Estados:

El estado (state) es una forma de guardar datos dentro de un componente para que recuerde información entre renderizados.

Sirve para almacenar y actualizar información que puede cambiar con el tiempo, como listas, formularios, contadores, etc. Cuando el estado cambia, el componente se vuelve a renderizar con la nueva información.

Se usa con el hook useState, así:
const [valor, setValor] = useState(valorInicial)

- **valor** es la variable con los datos actuales.
- **setValor** es la función para actualizar ese dato.
- **valorInicial** es el dato con el que empieza (por ejemplo, un número o un array vacío).

Ej: const [restaurants, setRestaurants] = useState([])

Prop:

Props (abreviatura de "properties") son datos que se pasan de un componente padre a un componente hijo. Sirven para que un componente reciba información desde otro.

Ej:

```
export default function RestaurantDetailScreen({ route }) {
  const { id } = route.params
  return (
    <View style={styles.container}>
      <TextRegular>Restaurant details. Id: {id}</TextRegular>
    </View>
  )
}
```

- Aquí route es una *prop* que recibe el componente RestaurantDetailScreen.
- Esta *prop* viene automáticamente al navegar desde otro componente usando React Navigation.
- Dentro de route está params, donde se guardan los datos que se pasaron (como el id del restaurante seleccionado).

Hooks:

Los **hooks** son funciones especiales que nos permiten usar funcionalidades de React (como el estado o el ciclo de vida) dentro de componentes **funcionales** (sin tener que usar clases).

Nos ayudan a:

1. **Guardar y actualizar información en el componente**.
2. **Ejecutar código automáticamente cuando los datos cambian o el componente se monta**.
3. **Compartir datos entre varios componentes (con contextos)**.

useState:

Es como la **memoria interna del componente**, donde puedes guardar datos que pueden cambiar, como un número, un texto o una lista.

Ej:

```
const [state, setState] = useState(valorInicial)
```

- state: el valor actual.
- setState: una función que te permite **cambiar ese valor**.
- valorInicial: el valor con el que empieza el estado.

Para cambiar el valor del state puedes usar setEstate.

Ej:

```
const [contador, setContador] = useState(0)
setContador(contador + 1)
```

useEffect:

Te permite **ejecutar código automáticamente** cuando:

- El componente se monta.
- Algún dato (estado o prop) cambia.

```
useEffect(() => {
  // Código que quieres ejecutar
}, [dependencias])
```

- [ ]: se ejecuta **solo una vez**, cuando el componente aparece por primera vez.
- [*dependencias*]: se ejecuta cada vez que *dependencias* cambie.
- Si **no pones el arreglo**, se ejecuta **cada vez que el componente se actualiza**.

useContext:

useContext es un **hook de React** que permite acceder al valor de un **contexto** desde un componente funcional. Un contexto es una herramienta que React proporciona para **compartir datos entre componentes sin necesidad de pasar props manualmente** por cada nivel del árbol de componentes.

useContext te permite **leer el valor actual** de un contexto React que ha sido proporcionado previamente usando un Context.Provider.

FlatList:

FlatList es un **componente incorporado** en React Native que se utiliza para **mostrar listas de datos de forma eficiente**.
Está optimizado para mostrar listas largas, renderizando solo los elementos visibles en pantalla y cargando más a medida que haces scroll.

```
<FlatList
  data={arrayDeDatos}
  renderItem={funciónParaRenderizarCadaElemento}
  keyExtractor={funciónParaObtenerClaveDeCadaElemento}
/>
```

- Data → El array de objetos que quieres mostrar.
- renderItem → Una función que define cómo se ve cada ítem de la lista. Recibe un objeto con una propiedad item (el elemento actual).
- keyExtractor → Una función que devuelve una clave única para ítem (React la necesita para identificar los elementos). Normalmente se usa el id.

EJ:

```
const datos = [
  { id: 1, nombre: 'Restaurante A' },
  { id: 2, nombre: 'Restaurante B' }
]

const renderElemento = ({ item }) => (
  <Text>{item.nombre}</Text>
)

<FlatList
  data={datos}
  renderItem={renderElemento}
  keyExtractor={item => item.id.toString()}
/>
```

Esto muestra una lista con los nombres "Restaurante A" y "Restaurante B".

Navigation y route:

```
export default function RestaurantDetailScreen({ navigation, route }) {
```

- navigation: para volver atrás o ir a otras pantallas.
- route: contiene **los parámetros que le pasaron** cuando se navegó a esta pantalla.

Por ejemplo:

```
navigation.goBack()
```

Para ir atrás

Y en route, para acceder a los datos, debes poner → **route.params**

Componentes visuales:

```
<DropDownPicker
  open={open}
  value={values.restaurantCategoryId}
  items={restaurantCategories}
  setOpen={setOpen}
  onSelectItem={ item => {
    setFieldValue('restaurantCategoryId', item.value)
  }}
  setItems={setRestaurantCategories}
  placeholder="Select the restaurant category"
  containerStyle={{ height: 40, marginTop: 20 }}
  style={{ backgroundColor: GlobalStyles.brandBackground }}
  dropDownStyle={{ backgroundColor: '#fafafa' }}
  />
```

- setOpen={setOpen}: función para cambiar el estado de open (abrir/cerrar el desplegable).
- onSelectItem={ item => { setFieldValue('restaurantCategoryId', item.value) }}:
  Cuando el usuario selecciona una opción, esta función actualiza el valor del campo restaurantCategoryId dentro del formulario de **Formik**.
- setItems={setRestaurantCategories}: permite actualizar la lista de categorías (por ejemplo, si se cargan dinámicamente desde el backend).
- placeholder="Select the restaurant category": texto que aparece cuando no hay ningún valor seleccionado.
- containerStyle={{ height: 40, marginTop: 20 }}: estilo del contenedor del dropdown.
- style={{ backgroundColor: GlobalStyles.brandBackground }}: estilo del campo del dropdown en sí.
- dropDownStyle={{ backgroundColor: '#fafafa' }}: estilo de la lista desplegable (cuando se abre).

```
<TextRegular style={styles.switch}>Is it available?</TextRegular>
<Switch
  trackColor={{ false: GlobalStyles.brandSecondary, true: GlobalStyles.brandPrimary }}
  thumbColor={va (property) availability: null .brandSecondary : '#f4f3f4'}
  value={values.availability}
  style={styles.switch}
  onValueChange={value =>
    setFieldValue('availability', value)
  }
/>
```

- style={styles.switch}: aplica un estilo definido para dar formato (por ejemplo, márgenes) al texto que acompaña al interruptor.
- "¿Está disponible?": texto que se muestra al usuario para indicar qué representa el interruptor.
- trackColor={{ false: GlobalStyles.brandSecondary, true: GlobalStyles.brandPrimary }}: define el color del fondo del switch cuando está desactivado (false) y cuando está activado (true), usando estilos personalizados.
- thumbColor={values.availability ? GlobalStyles.brandSecondary : '#f4f3f4'}: establece el color del botón (círculo deslizante del switch). Si el producto está disponible (availability es true), el color será brandSecondary; si no, será gris claro.
- value={values.availability}: enlaza el estado del switch con el campo availability del formulario de Formik, mostrando si el producto está disponible o no.
- style={styles.switch}: aplica estilos al switch (por ejemplo, márgenes), definidos en la hoja de estilos.
- onValueChange={value => setFieldValue('availability', value)}: cuando el usuario cambia el estado del switch, esta función actualiza el campo availability en el formulario gestionado por Formik.

```
<Formik
  validationSchema={validationSchema}
  initialValues={initialRestaurantValues}
  onSubmit={createRestaurant}>
  {({ handleSubmit, setFieldValue, values }) => (
    <ScrollView>
      /* Your views, form inputs, submit button/pressable */
    </ScrollView>
  )}
</Formik>
```

- validationSchema: el esquema de validación con Yup.
- initialValues: los valores iniciales de los campos.
- onSubmit: la función a ejecutar si la validación es correcta.
- handleSubmit: dispara la validación y se llama al pulsar el botón de enviar.
- values: contiene los valores actuales del formulario.
- setFieldValue: permite actualizar manualmente el valor de un campo (útil para selectores de imagen o dropdowns).

```
<DeleteModal
  isVisible={restaurantToBeDeleted !== null}
  onCancel={() => setRestaurantToBeDeleted(null)}
  onConfirm={() => removeRestaurant(restaurantToBeDeleted)}
>
  <TextRegular>Los productos de este restaurante también serán eliminados</TextRegular>
  <TextRegular>Si el restaurante tiene pedidos, no podrá ser eliminado.</TextRegular>
</DeleteModal>
```

El componente DeleteModal necesita tres propiedades:

- isVisible: una expresión booleana que se evalúa para mostrar u ocultar la ventana modal. En este caso solo se mostrara el deleteModal si restaurantToBeDeleted contiene algo, es decir, es diferente de null.
- onCancel: la función que se ejecutará cuando el usuario pulse el botón de cancelar.
- onConfirm: la función que se ejecutará cuando el usuario pulse el botón de confirmar.