquema de la base de datos al lanzar la aplicación. Finalmente **type** es para establecer a qué conectar (react-native, postgres,…).

Por supuesto necesitamos generar las entities en carpeta: **src/data/entity.** Las entity de typeorm son clases que **extienden de BaseEntity** con decorador **@Entity**. Ej para: src/data/entity/persona.ts también con decorador @Column para especificar que se almacena como: int en la base de datos el atributo edad

```
@Entity('persona')
export class Persona extends BaseEntity {
  //si no queremos autogenerado: @PrimaryColumn
  @PrimaryGeneratedColumn() id: number;
  @Column('text') nombre: string;
  @Column('int') edad: number;
}
```

```
const [personas, setpersonas] = useState<Persona[]>([]);

async function grabar(){
    const array = [ "Ana", "Martino", "Rebeca"];
    let neopersonas = [];
    for (let i=0;i<3;i++) {
        const randomPositionArray = Math.floor(array.length*Math.random());
        const nombre =  array[randomPositionArray];
        const edad = Math.round(Math.random()*100)+1;
        const persona = {
            nombre: nombre ,
            edad: edad
        };
        neopersonas.push(persona);
    }
    await PersonaRepository.save(neopersonas);
    const newpersonas = await PersonaRepository.find();
    setpersonas(newpersonas);
}

return (
  <View style={{flex:1}}>
      <FlatList
          data={personas}
          renderItem={
              (p)=><Text>{p.item.id + " " + p.item.nombre}</Text>
          }
          keyExtractor={(it,index) => ""+index}
      />
    <Button title="crear personas" onPress={grabar} />
  </View>
)
```

PersonaRespository.save() guarda todo el array a la vez ( podríamos haber puesto el save() dentro del bucle y guardar las personas una a una también ) PersonaRespository.find() obtiene todas las personas de la tabla

**Práctica 32**: a) Reproducir el ejemplo anterior ( usar únicamente la clase Persona no hace falta la otra clase )
b) Poner dos textinput. Uno para la edad y otro para el nombre. Ahora al pulsar en el botón, en lugar de datos aleatorios, se creará en la base de datos la persona introducida por el usuario

Se puede utilizar: .**save**() para actualizar en lugar de guardar ( si por ejemplo, guardamos con un id dado un objeto completo lo actualiza si encuentra el mismo id ) Pero tenemos una opción específica: .**update**()

```
await PersonaRepository.update(
{ //condición: las personas llamadas Ana
    nombre: "Ana",
},
{ //update a Ana María y edad 23
    nombre: "Ana María",
    edad: 23
});
```

En el ejemplo vemos que el primer objeto que le pasamos a update() establece las condiciones ( nombre Ana ) y el segundo objeto ( observar que está separado por comas ) se envía lo que se actualiza ( los atributos del objeto Persona que se omitan no se verían modificados en el update )

Podemos hacer el borrado con delete() de una forma similar. Enviamos un único objeto con la condición que tiene que cumplir para ser borrado:

```
await PersonaRepository.delete({
        edad: 80
});
```

**Práctica 33**: En la actividad del TodoList vamos a persistir en base de datos, y así una vez cerremos la aplicación y la volvamos a abrir se mantendrá la información de las tareas
Podemos utilizar el tipo boolean en TypeORM:
@Column('boolean') completada: boolean;
Para decir si se completó el todo. Observar que para evitar errores, cuando corresponda el update y el delete de la tarea, mejor usar el id como condición.

## Find

Las consultas con find() nos devuelven un array. Si queremos obtener un único objeto, usaremos: **findOneBy**() Un habitual es buscar por id Ej:

```
personaRepository.findOneBy({id: 1})
```

Ya hemos usado find. Pero tiene varias funciones que precisa ver aparte:

En este ejemplo vemos que podemos elegir las columnas que queramos que muestre mediante select ( si no se pone el select devuelve todas las columnas del objeto ) y le podemos pasar un where que soporta: AND

```
const findPersonas = await PersonaRepository.find({
    // si no ponemos select devuelve todas las columnas
    select: {
        nombre: true,
        edad: true
    },
    where: {
        //se puede filtrar varias condiciones: AND
        //personas de 18 a 67 años cuyo nombre incluya: "ar"
        edad: Between(18,67), // existe LessThan, MoreThan...
        nombre: Like("%ar%")
    }
});
```

Es equivalente a:

```
select nombre, edad
where edad >= 18 AND edad <= 67 AND nombre like '%ar%'
```

También podemos hacer un OR. Para ello le pasamos un array al where. Order mediante la cláusula order y limit mediante: take. El ejemplo equivale a:

```
const findPersonas = await PersonaRepository.find({
    where: [
        { edad: LessThan(16) },
        { edad: MoreThan(67) },
    ],
    order: {
        nombre: 'DESC',
        edad: 'DESC'
    },
    take: 3
});
```

```
select *
where edad < 16 OR edad > 67
order by nombre DESC, edad DESC
```

## @Entity y opciones en @Column, tipos de datos

Ya hemos visto algunos tipos de datos. Los tipos que soporta TypeORM dependen de la base de datos a la que están accediendo. En nuestro caso de SQLite/react-native son:

int, int2, int8, integer, tinyint, smallint, mediumint, bigint, decimal, numeric, float, double, real, double precision, datetime, varying character, character, native character, varchar, nchar, nvarchar2, unsigned big int, boolean, blob, text, clob, date

**@Entity** lo pondremos siempre decorando la clase que queramos que la maneje el ORM. Si ponemos: @Entity('personas') estamos diciendo que queremos que en la base de datos, la tabla se llame personas. Para especificar la clave principal tenemos: **@PrimaryColumn()** y **@PrimaryGeneratedColumn()** Donde el segundo nos genera un autoincremental. Para claves compuestas escribiremos @PrimaryColumn() en todas las implicadas:

```
@Entity()
export class User {
        @PrimaryColumn("text") firstName: string
        @PrimaryColumn("text") lastName: string
}
```

Si queremos que una columna admita nulos pondremos: **nullable** a true. Y si queremos que sus valores no se repitan pondremos: **unique** a true. En el siguiente ejemplo le pasamos un único json a @Column diciendo que puede ser nulo. Observar que hemos pasado: type para especificar el tipo de datos que guarde en DDBB

```
@Column({type: "boolean", nullable: true, unique: false}) vacunada: boolean
```

**Práctica 34**: Vamos a crear una actividad que emule una tienda de productos de comida. Tendremos tabs y en una de las tabs introducimos los productos (nombre, precio, stock y un boolean: descontinuado ) En la segunda tab hay un cuadro de texto para subnombre, otro para precio menor que el número dado y otro para precio mayor qué el número dado y un switch para elegir si ordenamos por nombre ascendente ( si el switch está a false no ordena )

## Relaciones

Para establecerlas lo haremos según lo que corresponda en base datos:

Para relaciones 1:N => @OneToMany()
Para relaciones N:1 => @ManyToOne()
Para relaciones N:M => @ManyToMany()

### *@OneToMany() y ManyToOne()*

Es un decorador que se pone en la parte: 1 de las relaciones 1:N y precisamos informar de: que entidad (tabla) es la parte: N de la relación y como se llama el campo en la otra entidad que guardará la información de la foreign key

```
@Entity()
export class Categoria  extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number;
    @Column('text')
    nombre: string;

    @OneToMany( ()=> Producto, (producto)=>producto.categoria)
    productos: Producto[]

}
```

En el ejemplo vemos: `@OneToMany( ()=> Producto, (producto)=>producto.categoria)` Así en el primer parámetro le pasamos la función: ()=>Producto que indica la otra entidad

En el segundo parámetro pasamos la función: (producto)=>producto.categoria que indica que la entidad: Producto se vincula mediante el campo: categoria con nuestra entidad Categoria

Observar que en el ejemplo también aparece: `productos: Producto[]` Así en: categoria.productos puede haber un array que refleje todos los productos de la categoría

En la otra entidad informamos de la otra parte de la relación:

Observamos que se hace parecido a antes

```
@Entity()
export class Producto  extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number;
    @Column("text")
    nombre: string;
    @Column("double")
    precio: number;

    @ManyToOne(()=>Categoria, (categoria)=>categoria.productos )
    categoria: Categoria
}
```

**Práctica 35**: Continuamos con la tienda. Crearemos una entidad categoría y tendremos un tab para introducir las categorías. Poner ya las relaciones Si hay problema con los datos almacenados recreamos la base de datos desde cero ( basta cambiar el nombre de la database y ejecutar: npx react-native start --reset-cache.
No nos preocuparemos ahora de como guardar el producto con la categoría vinculada

```
async function grabar(){

    let tuberculos = {
        nombre: "tuberculos"
    }

    //en tuberculosCategoria tendremos un id autogenerado
    const tuberculosCategoria = await CategoriaRepository.save(tuberculos);

    //como tenemos la categoria se la podemos pasar ahora al json producto
    let papas = {
        nombre: "papas",
        precio: 4.5,
        categoria: tuberculosCategoria
    }

    const papasProducto = await ProductoRepository.save(papas);
    console.log(papasProducto);
}
```

El ejemplo reproduce el caso en el que la categoría es nueva. Si fuera una ya existente en lugar del save() usaríamos un findOneBy():

```
const tuberculosCategoria = await CategoriaRepository.findOneBy({nombre: "tuberculos"});
```

**Práctica 36**: Continuamos con la tienda. Ahora en la tab para crear producto hay un input para poner el nombre de la categoría y ya debe funcionar la relación bien

Al recuperar con find() no nos aparece la información de la relación por defecto. Tenemos que enviarle a find() que relaciones queremos que nos cargue también al hacer la búsqueda, mediante: relations: {productos: true} estamos diciendo que queremos que cargue la información de los productos relacionados. En el ejemplo, si comentamos la línea de relations no nos dará información de los productos de cada categoría.

```
const categorias = await CategoriaRepository.find({
    relations: { productos: true}
});

for (const cat of categorias) {
    console.log(cat);
}
```

**Práctica 37**: Hacer otra tab en la actividad de la tienda donde se muestren las categorías en un FlatList y por cada categoría haremos un JSON.stringify() que permita ver los productos que tiene cada categoría

### *Relaciones N:M  uso de @ManyToMany()*

Si la relación es N:M  tenemos dos subcasos, según si la tabla intermedia de la relación N:M tiene atributos propios  Si tiene atributos propios entonces crearemos una nueva entidad para la tabla intermedia y lo gestionamos como relaciones 1:N  ( mediante @OneToMany() y @ManyToOne() ).  Si no tiene atributos  propios usaremos @ManyToMany()

Veamos un ejemplo y explicamos. Se modela que un propietario tiene muchas casas y una casa tiene muchos propietarios: N:M  No habrá atributos propios en la relación.

```
@Entity()
export class Propietario {
    @PrimaryGeneratedColumn()
    id: number;

    @Column('text')
    nombre: string;

    @ManyToMany( () => Casa, (casa)=>casa.propietarios  )
    @JoinTable({
        name: "propietarios_casas",
        joinColumn: {
            name: "propietarios",
            referencedColumnName: "id"
        },
        inverseJoinColumn: {
            name: "casas",
            referencedColumnName: "referenciaCatastral"
        }

    })
    casas: Casa[]
}

@Entity()
export class Casa {
    @PrimaryColumn('text')
    referenciaCatastral: string

    @Column('double')
    metros: number

    @ManyToMany( () => Propietario, (propietario)=>propietario.casas)
    propietarios: Propietario[]
}
```

La parte que es diferente a ejemplos anteriores es el @JoinTable Vamos a tratar de verlo mejor:

```
@ManyToMany( () => Casa, (casa)=>casa.propietarios )
@JoinTable({
        name: "propietarios_casas",
        joinColumn: {
                name: "propietarios",
                referencedColumnName: "id"
        },
        inverseJoinColumn: {
                name: "casas",
                referencedColumnName: "referenciaCatastral"
        }
})
casas: Casa[]
```

Le especificamos el nombre de la tabla que hace de intermedia: propietarios_casas

Recordar que estamos ubicados en la entity: Propietario. Desde ahí se quiere decir como llegar a la información de referencia de la otra tabla/entidad. A eso es lo que llamamos: **joinColumn** Vemos que en ese: joinColumn aparece:

```
                name: "propietarios",
                referencedColumnName: "id"
```

Ahí cuando aparece: name: “propietarios” le estamos diciendo que en la otra entidad nos vinculamos por el atributo: propietarios. Osea: Casa.propietarios

Adicionalmente en: **referencedColumnName** le decimos la columna nuestra ( de nuestra entidad Propietario ) que forma parte del vínculo. Osea: Propietario.id

Luego aparece inverseJoinColumn que es lo mismo que joinColumn pero visto desde la otra entidad: Casa. Y la información que dice es: el campo: Casa.referenciaCatastral está referenciado en el campo: Propietario.casas

Vamos a ver ahora como guardamos y recuperamos información en este tipo de relación

```
let propietarioEntity = await PropietarioRepository.save({nombre:"Rigoberto"});
console.log(propietarioEntity);

const casaDeRigoberto={
    referenciaCatastral: "PPPP",
    metros: 190,
}
let casaRigobertoEntity = await CasaRepository.save(casaDeRigoberto);

const casaDeRigoberto2={
    referenciaCatastral: "QQQQQ",
    metros: 40,
}
let casaRigoberto2Entity = await CasaRepository.save(casaDeRigoberto2);

propietarioEntity.casas = [casaRigobertoEntity,casaRigoberto2Entity];
propietarioEntity = await PropietarioRepository.save(propietarioEntity);

let rigobertoEntity = await PropietarioRepository.find({where:{nombre:"Rigoberto"}, relations:{casas:true}});

for( const propietario of rigobertoEntity){
    console.log(propietario);
}
```

Si bien se puede usar una opción para hacer las grabaciones en cascada, estamos evitándolo. ¿Si fuera nosotros directamente en sql como lo haríamos ? Primero crearíamos el propietario y obtendríamos el id que se genera ( observar que eso es lo mismo que hacemos en la primera linea del ejemplo. Guardamos el propietario sin información de sus casas )

Después crearíamos las dos casas de Rigoberto en la tabla casas ( que es justo lo que aparece en los dos siguientes CasaRepository.save() de nuestro ejemplo. Otra vez fijarse que únicamente estamos guardando casas. NO hay vínculo entre el propietario y las casas aún )

Finalmente en sql guardaríamos en la tabla intermedia: propietarios_casas los ids de propietario y casas para dejarlas así vinculadas. En nuestro ejemplo lo que tenemos es:

```
propietarioEntity.casas = [casaRigobertoEntity,casaRigoberto2Entity];
propietarioEntity = await PropietarioRepository.save(propietarioEntity);
```

Observar que AHORA SÍ establecemos el vínculo, ya que le pasamos el array de casas al propietario. Luego volvemos a ejecutar: PropietarioRepository.save() y así se guarda el vínculo

Finalmente el ejemplo concluye diciendo como recuperamos la información vinculada en un find: `await PropietarioRepository.find({where:{nombre:"Rigoberto"}, relations:{casas:true}});`

Para que nos devuelva las casas de un propietario cuando buscamos un propietario en la base de datos, le decimos que queremos que tenga en cuenta esa relación: **relations: {casas:true}**

**Práctica 38**: Reproducir el ejemplo anterior en un ejercicio aparte. Habrá varios tabs. Uno para introducir propietarios. Otro para introducir casas y una tercera tab para vincular un propietario con una casa. En una cuarta tab se mostrará en un FlatList los propietarios con JSON.stringify y debe aparecer la información de sus casas

El borrado en la tabla intermedia no se hace con delete Se hace volviendo a guardar pero ahora NO aparece la información que se quiere borrar:

```
propietarioEntity.casas = [] ; //le especificamos un array vacío => no tiene relaciones
await PropietarioRepository.save(propietarioEntity); // guardamos de nuevo el propietario
```

**Práctica 39**: Crear otra pestaña donde se puede poner en un input el id del propietario y en el otro la referencia catastral de la casa y que al pulsar en el botón: borrar se elimine el vínculo entre los dos objetos en la base de datos

## Transacciones

Hay que basarse en el objeto Datasource para crear una transacción. Si bien muchas veces tiene más sentido proteger con una transacción a nivel de Service ( recordar que los servicios son objetos que reflejan los casos de uso de nuestra app ) aprovechamos para ver como podemos hacernos nuestros repositorios personalizados:

```
export const PersonaRepository = dataSource.getRepository(Persona).extend({
  async guardarPadreHijo(hijo: Persona, padre: Persona){
    await dataSource.transaction(async (transactionalEntityManager) => {
      await transactionalEntityManager.getRepository(Persona).save(padre);
      await transactionalEntityManager.getRepository(Persona).save(hijo);

    });
  }
})
```

En el código de ejemplo vemos: `dataSource.getRepository(Persona).extend()` que nos permite ampliar el repositorio que genera TypeORM

Dentro de ese: extend() ponemos todos los métodos personalizados que queramos poner ( es habitual poner métodos como: findByName() etc ) En el ejemplo se ha credo una acción: guardarPadreHijo() donde se quiere emular que si uno de los dos save() no tiene lugar, el otro ejecutaría un rollback Observar que para generar la transacción tenemos que usar el dataSource directamente: dataSource.transaction() . La idea es que todo el código que ponemos dentro de esa función queda protegido por la transacción

**Práctica 40**: Reproducir el ejemplo anterior con la app de propietarios-casas. De tal forma que se pueda guardar un array de casas protegido por una transacción. Llamaremos al método del repositorio: saveAtOnce( casas: Casa[] )