Programación Multimedia y Dispositivos Móviles (UP3)

UP3: Tipos de navegación y estilos

Autores: Antonio Calabuig Puigvert y Sebastián Villa Ponce

Centro: IES Salvador Gadea

Departamento: Informática

Ciclo: Desarrollo de Aplicaciones Multiplataforma

Fecha: 29/10/2025

UP3: Tipos de navegación y estilos
1. Introducción y base teórica
2. Navigation-app
    2.1. Creación de proyecto “limpio”
    2.2. Instalación de NativeWind
    2.3. Uso de fuentes y colores personalizadas en NativeWind
    2.4. Navegación entre pantallas
    2.5. CustomButtons y Links AsChild
    2.6. Variantes de componentes
    2.7. Navegación con Stack (StackNavigation)
    2.8. Personalización del Stack de navegación
    2.9. Listado de productos y su detalle

1. Introducción y base teórica

A lo largo de esta práctica vamos a trabajar los siguientes elementos:

1. Nativewind: **NativeWind** es una librería que permite usar clases de **Tailwind CSS** en React Native.

   Esto facilita el diseño visual de las interfaces con una sintaxis rápida y familiar.

2. StackNavigation: El **Stack Navigation** organiza las pantallas como una pila (stack).
   Cada nueva pantalla se “apila” sobre la anterior, y al volver atrás se “desapila”.

3. Enviar argumentos entre páginas: Consiste en pasar datos de una pantalla a otra mediante el sistema de navegación.

```
// Pantalla A
navigation.navigate('Perfil', { nombre: 'Laura' });

// Pantalla B
const route = useRoute();
<Text>{route.params.nombre}</Text>
```

4. Botones personalizados: En lugar de usar el Button básico de React Native, se crean componentes propios con estilos y comportamiento personalizados.

5. Estructura de directorios: Crearemos una estructura de directorios como la siguiente:

```
project/
├── assets/        → Imágenes, fuentes
├── components/    → Componentes reutilizables
├── screens/       → Pantallas principales
├── navigation/    → Configuración de rutas
├── theme/         → Colores, estilos globales
├── utils/         → Funciones auxiliares
└── App.tsx
```

6. Temas y fuentes personalizadas: Permiten mantener coherencia visual y accesibilidad en toda la app.
7. Colores personalizados: Definir una paleta propia ayuda a reforzar la identidad visual.
8. Múltiples layouts: Los layouts son estructuras base que comparten varias pantallas.

```
layouts/
 ├── MainLayout.tsx  → Con header y tabs
 ├── AuthLayout.tsx  → Solo contenido central
```

9. El fichero `_layout`:

> En proyectos creados con **Expo Router**, cada carpeta dentro de `app/` puede tener un archivo especial llamado **`_layout.tsx`**.
>
> Ese archivo define el **layout base** o **estructura de navegación** que se aplica a todas las pantallas de esa carpeta.
>
> En otras palabras, `_layout.tsx` actúa como un **envoltorio (wrapper)** para todas las páginas "hijas" del mismo nivel.
> Un `_layout.tsx` suele contener el **navegador principal** (por ejemplo, un `Stack` o `Tabs`) y envoltorios de estilo o tema.
>
> Ejemplo de estructura visual:
>
> ```
> app/
>  ├── _layout.tsx       ← Layout principal
>  ├── index.tsx         ← Pantalla de inicio
>  ├── productos/
>  │    ├── _layout.tsx   ← Layout específico para productos
>  │    ├── index.tsx
>  │    └── detalle.tsx
>  ├── perfil.tsx
>  └── ajustes.tsx
> ```

> - El `app/_layout.tsx` define el **navegador raíz (Root Stack / Tabs / Drawer)**.
> - El `app/productos/_layout.tsx` puede definir un **navegador secundario** o un estilo particular solo para esa sección.

2. Navigation-app

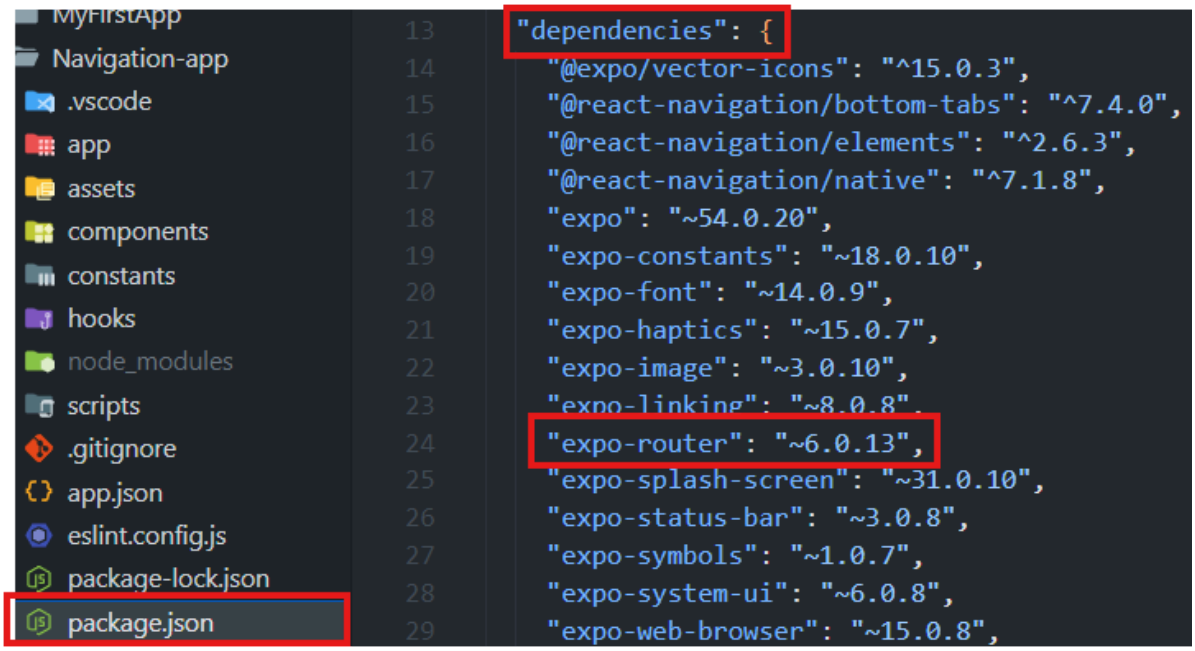
2.1. Creación de proyecto “limpio”

Vamos a empezar el proyecto entrando a la carpeta que va a contener nuestro proyecto en **React Native + Expo**. Para ello vamos a realizar los siguientes pasos:

1. Abrimos VSC en la ruta de la carpeta donde estará el proyecto y lanzamos el comando:

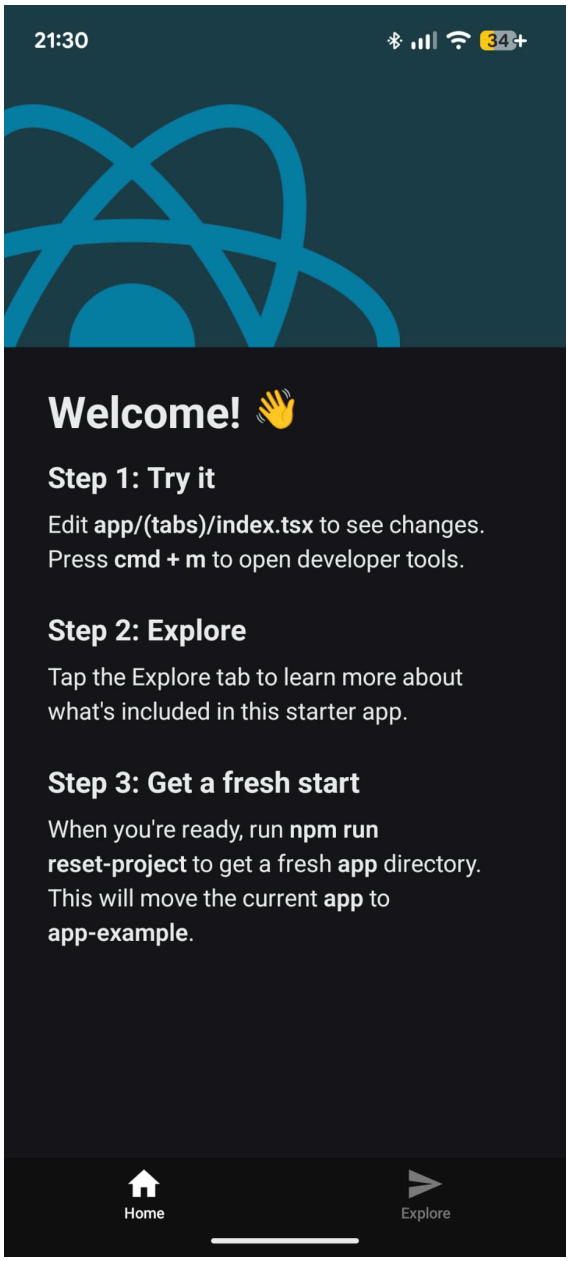
   ```
   npx create-expo-app@latest Navigation-app
   ```

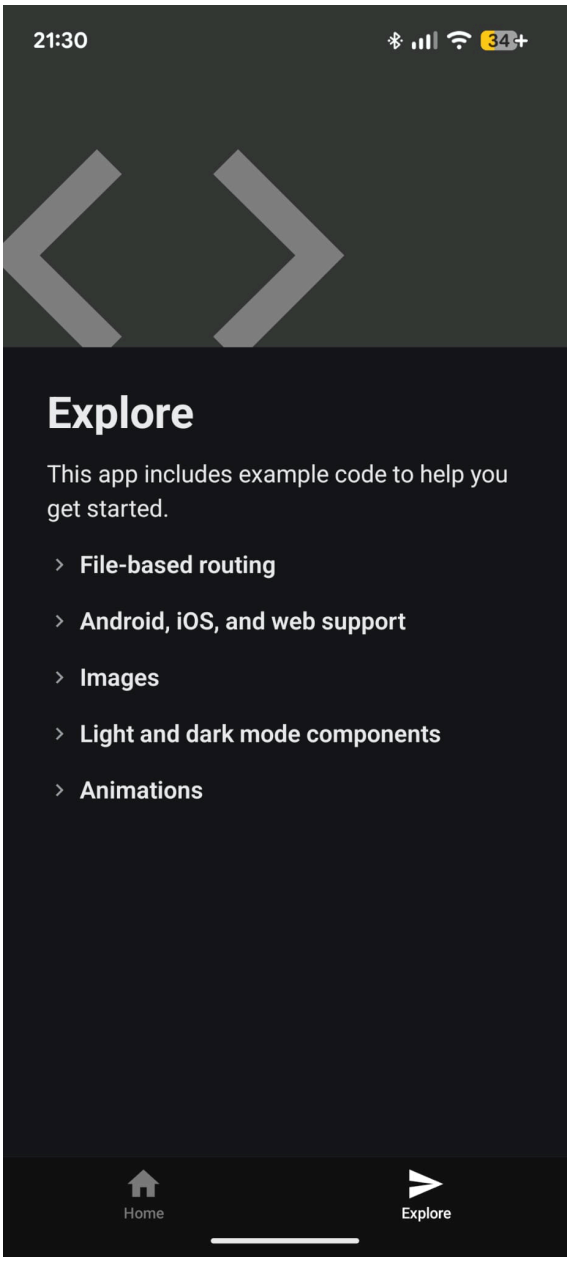
2. Accedemos al fichero `package.json` y comprobamos que dentro debería estar dentro de `dependencies` nuestra librería `expo-router` con la última versión:



3. Lanzamos la aplicación con `npm start` .

4. Al iniciar la aplicación veremos que nos salen dos pantallas como las siguientes



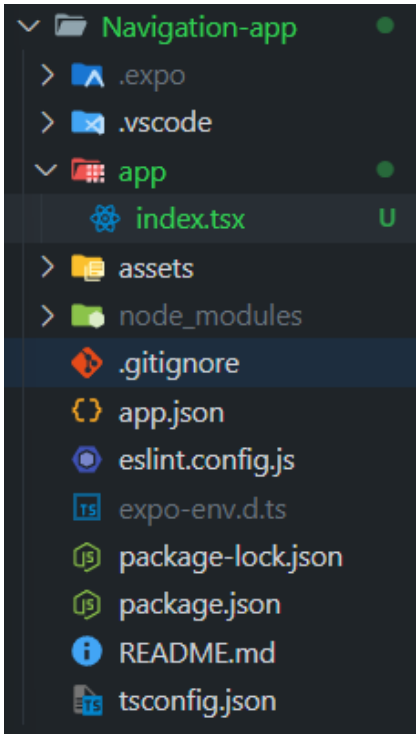


5. De cara a crear nuestro proyecto de 0 tenemos que eliminar muchas cosas y preparar la estructura, así que empezamos.
6. Borramos todo el contenido de la carpeta `app` .
7. Creamos un `index.tsx` dentro de la carpeta `app` .

8. Ejecutamos el snippet `rnfe` dentro del `index.tsx` .
9. Cambiamos el nombre del componente de `index` a `App` .
10. Cambiamos el texto a App.
11. Cambiamos la vista a una segura y comprobamos que se muestra la pantalla en blanco con nuestro texto `App` .
12. El código debería quedar así:

```
import { SafeAreaView } from 'react-native-safe-area-context';
import { Text } from 'react-native'
const App = () => {
  return (
    <SafeAreaView>
      <Text>App</Text>
      <StatusBar style="dark"/>
    </SafeAreaView>
  )
}
export default App;
```

13. Las carpetas de componentes, constantes, hooks y scripts las borramos. Más adelante si son necesarias, que lo serán las volveremos a crear.
14. El directorio del proyecto se debería de ver así:



> La carpeta app es directorio algo especial, salvo nombres reservados como `index.tsx` , todos **los ficheros** `.tsx` **que estén dentro de esta carpeta serán** lo que llamaremos **"pantallas" o "screens"**, es decir, las **vistas que contendrán los componentes** y elementos que aplicarán la funcionalidad. De esta forma **separamos** lo que será **vista de componente**, **aplicando** adecuadamente el **paradigma** de programación orientado a componentes.

15. Ya tenemos nuestro proyecto "limpio" para empezar. Nos saldrá la pantalla en blanco con el texto App a la izquierda arriba.
16. Vamos a utilizar NativeWind para trabajar en este proyecto, por lo que vamos a instalar el plugin "Tailwind CSS IntelliSense" en caso de no tenerlo instalado. Con este plugin tendremos disponible las clases con el control + espacio a la hora de trabajar en el código.

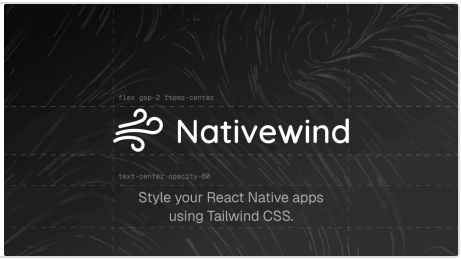
2.2. Instalación de NativeWind

1. Vamos a la página de NativeWind y seguimos la documentación de instalación:

Nativewind

Use Tailwind in React Native.

https://www.nativewind.dev/



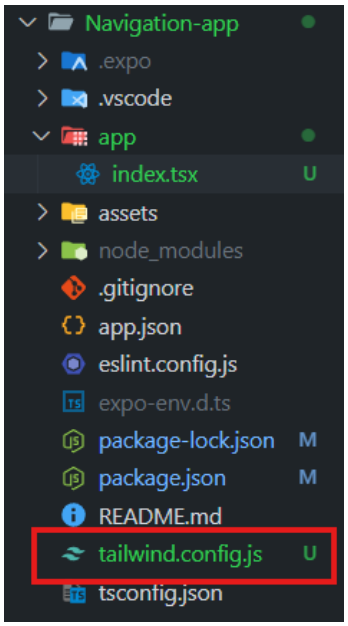
> En el caso de querer crear un NUEVO PROYECTO desde cero CON EXPO + NATIVEWIND podemos hacerlo directamente con el comando `npx rn-new --nativewind` , en nuestro caso el proyecto ya está creado por lo que haremos la instalación manual.

2. Lanzamos los siguientes comandos para instalar la librería y sus dependencias (podemos usar `npm` o Expo con `npx`):

```
npm install nativewind react-native-reanimated@~3.17.4 react-native-safe-area-context@5.4.0
npm install --dev tailwindcss@^3.4.17 prettier-plugin-tailwindcss@^0.5.11
```

```
PS D:\RN\Navigation-app> npm install nativewind react-native-reanimated@~3.17.4 react-native-safe-area-context@5.4.0

added 32 packages, removed 1 package, changed 2 packages, and audited 988 packages in 14s

182 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\RN\Navigation-app> npm install --dev tailwindcss@^3.4.17 prettier-plugin-tailwindcss@^0.5.11
npm warn config dev Please use --include=dev instead.

added 2 packages, and audited 990 packages in 8s

183 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\RN\Navigation-app>
```

3. Lanzamos `npx tailwindcss init` para crear el fichero `tailwind.config.js` .
4. Comprobamos que se ha realizado con éxito:




```
PS D:\RN\Navigation-app> npm install nativewind react-native-r
eanimated@~3.17.4 react-native-safe-area-context@5.4.0

added 32 packages, removed 1 package, changed 2 packages, and
audited 988 packages in 14s

182 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\RN\Navigation-app> npm install --dev tailwindcss@^3.4.17
 prettier-plugin-tailwindcss@^0.5.11
npm warn config dev Please use --include=dev instead.

added 2 packages, and audited 990 packages in 8s

183 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\RN\Navigation-app> npx tailwindcss init

Created Tailwind CSS config file: tailwind.config.js
PS D:\RN\Navigation-app>
```

5. Añadimos las líneas de `content y pressets` o directamente sustituimos el fichero entero y guardamos:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  // NOTE: Update this to include the paths to all files that contain Nativewi
nd classes.
  content: ["./App.tsx", "./components/**/*.{js,jsx,ts,tsx}"],
  presets: [require("nativewind/preset")],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

6. Añadimos otros elementos más a la lista de `content` , crearemos las estructuras más adelante:

```
content: [
    "./App.tsx",
    "./app/**/*.{js,jsx,ts,tsx}",
    "./components/**/*.{js,jsx,ts,tsx}",
    "./presentation/**/*.{js,jsx,ts,tsx}"
  ],
```

> TailWind se va a aplicar únicamente en cualquier componente que se encuentre dentro de los directorio que hemos especificado en `content` .

7. Dentro de la carpeta `app` creamos un fichero `global.css` que tendrá los estilos globales de la aplicación.

> Lo que recomienda la documentación es tenerlo en raíz pero lo pondremos en `app` para no juntarlo ni confundirlo con los archivos de configuración que hay en el directorio raíz.

8. Dentro del fichero `global.css` añadimos las directivas:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

9. Creamos el fichero `babel.config.js` en el directorio raíz:

```
module.exports = function (api) {
  api.cache(true);
  return {
    presets: [
      ["babel-preset-expo", { jsxImportSource: "nativewind" }],
      "nativewind/babel",
    ],
  };
};
```

10. Creamos el fichero `metro.config.js` en el directorio raíz con el código:

```
const { getDefaultConfig } = require("expo/metro-config");
const { withNativeWind } = require('nativewind/metro');

const config = getDefaultConfig(__dirname)

// Por defecto viene ./global.css pero el nuestro esta dentro de app
module.exports = withNativeWind(config, { input: './app/global.css' })
```

11. Creamos el fichero `_layout.tsx` dentro de la carpeta `app`.

> Los ficheros `_layout.tsx` son ficheros que actúan en una sección o subsección local y aplica estilos o recursos a todo su ámbito de trabajo (directorio local)

12. Creamos el componente principal con nombre `RootLayout` usando `rnfe` .
13. Cambiamos el nombre por defecto a `RootLayout` .
14. Importamos el fichero `global.css` en el fichero `_layout.tsx` y modificamos el componente de salida por `<Slot/>`:

```
// app/_layout.tsx
import "./global.css";
import { Slot } from 'expo-router';
const RootLayout = () => {
 return <Slot/>;
}
export default RootLayout;
```

> Tenemos en cuenta que en esta ocasión nuestro `global.css` está dentro de la misma carpeta que `index.tsx y _layout.tsx` pero si no lo estuviera tendríamos que poner la ruta relativa adecuada para alcanzarlo. En lugar de `./` habría que subir `../` o bajar `./carpeta/` .

15. Comprobamos que tras reiniciar el servidor si no recarga bien, la aplicación debería de funcionar y estar casi lista.

16. Accedemos al `app.json` y encendemos el “bundler” de Metro (es el empaquetador de código JS) dentro de la propiedad expo (expo → web → bundler):

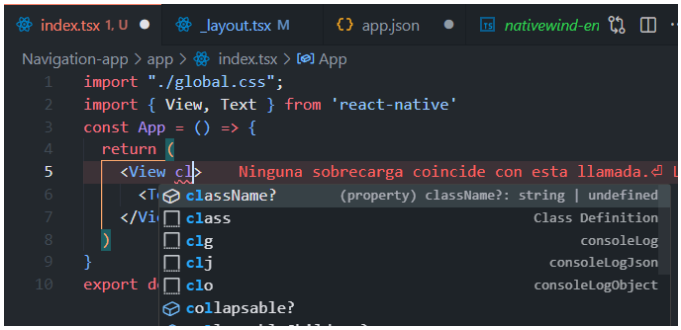
```
"web": {
  "output": "static",
  "favicon": "./assets/images/favicon.png",
  // Agregado manualmente
  "bundler": "metro"
}
```

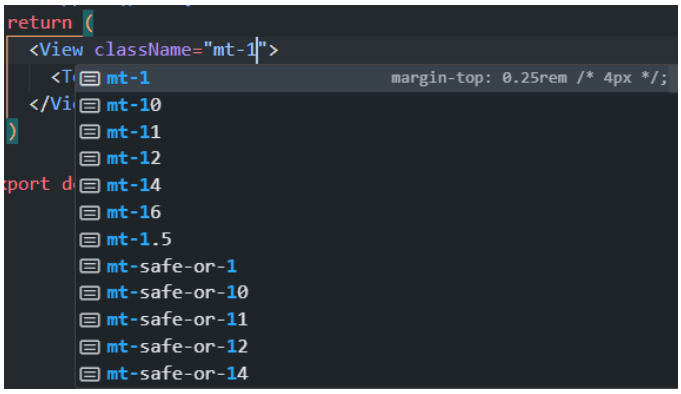
17. Para implementar React Native con TypeScript comprobamos que tenemos el fichero `nativewind-env.d.ts` creado en la raíz y que contiene la siguiente directiva con triple barra:

```
/// <reference types="nativewind/types" />
```

> Con esta directiva lo que conseguimos es habilitar el uso de `className` entre otras muchas propiedades necesarias para trabajar.

18. Vamos a comprobar que todo funciona. Vamos al fichero `index.tsx` y dentro del `<SafeAreaView>` deberíamos poder autocompletar el `className` .





19. Probamos a añadir al texto las siguientes clases:

```
// SafeAreaView
className="mt-10 ml-10"
// Text
className="text-3xl font-bold text-blue-500"
```

20. Paramos el servidor y lo relanzamos con `npm start -c` para activarlo limpiando caché ya que debe comprobar todas las dependencias y nuevos ficheros que hemos creado.

21. Deberíamos tener algo similar a esto:

2.3. Uso de fuentes y colores personalizadas en NativeWind

Vamos a incorporar y trabajar con varias fuentes y colores en este proyecto, Vamos con el proceso:

1. Descargamos las fuentes, estarán disponibles en Aules junto la práctica.
2. Incorporamos 3 de todas las fuentes en nuestra carpeta `fonts` (Black, Light y Medium)
3. Vamos al fichero `tailwind.config.js` y ponemos el siguiente código dentro de la propiedad `extend`:

```
//La propiedad fontFamily acepta un objeto con diferentes fuentes
// Cada fuente con el nombre de propiedad que deseamos que tenga en el proyecto
// tiene una lista de 2 elementos: Fuente a poner y fuente en caso de fallo
fontFamily: {
    'work-black': ['WorkSans-Black', 'sans-serif'],
    'work-light': ['WorkSans-Light', 'sans-serif'],
    'work-medium': ['WorkSans-Medium', 'sans-serif'],
}
```

4. Vamos a nuestro fichero `_layout.tsx` y dentro del componente importamos las fuentes mediante este código:

```
//Importamos la variable con las fuentes y la variable con el posible error de carga
const [fontsLoaded, error] = useFonts({
  'WorkSans-Black': require('../assets/fonts/WorkSans-Black.ttf'),
  'WorkSans-Light': require('../assets/fonts/WorkSans-Light.ttf'),
  'WorkSans-Medium': require('../assets/fonts/WorkSans-Medium.ttf'),
});
```

5. Para evitar que hayan problemas de desincronización debido a una carga lenta o fallo, vamos a mostrar un `Splash` que se cargará antes que las fuentes:

```
SplashScreen.preventAutoHideAsync();
```

6. Para quitar el Splash de carga creamos un efecto que tenga dependencia de las variables que estamos cargando.

```
//Nuestros disparadores de eventos son [fontsLoaded, error]
// Si hay un error, propagamos el error
//Si se cargan las fuentes se quita el Splash
useEffect(() => {
  if(error) throw error;
  if(fontsLoaded) SplashScreen.hideAsync();
}, [fontsLoaded, error]);
```

7. En caso de que no haya ni error ni carga, se evita la devolución del componente devolviendo `null` como en el código:

```
if(!fontsLoaded && !error) return null;
```

8. Comprobamos que no tenemos ningún error y nuestra aplicación funciona.
9. Vamos al fichero `index.tsx` y copiamos nuestro texto 3 veces para probar las 3 fuentes diferentes.

```
// Aplicar mediante style
<Text
style={{fontFamily: 'WorkSans-Black'}}
className="text-4xl">
  Forma 1
</Text>
// Aplicar mediante className
<Text
className="font-work-medium text-3xl">
  Forma 2
</Text>
<Text className="font-work-light text-2xl">
  Forma 3
</Text>
```

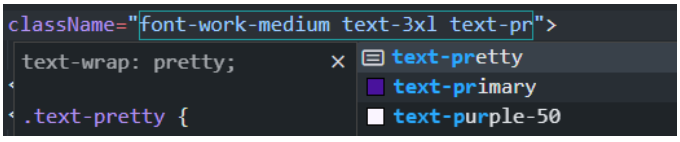
> En caso de NO CARGA recargamos la aplicación, si sigue sin cargar, paramos y recompilamos la aplicación limpiando caché.

10. El resultado sería:

11. Ya tenemos nuestras fuentes personalizadas, vamos con los colores. Volvemos a `tailwind.config.js` y en el mismo nivel de `fontFamily` ahora creamos la propiedad `colors` :

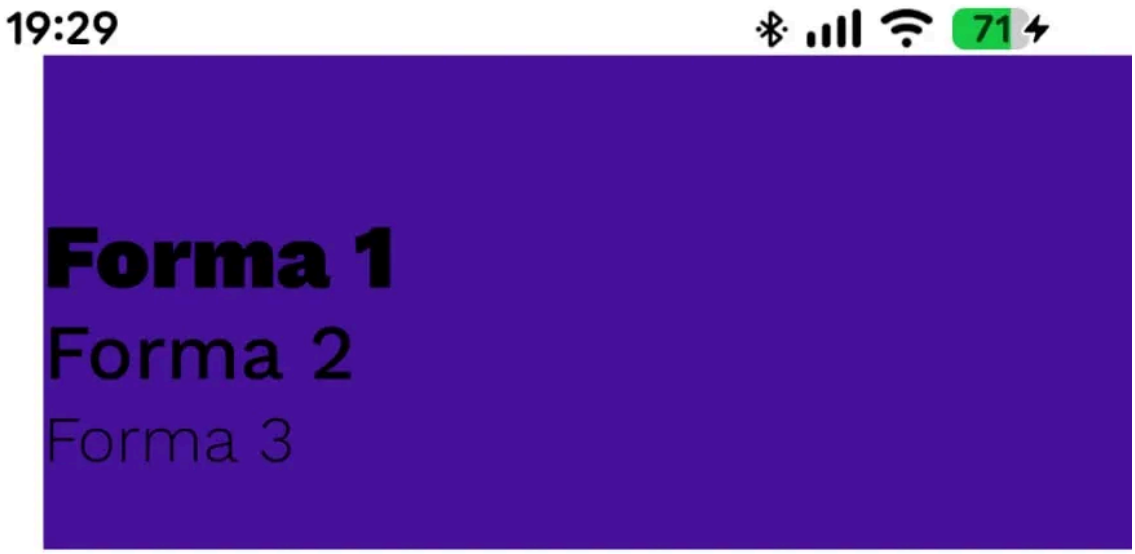
```
colors: {
  primary: '#49129C',
},
```

12. Comprobamos que se puede poner el color poniendo en el `className` utilizado en los textos de `index.tsx` el valor `text-primary` .



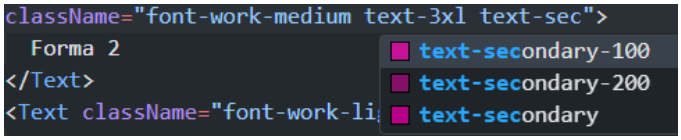
13. La propiedad de color no es solo aplicable a textos, cualquier elemento “pintable” como el fondo puede recibir esta propiedad pero con su respectivo prefijo autogenerado:

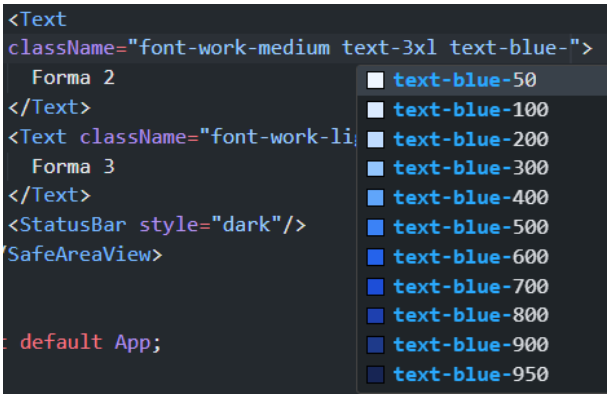
```
<SafeAreaView className="mt-10 ml-10 bg-primary">
```



14. Completamos la paleta de colores en el `tailwind.config.js` y la probamos:

```
colors: {
  primary: '#49129C',
  secondary: {
    DEFAULT: '#B40086',
    100: '#C51297',
    200: '#831266',
  },
  tertiary: '#EF2967',
},
```



> Si dentro de una propiedad de color entregamos un objeto en lugar de un color, podemos definir las variantes de color como cuando hemos usado las variantes del `text-blue`:
>
> 
>

15. El resultado sería:

2.4. Navegación entre pantallas

Vamos a comenzar la creación de varias pantallas, su cambio y comunicación entre ellas.

1. Dentro de `app` creamos el nuevo fichero `products.tsx` , usamos `rnfe` y cambiamos el nombre de componente y su exportación a `ProductsScreeen` .

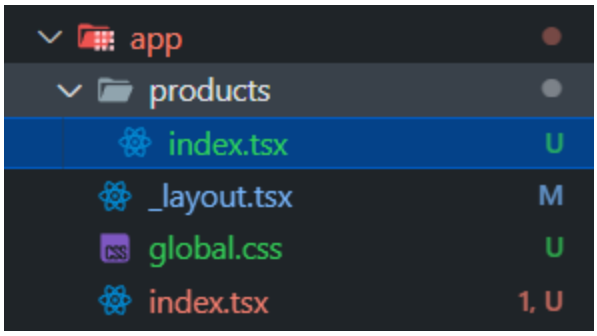
```
import { View, Text } from 'react-native'
const ProductsScreeen = () => {
  return (
    <View>
      <Text>ProductsScreen</Text>
    </View>
  )
}
export default ProductsScreeen;
```

2. Volvemos a `index.tsx` y creamos un nuevo elemento `<Link>` de `expo-router` :

```
<Link href='/products'>
</Link>
```

> Cuando vamos a introducir el `href` nos deberían salir todas las rutas disponibles, se decir, aquellas creadas en app con su componente `<Slot>` asociado.

3. Si pulsamos al texto vemos que nos cambia de pantalla de `ProductsScreen` pero tenemos los mismos problemas de visualización al usar un `<View>` por defecto en lugar de un `<SafeAreaView>` .
4. Para mantener el orden en las aplicaciones cuando estas vayan creciendo debemos tener una estructura de carpetas, en nuestro caso vamos a crear `app/products` y dentro vamos a mover el fichero de `products.tsx` que se va a convertir en el fichero principal de nuestra sección, debido a que será el fichero principal de una sección o subsección le cambiamos el nombre a `index.tsx` para que Expo lo busque y ejecute por defecto.

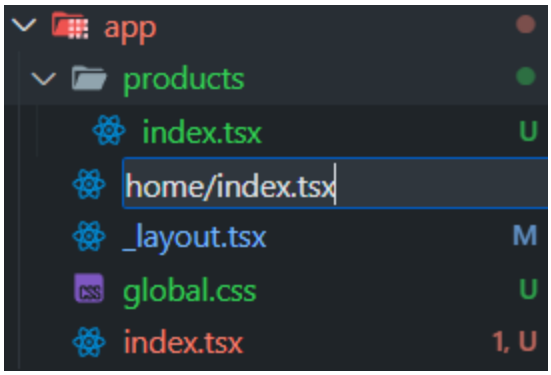


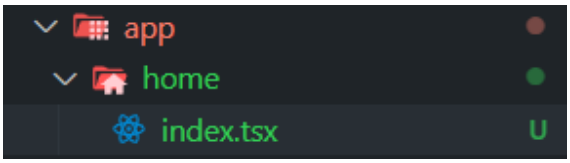
5. Hasta este momento hemos usado el fichero `index.tsx` de `app` como un fichero renderizador más, pero esa no es su función. Así que vamos a comentar todo lo realizado anteriormente en el fichero principal y vamos a empezar su verdadera construcción. Ponemos el siguiente componente en el índice principal:

```
return <Redirect href="/products/index"/>
```

Tras hacer esto, al recargar nuestra aplicación veremos que directamente accedemos a nuestro componente `products.tsx` .

6. Vamos a crear otra pantalla, para ello creamos un nuevo fichero llamado `home/index.tsx` y veremos como se nos crea la estructura automáticamente.

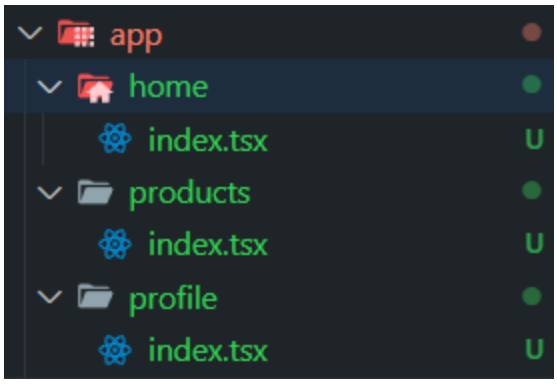




7. Creamos nuestro componente con `rnfe` en el nuevo índice y cambiamos el nombre a `HomeScreen` :

```
import { View, Text } from 'react-native'
const HomeScreen = () => {
  return (
    <View>
      <Text>Home</Text>
    </View>
  )
}
export default HomeScreen;
```

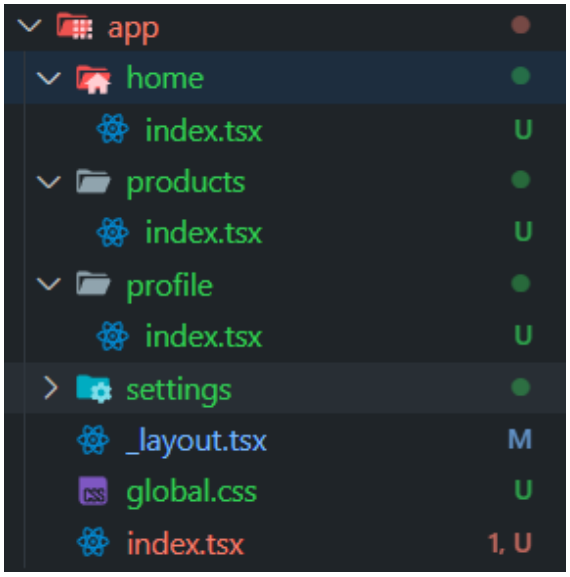
8. Copiamos la carpeta de Home o la duplicamos, cambiamos su nombre a `Profile` y hacemos lo pertinente con su fichero índice:




```
import { View, Text } from 'react-native'
const ProfileScreen = () => {
  return (
    <View>
      <Text>Profile</Text>
    </View>
```

```
  )
}
export default ProfileScreen;
```

9. Repetimos el proceso para `Setting` :




```
import { View, Text } from 'react-native'
const SettingsScreen = () => {
 return (
  <View>
   <Text>Settings</Text>
  </View>
 )
}
export default SettingsScreen;
```

10. Antes de continuar vamos a explicar un par de conceptos:

> `Expo` tiene la capacidad de decidir en función de cierta sintaxis con los nombres el comportamiento de sus directorios. Por defecto, en el directorio app buscará un índice que mostrar, al igual que en todos los subdirectorios. Pero si no existe un índice principal, el primer directorio que encuentre y tenga un índice será el que muestre, lo que nos provocará que mostrará el primer componente alfabéticamente ordenado.

> Si el nombre de un directorio está entre paréntesis, (home), ese directorio no tendrá su ruta visible en la barra de navegación, será alcanzable pero no visible.

11. Vamos al fichero `index.tsx` principal y modificamos la ruta a la que va por defecto:

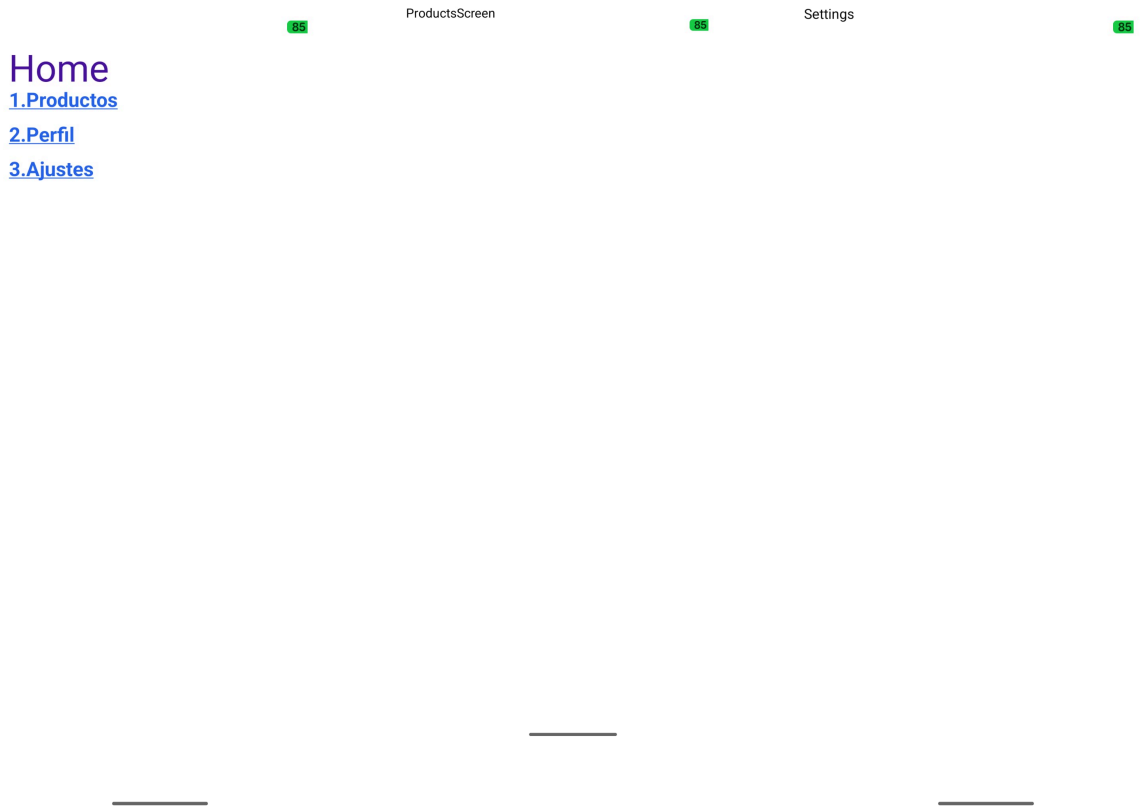
```
return <Redirect href="/home"/>;
```

12. Vamos al fichero índice de la carpeta `home` y creamos un menú como el que vemos a continuación:

```
import { Link } from 'expo-router';
import { View, Text } from 'react-native'
import { SafeAreaView } from 'react-native-safe-area-context';
const HomeScreen = () => {
   const menuItems = [
   { id: 1, label: 'Productos', path: '/products' },
   { id: 2, label: 'Perfil', path: '/profile' },
   { id: 3, label: 'Ajustes', path: '/settings' },
  ];



  return (
   <SafeAreaView>
    <View className="px-10">
     <Text className="text-5xl text-primary">Home</Text>
      {menuItems.map((item) => (
       <View key={item.id}className="flex-row items-center mb-3">
        <Text className=" text-2xl font-bold text-gray-600 mr-2">
         {item.id}.
        </Text>
        <Link href={item.path} class ="text-2xl text-blue-600 underline">
         {item.label}
        </Link>
       </View>
      )
     )}
    </View>
   </SafeAreaView>
  )
```

```
}
export default HomeScreen;
```

13. El resultado sería:



2.5. CustomButtons y Links AsChild

Vamos a empezar a crear componentes genéricos reutilizables a la vez que usamos componentes personalizados y mejoramos los enlaces.

1. Creamos la carpeta `app/components` y dentro de esta carpeta, otra llamada `shared` , esta carpta tendrá un fichero llamado `CustomButton.tsx` y creamos nuestro componente inicial.
2. Comentamos dentro del fichero `HomeScreen.tsx` todos los links y su generación ya que vamos a modificar la creación de estos.

```
<CustomButton color='primary'>Productos</CustomButton>
```

3. Vamos al fichero `CustomButton.tsx` y cambiamos la vista por un elemento `<Pressable>` .
4. Creamos nuestra interfaz `Props` que extiende de `PressableProps` .

```
interface Props extends PressableProps{
  children: string;
  color?: 'primary' | 'secondary' | 'tertiary'
}
```

5. Creamos una variable que contendrá la diferente nomenclatura de la etiqueta de los `className` para evitar la mala praxis de escribir en el `className` algo como `bg-${color}` .

```
const btnColor = {
    primary: 'bg-primary',
    secondary: 'bg-secondary',
    tertiary: 'bg-tertiary',
  }[color];
```

> Con la desestructuración de corchetes `[color]` se le da a la variable `btnColor` directamente el valor asociado a la propiedad, no se pasa por la llamada de `btnColor[color]` siendo `color` el valor de la propiedad de nuestra interfaz.

6. Le damos estilos al elemento `<Pressable>` :

```
<Pressable
  className={`p-3 rounded-md ${btnColor} active:opacity-90`}
>
  <Text className="text-white text-center">{children}</Text>
</Pressable>
```

7. En este punto ya tenemos un botón visible, falta su funcionalidad. Vamos al botón y añadimos el evento `onPress` y onLongPress tras haberlos añadido a la cabecera del componente. El componente quedará así:

```
import { View, Text, Pressable, PressableProps } from 'react-native'
interface Props extends PressableProps{
  children: string;
  color?: 'primary' | 'secondary' | 'tertiary'
}

const CustomButton = ({children, color = 'primary', onPress, onLongPres
s}: Props) => {

  const btnColor = {
    primary: 'bg-primary',
    secondary: 'bg-secondary',
    tertiary: 'bg-tertiary',
  }[color];

  return (
    <Pressable
      className={`p-3 rounded-md ${btnColor} active:opacity-90 mb-1`}
      onPress={onPress}
      onLongPress={onLongPress}
    >
```

```
      <Text className="text-white text-center">{children}</Text>
    </Pressable>
  )
}
export default CustomButton;
```

8. Ya podemos ir al índice principal y darle la función que hará el cambio de pantalla.

```
import { router } from 'expo-router';
//...
<CustomButton onPress={() => router.push('/products')} color='primary'>
Productos</CustomButton>
```

> El método `push` coloca una nueva página sobre la actual.

9. En el caso de querer trabajar los enlaces con `<Link>` en lugar de botones lo hacemos de la siguiente manera:

```
<Link href="/profile" asChild>
        <CustomButton color='primary'>Perfil</CustomButton>
      </Link>
```

> El `asChild` es una **propiedad especial** que indica que **el componente debe renderizar a su hijo directamente**, en lugar de generar su propio elemento por defecto. Se podría decir “Renderiza lo que haya dentro, pero aplica mi comportamiento sobre ese hijo.”

10. El índice debería quedar así:

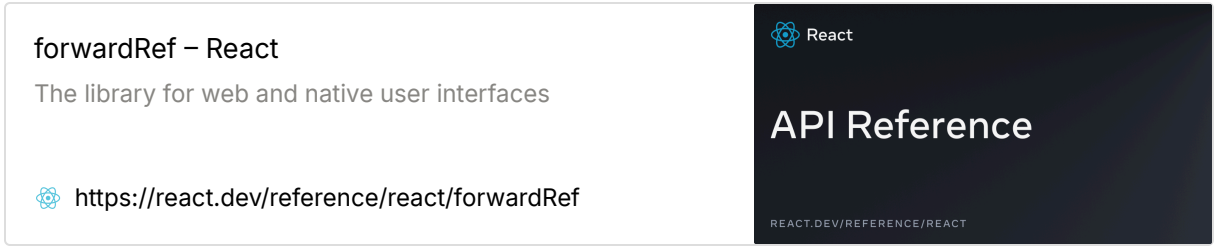
```
import { Link, router } from 'expo-router';
import { View, Text } from 'react-native'
import { SafeAreaView } from 'react-native-safe-area-context';
import CustomButton from '../components/shared/CustomButton';
const HomeScreen = () => {
```

```
  const menuItems = [
  { id: 1, label: 'Productos', path: '/products' },
  { id: 2, label: 'Perfil', path: '/profile' },
  { id: 3, label: 'Ajustes', path: '/settings' },
 ];



 return (
  <SafeAreaView>
   <View className="px-10">
    <Text className="text-5xl text-primary">Home</Text>

    <CustomButton onPress={() => router.push('/products')} color='prima
ry'>Productos</CustomButton>
    <Link href="/profile" asChild>
     <CustomButton color='primary'>Perfil</CustomButton>
    </Link>
    <Link href="/settings" asChild>
     <CustomButton color='primary'>Ajustes</CustomButton>
    </Link>
   </View>
  </SafeAreaView>
 )
}
export default HomeScreen;
```

2.6. Variantes de componentes

Se pueden dar situaciones en las que deseamos que un componente sea capaz de tener varias formas o se adapte al contexto en el que se llama, esto se consigue a través de los parámetros que se le pasa, vamos a ver cómo se hace.

> ACTUALIZACIÓN EN REACT
> Presuponiendo que trabajamos con las últimas versiones de React (v19+) ha cambiado un comportamiento vital que ahora nos ayuda. Antes, al encapsular un texto dentro de un botón, o un botón dentro de un enlace, cuando pasábamos una referencia (cuando hemos enviado nuestro "/products") solo llegaba al componente externo y para poder aplicar dicha referencia había que usar una función `forwardRef` para propagarla a componentes internos a través del `useRef` , ahora `forwardRef` se incorpora a las propiedades básicas de los elementos y componentes, por lo que se auto encapsula y propaga.



1. En nuestras propiedades incorporamos la nueva propiedad para la nueva funcionalidad:

```
variant?: 'contained' | 'text-only';
```

2. Antes del `return` ponemos nuestra comprobación de variante con elementos visuales simplificados:

```
if(variant === 'text-only'){
  return (
    <Pressable
      className={`p-3`}
      onPress={onPress}
      onLongPress={onLongPress}
    >
      <Text className="text-center">{children}</Text>
    </Pressable>
  );
}
```

3. Vamos a nuestro índice, creamos un nuevo botón con la propiedad variante:

```
<Link href="/settings" asChild>
  <CustomButton variant='text-only' color='primary'>Ajustes variante</Cu
stomButton>
</Link>
```

4. Veremos un resultado similar a este:

Ajustes variante

5. Para respetar el estilo visual creado anteriormente seguimos con la creación de una variable muy parecida al `btnColor` :

```
const textColor ={
   primary: 'text-primary',
   secondary: 'text-secondary',
   tertiary: 'text-tertiary',
}[color];
```

6. Ese nuevo color es enviado al `className` del texto de la variante y ajustamos la fuente del componente normal para que ambos tengan la misma fuente:

```
<Text className={`text-center ${textColor} font-work-medium`}>{childre
n}</Text>
```

7. Para seguir con la personalización del botón o componente, es habitual querer mandar un `className` y que este se aplique, para ello ampliamos nuestra interfaz:

```
// Interfaz
className?:string;
// Cabecera
```

```
({children, color = 'primary', onPress, onLongPress, variant = 'contained', className}: Props)
// Pressable className
className={`p-3 ${className}`}
```

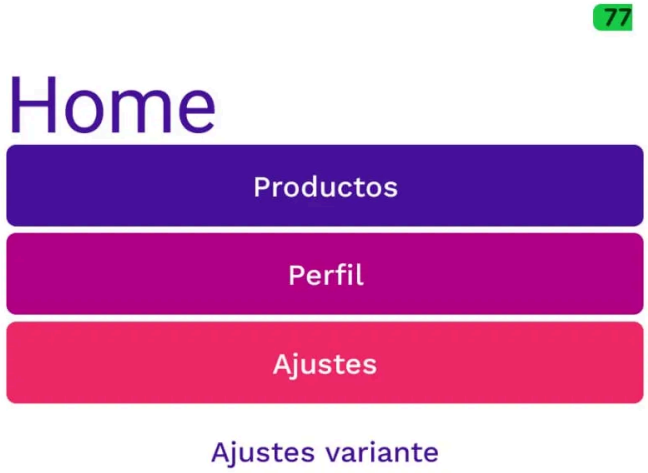
8. Para probarlo vamos al índice y le damos al `CustomButton` un `className="mb-10"` para ver un resultado similar a este:

```
<Link href="/profile" asChild>
  <CustomButton className="mb-10" color='primary'>Perfil</CustomButton>
</Link>
<Link href="/settings" asChild>
  <CustomButton className="mb-10" color='primary'>Ajustes</CustomButton>
</Link>
<Link href="/settings" asChild>
  <CustomButton className="mb-10" variant='text-only' color='primary'>
    Ajustes variante
  </CustomButton>
</Link>
```

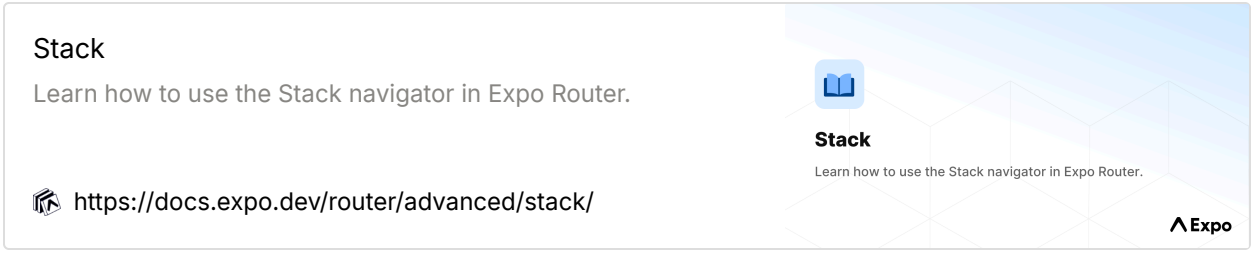
De esta forma tendremos una forma de personalizar totalmente nuestros componentes a través del `className` .

2.7. Navegación con Stack (StackNavigation)

Vamos a empezar a trabajar con el componente de navegación, para ello hacemos una pequeña modificación visual previa, cambiamos el color de Perfil a secundario y el de ajustes a terciario.

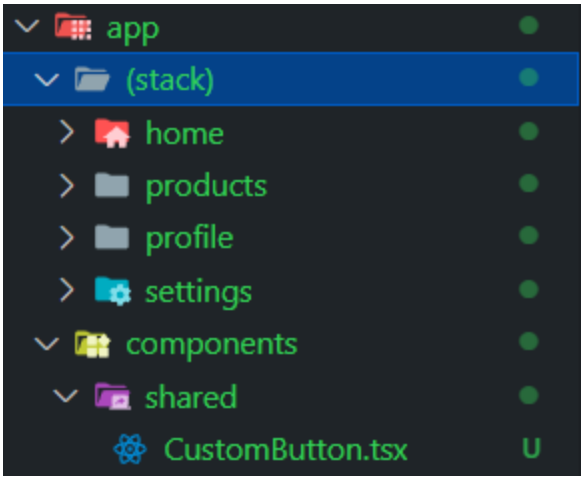


Aquí tenemos toda la documentación al respecto:



1. Accedemos al fichero `_layout.tsx` principal y cambiamos el componente `<Slot>` por un `<Stack>` de `Expo-router` . Ya tenemos disponible un navegador:

2. Si queremos personalizar los textos del `Stack` , títulos, pantallas... Tenemos que personalizar cada una de las pantallas, o como nosotros haremos, configuraremos el `Stack` . Para ello creamos una nueva carpeta con el nombre "(stack)" entre paréntesis porque no queremos que dicho componente o página sea visible al igual que su ruta.
3. Las carpetas `home, products, profile y settings` (las carpetas que contienen nuestras secciones) las metemos dentro de stack.



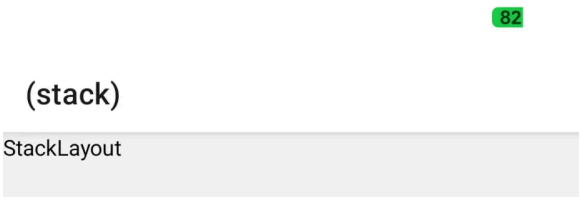
4. Creamos un fichero `_layout.tsx` dentro de `(stack)` al mismo nivel que las carpetas.

```
import { View, Text } from 'react-native'
const StackLayout = () => {
  return (
    <View>
      <Text>StackLayout</Text>
    </View>
  )
}
export default StackLayout;
```

5. Cambiamos el índice raíz del proyecto modificando su enlace a home:

```
return <Redirect href="/(stack)/home"/>
```

6. Veremos que ahora la aplicación nos inicia en la pantalla del `Stack` :



> En caso de no recargar o funcionar bien, si os pide reimportar algún componente como el `CustomButton` y no lo hace bien, simplemente borramos el import, lo ponemos de nuevo y relanzamos el servidor.

7. En el `_layout.tsx` del `stack` quitamos la vista y sus importaciones y ponemos el propio `<Stack>` y en el `_layout.tsx` principal quitamos el Stack y volvemos a poner `<Slot>` :

```
return <Stack/>;
```

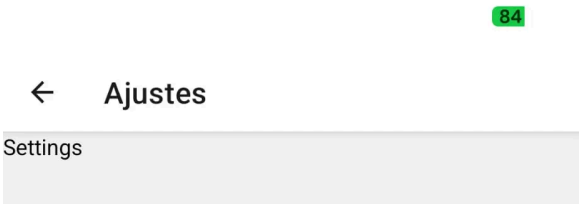
8. Ahora dentro de nuestro `StackLayout` podemos empezar a configurar el componente. Para ello ejecutamos el siguiente código:

```
import { Stack } from 'expo-router';
const StackLayout = () => {
 return (
  <Stack>
   <Stack.Screen
    name="home/index" // localizacion del fichero a renderizar
    options={{
     title: 'Home Screen' // titulo que se asocia al fichero
    }}
   />
  </Stack>
 );
}
export default StackLayout;
```

9. Replicamos el comportamiento para el resto de pantallas:

```
 return (
  <Stack>
   <Stack.Screen
    name="home/index"
```

```
      options={{
        title: 'Home Screen'
      }}
    />
    <Stack.Screen
      name="products/index"
      options={{
        title: 'Productos'
      }}
    />
    <Stack.Screen
      name="profile/index"
      options={{
        title: 'Perfil'
      }}
    />
    <Stack.Screen
      name="settings/index"
      options={{
        title: 'Ajustes'
      }}
    />
  </Stack>
);
```



2.8. Personalización del Stack de navegación

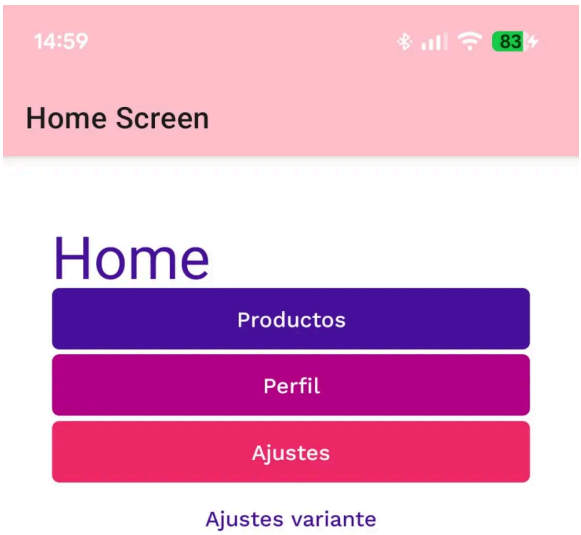
La configuración del `Stack` se puede hacer a nivel de la etiqueta `<Stack>` lo que aplicará a todas las pantallas, o a nivel local con cada `Stack.Screen` .

Dentro de las opciones de cada `Stack.Screen` tenemos muchas propiedades para personalizar nuestras transiciones:

1. Después del título, añadimos la propiedad `animation` y elegimos cualquiera de los valores para probar.
2. Dentro de `<Stack>` aplicamos el siguiente código:

```
screenOptions={{
   headerShown: true, // Se muestra la cabecera al ir a un enlace
   headerShadowVisible: true, // El Stack tiene sombra
   headerStyle: {
   backgroundColor: 'pink',
 },
   contentStyle:{
    backgroundColor: 'white', // Fondo de la zona de enlaces
}
```

3. Probamos a jugar con algunas propiedades a nuestro gusto.



4. Aquí tenemos algunas de las propiedades más interesantes:

Propiedad	Tipo	Descripción / Ejemplo
headerShown	boolean	Muestra u oculta la barra superior. 🧩 `headerShown: false`
headerStyle	objeto estilo	Cambia el color o fondo del header. 🎨 `{ backgroundColor: '#1e40af' }`
headerTintColor	string	Color del texto y botones del header. 💬 `'#fff'`
headerTitle	string o función	Título del header (texto o componente). 🖋️ `"Perfil de Usuario"`
headerTitleStyle	objeto estilo	Estilos del texto del título. 💅 `{ fontSize: 22, fontWeight: '700' }`
headerTitleAlign	`"center"` / `"left"`	Alineación del título. 📍 `"center"`
headerBackTitleVisible	boolean	Oculta el texto del botón “volver”. 🚫 `false`
headerTransparent	boolean	Hace el header transparente. Ideal para fondos con imágenes. 🌫️ `true`
headerShadowVisible	boolean	Quita la sombra inferior del header. 🚫 `false`
headerRight / **headerLeft**	función	Inserta componentes personalizados (iconos, botones...). 🔘 `headerRight: () => <IconButton />`
headerBackground	componente	Permite renderizar un fondo completo (por ejemplo, un gradiente).
animation	string	Tipo de animación entre pantallas. 🎞️ `'slide_from_right'`, `'fade_from_bottom'`
gestureEnabled	boolean	Permite deslizar para volver atrás en iOS/Android. 👆 `true`
contentStyle	objeto estilo	Fondo y padding del área de contenido. 🎨 `{ backgroundColor: '#f9fafb' }`
statusBarStyle	`"light"` / `"dark"`	Color de los iconos de la barra de estado.
orientation	`"portrait"` / `"landscape"`	Fija la orientación de pantalla.

2.9. Listado de productos y su detalle

Empezamos con la primera pantalla de nuestra aplicación, en ella vamos a mostrar una lista de productos que tendrán accesible cada uno de ellos un detalle con mayor cantidad de información.

1. Creamos una nueva carpeta llamada `store` en la raíz del proyecto, en ella estarán todos los datos que vamos a obtener de forma simulada o externa (API).
2. Creamos un fichero llamado `products.store.ts` y pegamos la siguiente lista en el fichero:

```
export const products = [
  {
    id: '1',
    title: 'Auriculares Inalámbricos',
    description:
      'Experimenta una calidad de sonido premium con estos auriculares ina
lámbricos. Diseñados para la comodidad y el uso prolongado, son perfect
os para amantes de la música y profesionales por igual.',
    price: 99.99,
  },
  {
    id: '2',
    title: 'Reloj Inteligente',
    description:
      'Mantente conectado y sigue tu estado físico con este elegante y mod
erno reloj inteligente. Equipado con funciones como monitoreo de frecuen
cia cardíaca, GPS y más.',
    price: 149.99,
  },
  {
    id: '3',
    title: 'Altavoz Bluetooth',
    description:
      'Portátil y potente, este altavoz Bluetooth ofrece un sonido claro y nítid
o donde quiera que vayas. Ideal para fiestas, actividades al aire libre y má
s.',
```

```
    price: 59.99,
  },
  {
    id: '4',
    title: 'Ratón para Gaming',
    description:
      'Mejora tu experiencia de juego con este ratón para gaming de alta pre
cisión. Diseñado ergonómicamente y construido para durar, es imprescind
ible para cualquier jugador.',
    price: 39.99,
  },
  {
    id: '5',
    title: 'Monitor 4K',
    description:
      'Disfruta de impresionantes visuales con este monitor 4K. Perfecto par
a juegos, trabajo o entretenimiento, ofrece colores vibrantes y detalles níti
dos.',
    price: 299.99,
  },
];
```

3. Dentro de nuestro componente `ProductsScreen` empezamos con el siguiente código:

```
const ProductsScreeen = () => {
  return (
    <View className="flex flex-1 px-4">
    </View>
  );
};
```

4. Seguimos creando el componente que contendrá la lista:

```
const ProductsScreeen = () => {
 return (
  <View className="flex flex-1 px-4">
    <FlatList
      data={ products }
      keyExtractor={ ( item ) => item.id }
      renderItem={ ( { item } ) => () }
    />
  </View>
 );
};
```

5. Rellenamos la función que va a renderizar nuestra listacon el título y la descripción:

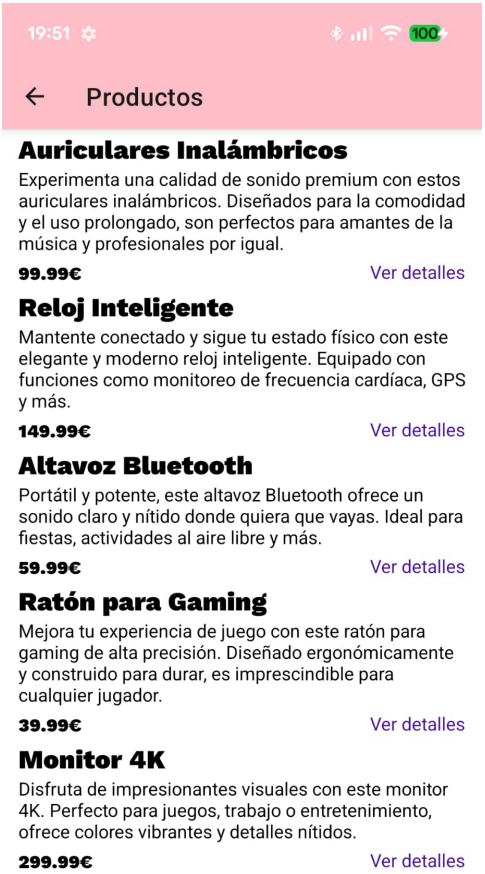
```
const ProductsScreeen = () => {
 return (
  <View className="flex flex-1 px-4">
    <FlatList
      data={ products }
      keyExtractor={ ( item ) => item.id }
      renderItem={ ( { item } ) => (
        <View className="mt-1">
      <Text className="text-2xl font-work-black">{ item.title }</Text>
      <Text className="">{ item.description }</Text>
     </View>
      ) }
    />
  </View>
 );
};
```

6. Agregamos otra vista o “div” que contenga el precio y el enlace a los detalles:

```
import { products } from '@/store/products.store';
import { Link } from 'expo-router';
import { FlatList, Text, View } from 'react-native';

const ProductsScreeen = () => {
 return (
  <View className="flex flex-1 px-4">
    <FlatList
      data={ products }
      keyExtractor={ ( item ) => item.id }
      renderItem={ ( { item } ) => (
        <View className="mt-1">
      <Text className="text-2xl font-work-black">{ item.title }</Text>
      <Text className="">{ item.description }</Text>
      <View className="flex flex-row justify-between mt-1">
       <Text className="font-work-black">{ item.price }€</Text>
       <Link href="/" className="text-primary">
        Ver detalles
       </Link>
      </View>
     </View>
      ) }
    />
  </View>
 );
};
```

> DIFERENTES POSIBLES PROBLEMAS
> 1. La lista no está bien importada: Asegurate que el `products.store` tiene el export sin el default y que el import recibe la lista nombrada.
> 2. La lista está `undefined` o vacía: Comprueba con un `console.log('products:', products)` que la lista es alcanzable y tiene algo.
> 3. Posibles warnings por dependencias desactualizadas: ejecuta el comando `npx expo install --fix` para que el doctor de `Expo` detecte las dependencias no actualizadas y las instale actualizadas.

7. El resultado debería ser algo así:



---

8. Vamos a crear la pantalla de detalles del producto. Para ello vamos a crear un nuevo archivo en la carpeta `products` con el nombre `[id].tsx` con el siguiente código:

```
import { View, Text } from 'react-native'
const ProductoScreen = () => {
  return (
    <View>
      <Text>ProductoScreen</Text>
    </View>
```

```
  )
}
export default ProductoScreen;
```

9. Vamos al enlace que pusimos en el índice de productos de “Ver detalles” y cambiamos el enlace al siguiente código:

```
<Link href={ `/(stack)/products/${ item.id }` } className="text-primary">
```

10. El componente debería quedar así:

```
import { products } from '@/store/products.store';
import { Link } from 'expo-router';
import { FlatList, Text, View } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';
const ProductsScreeen = () => {
 return (
  <View className="flex flex-1 px-4">
   <FlatList
    data={ products }
    ListEmptyComponent={ <Text>No hay productos.</Text> }
    keyExtractor={ ( item ) => item.id }
    renderItem={ ( { item } ) => (
     <View className="mt-1">
      <Text className="text-2xl font-work-black">{ item.title }</Text>
      <Text className="">{ item.description }</Text>
      <View className="flex flex-row justify-between mt-1">
       <Text className="font-work-black">{ item.price }</Text>
       <Link
          href={ `/(stack)/products/${ item.id }` }
          className="text-primary"
       >
        Ver detalles
       </Link>
      </View>
```

```
        </View>
      ) }
    />
  </View>
 );
};
export default ProductsScreeen;
```

ProductoScreen

11. Ahora vamos a recibir el argumento y tratar con él. Para ello ponemos el siguiente código:

```
const params = useLocalSearchParams();
console.log( params );
```

12. Comprobamos que nos funciona el `console.log( params )` cuando entramos en un producto y ajustamos el código a `const {id} = useLocalSearchParams();` y borramos la salida por terminal.
13. Buscamos nuestro producto con el siguiente código y en caso de error lo controlamos:

```
const product = products.find(p => p.id === id);
if(!product){
 return <Redirect href='/(stack)/products'/>
}
```

14. En la devolución de nuestro componente ahora podemos poner los datos:

```
return (
 <View className="px-5 mt-2">
  <Text className="font-work-black text-2xl">{product.title}</Text>
```

```
    <Text className="">{product.description}</Text>
    <Text className="font-work-black text-2xl">{product.price}</Text>
  </View>
);
```