

Performance Overview

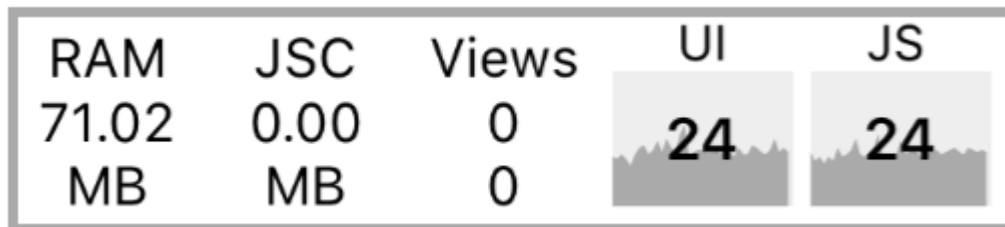
A compelling reason for using React Native instead of WebView-based tools is to achieve 60 frames per second and a native look and feel to your apps. Where possible, we would like for React Native to do the right thing and help you to focus on your app instead of performance optimization, but there are areas where we're not quite there yet, and others where React Native (similar to writing native code directly) cannot possibly determine the best way to optimize for you and so manual intervention will be necessary. We try our best to deliver buttery-smooth UI performance by default, but sometimes that isn't possible.

This guide is intended to teach you some basics to help you to troubleshoot performance issues, as well as discuss common sources of problems and their suggested solutions.

What you need to know about frames

Your grandparents' generation called movies "moving pictures" for a reason: realistic motion in video is an illusion created by quickly changing static images at a consistent speed. We refer to each of these images as frames. The number of frames that is displayed each second has a direct impact on how smooth and ultimately life-like a video (or user interface) seems to be. iOS devices display 60 frames per second, which gives you and the UI system about 16.67ms to do all of the work needed to generate the static image (frame) that the user will see on the screen for that interval. If you are unable to do the work necessary to generate that frame within the allotted 16.67ms, then you will "drop a frame" and the UI will appear unresponsive.

Now to confuse the matter a little bit, open up the Dev Menu in your app and toggle `Show Perf Monitor`. You will notice that there are two different frame rates.



JS frame rate (JavaScript thread)

For most React Native applications, your business logic will run on the JavaScript thread. This is where your React application lives, API calls are made, touch events are processed, etc... Updates to native-backed views are batched and sent over to the native side at the end of each iteration of the event loop, before the frame deadline (if all goes well). If the JavaScript thread is unresponsive for a frame, it will be considered a dropped frame. For example, if you were to call `this.setState` on the root component of a complex application and it resulted in re-rendering computationally expensive component subtrees, it's conceivable that this might take 200ms and result in 12 frames being dropped. Any animations controlled by JavaScript would appear to freeze during that time. If anything takes longer than 100ms, the user will feel it.

This often happens during `Navigator` transitions: when you push a new route, the JavaScript thread needs to render all of the components necessary for the scene in order to send over the proper commands to the native side to create the backing views. It's common for the work being done here to take a few frames and cause jank because the transition is controlled by the JavaScript thread. Sometimes components will do additional work on `componentDidMount`, which might result in a second stutter in the transition.

Another example is responding to touches: if you are doing work across multiple frames on the JavaScript thread, you might notice a delay in responding to `TouchableOpacity`, for example. This is because the JavaScript thread is busy and cannot process the raw touch events sent over from the main thread. As a result, `TouchableOpacity` cannot react to the touch events and command the native view to adjust its opacity.

UI frame rate (main thread)

Many people have noticed that performance of `NavigatorIOS` is better out of the box than `Navigator`. The reason for this is that the animations for the transitions are done entirely on the main thread, and so they are not interrupted by frame drops on the JavaScript thread.

Similarly, you can happily scroll up and down through a `ScrollView` when the JavaScript thread is locked up because the `ScrollView` lives on the main thread. The scroll events are dispatched to the JS thread, but their receipt is not necessary for the scroll to occur.

Common sources of performance problems

Running in development mode (`dev=true`)

JavaScript thread performance suffers greatly when running in dev mode. This is unavoidable: a lot more work needs to be done at runtime to provide you with good warnings and error messages, such as validating `propTypes` and various other assertions. Always make sure to test performance in release builds.

Using `console.log` statements

When running a bundled app, these statements can cause a big bottleneck in the JavaScript thread. This includes calls from debugging libraries such as redux-logger, so make sure to remove them before bundling. You can also use this babel plugin that removes all the `console.*` calls. You need to install it first with `npm i babel-plugin-transform-remove-console --save-dev`, and then edit the `.babelrc` file under your project directory like this:

```
{
  "env": {
    "production": {
      "plugins": ["transform-remove-console"]
    }
  }
}
```

This will automatically remove all `console.*` calls in the release (production) versions of your project.

It is recommended to use the plugin even if no `console.*` calls are made in your project. A third party library could also call them.

ListView initial rendering is too slow or scroll performance is bad for large lists

Use the new `FlatList` or `SectionList` component instead. Besides simplifying the API, the new list components also have significant performance enhancements, the main one being nearly constant memory usage for any number of rows.

If your `FlatList` is rendering slow, be sure that you've implemented `getItemLayout` to optimize rendering speed by skipping measurement of the rendered items.

JS FPS plunges when re-rendering a view that hardly changes

If you are using a `ListView`, you must provide a `rowHasChanged` function that can reduce a lot of work by quickly determining whether or not a row needs to be re-rendered. If you are using immutable data structures, this would only need to be a reference equality check.

Similarly, you can implement `shouldComponentUpdate` and indicate the exact conditions under which you would like the component to re-render. If you write pure components (where the return value of the render function is entirely dependent on props and state), you can leverage `PureComponent` to do this for you. Once again, immutable data structures are useful to keep this fast -- if you have to do a deep comparison of a large list of objects, it may be that re-rendering your entire component would be quicker, and it would certainly require less code.

Dropping JS thread FPS because of doing a lot of work on the JavaScript thread at the same time

"Slow Navigator transitions" is the most common manifestation of this, but there are other times this can happen. Using `InteractionManager` can be a good approach, but if the

user experience cost is too high to delay work during an animation, then you might want to consider `LayoutAnimation`.

The `Animated` API currently calculates each keyframe on-demand on the JavaScript thread unless you set `useNativeDriver: true`, while `LayoutAnimation` leverages `Core Animation` and is unaffected by JS thread and main thread frame drops.

One case where I have used this is for animating in a modal (sliding down from top and fading in a translucent overlay) while initializing and perhaps receiving responses for several network requests, rendering the contents of the modal, and updating the view where the modal was opened from. See the [Animations guide](#) for more information about how to use `LayoutAnimation`.

Caveats:

- `LayoutAnimation` only works for fire-and-forget animations ("static" animations) -- if it must be interruptible, you will need to use `Animated`.

Moving a view on the screen (scrolling, translating, rotating) drops UI thread FPS

This is especially true when you have text with a transparent background positioned on top of an image, or any other situation where alpha compositing would be required to re-draw the view on each frame. You will find that enabling `shouldRasterizeIOS` or `renderToHardwareTextureAndroid` can help with this significantly.

Be careful not to overuse this or your memory usage could go through the roof. Profile your performance and memory usage when using these props. If you don't plan to move a view anymore, turn this property off.

Animating the size of an image drops UI thread FPS

On iOS, each time you adjust the width or height of an `Image` component it is re-cropped and scaled from the original image. This can be very expensive, especially for large images. Instead, use the `transform: [{scale}]` style property to animate the size. An example of when you might do this is when you tap an image and zoom it in to full screen.

My TouchableX view isn't very responsive

Sometimes, if we do an action in the same frame that we are adjusting the opacity or highlight of a component that is responding to a touch, we won't see that effect until after the `onPress` function has returned. If `onPress` does a `setState` that results in a lot of work and a few frames dropped, this may occur. A solution to this is to wrap any action inside of your `onPress` handler in `requestAnimationFrame`:

```
handleOnPress() {  
  requestAnimationFrame(() => {  
    this.doExpensiveAction();  
  });  
}
```


Slow navigator transitions

As mentioned above, `Navigator` animations are controlled by the JavaScript thread. Imagine the "push from right" scene transition: each frame, the new scene is moved from the right to left, starting offscreen (let's say at an x-offset of 320) and ultimately settling when the scene sits at an x-offset of 0. Each frame during this transition, the JavaScript thread needs to send a new x-offset to the main thread. If the JavaScript thread is locked up, it cannot do this and so no update occurs on that frame and the animation stutters.

One solution to this is to allow for JavaScript-based animations to be offloaded to the main thread. If we were to do the same thing as in the above example with this approach, we might calculate a list of all x-offsets for the new scene when we are starting the transition and send them to the main thread to execute in an optimized way. Now that the JavaScript thread is freed of this responsibility, it's not a big deal if it drops a few frames while rendering the scene -- you probably won't even notice because you will be too distracted by the pretty transition.

Solving this is one of the main goals behind the new [React Navigation](#) library. The views in `React Navigation` use native components and the [Animated](#) library to deliver 60 FPS animations that are run on the native thread.

Is this page useful?  

 Edit this page

*Last updated on **Jun 21, 2023***