

Animated image component in React Native

Let's see how we can create the animated image component with pan and zoom capability in React Native similar to what LinkedIn and Instagram are using in their apps.

13 min read · Dec 23, 2023



Varun Kukade

Follow



Listen



Share

To get an idea about what the final component would look like, here is the video:



We will be using three popular libraries in react native:

1. React native fast Image : This is an improved version of the core Image component of react native with improved image caching capability. You can read more about it here — <https://github.com/DylanVann/react-native-fast-image>
You can also use the core Image component of react native for this section.

2. React native reanimated : For those who don't know, this is an animation library in React native which helps us to create beautiful animations. React native also provides us with the core Animated API. There are some differences between them. For this section of zooming we are using reanimated because gesture handler version 2 is not compatible with core react native Animated API. Also, advanced animations beyond scaling, translate or opacity, etc reanimated becomes an obvious choice as it can do animations on UI thread completely. I will make a different blog regarding the reanimated in depth and how it helps us to perform animations completely on the UI thread in 60 fps without re-rendering/going to the javascript thread. You can read more about this here - <https://docs.swmansion.com/react-native-reanimated/>

3. React native gesture handler : This is the library which helps us to detect various gestures on the app. Gestures like Pan gesture (dragging things around the x and y axis) , pinch gesture (scaling things up and down) , long press gesture, tap gesture, etc. You can read more about this here — <https://docs.swmansion.com/react-native-gesture-handler/docs/category/fundamentals>

First of all let's install, and set these libraries -

Note : The following steps/commands may change in future as these libraries improve. Hence it's a better idea to directly follow the steps from their main GitHub repo/docs.

Open the terminal, and cd into the project folder and run the following commands.

1. For react-native-fast-image head over to the [GitHub](#) page and complete the necessary steps to set this up which include —

```
yarn add react-native-fast-image  
cd ios && pod install
```

2. For react-native-reanimated head over to the [installation](#) page and complete the necessary steps to set this up which include —

```
yarn add react-native-reanimated
OR
npm install react-native-reanimated
```

Add react-native-reanimated/plugin plugin to your babel.config.js .
react-native-reanimated/plugin has to be listed last in plugins list.

```
module.exports = {
  presets: [
    ... // don't add it here :)
  ],
  plugins: [
    ...
    'react-native-reanimated/plugin',
  ],
};
```

Clear Metro Bundler Cache -

```
yarn start - reset-cache
OR
npm start - - reset-cache
```

Install ios pods -

```
cd ios
pod install
cd ..
```

3. For react-native-gesture-handler head over to the installation page and complete the necessary steps to set this up which include —

```
yarn add react-native-gesture-handler
OR
npm install - save react-native-gesture-handler

cd ios
pod install
cd ..
```

After installation, wrap your entry point with `<GestureHandlerRootView>` or `gestureHandlerRootHOC`. It's important to keep the `GestureHandlerRootView` as close to the actual root view as possible.

For example:

```
import { GestureHandlerRootView } from 'react-native-gesture-handler';
```

```
export default function App() {
  return (
    <GestureHandlerRootView style={{ flex: 1 }}>
      {/* content */}
    </GestureHandlerRootView>
  );
}
```

Important: You only need to do next step if you're using React Native 0.59 or lower. Since v0.60, linking happens automatically.

```
react-native link react-native-gesture-handler
```

For React Native 0.61 or greater, add the library as the first import in your `app.tsx/app.jsx` file:

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

Now we are done with the setup and installation, clean the build for Android and iOS and rebuild the app completely.

For Android use the following commands in the project folder -

```
rm -rf node_modules
yarn cache clean
yarn install
cd android
./gradlew clean
cd ..
```

For ios use the following commands in the project folder -

```
cd ios
rm -rf ~/Library/Caches/CocoaPods
rm -rf Pods
rm -rf ~/Library/Developer/Xcode/DerivedData/*
pod deintegrate
pod setup
pod install
cd ..
```

You can also clean build from android studio and xcode.

For Android studio : Open Android studio, open Android folder of the project in Android studio, click on Build in the toolbar, and tap on “Clean Project”. After the project is cleaned, head over to “File” in the toolbar and find “Invalidate Caches” and perform the invalidate cache action.

For Xcode: Open the project in Finder. Head over to the ios folder. Find the .xcworkspace folder and double-tap on it. Xcode will be opened for your project. Now find the “Product” option in the toolbar and click on “Clean Build Folder”. After the build is cleaned, head over to the Xcode settings and location section. Click on

the right arrow for the derived data location path. Delete all derived data in the derived data folder.

Now let's start coding...

Step 1 : First we will create a reusable component for this feature. For any image where you want this functionality, you can reuse this component for that image. Let's name it AnimatedFastImage.

```
import FastImage, { FastImageProps } from 'react-native-fast-image';

export const AnimatedFastImage: FC<FastImageProps> = ({
  resizeMode = FastImage.resizeMode.contain,
  style,
  ...props
}): ReactElement => {

  return <></>
};
```

AnimatedFastImage will accept resize mode for the image. Values can be “contain”, “center”, “cover”, or “stretch”. The default value would be “contain”.

It will accept all the props which react-native-fast-image accepts. You can read more about props supported by it [here](#).

Step 2 : Now to have the zooming, and dragging functionality for the image, we would need to animate the image. Now I will explain to you why.

Concept — The gesture handler can detect the gesture performed by the user on the screen. But to change the UI (move, zoom image) programmatically based on input given by the gesture handler we would need to change the styles of the image. We normally use Stylesheet to assign the styles to the react native components. But to change styles based on the input of the gesture handler, we need dynamic styles which would change very frequently and fast.

Example: Imagine if we drag/move Image from x= 0 to x=20. The gesture handler will give us the updated value of x as 0, 0.01, 0.02, 0.03 up to 20. Now based on that if we need to change the styles, we will also need to change the styles very frequently so that it gets updated on the UI.

One option here is to use `useState`. We can store the `x` value and every time the gesture handler gives us the updated value we update our `useState` and hence complete component will be re-rendered again. Hence for every updated value 0, 0.01, 0.02, 0.03 up to 20, component will be re-rendered. As you guessed right, this is a major performance issue. We are rendering the component a lot of times and hence animation will not be smooth at all. Because we are doing everything on the javascript thread. `useState` lies on the javascript thread.

Threads : Major threads are of two types, javascript and UI threads.

The Javascript thread does all javascript calculations and the UI thread does UI calculations (rendering, displaying, calculating and updating layouts/values). If we update the state, javascript does the calculations and updates the `useState`. Now to see the updated value on the screen, the JS thread sends the updated value to the UI thread. How? through bridge. Imagine a bridge between UI and JS thread and communication and data exchange happens through this bridge. Now as we can imagine bridge will introduce overhead and complexity and will take some time to exchange and transfer data.

Hence if we update the `useState` very frequently it will involve a lot of to-and-fro data exchanges between javascript and UI threads. Hence it will cause a lot of delay and animation would be laggy.

Reanimated solves this problem. It does all the calculations on the UI thread itself. It spawns the new JS thread on the UI thread and hence does all JS and UI calculations on the UI thread itself. Hence there is no to and fro data exchange and no delay. Through reanimated, in the process of animation, there is not a single re-render on the JS side and animation feels a lot smoother.

Get Varun Kukade's stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email

Subscribe

Hence reanimated gives us the capability to create the state on the UI thread using the `useSharedValue` hook.

Also, functions are created on the UI thread itself. These functions are termed worklets.

Any updates to the `useSharedValue` variables should be done from worklets only.

Now in our use-case, we need to make style changes to the image. Hence we will create `useSharedValue` variables and update `useSharedValue` variables using worklets. `useSharedValue` variable would hold the style properties in our case.

Let's code it then...

```
import {useSharedValue} from 'react-native-reanimated';

const scale = useSharedValue(1);
const offset = useSharedValue({ x: 0, y: 0 });
```

Here we created a scale for zooming and offset for keeping the eye on the image position in the x and y direction. You just need to pass the default value to the hook. It accepts all types of data types similar to `useState`.

Let's give these values to our image component. For that let's create image component first

```
import Animated from 'react-native-reanimated';
import FastImage from 'react-native-fast-image';

const AnimatedFastImageComponent: any = Animated.createAnimatedComponent(FastImage);
```

Reanimated gives us the `createAnimatedComponent` function which creates an animated version of any react native component.

For `Flatlist`, `Image`, `ScrollView`, `Text` and `View`, we can directly use

```
<Animated.FlatList></Animated.FlatList>
<Animated.Image></Animated.Image>
```

```
<Animated.ScrollView></Animated.ScrollView>
<Animated.Text></Animated.Text>
<Animated.View></Animated.View>
```

Now let's give the styles to this component

```
import Animated, { useAnimatedStyle, useSharedValue } from 'react-native-reanimated';
import FastImage from 'react-native-fast-image';

const AnimatedFastImageComponent: any = Animated.createAnimatedComponent(FastImage);

const scale = useSharedValue(1);
const offset = useSharedValue({ x: 0, y: 0 });

const animatedStyle = useAnimatedStyle(() => ({
  transform: [
    { scale: scale.value },
    { translateX: offset.value.x },
    { translateY: offset.value.y },
  ],
}));

return (
  <AnimatedFastImageComponent
    {...props}
    style={[animatedStyle, style]}
    resizeMode={resizeMode}
  />
);
```

Here useAnimatedStyle is a hook given by reanimated which is a worklet and lies on the UI thread. It accepts the style properties which we need to change dynamically on the UI thread. In our case, we used translateX and translateY for moving the image in the x and y direction and scale for zooming the image.

Step 3 : Now we have an image ready with dynamic styles. But how the updates will happen? The answer is gesture handler will help us with that.

Let's add a gesture handler.

```

import Animated, { useAnimatedStyle, useSharedValue } from 'react-native-reanimated';
import FastImage from 'react-native-fast-image';
import { Gesture, GestureDetector } from 'react-native-gesture-handler';

const AnimatedFastImageComponent: any = Animated.createAnimatedComponent(FastImage);

const scale = useSharedValue(1);
const offset = useSharedValue({ x: 0, y: 0 });

const animatedStyle = useAnimatedStyle(() => ({
  transform: [
    { scale: scale.value },
    { translateX: offset.value.x },
    { translateY: offset.value.y },
  ],
}));

const composed = Gesture.Simultaneous(pinchGesture, panGesture);

return (
  <GestureDetector gesture={composed}>
    <AnimatedFastImageComponent
      {...props}
      style={[animatedStyle, style]}
      resizeMode={resizeMode}
    />
  </GestureDetector>
);

```

Here we add **GestureDetector** provided by the gesture handler. In our case, it detects the gesture of the first child component under it.

If you have the first child component under GestureDetector only to group the subsequent child views consider using the collapsable property. More on this [here](#).

Now let's see what is composed — GestureDetector has the capability to detect multiple gestures simultaneously. In our case, we would be zooming and moving image simultaneously in one go. Hence we used Gesture.Simultaneous to group pinchGesture and panGesture together.

If we want only one gesture at a time, we can directly use it like :

```

const tap = Gesture.Tap().onStart(() => {

```

```

        console.log('tap');
    });

    return (
        <GestureDetector gesture={tap}>
            <AnimatedFastImageComponent
                {...props}
                style={[animatedStyle, style]}
                resizeMode={resizeMode}
            />
        </GestureDetector>
    );

```

Now let's handle pinchGesture, and panGesture one by one.

Step 4 : Lets see panGesture

```

const start = useSharedValue({ x: 0, y: 0 });
const offset = useSharedValue({ x: 0, y: 0 });

const panGesture = useMemo(
    () =>
        Gesture.Pan()
            .enabled(true)
            .onUpdate(e => {
                offset.value = {
                    x: e.translationX + start.value.x,
                    y: e.translationY + start.value.y,
                };
            })
            .onEnd(() => {
                //if user lifts the finger while moving image in x and y direction, t
                offset.value = {
                    x: withSpring(0, {
                        stiffness: 60,
                        overshootClamping: true,
                    }),
                    y: withSpring(0, {
                        stiffness: 60,
                        overshootClamping: true,
                    }),
                };
                start.value = {
                    x: 0,
                    y: 0,
                };
            })

```

```
[offset, start],  
);
```

Here gesture handler provides us with the values when the pan gesture action is performed. On every update to the gesture, we would be receiving values in the `onUpdate` method. We also have a new shared value “start” to keep track of the start position of the image whenever the user ends the gesture.

Let's understand this by example —

1. At first “start” and “offset” x and y values will be 0.

```
const start = useSharedValue({ x: 0, y: 0 });  
const offset = useSharedValue({ x: 0, y: 0 });
```

2. Now if the image is moved 20 units in x direction, then `e.translationX` will contain 20. Hence total distance covered will be 20. For now, consider we haven't moved the image in y direction hence `e.translationY` will be 0

```
offset.value.x = e.translationX + start.value.x = 20 + 0 = 20  
offset.value.y = e.translationY + start.value.y = 0 + 0 = 0
```

3. Now as you can see we have used `offset.value.x` and `offset.value.y` value to give it to `translateX` and `translateY` styles property of the image. Hence here image will be moved in the x direction for 20 units and the image will stay as it is in the Y direction as `offset.value.y` is 0.

```
const animatedStyle = useAnimatedStyle(() => ({  
  transform: [  
    { scale: scale.value },  
    { translateX: offset.value.x },  
    { translateY: offset.value.y },  
  ],  
}));
```

One thing to note is `Gesture.Pan()` lies in the UI thread and hence whatever updates we make will be on the UI thread completely.

4. Now at the end when the user lifts the finger and stops the gesture, we need to get an image in its original position. Hence we reset the values of offset and start. Here we used spring animation provided by `reanimated` just to have a spring effect.

Step 5 : Now let's see `pinchGesture`

```
const MAX_SCALE = 2;
const MIN_SCALE = 1;

const scale = useSharedValue(1);
const savedScale = useSharedValue(1);

const pinchGesture = useMemo(
  () =>
    Gesture.Pinch()
      .onUpdate(e => {
        scale.value = savedScale.value * e.scale;
      })
      .onEnd(() => {
        savedScale.value = scale.value;
        if (savedScale.value < MIN_SCALE) {
          //do not allow zooming out below original/min scale
          scale.value = withSpring(MIN_SCALE, {
            stiffness: 60,
            overshootClamping: true,
          });
          savedScale.value = MIN_SCALE;
        } else if (savedScale.value > MAX_SCALE) {
          //do not allow zooming beyond max scale
          scale.value = withSpring(MAX_SCALE, {
            stiffness: 60,
            overshootClamping: true,
          });
          savedScale.value = MAX_SCALE;
        }
      }),
  [scale, savedScale],
);
```

Here gesture handler provides us with the values when the pinch gesture action is performed. On every update to the gesture, we would be receiving values in the `onUpdate` method. Here default value of scale would be 1 representing no zooming at all. Scale value < 1 will represent zooming out and scale value > 1 will represent zooming in. Here we also have used `savedScale.value` to keep track of the last value up to which user has scaled previously.

Let's understand this by example —

1. At first “scale” and “savedScale” values will be 1.
2. Now if the image is zoomed in/out we will be updating the scale value using the last scale value in the onUpdate function.
3. Now when the user lifts a finger, we want to keep the image in that scale only. Hence here we won't reset scale values. But we will update the current scale value as the last updated scale value.

```
savedScale.value = scale.value;
```

4. Now in this case we are disabling zooming out more than the original scale of the image. Hence we set MIN_SCALE as 1 saying that you cannot zoom out the component below this. Now when the gesture ends, and if the latest scale value is < 1 , we make the latest scale value = 1 using the spring effect.
5. Also we don't want users to zoom in infinite times. Hence we set MAX_SCALE as 2 and if the latest scale value > 2, we make the latest scale value to 2 saying that this is the max zoom-in you can perform.
6. After each update of gesture and at the end of the gesture, when the scale value gets updated it will be reflected in UI in no time as we have given scale value to scale property of the image .

```
const animatedStyle = useAnimatedStyle(() => ({
  transform: [
    { scale: scale.value },
    { translateX: offset.value.x },
    { translateY: offset.value.y },
  ],
}));
```

You can change and play around these values to achieve your desired behaviour.

Now our component is ready. Here's is complete code :

```

import React, {FC, ReactElement, useMemo} from 'react';
import FastImage, {FastImageProps} from 'react-native-fast-image';
import {Gesture, GestureDetector} from 'react-native-gesture-handler';
import Animated, {
  useAnimatedStyle,
  useSharedValue,
  withSpring,
} from 'react-native-reanimated';

const AnimatedFastImageComponent: any = Animated.createAnimatedComponent(
  FastImage as any,
);

const MAX_SCALE = 2;
const MIN_SCALE = 1;

export const AnimatedFastImage: FC<FastImageProps> = ({
  resizeMode = FastImage.resizeMode.contain,
  style,
  ...props
}): ReactElement => {
  const scale = useSharedValue(1);
  const savedScale = useSharedValue(1);
  const offset = useSharedValue({x: 0, y: 0});
  const start = useSharedValue({x: 0, y: 0});

  const pinchGesture = useMemo(
    () =>
      Gesture.Pinch()
        .onUpdate(e => {
          scale.value = savedScale.value * e.scale;
        })
        .onEnd(() => {
          savedScale.value = scale.value;
          if (savedScale.value < MIN_SCALE) {
            //do not allow zooming out below original/min scale
            scale.value = withSpring(MIN_SCALE, {
              stiffness: 60,
              overshootClamping: true,
            });
          }
          savedScale.value = MIN_SCALE;
        } else if (savedScale.value > MAX_SCALE) {
          //do not allow zooming beyond max scale
          scale.value = withSpring(MAX_SCALE, {
            stiffness: 60,
            overshootClamping: true,
          });
          savedScale.value = MAX_SCALE;
        }
      ),
    [scale, savedScale],
  );

```

```

);

const panGesture = useMemo(
  () =>
    Gesture.Pan()
      .enabled(true)
      .onUpdate(e => {
        offset.value = {
          x: e.translationX + start.value.x,
          y: e.translationY + start.value.y,
        };
      })
      .onEnd(() => {
        //if user take off finger while moving image in x and y direction, ta
        offset.value = {
          x: withSpring(0, {
            stiffness: 60,
            overshootClamping: true,
          }),
          y: withSpring(0, {
            stiffness: 60,
            overshootClamping: true,
          }),
        };
        start.value = {
          x: 0,
          y: 0,
        };
      }),
    [offset, start],
);

const animatedStyle = useAnimatedStyle(() => ({
  transform: [
    {scale: scale.value},
    {translateX: offset.value.x},
    {translateY: offset.value.y},
  ],
})));

const composed = Gesture.Simultaneous(pinchGesture, panGesture);

return (
  <GestureDetector gesture={composed}>
    <AnimatedFastImageComponent
      {...props}
      style={[animatedStyle, style]}
      resizeMode={resizeMode}
    />
  </GestureDetector>
);
};

```

You can find GitHub repo here — <https://github.com/varunkukade/Animated-Image>

That's it for this article. This was my first article in Medium. Do let me know if you have any feedback for me to improve.

If you found this tutorial helpful, don't forget to give this post 50 claps 🖐️ and follow 🚀 if you enjoyed this post and want to see more. Your enthusiasm and support fuel my passion for sharing knowledge in the tech community.

Edit : I covered wide range of topics on react native. You can find more of such articles on my profile -> <https://medium.com/@varunkukade999>

Stay tuned for more in-depth tutorials and insights on React Native development.

React Native Development

React Native Developers

React Native

React Native Tutorial

Mobile App Development



Follow

Written by Varun Kukade

390 followers · 27 following

Mobile Engineer 🚀 <https://github.com/varunkukade> varunkukade999@gmail.com

No responses yet



Write a response

What are your thoughts?

Open in app ↗

Sign up

Sign in

Medium

Search



Part 1: Push Notifications in React Native 2024



Varun Kukade

Part 1: Push Notifications in React Native 2024

I recently had a chance to deep dive into Push notifications at my workplace and learned a lot of things. I had to start with what is push...

Sep 29, 2024




531



8



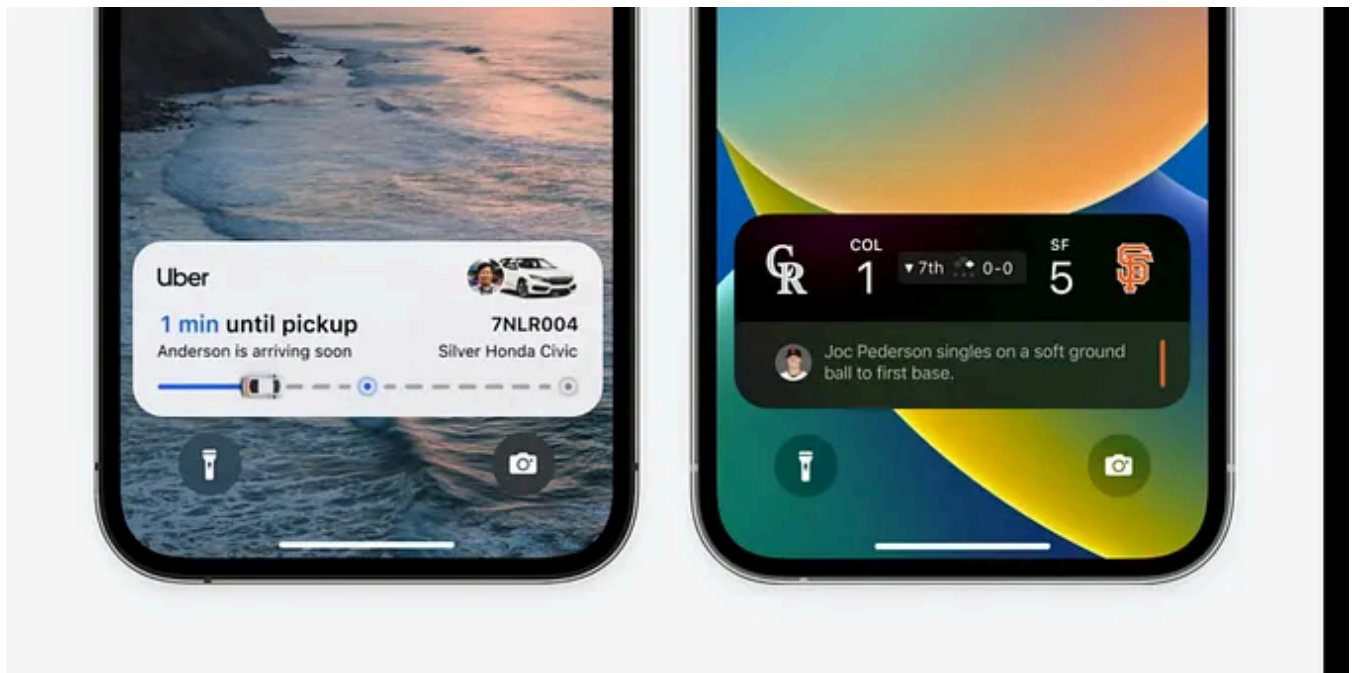



 Varun Kukade

Part 1: Build your first AI agent + chatbot with Langchain and LangGraph in Python

In this article, we'll learn how to build a simple AI chatbot + agent using LangChain and LangGraph. If you're already familiar with AI...

Jul 20, 2025  27  2

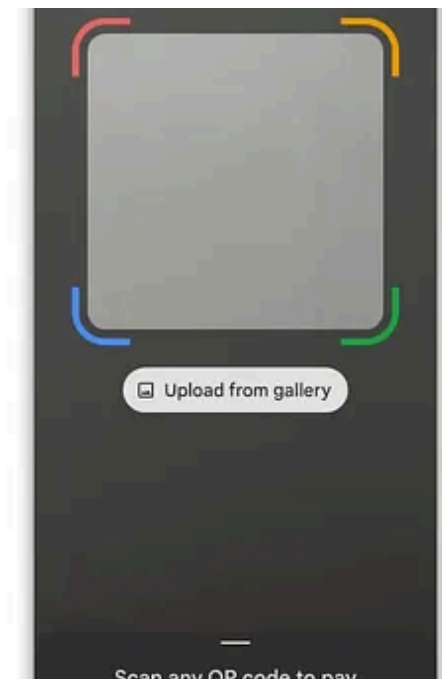


 Varun Kukade

Part 1: IOS Live Activities in React Native

At my workplace, Altir, I recently integrated IOS Live Activities into one of our core products, Hyperfuel, using React Native. During the...

Apr 28, 2025 🖱 155 💬 1



Varun Kukade

QR code scanner in React Native

In this article, we will see how to create a QR code scanner in react native.

Mar 17, 2024 🖱 296 💬 3



See all from Varun Kukade

Recommended from Medium

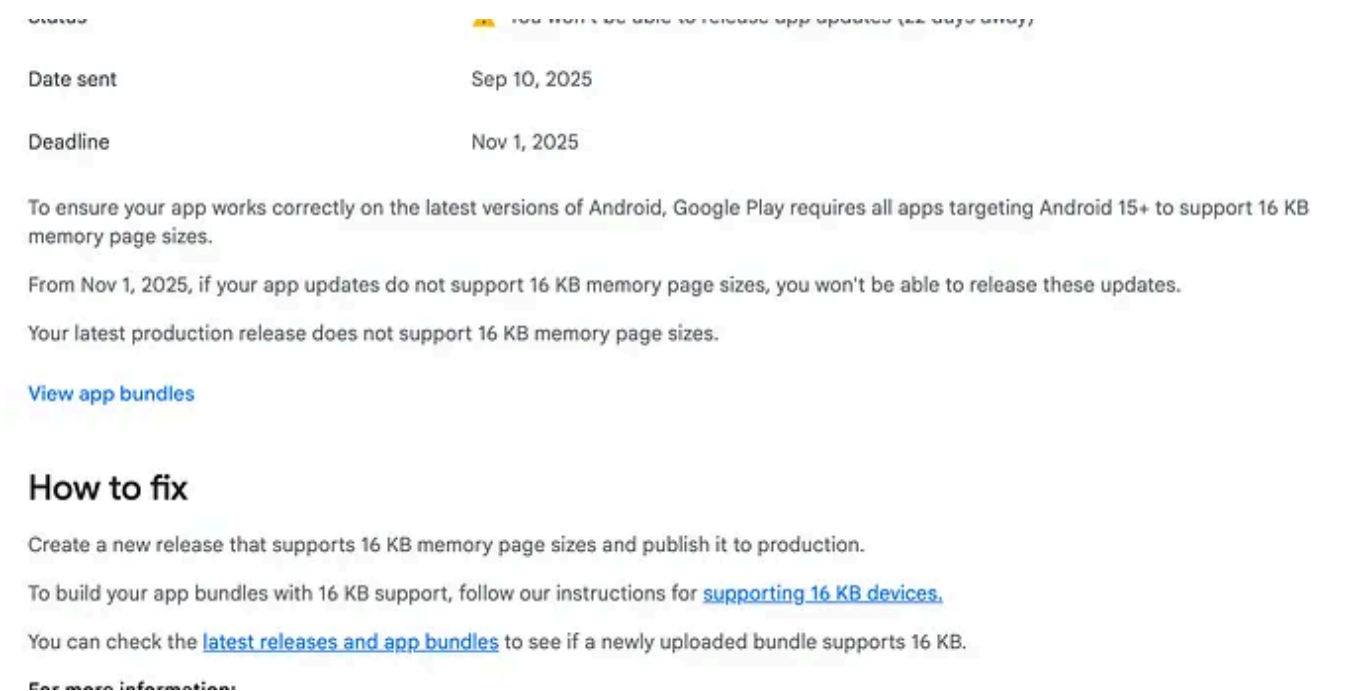


 Shanavas Shaji

Upgrading to Expo 54 and React Native 0.81: A Developer's Survival Story

When Expo 54 dropped, I didn't jump in just for the shiny new features. I had three pressing reasons:

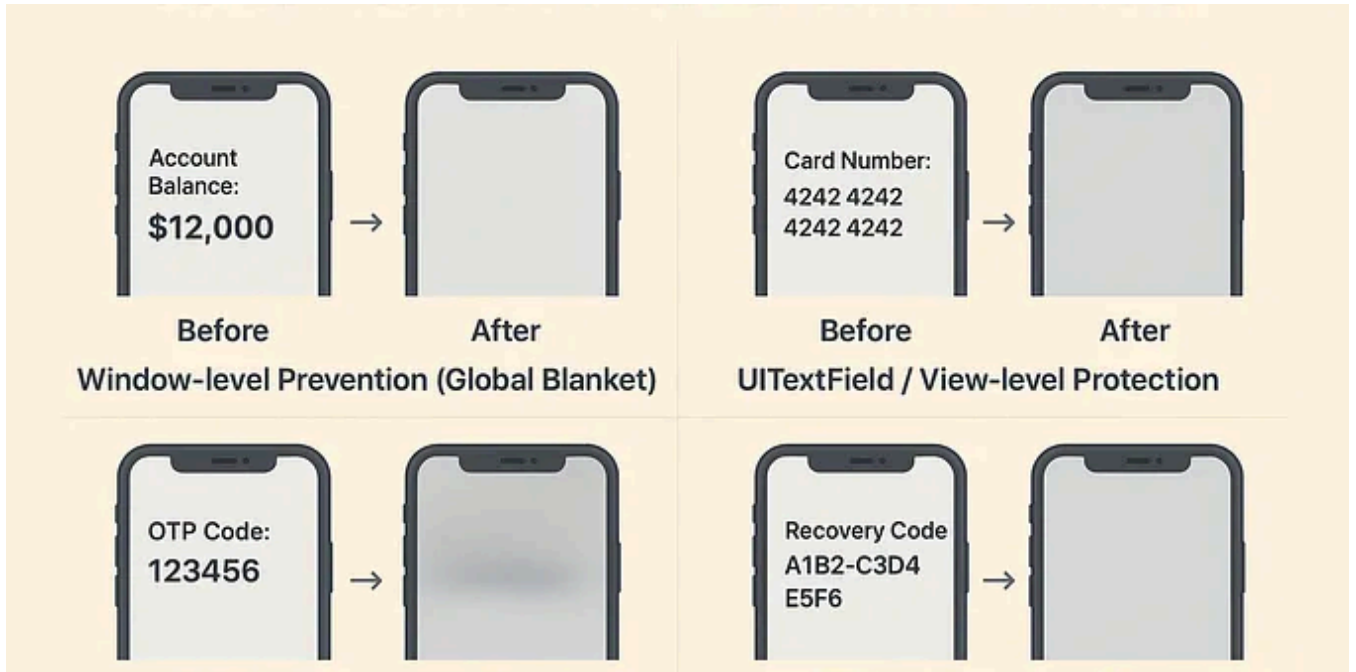
Sep 12, 2025  151  2




 Hemanth KV

Fixing the 16 KB Page Size Issue in React Native Apps (Android 15+ Compatibility)

 Updated for October 2025



 Neeshu Kumar

Screenshot Prevention in iOS: Advanced Techniques Inspired by Banking & Authenticator Apps

If you've ever used a banking app or an authenticator app, you've probably noticed a unique security feature: you can't take a screenshot...



No User Left Behind: Accessibility in React Native

Inclusive code creates inclusive communities.

★ Sep 2, 2025 🖱️ 1



 In React Native Lab by Jayant Kumar

`useKeyboard` Hook in React Native: Manage Keyboard Visibility Gracefully

Not a Medium Member? “Read For Free”

★ Dec 16, 2025 🖱️ 190



Feature	FlatList	FlashList
Library	React Native core	@shopify/flash-list
Rendering	Virtualized list	Optimized virtualization
Performance	Moderate	3–10× faster
Memory Usage	Higher	Lower
Required Setup	None	Needs <code>estimatedItemSize</code>
Smoothness	Can drop frames	Consistent 60 FPS
Sticky Headers	✅ Supported	✅ Supported
Cross-Platform	iOS, Android, Web (limited)	iOS, Android



Shreyak

FlashList vs FlatList in React Native: Which One Should You Use for Better Performance?

Discover the key differences between FlashList and FlatList in React Native. Learn how FlashList delivers smoother performance, faster...

🌟 Nov 3, 2025



See more recommendations