
https://reactnative.dev/



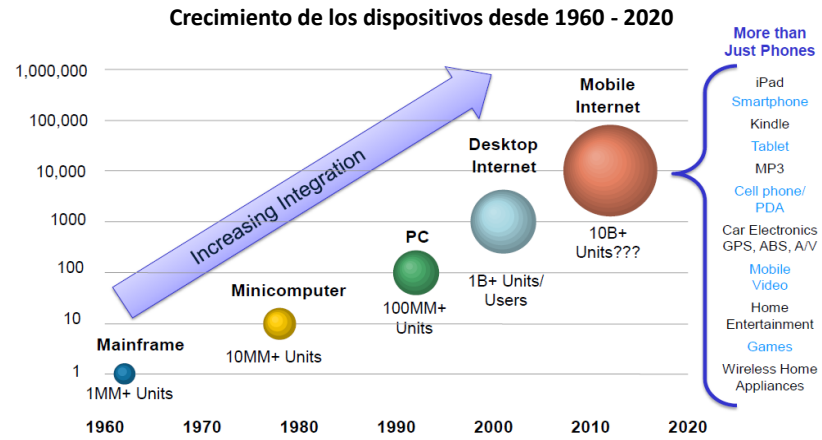
# EVOLUCIÓN DEL DESARROLLO PARA DISPOSITIVOS MÓVILES

# Introducción

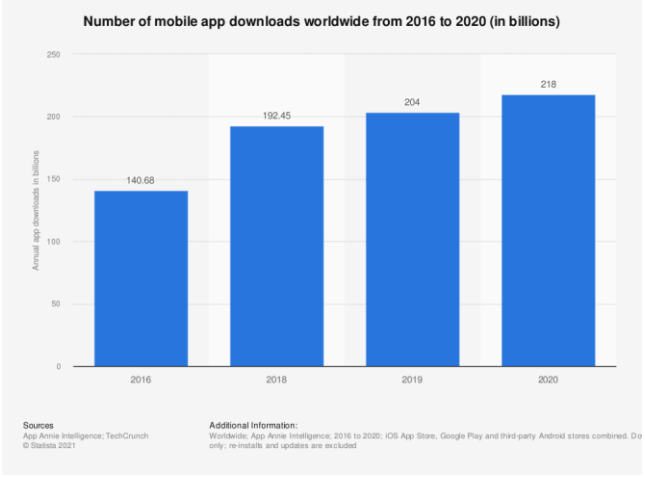
- En muy pocos años los diferentes dispositivos (pc, telefonos moviles, tablets, TV, etc) han llegado a convertirse en un elemento esencial en nuestras vidas.
- El continuo incremento del uso de los teléfonos inteligentes y tabletas ha provocado que el desarrollo de aplicaciones móviles sea una de las disciplinas más populares de creación de software.
  - https://www.businessofapps.com/data/app-revenues/
- La expansión del desarrollo de aplicaciones móviles ha provocado que los usuarios se separen de la informática de escritorio. Se han decantado por la inmediatez, la portabilidad y la ergonomía de los dispositivos móviles.
- Se ha pasado de una forma vertiginosa de los MainFrame a los dispositivos Mobile Internet y al IoT (Internet Of Things).
- Actualmente hay miles de modelos y los diferentes fabricantes siguen innovando intentando destacar sobre los de la competencia.



# Introducción

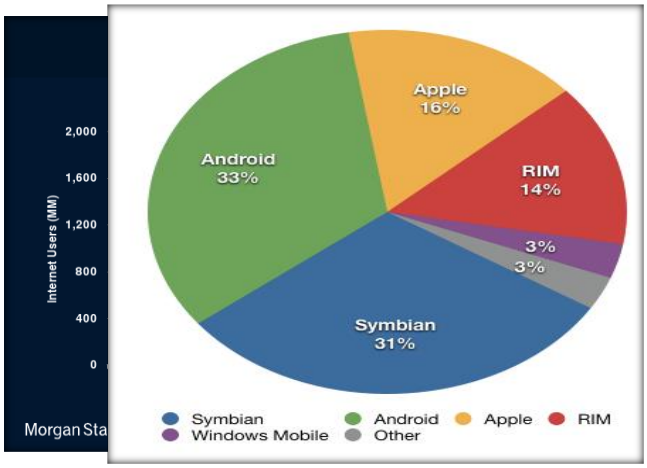


# Introducción






# Introducción



# Retos de aplicaciones móviles.

- Fragmentación de Sistema Operativo
  - Multitud de plataformas móviles aumenta la fragmentación a la hora de elección del Sistema Operativo
- Multitud de productos y equipos
  - A la hora de realizar una aplicación móvil deberemos tener en cuenta cada plataforma y sus múltiples equipos en los que funcionará
- Consistencia de aplicaciones
  - Nuestras aplicaciones deberán ser consistentes en todas las plataformas desarrolladas para ella.
- Fragmentación de funcionalidades
  - Las funcionalidades y capacidades de cada dispositivo varían entre plataformas.



# Fragmentación de Sistema Operativo

Worldwide Smartphone Sales to
End Users by Operating System in 1Q17

| Operating System | 1Q17 Units | 1Q17 Market Share (%) | 1Q16 Units | 1Q16 Market Share (%) |
|---|---|---|---|---|
| Android | 327,163.6 | 86.1 | 292,746.9 | 84.1 |
| iOS | 51,992.5 | 13.7 | 51,629.5 | 14.8 |
| Other OS | 821.2 | 0.2 | 3,847.8 | 1.1 |
| **Total** | **379,977.3** | **100.0** | **348,224.2** | **100.0** |

http://www.gartner.com/newsroom/id/3725117

# Requisitos de desarrollo

| Mobile OS | Operating System | Software/IDEs | Programming Language | Marquet / Policy |
|---|---|---|---|---|
| iOS | Mac only | Xcode | Objective C, Swift | Yes / Yes |
| Android | Windows/Mac/ Linux | Eclipse/Java/ADT | Java, Kotlin | Op. / Lax |
| BlackBerry | Windows mainly | Eclipse/JDE, Java | Java | Yes / Yes |
| Symbian | Windows/Mac/ Linux | Carbide.c++ | C++ | Yes / Yes |
| WebOS | Windows/Mac/ Linux | Eclipse/WebOS plugin | HTML/JavaScript/C++ | |
| Windows 7 Phone | Windows mainly | Visual Studio 2010 | C#, .NET, Silverlight or WPF | Yes / Yes |



# Fragmentación de funcionalidades

- Buttons
- Camera
- Media (audio recording)
- Notification (sound)
- Notification (vibration)
- Accelerometer
- Compass
- Geo location (GPS)
- Contacts
- File
- Storage (database)

# Diseño Adaptativo

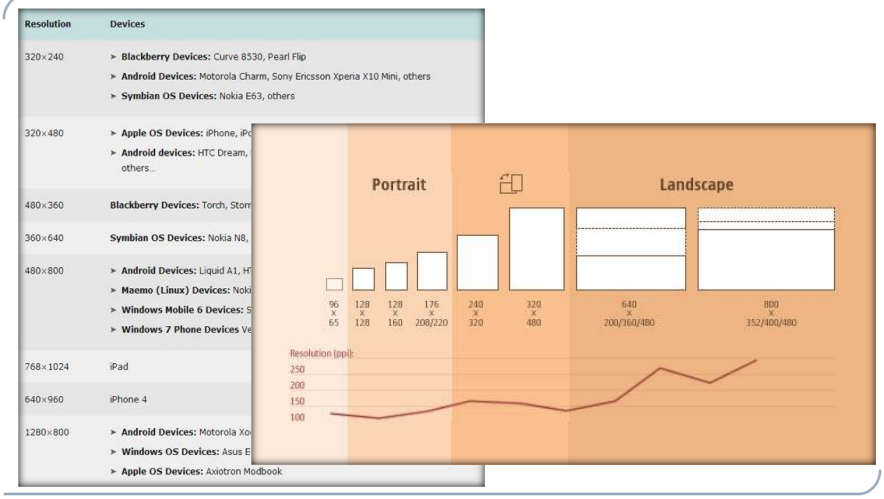
- Es un enfoque de diseño destinado a la elaboración de sitios/aplicaciones para proporcionar un entorno óptimo de:
  - Lectura Fácil
  - Navegación correcta con un número mínimo de cambio de tamaño
  - Planificaciones y desplazamientos
- Con la irrupción multitud de nuevos dispositivos y el que el acceso a internet se realiza ya mayoritariamente desde dispositivos diferentes a los tradicionales ordenadores ha obligado a seguir dicho enfoque en las aplicaciones WEB.
- Contempla la definición de múltiples elementos antes de la realización de la programación real.
  - Elementos de página en las unidades de medidas correctas
  - Imágenes flexibles
  - Utilización de CSS dependiendo de la aplicación



# Resolución

- Los dispositivos móviles tienen una característica distintiva, y es su resolución de pantalla.
- Es necesario conocer cuales son las resoluciones más comunes en este tipo de dispositivos móviles, de los gadgets más utilizados, etc.
- Las resoluciones van cambiando de forma muy rápida y en dispositivos nuevos

# Resolución






# Resolución

- También deberemos tener en cuenta la resoluciones de otros dispositivos como:
  - Tablets
  - TV SmartTV
  - Pizarras electrónicas, etc

| | |
|---|---|
| **Pizarras** | **10.1" y 11.6" (2560x1440, 1920x1080, 1366x768), 17" (1920x1080)** |
| **PC** | **12" (1280x800), 14" (1920x1080, 1366x768), 15.6" (1920x1080)** |
| **Family hub** | **23" (1920x1080), 27" (2560x1440)** |

# Orientación de Página

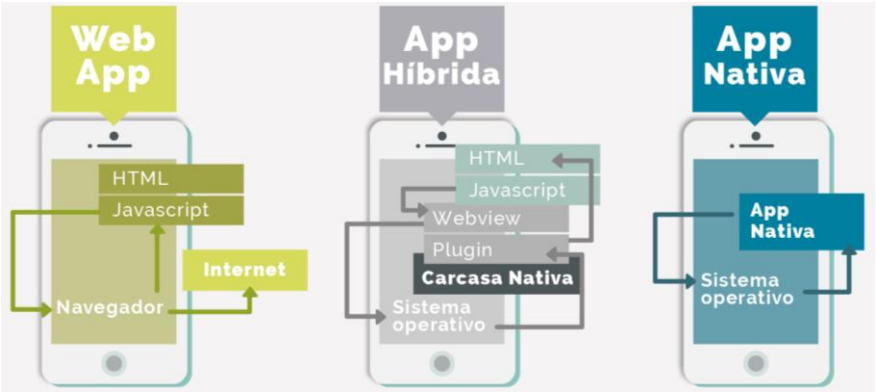
- La orientación del papel es la forma en la que una página rectangular está orientada y es visualizada.
- Los dos tipos más comunes son:
  - Landscape (Horizontal)
  - Portrait (Vertical)



# Cuestiones previas

- ¿En qué dispositivo(s) se ejecutará nuestra aplicación?
- ¿Debe ejecutarse en un navegador o como una aplicación nativa?
- ¿Qué lenguajes(s) conocemos o deberíamos usar?
  - HTML5/JavaScript, Objective C, Java ME, Flash, Silverlight, C#, Dalvik VM, Java FX
- ¿Qué marco de desarrollo multiplataforma deberíamos usar? ¿Tienen su propia API para aprender?
- Si es JavaScript, ¿qué bibliotecas de JavaScript deberíamos incorporar?
- ¿Cómo desarrollamos/depuramos?
  - ¿Qué emuladores/simuladores están disponibles?
  - ¿Qué kits de herramientas de desarrollo utilizamos?

# Estrategias de desarrollo






# Estrategias de desarrollo: Nativa

- Una aplicación nativa está diseñada para ejecutarse en un sistema operativo móvil específico.
- No se ejecutará en otros sistemas operativos móviles.
- Las mayores ventajas son que normalmente pueden acceder fácilmente a todas las funciones del dispositivo elegido y las características de su SO.
- Las mayores desventajas son el coste, el tiempo y el conocimiento al no poder reutilizar prácticamente nada y tener que desarrollar para cada SO.
- Es la estrategia apropiada para aquellas aplicaciones que requieran sacar el máximo partido a los dispositivos.

# Estrategias de desarrollo: Híbrida

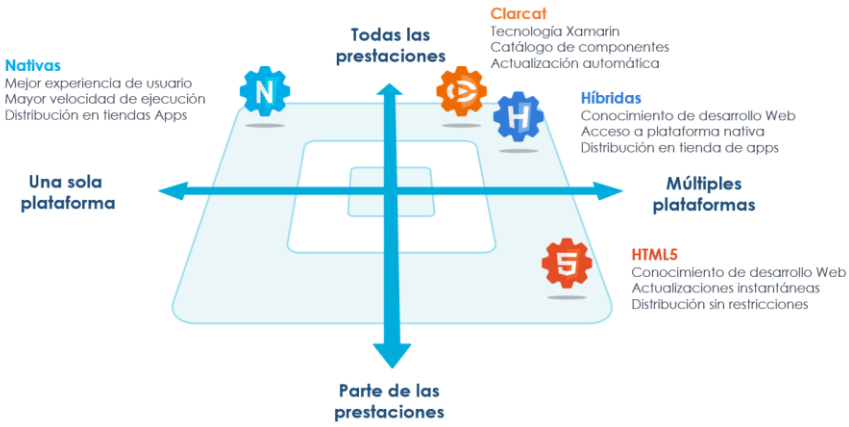
- Una aplicación híbrida está diseñada para funcionar en múltiples plataformas.
- Se escribe utilizando un único lenguaje de código estándar (como C# o una combinación de HTML5 y Javascript) que luego se compilará para ejecutarse en cada plataforma. Las interacciones específicas del dispositivo normalmente se gestionarán mediante el uso de componentes para ese sistema operativo.
- La mayor ventaja es que permiten la compatibilidad con múltiples sistemas operativos a un menor coste que nativas.
- Como desventajas no pueden sacar el máximo partido al dispositivo y SO, pudiendo dar lugar a desarrollos sumamente farragosos.



# Estrategias de desarrollo: WebApps

- Una WebApps es una aplicación desarrollada con tecnologías Web está diseñada para funcionar en múltiples plataformas, se empaqueta como una aplicación nativa pero se ejecuta en un navegador interno (como Webkit). Se escribe utilizando las tecnologías Web conocidas (una combinación de HTML5, CSS y JavaScript) que luego se empaquetará para ejecutarse en cada plataforma. Las aplicaciones se ejecutan dentro de contenedores destinados a cada plataforma y se basan en enlaces API que cumplen con los estándares para acceder a las capacidades de cada dispositivo, como sensores, datos, estado de la red, … mediante el uso de bibliotecas hibridas JavaScript/Nativa (como Cordova).
- La mayor ventaja es que permiten el desarrollo de una única aplicación para múltiples sistemas operativos al menor coste.
- Su mayor desventaja son las interacciones con el dispositivo que están muy limitadas, restringiendo su uso a aplicaciones de gestión donde los contenidos y el bajo coste son los factores principales.
- Una Progressive web apps (PWA) es una solución basada en la web tradicional, que mediante el uso de Service Workers y otras tecnologías se comportan más como aplicaciones normales que como aplicaciones web (pueden seguir ejecutándose en segundo plano sin tener que vivir dentro del navegador), cualquier usuario puede ‘instalarla’ en la pantalla de inicio de su dispositivo.

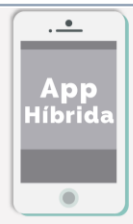
# Estrategias de desarrollo

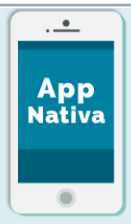





# Estrategias de desarrollo




| | | | |
|---|---|---|---|
| Coste de Desarrollo | Bajo | Medio | Alto |
| Tiempo de Desarrollo | Corto | Medio | Largo |
| Mantenimiento | Fácil | Medio | Complejo |
| Experiencia de Usuario | Buena | Bastante Buena | Excelente |
| Funcionalidad Offline | Compleja | Compleja | Fácil |
| Acceso al dispositivo | Parcial | Alto/Complejo | Completo |
| Velocidad | Rápida | Rápida | Muy Rápida |
| App Stores | No disponible | Disponible(con limitaciones) | Disponible |
| Portabilidad del código | Completa | Alta | Nula |
| Seguridad | Normal | Normal | Alta |

# INTRODUCCIÓN



## Introducción

- React Native (RN) es un marco de código abierto para crear aplicaciones de Android e iOS utilizando React y las capacidades nativas de la plataforma de aplicaciones.
- Facebook lanzó React Native en 2015 y lo ha mantenido desde entonces. La comunidad está explorando plataformas más allá de Android e iOS con proyectos como React Native Windows , React Native macOS y React Native Web .
- Con React Native, se utiliza JavaScript para acceder a las API de su plataforma, así como para describir la apariencia y el comportamiento de su interfaz de usuario utilizando componentes de React: paquetes de código reutilizable y anidable.
- React Native combina las mejores partes del desarrollo nativo con React, la mejor biblioteca de JavaScript de su clase para crear interfaces de usuario.
- Las primitivas de React se procesan en la interfaz de usuario de la plataforma nativa, lo que significa que la aplicación utiliza las mismas API de plataforma nativa que otras aplicaciones.

# Características

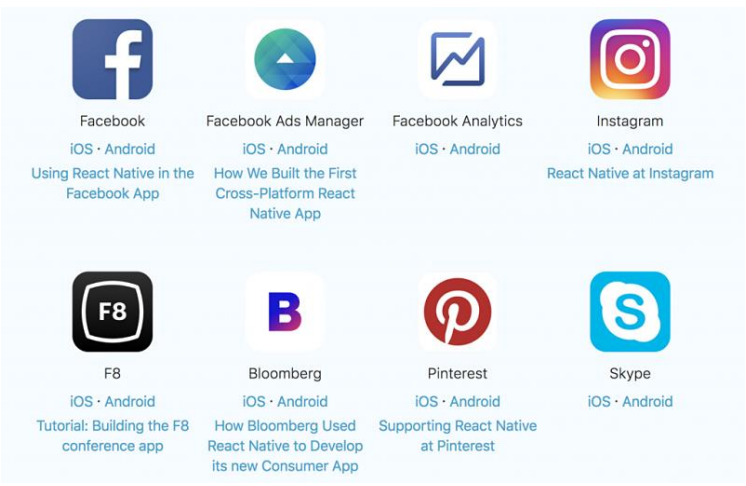
- Compatibilidad Cross-Platform: ya que la mayoría de las APIs de React Native lo son de por sí, lo cual ayuda a crear aplicaciones que pueden ser ejecutadas tanto en iOS como Android simultáneamente con el mismo código base.
- Funcionalidad nativa: funcionan de la misma manera que una aplicación nativa real creada para cada uno de los sistemas usando su lenguaje nativo propio, sin el uso de un WebView.
- Curva de aprendizaje sencilla: React Native es extremadamente fácil de leer y sencillo de aprender ya que se basa en los conceptos fundamentales del lenguaje JavaScript, siendo especialmente intuitivo para los ya expertos en dicho lenguaje.
- Actualizaciones instantáneas (para desarrollo y/o test): con la extensión de JavaScript, los desarrolladores tienen la flexibilidad de subir los cambios contenidos en la actualización directamente al dispositivo sin tener que pasar por las tiendas de aplicaciones propias de cada sistema y sus tediosos ciclos de procesos obligatorios previos.
- Experiencia positiva para el desarrollador: Ofrece varias características importantes como el Hot reloading que nos refresca la app en el momento en que guardamos cambios o el uso del debugger de las herramientas de desarrollados del navegador Google Chrome.



# Advertencias

- No es un framework que nos permite ejecutar aplicaciones que tenemos ya desarrolladas con ReactJS en un dispositivo móvil.
- No funciona como aplicaciones con PhoneGap o Cordova que nos permiten tener una página web y genera un WebView, sino que genera una aplicación nativa con un rendimiento casi similar al nativo.
- No va a convertir todo el código una aplicación nativa, transpilado con Java o con ObjectiveC.
- No va a evitar tener que tocar el código tanto en Android como en iOS, aunque permita integrar todas las partes que se tengan de ambos sistemas en una sola aplicación, con algunas diferencias pero con una parte del código compartido y escrito en JavaScript, utilizando JSX para definir las interfaces.

# Quien la utiliza

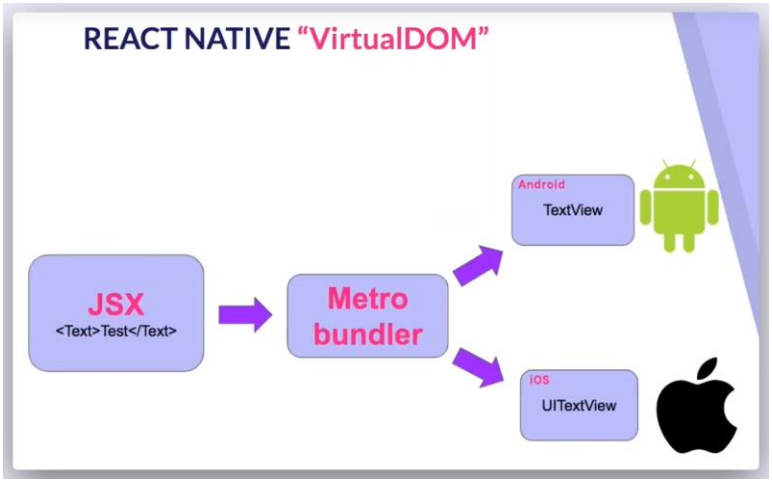





# Cómo funciona React Native

- En React existe un “VirtualDOM”, en el que tenemos nuestro JSX (una extensión que permite introducir XHTML embebido dentro del JavaScript), en el cual definimos los documentos HTML, y estos se transforman en componentes del navegador a través de JavaScript.
- Con React Native ocurre algo parecido, ya que tenemos nuestros componentes JSX, que van a ser distintos a los componentes HTML y que tendrán otros tags y otros nombres, ya que no estamos utilizando HTML.
- Lo que va a suceder es que el compilador que tiene React Native los va a convertir en elementos nativos de la interfaz para Android y para iOS, lo cual va a permitir que estas aplicaciones tengan un look and feel parecido a aplicaciones nativas, un rendimiento prácticamente igual y una experiencia de navegación y de usuario muy similar a las aplicaciones nativas, ya que lo que se está generando es interfaz nativa.

# Cómo funciona React Native

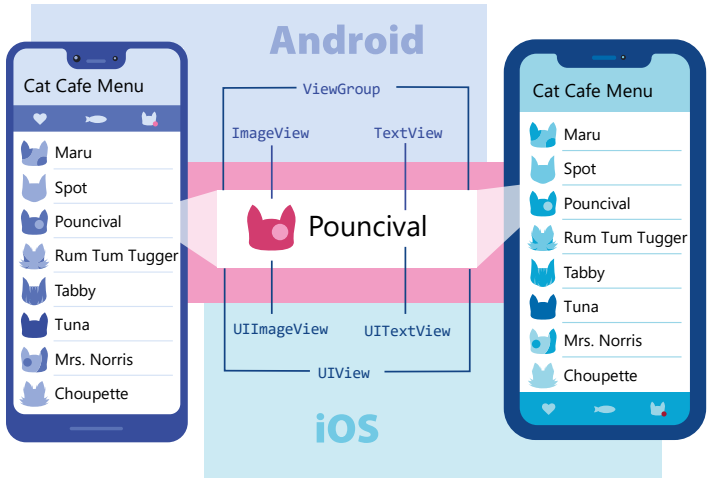





# Vistas

- En el desarrollo de Android e iOS, una vista es el componente básico de la interfaz de usuario: un pequeño elemento rectangular en la pantalla que se puede usar para mostrar texto, imágenes o responder a la entrada del usuario.
- Incluso los elementos visuales más pequeños de una aplicación, como una línea de texto o un botón, son tipos de vistas.
- Algunos tipos de vistas pueden contener otras vistas.
- En React Native hay que distinguir claramente entre componentes principales y componentes nativos

# Vistas






# Componentes nativos

- En el desarrollo de Android, se escriben vistas en Kotlin o Java; en el desarrollo de iOS, se usa Swift u Objective-C. Con React Native, se puede invocar estas vistas con JavaScript utilizando componentes de React. En tiempo de ejecución, React Native crea las vistas correspondientes de Android e iOS para esos componentes. Debido a que los componentes de React Native están respaldados por las mismas vistas que Android e iOS, las aplicaciones de React Native se ven, se sienten y funcionan como cualquier otra aplicación. Estos componentes respaldados por plataforma se denomina componentes nativos.
- React Native viene con un conjunto de componentes nativos esenciales y listos para usar que puedes utilizar para comenzar a crear tu aplicación hoy. Estos son los componentes principales de React Native .
- React Native también permite crear tus propios componentes nativos para Android e iOS para satisfacer las necesidades únicas de tu aplicación. También disponemos de un ecosistema próspero de estos componentes aportados por la comunidad.
  - https://reactnative.directory/

# Componentes principales

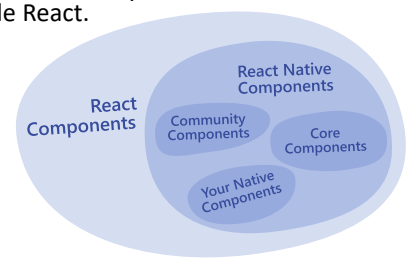
- React Native tiene muchos componentes básicos para todo, desde controles de formulario hasta indicadores de actividad.

| React Native | Android | iOS | Web | Descripción |
|---|---|---|---|---|
| <View> | <ViewGroup> | <UIView> |  | Un contenedor sin desplazamiento que admite el diseño con flexbox, estilo, algunos controles táctiles y de accesibilidad |
| <Text> | <TextView> | <UITextView> | <p> | Muestra, diseña y anida cadenas de texto e incluso maneja eventos táctiles |
| <Image> | <ImageView> | <UIImageView> | <img> | Muestra diferentes tipos de imágenes |
| <ScrollView> | <ScrollView> | <UIScrollView> |  | Un contenedor de desplazamiento genérico que puede contener múltiples componentes y vistas |
| <TextInput> | <EditText> | <UITextField> | <input type="text"> | Permite al usuario introducir texto |



# Componentes

- Debido a que React Native usa la misma estructura de API que los componentes de React, se deberá comprender las API de los componentes de React antes de comenzar.
- Con React, se pueden crear componentes utilizando clases o funciones. Originalmente, los componentes de clase eran los únicos componentes que podían tener estado. Pero desde la introducción de la API Hooks de React, puede agregar estado y más a los componentes de funciones.
- Los Hooks se introdujeron en React Native 0.59. y son la forma de cara al futuro de escribir los componentes de React.



# React Native Bridge

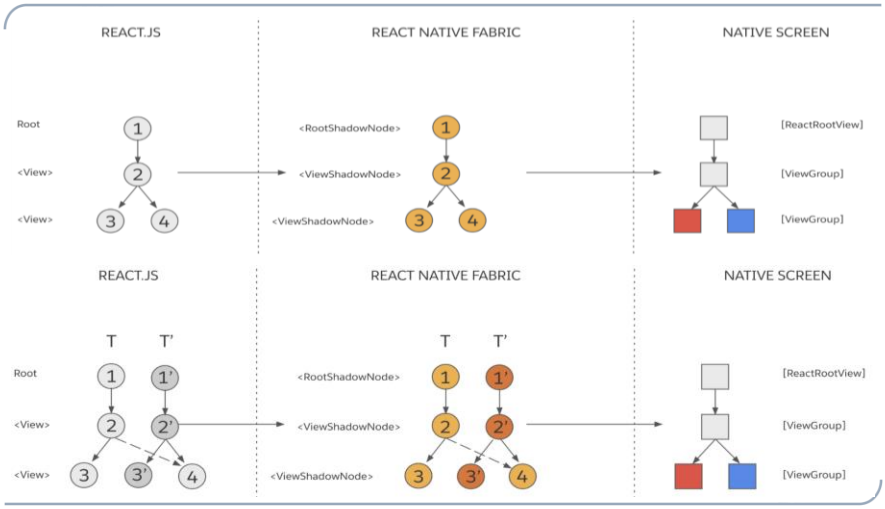
- El JavaScript se ejecuta nativamente, no se compila o transpila a Java o a ObjectiveC.
- Estos es así porque React Native está generando una especie de doble thread, en el cual tenemos uno corriendo todo el código nativo, toda la parte que sigue ejecutando módulos nativos como la interfaz o cualquier librería que tengamos integrada ya existente con programación en Android en iOS, y por otro tenemos corriendo una máquina virtual ejecutando JavaScript.
- El bridge de React Native es el que va a permitir la comunicación entre ambos threads. Es distinto al brige que podemos tener en aplicaciones HTML convertidas, que es el que da la funcionalidad a elementos nativos, ya que en este caso es este bridge el que va a comunicar en JavaScript con la parte nativa, para el paso de información o el acceso de cualquier componente del dispositivo.
- De esta forma vamos a conseguir también un rendimiento de ejecución, no solamente en la interfaz, cercano al nativo, y la parte de deterioro va a ser en la comunicación que tengamos que hacer entre una parte y la otra.



# React Native Bridge


- Módulos/códigos nativos: son todos los módulos necesarios (tanto de iOS como de Android) para que cuando hagamos nuestra RN app no tengamos que preocuparnos de escribir código nativo.
- Javascript VM: es la Virtual Machine de Javascript que ejecutará nuestro código JS. Tanto en iOS como en Android se utiliza JavascriptCore, que es el motor Javascript que utiliza Webkit para Safari. Esta pieza de software ya viene incluida en los dispositivos iOS, pero no en Android. Por lo que RN empaquetará esta pieza dentro del apk, lo que incrementa el tamaño en 3-4 megabytes.
- React Native Bridge: es un bridge React Native escrito en C++/Java y es responsable de comunicar los hilos de Javascript y de nativo. Entre ellos se hablan en un protocolo de mensajes.

# React Native VirtualDOM: Fabric






## ENTORNO

# Trabajar con React Native

- El equipo de React Native propone dos estrategias para trabajar y configurar el entorno de desarrollo:
  - Si se es nuevo en el desarrollo móvil, la forma más fácil de comenzar es con Expo CLI.
  - Si ya se está familiarizado con el desarrollo móvil, es posible que se desee trabajar a mas bajo nivel con React Native CLI.
- La estrategia es relevante porque marca la configuración inicial y las opciones disponibles.
- Debido a que no se crea ningún código nativo cuando se usa Expo para crear un proyecto, no es posible incluir módulos nativos personalizados más allá de las API de React Native y los componentes que están disponibles en la aplicación cliente de Expo. Si se necesitará incluir código nativo, se deberá "expulsar" la configuración para crear tus propias compilaciones nativas y para continuar trabajando en modo React Native CLI. Una vez "expulsada" la configuración no hay vuelta atrás.



# Expo CLI

- Expo (https://expo.dev/) es un conjunto de herramientas creado en torno a React Native y, si bien tiene muchas características, la más relevante para es que puede ayudar a escribir una aplicación React Native en minutos. Además, distribuyen una librería Expo SDK que facilita integrar capacidades de los móviles como Push Notifications, Facebook login, Instant updating, etc.
- Solo necesitarás una versión reciente de Node.js y un teléfono o emulador. Si se desea probar React Native directamente en el navegador web antes de instalar cualquier herramienta, se puede probar con https://snack.expo.dev/.
- Para instalarlo:
  - npm install -g expo-cli
- Para crear la aplicación:
  - expo init curso-app
  - cd curso-app

# Expo CLI

- Para arrancar el servidor de desarrollo:
  - npm start
- Se puede acceder al servidor de desarrollo a través del su terminal o usar la interfaz basada en web: http://localhost:19002/ (puerto por defecto).
- Expo CLI permite ejecutar la aplicación React Native en un dispositivo físico sin configurar un entorno de desarrollo o en un emulador.
- Para ejecutar la aplicación React Native en un dispositivo móvil hay que instalar la aplicación cliente Expo Go en el teléfono iOS o Android y conéctese a la misma red inalámbrica que el servidor de desarrollo.
- Para abrir el proyecto desde el terminal se usa la aplicación cliente Expo Go para escanear el código QR suministrado por el servidor.
- Expo Go permite ejecutar las aplicaciones en desarrollo sin su instalación en el disposivo, por lo que no requiere activarle el modo desarrollador.
- Si se desea ejecutar la aplicación en el simulador de iOS o en un dispositivo virtual de Android, es necesario tener instalados los emuladores igual que en React Native CLI.



# React Native CLI

- React Native CLI es una herramienta de línea de comandos que permite crear o actualizar proyectos React Native (no se recomienda su instalación global):
  - npx react-native upgrade
- Los proyectos típicos de React Native se componen esencialmente de un proyecto de Android, un proyecto de iOS y un proyecto de JavaScript. Requieren Xcode o Android Studio para continuar. Las instrucciones de instalación son un poco diferentes según el sistema operativo de desarrollo y si desea comenzar a desarrollar para iOS o Android.
  - Android:
    - Windows y Linux: Requiere Node, React Native CLI, un JDK y Android Studio.
    - macOS: Requiere Watchman, Node, React Native CLI, un JDK y Android Studio.
  - iOS (se requiere un Mac para crear proyectos con código nativo para iOS):
    - macOS: Se necesita Watchman, Node, React Native CLI, Xcode y CocoaPods.
- Se necesitará un dispositivo físico (conectado con USB) o, más comúnmente, un emulador para ejecutar la aplicación React Native. Tanto Android Studio (Android Virtual Devices) como XCode (iOS Simulator) disponen de las utilidades para instalar y gestionar los emuladores.

# React Native CLI

- React Native CLI se puede usar para generar un nuevo proyecto. Se puede acceder sin instalar nada globalmente usando npx:
  - npx react-native init curso-app
- Para ejecutar la aplicación React Native, primero se deberá iniciar Metro, el paquete de JavaScript que se incluye con React Native. Metro "toma un archivo de entrada y varias opciones, y devuelve un solo archivo JavaScript que incluye todo su código y sus dependencias". Para iniciar Metro (en su propia terminal):
  - npx react-native start
- Para iniciar la aplicación, en una nueva terminal:
  - npx react-native run-android ó npx react-native run-ios
- Si todo está configurado correctamente, se debería ver la aplicación ejecutándose en breve en el dispositivo físico (en modo desarrollador) o el emulador.
- Una vez modificada la aplicación, en la terminal Metro, se debe presionar la tecla r dos veces o seleccionando Reload desde el menú del desarrollador (Ctrl + M o ⌘M) para ver los cambios.



Core Components

# DISEÑO

# Componentes

- React y React Native usan JSX, una sintaxis que permite escribir elementos dentro de JavaScript.
- Los componentes permiten dividir la interfaz de usuario en piezas independientes y reutilizables, y pensar en cada pieza aisladamente.
- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente.
- Sigue un modelo de composición jerárquico con forma de árbol de componentes.
- La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- React Native incluye algunas docenas de componentes que se pueden usar de inmediato. Se pueden construir componentes más complejos combinando los componentes principales de muchas maneras interesantes.



# Componentes

- Conceptualmente, los componentes son como las funciones de JavaScript. Aceptan entradas arbitrarias (llamadas "props" abreviatura de propiedades) y devuelven elementos que describen lo que debería aparecer en la pantalla.
- La forma más sencilla de definir un componente es escribir una función de JavaScript:

```
function Saludo(props) {
  return <Text>Hola {props.name}</Text>;
}
```

- Esta función es un componente React válido porque acepta un único argumento de objeto "props" con datos y devuelve un elemento React.
- También puede usar una clase ES6 para definir un componente:

```
class Saludo extends Component {
  render() {
    return <Text>Hola {this.props.name}</Text>;
  }
}
```

# Propiedades

- Las propiedades permiten personalizar los componentes.
- Las propiedades toman la sintaxis de los atributos HTML.
- Las propiedades se encapsulan en un único objeto (array asociativo o diccionario), donde las claves son los nombres de los atributos y los valores son los valores asignados a los atributos, y se almacenan en la propiedad heredada this.props.
- Las propiedades son inmutables, de solo lectura. Aunque React es bastante flexible tiene algunas reglas estrictas: Los componentes no deben modificar las propiedades.
- Los componentes deben comportarse como “funciones puras“ porque no deben cambiar sus entradas y siempre devuelven el mismo resultado para las mismas entradas.



# Estado

- El estado es similar a las propiedades pero puede cambiar, es privado y está completamente controlado por el componente.
- El estado local era una característica disponible solo para las clases.
- El estado es la propiedad this.state, que se hereda de la clase componente, y es un objeto que encapsula todas la propiedades de estado que se vayan a utilizar en el render(). Los cambios en el estado provocan que se vuelva a ejecutar el método render.
- El estado se inicializa en el constructor, que obligatoriamente tiene que invocar al constructor del padre para propagar las propiedades:

```
class Contador extends React.Component {
  constructor(props) {
    super(props);
    this.state = { contador: +props.init, delta: +props.delta };
  }
```

# Hook de estado

- useState es un Hook (función) que se llama dentro de un componente de función para agregarle un estado local y que React lo mantenga
- useState recibe como argumento el estado inicial y devuelve un par variable/función: el valor de estado actual (referencia) y la función que permite actualizarlo.

```
function Coordenadas() {
  let [punto, setPunto] = useState({x: 0, y: 0});
```

- La sintaxis de desestructuración de un array permite dar diferentes nombres a las variables y funciones de estado que declaramos llamando a useState.



# Hook de estado

- En un componente de función no existe this por lo que no se puede asignar o leer this.state, se usan directamente las variables sin this.
- La función de actualización de estado recibe el nuevo valor de estado y maneja la complejidad del cambio de estado y re-renderizados. Se puede llamar desde un controlador de eventos o desde cualquier otro lugar.
- Se puede usar el Hook de estado más de una vez en un mismo componente.
- Se pueden crear Hooks personalizados para reutilizar el comportamiento con estado entre diferentes componentes.

# Hook de estado

```
export default function Contador(props) {
  const [contador, setContador] = useState(props.init ?? 0);
  return (
    <div>
      <h1 data-testid="pantalla">{contador}</h1>
      <p>
        <input type="button" value="-"
          onClick={() => setContador(contador - 1)} />
        <input type="button" value="+"
          onClick={() => setContador(contador + 1)} />
      </p>
    </div>
  );
}
```



# Eventos

- Algunos componentes desencadenan eventos o devoluciones de llamada ( en respuesta a la entrada del usuario.
- En React los eventos de los elementos se manejan de una forma muy similar a como se manejan los eventos de los elementos DOM, pero con algunas diferencias importantes:
  - Los eventos se nombran usando camelCase, en lugar de minúsculas.
  - Con JSX se pasa una función como controlador de eventos: el objeto, no la invocación.
- Sintaxis:
  - JSX: <Button onPress={this.sube}>Sube</Button>

# Estilo

- Con React Native, se diseñan las aplicación usando JavaScript. Todos los componentes principales aceptan una propiedad llamada style.
- Los nombres y valores de estilo generalmente coinciden con la forma en que funciona CSS en la web, excepto que los nombres se escriben usando mayúsculas y minúsculas, por ejemplo, backgroundColor en lugar de background-color.
- El propiedad style puede ser un objeto JavaScript plano. También se puede pasar una matriz de estilos: el último estilo de la matriz tiene prioridad, por lo que se puede usar esto para heredar estilos.
- Un componente puede especificar el diseño de sus elementos secundarios utilizando el algoritmo Flexbox.
- Flexbox está diseñado para proporcionar un diseño uniforme en diferentes tamaños de pantalla.
- La altura y el ancho de un componente determinan su tamaño en la pantalla, se establecen como propiedades de estilo.



# Vista

- El componente View es el más fundamental para crear una interfaz de usuario, es un contenedor que admite el diseño con flexbox, estilos, algunos controles táctiles y de accesibilidad.

  <View></View>
- View se asigna directamente a la vista nativa equivalente en cualquier plataforma en la que se esté ejecutando React Native, ya sea un UIView, , android.view, etc.
- View está diseñado para anidarse dentro de otras vistas y puede tener de 0 a muchos hijos de cualquier tipo.
- El contenido de la vista que desborda el área de visualización no es accesible, no dispone de Scroll automático.
- En caso de ser previsible dicho desbordamiento es conveniente utilizar una especialización como ScrollView que si da soporte a los desplazamientos.

# ScrollView

- ScrollView es un contenedor de desplazamiento genérico que puede contener múltiples componentes y vistas. Los elementos desplazables pueden ser heterogéneos y puede desplazarse tanto vertical como horizontalmente (estableciendo la propiedad horizontal).

  <ScrollView></ScrollView>
- ScrollViews se puede configurar para permitir la paginación a través de las vistas mediante gestos de deslizamiento mediante el uso de la propiedad pagingEnabled. El deslizamiento horizontal entre vistas también se puede implementar en Android usando el componente ViewPager .
- En iOS, se puede usar un ScrollView con un solo elemento para permitir que el usuario haga zoom en el contenido. Configurando las propiedades maximumZoomScale y minimumZoomScale, el usuario podrá usar gestos de pellizcar y expandir para acercar y alejar.



# ScrollView

- Si desea representar un conjunto de datos dividido en secciones lógicas, tal vez con encabezados de sección, similares a UITableViewlos de iOS, entonces una SectionList es el camino a seguir. ScrollView funciona mejor para presentar una pequeña cantidad de cosas de un tamaño limitado. Se representan todos los elementos y vistas del ScrollView, incluso si no se muestran actualmente en la pantalla.
- Si se tiene una larga lista de elementos que no caben en la pantalla, se debe utilizar componentes alrededor de <VirtualizedList> en su lugar. La virtualización mejora enormemente el consumo de memoria y el rendimiento de listas grandes al mantener una ventana de representación finita de elementos activos y reemplazar todos los elementos fuera de la ventana de representación con espacios en blanco del tamaño adecuado.
- React Native proporciona un conjunto de componentes para presentar listas de datos. Los principales son FlatList y SectionList.

# FlatList

- El componente FlatList muestra una lista desplegable de datos cambiantes, pero con una estructura similar. FlatList funciona bien para largas listas de datos, donde la cantidad de elementos puede cambiar con el tiempo. A diferencia del más genérico ScrollView, el FlatList solo representa los elementos que se visualizan actualmente en la pantalla, no todos los elementos a la vez.
- El componente FlatList requiere dos propiedades:
  - data es la fuente de información de la lista.
  - renderItem toma cada elemento de dato y devuelve el componente formateado para renderizarlo.

```
<FlatList
    data={[{key: 'Devin'},{key: 'Dan'}, {key: 'Dominic'}, {key: 'Jackson'}, {key:
        'James'}, {key: 'Joel'}, {key: 'John'}, {key: 'Jillian'}, {key: 'Jimmy'}, {key:
        'Julie'},]}
    renderItem={({item}) => <Text style={styles.item}>{item.key}</Text>}
  />
```



# SectionList

- Si desea representar un conjunto de datos dividido en secciones lógicas, tal vez con encabezados de sección, similares a UITableViewlos de iOS, entonces una SectionList es el camino a seguir.
- Es una vista de alto rendimiento para renderizar listas con secciones, compatible con las funciones más útiles:
  - Totalmente multiplataforma.
  - Visibilidad configurable con callbacks.
  - Soporte de encabezados y pies de lista.
  - Soporte de encabezados y separadores de sección.
  - Soporte de separadores de elementos.
  - Compatibilidad con la representación de elementos y datos heterogéneos.
  - Pull para actualizar.
  - Scroll loading.

# SectionList

```
const DATA = [
  { title: 'Main dishes', data: ['Pizza', 'Burger', 'Risotto'] },
  { title: 'Sides', data: ['French Fries', 'Onion Rings', 'Fried Shrimps'] },
  { title: 'Drinks', data: ['Water', 'Coke', 'Beer'] },
  { title: 'Desserts', data: ['Cheese Cake', 'Ice Cream'] },
]
<SectionList
  sections={DATA}
  keyExtractor={(item, index) => item + index}
  renderItem={({ item }) => (
    <View style={styles.item}><Text style={styles.title}>{item}</Text></View>
  )}
  renderSectionHeader={({ section: { title } }) => (
    <Text style={styles.header}>{title}</Text>
  )}
/>
```



# StatusBar

- El componente StatusBar sirve para controlar la barra de estado de la aplicación. La barra de estado es la zona, generalmente en la parte superior de la pantalla, que muestra la hora actual, la información de la red Wi-Fi y movil, el nivel de la batería y/u otros íconos de estado.

```
const STYLES = ['default', 'dark-content', 'light-content']
const TRANSITIONS = ['fade', 'slide', 'none']
const [hidden, setHidden] = useState(false)

<StatusBar animated hidden={hidden} style='light'
  barStyle={STYLES[0]} showHideTransition={TRANSITIONS[0]}
/>
<Button title='Cambia' onPress={() => setHidden(!hidden)} />
```

# Manejo de entradas y salidas

- Manejo de salidas
  - Text: componente para mostrar texto.
  - Image: componente para mostrar imágenes.
  - ImageBackground: componente para mostrar imágenes que permite agregarle los elementos secundarios que se desee superponer.
  - ActivityIndicator: componente que muestra un indicador de carga circular animado.
- Manejo de entradas
  - TextInput: componente para introducir texto en la aplicación a través de un teclado.
  - Switch: componente para introducir una entrada booleana.
  - Button: componente botón básico para manejar toques.



# Text

- Text es el componente de React para mostrar texto que admite anidamiento, estilo y manejo táctil.

```
<Text style={styles.titleText} onPress={onPressTitle}>
    Hola mundo
    {"\n"}
</Text>
```

- Tanto Android como iOS permiten mostrar texto con formato al anotar rangos de una cadena con un formato específico como texto en negrita o en color ( NSAttributedStringen iOS, SpannableStringen Android). En la práctica, esto es muy tedioso. Para React Native, se decidio usar el paradigma web para esto, donde se puede anidar texto para lograr el mismo efecto.

```
<Text style={styles.bold}>
    I am bold <Text style={styles.red}> and red</Text>
</Text>
```

# Text

- El elemento <Text> es único en relación con el diseño: todo lo que hay dentro ya no usa el diseño de Flexbox sino el diseño de texto. Esto significa que los elementos dentro de un <Text> ya no son rectángulos, sino que se ajustan cuando ven el final de la línea.
- En la web, la forma habitual de establecer una familia de fuentes y un tamaño para todo el documento es aprovechar las propiedades de CSS heredadas: Todos los elementos del documento heredarán esta fuente a menos que ellos o uno de sus padres especifique una nueva regla.
- React Native es más estricto: no se puede tener un texto directamente en un <View>, se debe envolver todos los textos dentro de componentes <Text>.
- React Native todavía tiene el concepto de herencia de estilo, pero se limita a los subárboles de texto, por lo que se pierde la capacidad de configurar una fuente predeterminada y un tamaño para un subárbol completo de componentes (View).



# Text

- En React Native, la propiedad de estilo fontFamily solo acepta un solo nombre de fuente, a diferencia de CSS.
- La forma recomendada de usar fuentes y tamaños consistentes en la aplicación es crear un componente base MyStyledText que los incluya y usar este componente en toda aplicación.

```
const MyStyledText = ({ children }) => (
  <Text style={styles.appText}>{children}</Text>
)
```

- También se puede usar este componente para crear componentes más específicos para otros tipos de texto, de esta manera nos aseguramos que obtengamos los estilos de un componente de nivel superior, pero nos deja la capacidad de agregarlos/anularlos en casos de uso específicos.

```
const MyAppHeaderText = ({ children }) => (
  <MyStyledText>
    <Text style={{ fontSize: 20 }}> {children} </Text>
  </MyStyledText>
)
```

# Image

- <Image> es el componente de React para mostrar diferentes tipos de imágenes: importadas, imágenes de red, recursos estáticos, imágenes locales temporales e imágenes del disco local, como de la cámara.

```
import importImg from '../assets/favicon.png'
<View>
  <Image source={importImg} style={{ width: 40, height: 40 }} />
  <Image source={require('../assets/favicon.png')} style={{ width: 40, height: 40 }}
/>
  <Image source={{ uri: 'https://reactnative.dev/img/tiny_logo.png', width: 100,
height: 100 }} />
  <Image source={{ uri:
'
TAqAAAAEXRFWHRTb2Z0d2FyZQBwbmdjcnVzaEB1SfMAAABQSURBVGje7dSxCQB
ACARB+2/ab8BEeQNhFi6WSYzYLYudDQYGBgYGBgYGBgYGBgZmcvDqYGBgmh
ivGQYGBgYGBgYGBgYGBgbmQw+P/eMrC5UTVAAAAABJRU5ErkJggg==' ,
width: 40, height: 40 }} />
</View>
```



# Image

- La propiedad source también puede contener varias URL remotas, especificadas junto con su ancho y alto y potencialmente con escala u otros argumentos de URI. El lado nativo luego elegirá lo mejor uri para mostrar en función del tamaño medido del contenedor de la imagen. Se puede agregar una propiedad cache para controlar cómo interactúa la solicitud en red con el caché local.
- Los formatos admitidos actualmente son png, jpg, jpeg, bmp, gif, webp, psd(solo iOS). Además, iOS admite varios formatos de imagen RAW. Al compilar su propio código nativo, GIF y WebP no son compatibles de manera predeterminada en Android, se deberá agregar algunos módulos opcionales.

# ImageBackground

- Una solicitud de característica común de los desarrolladores familiarizados con la web es background-image.
- Para manejar este caso de uso, sepuede usar el componente <ImageBackground>, que tiene las mismas propiedades que <Image>, y permite agregarle los elementos secundarios que se desee superponer.

```
<ImageBackground source={importImg} resizeMode='cover'
style={{ flex: 1 }}>
  <MyAppHeaderText>{title}</MyAppHeaderText>
</ImageBackground>
```



# ActivityIndicator

- Componente que muestra un indicador de carga circular animado.
- Se debe establecer el color a través de la propiedad color del componente (no en el estilo). El tamaño se establece en la propiedad size del componente (no en el estilo): 'small' (por defecto) y 'large'.

```
export const AjaxActivityIndicator = () => (
  <View
    style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
    <ActivityIndicator size='large' color='lightskyblue' />
  </View>
)
```

# TextInput

- TextInput es el componente central que permite al usuario introducir texto en la aplicación a través de un teclado. Las propiedades brindan capacidad de configuración para varias funciones, como la corrección automática, el uso automático de mayúsculas, el texto de marcador de posición y diferentes tipos de teclado, como un teclado numérico.
- El caso de uso más básico es mostrar un TextInput y suscribirse al evento onChangeText para leer la entrada del usuario. También hay otros eventos, como onSubmitEditing (cuando se presiona el botón Enviar de la entrada de texto) y onFocus. Los dos métodos expuestos a través del elemento nativo son .focus() y .blur() que ponen o quitan el foco en el TextInput mediante programación.



# TextInput

```
const EntradasTextInput = () => {
  const [text, onChangeText] = React.useState('Hola mundo')
  const [number, onChangeNumber] = React.useState(null)
  return (
    <View>
      <TextInput style={styles.input} multiline numberOfLines={4}
        value={text} onChangeText ={onChangeText}
      />
      <Text>{text}</Text>
      <TextInput style={styles.input} value={number} onChangeText={onChangeNumber}
        placeholder='introduce un número' keyboardType='numeric'
      />
      <Text>{number}</Text>
    </View>
  )
}
```

# Teclado

- El API Keyboard permite escuchar eventos nativos del teclado y reaccionar ante ellos. Los eventos disponibles son: keyboardWillShow, keyboardDidShow, keyboardWillHide, keyboardDidHide, keyboardWillChangeFrame, keyboardDidChangeFrame.
- El método dismiss() cierra el teclado activo y elimina el foco.

```
const DemoTeclado = () => {
 const [keyboardStatus, setKeyboardStatus] = useState('Sin usar')
 useEffect(() => {
  const showSubscription = Keyboard.addListener('keyboardDidShow', () => { setKeyboardStatus('Visible') })
  const hideSubscription = Keyboard.addListener('keyboardDidHide', () => { setKeyboardStatus('Oculto') })
  return () => {
   showSubscription.remove()
   hideSubscription.remove()
  }
 }, [])
 return (
  <View style={{ flex: 1 }}>
   <Text style={styles.status}>{keyboardStatus}</Text>
   <TextInput style={{ margin: 5, borderWidth: 0.5, }} onSubmitEditing={Keyboard.dismiss} />
  </View>
 )
}
```



# KeyboardAvoidingView

- El componente KeyboardAvoidingView sirve para resolver el problema común de las vistas que deben apartarse del camino del teclado virtual. Se puede ajustar automáticamente su altura, posición o relleno inferior en función de la altura del teclado.

```
const KeyboardAvoidingComponent = () => {
 return (
  <KeyboardAvoidingView behavior={Platform.OS === 'ios' ? 'padding' : 'height'}
   style={{ flex: 1 }}>
   <TouchableWithoutFeedback onPress={Keyboard.dismiss}>
    <View>
     <EntradasTextInput />
     <Button title='Submit' onPress={() => null} />
    </View>
   </TouchableWithoutFeedback>
  </KeyboardAvoidingView>
 )
}
```

# Switch

- Representa una entrada booleana y se visualiza como un botón deslizante.
- Este componente requiere un controlado en onValueChange para que se actualice la propiedad value para que refleje las acciones del usuario. Si el estado de la propiedad value no se actualiza, el componente seguirá representando el value original en lugar del resultado esperado de las acciones del usuario.

```
const ToggleButton = () => {
 const [isEnabled, setIsEnabled] = React.useState(false)

 return (
  <View>
   <Switch
    trackColor={{ false: '#767577', true: '#81b0ff' }}
    thumbColor={isEnabled ? '#f5dd4b' : '#f4f3f4'}
    ios_backgroundColor='#3e3e3e'
    value={isEnabled} onValueChange={() => setIsEnabled(previousState => !previousState)}
   />
  </View>
 )
}
```



# Manejo de toques

- Los usuarios interactúan con las aplicaciones móviles principalmente a través del tacto. Pueden usar una combinación de gestos, como tocar un botón, desplazarse por una lista o hacer zoom en un mapa. React Native proporciona componentes para manejar todo tipo de gestos comunes, así como un sistema integral de respuesta de gestos para permitir un reconocimiento de gestos más avanzado, aunque probablemente baste con el botón básico.
- El componente Button proporciona un botón básico que se representa bien en todas las plataformas.

```
<Button title="No tocar"
 onPress={() => { alert('Booom!!!');  }}
/>
```

- La propiedad title contiene el texto mostrado en el botón. Al presionar el botón, se llamará a la función onPress.

# Manejo de toques

- Si el botón básico no es suficiente, React Native proporciona componentes "tocables" que brindan la capacidad de capturar gestos de toque y pueden mostrar reacciones (efectos visuales) cuando se reconoce un gesto.
- Sin embargo, estos componentes no proporcionan ningún estilo predeterminado, por lo que deberá envolver a otros componentes.
- El componente "tocable" a elegir dependerá del tipo de reacción que se desee proporcionar:
  - En general, se puede usar TouchableHighlight en cualquier lugar donde se usaría un botón o enlace en la web. El fondo de la vista se oscurecerá cuando el usuario presione el botón.
  - En Android, se puede usar TouchableNativeFeedback para mostrar las ondas de superficie que responden al toque del usuario.
  - Se puede usar TouchableOpacity para reducir la opacidad del botón, lo que permite que se vea el fondo mientras el usuario mantiene la pulsación.
  - TouchableWithoutFeedback permite manejar un gesto de toque pero sin efectos visuales.
- En algunos casos, es posible que se desee detectar cuándo un usuario presiona y mantiene presionada una vista durante un período de tiempo determinado. Estas pulsaciones largas se pueden gestionar con onLongPress en cualquiera de los componentes "tocables".



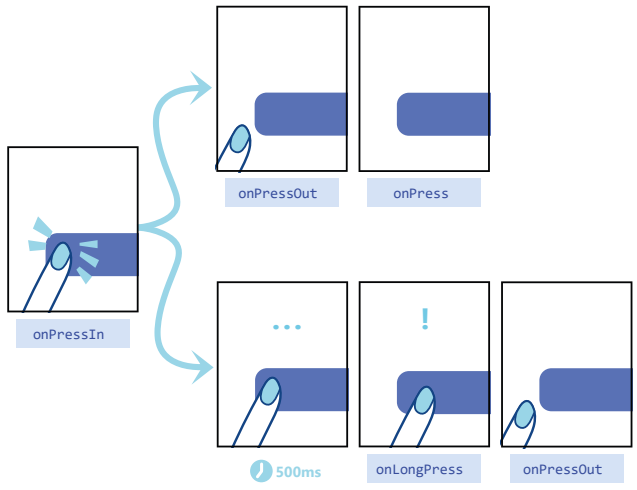
# Manejo de toques

- Pressable es un contenedor de componentes básicos que puede detectar varias etapas de interacciones de prensa en cualquiera de sus hijos definidos.

```
<Pressable onPress={onPressFunction}>
  <Text>I'm pressable!</Text>
</Pressable>
```

- En un elemento envuelto por Pressable:
  - onPressIn se llama cuando se activa una pulsación.
  - onPressOut se llama cuando el gesto de pulsación está desactivado.
- Después de presionar onPressIn, sucederá una de dos cosas:
  - La persona quitará su dedo, disparando onPressOut primero y onPress a continuación.
  - Si la persona deja el dedo más de 500 milisegundos antes de retirarlo, se activa onLongPress (onPressOut se seguirá disparando cuando quiten el dedo).

# Manejo de toques



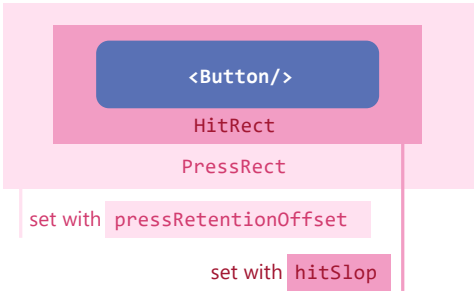



# Manejo de toques

Los dedos no son los instrumentos más precisos, y es común que los usuarios activen accidentalmente el elemento incorrecto o se pierdan el área de activación.

Para ayudar, Pressable tiene un HitRect opcional que se puede usar para definir la distancia máxima del toque del elemento envuelto. El toque puede comenzar en cualquier lugar dentro de la zona HitRect.

PressRect permite que las presiones se muevan más allá del elemento y HitRect mientras mantiene la activación y es elegible para "presionar": piense en deslizar el dedo lentamente lejos de un botón que está presionando.



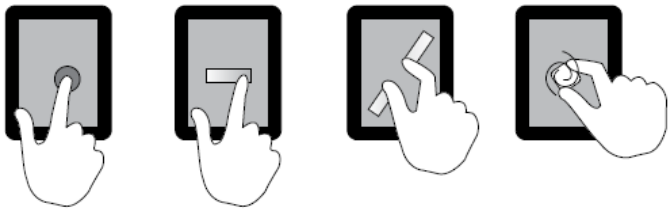
# Modal

- El componente Modal es una forma básica de presentar contenido sobre una vista envolvente.

```
const ModalView = ({ texto }) => {
 const [modalVisible, setModalVisible] = React.useState(false)
 return (
  <View style={modalStyles.centeredView}>
   <Modal animationType='slide' transparent visible={modalVisible}>
    <View style={modalStyles.centeredView}>
     <View style={modalStyles.modalView}>
      <Text style={modalStyles.modalText}>{texto}</Text>
      <Button title='Cerrar' onPress={() => setModalVisible(!modalVisible)} />
     </View>
    </View>
   </Modal>
   <Button title='Ver modal' onPress={() => setModalVisible(true)} />
  </View>
 )
}
```



# Gestos



Los marcos basados en el tacto deben entender los siguientes tipos de gestos táctiles:

- **Tap**: Un solo toque en la pantalla.
- **Double tap**: Dos toques rápidos en la pantalla.
- **Swipe** (deslizar): Mover un solo dedo por la pantalla de izquierda a derecha o de arriba a abajo.
- **Pinch or spread** (pellizcar o separar): Tocar la pantalla con dos dedos y juntarlos en un movimiento de pellizco, o separarlos para invertir la acción.
- **Rotate** (girar): colocar dos dedos en la pantalla y girarlos en el sentido de las agujas del reloj o en el contrario, normalmente para girar un objeto en la pantalla

# Sistema de respuesta a gestos

- El sistema de respuesta de gestos administra el ciclo de vida de los gestos en la aplicación. Un toque puede pasar por varias fases a medida que la aplicación determina cuál es la intención del usuario. Por ejemplo, la aplicación necesita determinar si el toque se está desplazando, deslizando sobre un widget o tocando. Esto incluso puede cambiar durante el trascurso de un toque. También puede haber múltiples toques simultáneos.
- El sistema de respuesta táctil es necesario para permitir que los componentes negocien estas interacciones táctiles sin ningún conocimiento adicional sobre sus componentes principales o secundarios.
- Cada acción debe tener los siguientes atributos:
  - Comentarios/resaltado: mostrar al usuario qué se está manejando su toque y qué sucederá cuando suelte el gesto
  - Capacidad de cancelación: al realizar una acción, el usuario debería poder abortarla a mitad del toque arrastrando el dedo
- Estas funciones hacen que los usuarios se sientan más cómodos al usar una aplicación, ya que les permite experimentar e interactuar sin temor a cometer errores.
- El sistema de respuesta puede ser complicado de usar. React Native proporciona la abstracción Touchable y componentes especifico que utilizan el sistema de respuesta y permiten configurar las interacciones de toque de forma declarativa.



# Ciclo de vida de la respuesta

- Una vista puede convertirse en la respuesta táctil al implementar los métodos de negociación correctos. Hay dos métodos para preguntarle a la vista si quiere convertirse en respondedor:
  - View.props.onStartShouldSetResponder: (evt) => true, ¿Esta vista quiere convertirse en respondedor al comienzo de un toque?
  - View.props.onMoveShouldSetResponder: (evt) => true, Llamado para cada movimiento táctil en la vista cuando no es el que responde: ¿esta vista quiere "reclamar" la capacidad de respuesta táctil?
- Si la Vista devuelve verdadero e intenta convertirse en el respondedor, ocurrirá una de las siguientes cosas:
  - View.props.onResponderGrant: (evt) => {}, la vista es ahora el respondedor, responde a los eventos táctiles. Este es el momento de destacar y mostrar al usuario lo que está pasando
  - View.props.onResponderReject: (evt) => {}, hay otro respondedor en este momento y no lo soltará

# Ciclo de vida de la respuesta

- Si la vista es respondedor, se pueden disparar los eventos:
  - View.props.onResponderMove: (evt) => {}, El usuario está moviendo su dedo
  - View.props.onResponderRelease: (evt) => {}, Se dispara al final del toque, es decir, "touchUp"
  - View.props.onResponderTerminationRequest: (evt) => true, Otro quiere convertirse en respondedor. ¿Debería esta vista liberar al respondedor? Devolver verdadero permite la liberación
  - View.props.onResponderTerminate: (evt) => {}, Se a quitado el respondedor de la Vista. Puede haber sido asumido por otras vistas después de una llamada a onResponderTerminationRequest, o puede ser reclamado por el sistema operativo sin preguntar (sucede con el centro de control/centro de notificaciones en iOS)



# PanResponder

- Solo un único componente puede responder a eventos táctiles a la vez: el componente que responde a los eventos posee un "bloqueo de interacción" global.
- La API PanResponder nos ayuda a administrar qué componente posee este bloqueo a través de un conjunto de devoluciones de llamada. PanResponder concilia varios toques en un solo gesto. Hace que los gestos de un solo toque sean resistentes a toques adicionales y se puede usar para reconocer gestos multitáctiles básicos.
- A cada una de las devoluciones de llamada se le pasa un objeto evento y gestureState que contiene información sobre los eventos táctiles (por ejemplo, posición y velocidad).

  onPanResponderMove: (event, gestureState) => {}

# PanResponder

- Un evento nativo es un evento táctil sintético con forma de PressEvent.
- El objeto gestureState contiene:
  - stateID: ID del estado del gesto: persiste siempre que haya al menos un toque en la pantalla
  - moveX, moveY: las últimas coordenadas de pantalla del toque movido recientemente
  - x0, y0: las coordenadas de la pantalla de la concesión del respondedor
  - dx, dy: distancia acumulada del gesto desde que comenzó el toque
  - vx, vy: velocidad actual del gesto
  - numberActiveTouches: número de toques actualmente en la pantalla



# PanResponder

- Para crear un PanResponder se utiliza PanResponder.create(callbacksObject). El resultado es un conjunto de propiedades que se pueden pasar al View (propiedades de manejo de eventos táctiles de nivel inferior). Por lo general, se envuelve el resultado con useRef, ya que solo se debe crear uno único PanResponder para el ciclo de vida del componente.

```
const panResponder = React.useRef(
  PanResponder.create({
    onStartShouldSetPanResponder: (evt, gestureState) => true,
    onStartShouldSetPanResponderCapture: (evt, gestureState) => true,
    onMoveShouldSetPanResponder: (evt, gestureState) => true,
    onMoveShouldSetPanResponderCapture: (evt, gestureState) => true,
  ).current;

  return <View {...panResponder.panHandlers} />;
```

# PanResponder

- El conjunto completo de devoluciones de llamada que se pueden pasar son:
  - onStartShouldSetPanResponder: (event, gestureState) => {}
  - onStartShouldSetPanResponderCapture: (event, gestureState) => {}
  - onMoveShouldSetPanResponder: (event, gestureState) => {}
  - onMoveShouldSetPanResponderCapture: (event, gestureState) => {}
  - onPanResponderReject: (event, gestureState) => {}
  - onPanResponderGrant: (event, gestureState) => {}
  - onPanResponderStart: (event, gestureState) => {}
  - onPanResponderEnd: (event, gestureState) => {}
  - onPanResponderRelease: (event, gestureState) => {}
  - onPanResponderMove: (event, gestureState) => {}
  - onPanResponderTerminate: (event, gestureState) => {}
  - onPanResponderTerminationRequest: (event, gestureState) => {}
  - onShouldBlockNativeResponder: (event, gestureState) => {}



# Bibliotecas de componentes

- Más allá de los componentes principales de React Native, hay muchas bibliotecas de componentes de código abierto. Las bibliotecas varían en tamaño y forma, desde componentes individuales que unen la funcionalidad de la plataforma nativa (por ejemplo, mapas, videos, etc.) hasta grandes colecciones de componentes.
- Al comenzar una nueva aplicación, vale la pena considerar usar una colección de componentes. Estos proporcionan potencialmente decenas o cientos de componentes multiplataforma diseñados consistentemente y listos para usar para acelerar su proceso de desarrollo.
- Las colecciones de componentes aceleran el desarrollo al permitir concentrarse en la arquitectura y la lógica empresarial de la aplicación, sin centrarse en los matices del diseño y el estilo en React Native. Si no se está trabajando con un diseñador o equipo de diseño dedicado, estas es una excelente manera de crear una aplicación con una apariencia muy pulida. Sin embargo, las colecciones de componentes no son para todas las aplicaciones: pueden agregar mucho código y complejidad, y es posible que no se puedan personalizar tanto como se desee.
- Actualmente, las colecciones de componentes más grandes son:
  - React Native Elements
  - NativeBase

# ESTILOS



## Estilo

- Con React Native, diseñas tu aplicación usando JavaScript. Todos los componentes principales aceptan una propiedad llamada style.
- Los nombres y valores de estilo generalmente coinciden con la forma en que funciona CSS en la web, excepto que los nombres se escriben usando mayúsculas y minúsculas, por ejemplo, backgroundColor en lugar de background-color.
- El propiedad style puede ser un objeto JavaScript plano:

```
style={{ color: 'blue', fontWeight: 'bold', fontSize: 20 }}
```

- A medida que el estilo crece en complejidad, a menudo es más fácil de usar un StyleSheet, una abstracción similar a las hojas de estilo CSS, para definir varios estilos en un solo lugar facilitando su mantenibilidad y reutilización.

```
const styles = StyleSheet.create({
  container: { flex: 1, padding: 5,  backgroundColor: "#eaeaea", },
  title: { fontSize: 30, borderColor: "#20232a", borderRadius: 6, },
  bigBlue: { color: 'blue', fontWeight: 'bold', fontSize: 20 },
});
```

# Estilo

- Una vez definido el objeto estilo se puede reutilizar tantas veces como sea necesario y, si se exporta e importa, incluso en cualquier sitio.

```
<View style={styles.container}>
    <Text style={styles.title}>Titulo</Text>
    <Text style={styles.bigBlue}>Contenido</Text>
</View>
```

- El propiedad style puede recibir una matriz de estilos: el último estilo de la matriz tiene prioridad, por lo que se puede usar para heredar estilos.

```
<Text style={[styles.title, styles.bigBlue]}>Titulo Azul</Text>
```

- Un patrón común es hacer que su componente acepte un style que a su vez se usa para diseñar sus subcomponentes, se puede usar esto para simular que los estilos "caigan en cascada" como ocurre en CSS.
- Las propiedades del objeto StyleSheet pueden equivaler a los Class de CSS, asignarles nombres conceptuales es una buena manera de agregar semántica a los componentes de bajo nivel en la función de renderizado y hace que el código sea más fácil de entender.



# Colores

- Las propiedades de color suelen coincidir con el funcionamiento de CSS en la web. Los colores se pueden expresar con:
  - RGB (Rojo Verde Azul) y RGBA (canal alfa: opacidad): admite tanto en notación hexadecimal (1 ó 2 dígitos) como funcional (con rgb() y rgba()): '#f0f', '#ff00ff00', 'rgb(255, 0, 255)', 'rgba(255, 0, 255, 1.0)'
  - HSL (Tono Saturación Luminosidad): en notación funcional (con hsl() y hsla()): 'hsl(360, 100%, 100%)', 'hsla(360, 100%, 100%, 1.0)'
  - Nombre: también puede usar cadenas con nombres de colores (solo admite nombres en minúsculas). La implementación de colores con nombre sigue la especificación CSS3/SVG.
- Tanto Android como iOS disponen de guías generales sobre el uso del color en cada plataforma que se deberían respetar.
- React Native tiene varias API de color diseñadas para permitir aprovechar al máximo el diseño de la plataforma y las preferencias del usuario.
  - PlatformColor: permite hacer referencia al sistema de color de la plataforma.
  - DynamicColorIOS (solo iOS) permite especificar qué colores deben usarse en modo claro u oscuro.

# Alto y ancho

- La altura y el ancho de un componente determinan su tamaño en la pantalla. La forma establecer las dimensiones fijas de un componente es agregando valores fijos a width y height en el estilo. Las dimensiones en React Native no tienen unidades y representan píxeles independientes de la densidad (no existe un mapeo universal de puntos a unidades físicas de medida).

```
<Image source={importImg} style={{ width: 40, height: 40 }} />
```

- Si se desea llenar una determinada parte de la pantalla, se pueden utilizar valores porcentuales pero las dimensiones porcentuales requieren de elementos primarios (contenedores) con un tamaño definido.

```
<View style={{ height: '25%', backgroundColor: 'powderblue' }} />
<View style={{ height: '75%', backgroundColor: 'skyblue' }} />
```



# Alto y ancho

- La forma general de establecer la altura y el ancho es mediante dimensiones flexible gestionadas por Flexbox.
- Se utiliza flex en el estilo de un componente para que el componente se expanda y se reduzca dinámicamente según el espacio disponible en su contenedor. Normalmente se usará flex: 1, que le dice a un componente que llene todo el espacio disponible, compartiéndolo uniformemente entre otros componentes con el mismo contenedor. El espacio se dividirá de acuerdo a los flex de cada elemento. Cuanto mayor sea el flex dado, mayor será la proporción de espacio que ocupará un componente en comparación con sus hermanos.
- Un componente solo puede expandirse para llenar el espacio disponible si su contenedor tiene dimensiones mayores a 0. Si uno de los contenedor no tiene flex o width y height fijos, el contenedor tendrá dimensiones de 0 y los hijos flex no serán visibles.
- Con minWidth/maxWidth y minHeight/maxHeight se pueden establecer las restricciones de tamaño máximo y mínimo (en píxeles o porcentajes), tienen mayor prioridad que todas las demás propiedades y siempre serán respetadas.

# Posicionamiento

- position define cómo se posiciona dentro de su contenedor.
  - relative (predeterminado): Cuando se posiciona relativamente, el elemento se coloca de acuerdo con el flujo normal del diseño y luego se desplaza en relación con esa posición en función de los valores de top, right, bottom y left. El desplazamiento no afecta la posición de ningún elemento hermano o padre.
  - absolute: Cuando se coloca de forma absoluta, un elemento no forma parte del flujo de diseño normal. Se presenta independiente de sus hermanos. La posición se determina en función de los valores top, right, bottom y left.
- zIndex controla qué componentes se muestran encima de otros, los de mayor valor se mostrarán sobre los de menor valor. Los componentes se procesan según su orden en el árbol del documento, por lo que los componentes posteriores se dibujan sobre los anteriores. zIndex puede ser útil en animaciones o posicionamiento absoluto donde no se desea este comportamiento.



# Márgenes, rellenos y bordes

- Los márgenes son el espacio que hay entre el borde de un elemento y su exterior: margin, marginLeft, marginRight, marginHorizontal, marginTop, marginBottom, marginVertical, marginStart, marginEnd
- El relleno es el espacio entre el borde de un elemento y su contenido. Para los elementos de tamaño flexible, el relleno aumentará el tamaño del elemento y compensará la ubicación de los elementos secundarios: padding, paddingLeft, paddingRight, paddingHorizontal, paddingTop, paddingBottom, paddingVertical, paddingStart, paddingEnd
- El borde es la línea que delimita al elemento: borderWidth, borderLeftWidth, borderRightWidth, borderTopWidth, borderBottomWidth, borderStartWidth, borderEndWidth, borderRadius

# direction

- direction especifica el flujo direccional del interfaz de usuario, para dar soporte a las configuraciones idiomáticas que se leen de derecha a izquierda y no para realizar el alineamiento de los elementos secundarios. Afecta a las propiedades start y end del margen y el relleno.
  - inherit (predeterminado): valor predeterminado excepto para el nodo raíz, que tendrá un valor basado en la configuración regional actual.
  - LTR: El texto y los elementos secundarios se disponen de izquierda a derecha. El margen y el relleno start del elemento se aplican en el lado izquierdo y end en el lado derecho.
  - RTL: El texto y los elementos secundarios se disponen de derecha a izquierda. El margen y el relleno start del elemento se aplican en el lado derecho y end en el lado izquierdo.



# Flexbox

- Un componente especifica el diseño y tamaño de sus elementos secundarios utilizando el algoritmo Flexbox. Flexbox está diseñado para proporcionar un diseño uniforme y adaptativo en diferentes tamaños de pantalla. La altura y el ancho de un componente secundarios se calcula dinámicamente si no esta definida explícitamente en su estilo.
- El comportamiento de Flexbox se define a través de propiedades de estilo y que en CSS en la web, con algunas excepciones:
  - El parámetro flex solo admite un solo número.
  - Los valores predeterminados son diferentes en: flexDirection es column en lugar de row, alignContent es flex-start en lugar de stretch, flexShrink es 0 en lugar de 1.

# flexDirection

- flexDirection controla la dirección en la que se disponen los elementos secundarios de un contenedor. Esto también se conoce como el eje principal. El eje transversal es el eje perpendicular al eje principal, o el eje en el que se disponen las líneas de envoltura.
  - column (predeterminado): Alinea los secundarios verticalmente, de arriba a abajo. Si el ajuste está habilitado, la siguiente línea comenzará a la derecha del primer elemento en la parte superior del contenedor.
  - column-reverse: Alinea los secundarios de abajo hacia arriba. Si el ajuste está habilitado, la siguiente línea comenzará a la derecha del primer elemento en la parte inferior del contenedor.
  - row: Alinea los secundarios horizontalmente, de izquierda a derecha. Si el ajuste está habilitado, la siguiente línea comenzará debajo del primer elemento a la izquierda del contenedor.
  - row-reverse: Alinea los secundarios de derecha a izquierda. Si el ajuste está habilitado, la siguiente línea comenzará debajo del primer elemento a la derecha del contenedor.



# justifyContent

- justifyContent define cómo alinear y espaciar a los elementos secundarios dentro del eje principal de su contenedor.
  - flex-start (predeterminado): Alinea los elementos secundarios al inicio del eje principal del contenedor sin espacio entre elementos.
  - flex-end: Alinea los elementos secundarios al final del eje principal del contenedor sin espacio entre elementos.
  - center: Alinea los elementos secundarios en el centro del eje principal del contenedor sin espacio entre elementos.
  - space-between: Separa uniformemente los elementos secundarios a lo largo del eje principal del contenedor, distribuyendo el espacio restante entre los secundarios pero sin espacio entre el primer y último elemento a los bordes.
  - space-around: Separa uniformemente los elementos secundarios a lo largo del eje principal del contenedor, distribuyendo el espacio restante entre los secundarios, incluyendo los bordes, pero siendo la mitad del espacio ente el primer y último elemento a sus bordes.
  - space-evenly: Separa uniformemente los elementos secundarios a lo largo del eje principal del contenedor, distribuyendo el espacio restante entre los secundarios, incluyendo los bordes, siendo todos exactamente iguales.

# alignItems y alignSelf

- alignItems define cómo alinear a los elementos secundarios a lo largo del eje transversal de su contenedor. Es la alineación vertical de las filas o la horizontal de las columnas.
  - stretch (predeterminado): Estirar los elementos secundarios de un contenedor para que coincidan con los bordes del eje transversal del contenedor.
  - flex-start: Alinea los elementos secundarios al inicio del eje transversal del contenedor.
  - flex-end: Alinea los elementos secundarios al final del eje transversal del contenedor.
  - center: Alinea los elementos secundarios en el centro del eje transversal del contenedor.
  - baseline: Alinea los elementos secundarios a lo largo de una línea de base común. Los elementos secundarios se pueden configurar para que sean la línea base de referencia para sus padres.
- alignSelf tiene las mismas opciones y efectos que alignItems pero solo afecta al elemento secundario para el que está definida, anulando la opción establecida por su contenedor con alignItems.



# flexWrap y alignContent

- flexWrap define que sucede cuando los elementos secundarios superan el tamaño del contenedor a lo largo del eje principal.
  - nowrap (predeterminado): indica que los elementos secundarios se condensarán en una única fila o columna.
  - wrap: indica que se añadan nuevas filas o columnas hacia el final para contener los elementos secundarios que desbordan al eje principal.
  - wrap-reverse: indica que se añadan nuevas filas o columnas hacia el principio para contener los elementos secundarios que desbordan al eje principal.
- alignContent define la distribución de los elementos secundarios en filas o columnas cuando se activa flexWrap:
  - flex-start, flex-end, space-between, space-around, center, stretch

# Elementos secundarios

- flex definirá cómo los elementos secundarios van a "llenar" el espacio disponible a lo largo de su eje principal. El espacio se dividirá de acuerdo con la propiedad flex de cada elemento.
- flexBasis es una forma independiente de proporcionar el tamaño inicial, antes de que flexGrow y flexShrink realicen los cálculos, de un elemento secundario en el eje transversal. Es similar a establecer el width si su contenedor tiene flexDirection:'row' o el height si es un flexDirection:'columna'.
- flexGrow especifica el factor de crecimiento de un elemento secundario en su eje principal. El factor de crecimiento especifica qué cantidad del espacio restante, dentro del contenedor flexible, debería ocupar el elemento en cuestión. flexShrink especifica el factor de reducción de un elemento secundario en su eje principal.
- El espacio restante es el tamaño del contenedor menos el tamaño de todos los elementos secundarios juntos. Si todos los elementos dentro del contenedor tienen el mismo factor de crecimiento o reducción, todos los elementos reciben la misma cantidad del espacio restante. De lo contrario, el espacio restante se distribuye en función de los diferentes factores estableciendo las proporciones.
- flexGrow y flexShrink se combinan para permitir que los elementos secundarios crezcan y se encojan según sea necesario. Aceptan cualquier valor en punto flotante >= 0, siendo 0 el valor predeterminado.



# Animaciones

- Las animaciones son muy importantes para crear una gran experiencia de usuario e interfaces reactivos. Los objetos estacionarios deben vencer la inercia a medida que comienzan a moverse. Los objetos en movimiento tienen impulso y rara vez se detienen de inmediato. Las animaciones permiten transmitir un movimiento físicamente creíble en el interfaz.
- React Native proporciona dos sistemas de animación complementarios: Animated para el control granular e interactivo de valores específicos y LayoutAnimation para transacciones de diseño global animadas.

# APIs



## Alertas

- El API Alert permite con alert() mostrar un cuadro de diálogo de alerta con el título y el mensaje especificados.
- Opcionalmente, se puede proporcionar una lista de botones, por defecto solo sale el de 'Aceptar'. Al tocar cualquier botón, se activará onPress y se descartará la alerta. En iOS, se puede especificar cualquier número de botones y, opcionalmente, su estilo (AlertButtonStyle). En Android se pueden especificar como máximo tres botones y, opcionalmente, se pueden descartar tocando fuera del cuadro de alerta.

```
Alert.alert('My Title', 'My message', [
      { text: 'Aceptar', onPress: () => Alert.alert('Accepted') },
      { text: 'Cancelar', onPress: () => Alert.alert('Canceled') },
      { text: 'Ignorar', onPress: () => Alert.alert('Ignored') },
    ],
    { cancelable: true, onDismiss: () => Alert.alert('Dismissed') },
)
```

- La alerta prompt(), que solicita al usuario que introduzca cierta información, está disponible solo en iOS.

# Vibración

- El API Vibration hace vibrar el dispositivo una duración fija, es una notificación.
- Las aplicaciones de Android deben solicitar el permiso android.permission.VIBRATE. En iOS, la vibración se implementa como una llamada a AudioServicesPlaySystemSound(kSystemSoundID_Vibrate).
- El método vibrate() puede tomar una matriz de números como argumento pattern con que representan el patrón de vibración. Se puede configurar repeat a true para ejecutar el patrón de vibración en un bucle hasta que se llame a cancel().
- En Android, la duración de la vibración tiene un valor predeterminado de 400 milisegundos y se puede especificar una duración de vibración arbitraria pasando un valor numérico único en milisegundos para el argumento pattern. En la matriz pattern, los valores impares representan la duración de la vibración en milisegundos y los pares representan el tiempo de separación.
- En iOS, la duración de la vibración es fija: 400 milisegundos (aproximadamente). Los valores de la matriz representan el tiempo de separación (la vibración es fija).

```
const SECONDS = 1000;
Vibration.vibrate(10 * SECONDS)
Vibration.vibrate([1 * SECONDS, 5 * SECONDS, 2 * SECONDS], true) // wait 1s, vibrate 5s, wait 2s
Vibration.cancel()
```



# Apariencia

- El API Appearance, con el método getColorScheme(), y el hook useColorScheme exponen las preferencias de apariencia del usuario: su esquema de color preferido (claro u oscuro).

```
const colorScheme = Appearance.getColorScheme();
const colorScheme = useColorScheme();
if (colorScheme === 'dark') {
```

- Los esquemas de color admitidos:
  - 'light': el usuario prefiere un tema de color claro.
  - 'dark': el usuario prefiere un tema de color oscuro.
  - nulo: el usuario no ha indicado un tema de color preferido.
- Cualquier lógica o estilo de renderizado que dependa del esquema de color preferido por el usuario debe intentar llamar a esta función en cada renderizado, en lugar de almacenar el valor en caché.
- El API Appearance dispone de addChangeListener() y removeChangeListener() para agregar y eliminar un controlador de eventos que se active cuando cambien las preferencias de apariencia.

# Estado de la aplicación

- El API AppState permite consultar si la aplicación está en primer o segundo plano e interceptar los cambios en el estado.
- Para ver el estado actual, se puede consultar AppState.currentState, que se mantendrá actualizado:
  - 'active': La aplicación se ejecuta en primer plano
  - 'background': La aplicación se ejecuta en segundo plano. El usuario esta en otra aplicación, en la pantalla de inicio o, solo en Android, en otra Activity, incluidas de la propia aplicación.
  - 'inactive' (solo iOS): este es un estado que ocurre cuando se cambia entre el primer plano y el fondo o durante períodos de inactividad, como al entrar en la vista multitarea, abrir el Centro de notificaciones o en el caso de una llamada entrante.
- Los eventos disponibles son: change y memoryWarning (ante la necesidad de lanzar una advertencia de memoria o liberarla). Los eventos focus y blur solo están disponibles en Android.



# Dimensiones

- El useWindowDimensions actualiza el valor width y height automáticamente cuando cambia el tamaño de la pantalla.

```
const window = useWindowDimensions()
<Text>{`Window Dimensions: height - ${window.height}, width - ${window.width}`}</Text>
```

- El API Dimensions permite consultar el tamaño de la pantalla o de la ventana de la aplicación (height, width, scale, fontScale) e interceptar los cambios en el estado.

```
const window = Dimensions.get("window");
const screen = Dimensions.get("screen");
```

- El evento change se activa cuando cambia una propiedad dentro del objeto Dimensions y recibe un DimensionsValue con {window, screen}.

# Compartir

- El API Share permite abrir el cuadro de diálogo para compartir contenido de texto.
- El método Share.share recibe un objeto con: message (el mensaje para compartir), url (la URL para compartir, solo iOS) y title (título del mensaje, solo Android). Opcionalmente, recibe un objeto con las opciones: dialogTitle para Android y subject (el asunto para compartir por correo electrónico), excludedActivityTypes y tintColor para iOS.
- El método Share.share devuelve una Promesa que se resolverá con un objeto {action, activityType}. En Android, se resolverá inmediatamente con action igual a Share.sharedAction y activityType a null. En iOS, si se resuelve la Promesa, action será igual a Share.sharedAction y activityType al modo de compartir o action será igual a Share.dismissedAction si el usuario descartó el cuadro de diálogo. Algunas opciones para compartir no aparecerán ni funcionarán en el simulador de iOS.



# Compartir

```
const onShare = async () => {
  try {
    const result = await Share.share({ message: 'Texto a compartir', })
    if (result.action === Share.sharedAction) {
      if (result.activityType) {
        setStatusShared(`ActivityType: ${result.activityType}`)
      } else {
        setStatusShared('No ActivityType')
      }
    } else if (result.action === Share.dismissedAction) {
      setStatusShared('Dismissed')
    }
  } catch (error) {
    setStatusShared(`ERROR: ${error.message}`)
  }
}
```

# Temporizadores

- Los temporizadores son una parte importante de una aplicación y React Native implementa los temporizadores del navegador y algunos adicionales.
  - setTimeout, clearTimeout
  - setInterval, clearInterval
  - setImmediate, clearImmediate
  - requestAnimationFrame, cancelAnimationFrame
- requestAnimationFrame(fn) no es lo mismo que setTimeout(fn, 0): el primero se disparará después de que todos los frames se hayan vaciado, mientras que el segundo se disparará lo más rápido posible (más de 1000x por segundo en un iPhone 5S).
- setImmediate se ejecuta al final del bloque de ejecución de JavaScript actual, justo antes de enviar la respuesta por lotes de vuelta a nativo y, si se llama a setImmediate dentro del callback de otro setImmediate, se ejecutará de inmediato, sin ceder en medio a nativo.
- Promise utiliza setImmediate para su implementación de asincronía.



# InteractionManager

- Una de las razones por las que las aplicaciones nativas bien construidas parecen tan fluidas es porque evitan operaciones costosas durante las interacciones y las animaciones. En React Native, actualmente tiene la limitación de que solo hay un único subproceso de ejecución de JavaScript, pero se puede usar el API InteractionManager para asegurar de que un trabajo de ejecución prolongada esté programado para comenzar después de que se hayan completado las interacciones/animaciones, esto permite que las animaciones de JavaScript se ejecuten sin problemas.
- Las aplicaciones pueden programar tareas para que se ejecuten después de interacciones:

```
InteractionManager.runAfterInteractions(() => {
  // ...long-running synchronous task...
});
```

- El sistema de manejo de toques considera que uno o más toques activos son una 'interacción' y retrasará los callbacks de runAfterInteractions() hasta que todos los toques hayan finalizado o cancelado.

# NAVEGACIÓN



## Navegación entre pantallas

- Las aplicaciones móviles rara vez se componen de una sola pantalla. La gestión de la presentación y la transición entre múltiples pantallas normalmente se maneja mediante lo que se conoce como navegador.
- React Native originalmente incluía una API de navegación incorporada, pero finalmente se eliminó y no dispone de ningún mecanismo propio pero admite múltiples implementaciones:
  - React Router es una librería para gestionar rutas en aplicaciones que utilizan ReactJS, indicada para los que ya la conocen.
  - React Navigation proporciona una solución de navegación sencilla, con la capacidad de presentar patrones comunes de navegación de pila y navegación con pestañas tanto en Android como en iOS.
  - React Native Navigation proporciona una alternativa a React Navigation cuando se está integrando React Native en una aplicación que ya administra la navegación de forma nativa o se busca una biblioteca que proporcione navegación nativa en ambas plataformas.

# Desafíos de navegación

- La navegación funciona de manera diferente en cada plataforma nativa. iOS usa controladores de vista, mientras que Android usa actividades . Estas APIs específicas de cada plataforma funcionan de manera diferente técnicamente y aparecen de manera diferente para el usuario. Las bibliotecas de navegación de React Native intentan admitir la apariencia de cada plataforma, al mismo tiempo que brindan una única API de JavaScript consistente.
- Las APIs de navegación nativas no se corresponden con las "vistas". Los componentes de React Native como View, Texty Image, se asignan aproximadamente a una "vista" nativa subyacente, pero en realidad no hay un equivalente para algunas de las API de navegación. No siempre hay una forma clara de exponer estas API a JavaScript.
- La navegación en el móvil es con estado. En la web, la navegación suele ser sin estado, donde una URL (ruta) lleva al usuario a una sola pantalla/página. En dispositivos móviles, el historial del estado de navegación del usuario se mantiene en la aplicación para que el usuario pueda volver a las pantallas anteriores; una pila de pantallas puede incluso incluir la misma pantalla varias veces.



# React Router

- React Router es una librería para gestionar rutas en aplicaciones que utilicen ReactJS, es una de las mas conocidas. Para instalar la versión adaptada para React Native:
  - npm install react-router-native -save
- Para habilitar la compilación en Expo para Native Web hay que instalar:
  - npm install @expo/webpack-config --save-dev
- El fichero webpack.config.js en la raiz del proyecto permite configurar Webpack:

```
const path = require('path')
const createConfigAsync = require('@expo/webpack-config')
module.exports = async function (env, argv) {
  const config = await createConfigAsync(env, argv)
  config.module.rules.push({ test: /\.js$/, loader: 'babel-loader',
    include: [ path.join(__dirname, 'node_modules/react-router-native') ]
  })
  return config
}
```

# React Navigation

- react-navigation es la biblioteca de navegación semioficial y más popular. Maneja muy bien la mayoría de los desafíos descritos anteriormente y es lo suficientemente configurable para la mayoría de las aplicaciones.
- React Navigation se compone de algunas utilidades principales y los navegadores que se utilizan para crear la estructura de navegación en la aplicación.
  - npm install @react-navigation/native @react-navigation/native-stack
- Para instalar las bibliotecas con las dependencias utilizadas por la mayoría de los navegadores:
  - Si es un proyecto administrado por Expo:
    - expo install react-native-screens react-native-safe-area-context
  - Si es un proyecto React Native simple:
    - Configurar un <NavigationContainer> (@react-navigation/native)
    - Crear un navegador:
      - createStackNavigator (@react-navigation/stack),
      - createBottomTabNavigator (@react-navigation/bottom-tabs),
      - createDrawerNavigator (@react-navigation/drawer)
    - Definir las pantallas en nuestra app



# NavigationContainer

- NavigationContainer es el componente que gestiona el árbol de navegación y contiene el estado de navegación. Este componente debe envolver toda la estructura de los navegadores, por lo general, en la raíz de la aplicación.

```
function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

# Navigator

- El Navigator es un componente de React que decide cómo representar las pantallas que se han definido. Contiene elementos Screen para definir la configuración de las pantallas.
- Los navegadores disponibles son:
  - Stack: proporciona una forma para que su aplicación haga la transición entre pantallas donde cada nueva pantalla se coloca encima de una pila. Está configurado para tener el aspecto familiar de iOS y Android.
  - Native Stack: implementación de Stack que utiliza las API nativas UINavigationController en iOS y Fragment en Android.
  - Drawer: Componente que representa un cajón de navegación que se puede abrir y cerrar mediante gestos.
  - Bottom Tabs: Una barra simple de pestañas en la parte inferior de la pantalla que le permite cambiar entre diferentes rutas. Las rutas se inicializan de forma perezosa: los componentes de pantalla no se montan hasta que se enfocan por primera vez.
  - Material Bottom Tabs: Adaptación a material-design de Bottom Tabs.
  - Material Top Tabs: Adaptación a material-design de Bottom Tabs en la parte superior de la pantalla.



# Native Stack Navigator

- El navegador de pila nativo (stack) de React Navigation proporciona una forma para que la aplicación haga la transición entre pantallas y administre el historial de navegación. La aplicación introduce y extrae elementos de la pila de navegación a medida que los usuarios interactúan con ella, y esto hace que el usuario vea diferentes pantallas.
  - npm install @react-navigation/native-stack
- createNativeStackNavigator es una función que devuelve un objeto que contiene 2 propiedades: Navigator y Screen. Ambos son componentes React que se utilizan para configurar el navegador.

```
import { createNativeStackNavigator } from '@react-navigation/native-stack';
const Stack = createNativeStackNavigator();
```

- El Navigator debe contener elementos Screen para definir la configuración de las rutas.

```
<Stack.Navigator>
    <Stack.Screen name="Home" component={HomeScreen} />
    <Stack.Screen name="Settings" component={Settings} />
</Stack.Navigator>
```

# Native Stack Navigator

- Stack.Navigator es un componente que toma la configuración de las rutas como sus elementos secundarios, con propiedades adicionales para la configuración y representa el contenido.

```
<Stack.Navigator initialRouteName="Home" screenOptions={{ headerMode:
'screen', headerTintColor: 'white', headerStyle: { backgroundColor: 'tomato' }, }} >
```

- Cada componente Stack.Screen tiene obligatoriamente las propiedades name que establece el nombre de la ruta y component que especifica el componente a representar para la ruta. Acepta options, un objeto o una función que devuelve un objeto, que contiene varias opciones de configuración.

```
<Stack.Screen name="Home" component={HomeScreen} options={{ title: 'Main',
        headerTintColor: '#fff', }} />
<Stack.Screen name="Home" component={HomeScreen} options={({ route }) =>
        ({ title: `Hola ${route.params.name}` })} />
```

- A menudo es necesario actualizar la configuración options de la pantalla activa desde el propio componente de pantalla montado.

```
navigation.setOptions({ title: 'Updated!' })
```



# Componente pantalla

- Un componente pantalla es un componente que usamos en nuestra configuración de ruta y representan cada una de las pantalla a las que se navega. Por convención, utilizan el sufijo Screen en el nombre del componente.

```
function MessagesScreen({ navigation, route, state }) {
 return ( … )
}
```

- Cuando React Navigation representa la componente pantalla le inyecta las propiedades:
  - navigation: contiene varias funciones de conveniencia (basadas en el tipo de navegador actual) para interactuar con el navegador.
  - route: contiene información diversa sobre la ruta actual.
- Los hooks useNavigation y useRoute dan acceso navigation y route cuando no se pueden pasar directamente al componente o no se quiere pasarlos en el caso de elementos secundarios profundamente anidado.

```
const navigation = useNavigation();
const route = useRoute();
```

# navigation

- Comunes:
  - navigate: ir a otra pantalla, averigua la acción que debe realizar para hacerlo
  - reset: borrar el estado del navegador y reemplazarlo con una nueva ruta
  - goBack: cerrar la pantalla activa y retroceder en la pila
  - isFocused: comprobar si la pantalla tiene el foco
  - setParams: realizar cambios en los parámetros de la ruta
  - setOptions: actualizar las opciones de la pantalla
  - dispatch: enviar un objeto de acción para actualizar el estado de navegación
  - addListener: suscribirse a actualizaciones de eventos de los navegadores
- Pila:
  - replace: reemplazar la ruta actual por una nueva
  - push, pop: apilar y desapilar la ruta en la pila
  - popToTop: ir a la parte superior de la pila
- Pestañas:
  - jumpTo: ir a una pantalla específica en el navegador de pestañas
- Cajón:
  - jumpTo: ir a una pantalla específica en el navegador de cajones
  - openDrawer, closeDrawer, toggleDrawer: abrir, cerrar o alternar el estado del cajón



# route

- key: Clave única de la pantalla. Creado automáticamente o agregado mientras se navega a esta pantalla.
- name: Nombre de la pantalla. Definido en la jerarquía de componentes del navegador.
- path: Existe una cadena opcional que contiene la ruta que abrió la pantalla cuando la pantalla se abrió a través de un enlace profundo.
- params: Un objeto opcional que contiene parámetros que se define durante la navegación.
- state: Un objeto opcional que contiene el estado de navegación de un navegador secundario anidado dentro de esta pantalla.

# Navegación

- Se puede llamar a la función navigate(), del navigation recibido, con el nombre de la ruta a la que mover al usuario, la ruta inicia la pila.

```
<Button title="Go Home" onPress={() =>
        navigation.navigate('Home')}
```

- Se puede llamar a push() para mover y agregar la ruta al historial de navegación existente. Esto permite volver a la ruta anterior con goBack() o al principio con popToTop().

```
<Button title="Go Details" onPress={() =>
        navigation.push('Details')} />
<Button title="Go back" onPress={() => navigation.goBack()} />
<Button title="Go first" onPress={() => navigation.popToTop()} />
```

- Con jumpTo() se restringe la navegación a una ruta existente en las pestañas o el cajón.



# Navegación

- El componente Link representa un componente que puede navegar a una pantalla al ser presionado.

```
<Link to={{ screen: 'Details', params: { id: 66 } }}> Go Details</Link>
```

- Si se desea un comportamiento diferente para la navegación en la página, como replace en lugar de navigate, la propiedad action permite personalizarlo:

```
<Link to={{ screen: 'Details', params: { id: 66 } }}
action={StackActions.replace('Details', { id: 66 })}> Go Details</Link>
```

- El hook useLinkTo permite navegar a una pantalla usando una ruta en lugar de un nombre de pantalla según las opciones linking. Devuelve una función que recibe la ruta a la que navegar.

```
const linkTo = useLinkTo();
onPress={() => linkTo('/profile/66')}>
```

- El hook useLinkProps permite construir componentes de enlace personalizados que permiten navegar a una pantalla usando una ruta en lugar de un nombre de pantalla en función de las opciones linking.

# Pasar parámetros

- Los parámetros se pasan a una ruta encapsulados en un objeto como segundo parámetro de navigate(), push() o jumpTo().

```
navigation.navigate('Details', { count: 86 })
```

- Los parámetros se recuperan a través del objeto route inyectado.

```
const { count } = route.params
```

- Las pantallas también pueden actualizar sus parámetros, al igual que pueden actualizar su estado.

```
navigation.setParams({ count: route.params.count + 1});
```

- También se pueden definir algunos parámetros iniciales para la pantalla, a utilizar si no se especifican al navegar:

```
<Stack.Screen name="Details" component={DetailsScreen}
initialParams={{ count: 0 }} />
```

- Como buena practica, los parámetros deben contener los datos mínimos necesarios para mostrar una pantalla, nada más.



# Ciclo de vida de navegación

- React Navigation emite eventos a los componentes de pantalla que se suscriben a ellos. Se pueden escuchar los eventos focus y blur para saber cuándo una pantalla gana o pierde el foco. Hay algunos eventos principales como focus, blur, state y beforeRemove que funcionan para todos los navegadores, así como eventos específicos que funcionan solo para ciertos navegadores.
- Dentro de una pantalla, se pueden agregar oyentes con el método addListener del navigation, pasándole dos argumentos: el tipo de evento y un controlador de eventos, y devuelve una función a llamar para darse de baja del evento.

```
React.useEffect(() => {
  const unsubscribe = navigation.addListener('focus', () => { … });
  return unsubscribe;
 }, [navigation]);
```

- Se pueden usar los hooks useFocusEffect y useIsFocused para hacerlo más fácil.

# Estado de navegación

- El estado de navegación es donde React Navigation almacena la estructura de navegación y el historial de la aplicación. Es útil cuando se necesita realizar operaciones avanzadas. Se puede obtener el estado actual con navigation.getState() o el hook useNavigationState() cuando cambien los valores.
- Las propiedades presentes en el estado son:
  - type: Tipo de navegador al que pertenece: stack, , tab, drawer.
  - key: Clave única para identificar al navegador.
  - routeNames: Lista de nombres de las pantallas definidas en el navegador.
  - routes: Lista de objetos de ruta (pantallas) que se representan en el navegador. También representa el historial en un navegador de pila. Debe haber al menos un elemento presente en esta matriz.
  - index: Índice del objeto de ruta enfocado en la routes matriz.
  - history: Una lista de elementos visitados. Solo está presente en los navegadores de pestañas y cajones.



# Navegadores anidados

- A veces resulta útil anidar varios navegadores de diferentes tipos (pila, cajón o pestañas), aunque hay reducir al mínimo los navegadores anidados e intentar lograr el comportamiento deseado con la menor anidación posible.
- Al anidar navegadores, hay que tener en cuenta:
  - Cada navegador guarda su propio historial de navegación
  - Cada navegador tiene sus propias opciones
  - Cada pantalla en un navegador tiene sus propios parámetros
  - Las acciones de navegación son manejadas por el navegador actual
  - Los métodos específicos del navegador están disponibles en los navegadores anidados dentro
  - Los navegadores anidados no reciben los eventos de los navegadores que les contienen
  - La interfaz de usuario del navegador principal se representa sobre el del navegador secundario

# Personalizar encabezados

- A veces, se necesita más control que solo cambiar el texto y los estilos del título: headerTitle acepta un componente en sustitución del título, headerRight permite controlar el lado derecho del encabezado, headerBackTitle, headerBackTitleStyle y headerBackImageSource permiten cambiar el comportamiento de la etiqueta y la imagen del botón Atrás:

```
function LogoTitle(props) {
  return (
    <View style={{ flexDirection: 'row', alignItems: 'center', flex: 1, }}>
      <Image style={{ width: 50, height: 50 }} source={require('../assets/favicon.png')} />
      <Text style={{ flex: 1, marginStart: 5 }}>{props.children}</Text>
    </View>
  )
<Stack.Screen name='Home' component={HomeScreen}
        options={{
          title: 'Main',
          headerTitle: props => <LogoTitle {...props} />,
          headerRight: () => (
            <Button title='Info' color='blue' onPress={() => alert('This is a button!')} />
          ),
        }} />
```



# Drawer Navigator

- El navegador de cajón (drawer) de React Navigation proporciona un componente que representa un cajón de navegación que se puede abrir y cerrar mediante gestos.
  - npm install @react-navigation/drawer react-native-gesture-handler react-native-reanimated
  - expo install react-native-gesture-handler react-native-reanimated
- La primera importación (en la parte superior y sin nada más antes) de App.js o index.js debe ser:

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

- Agregar el complemento de Babel en babel.config.js:

```
return {
    presets: ['babel-preset-expo'],
    plugins: ['react-native-reanimated/plugin'],
  };
```

- Reiniciar el servidor de desarrollo y borrar la caché de paquetes:
  - expo start --clear

# Drawer Navigator

- Para crear los objetos Navigator y Screen:

```
import { createDrawerNavigator } from '@react-navigation/drawer';
const Drawer = createDrawerNavigator();
function DrawerMainScreen() {
  return (
    <Drawer.Navigator>
      <Drawer.Screen name="Feed" component={Feed} />
      <Drawer.Screen name="Article" component={Article} />
    </Drawer.Navigator>
  );
}
```



# Bottom Tabs Navigator

- El navegador de pestañas de React Navigation proporciona una barra simple de pestañas en la parte inferior de la pantalla que le permite cambiar entre diferentes rutas.
  - npm install @react-navigation/bottom-tabs
- Para crear los objetos Navigator y Screen:

```
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
const Tab = createBottomTabNavigator();

function TabMainScreen() {
  return (
    <Tab.Navigator>
      <Tab.Screen name="Home" component={HomeScreen} />
      <Tab.Screen name='Messages' component={MessagesScreen} options={{
        tabBarBadge: 3, tabBarLabel: 'Send', tabBarIcon: ({ focused, color, size }) => (
          <Ionicons name={focused ? 'ios-send' : 'ios-send-outline'} color={color} size={size} />
        ), }} />
    </Tab.Navigator>
  );
}
```

# CONECTIVIDAD



## Acceso remoto

- Muchas aplicaciones móviles necesitan cargar recursos desde una URL remota. Es posible que desee realizar una solicitud POST a una API REST, o que necesite obtener una parte del contenido estático de otro servidor.
- React Native proporciona una implementación nativa del API Fetch para las necesidades de red.
- La API XMLHttpRequest está integrada en React Native. Esto significa que se puede usar bibliotecas de terceros como frisbee o axios que dependen de ella, o se puede usar la API XMLHttpRequest directamente.
- React Native también es compatible con WebSockets, un protocolo que proporciona canales de comunicación full-duplex a través de una única conexión TCP.

# Restricciones

- El modelo de seguridad para XMLHttpRequest es diferente al de la web, ya que no existe el concepto de CORS en las aplicaciones nativas.
- De forma predeterminada, iOS bloqueará cualquier solicitud que no esté cifrada mediante SSL (https). Si se necesita buscar desde una URL de texto simple (http), primero se deberá agregar una excepción de seguridad de transporte de aplicaciones. Si se sabe de antemano a qué dominios se accederán, es más seguro agregar excepciones solo para esos dominios; si los dominios no se conocen hasta el tiempo de ejecución, se puede deshabilitar ATS por completo. A partir de enero de 2017, la revisión de la App Store de Apple requiere una justificación razonable para deshabilitar ATS .
- En Android, a partir del nivel de API 28, el tráfico de texto sin cifrar también está bloqueado de forma predeterminada. Se puede anular mediante la configuración android:usesCleartextTraffic en el manifiesto de la aplicación.



# Fetch

```
const [isLoading, setIsLoading] = React.useState(true)
const [item, setItem] = React.useState({})

const getItem = async () => {
  try {
    const response = await fetch(
      `http://192.168.1.156:4321/api/contactos/${id}`, { headers: { Accept: 'application/json' }}
    )
    const data = await response.json()
    setItem(data)
  } catch (error) {
    console.error(error)
  } finally {
    setIsLoading(false)
  }
}
React.useEffect(() => getItem(), [])

return isLoading ? (<ActivityIndicator style={{ flex: 1 }} />) : (<Text>{JSON.stringify(item)}</Text>)
```

# WebSocket

```
var ws = new WebSocket('ws://host.com/path');

ws.onopen = () => {
  // connection opened
  ws.send('something'); // send a message
};
ws.onmessage = (e) => {
  // a message was received
  console.log(e.data);
};
ws.onerror = (e) => {
  // an error occurred
  console.log(e.message);
};
ws.onclose = (e) => {
  // connection closed
  console.log(e.code, e.reason);
};
```



# PLATAFORMAS CRUZADAS

# Código específico de la plataforma

- Al crear una aplicación multiplataforma, se desea reutilizar la mayor cantidad de código posible. Pueden surgir escenarios en los que tenga sentido que el código sea diferente, por ejemplo, es posible que desee implementar componentes visuales separados para Android e iOS.
- Ciertos componentes pueden tener propiedades que funcionan solo en una plataforma. React Native proporciona el API Platform que detecta la plataforma en la que se ejecuta la aplicación. Se puede usar la lógica de detección para implementar código específico de la plataforma y es una buena opción cuando solo pequeñas partes de un componente sean específicas de la plataforma.
- React Native dispone de componentes y APIs exclusivos de una plataforma que no están disponibles para las demás. Cuando el código específico para cada plataforma es más complejo o se desea utilizar recursos exclusivos, se debe considerar dividir el código en archivos separados. React Native dispone de sub extensiones de archivo específicas de la plataforma, detectará cuando un archivo tiene una sub extensión .ios. o .android. y cargará el archivo de la plataforma relevante cuando lo requieran otros componentes.



# Código específico de la plataforma

- Si se dispone de versiones especificas de plataforma:
  - Menu.js
  - Menu.ios.js
  - Menu.android.js
- Cuando se solicita el componente, React Native cargará automáticamente el archivo correcto según la plataforma en ejecución.

```
import Menu from './Menu';
```

- También se puede usar la extensión .native.js cuando se necesita compartir un módulo entre NodeJS/Web y React Native pero no presenta diferencias entre Android/iOS. Esto es especialmente útil para proyectos que tienen código común compartido entre React Native y ReactJS.

# Plataforma

- El API Platform detecta la plataforma en la que se ejecuta la aplicación y expone las constantes comunes y específicas disponibles relacionadas con dicha plataforma.
- A primer nivel:
  - OS: Valor cadena que representa el sistema operativo actual ('android', 'ios').
  - Version: Versión del sistema operativo (número en Android y cadena en iOS).
  - isTesting: Si la aplicación se ejecuta en modo pruebas
  - isTV: Si el dispositivo es un televisor.
  - isPadiOS: Si el dispositivo es un iPad (solo en iOS).
  - constants: Objeto que contiene todas las constantes comunes y específicas disponibles relacionadas con la plataforma
- Las constantes comunes de constants son:
  - reactNativeVersion: Información sobre la versión de React Native. Las propiedades son major, minor, patch y, opcionalmente, prerelease.
  - isTesting: Modo pruebas



# Plataforma

- Las constantes específicas de Android en constants son:
  - Version: Número de versión del sistema operativo
  - Release: Cadena con la release del sistema operativo
  - Serial: Número de serie de hardware del dispositivo
  - Fingerprint: Identificador único de la compilación
  - Model: Nombre visible para el usuario final del dispositivo
  - Brand: Marca visible para el consumidor con la que se asociado al producto/hardware.
  - Manufacturer: Fabricante del dispositivo
  - ServerHost:
  - uiMode: Tipo de IU: 'car', 'desk', 'normal','tv', 'watch' and 'unknown'.
- Las constantes específicas de iOS en constants son:
  - systemName: Nombre del sistema operativo
  - osVersion: Cadena con la versión del sistema operativo
  - interfaceIdiom: Tipo de IU: unspecified, phone, pad, tv, carPlay, mac
  - forceTouchAvailable: Indica la disponibilidad de 3D Touch en el dispositivo

# Plataforma

- El método Platform.select() recibe un objeto, donde las propiedades pueden ser 'ios', 'android', 'native' o 'default' , y devuelve el valor más adecuado para la plataforma en la que se está ejecutando. Si no existe la propiedad de la plataforma devuelve native o, si tampoco existe, default.

```
const styles = StyleSheet.create({
  container: {
    ...Platform.select({
      ios: {  backgroundColor: 'red' },
      android: { backgroundColor: 'green' },
      default: { backgroundColor: 'blue' }
    })
  }});
```

- Dado que las propiedades aceptan cualquier valor, también se puede usar para cargar componentes, imagenes o ejecutar códigos específicos de la plataforma:

```
const Component = Platform.select({
  native: () => require('ComponentForNative'),
  default: () => require('ComponentForWeb')
})();
<Component />;
```



# Color de la plataforma

- El API PlatformColor permite acceder a los colores nativos en la plataforma de destino proporcionando el nombre del color nativo (debe coincidir con la cadena tal como existe en la plataforma nativa donde se ejecuta la aplicación).
  - Android: ?android:attr ó @android:color
  - iOS: Standard Colors ó UI Element Colors
- A la función PlatformColor se pasa una cadena y devolverá el color nativo correspondiente siempre que exista en esa plataforma. Si se le pasa más de un valor, tratará el primer valor como predeterminado y el resto como alternativas.

```
...Platform.select({
    ios: {
      color: PlatformColor('label'),
      backgroundColor: PlatformColor('systemTealColor'),
    },
    android: {
      color: PlatformColor('?android:attr/textColor'),
      backgroundColor: PlatformColor('@android:color/holo_blue_bright'),
    },
    default: { color: 'black' }
  })
```

# DrawerLayoutAndroid (Android)

- En Android (Material Design), un DrawerLayout es un contenedor especial de la librería de soporte, que alberga dos tipos de contenido, el contenido principal y el contenido para el Navigation Drawer (el típico menú lateral deslizante o vista de navegación). El contenido principal es el contenedor que veremos en la actividad normalmente.
- El componente DrawerLayoutAndroid de React Native envuelve a DrawerLayout (solo Android). El contenido del componente se representa como la vista principal y el contenido de la vista de navegación se establece a través de la propiedad renderNavigationView.
- Inicialmente, la vista de navegación no está visible en la pantalla, mediante los métodos openDrawer() y closeDrawer() se puede mostrar y ocultar. Las propiedades drawerPosition y drawerWidth permiten establecer desde que lado de la ventana se va a desplegar y el ancho de la vista de navegación.



# DrawerLayoutAndroid

```
const drawer = useRef(null)
const [hidden, setHidden] = useState(true)
const changeNavigationShow = () => {
 hidden ? drawer.current.openDrawer() : drawer.current.closeDrawer()
 setHidden(!hidden)
}
const navigationView = () => (
 <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center', padding: 5 }}>
  <Text>Opción 1</Text><Text>Opción 2</Text>
 </View>
)
return (<>
   <View style={{ flexDirection: 'row', alignItems: 'center' }}>
    <Button title='M' onPress={changeNavigationShow} />
    <Text>My App</Text>
   </View>
   <DrawerLayoutAndroid ref={drawer} drawerWidth={100} drawerPosition='left'
            renderNavigationView={navigationView} >
     <View> {/* ... */} </View>
   </DrawerLayoutAndroid>
  </>)
}
```

# SafeAreaView (iOS)

- El propósito de SafeAreaView es representar contenido dentro de los límites del área segura de un dispositivo. Actualmente solo es aplicable a dispositivos iOS con iOS versión 11 o posterior.
- SafeAreaView representa el contenido anidado y aplica relleno automáticamente para reflejar la parte de la vista que no está cubierta por las barras de navegación, las barras de pestañas, las barras de herramientas y otras vistas antecesoras.
- Además, y lo que es más importante, los rellenos de SafeAreaView reflejan las limitaciones físicas de la pantalla, como las esquinas redondeadas o las muescas de la cámara (como el área de alojamiento del sensor en el iPhone 13).

```
<SafeAreaView style={{flex: 1}}>
    <Text style={styles.text}>Page content</Text>
</SafeAreaView>
```



# SafeAreaView (iOS)

- El propósito de SafeAreaView es representar contenido dentro de los límites del área segura de un dispositivo. Actualmente solo es aplicable a dispositivos iOS con iOS versión 11 o posterior.
- SafeAreaView representa el contenido anidado y aplica relleno automáticamente para reflejar la parte de la vista que no está cubierta por las barras de navegación, las barras de pestañas, las barras de herramientas y otras vistas antecesoras.
- Además, y lo que es más importante, los rellenos de SafeAreaView reflejan las limitaciones físicas de la pantalla, como las esquinas redondeadas o las muescas de la cámara (como el área de alojamiento del sensor en el iPhone 13).

```
<SafeAreaView style={{flex: 1}}>
    <Text style={styles.text}>Page content</Text>
</SafeAreaView>
```

© JMA 2020. All rights reserved

# InputAccessoryView (iOS)

- El componente InputAccessoryView permite la personalización de la vista de accesorios de entrada de teclado en iOS. La vista de accesorios de entrada se muestra sobre el teclado cada vez que un TextInput tiene el foco. Este componente se puede utilizar para crear barras de herramientas personalizadas.
- Se debe envolver la vista de la barra de herramientas personalizada con el componente InputAccessoryView y asignar a la propiedad nativeID un identificador unico. Dicho nativeID se usara en las propiedades inputAccessoryViewID de los TextInput en que se dese la barra personalizada.

```
const inputAccessoryViewID = 'resetBarID';
// …
<InputAccessoryView nativeID={inputAccessoryViewID}>
      <Button onPress={() => setText(initialText)} title="Reset" />
</InputAccessoryView>
<TextInput inputAccessoryViewID={inputAccessoryViewID}
       onChangeText={setText} value={text} />
```



# Módulos nativos

- A veces, una aplicación React Native necesita acceder a una API de plataforma nativa que no está disponible de forma predeterminada en JavaScript, por ejemplo, las API nativas para acceder a Apple o Google Pay. Tal vez se desee reutilizar algunas bibliotecas existentes de Objective-C, Swift, Java o C++ sin tener que volver a implementarlas en JavaScript, o escribir código de subprocesos múltiples de alto rendimiento para cosas como el procesamiento de imágenes.
- El sistema NativeModule expone instancias de clases Java/Objective-C/C++ (nativas) a JavaScript (JS) como objetos JS, lo que le permite ejecutar código nativo arbitrario desde JS.
- Hay dos formas de escribir un módulo nativo para su aplicación React Native:
  - Directamente dentro de los proyectos iOS/Android de su aplicación React Native
  - Como un paquete NPM que puede ser instalado como una dependencia por las aplicaciones React Native