Construyendo una tienda...

Ejercicio 1 - Integrando nuestro contador

Empezaremos utilizando el componente `Counter` que hemos utilizado en el tutorial de ejemplo. Como vamos a implementar una tienda online, utilizaremos ese componente para determinar el número de productos que queremos añadir a nuestro carrito. Pero para ello será necesario hacer unos cuantos retoques...

Cosas a tener en cuenta... Aunque sea JavaScript y parezca que estamos haciendo algo similar a JS + HTML, en React Native es necesario utilizar elementos nativos del entorno mobile.

- El componente `<div>` de la web tendría su correspondencia con el componente `<View>` de React Native.
- El componente `<button>` de la web tendría su correspondencia con el componente `<TouchableOpacity>` de React Native. (El `onClick` pasaría a llamarse `onPress`)
- Los textos en React Native tienen que ir siempre dentro de un componente `<Text>` sino la compilación fallará.
- Los estilos creados se pasan a los componentes a través de la prop `style` .

Paso 1

Crear en nuestra carpeta de `components` una subcarpeta que llamaremos `counter` donde crearemos el fichero `Counter.tsx` que contenga el comportamiento del contador visto previamente.

Haciendo uso de los componentes de estilos de que hemos añadido (ReactNativeElements), nos debería quedaría algo similar a lo siguiente...

```
import { Button, Text } from 'react-native-elements';
import { View } from 'react-native';
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  const onIncrementCount = () => {
    setCount(count + 1);
  };

  return (
    <View>
      <Text>Current value: {count}</Text>
      <Button onPress={onIncrementCount} title="Increment Count" />
    </View>
  );
};

export default Counter;
```

Paso 2

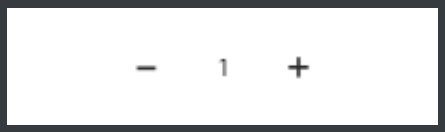
El siguiente paso será verlo en funcionamiento. Una vez creado el componente vamos a incluirlo en alguna de nuestras pantallas para poder visualizarlo. Podemos utilizar nuestro componente `Products` para invocar al componente `Counter` y así verlo en la página principal de la aplicación.

Paso 3

Añadir un segundo botón que nos permita hacer la acción inversa, es decir, que nos permita reducir el valor del contador. Una vez creado el botón, añadir la lógica necesaria para evitar números negativos y evitar reducir el valor del contador por debajo de 0.

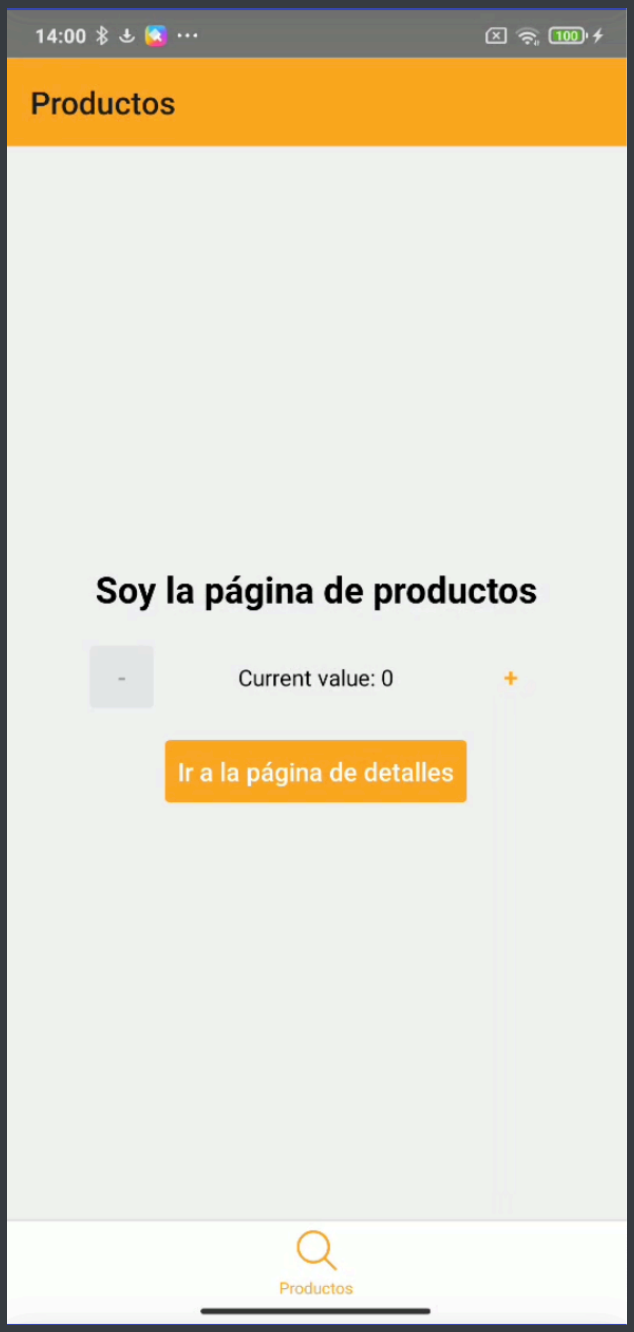
Paso 4

Haz unos últimos retoques y aplica unos cuantos estilos a nuestro contador para que se vea algo similar a esto:



Podemos aplicar estilos a nuestros componentes de la misma forma como hace el componente `Products` haciendo uso de `products-styles.ts`

Una vez terminado debería verse algo como esto...



Ejercicio 2 - Lista de productos

Este ejercicio consistirá en llenar la pantalla de productos con datos. Para ello realizaremos una petición HTTP para obtener los datos a mostrar y crear un componente que nos permita visualizarlos en pantalla.

Paso 1

Crearemos un nuevo servicio que se encargue de realizar la llamada de red para obtener los datos. Para ello, en nuestra carpeta `services` empezaremos creando un nuevo fichero que llamaremos `ProductsService.ts` y que se tendrá que ver algo así...

```
import axios from 'axios';

const ProductsService = {
  getProducts: () => {
    return
axios.get('https://raw.githubusercontent.com/RoiDopazo/resources/master/
react-tuto/data.json');
  }
};

export default ProductsService;
```

Básicamente, estamos definiendo un método dentro de ProductsService (getProducts) que se encarga de utilizar `axios` para realizar la petición de red al endpoint especificado.

Paso 2

En este paso trataremos de invocar el método que acabamos de crear. Como vimos anteriormente en la guía, el hook `useEffect` se usa normalmente para realizar peticiones asíncronas como son las llamadas de red. En este paso, crearemos un `useEffect` en nuestro `Products.tsx` que invoque dicho método y nos muestre en la consola los datos obtenido.

Al tratarse de una llamada asíncrona, necesitamos resolverla como tal. Echa un vistazo a como manejar funciones asíncronas en JavaScript en este link

```
// Dentro de nuestro useEffect podemos declarar una funcion asíncrona de la siguiente forma
const getAsyncProducts = async () => {
  const result = await {Funcion a invocar}.
  // Aquí tendríamos los valores retornados para la {Funcion a invocar} y ya los podríamos imprimir por pantalla.
}

// Una vez definida nuestra función, solo tendríamos que invocarla para que se ejecute.
getAsyncProducts();
```

> 🤔 Recuerda lo visto sobre useEffect y su segundo argumento (el array de dependencias). Cuantas veces nos interesa realizar esta petición? Cada vez que se produzca un re-renderizado? o solomante una vez?

Paso 3

Ahora somos capaces de mostrar nuestros productos por consola, el siguiente paso consistirá en guardarlos en un estado de React. Al guardarlos en un estado estaremos diciendole a React que se repinte automáticamente en cuanto modifiquemos ese estado.

Empezaremos creando un estado en nuestro `Products.tsx` que podremos llamar... `products` e inicializarlo por defecto como una lista vacía `[]`. Ahora, al terminar la petición web, en vez de mostrar los datos por consola, podemos almacenarlos en este nuevo estado y ver que se guardan correctamente.

Añade un console.log justo antes del return que imprima por consola el valor de dicho estado. Si todo va bien... debería mostrar dos logs:

- Primer renderizado -> `products = []`

- Segundo renderizado -> `products = [{...}]`

Paso 4

Ummm... los datos de los productos tardan unas milésimas de segundos en llegar, sería muy interesante poner un `loader` mientras no tenemos los datos. Vamos con ello.

Para empezar... necesitaremos un componente de `loador` , algo que nos indique que está cargando. Tenemos dos opciones, podemos crearlo nosotros mismos y hacerlo a nuestra idea y semejanza o, ya que hemos introducido React Native Elements, buscar en la documentación si existe algún componente que nos ofrezca lo que buscamos (si que existe) => https://reactnativeelements.com/docs/

Una vez tengamos nuestro componente de `loader` vamos a mostrarlo en la pantalla y ver como nos queda... podemos mostrarlo justo debajo del `Counter` en nuestra pantalla de `Products`

Ahora solo nos faltaría mostrarlo solo cuando no tenemos productos... Muy fácil! Con un simple `if` podemos hacer que se muestre una cosa u otra en función de nuestra variable `products` .

> 📌 La forma más sencilla sería comprobando su 'length'

Una vez implementado el `if` el flujo de React debería quedarnos más o menos así.

- Primer renderizado -> `products = []` -> mostramos únicamente el loader
- Segundo renderizado -> `products = [{...}]` -> mostramos la página de productos tal cual la teníamos (con el título, el counter y el botón)

Paso 5

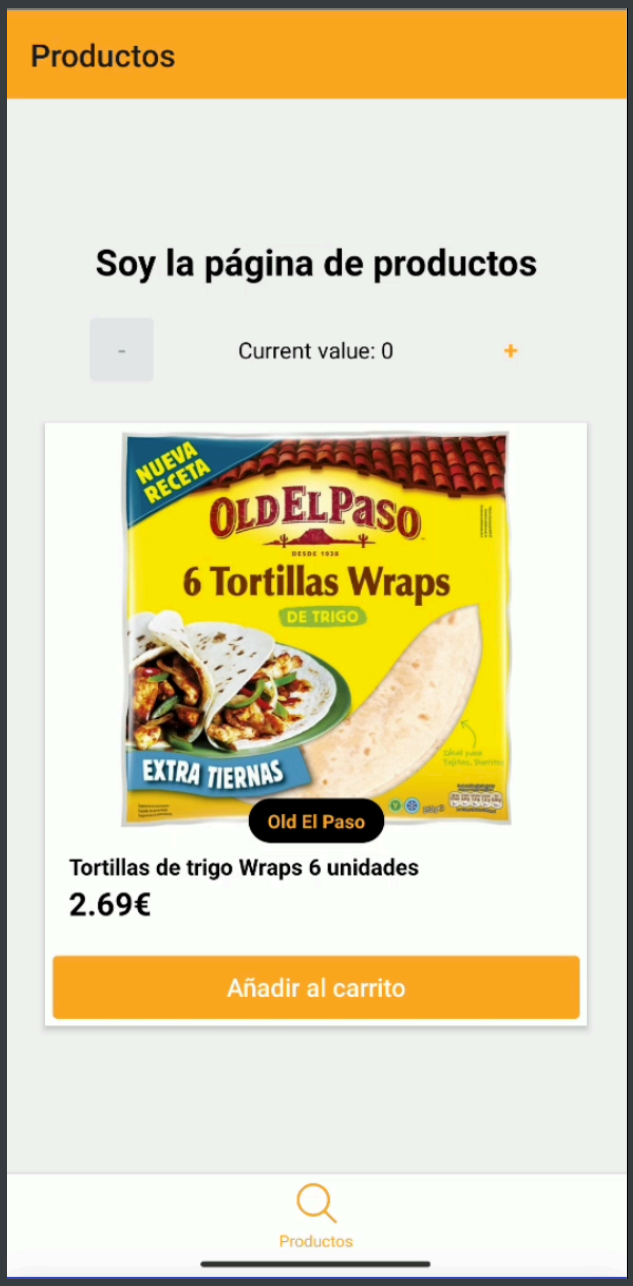
Tenemos el `loader` listo para mostrarse mientras no tenemos los datos, el siguiente paso consistirá en mostrar la lista de productos y nos preguntaremos... Cómo queremos que se vean? Qué datos voy a mostrar? Lo primero que debemos hacer es investigar un poco los datos que nos llegan y pensar un par de ideas para mostrarlo. Las soluciones típicas suelen ser en forma de `cards` o como elementos de una `lista`. Fuese lo que fuese la idea que tengas en mente, crea un componente (dentro de la carpeta `components` si, puesto que es algo que vamos a reusar) que nos sirva para mostrar un `product` (podemos llamarle `ProductItem`).

> 📝 Recuerda que estamos creando un componente lo más génerico y reutilizable posible. Si queremos mostrar en él el nombre del producto o su precio, la mejor forma sería pasándolo a través de las props.

Podemos probar cómo se vería ese componente llamándolo desde nuestro componente de `Products`.

```
return (
  ...
  {/* Podemos pasarle los datos de uno de nuestros productos a través de las props */}
  <ProductItem item={products[0]} />
  ...
);
```

Una posible forma de mostrar nuestros productos es la siguiente



Paso 6

Una vez tenemos un item creado solo nos faltaría hacer que se vieran todos. Para ello, emplearemos una flatList. El componente `flatList` se puede configurar fácilmente con tres elementos esenciales:

- `data` -> Array de datos.
- `renderItem` -> Componente React para indicar como se tiene que ver cada item
- `keyExtractor` -> Función para indicar a React como identificar únicamente cada item

de la lista (usado para optimizaciones propias de react).

Traduciéndolo a nuestra aplicacion...

- `data -> products`
- `renderItem -> ProductItem`
- `keyExtractor` -> (item) => item.name

Con todos los productos debería quedarnos algo así...

Ejercicio 3 - Creando la página de detalles

Paso 1

El pimer paso consistirá en enviar los datos del producto a la pantalla de detalles. Actualmente, en el componente `Products` tenemos un botón que nos permite viaja a la página de detalles pero ahora necesitamos que ese funcionalidad se aplica a cada producto que tenemos. Por ello, es necesario mover esa lógica al componente `ProductItem` que simboliza cada uno de los ítems.

Hacer que nuestro `ProductItem` sea clickable y nos redirija a la página de detalles. Para ello podemos utilizar un componente `<TouchableOpacity>` y añadir la función del click a través de la prop `onPress`.

Paso 2

El siguiente paso consistirá en enviar los datos del `item` que hemos clicado a componente de detalles. En la documentación de React Native Navigation podemos encontrar como hacerlo. Este sería un ejemplo:

```
const HomeScreen = ({ navigation }) => {
  return (
    <Button
      title="Go to Jane's profile"
      onPress={() =>
        navigation.navigate('Profile', { name: 'Jane' })
      }
    />
  );
};
```

Y en la página de detalles podemos acceder al elemento enviado a través de la prop `route`.

```
const ProfileScreen = ({ navigation, route }) => {
  return <Text>This is {route.params.name}'s profile</Text>;
};
```

Comprueba con un `console.log` que los datos recibidos son los correctos y corresponde al item que acabas de clickar.

Paso 3

Crea una interfaz para mostrar los datos del producto.

Debe quedar algo así...

Paso 4

Pues ya tenemos lista la pantalla de detalles pero en la barra superior nos aparece el texto de `product-details` que queda un poco raro. Tenemos varias opciones, una de ellas sería quitar la header y dejar únicamente el botón de `back` para volver a atrás y otra opción podría ser mostrar el nombre del articulo en esa cabecera.

Para mostrar el nombre del artículo podemos emplear la función que nos proporciona el `navigation` de React Navigation.

```
navigation.setOptions({ headerTitle: 'title' });
```

Podemos llamar a este método y pasarle el nombre de nuestro product haciendo uso de un `useEffect` . Una vez implementado, la cabecera debería verse tal que así.

Ejercicio 4 – Añadiendo productos a un carrito.

Paso 1

El objetivo de este ejercicio será incluir la funcionalidad de añadir productos a al carrito. Vamos empezar incluyendo el counter en cada unos de nuestros `ProductItem` que forma la lista de productos.

El item debería verse ahora tal que así.

Paso 2

Necesitamos tener un sitio donde poder guardar los elementos que vayamos añadiendo a la cesta, por lo que necesitaremos un estado.

Intenta hacer que nuestro componente `App />` imprima por consola el valor de este nuevo estado que tenemos que crear. Al principio estaría vacío => `[]` pero podemos utilizar el botón de `añadir a la cesta` para llenar esa lista (Por simplicidad, en este paso asumiremos que solo añadimos 1 item, por lo que podemos ignorar el valor del `Counter`

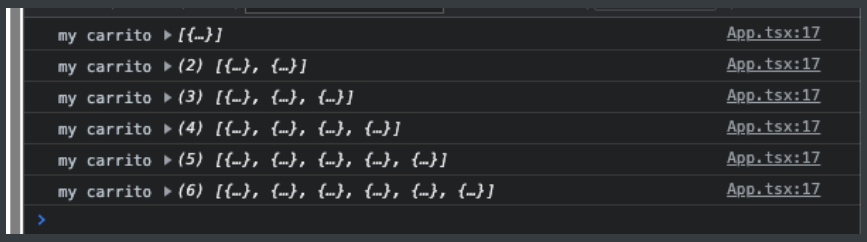
de momento).

Idealmente, cada vez que añadamos un item a la cesta, el componente `App` debe imprimir por pantalla el estado actualizado.

> 📝 Vas a tener que usar el hook useState para crear un estado nuevo y su función modificadora te permitirá actualizar su valor y así poder añadir a la lista.

Revisa la documentación para ver como pasar props a nuestros `Stact` screens => documentation

Por consola, cada vez que clickemos en el botón debería aparecernos lo siguiente:



Paso 3

Evitar que si añadimos al carrito el mismo producto varias veces se introduzca un nuevo elemento en la lista. La mejor opción pasaría por:

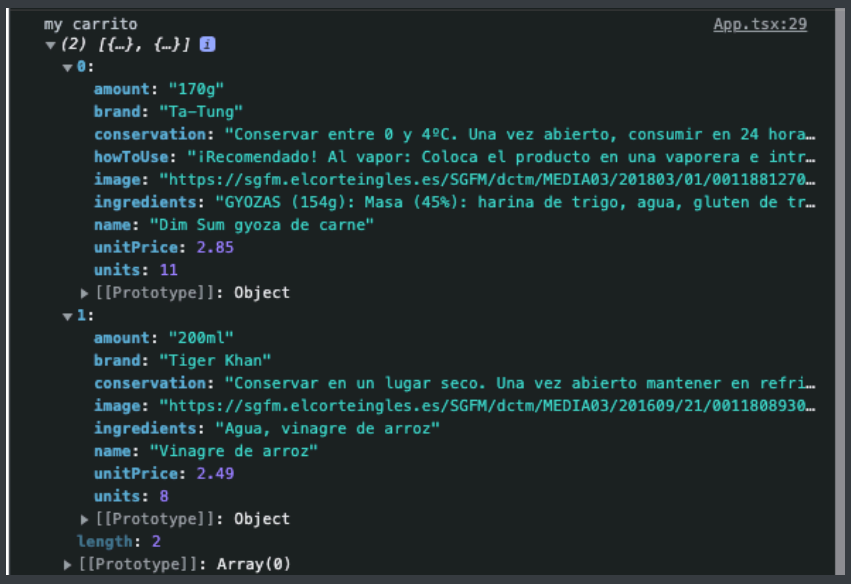
- Guardar un elemento nuevo junto con los datos del producto que nos diga cuantos hemos añadido, ejemplo:

```
{ brand: 'xxxx', name: 'yyyy', units: 1 }
```

- Cada vez que añadamos un elemento mirar si se encuentra ya en el carrito y
  - Si ya está -> incrementar el valor de `units` en 1.
  - Si no está -> añadirlo e incluir el valor del campo `units` a 1.

> 📝 Los Arrays JavaScript tienen un método find que puedes utilizar para resolver esto -> Array.find

Así es como debería verse por consola nuestro carrito.



Paso 4

En vez de hacer el incremento de 1 en 1, haz que el número de productos que queremos añadir sea el valor de nuestro `Counter`.

Modifica el `Counter` para que el valor mínimo que queremos que acepte sea 1 en vez de 0.

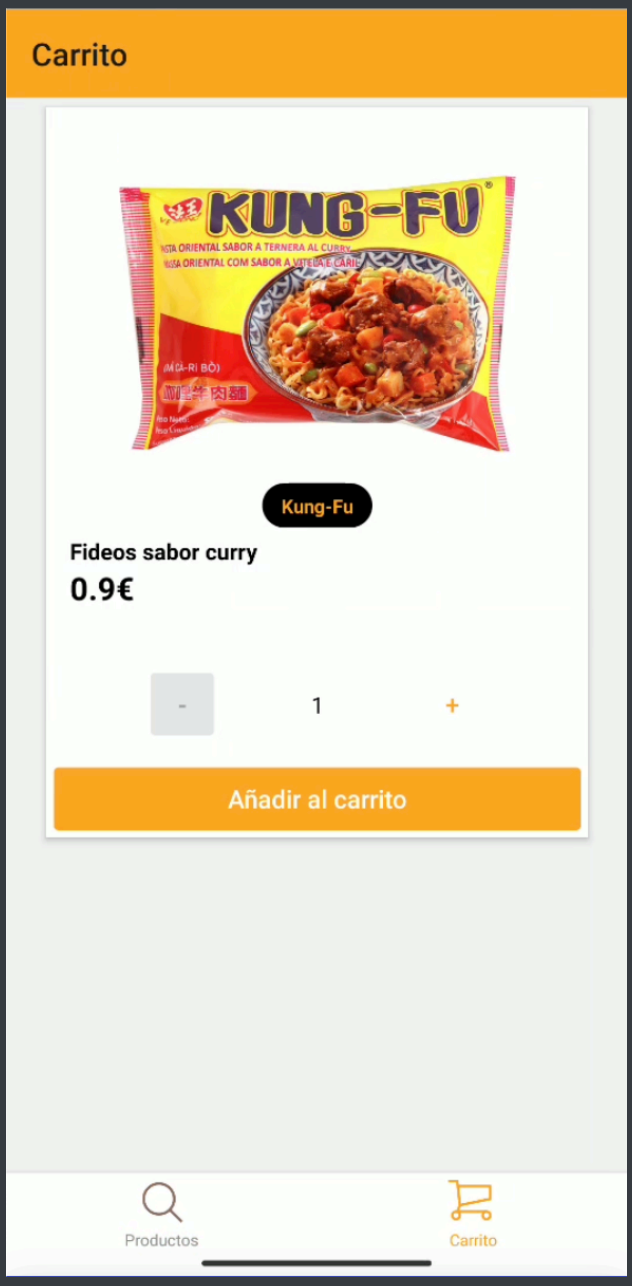
Ejercicio 5 - Ver mi carrito

Paso 1

Una vez que tengamos la lógica lista para añadir elementos al carrito, vamos crear una página que nos permite ver nuestro carrito actual.

Añade una nueva `Tab` junto con la de `Product` que sea para visualizar el carrito. Crea una lista en esa nueva página para mostrar (al igual que hacíamos en `Products`) la lista de ítems pero esta vez, nuestros datos serán los que nos lleguen del estado del carrito.

El resultado de esa página sería algo como la siguiente:



Paso 2

Vamos añadir la prop `isItemAddedToCart` al componente `ProductItem` para diferenciar los ítems que están añadidos al carrito de los que no. Con esta prop, podremos dar un estilo diferente a elementos que mostramos en la página del carrito haciendo que se vea algo así (el botón de eliminar lo implementaros luego).

> 📝 Esta prop nos sirve para modificar `ProductItem` de manera condicional, de tal forma que los elementos que vemos en `Products` queden tan cual estaban

Paso 3

Implementar la lógica de eliminar de la cesta.