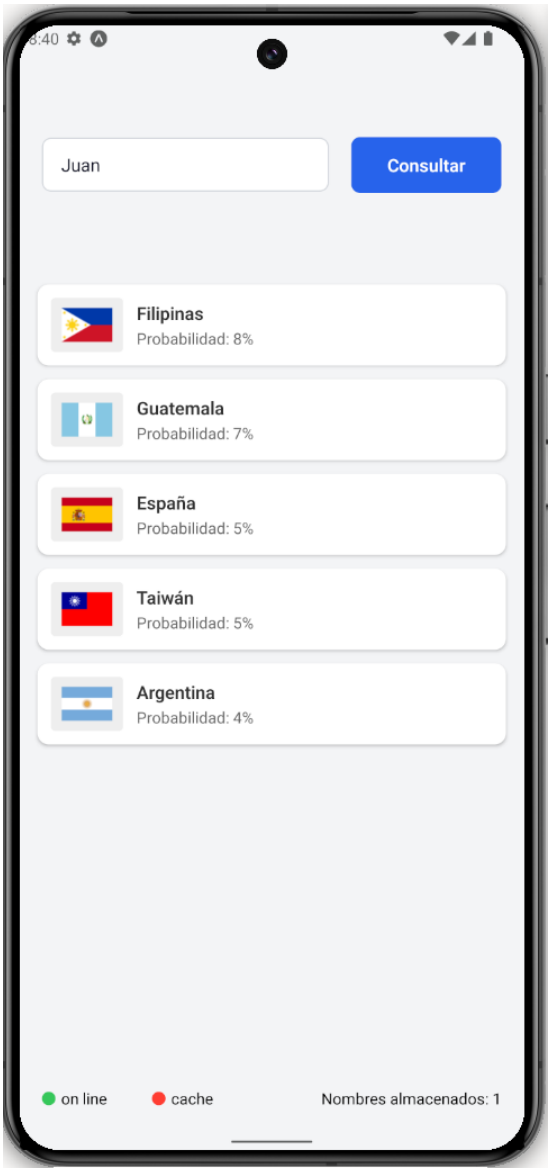
TUTORIAL 17

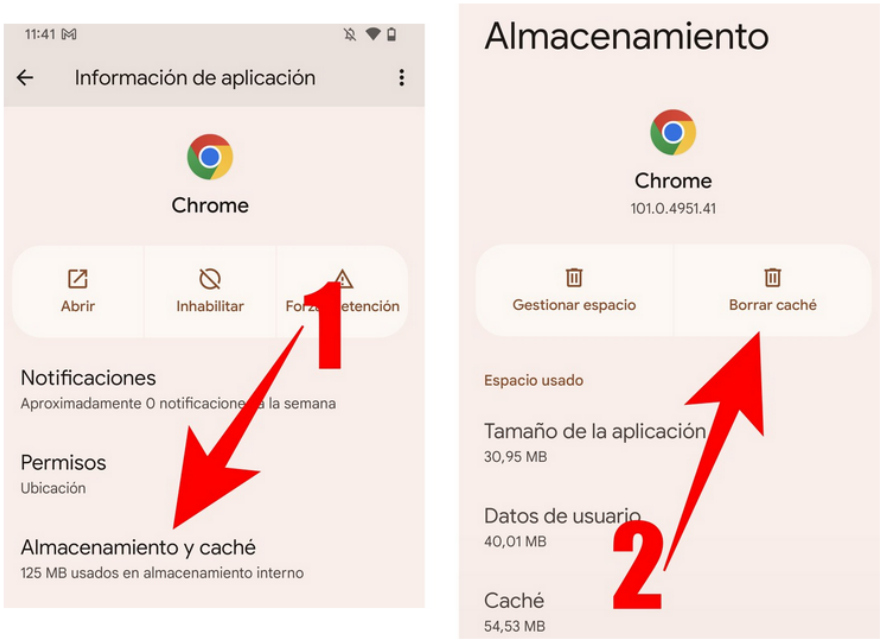
ASYNC STORAGE



Nuestra app depende de la conexión a internet para funcionar. Vamos a implementar un modo off-line de forma que si no hay conexión (o el api no está disponible), podamos seguir consultando los nombres que ya hayamos consultado previamente. Para ello, guardaremos los nombres consultados en el almacenamiento interno del dispositivo.

1.- El Async Storage

Cada app de un dispositivo móvil de cualquier tipo (android, iOS) dispone de un área privada donde puede almacenar datos. Por ejemplo, en Android el usuario puede borrarlos desde el menú de la aplicación.



Vamos a aprovechar ese espacio de almacenamiento reservado a nuestra app para guardar en él todos los nombres que consultamos. De esa manera, si alguna vez no hay conexión, podremos recuperar esos nombres. Crearemos así un modo ***off-line***

1. Abre el proyecto del tutorial anterior y crea rama una llamada **tutorial17**
2. Ejecuta el siguiente comando para instalar la librería **Async Storage,** que permite el acceso al almacenamiento interno:

npx expo install @react-native-async-storage/async-storage

La librería ***Async Storage*** *permite guardar parejas* ***clave-valor*** *en el almacenamiento interno. Como su nombre indica, todas las funciones que trabajan con ella son* ***asíncronas****, porque acceder al almacenamiento es una operación costosa (aunque mucho menos que consultar un api)*

3. En el archivo **ConsultasApi.ts** cambia el nombre de la función **consultarProbabilidades** para que se llame **consultarProbabilidadesApi**

Hacemos este cambio porque pronto vamos a tener dos formas de consultar los nombres:

- ***consultarProbabilidadesApi*** → *usa el api para obtener las probabilidades*
- ***consultarProbabilidadesOffLine*** → *usa el almacenamiento interno*

Antes de programar el acceso a almacenamiento interno, vamos a crear dos componentes útiles para monitorizar su uso.

2.- El componente BotonModo

Este componente va a ser un botón situado en la parte inferior cuyo color nos mostrará si estamos en modo online/offline. Al pulsarlo, cambiaremos de modo.

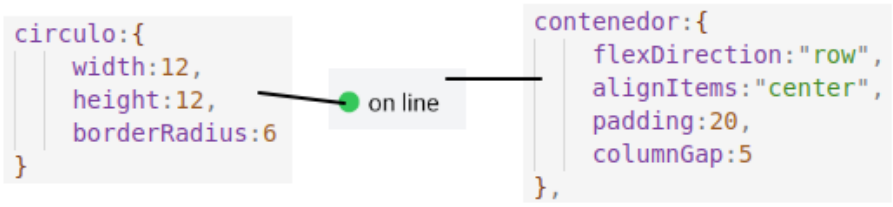
1. Abre **App.tsx** y añade una variable de estado llamado **onLine** con valor inicial **true** y su setter se llamará **setOnLine**

```
2.  export default function App() {
3.    const [onLine, setOnLine] = useState(false)
4.    // resto omitido
5.  }
```

> *Esta variable de estado valdrá **true** cuando estemos en modo offline*

6. En **components** crea un archivo llamado **BotonModo.tsx**
7. Abre **BotonModo.tsx** y teclea **rnfs+Intro**
8. Añade a la función **Boton** estos props:
   - **texto** → Es el mensaje que se mostrará en el botón
   - **activado** → Vale true si el modo off-line está activado
   - **setActivado** → Es la función que cambia el valor de **activado**

9. Como ejercicio, programa este diseño sin mirar la solución. Ten en cuenta que:
   - El contenedor del componente es un **Pressable**, que al ser pulsado cambia el valor de **activado** por su contrario
   - El color del círculo es **#ff3b30** para el modo activado y **#34c759** para el modo desactivado




```
1.  export default function BotonModo({texto,activado,setActivado}) {
2.      const activado = activado ? "#34c759 " : "#ff3b30"
3.      return (
4.      <Pressable style={styles.contenedor} onPress={ ()=> setActivado(!activado) }>
5.        <View style={[styles.circulo, {backgroundColor:color}]}/>
6.        <Text>{texto}</Text>
7.      </Pressable>
8.    )
9.  }
```

10. Abre **App.tsx** e inserta al final del contenedor principal un **BotonModo** pasándole como props la variable de estado **onLine** y su setter **setOnLine**

```
1.  export default function App() {
2.    // inicio omitido
3.    return (
4.      <View style={styles.contenedorPrincipal}>
5.          // resto omitido
6.          <BotonModo texto={"on line"} activado={onLine} setActivado={setOnLine}/>
7.      </View>
8.    );
9.  }
```

3.- El componente InformacionNombres

Este componente va a ser un texto situado en la parte inferior, que nos muestra cuántas palabras disponibles hay en el almacenamiento interno

1. Abre **App.tsx** y añade una variable de estado llamada **totalNombresOffLine** con valor inicial **0** y su setter se llamará **setTotalNombresOffLine**

```
export default function App() {
  const [totalNombresOffLine, setTotalNombresOffLine] = useState(0)
  // resto omitido
}
```

Esta variable guardará el número de nombres que hay en el almacenamiento interno

2. En **components** crea un archivo llamado **InformacionNombres.tsx**
3. Añade a la función **InformacionNombres** este prop:
   - **totalNombresOffLine** → Será la cantidad de nombres almacenados en el almacenamiento interno

```
export default function InformacionNombres({totalNombresOffLine}) {
  // resto omitido
}
```

4. Diseña el componente de forma que solo haya un texto (sin estilo) que muestre un mensaje informando cuántas palabras hay en el almacenamiento

```
export default function InformacionNombres({totalNombresOffLine}) {
  return (
    <View>
      <Text>Nombres almacenados: {totalNombresOffLine}</Text>
    </View>
  )
}
```

5. Coloca debajo el **BotonModo** un **InformacionNombre**, pasando como prop la variable **totalNombresOffLine.** Como ejercicio, sin mirar la solución encierra ambos en un **View** que tenga estilo para poner ambos de esta forma:


```
<View style={styles.filaFondo}>
  <BotonModo texto={"on line"} activado={onLine} setActivado={setOnLine}/>
  <InformacionNombres totalNombresOffLine={totalNombresOffLine}/>
</View>
```

```
filaFondo:{
  flexDirection:"row",
  alignItems:"center",
  justifyContent:"space-between",
  paddingRight:20
},
```

4.- Consultar la cantidad de datos almacenados

Lo primero que vamos a hacer es consultar la cantidad de nombres almacenados en el almacenamiento interno. Para ello usaremos la función asíncrona ***AsyncStorage.getAllKeys****, que nos da una lista con todas las claves almacenadas (pero no sus valores). Contándolas, podremos saber cuántos nombres hay.*

1. En **helpers** crea un archivo llamado **ConsultaAlmacenamientoInterno.ts**
2. Crea y exporta una función asíncrona llamada **getNumeroNombresOffLine**, que llame a la función **AsyncStorage.getAllKeys** y nos devuelva la cantidad de nombres almacenados

```
1.  async function getNumeroNombresOffLine():Promise<number>{
2.      const claves = await AsyncStorage.getAllKeys()
3.      return claves.length
4.  }
```

*Como **AsyncStorage.getAllKeys** es asíncrona, **getNumeroNombresOffLine** también debe serlo*

3. En la función **App,** crea la función asíncrona **actualizarNumeroNombres**, que llame a **getNumeroNombresOffLine** y almacene el resultado en **totalNombresOffLine**

```
1.  async function actualizarNumeroNombres(){
2.    const total = await getNumeroNombresOffLine()
3.    setTotalNombresOffLine(total)
4.  }
```

4. En la función **App** haz que se llame a **actualizarNumeroNombres** cada vez que cambie la variable de estado **listaProbabilidades**

```
1.  export default function App() {
2.    // variables de estado omitidas
3.    useEffect( () => { actualizarNumeroNombres() }, [listaProbabilidades])
4.    // resto omitido
5.  }
```

☞ ***¿Por qué debe llamarse actualizarNumeroNombres cuando cambia listaProbabilidades?*** *Recuerda que **totalNombresOffLine** guarda la cantidad de nombres almacenados, y ese valor puede cambiar tras la finalización de cada consulta (porque cuando realizamos una consulta, se almacena el resultado). Por eso, el lugar correcto para llamar a esa función es tras realizar una consulta.*

5.- Guardar datos en el Async Storage

*La función asíncrona **AsyncStorage.setItem** guarda una pareja clave-valor (ambas **string**) en el almacenamiento interno. Por facilidad, los valores se suelen guardar en en formato **json**, aunque no es obligatorio.*

1. En **helpers** crea un archivo llamado **ConsultaAlmacenamientoInterno.ts**
2. En dicho archivo crea y exporta una función asíncrona llamada **guardarProbabilidad**, que reciba el nombre de una persona y su lista de probabilidades y la guarde en el almacenamiento interno.

```
1.  async function guardarProbabilidad(nombre:string, probabilidades:Array<Probabilidad>){
2.      const json = JSON.stringify(probabilidades)
3.      await AsyncStorage.setItem(nombre,json)
4.  }
5.  export {guardarProbabilidad}
```

*El método **JSON.stringify** recibe un objeto cualquiera y devuelve un **string** con ese objeto convertido en json*

3. Abre **ConsultasApi.ts** y modifica la función **consultarProbabilidadesApi,** para que después del bucle que inyecta el campo país a la respuesta del api, se guarde en el almacenamiento el nombre (como clave) junto con su lista de probabilidades (el valor)

```
async function consultarProbabilidadesApi(nombre:string):Promise<Array<Probabilidad>>{
    const endpoint = `https://api.nationalize.io/?name=${nombre}`
    const respuestaServidor = await axios.get(endpoint)
    const resultado = respuestaServidor.data.country
    for(let objeto of resultado){
        objeto.pais = await consultarNombrePais(objeto.country_id)
    }
    await guardarProbabilidad(nombre,resultado)
    return resultado
}
```

Haciendo esto, si el nombre no se encuentra en el almacenamiento interno, lo crea y le asocia su lista de probabilidades. Si el nombre ya existe, lo actualiza. De esta forma, cada consulta al api actualiza el almacenamiento interno

4. Ejecuta la app, comprueba que cada vez que consultas un nombre diferente se incrementa el contador de los números almacenados y si pones un nombre repetido no se incrementa

6.- Activación del modo off-line

*Ahora mismo podemos pasar al modo off-line (aunque todavía siga consultando el api) pulsando en el **BotonModo**. Vamos a hacer que si al consultar el api se produce un error, automáticamente se pase al modo off-line.*

1. Modifica la función **botonPulsado** para que la app entre en modo off-line en caso de producirse un error en la consulta al api

```
function botonPulsado(){
  if(validarNombre()){
    setCapaActiva(2)
    consultarProbabilidadesApi(nombre)
      .then( respuesta => {
        setListaProbabilidades(respuesta)
        setCapaActiva(3)
      }).catch( error => {
        Alert.alert("Error",error.toString())
        setCapaActiva(1)
        setOnLine(false)
      })
  }else{
    Alert.alert("Error","El nombre no puede dejarse vacío")
  }
}
```

2. Quita la conexión a internet del móvil (o el emulador) y comprueba que tras el error de conexión la app entra en modo offline

Si usas el emulador, quita la conexión con la app ya abierta, o expo no la podrá abrir

7.- Recuperación de datos del almacenamiento interno

Vamos ahora a programar el modo off-line, de manera que en ese modo la consulta de nombres se realice sobre los nombres almacenados internamente.

*Para recuperar el valor asociado a una clave del almacenamiento interno se usa la función asíncrona **AsyncStorage.getItem.** Esta función devolverá **null** si la clave consultada no existe en el almacenamiento.*

1. Abre **ConsultaAlmacenamientoInterno.ts,** crea y exporta una función asíncrona llamada **consultarProbabilidadesOffLine**, que reciba un nombre y nos devuelva la lista de probabilidades asociada a ese nombre, o una lista vacía si ese nombre no está almacenado

```
async function consultarProbabilidadesOffLine(nombre:string){
    let lista = []
    const json = await AsyncStorage.getItem(nombre)
    if(json!=null){
        lista = JSON.parse(json)
    }
    return lista
}
```

2. Abre **ConsultasApi.ts,** crea y exporta una función asíncrona llamada **consultarProbabilidades** que reciba como parámetros un nombre y un **boolean** para indicar el modo (online/offline), y haga lo siguiente:
    - En caso de estar en modo online, se llamará a **consultarProbabilidadesApi**
    - En caso de estar en modo offline, se llamará a **consultarProbabilidadesOffLine**

```
async function consultarProbabilidades(nombre:string,online:boolean){
    return onLine ? consultarProbabilidadesApi(nombre) : consultarProbabilidadesOffLine(nombre)
}
```

3. En el archivo **App.tsx** modifica **botonPulsado** para que llame a **consultarProbabilidades** en lugar de **consultarProbabilidadesApi.** Tendrás que pasarle el nombre y la variable **offLine**

```
function botonPulsado(){
  if(validarNombre()){
    setCapaActiva(2)
    consultarProbabilidades(nombre,onLine)
      .then( respuesta => {
        setListaProbabilidades(respuesta)
        setCapaActiva(3)
      })
      .catch( error => {
        Alert.alert("Error",error.toString())
        setCapaActiva(1)
        setOffLine(true)
      })
  }else{
    Alert.alert("Error","El nombre no puede dejarse vacío")
  }
}
```

4. Ejecuta la app y comprueba que en el modo off-line puedes consultar los nombres que ya habías consultado y almacenado previamente

8.- Comprobar si un dato está en el almacenamiento interno

El modo off-line tiene una ventaja inesperada. Si nos damos cuenta, las consultas de nombres en el modo offline van mucho más rápidas que las consultas hechas a la api. Así que podemos plantearnos esto:

Si un nombre ya ha sido consultado, vamos a usar sus datos almacenados localmente. En caso contrario, acudimos al api

*Esto se conoce como un **sistema de caché**. Tenemos una fuente de datos rápida (el almacenamiento interno) y otra lenta (el api). Usamos la lenta solo cuando el dato buscado no está en la rápida.*

*El inconveniente de usar el almacenamiento como **caché** es que el api puede actualizar sus datos y el almacenamiento interno estará desactualizado, porque los nombres consultados ya no se actualizarán. Para que esto no pase, pondremos un **BotonModo** que permita indicar si en el modo online queremos usar la caché o no*

1. Abre **App.tsx** y añade una variable de estado llamada **usarCache** con valor inicial **true** (por defecto, se usará la caché en modo online) y su setter

```
export default function App() {
  const [usarCache, setUsarCache] = useState(true)
  // resto omitido
}
```

2. Coloca al lado del **BotonModo** para alternar el modo online, un nuevo **BotonModo** que permita alternar el uso de la caché (ambos se encerrarán en un **View** que distribuya en fila sus elementos, para que estén uno junto al otro, y alejados del texto de su derecha)

```
<View style={styles.filaFondo}>
  <View>
    <BotonModo texto={"on line"} activado={onLine} setActivado={setOnLine}/>
    <BotonModo texto={"cache"} activado={usarCache} setActivado={setUsarCache}/>
  </View>
</View>
```

3. Abre **ConsultaAlmacenamientoInterno.ts,** crea y exporta una función asíncrona llamada **existeNombre**, que reciba un nombre y devuelva **true** si existe en el almacenamiento interno. Esto se hace consultando todas las claves almacenadas y viendo si en ellas está el nombre buscado

```
async function existeNombre(nombre:string):Promise<boolean>{
    const claves = await AsyncStorage.getAllKeys()
    return claves.includes(nombre)
}
```

*Las listas tienen un método llamado **includes** para saber si un dato está en la lista*

4. Abre **ConsultasApi.ts** y modifica la función **consultarProbabilidades** para que reciba un tercer parámetro llamado **usarCache.**

5. Modifica dicha función para que en el modo on-line, si se usa la caché, primero se compruebe si la clave existe en el almacenamiento, devolviendo dicho valor. En caso contrario, se accederá al api

```
async function consultarProbabilidades(nombre:string,online:boolean, usarCache:boolean){
    let resultado = []
    if(onLine){
        if(usarCache){
            const existe = await existeNombre(nombre)
            if(existe){
                resultado = await consultarProbabilidadesOffLine(nombre)
            }else{
                resultado = await consultarProbabilidadesApi(nombre)
            }
        }else {
            resultado = await consultarProbabilidadesApi(nombre)
        }
    }else{
        resultado = await consultarProbabilidadesOffLine(nombre)
    }
    return resultado
}
```

*Para que se vea más claro el algoritmo, se ha optado por usar **if-else** en lugar de asignaciones condicionales encadenadas*

6. Abre **App.tsx** y modifica la función **botonPulsado** para que se pase como tercer parámetro de **consultarProbabilidades** la variable **usarCache**

```
function botonPulsado(){
  if(validarNombre()){
    setCapaActiva(2)
    consultarProbabilidades(nombre,onLine, usarCache)
      .then( respuesta => {
        setListaProbabilidades(respuesta)
        setCapaActiva(3)
      })
      .catch( error => {
        Alert.alert("Error",error.toString())
        setCapaActiva(1)
        setOffLine(true)
      })
  }else{
    Alert.alert("Error","El nombre no puede dejarse vacío")
  }
}
```

7. Ejecuta la app y comprueba que cuando la caché está activada, la consulta de los nombres almacenados es muy rápida, y la de los nuevos nombres es lenta

9.- Borrar todos los datos almacenados

Conforme se hace uso de la caché el dispositivo va llenando su espacio de almacenamiento, y aunque el usuario siempre lo puede borrar desde el sistema operativo, vamos a ofrecer una opción para borrarlo. Haremos que al pulsar el mensaje con el número de mensajes almacenados, se nos pregunte si deseamos borrarlos.

*Para borrar todos los datos del almacenamiento interno, se usa la función asíncrona **AsyncStorage.clear()***

1. Abre **ConsultaAlmacenamientoInterno.ts,** crea y exporta una función llamada **borrarNombresOffLine** y prográmala de forma que se borren todos los nombres almacenados.

```
async function borrarNombresOffLine(){
    await AsyncStorage.clear()
}
```

2. Abre **App.tsx** y añade en la función **App** una función llamada **borrarNombres** que muestre una ventana emergente de confirmación, y en caso de que el usuario confirme el borrado, llame a **borrarNombresOffLine**

```
function borrarNombres() {
  Alert.alert(
    "¿Desea borrar todos los datos?",
    "Los datos eliminados no pueden ser recuperados",
    [{text:"Aceptar", onPress: () => borrarNombresOffLine() },
      {text:"Cancelar"}]
  )
}
```

3. Modifica la función anterior para actualizar **listaProbabilidades** a una lista vacía tras la llamada a **borrarNombresOffLine**

```
function borrarNombres() {
  Alert.alert(
    "¿Desea borrar todos los datos?",
    "Los datos eliminados no pueden ser recuperados",
    [{text:"Aceptar", onPress: () => {
                                    borrarNombresOffLine()
                                    setListaProbabilidades([])
                                  }},
      {text:"Cancelar"}]
  )
}
```

☞ ***¿Por qué hay que hacer esto?*** *Realmente para borrar los nombres almacenados no es necesario hacer esa llamada, pero recuerda que el texto de abajo a la derecha se actualiza cuando cambia la lista de probabilidades. Por ese motivo hacemos la llamada, para forzar esa actualización tras borrar todos los nombres almacenados*

10.- Versión TypeScript

*Terminamos el tutorial con la versión **TypeScript** de los componentes generados*

1. Abre **BotonModo.tsx** y crea el tipo **BotonModoProps**, teniendo en cuenta que el setter es de tipo **React.Dispatch<React.SetStateAction<boolean>>**

```
type BotonModoProps = {
  texto:string
  activado:boolean
  setActivado:React.Dispatch<React.SetStateAction<boolean>>
}
export default function BotonModo({texto, activado, setActivado}:BotonModoProps) {
   // resto omitido
}
```

2. Abre **InformacionNombres.tsx** y crea el tipo **InformacionNombresProps**

```
type  InformacionNombresProps = {
  totalNombresOffLine: number
}
export default function InformacionNombres({totalNombresOffLine}:InformacionNombresProps) {
  // resto omitido
}
```

3. Haz un commit titulado “terminado el tutorial 17”
4. Sitúate en la rama **main** y mezcla en ella la rama **tutorial17**
5. Haz **push**