{callstack}

React Native Optimization

THE ULTIMATE GUIDE

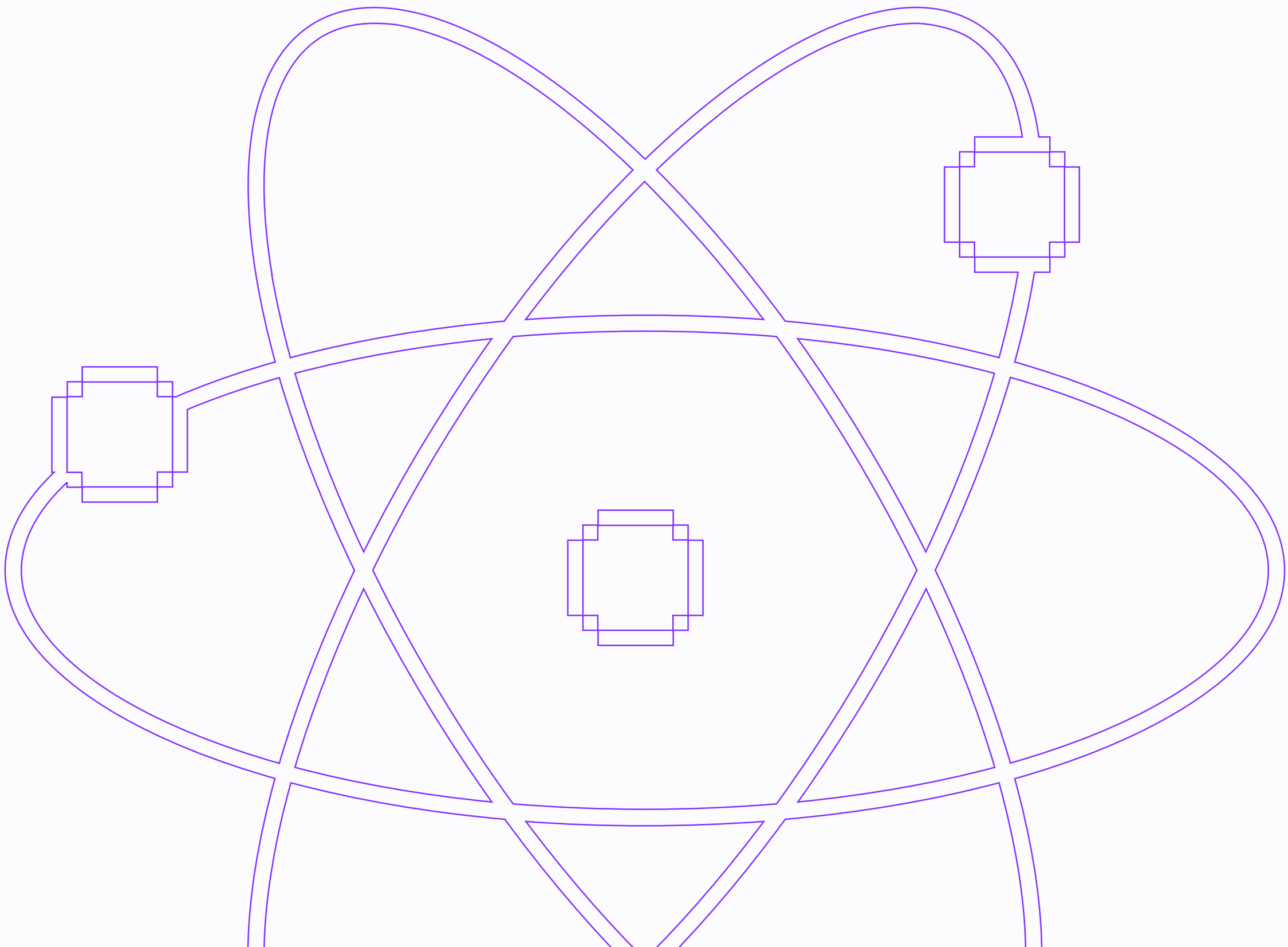


TABLE OF CONTENTS

Part 3: Bundling

Acknowledgements

PREFACE

React Native has come a long way. From the early days as a promising experiment in cross-platform development to adoption by some of the world's largest enterprises, its evolution has been nothing short of remarkable. The framework has matured, proving capable of handling the demands of high-scale applications while continuing to refine and push its technical boundaries.

For years, we have seen React Native introduce new features, often groundbreaking, with every release. Today, we see steady, low-level improvements that often enhance performance, stability, and scalability. These refinements reflect how React Native is being adopted more widely, reinforcing its role as a mature and enterprise-ready framework.

A significant milestone in this journey was the introduction of the New Architecture, which redefined many core aspects of the framework, including how we think about development in general and performance optimizations. It brought new features and best practices, enabling developers to build even more efficient and scalable applications.

And so, we decided it was time to overhaul our guide to optimization completely. The landscape has changed, and so must our approach. Some techniques that were once essential are no longer relevant; others have taken on new importance. In this new edition, we aim to equip developers with the latest insights, tools, and strategies to make the most of React Native's evolving capabilities.

Whether you're an experienced React Native engineer or just getting started, this guide goes beyond performance best practices. It also provides essential knowledge to help you understand React Native in general, including what happens under the hood. We rely on that knowledge every day to build better, more efficient, and scalable applications, and we hope it will help you do the same.

— Mike Grabowski, Callstack's Founder & CTO

HOW TO READ THIS BOOK

This book is intended for React Native developers of various levels of experience. We believe that both newcomers and seasoned React Native devs, regardless of whether they come from a web or native app development background, will find something valuable and applicable in their apps.

We acknowledge that at a given time you may not be interested in all the optimizations presented in this book. Oftentimes, your focus will be on a specific area, such as optimizing React re-renders. That's why, although you can read the whole book linearly, we made sure it's easy to open on any given chapter and get right to the topic of your interest.

What's waiting for you inside?

To make it easier for everyone to find relevant information, we split this guide into three parts focusing on different kinds of optimization you may be particularly interested in: the React side, the Native side, and overall build-time optimizations. All three parts contain an introduction to the main topic, detailed guides, and best practices to help you improve the most important metrics describing your app's performance: FPS (Frames Per Second) and TTI (Time to Interactive).

Here's what you can expect in each part:

- **Introduction**—where we present key topics, terms, and ideas related to the main topic.
- **Guides**—where we explain how to use specialized tools and measure important metrics.
- **Best practices**—where we show you what to do to make your app run and initialize faster.

Conventions used in this book

To better illustrate the ideas presented in this book, we've included a lot of short code snippets, screenshots from the tools we use, and diagrams.

We link to external resources, such as official docs or libraries, but also to other chapters of this book.

To give you extra context on certain topics, we decided to put them in callouts—while not intended for linear reading, we highly recommend not skipping those when reading.

About Callstack

At Callstack, we are committed to advancing React Native and empowering developers to build high-performance applications. As core contributors and Meta partners, we work closely with the community—shaping proposals, maintaining key modules, and driving the framework's evolution. Our team actively contributes to React Native's monthly releases, ensuring developers have access to the latest improvements. By sharing our expertise, tools, and best practices, we help teams solve performance challenges, optimize their apps, and push the boundaries of what's possible with React Native.

Let's get in touch

React Native thrives because of its community, and you can be part of its evolution. Whether by optimizing your own apps, contributing to open-source projects, or staying informed about the latest advancements, your involvement helps push the ecosystem forward.

Here's how you can stay engaged:

- **Stay up to date**—subscribe to our newsletter and follow our latest insights, open-source contributions, and best practices.
- **Join the conversation**—Join our Discord server to discuss proposals, share your ideas, and help shape the future of React Native.
- **Contribute**—explore and collaborate on our open-source projects.

And if you want to get in touch, contact us anytime. Let's build a faster, better React Native together!

WHY PERFORMANCE MATTERS

When building mobile apps, performance is not just a technical concern—it's a user experience priority. A fast, responsive app can make the difference between a delighted user and one who abandons your app entirely. But what does "fast" really mean? Is it about raw numbers or maybe a user's perception?

The user's perspective

Perceived performance is all about how fast your app *feels* to the user. It's not just about raw numbers or benchmarks—it's about creating the illusion of speed.

In the 1940s, an office building in New York faced complaints about slow elevators. To address this, instead of speeding up the elevators, the building manager installed mirrors near the elevators. This simple change distracted people by giving them something to do (look at themselves), reducing perceived wait times, and effectively solving the issue. While this story may be more of an urban legend, the concept itself is rooted in valid psychological principles about perceived time and distraction that influence the feeling of how fast a particular event is.

When a mobile app takes a few seconds to load, showing a splash screen, skeleton UI, or even a game to play can make users feel like the app is responsive and ready to use. That's why perceived performance is often more important than actual performance in shaping user satisfaction.

However, focusing solely on perceived performance can be misleading. While tricks like animations, placeholders, or preloading content can improve the user's perception, they don't address the underlying performance issues. That's where measurable metrics like TTI and FPS show their value.

The metrics that matter: TTI and FPS

Out of the many metrics you can measure and monitor, two have the most impact on how users perceive the speed of your app. It's how quickly they can interact with the app described by TTI and how fluid the app feels when interacting with it described by FPS at a given time.

Time to Interactive (TTI)

Measuring the app's boot-time performance is described by the TTI metric. It measures how quickly your app becomes usable after launch. This is the moment when users can start interacting with the app without delays or jank. But TTI isn't just about the app's initial load. Companies often track variations of this metric, such as Time to Home (how long it takes to load the home screen) or Time to Specific Screen (how long it takes to navigate to a key feature). These metrics are especially important for apps with complex navigation flows or heavy initial data fetching.

If your app takes too long to load, users may abandon it before they even get a chance to explore its features or simply get fed up with it being slow anytime they try to do something, making them feel miserable and starting to associate your app, and sometimes even its brand, with that feeling.

Frames Per Second (FPS)

Once your app is up and running, FPS becomes the key metric for runtime performance. FPS measures how smoothly your app responds to user interactions, such as scrolling, swiping, or tapping buttons. A high FPS (ideally 60 frames per second or more) ensures that animations and transitions feel smooth and natural. When FPS drops, user experience lags, and animations begin to stutter, making your app feel unpolished or slow to respond. This is especially critical for apps with rich visual content or complex interactions.

What does fast even mean?

As much as we love data and a scientific approach to validating our hypotheses, we realize that there is insufficient amount of independent and high-quality research on what "fast" means for mobile users. Usually, it's in the form of big tech companies boasting about their conversion numbers or paid reports based on surveys.

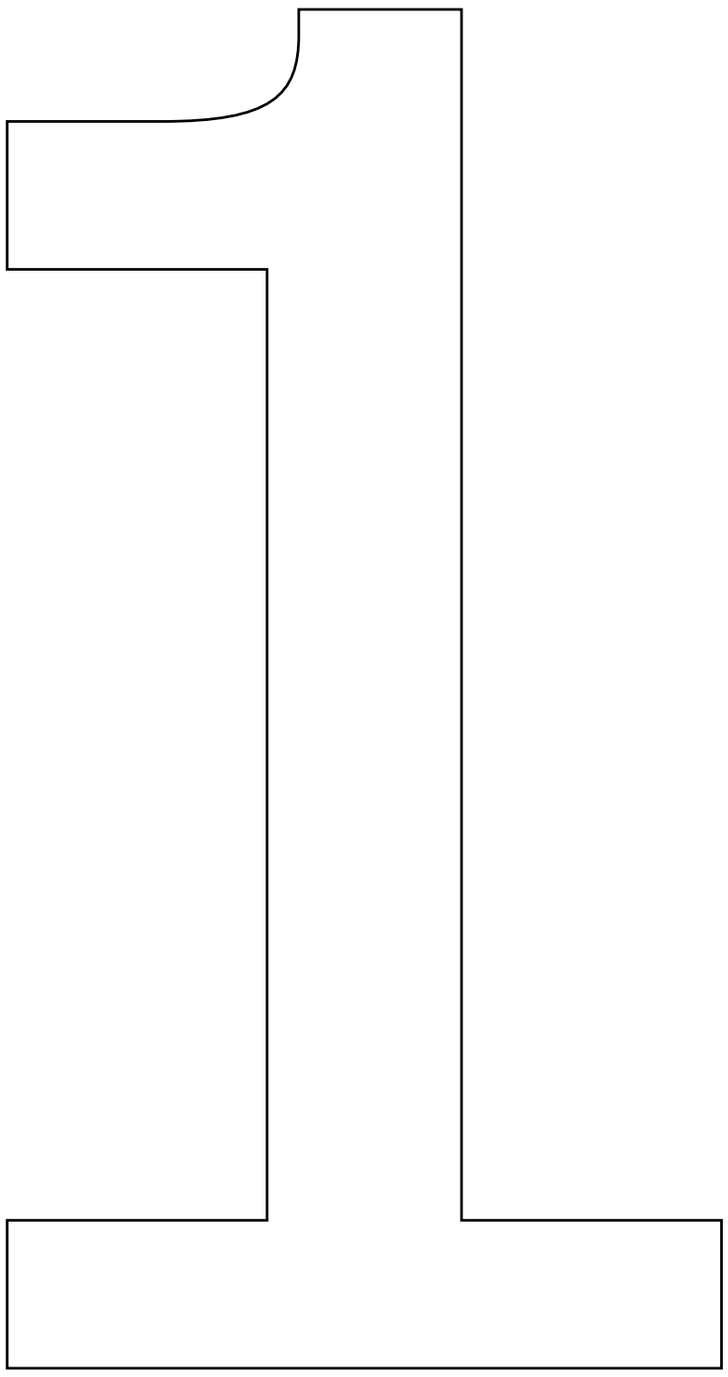
Regardless, what's even more important is that the feeling of speed is a moving target. With access to higher-end and better-performing devices, while using plenty of different apps on a single smartphone, users' expectations are growing higher. It's also worth remembering that these expectations are vastly dependent on the demographic—e.g., younger generations will typically expect apps to load and operate faster than their parents and grandparents, who are used to things taking more time. The expectations and incentives to improve are also different for internal enterprise app users compared to individual customers.

What will typically give you the most reliable indication of whether your app is fast enough is knowing your users and your competition. The analytics you gather on your users' behavior will give you personalized insights into places where they tend to drop. Analyzing whether your competitors' apps are objectively faster or slower, on the other hand, will hint at whether performance is potentially the reason you're losing users.

In the first two parts of this book, we'll focus on how you can improve the runtime performance through JavaScript, React, and native optimizations. We will guide you through all the necessary tools and techniques you can utilize to better understand what influences FPS so you can fix it anywhere.

In the third part, we'll focus on the bundling processes of your app, which happen both on the JavaScript and native sides and are highly correlated with the boot-time performance. Similar to the first two parts of the guide, you'll be presented with everything you need to have a better overview of what's causing your app to load slowly and how to make it fast.

PART



JAVASCRIPT

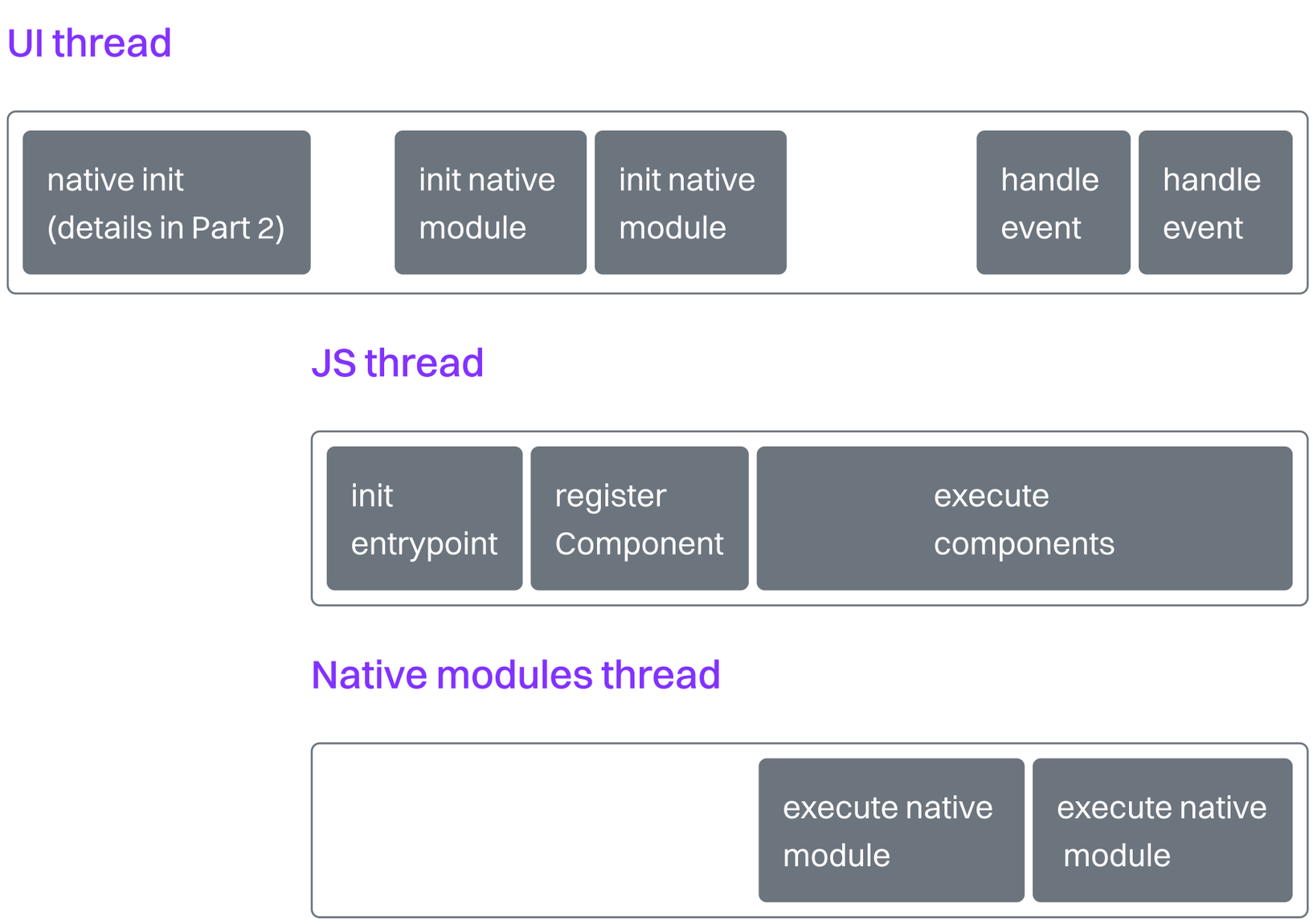
Guides and techniques to improve FPS by optimizing JavaScript and React side of React Native

Introduction

In the first part of this guide, we'll focus on the JavaScript part of the React Native ecosystem. As you probably already realize, React Native leverages the native platform primitives and glues together various technologies: Kotlin or Java for Android, Swift or Objective-C for iOS, C++ for React Native's core runtime, and JavaScript executed by the Hermes engine. It's a lot to take in, so let's break it down a bit and focus on one platform for simplicity: Android.

When the user opens a native Android app, it typically goes through a similar initialization path: the app opens on the main thread, it loads the native code—usually written in Kotlin, Java, or C++—into memory, and then executes this code, which usually results in some kind of UI being shown to the user. It's not so different for an Android app created with React Native. After all, it's a native app but with a twist: part of the native code being executed comes from React Native.

JavaScript initialization path



A simplified threading model of React Native in New Architecture presenting context in which JS is executed

React Native initialization code handles a handful of key mechanisms:

- Initialization of React Native internals—cross-platform C++ React renderer, JavaScript Interface (JSI), Hermes JS engine, layout engine (Yoga)—**on the main thread**.
- Initialization of the **JavaScript thread** that runs JS and communicates back and forth with the main thread through the JSI.

- Initialization of the **Native modules thread** that runs lazily loaded Turbo Modules.

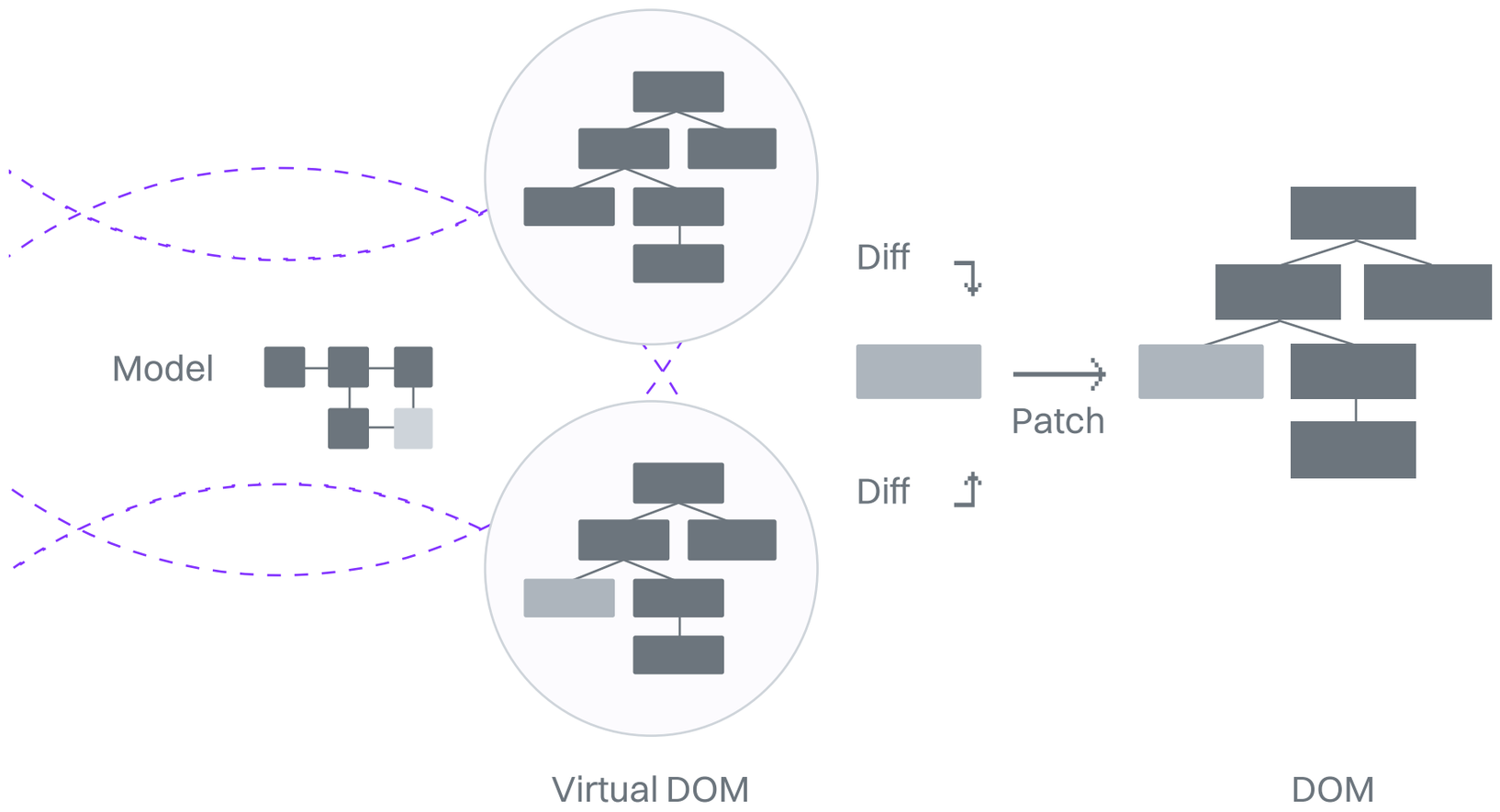
Notice how rendering logic (Kotlin, C++) and the so-called "business logic" are, by design, run on separate threads—the main thread vs. the JS thread. Since this part is about JavaScript and React, we'll focus on the JavaScript side that enables React to run on a dedicated JS thread and how this model affects the performance of your React Native apps.

According to an internal survey of 100 React Native developers at Callstack, 80% of the performance challenges they face in mobile, TV, and desktop React Native apps originate from the JavaScript side. This sample may not be representative of the real world due to its small size. However, seeing what the React Native Community is struggling with, it's pretty safe to assume that *most* of the performance issues are related to JavaScript. As such, it's wise to focus on these issues before moving to more advanced native optimizations described in Part 2: Native.

React re-rendering model

React takes care of rendering and updating the application UI based on its state for you, regardless of the environment (or platform): web, iOS, Android, or anything else. The **`react`** library itself is tiny and consists mostly of public API definitions, some cross-platform functionality, and a reconciliation algorithm, which is responsible for efficiently updating the UI to reflect component state changes, effectively describing "what to do".

What makes this model powerful, though, is splitting the "what" from the "how" and deferring the latter to specialized renderers that manage how these updates are applied to different environments: **`react-dom`** for the web, **`react-native`** for iOS and Android, or **`react-native-windows`** for Windows, to name a few. This model allows you to define components of your app universally for all supported platforms simultaneously, and compose the final interface out of these smaller building blocks. With such an approach, you don't control the application rendering lifecycle; React and its renderer do the job for you.



Visual description of React reconciliation algorithm

In other words, deciding when to repaint things on screen is purely React's responsibility, while deciding about the "how" is the renderer's responsibility. React looks out for the changes you have done to your components, compares them, and, by design, only performs the required and the smallest number of actual updates.

React component re-renders in the following cases:

- Parent component re-renders
- State (including hooks) changes
- Props change
- Context changes
- Force update (escape hatch)

Before we move on to the practices that may or may not be relevant to your app's specific use case, let's start with something that you will find useful in every app you're going to build: profiling and measuring.

GUIDE

HOW TO PROFILE JS AND REACT CODE

When optimizing performance, we want to know exactly which code path is the source of a slowdown. With some experience and knowledge of the codebase and common performance pitfalls, developers can often identify the issues intuitively and expect the app to run faster. In reality, though, apps are usually too complex for anyone to know the whole system by heart. Typical blind optimizations we make have little impact on the overall performance. To be precise with our actions, we must make decisions guided by data.

The data comes from measuring performance using specialized tools. This process is often referred to as "profiling", as it involves analyzing a program to create a "profile" of its execution. This profile typically includes information about which parts of the program consume the most resources, such as CPU time or memory, helping identify performance bottlenecks.

Profiling with React Native DevTools

In the context of a React Native app, it's helpful to be able to profile React itself to inspect unnecessary re-renders. The best tool to profile React running in a mobile app is React Native DevTools, and that's what we'll focus on in this guide.

React Profiler is integrated as the default plugin in React Native DevTools and can produce a flame graph of the React rendering pipeline as a result of profiling. We can leverage this data to analyze the app's re-rendering issues.

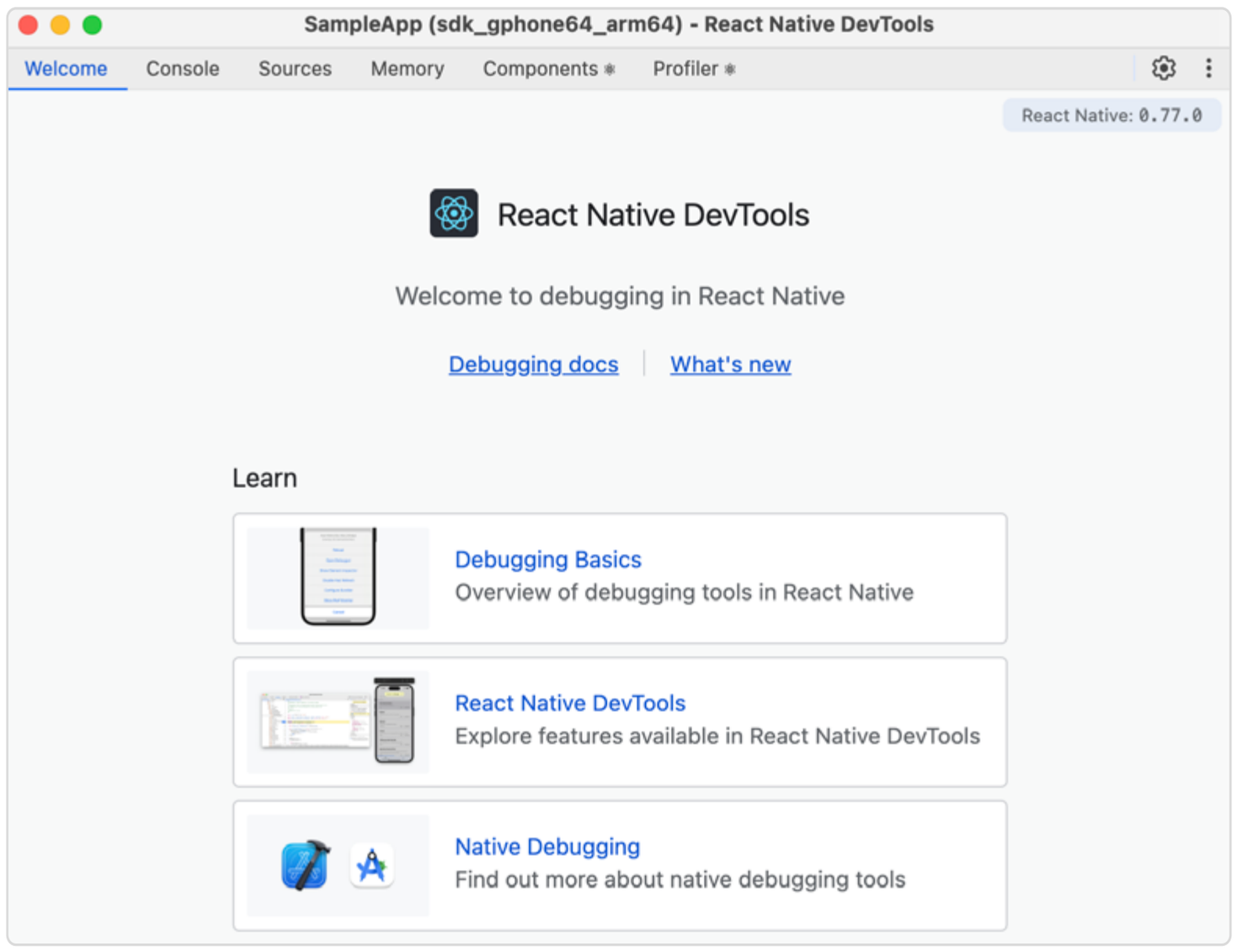
Here is the code we're about to profile:

```
export const App = () => {
  const [count, setCount] = React.useState(0);
  const [second, setSecond] = React.useState(0);
  return (
    <View style={styles.container}>
```

```
      <Text>Welcome!</Text>
      <Text>{count}</Text>
      <Text>{second}</Text>
      <Button onPress={() => setCount(count + 1)} title="Press one"
/>
      <Button onPress={() => setSecond(second + 1)} title="Press
two" />
    </View>
  );
};

const Button = ({onPress, title}) => {
  return (
    <Pressable style={styles.button} onPress={onPress}>
      <Text>{title}</Text>
    </Pressable>
  );
};
```

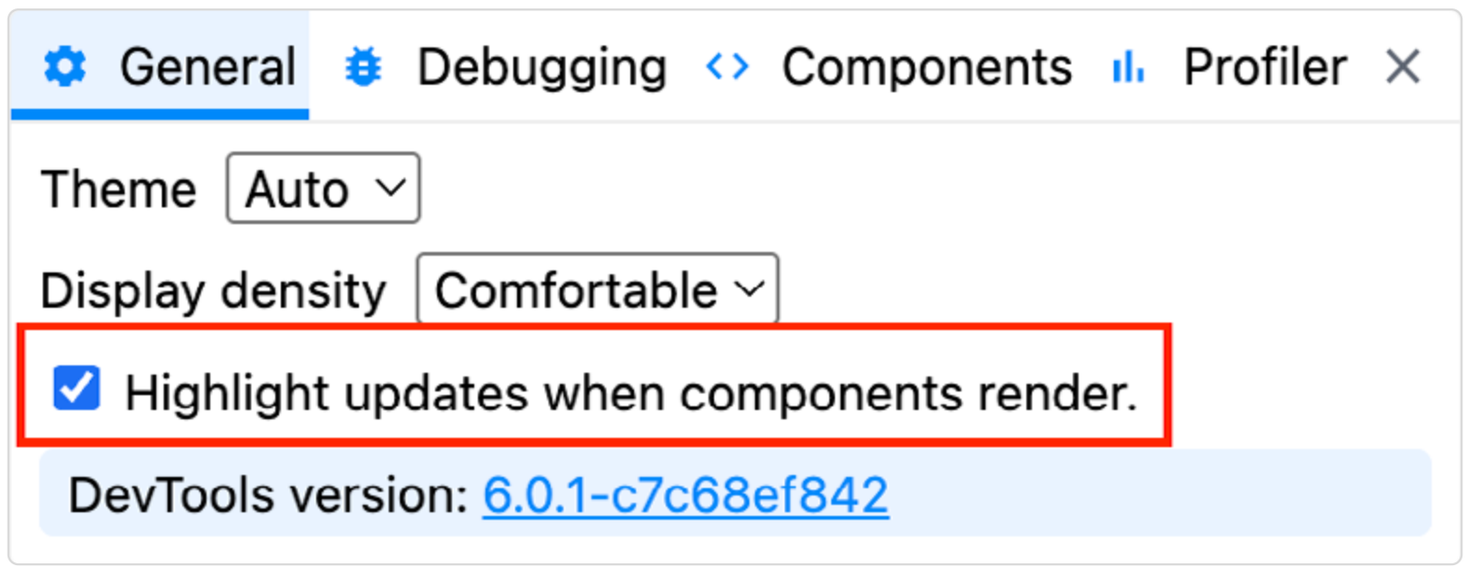
When rendered, the `<App />` component will display a welcome message with two counters and buttons controlled by two separate state handlers. We're now ready to profile this React code with React Native DevTools.

You can access it by either pressing `j` in the Metro dev server or from the app using the React Native Dev Menu—accessed through a shake gesture on a device, `Cmd+M` on Android or `Cmd+D` on iOS—and pressing "Open DevTools". It will present you with a welcome screen, pointing to learning resources on the basics of debugging, DevTools features, and native debugging. We highly recommend getting familiar with these resources, some overlapping with the materials presented in this guide.



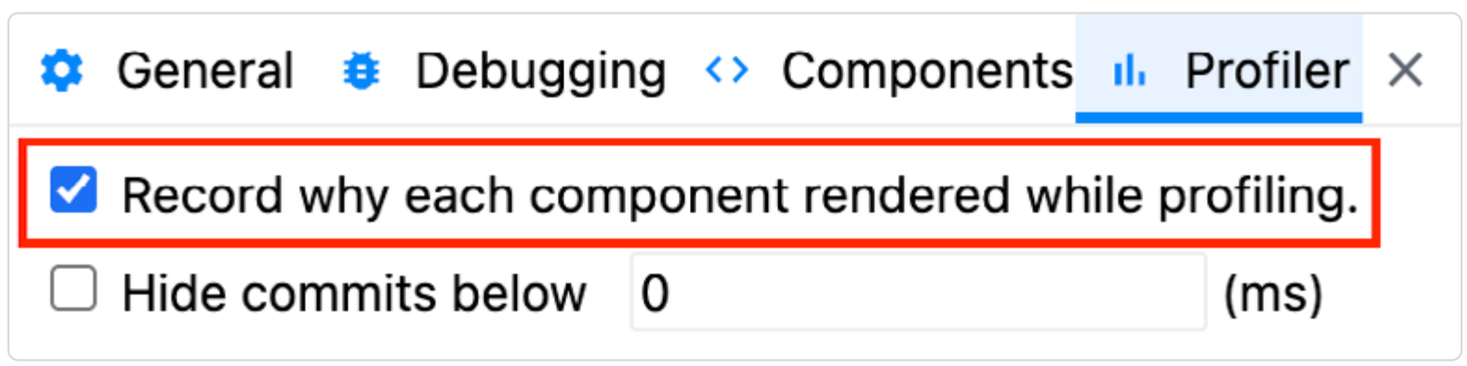
React Native DevTools welcome screen

Now, open the React Native DevTools. Go to the Profiler tab on top. Press the "gear" button to configure it to "Highlight updates when components render"



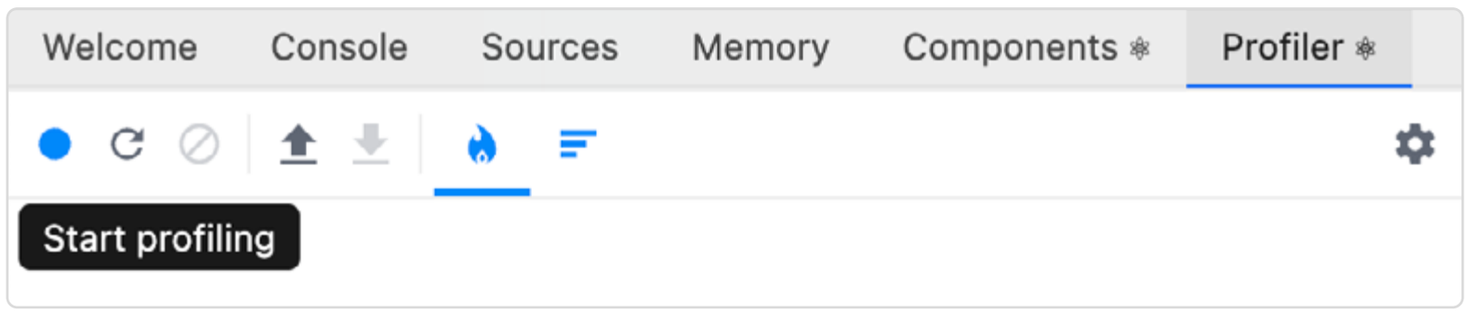
General profiler settings accessed from "View Settings" icon in the Profiler tab of React Native DevTools

and "Record why each component rendered while profiling".



Profiler settings accessed from "View Settings" icon in the Profiler tab of React Native DevTool

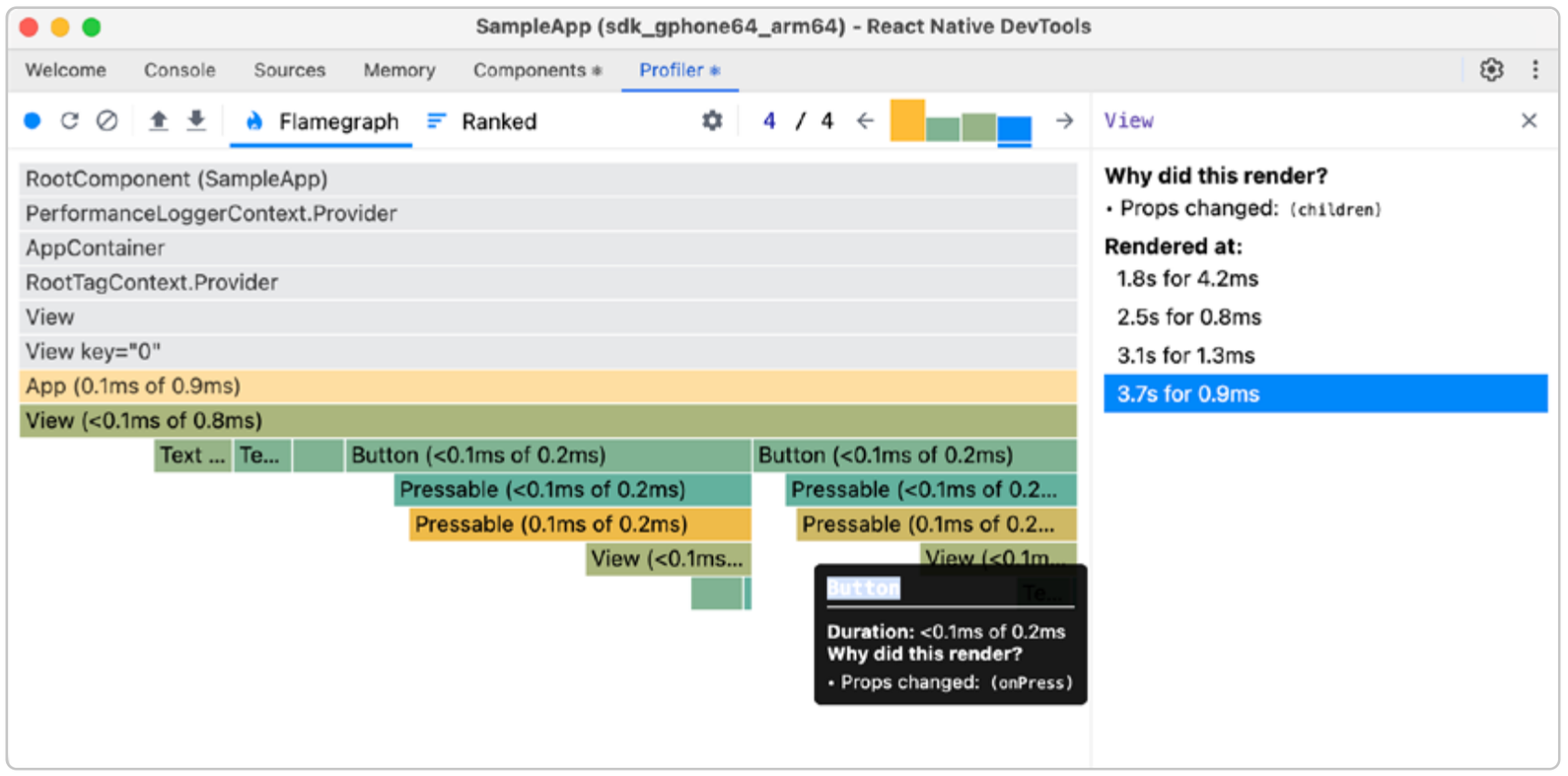
Finally, press the "Start profiling" or "Reload and start profiling" button on the top left of the screen to start profiling. It doesn't matter in our use case since we're inspecting reactions to user input. However, when profiling an app's startup, "Reload and start profiling" is a lifesaver.



"Start profiling" button.

In our sample app, click both buttons a few times and observe real-time feedback from the DevTools highlighting components that are being re-rendered. The screenshots reflect a profile produced by first pressing "Press one" twice and then "Press two" twice, producing four subsequent "commits" by the React renderer.

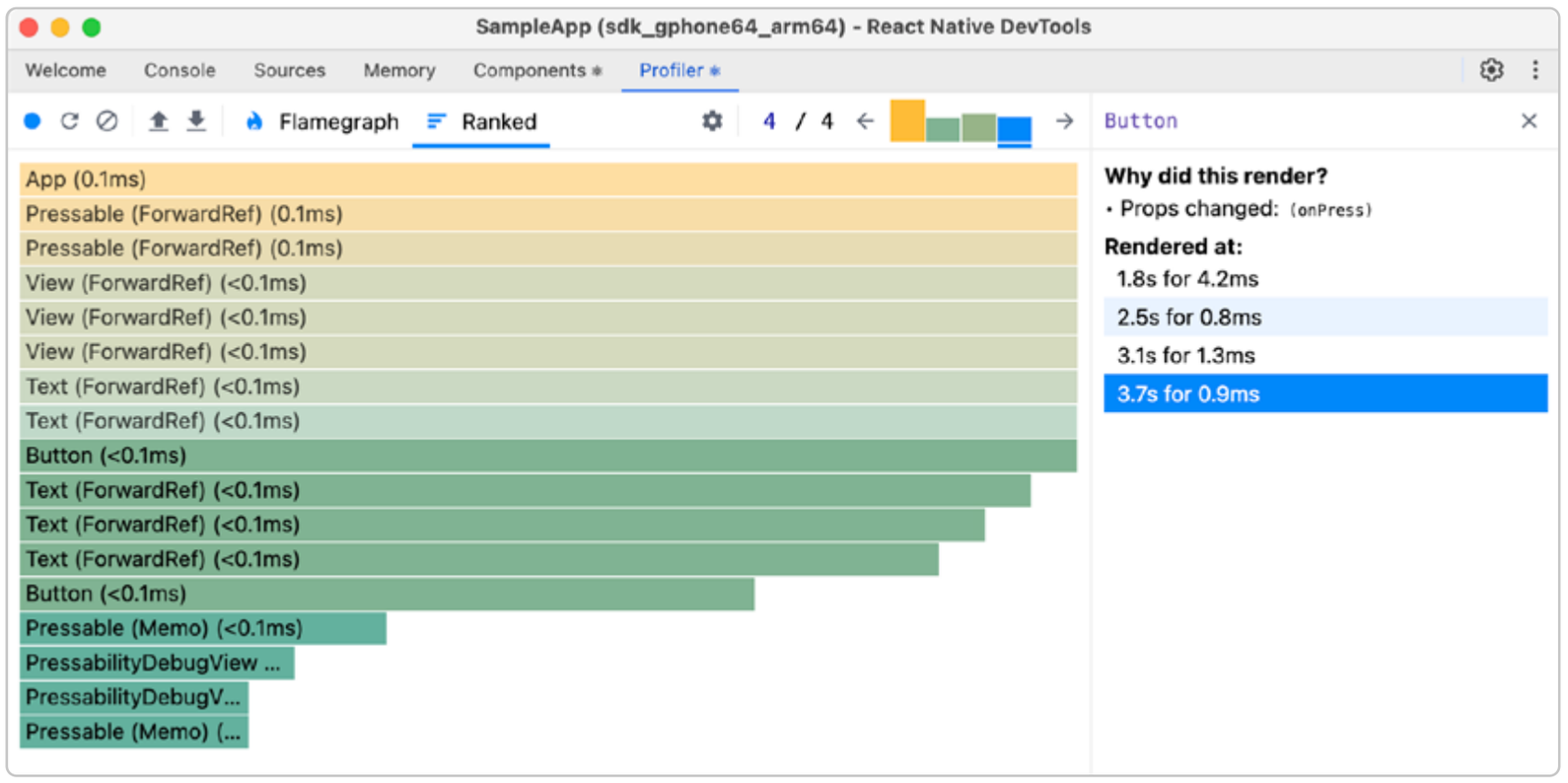
You can learn more about React's render and commit process in the official docs.



A picture of React Profiler showing a flame graph representation of React components rendered and re-rendered throughout the profiling session

You can press the green **`View`** component on the flame graph to "zoom in" to its child components. It's an essential UX paradigm implemented by the Profiler, and hopefully, you'll learn to enjoy it soon. In this case, you'll notice that both **`Button`**s are rendered in every commit, which seems redundant because no props are changing. But are they?

To get an alternative view of the situation, we can access the Ranked view, showing us the components from slowest (yellow) to fastest (green). It's pretty similar to a "Bottom-up" view you may know from other profiling tools, such as Chrome DevTools or native profilers.



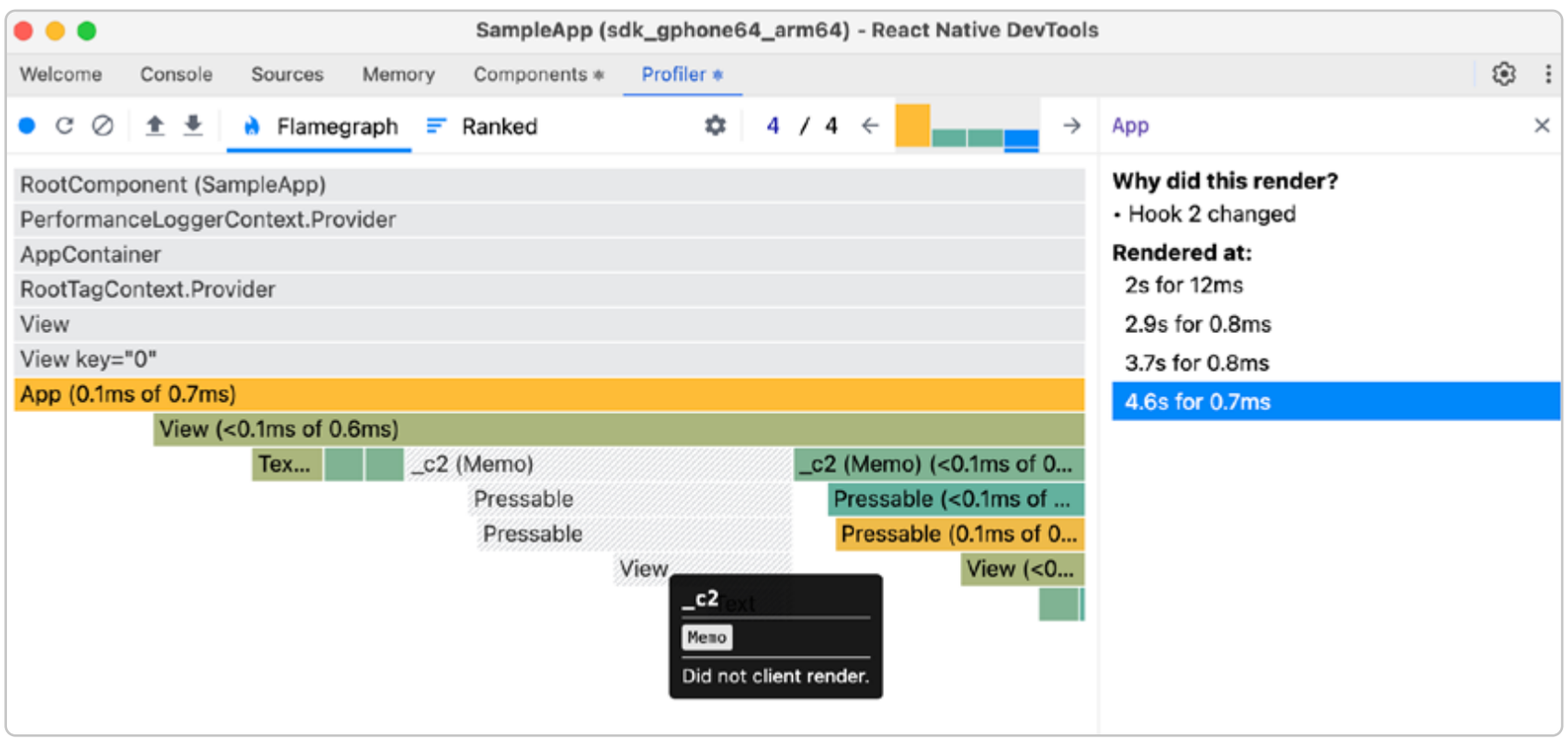
A picture of React Profiler showing a Ranked representation of React components rendered and re-rendered throughout the profiling session

In each view, we can see that a **`Button`** component is rendered twice, which reflects our two buttons getting re-rendered with every state update.

Looking at our code, this is expected—unless we're using React Compiler, which would automatically memoize this code to produce fewer re-renders. So, let's memoize it by hand.

```
// wrap inline functions with `useCallback`
const onPressHandler = useCallback(() => setCount(count + 1),
[count]);
const secondHandler = useCallback(() => setSecond(second + 1),
[second]);
// ...
<Button onPress={onPressHandler} title="Press one" />
<Button onPress={secondHandler} title="Press two" />
// wrap Button component with `memo`
const Button = memo(({onPress, title}) => {
  // ...
});
```

After wrapping **`Button`** with **`React.memo`** and its **`onPress`** handlers with **`React.useCallback`**, we get a different picture from the profiler, showing the same number of 4 React commits. However, now, only the button that's actually pressed is re-rendered, while the other one gets memoized (with auto-generated **`_c2`** name), which is indicated by the green color of this component and its children.



A flame graph of the same flow but no with one of the buttons not being updated due to memoization

Now, you're well-equipped to start your adventure with profiling any React application—whether it's a web app or mobile app. We strongly advise you to take a break from reading now and try it in the apps you develop on a daily basis. You'll notice the flame graph output will be much noisier due to the bigger complexity of your React app.

Remember to focus on the components marked with yellow as the ones where React spends the most time. And leverage the "Why did this render?" information.

Profiling JavaScript Code

React Native DevTools provide great utilities for profiling not only React but also JavaScript code. After all, it's based on Chrome DevTools, reusing its backend. In this section, we will

focus on profiling the CPU, examining what's on the call stack, and determining how you can detect long-running operations. If you want to inspect memory-related issues, you can check the How to Hunt JS Memory Leaks chapter.

After launching React Native DevTools, you can go to the "JavaScript Profiler" Tab and start recording the JavaScript CPU Profile.

If you don't see the "JavaScript Profiler" tab, navigate to settings by clicking on the gear icon at the top right, and from the experiments section, enable JavaScript Profiler.

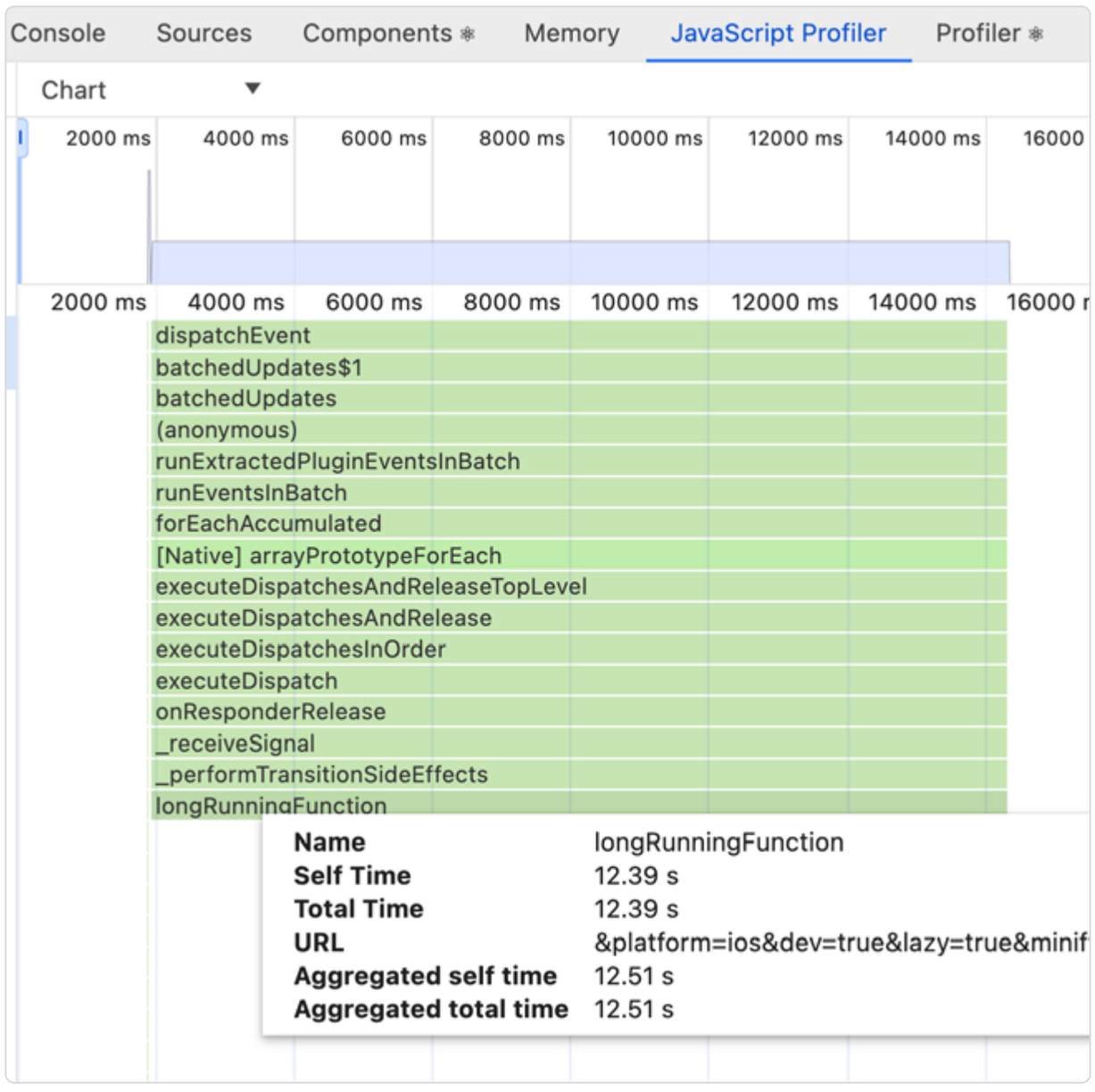
At the top left, you can select the preferred view. You can choose from *Chart, Heavy (Bottom-Up),* or *Tree*. Depending on what you want to achieve, experiment with each view.

Let's experiment with this long-running function:

```
const longRunningFunction = () => {
  let i = 0;
  while (i < 1000000000) {
    i++;
  }
};
```

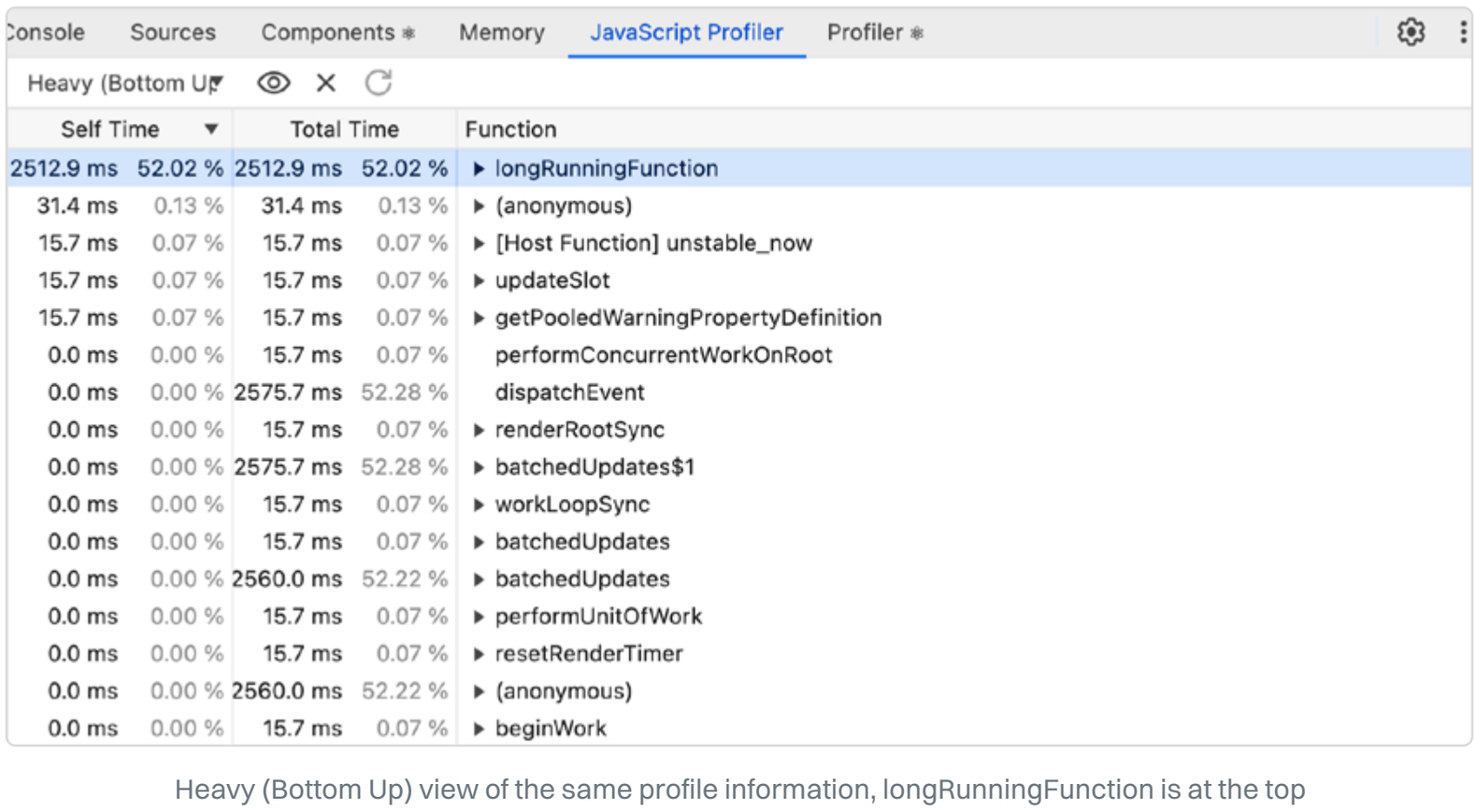
Start recording the profile and stop once it's finished. Once the profile is created, we can see that the `longRunningFunction` took over 12 seconds to run, combined with the whole call stack, which led to calling this function. This effectively makes our JS part of the app unresponsive for those 12 seconds, but the native UI part should still be responsive due to React Native's threading model.

In mobile apps, we want our functions never to block the JS thread for more than 16 ms to achieve 60 FPS and, ideally, 8 ms for 120 FPS, which is getting more popular. Read more about FPS in the How to Measure JS FPS chapter.



Profiler output showing a flame chart with long-running longRunningFunction taking over 12 seconds to run

If the Chart (or flame graph, as React DevTools calls it) information is hard to read, you can try other views, such as Heavy (Bottom Up), where you'll be able to sort by the longest or shortest running function calls.



Heavy (Bottom Up) view of the same profile information, longRunningFunction is at the top

Profiler makes it easier to spot functions that run longer than expected, tracking them down with surgical precision and realizing where a function gets called. This superpower is worth leveraging in your day-to-day work as a software engineer.

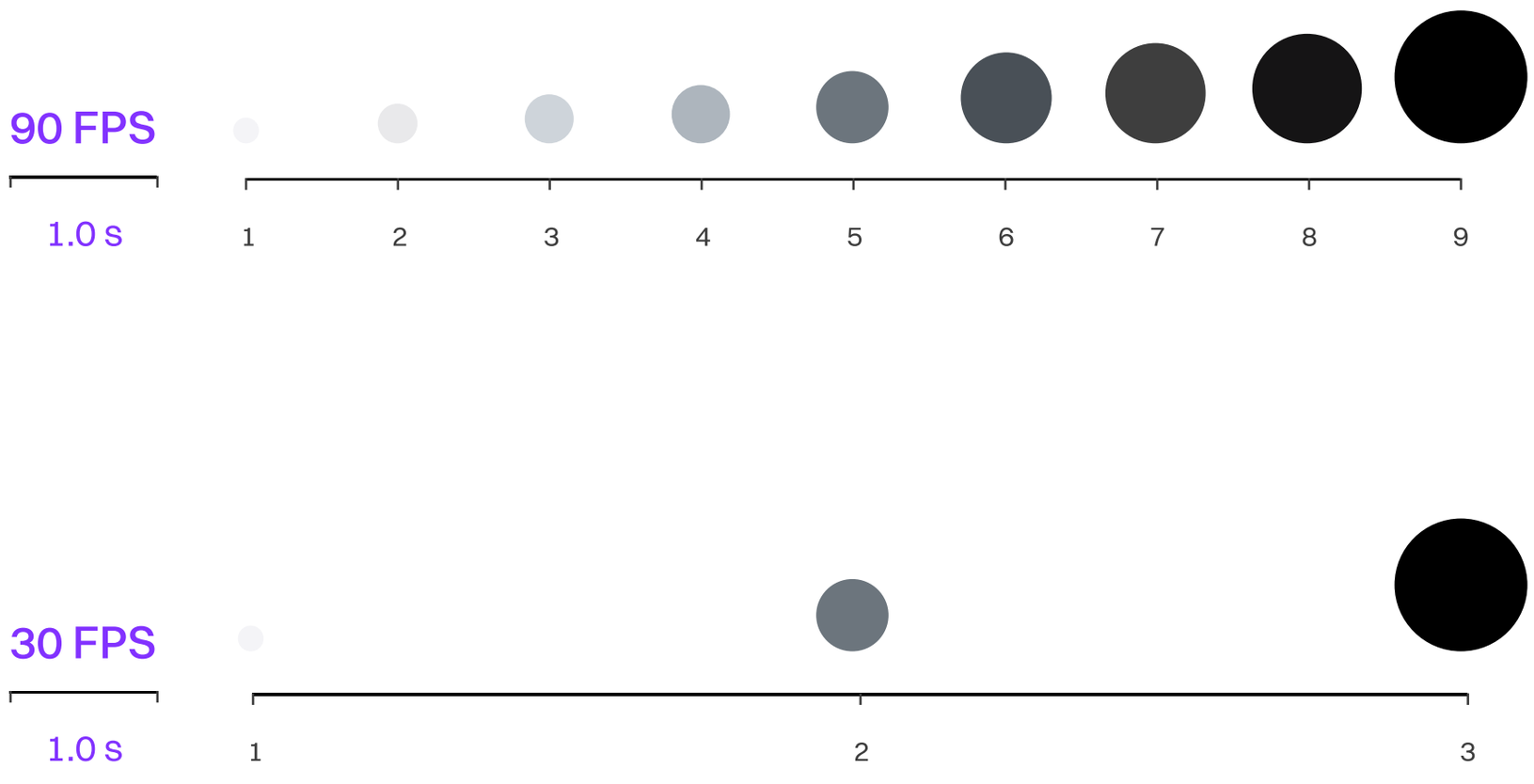
GUIDE

HOW TO MEASURE JS FPS

Mobile users expect smooth, well-designed interfaces that respond instantly and provide clear visual feedback. As a result, applications often include numerous animations that must run alongside other processes, such as data fetching and state management. If your application fails to provide a responsive interface, lags user input, or feels "janky" at times, it's a sign the UI is "dropping frames". Users hate it.

What is FPS

Getting your code to display anything on the screen involves compiling the code to an executable format, which, using various techniques, draws pixels on the screen. A single drawing is called a "frame". If you want your UI not to be static—which you do—your app will need to draw many frames every second. Most mobile devices have a refresh rate of 60 Hz, meaning they can display 60 frames per second, or 60 FPS. That's our metric.



A difference between 30 and 90 FPS

Most people perceive 60 FPS as smooth motion. However, as technology advances, users have access to screens that can refresh 120 or even 240 times a second. Once a user enjoys a 120 Hz screen, chances are they'll consider 60 Hz "choppy," "laggy," or "slow". It means that our target of 16,6 ms to render a frame (1/60 s) is moving, and we should aim for 8,3 ms per frame.

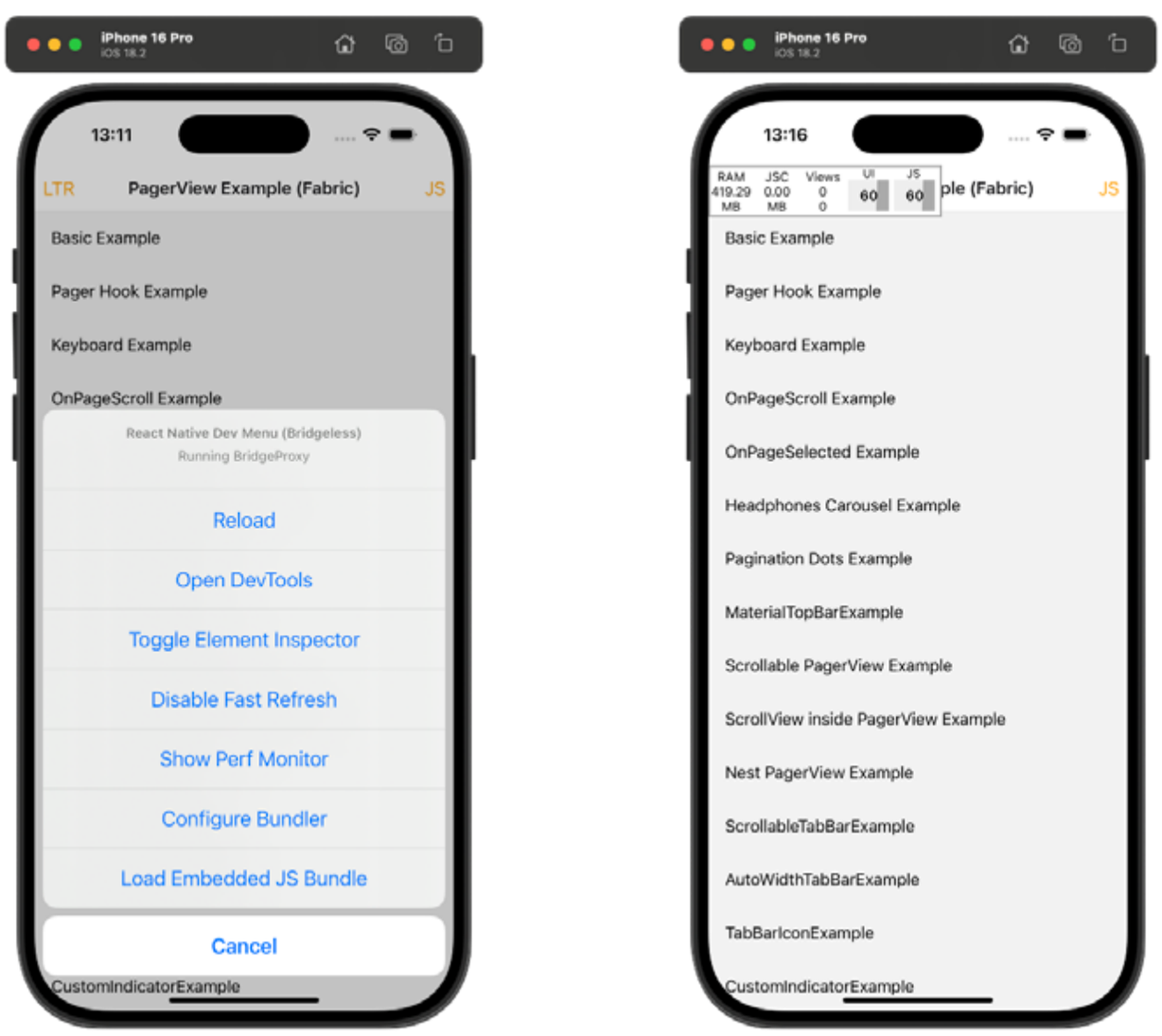
If you fail to think about your app's performance and choose the right tools to tackle this challenge, you're on the right track to dropping frames sooner or later.

React Perf Monitor

Thankfully, React Native comes with a handy tool called React Perf Monitor, which can display FPS live as an overlay on top of your app. You can open it by selecting it from the React Native Dev Menu.

You can open DevMenu by shaking the device or by using dedicated keyboard shortcuts:

- iOS Simulator: **`Ctrl`** + **`Cmd ⌘`** + **`Z`** (or *Device*> *Shake*).
- Android emulators: (**`Cmd ⌘`** / **`Ctrl`**)+ **`M`**.



React Native Dev Menu drawer

Perf Monitor visible in the top-left corner

It opens a pane in the upper left corner of the application. You can hide it with the "Hide Perf Monitor" option by accessing the React DevMenu once again.

This allows you to monitor memory usage and when the FPS slows down. There are actually two FPS monitors—one for the UI (or Main) thread and another for the JS thread. This allows you to quickly realize whether a particular interaction slowdown originates from JavaScript or native.

Always **disable** development mode when measuring performance.

On Android, it's accessible from Dev Menu in Settings > JS Dev Mode.

On iOS, you'll need to run Metro in dev mode.

As we learned in the How to Profile JS and React Code chapter, when the FPS drops during animation or some operation, you can use a profiler to narrow down the cause.

Flashlight

FPS in the Dev Menu is easy to see but hard to track and measure. It's quite convenient to get a view of, e.g., the average FPS for a particular interaction or screen load. This is where tools like Flashlight shine. It works only on Android, though.

To get started, make sure you have Flashlight installed and available in your terminal. Open up the Android application you want to measure on a given Android emulator, and then run:

```
> flashlight measure
```

You will see the diagram with various performance metrics appear in real time as you interact with the application.

Getting an average FPS is only one of the features available in a lighthouse-like report produced by the tool. The report also includes the app's overall performance score based on the metrics collected during the measurement, such as the average FPS number, average CPU, and RAM usage.

All collected measurements are saved in a JSON file that you can store and compare between different versions of your code. It's a good solution for automated local performance monitoring and to prove that the FPS has improved reliably.

GUIDE

HOW TO HUNT JS MEMORY LEAKS

When a program is executed on a target device or virtual machine, it always occupies some space in the device's Random Access Memory (RAM), which is allocated specifically for that program to execute correctly. When that memory allocation is unexpectedly retained, we say it's "leaking"; hence, we deal with memory leaks.

Memory leaks can be hard to trace without the right tools. In most cases, they arise from programmer errors. If your app constantly uses more and more memory without releasing it, you most likely have a leak somewhere. You might try to play the guessing game of what is leaking, but it's way easier to profile your app with the right tools that will help you find it. Let's start by defining what memory leaks are.

Anatomy of a memory leak in a JavaScript app

JavaScript engines, such as Hermes—and any other interpreted languages that leverage a virtual machine to execute—implement a special program called garbage collector (GC). As the name implies, it "collects garbage", where "garbage" refers to memory addresses that are no longer needed for the application to run and can be freed for other programs to use. GC periodically scans allocated objects and determines if an object is no longer needed. Fun fact: Hermes' garbage collector is called Hades.

Although garbage collectors use advanced techniques to reliably and safely determine which memory to free, in some cases, your code may sometimes trick the GC into keeping an object allocated when it shouldn't be. That's how you end up with **memory leaks** that are often hard to track without dedicated tools.

In short, a memory leak happens when a program fails to release the memory that is no longer needed.

Here are a few examples of code that leaks memory for better understanding.

1. Listeners that don't stop listening:

```
const BadEventComponent = () => {
  useEffect(() => {
    const subscription = EventEmitter.addListener('myEvent',
handleEvent);
    // Missing cleanup function
  }, []);

  return <Text>Listening for events...</Text>;
};
```

2. Timers that don't stop counting:

```
const BadTimerComponent = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      setCount(prev => prev + 1);
    }, 1000);
    // Missing cleanup function
  }, []);

  return <Text>Count: {count}</Text>;
};
```

3. Closures that retain a reference to large objects:

```
class BadClosureExample {
  private largeData: string[] = new Array(1000000).fill('some
data');

  createLeakyFunction(): () => number {
    // Entire largeData array is captured in closure
    return () => this.largeData.length;
  }
}

// Fixed
class GoodClosureExample {
  private largeData: string[] = new Array(1000000).fill('some
data');

  createEfficientFunction(): () => number {
```

```
    // Only capture the length, not the entire array
    const length = this.largeData.length;
    return () => length;
  }
}
```

Now that you know what typical memory leaks look like, let's see what tools you can use to spot them easily.

Hunting memory leaks with React Native DevTools

The new React Native DevTools builds on top of Chrome DevTools, so if you are familiar with web debugging, you should feel right at home. However, if you want to learn more about Chrome DevTools, you should check out their official documentation.

React Native DevTools offers three ways to profile your app's memory:

- **Heap snapshot**—shows memory distribution among your page's JavaScript objects and related DOM objects.
- **Allocation instrumentation on the timeline**—this profile type shows instrumented JavaScript memory allocations over time. Once a profile is recorded, you can select a time interval to see objects that were allocated within it and still live by the end of the recording. Use this profile type to isolate memory leaks.
- **Allocation sampling**—records memory allocations using the sampling method. This profile type has minimal performance overhead and can be used for long-running operations. It provides good approximations of allocations broken down by the JavaScript execution stack.

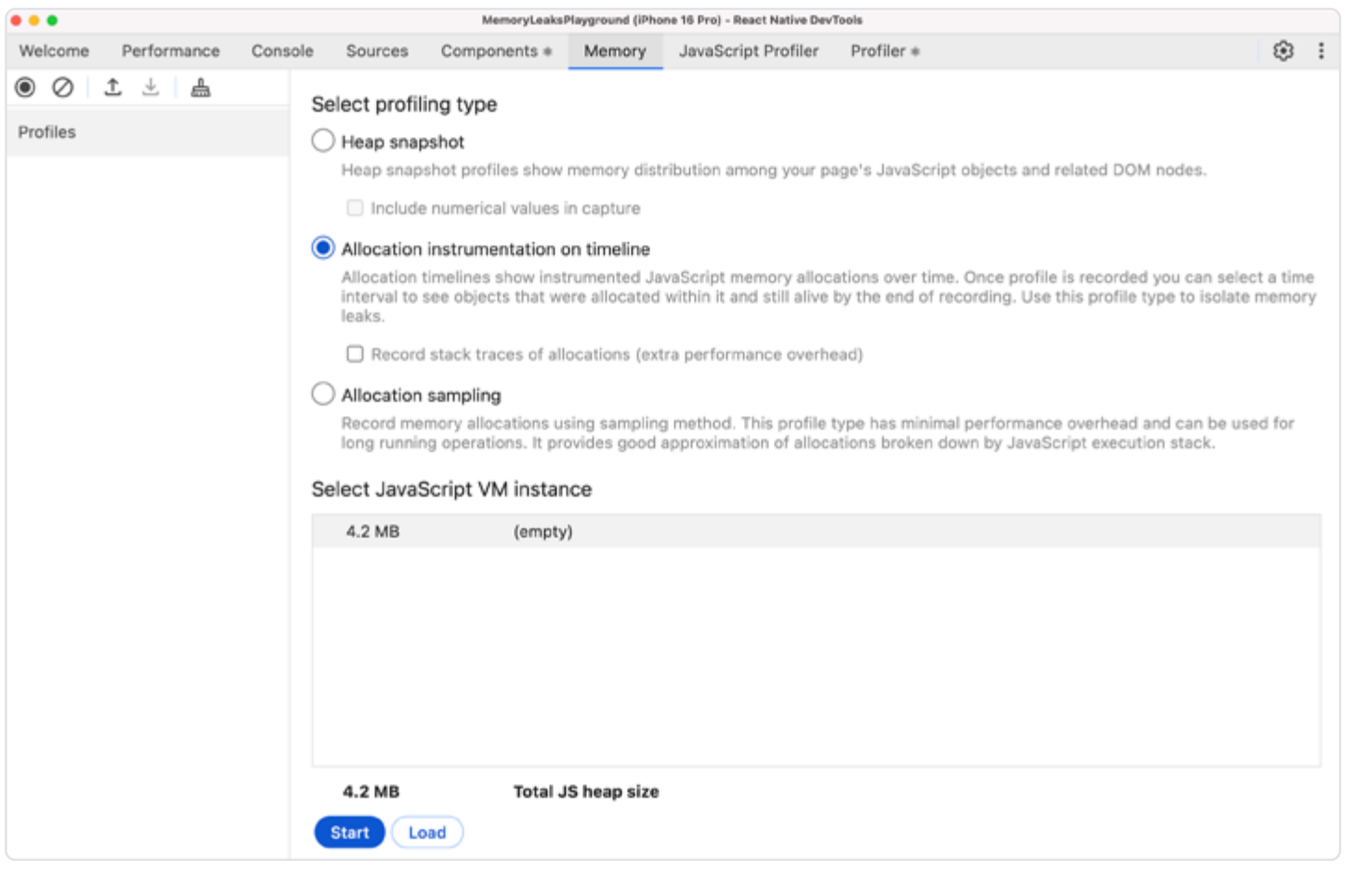
We'll focus on **Allocation instrumentation on the timeline,** which should help us find allocations that are not being freed up. Let's run it on this piece of code:

```
// Global variable to store closures
let leakyClosures: Funciton[] = [];

// Generate some dummy data
const generateLargeData = () => {
  return new Array(1000).fill(null).map(() => ({
    id: Math.random().toString(),
    data: new Array(500).fill('??').join(''),
    timestamp: new Date().toISOString(),
    nested: {
      moreData: new Array(100).fill({
        value: Math.random(),
        text: 'nested data that consumes memory',
      }),
    },
  }));
};
```

```
const createClosure = () => {
  const newDataLeak = generateLargeData();  // Creates large data
array
  return () => {
    newDataLeak.forEach((data) => data.id); // Closure captures
newDataLeak
  };
};

// Create many closures that each capture their own large data
const createManyLeaks = () => {
  for (let i = 0; i < 10; i++) {
    const leakyClosure = createClosure();
    leakyClosures.push(leakyClosure);  // Store reference to
prevent GC

    // Trigger all closures to keep data "active"
    leakyClosures.forEach(closure => closure());
  }
};
```

Code showing the third issue from the previous section (closures retaining reference to large data)

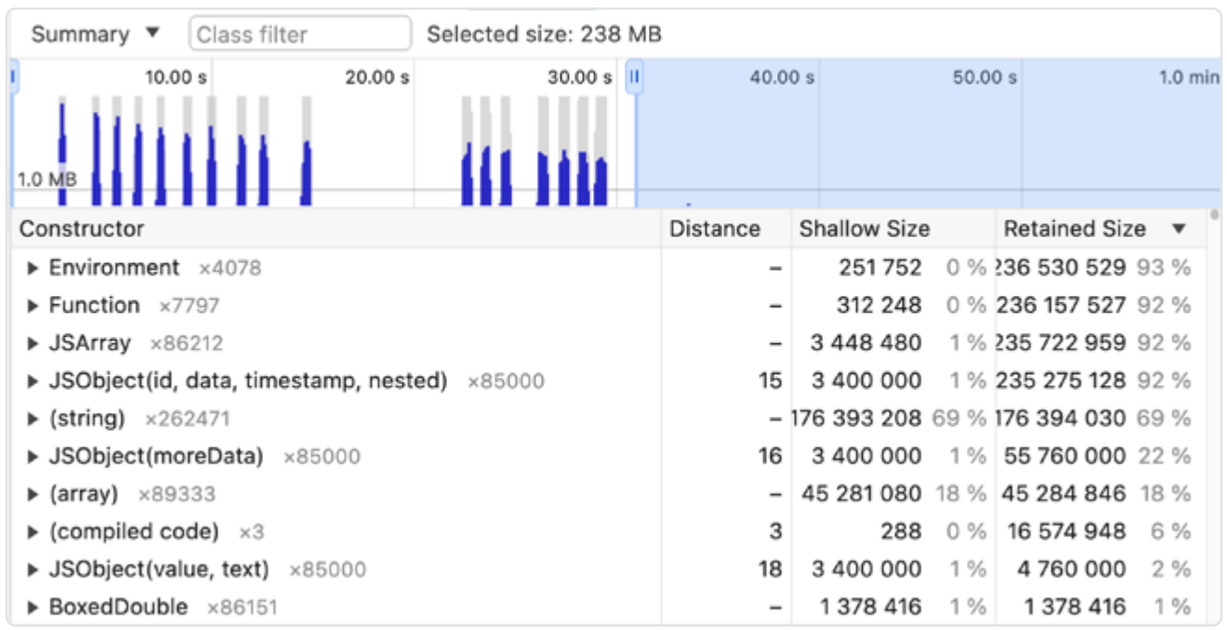
Once you launch DevTools, look for the "Memory" tab and select "Allocation instrumentation on timeline". Then, you can click "Start" at the bottom.



Memory panel in React Native DevTools

You have a leak when you finish profiling, and an object is still allocated in memory! But as mentioned before, GC runs periodically, so your objects might not get deallocated instantly. You might need to allocate something else to trigger the deallocation of existing objects.

Here is what the profiler looks like after capturing a snapshot:



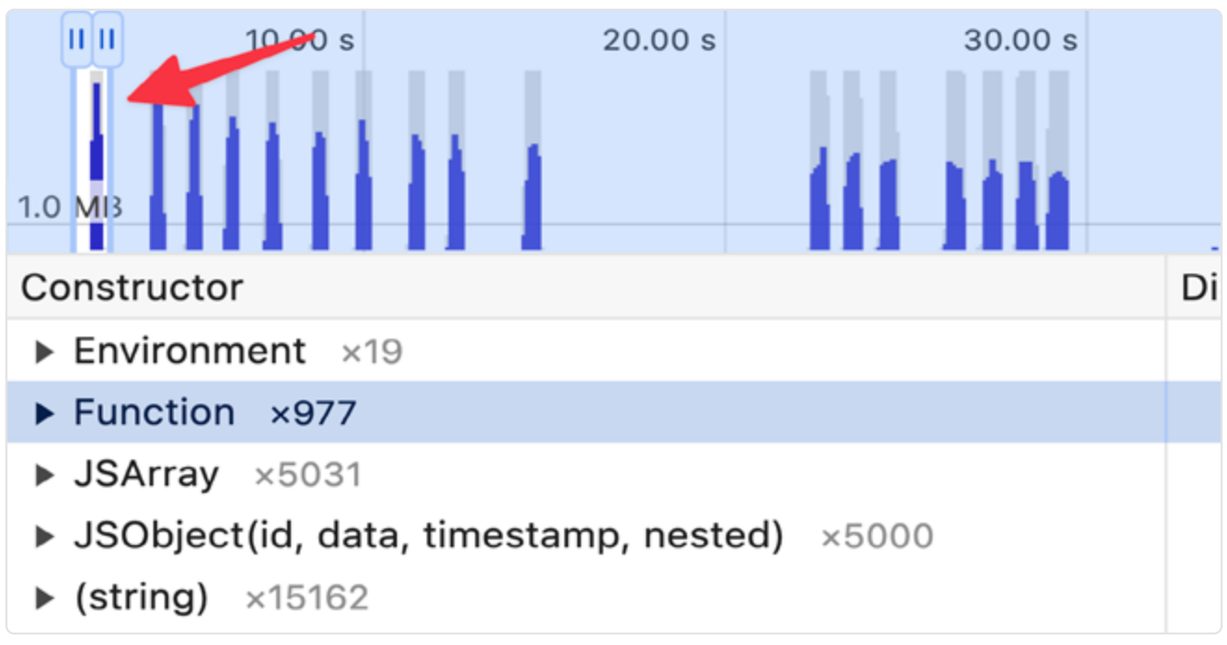
Output from the Allocation instrumentation tool showing a potential memory leak with blue bars

There are a few things to keep in mind about the Allocation instrumentation tool:

- Blue bars at the timeline represent allocations.
- If the blue bar turns grey, the object is deallocated.
- Below, we have a list of list of constructors that were called. Right next to it, there are two metrics:
  - **Shallow size**—the size of memory that is held by the object.
  - **Retained size**—the memory size that will be freed after the object is deleted.

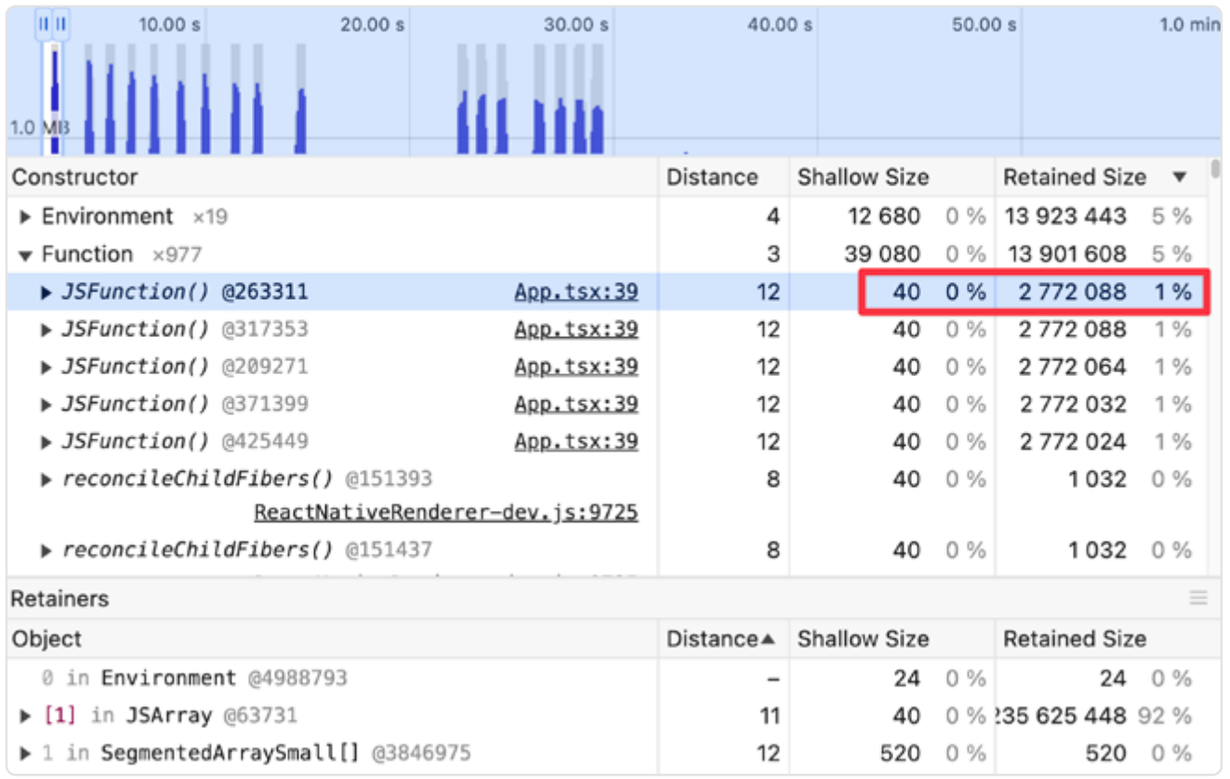
As you can see, over 30 seconds and multiple calls to the `createManyLeaks` function, GC cleared only some objects from the memory (shown on the graph as grey), but the blue parts are still alive. This graph screams that multiple leaks are happening.

Let's focus on the first spike and investigate what's causing this.



Focused view of allocation spike

Now that the graph is focused on the first spike, we can see which constructors are being called. Let's expand the `Function` constructor and see what we have there.



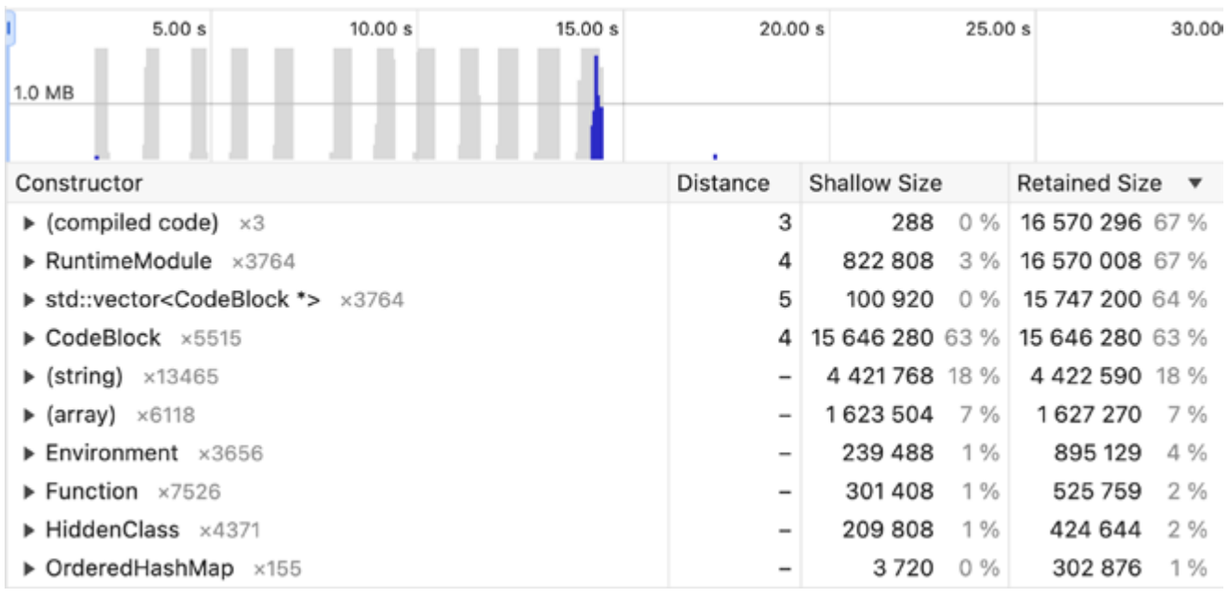
The discrepancy between shallow and retained size for JSFunction

As you can see, there is a call to the **`JSFunction()`** constructor whose shallow size is 40 bytes, but the retained size is **2 772 088 bytes(!)**. It's the leak that we were looking for! This closure is tricking GC into thinking that the objects are still relevant when they are not. DevTools even show us the exact line where this constructor was called.

After finding the leak, you should fix it and re-test. Let's alter the **`createManyLeaks`** function and comment-out pushing closures to an array to prevent retaining them.

```
const createManyLeaks = () => {
  for (let i = 0; i < 5; i++) {
    const leakyClosure = createLeak();
    leakyClosure(); // just call it
    // leakyClosures.push(leakyClosure);
    // leakyClosures.forEach(closure => closure());
  }
}
```

As you can see, all the bars are now grey (except the last one, which wasn't *yet* deallocated), which means there are no leaks!



Memory allocation profile with no leaks

Now you know how to track down and fix memory leaks! In the next chapter, we will focus on Controlled Components.

BEST PRACTICE

UNCONTROLLED COMPONENTS

The React programming model revolves around the idea of re-rendering the whole app—represented by a tree of components—every time there's a state change. However, while this model works for most use cases in UI programming, it's only an abstraction. And, as with any other abstraction, it simplifies reality so that it's easier to reason about but at the cost of correctness. A good abstraction will contain escape hatches—it's like permission from its authors to venture off of it to achieve our goals as programmers.

React is no different and offers a variety of escape hatches, such as refs, which allow bypassing Reacts re-rendering logic and open the door for components that are not driven by state updates. We call these "uncontrolled components", and it's a powerful pattern we'll explore in this chapter, especially in the context of the legacy asynchronous architecture of React Native.

Controlled TextInput in legacy architecture

Almost every React Native application you'll write will contain some form of input, such as text or voice. Let's focus on text, which is by far the most common and, in React Native apps, is represented by the `TextInput` component. This component can be either controlled—through React props—or uncontrolled, leveraging refs and callback props to communicate back to the React model.

Let's see an example of such a controlled text input first:

```
const DummyTextInput = () => {
  const [value, setValue] = useState("Text");

  const onChangeText = (text) => {
    setValue(text);
  };

  return (
    <TextInput
```

```
        onChangeText={onChangeText}
        value={value}
      />
    );
};
```

The above code sample will cause the native text input to update its value through the React model whenever a user inputs text. The speed at which input comes is often quite fast or even guided by some autocomplete. Pair that with a low-end Android device or an app doing some heavy computing while the user is typing, and your users may end up with lags and dropped frames. Even worse, in legacy asynchronous architecture, your React state may get out of sync with the native input state, causing unexpected behaviors.

The problem with controlled TextInput de-synchronization shouldn't exist in New Architecture.

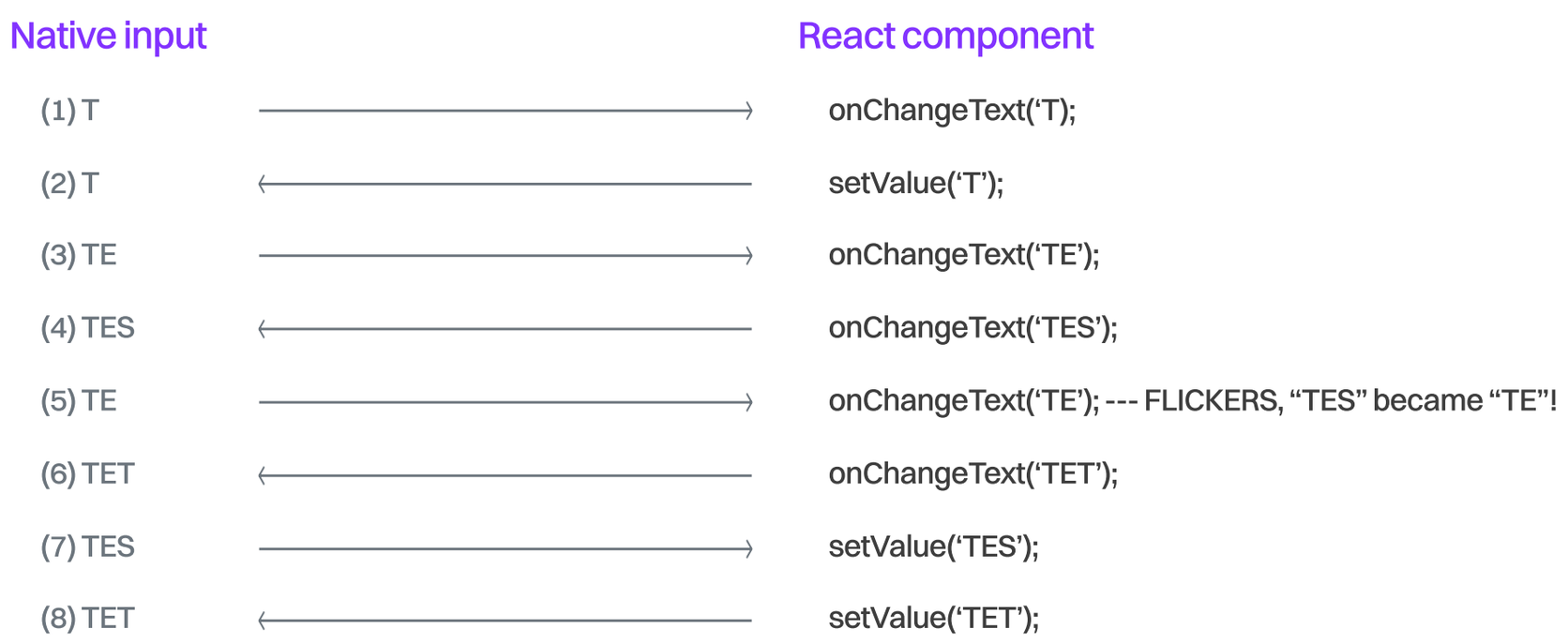
To better understand what is happening here, let's first look at the order of standard operations that occur while the user is typing and populating your **`<TextInput />`** with new characters.

A diagram showing what happens while typing TEST

As soon as the user starts inputting a new character into the native input, an update is sent to React Native via the **`onChangeText`** prop (operation 1 on the diagram above). React processes that information and updates its state accordingly by calling **`setState`**. Next, a controlled component synchronizes its JavaScript value with the native component value (operation 2 on the diagram above).

There are benefits to such an approach. React is a source of truth that dictates the value of your inputs. This technique lets you alter the user input as it happens by performing a validation, masking it, or completely modifying it.

Unfortunately, while ultimately cleaner and more compliant with the way React works, the approach mentioned above has one downside. It is most noticeable when there are limited resources available and/or the user is typing at a very high rate.



A diagram showing what happens while typing TEST too fast

When the updates via `onChangeText` arrive before React Native synchronizes each of them back, the interface starts flickering. The first update (operation 1 and operation 2) performs without issues as the user starts typing `T`.

Next, operation 3 arrives, followed by operation 4. The user typed `E` & `S` while React Native was busy doing something else, delaying the synchronization of the letter `E` (operation 5). As a result, the native input will temporarily change its value back from `TES` to `TE`.

Now, the user was typing fast enough to actually enter another character when the value of the text input was set to `TE` for a second. As a result, another update arrived (operation 6) with the value of `TET`. This wasn't intentional—the user wasn't expecting the value of its input to change from `TES` to `TE`.

Finally, operation 7 synchronized the input back to the correct input received from the user a few characters before (operation 4 informed us about `TES`). Unfortunately, it was quickly overwritten by another update (operation 8), which synchronized the value to `TET`—the final value of the input.

The root cause of this situation lies in the order of operations. Things would have run smoothly if operation 5 had been executed before operation 4. Also, if the user hadn't typed `T` when the value was `TE` instead of `TES`, the interface would have flickered, but the input value would have remained correct.

Uncontrolled TextInput

One solution for the synchronization problem is to remove the `value` prop from `TextInput` entirely and make it an uncontrolled component.

```
const DummyTextInput = () => {
  const [value, setValue] = useState("Text");

  const onChangeText = (text) => {
```

```
    setValue(text);
  };

  return (
    <TextInput
      onChangeText={onChangeText}
-     value={value}
    />
  );
};
```

As a result, the data will flow only one way, from the native **`RCTSinglelineTextInputView`** on iOS or **`AndroidTextInputNativeComponent`** on Android to the JavaScript side. Native components emit **`onChangeText`** and control the internal input state on their own, eliminating the synchronization issues described earlier.

BEST PRACTICE

HIGHER-ORDER SPECIALIZED COMPONENTS

In a React Native application, almost everything is a component. At the end of the component hierarchy are the so-called primitive components, such as `Text`, `View`, or `TextInput`. These components are implemented by React Native and provided by the platform you are targeting to support the most basic user interactions.

When building our application, we compose it out of smaller building blocks. To do so, we use primitive components. For example, to create a login screen, we would use a series of `TextInput` components to register user details and a `Pressable` component to handle user interaction.

This approach is valid from the very first component that we create within our application and holds through the final stage of its development.

In addition to the primitive components, the `react-native` package and third-party libraries come with various higher-order components designed and optimized to serve a particular purpose. Not using them can affect your application's performance, especially as you populate your state with real production data.

Displaying lists

Let's take lists as an example. Every application contains a list at some point. The default way we know from web development is to create a list of elements would be to combine `<View />` components inside a `<ScrollView />`:

```
import { View, Text, ScrollView } from 'react-native';

const NUMBER_OF_ITEMS = 10;

const App = () => {
  const items = Array.from({ length: NUMBER_OF_ITEMS }, (_, index)
```

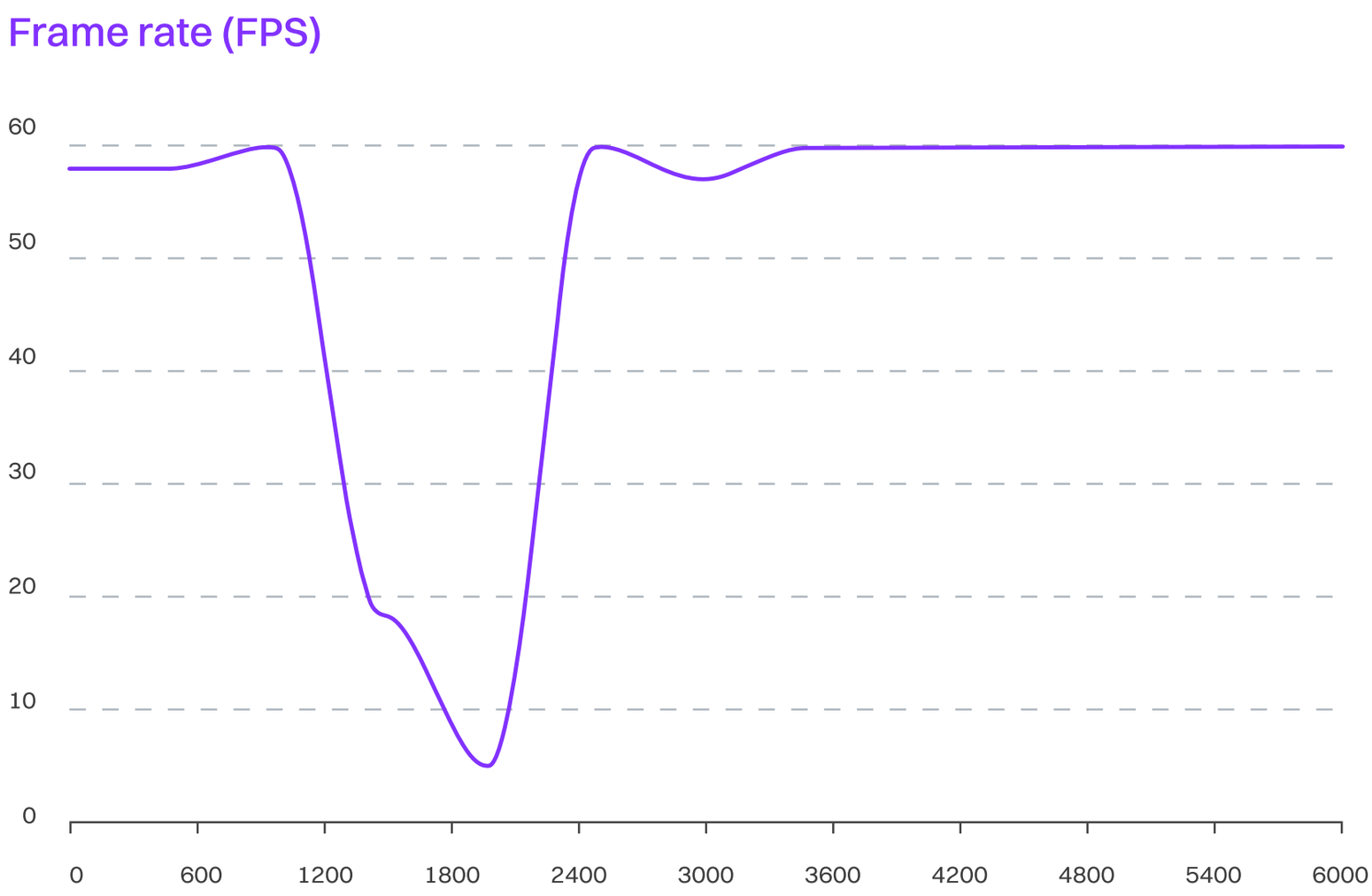
```
=> `Item ${index + 1}`);

  return (
      <ScrollView>
        {items.map((item, index) => (
          <View key={index} >
            <Text >{item}</Text>
          </View>
        ))}
      </ScrollView>
    );
};

export default App;
```

Source: https://snack.expo.dev/@callstack-snack/9374a7

Such an example, while working well in the initia stages of application development, will quickly become problematic. Let's look at what happens when the query returns 5000 items for this list:

```
- const NUMBER_OF_ITEMS = 10;
+ const NUMBER_OF_ITEMS = 5000;
```

As you can see on the graph below, performance, measured by FPS, dropped significantly to the point where the application became unresponsive for around a second.



A graph illustrating the FPS drop when initiating list using ScrollView component

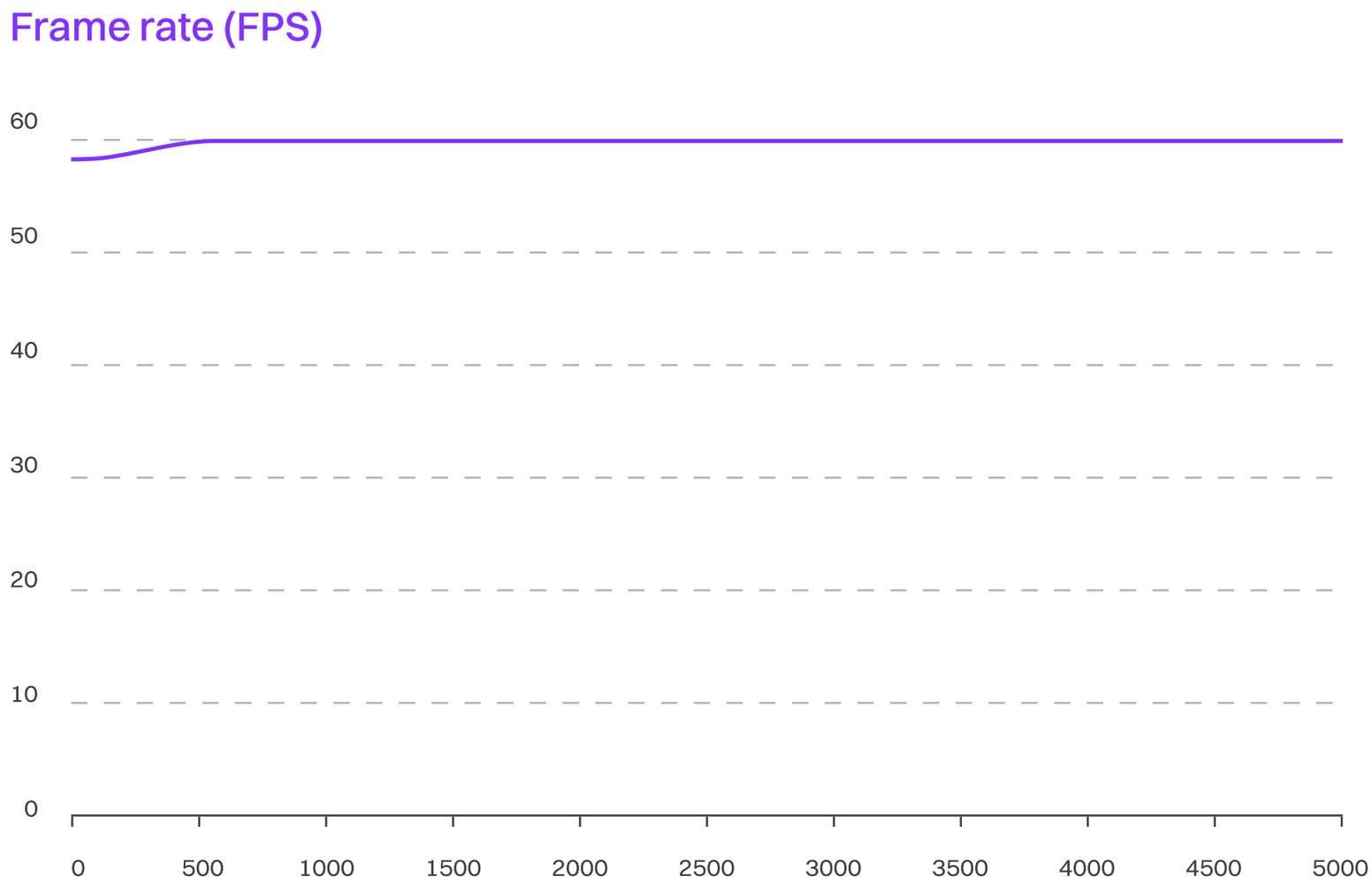
Switching to FlatList

The abovementioned issue can be quickly resolved by moving from a primitive solution to a specialized `FlatList` component. FlatList comes bundled with React Native and is designed to do one thing well: display a large number of elements in a list. Let's replace our ScrollView with a FlatList and see how it affects performance:

```
-import { View, Text, ScrollView } from 'react-native';
+import { View, Text, FlatList } from 'react-native';

const NUMBER_OF_ITEMS = 5000;

const App = () => {
  const items = Array.from({ length: NUMBER_OF_ITEMS }, (_, index)
=> `Item ${index + 1}`);

+ const renderItem = ({ item }) => (
+   <View>
+     <Text>{item}</Text>
+   </View>
+ );

  return (
-     <ScrollView>
-       {items.map((item, index) => (
-         <View key={index} >
-           <Text >{item}</Text>
-         </View>
-       ))}
-     </ScrollView>
+     <FlatList
+       data={items}
+       renderItem={renderItem}
+       keyExtractor={(item, index) => index.toString()}
+     />
  );
};

export default App;
```

Source: https://snack.expo.dev/@callstack-snack/c17e89

The example now runs without dropping frames as we scroll.



A graph illustrating the FPS performance when initiating list using FlatList component

The difference is quite stark. You might be wondering what causes such a significant performance improvement. At the end of the day, `<FlatList />` uses the same primitive components under the hood and eventually displays `View` s inside a `ScrollView`.

How FlatList is faster than a ScrollView

The key lies in the logic abstracted away within `<FlatList />` component. It contains a lot of heuristics and advanced JavaScript calculations to reduce the number of extraneous renderings that happen while you're displaying the data on screen and to make the scrolling experience always run at 60 FPS.

FlatList internally uses a `VirtualizedList`, which implements "windowin"—a techn'que that only renders and mounts items currently visible in the viewport plus a small buffer. As you scroll, it dynamically unmounts items that move out of view and mounts new ones coming into view, maintaining a finite render window of active items. This significantly reduces memory usage and ensures smooth scrolling performance in most scenarios.

Rendering complex elements in a FlatList

However, using `FlatList` alone may not be enough in some cases. Its performance optimizations rely on not rendering elements that are currently off-screen.

That said, the most costly part of the entire process are layout measurements. `FlatList` must measure your layout to determine how much space in the scroll area should be reserved for upcoming elements. For complex list elements, this may slow down the interaction, as the component will have to wait for all the items to render to measure the layout.

To mitigate this, you can implement **`getItemLayout()`** to define the element height up front, eliminating the need for measurement.

```
import { View, Text, FlatList } from 'react-native';

const NUMBER_OF_ITEMS = 10;
+ const ITEM_HEIGHT = 50; // Define a height of a list item
component or calculate it

const App = () => {
  const items = Array.from({ length: NUMBER_OF_ITEMS }, (_, index)
=> `Item ${index + 1}`);

  const renderItem = ({ item }) => (
-   <View>
+   <View style={{ height: ITEM_HEIGHT }}>
      <Text>{item}</Text>
    </View>
  );

+ const getItemLayout = (_, index) => ({
+   length: ITEM_HEIGHT,
+   offset: ITEM_HEIGHT * index,
+   index,
+ });

  return (
    <FlatList
      data={items}
      renderItem={renderItem}
      keyExtractor={(item, index) => index.toString()}
+     getItemLayout={getItemLayout}
    />
  );
};

export default App;
```

Source: https://snack.expo.dev/@callstack-snack/2f7253

It is not straightforward for items without a constant height. However, you can calculate the value based on the number of lines of text and other layout constraints.

FlashList as an alternative to FlatList

As already discussed, **`FlatList`** drastically improves the performance of a huge list compared to **`ScrollView`**. Despite being a performant solution, it has some caveats, such as rendering blank spaces while scrolling, laggy scrolling, and a list not being snappy. Also, **`FlatList`** was

designed to keep certain elements in memory, which adds overhead to the device and eventually slows the list down.

Following the tips in the official documentation can minimize the frequency of the aforementioned symptoms to some extent. Still, in most cases, we want smoother and snappier lists without extra work.

With `FlatList`, the JS thread is busy most of the time, and we always fancy having that 60 FPS tag associated with it when we're scrolling the list.

So, how should we approach such issues? Luckily, Shopify developed a pretty good drop-in replacement, `FlashList`.

```
import React from 'react';
import { View, Text } from 'react-native';
import { FlashList } from "@shopify/flash-list";

const NUMBER_OF_ITEMS = 10;
const ITEM_HEIGHT = 50;

const App = () => {
  const items = Array.from({ length: NUMBER_OF_ITEMS }, (_, index)
=> `Item ${index + 1}`);

  const renderItem = ({ item }) => (
    <View style={{ height: ITEM_HEIGHT }}>
      <Text>{item}</Text>
    </View>
  );

  return (
    <FlashList
      data={items}
      renderItem={renderItem}
      estimatedItemSize={ITEM_HEIGHT}
    />
  );
};

export default App;
```

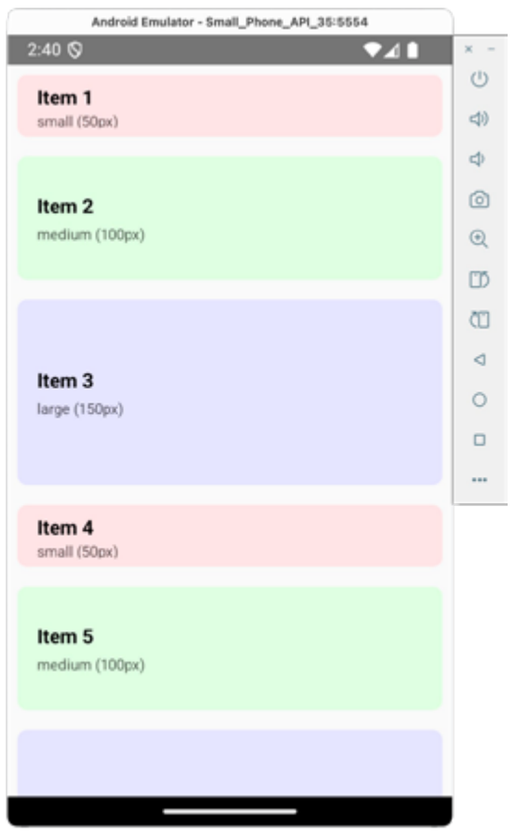
The library works on top of RecyclerListView, leveraging its recycling capability and fixing common pain points such as complicated API, using cells with dynamic heights, or first render layout inconsistencies.

`FlashList` recycles views outside of the viewport and reuses them for other items. If the list has different items, it uses a recycle pool to use the item based on its type. It's crucial to keep the list

items as light as possible, without any side effects; otherwise, it will hurt the list's performance.

Estimated item size

Aside from the `data` and `renderItem` props that you already know from working with `FlatList`, there's one additional and quite important prop: `estimatedItemSize`. It is the approximate size of the list item that is used by `FlashList` to decide how many items to render before the initial load and while scrolling.

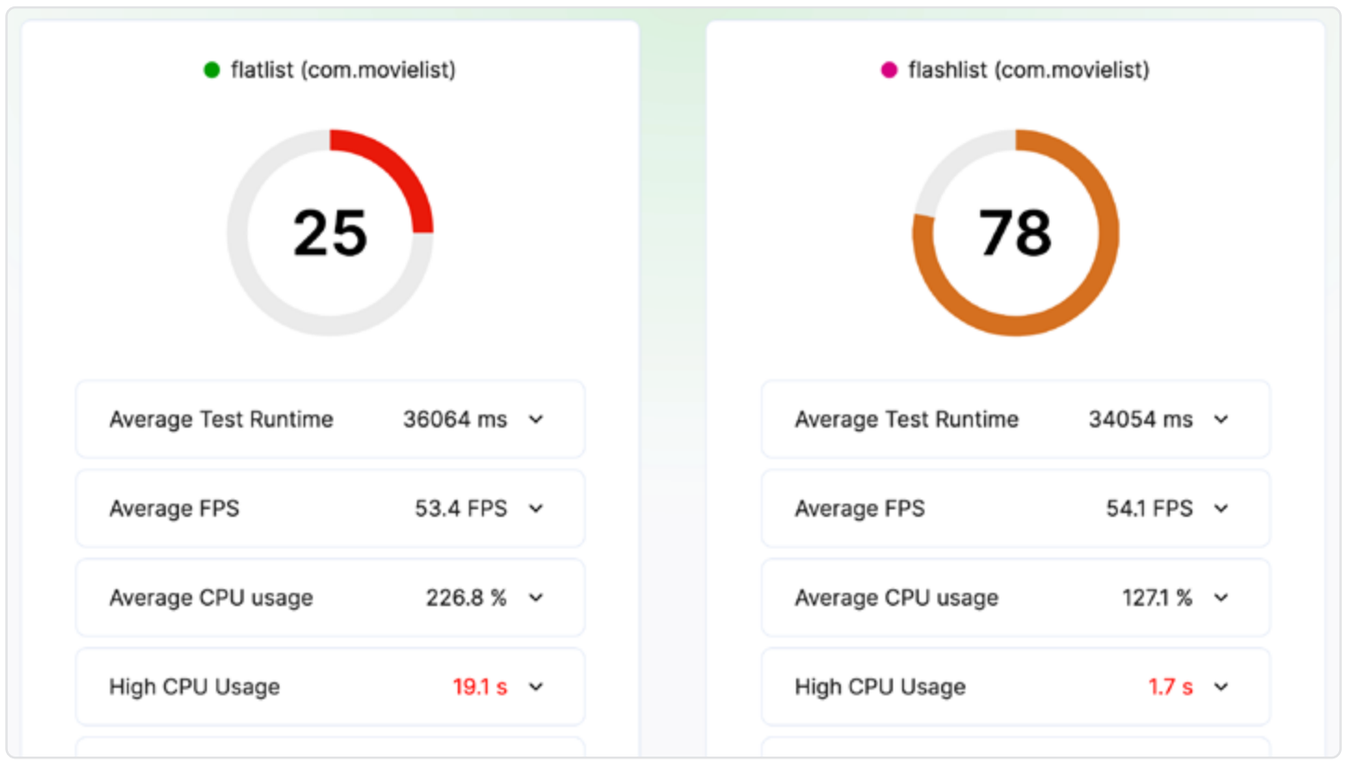


An example of differed sized-items

If your list contains items of different sizes, you can calculate the average size of the items (50px, 100px, and 150px) to get the estimated item size (50px + 100px + 150px) / 3 = 100px.

> If you do not supply this value, it will be provided by the list as a warning on the first render. It is recommended not to ignore that warning and define this prop explicitly before the list gets to your users.

How much faster is it?



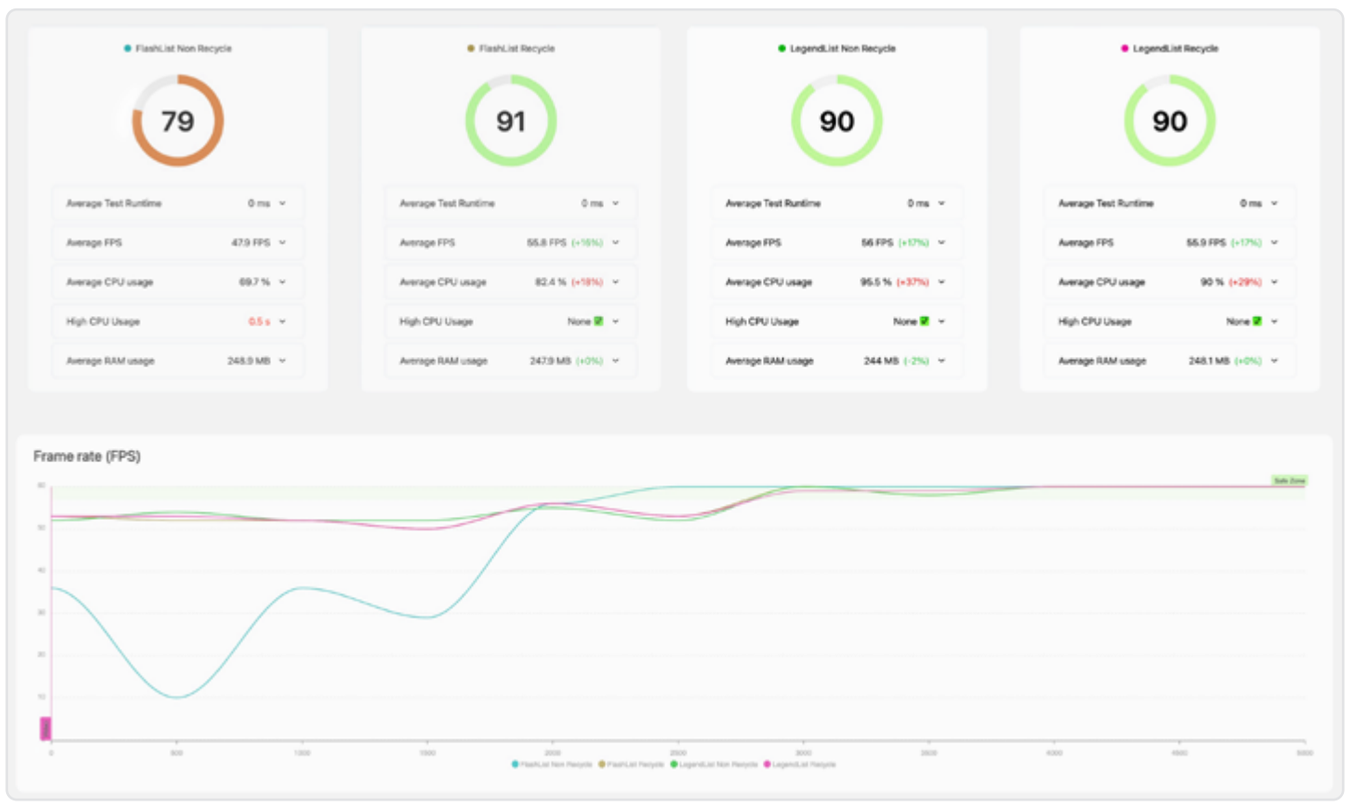
A report comparing performance when running FlatList and FlashList examples

As you can see from the report, FlashList significantly outperforms FlatList with a performance score of 78/100 vs. 25/100, demonstrating superior efficiency across all metrics, including faster runtime and lower resource consumption. The frame rate graph further reinforces FlashList's dominance, showing more stable performance at ~54 FPS. It looks like there's still some room for improvement.

You can also leverage `FlashList` FlashList's callback functions to measure performance and rendering times. The Metrics section of the documentation provides more information about available helpers.

Keep an eye on Legend List

As of early 2025, Jay Meistrich has intensively developed a new alternative to FlatList called Legend List. It is currently versioned as 1.0-beta and presents a very similar performance to FlashList. It leverages New Architecture possibilities and is implemented fully in JavaScript. While not production-ready yet, it's definitely something that the React Native community should keep an eye on.



A comparison of LegendList and FlashList performance

Using specialized components may not always yield the fastest results possible, but as shown by the examples of FlatList, FlashList, and LegendList, they're easily replaceable, giving you more options to pick from and validate. And as with any other third-party dependencies, with newer versions, you can get more optimized code for free—or not! Always measure.

BEST PRACTICE

ATOMIC STATE MANAGEMENT

A React app will re-create itself from scratch each time, based on the inputs it gets in the form of state, props, or context. Any React component will re-render itself based on these inputs and when its parent component updates. This is the promise of React that we enjoy so much as developers because it gives us predictability and certainty that our changes are always in sync with the whole app.

However, the same mechanism often causes performance issues due to unnecessary re-renders that are propagated further and further into a component tree if not carefully optimized. At least as long as we don't use React Compiler.

The higher a re-render occurs in the component tree, the more likely it is to propagate changes to leaf components unnecessarily. This is the case with the global app state, which we'll find in almost any React app. If not optimized, a change in the global store will cause re-renders in components that are not even affected by this change.

We often observe it in codebases that rely heavily on React's **`Context`** or external libraries such as Redux.

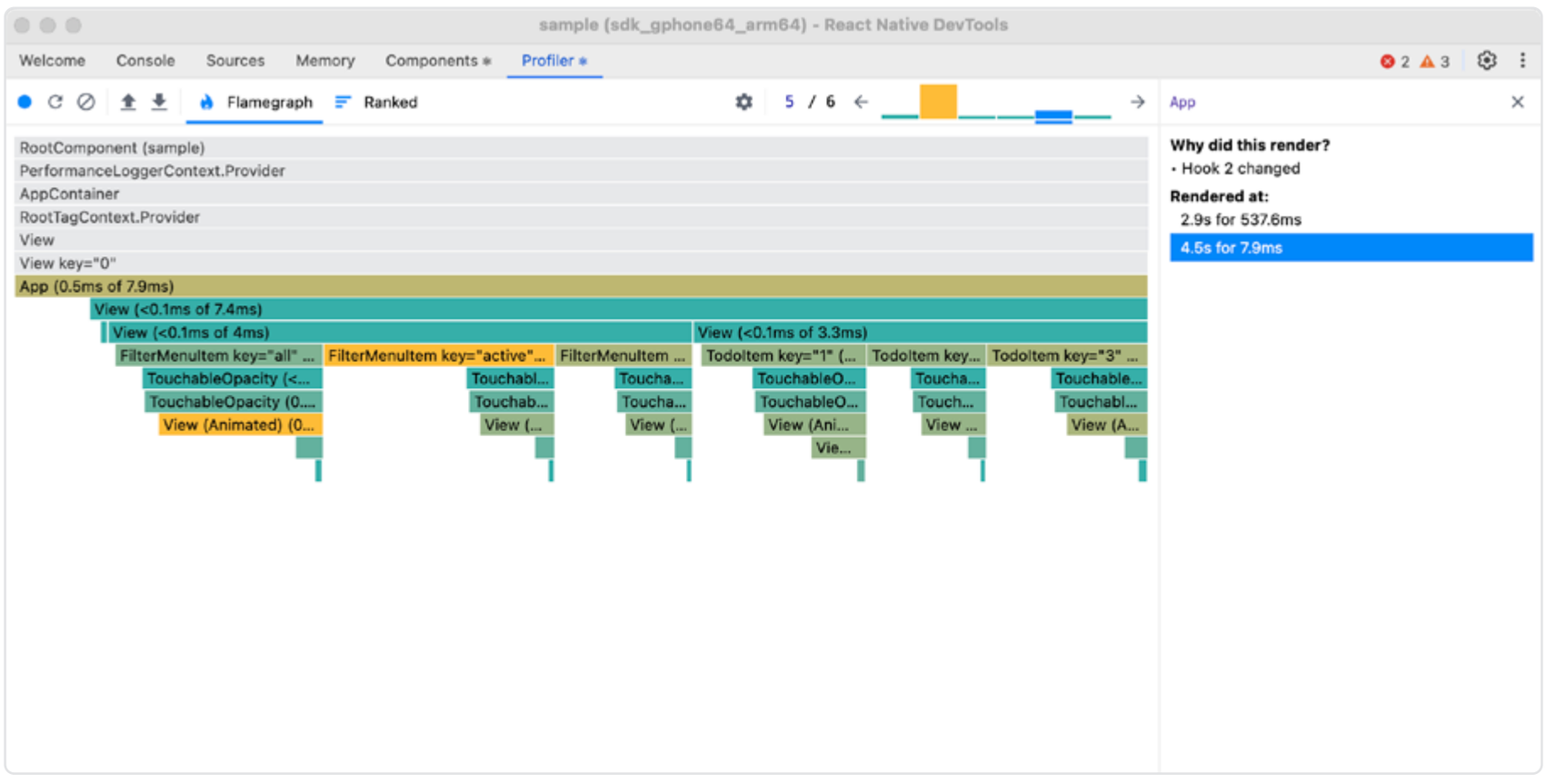
Let's examine a small code snippet illustrating an **`App`** component that holds state for **`filter`** and **`todos`**, which are used by **`FilterMenuItem`** and **`TodoItemList`** components that are not memoized.

```
import React, { useState } from 'react';
import { View, Text, TouchableOpacity } from 'react-native';

const App = () => {
  const [filter, setFilter] = useState('all');
  const [todos, setTodos] = useState(initialState);

  const filteredTodos = todos.filter(todo => {
    if (filter === 'active') return !todo.completed;
```

```
  if (filter === 'completed') return todo.completed;
  return true;
});

return (
  <View>
    {['all', 'active', 'completed'].map((filterType, index) => (
      <FilterMenuItem
        key={index}
        title={filterType}
        currentFilter={filter}
        onChange={setFilter}
      />
    ))}

    {/* Todo list section - re-renders entirely when filter
changes */}
    {filteredTodos.map((todo, index) => (
      <TodoItem key={index} item={todo} onChange={setTodos} />
    ))}
  </View>
);
};

export default App;
```

Source with runnable example available at https://snack.expo.dev/@callstack-snack/3762d2

As a result of either **`filter`** or **`todos`** state change, all components: **`App`**, **`FilterMenu`**, and **`TodoItem`** will re-render. Updating a **`TodoItem`** will cause a re-render of the **`FilterMenu`** even though it's not dependent on **`todos`**. This is not ideal, but if we examine the components in the React Native DevTools, we'll see that they're either altered by a hook change or a parent component re-render.



A flame graph presenting re-render of the whole UI tree when toggling TodoItem status

To avoid unnecessary re-renders, we can either hand-memoize them using `React.memo`, `useMemo` and `useCallback` hooks, or turn on React Compiler to do it for us automatically. Or there's a third way—we can break out from the React model and leverage atomic or signal-based state management libraries.

Atomic state management

Let's focus on the atomic state. It's a bottom-up approach where the state is broken down into small, independent units called atoms. Instead of maintaining a large centralized store, each atom represents a minimal piece of state that can be updated and subscribed to individually outside the React rendering model. This allows for more granular control and better performance through targeted re-renders.

There are many libraries that offer atomic, bottom-up state management, such as Zustand, Recoil, or Jotai. In the following example, we'll use Jotai.

Jotai means "state" in Japanese, while Zustand means "state" in German.

With Jotai, we define atoms that represent a piece of state. All you need is to specify an initial value, which can be primitive or a more complex data structure:

```
const filterAtom = atom("all");
const todosAtom = atom(initialState);
```

We can then use the `useAtom` hook to get access to the filter atom getter and setter:

```
const FilterMenuItem = ({ title, filterType }) => (
  const [filter, setFilter] = useAtom(filterAtom);

  return (
    <TouchableOpacity onPress={() => setFilter(filterType)}>
      <Text>{title}</Text>
    </TouchableOpacity>
  );
)
```

Or we can use `useSetAtom` to only access the setter for todos:

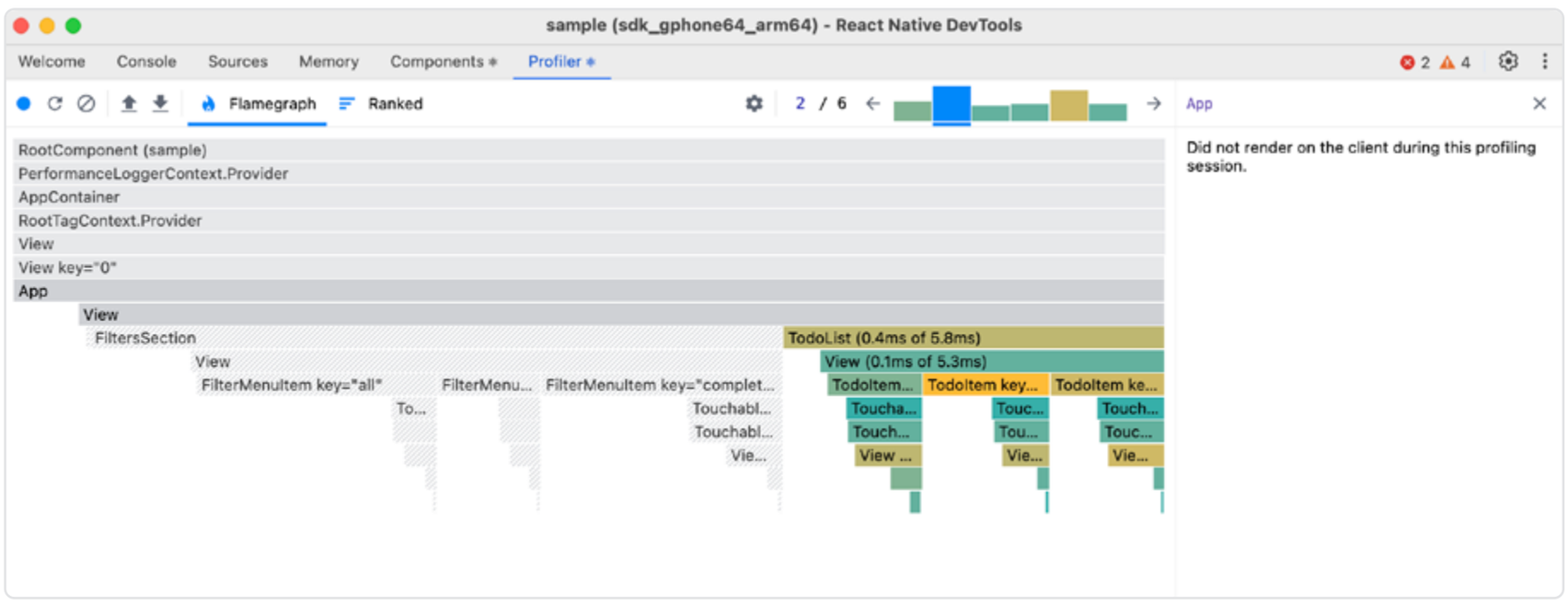
```
export const TodoItem = ({ item }) => {
  const setTodos = useSetAtom(todosAtom);

  return (
     <TouchableOpacity
      onPress={() => {
```

```
        setTodos(prev => prev.map(todo =>
          todo.id === item.id
            ? { ...todo, completed: !todo.completed }
            : todo
        ));
      }}
    >
      {/* ... */}
    </TouchableOpacity>
  );
};
```

Source with runnable example available at https://snack.expo.dev/@callstack-snack/60ed3f

Thanks to the atomic approach to state, we can prevent state changes of unrelated components without using any optimization techniques, such as memoization.

In our simple scenario, changing a filter will re-render all components that read from the filter atom: all **`FilterMenuItem`**s—since they each subscribe to the filter state—and the root component with its **`TodoItem`**s—since they depend on the filtered results. However, when toggling a todo's completion status, only components subscribed to the todos atom will re-render—the **`FilterMenuItem`**s remain unchanged since they don't depend on the todos state.



A flame graph presenting re-render of the only affected TodoList component when toggling item status

Once again, getting out of the React programming model can benefit our app's overall performance. For years, atomic state libraries helped us reduce the overall number of re-renders while leveraging React APIs such as hooks and **`useSyncExternalStore`** under the hood. Now that we have access to the React Compiler, it may be unwise to migrate your whole app to a new library for the sake of performance when the compiler can do the heavy work for us. You can read more about it in the React Compiler chapter.

BEST PRACTICE

CONCURRENT REACT

The perception of how quickly (and smoothly) apps load and respond to user interaction is sometimes more important than raw performance—especially when fetching the resources from a remote server, which tends to be the slowest part of many apps. While you may not be able to make your app run faster physically, you may be able to improve how fast it *feels* to your users. Since it's about relative perception, we call this perceived performance.

A good rule of thumb for improving perceived performance is that it's usually better to provide a quick response and regular status updates than make the user wait until an operation is fully completed. This is a core idea of Concurrent React.

As a foundational update to React's rendering model, Concurrent React was introduced in React 18 and brought to React Native in version 0.69. Instead of completing updates in a single uninterrupted process, Concurrent React allows updates to be paused, reprioritized, or even abandoned when necessary. This ensures critical interactions like user input are handled as a top priority without delay, improving overall user experience.

While concurrency is a behind-the-scenes mechanism, it unlocks several powerful features, including interruptible rendering, time-slicing, and automatic batching. These capabilities enable React Native to prepare and manage UI updates more efficiently, especially in resource-intensive scenarios like rendering large datasets or managing complex interactions.

To use Concurrent React features like `useTransition`, `useDeferredValue`, automatic batching, and `Suspense` for data fetching in React Native, you must migrate to the New Architecture, which is the default for new projects starting from React Native 0.76.

Handling slow components

In many applications, certain components are expensive to render and difficult to optimize—for example, when performing complex computations or integrating third-party libraries. Instead of allowing these heavy components to block your UI, you can use the `useDeferredValue` hook to prevent them from interfering with higher-priority tasks.

The `useDeferredValue` hook works like a buffer for your app's updates. Imagine writing a note while your friend is reading over your shoulder. If they interrupt you to comment on what you're writing, you might need to pause your note-taking momentarily. Similarly, `useDeferredValue` allows React to "pause" rendering less critical updates to ensure user interactions like typing or clicking remain smooth.

In practice, it ensures that components relying on frequently changing values don't slow down your app. Instead, React prioritizes rendering parts of the UI that matter most at a given time, like updating the display as the user types, while deferring heavier computations or updates in the background.

```
import { ActivityIndicator, Button } from "react-native";
import { useDeferredValue, useState } from "react";
import CounterNumber from "@/components/CounterNumber";
import SlowComponent from "@/components/SlowComponent";

function DeferredScreen() {
  const [count, setCount] = useState(0);
  const deferredCount = useDeferredValue(count);

  return (
    <>
      <CounterNumber count={count} />
      <SlowComponent count={deferredCount} />
      {count !== deferredCount ? <ActivityIndicator /> : null}
      <Button onPress={() => setCount(count + 1)} title="Increment"
/>
    </>
  );
}

export default DeferredScreen;
```

In the above example, the `useDeferredValue` hook ensures that updates to `SlowComponent` are deferred while keeping the `CounterNumber` responsive. The `ActivityIndicator` appears whenever the deferred value lags behind the immediate value, providing clear visual feedback to the user. This pattern is particularly useful when rendering expensive components or performing time-consuming computations.

Remember to wrap computation-heavy components you're passing deferred values to in `React.memo()` to prevent unnecessary re-renders caused by parent component re-renders.

Presenting stale value while waiting for an update

When dealing with asynchronous work such as data fetching, a pattern that occurs in every app, it's very common to show a loading state while waiting for an operation to finish. However, this can lead to a jarring user experience where content disappears and is replaced by loading indicators frequently.

The `useDeferredValue` hook offers a more sophisticated approach by allowing you to show stale value while the update is not done yet.

```
import { useState, useDeferredValue, Suspense } from 'react';
import { View, TextInput } from 'react-native';
import SearchResults from './SearchResults';

function SearchScreen() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);

  return (
    <View>
      <TextInput
        value={query}
        onChangeText={setQuery}
        placeholder="Search items..."
      />
      <Suspense fallback={<LoadingSpinner />}>
        <SearchResults query={deferredQuery} />
      </Suspense>
    </View>
  );
}
```

During the initial render, the `query` and `deferredQuery` values are equal. When the user starts typing inside `TextInput`, the `query` property updates immediately, but the `deferredQuery` keeps a stale value until the update (search operation) is done.

You can improve visual feedback to better present this experience by introducing a property that will keep the value if the search results in this example are stale:

```
import { useState, useDeferredValue, Suspense } from 'react';
 import { View, TextInput } from 'react-native';
 import SearchResults from './SearchResults';
```

```
  function SearchScreen() {
    const [query, setQuery] = useState('');
    const deferredQuery = useDeferredValue(query);
+   const isStale = query !== deferredQuery;

    return (
      <View>
        <TextInput
          value={query}
          onChangeText={setQuery}
          placeholder="Search items..."
        />
        <Suspense fallback={<LoadingSpinner />}>
-         <SearchResults query={deferredQuery} />
+         <View style={[
+           isStale && { opacity: 0.8 }
+         ]}>
+           <SearchResults query={deferredQuery} />
+         </View>
        </Suspense>
      </View>
    );
  }
```

When we wait for an update, the search results will have decreased opacity. Once the update finishes, fresh results will be presented, and the **`defferedQuery`** will be the same as a **`query`**.

Using transitions for non-critical updates

Transitions allow developers to handle non-critical UI updates, such as secondary state changes, by marking them as low-priority. This ensures that urgent interactions, like user input, are processed first. Unlike **`useDeferredValue`**, which defers the rendering of a specific value, **`useTransition`** defers the entire update process, ensuring smoother management of state changes.

```
import { Button, SafeAreaView, StyleSheet } from "react-native";
import { useState, useTransition } from "react";
import CounterNumber from "@/components/CounterNumber";
import SlowComponent from "@/components/SlowComponent";

function TransitionScreen() {
  const [count, setCount] = useState(0);
  const [slowCount, setSlowCount] = useState(0);
  const [isPending, startTransition] = useTransition();

  const handleIncrement = () => {
    setCount((prevCount) => prevCount + 1);
```

```
  startTransition(() => {
    setSlowCount((prevSlowCount) => prevSlowCount + 1);
  });
};

return (
  <SafeAreaView style={styles.container}>
    <CounterNumber count={count} />
    <SlowComponent count={slowCount} />
    {isPending ? <LoadingSpinner /> : null}
    <Button onPress={handleIncrement} title="Increment" />
  </SafeAreaView>
);
}

export default TransitionScreen;
```

By separating critical and non-critical updates, the UI remains responsive and delivers a smooth user experience. The `useTransition` hook handles non-urgent updates like the `slowCount` state, ensuring critical updates like the `count` state for `CounterNumber` are processed instantly while the deferred updates are performed in the background.

How to choose between `useDeferredValue` and `useTransition`

`useDeferredValue` is best used for deferring the rendering of a single value. This ensures that components relying on that value don't block critical updates. For example, when rendering a computationally heavy component based on frequently changing input, `useDeferredValue` ensures the rest of the UI remains responsive.

On the other hand, `useTransition` defers entire updates or rendering tasks by marking them as low-priority. It's most effective when multiple states or components need to be updated in the background, such as when transitioning between views or updating a broader section of the UI. In React Native apps, view transitions are usually handled by some kind of native navigation, such as React Navigation with a native stack navigator. However, some navigators are JS-based and could be subject to this optimization technique.

To decide between the two, consider the scope of the updates. If you're managing a single prop or value, `useDeferredValue` is likely the better choice. If you're managing transitions that involve multiple updates or broader UI changes, `useTransition` is more appropriate.

Automatic batching

React 18 introduced automatic batching, a process in which React groups multiple state updates into a single re-render for better performance. Previously, React only batched updates inside React event handlers. Updates inside Promises, `setTimeout`, native event handlers, or any other place were not batched by default by React.

```
// Before: only React events were batched.
setTimeout(() => {
  setProductQuantity(quantity => quantity + 1);
  setIsCartOpen(isOpen => !isOpen);
  // React will render twice, once for each state update (no
batching)
}, 1000);

// After: updates inside of timeouts, promises etc.
setTimeout(() => {
  setProductQuantity(quantity => quantity + 1);
  setIsCartOpen(isOpen => !isOpen);
  // React will only re-render once at the end (thanks to
batching!)
}, 1000);
```

Automatic batching also prevents your component from being in a "half-finished" state, where only one state was updated, which may cause unexpected visual glitches.

If you want to learn more about this topic, we highly recommend reading this page inside the official React documentation.

Concurrent React features, including `useDeferredValue`, `useTransition`, and automatic batching, enable better-perceived performance by intelligently managing rendering tasks and state updates without blocking the UI. These powerful capabilities can be incrementally adopted in your apps, allowing you to enhance performance and responsiveness where needed most, especially in resource-intensive scenarios.

BEST PRACTICE

REACT COMPILER

As you already know, many performance issues in React Native apps come from React components being re-rendered more often than necessary. To tackle this over-rendering, developers can employ various memorization techniques or break out of the React rendering model for certain use cases, such as global state management. The problem with hand-written memoization is that it makes code harder to read and reason about. There has to be a better way, some smart program that would memoize our functions and components automatically. And there is one—it's React Compiler.

React Compiler is a new tool from the React core team, designed to automatically optimize React applications at build time. It analyzes component structures and applies memoization techniques to reduce unnecessary re-renders. Unlike manual optimizations using **`React.memo`**, **`useMemo`**, or **`useCallback`**, the compiler automates this process, making it easier to achieve optimal performance without additional effort from developers.

At the time of writing this Guide (January 2025), React Compiler is still in beta. Companies like Meta are already using it in production, but whether you can use it depends on how well your code follows the Rules of React. If you're eager to try it out, you can install the beta version (tagged as **`beta`** on npm) or experiment with daily builds (tagged as **`experimental`**).

Preparing your codebase for the React Compiler

Before adding React Compiler to your codebase, it's good to prepare it for the new tool by installing an unobtrusive ESLint plugin. The React Compiler ESLint plugin is a helpful tool that detects potential issues in real time, flags violations of the Rules of React, and warns about optimization blockers.

Even if you're not using the compiler yet, enabling this plugin improves code quality and ensures a smoother transition when you decide to adopt it.

React Compiler is compatible with React 17+ applications and libraries. However, its effectiveness depends on adherence to React's best practices. It does not optimize class components, outdated patterns, or components that break the Rules of React.

To install the plugin, add `eslint-plugin-react-compiler` from the `beta` tag to your dev dependencies:

```
> npm install -D eslint-plugin-react-compiler@beta
```

Then, configure ESLint to enforce compiler rules:

```
import reactCompiler from 'eslint-plugin-react-compiler'

export default [
  {
    plugins: {
      'react-compiler': reactCompiler,
    },
    rules: {
      'react-compiler/react-compiler': 'error',
    },
  },
]
```

eslint.config.js

And you're ready to fix any violations of the Rules of React in your codebase.

Running the compiler

Now that we're all set with the linter, it's time to address the elephant in the room: the Babel plugin that holds the compiler:

```
> npm install -D babel-plugin-react-compiler@beta
```

Configure it in your Babel config in the following way:

```
const ReactCompilerConfig = {
```

```
  target: '19' // <- pick your 'react' version
};

module.exports = function () {
  return {
    plugins: [
      ['babel-plugin-react-compiler', ReactCompilerConfig],
    ],
  };
};
```

babel.config.js

If you want to use React Compiler on a project that's lower than React Native 0.78 (which ships with React 19), you'll need to add one extra package: **`react-compiler-runtime@beta`** and configure the **`target`** to "**`18`**" accordingly in **`ReactCompilerConfig`**. Now you're all set! Or almost set.

You will likely notice that compiler may fail on certain files. For such occasions, the Babel plugin allows for some level of customization with **`sources`** config, where you can skip some components or even whole directories:

```
const ReactCompilerConfig = {
  sources: (filename) => {
    return filename.indexOf('src/path/to/dir') !== -1;
  },
};
```

babel.config.js

This way you, can add React Compiler incrementally into your project, lowering the risk of regressions caused by bugs in beta software. Let's see how the compiler affects our source code with a short example of a **`TextInput`** with **`onChangeText`** callback:

```
export default function MyApp() {
  const [value, setValue] = useState('');
  return (
    <TextInput
      onChangeText={() => {
        setValue(value);
      }}>
      Hello World
    </TextInput>
  );
}
```

Source code before running compiler

```
import { c as _c } from 'react/compiler-runtime';
export default function MyApp() {
  const $ = _c(2);
  const [value, setValue] = useState('');
  let t0;
  if ($[0] !== value) {
    t0 = <TextInput onChangeText={() => setValue(value)}>Hello
World</TextInput>;
    $[0] = value;
    $[1] = t0;
  } else {
    t0 = $[1];
  }
  return t0;
}
```

After running compiler: more code, less re-renders

Notice how React Compiler transformed our code. It imports a `c` function from the `react/compiler-runtime` and uses it to determine what to render based on state. The `c(n)` function is a polyfill of `useMemoCache(n)` from React 19, allowing the same behavior in earlier versions of React. It creates an array that persists across renders, similar to `useRef`. If we were to implement it using useRef, it would look like this:

```
function useMemoCache(n) {
  const ref = useRef(Array(n).fill(undefined));
  return ref.current;
}
```

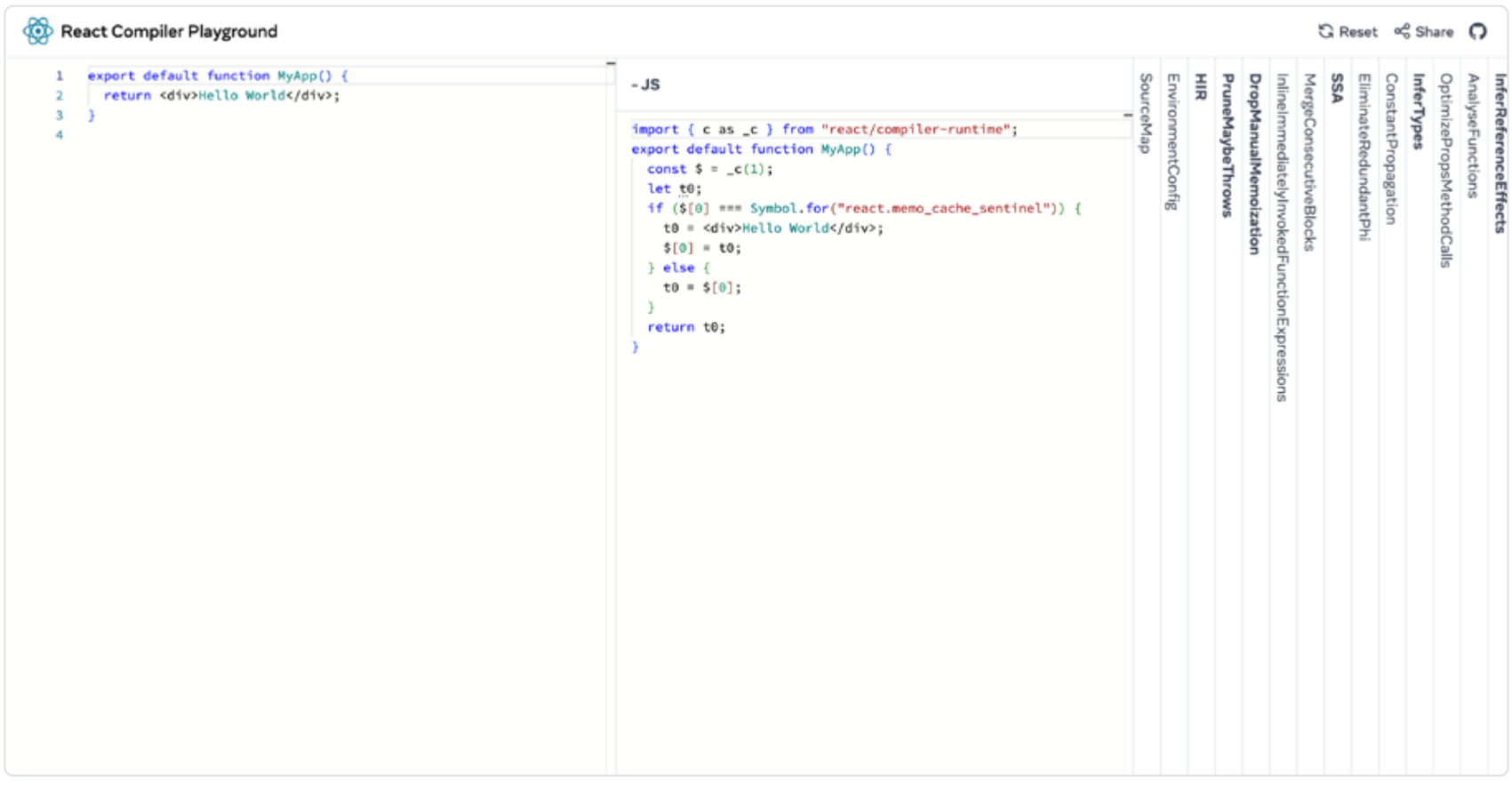
`useMemoCache` creates an array with `n` slots that persist across renders

In the example above, `const $ = _c(2)` works like `useMemoCache(2)`, returning an array with two slots. `$[0]` holds the previous `value`, while `$[1]` stores the last rendered `TextInput`. When `value` changes, a new `TextInput` is stored in `$[1]`. If `value` remains the same, React reuses the cached component instead of re-rendering it.

React Compiler uses **shallow comparison**, just like `React.memo` and `useMemo`, so be careful when using objects or arrays as props. If their reference changes, they will be treated as new values.

React Compiler Playground

If you want to understand how the React Compiler transforms your code, the React Compiler Playground is your friend. It allows you to inspect how the compiler optimizes components, test different structures, and debug compiler output. This interactive environment helps you see what changes the compiler makes and identify any unexpected behaviors in your components.



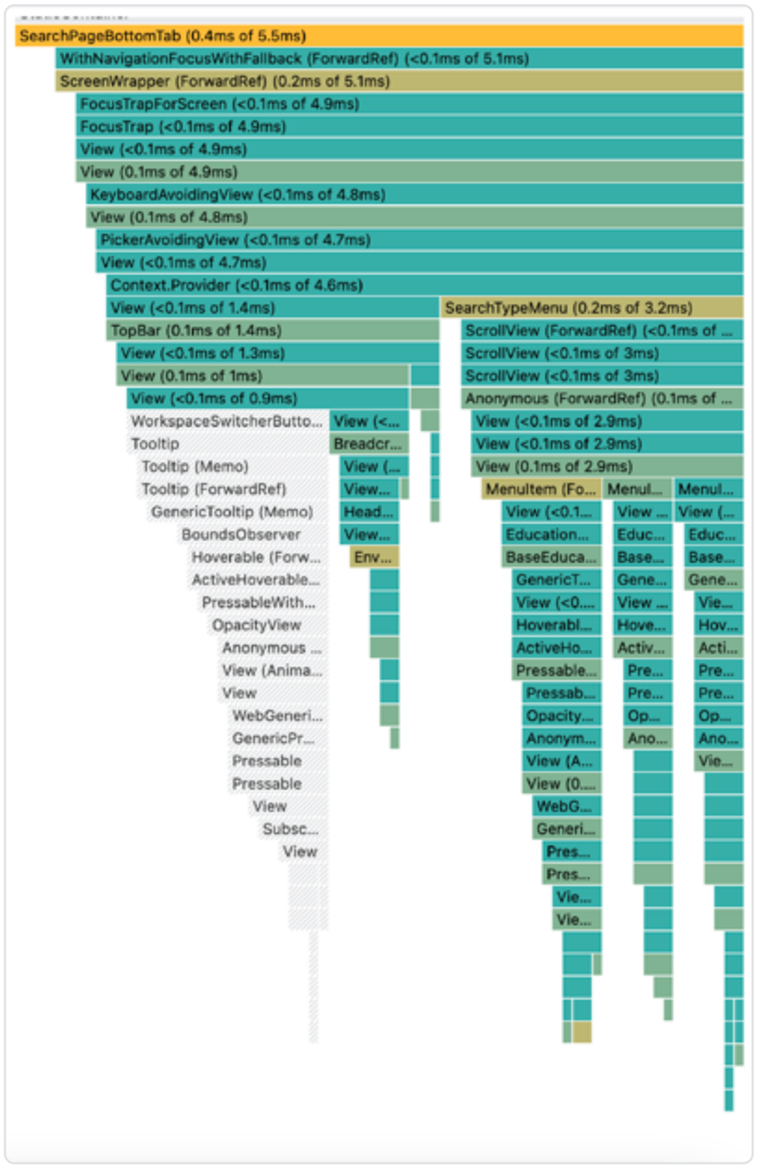
Is it time to remove manual memoization?

Not yet. While the React Compiler automates memoization, it does not fully replace `React.memo`, `useMemo`, or `useCallback` in all cases. If your code heavily relies on these optimizations, you should keep them in place until the compiler reaches a stable release and official recommendations from the React team confirm that manual memoization is no longer needed. We expect the linter to guide you on which optimizations are unnecessary.

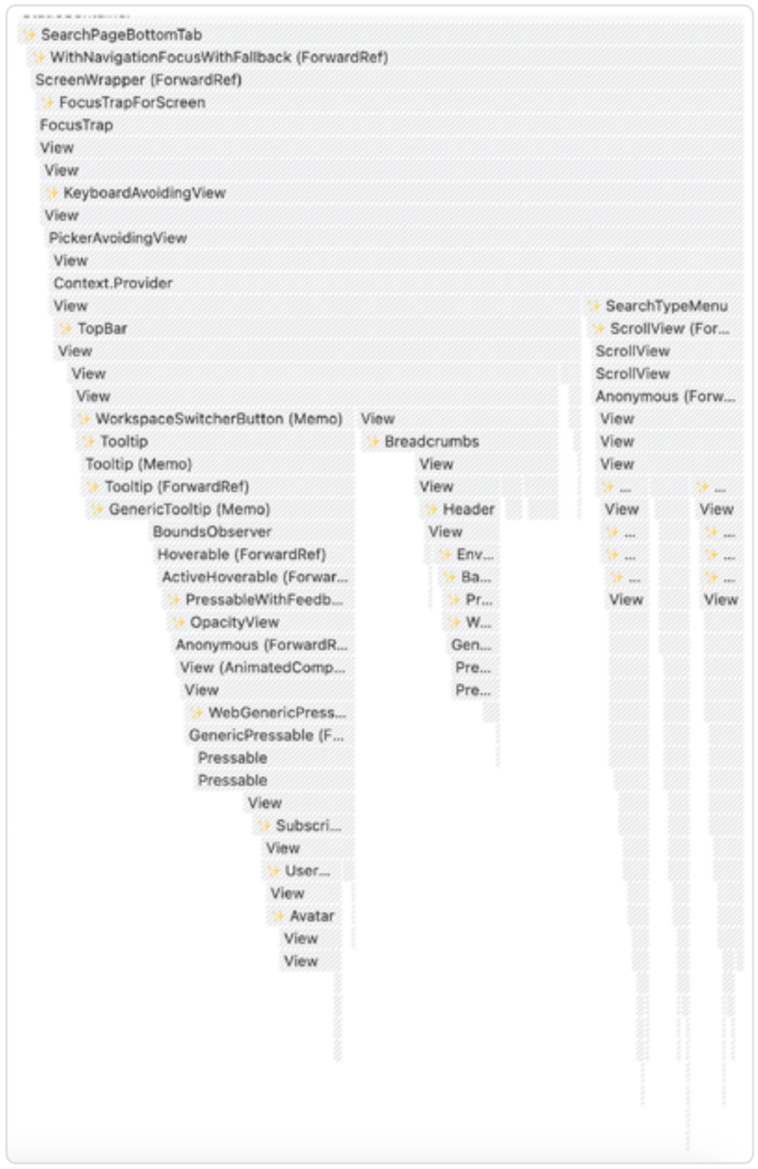
What performance improvements can you expect?

React Compiler aims to reduce unnecessary re-renders and minimize the need for manual optimizations. By automatically memoizing component calculations, it prevents cascading updates and improves performance, particularly in large applications with deep component trees.

Testing the React Compiler on the Expensify app showed measurable improvements in performance. For example, one of its most important metrics, "Chat Finder Page Time to Interactive" (effectively TTI) improved by 4.3%. While the React Compiler significantly reduces unnecessary re-renders, applications already optimized with manual memoization techniques may see only marginal improvements.



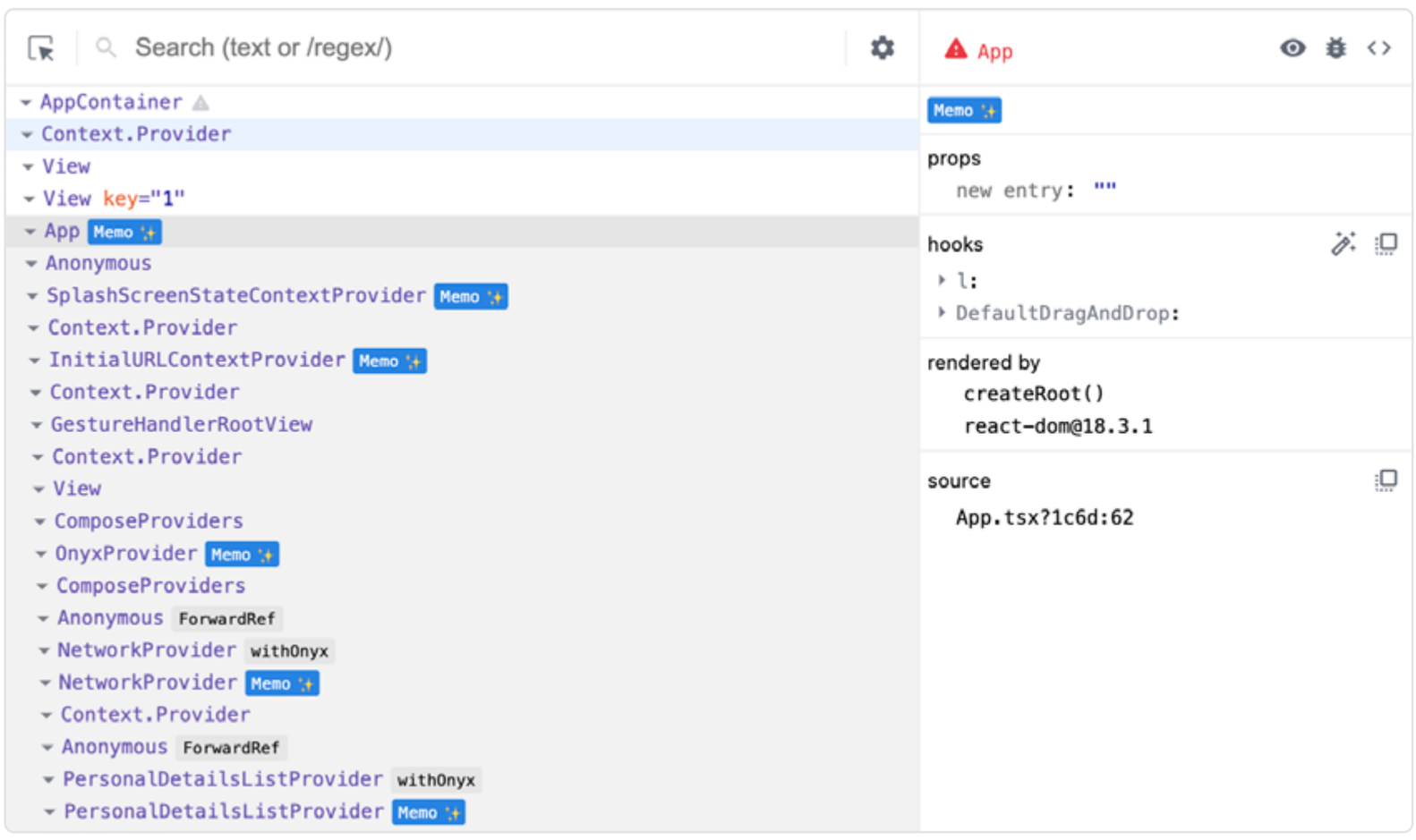
A single update in `SearchPageBottomTab` causes cascading re-renders across deeply nested components, increasing rendering time



React Compiler blocks unnecessary updates, reducing the re-render scope and ensuring that only truly affected components update

How to verify optimizations?

Open React DevTools to see which components have been optimized by the React Compiler. Optimized components are labeled with a **`Memo`** ✨ badge, which shows that the compiler has applied optimizations.



React Compiler is built for universal React but primarily tested in web environments. Enabling it in React Native might require additional steps to ensure `Memo` ✨ badges appear correctly.

Since React Native includes its own version of `react-devtools`, you may need to override it in your `package.json` and ensure it's updated to version 6.0.1 or newer. Without this, `Memo` ✨ badges may not appear.

To check your React DevTools backend version, open DevTools and click the ⚙ settings icon.

BEST PRACTICE

HIGH-PERFORMANCE ANIMATIONS WITHOUT DROPPING FRAMES

Animations play a crucial role in creating engaging mobile experiences. In React Native, ensuring smooth animations that run at least at 60 frames per second (FPS) is essential but sometimes challenging, especially when handling high-load operations. When animations drop frames or appear janky, they can significantly impact the user experience and make your app feel unpolished. In this chapter, we'll explain which tools you can use to ensure your animations are running as expected, their limitations, and best practices to follow.

Understanding the main thread in React Native

First, let's understand what the main thread is. In React Native, the main thread (also called the UI thread) is the primary thread responsible for handling UI rendering and user interactions. It's the thread where all visual updates happen and where the user interface responds to touch events.

In React Native, the terms "main thread" and "UI thread" are used interchangeably. When developers talk about either one, they refer to the same thing.

This is different from the JavaScript thread, where most of your React Native code runs, which we described more broadly in the introduction.

React Native Reanimated

When you want to add animations to your React Native app, you should go with React Native Reanimated, a powerful library that provides first-class support for performant animations.

React Native Reanimated is an industry standard for making animations. As of January 2025, it's a battle-tested solution with almost 1 million weekly downloads.

With Reanimated, you can easily run animations directly on the UI thread, thanks to the concept of worklets. Worklets are short-running JavaScript functions that can be run on the UI thread. They can also be run on a JavaScript thread just as you would run a function in your code.

Pioneered by Reanimated, worklets are a powerful concept that has found its way into other community libraries, such as VisionCamera or LiveMarkdown. Such libraries can use the **`createWorkletRuntime`** API to create a new JS runtime, which can be used to run worklets on different threads than the JS or UI thread.

Most of the time, when working with Reanimated, the code runs on the UI thread by default, such as in the case of the **`useAnimatedStyle`** hook:

```
const style = useAnimatedStyle(() => {
  console.log('Running on the UI thread');

  return { opacity: 0.2 };
});
```

Reanimated comes with a variety of helpers. One that we'd like to pay special attention to is **`runOnUI`**. It allows you to run any code on the UI thread manually. It can be used, for example, with an **`useEffect`** to start an animation on component mount/unmount or with **`measure`** and **`scrollTo`** functions, which have implementations only on the UI thread. You can use it like this:

```
runOnUI(() => {
  console.log('Running on the UI thread');
});
```

There's also another utility function caled **`runOnJS`**, which you can leverage to run functions that are not worklets and couldn't run on UI thread, e.g., functions that come from external libraries.

```
// Function that needs to run on JS thread
const notifyCompletion = () => {
  console.log('Animation completed!');
  // Could be analytics tracking, state updates, etc.
};
```

```
const triggerAnimation = (targetValue: number) => {
  progress.value = targetValue;
  scale.value = withTiming(
    targetValue,
    { duration: 400 },
    (finished) => {
      if (finished) {
        // Call JS thread function from UI thread
        runOnJS(notifyCompletion)();
      }
    },
  );
};

const animatedStyle = useAnimatedStyle(() => {
  return {
    opacity: progress.value,
    transform: [{ scale: scale.value }],
  };
});
```

The choice of which thread to run your code depends on several factors:

UI thread (main thread)

- Best for visual animations, transforms, and layout changes.
- Use when you need direct manipulation of UI elements.

JavaScript thread

- Best for complex calculations, data processing, and state management.
- Use when you need to access React state or props or perform business logic.

InteractionManager

React Native comes with `InteractionManager` API, which allows long-running work to be scheduled after any interactions/animations have been completed. This allows JavaScript animations to run smoothly, helping avoid jank and dropped frames.

When users interact with the app (e.g., tapping a button or handling gestures), executing heavy tasks on the JavaScript thread simultaneously with UI updates can lead to performance issues and undesirable behavior. With `InteractionManager`, we can schedule non-critical tasks to run after all current interactions have been completed:

```
InteractionManager.runAfterInteractions(() => {
  console.log('Running after interactions');
});
```

The touch handling system considers one or more active touches to be an 'interaction' and will delay **`runAfterInteractions()`** callbacks until all touches have ended or been canceled.

You can also register animations with **`InteractionManager.createInteractionHandle`** function, which, when cleared manually after interaction, will run all the scheduled actions.

```
const handle = InteractionManager.createInteractionHandle();
// run animation... (`runAfterInteractions` tasks are queued)
// later, on animation completion:
InteractionManager.clearInteractionHandle(handle);
// queued tasks run if all handles were cleared
```

Pairing it with React Navigation

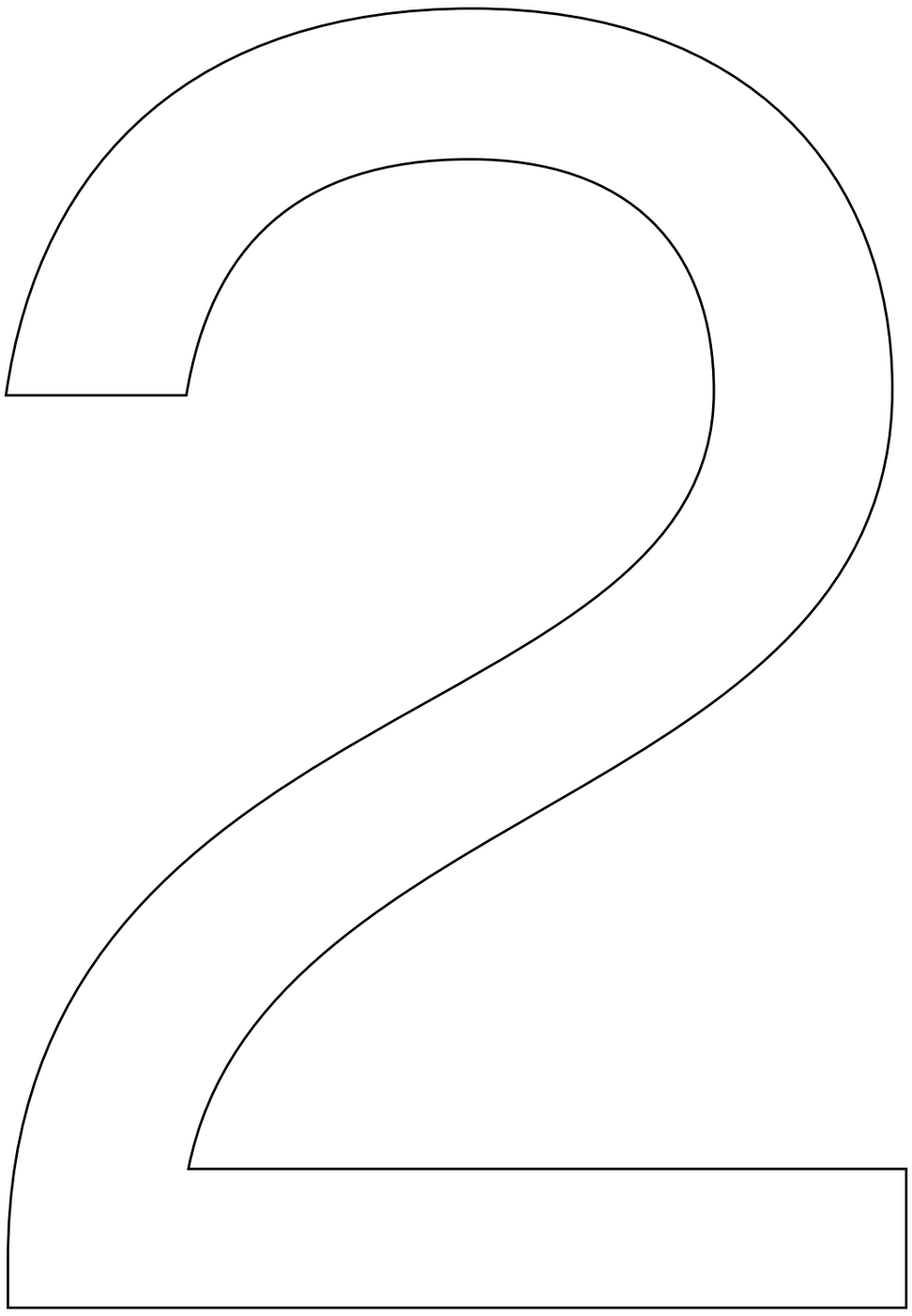
A typical scenario where you'd like to use **`InteractionManager`** is with React Navigation's **`useFocusEffect`**. This hook runs the effect as soon as the screen comes into focus. It often means that if there is an animation for the screen change, it might not have finished yet.

React Navigation runs its animations on the UI thread, so it's not a problem in many cases. However, if the effect updates the UI or renders something expensive, it can affect the animation performance. In such cases, we can use **`InteractionManager`** to defer our work until the animations or gestures have finished:

```
useFocusEffect(
  React.useCallback(() => {
    const task = InteractionManager.runAfterInteractions(() => {
      // Expensive task eventually updating UI
    });
    return () => task.cancel();
  }, [])
);
```

Alternatively to **`InteractionManager`**, which is a React Native-specific API, you can also leverage the latest additions to React, such as the **`startTransition`** hook, to move non-critical updates for later execution to not block the UI update. Read more about it in the Using transitions for non-critical updates chapter.

PART



NATIVE

Guides and techniques to improve FPS by optimizing Native side of React Native on iOS, Android, and in C++

Introduction

In Part 1, we explored a simplified app start model for a React Native application on Android or iOS. We also learned that up to 80% of performance issues come from the JavaScript thread, which leads to the conclusion that at least 20% of such issues are related to the native side. This leaves quite a lot of room for improvement.

Therefore, Part 2 will focus on the native side of the timeline. We'll start with the most crucial part impacting the app startup—the code that runs before the JS thread is initialized. Then, we'll follow up with runtime optimizations that can help us achieve better performance after the app has started.

Process (pre-)initialization

Getting the app to display our UI on the screen is not that simple. First, the operating system (we'll focus on Android for the sake of this example) needs to initialize the process that will bootstrap the app. In fact, through different mechanisms, the OS may decide to perform such an initialization fully or partially, even before the user opens the app.

This technique is referred to as a warm or hot startup on Android, and you can read more about it on the Android Developer portal. iOS supports a similar optimization technique called prewarming.

Understanding these mechanisms is particularly important when measuring app startup reliably, where we always need to focus on cold startup. During initialization, the operating system will also create a secure sandbox for the app's in-memory data and notify other system services about it.

App start and restart

Once the process is initialized, the OS will enter application initialization (using `Application.onCreate`) and launch the Activity and its `Activity.onCreate()` method. This is where native engineers can initialize their code, and the React Native journey can finally start. The OS will then call other app lifecycle methods, such as `Activity.onStart()`, and get to the UI rendering, where the UI thread will process layout and drawing tasks, measure views, and display them on the screen.

React Native's threading model

Wait, what's a UI thread? If you have a JavaScript background, you may be intimidated by the topic of threads and a threading model. After all, JS is a single-threaded language that relies on concurrency instead and avoids all the pitfalls (and benefits) that threaded languages bring.

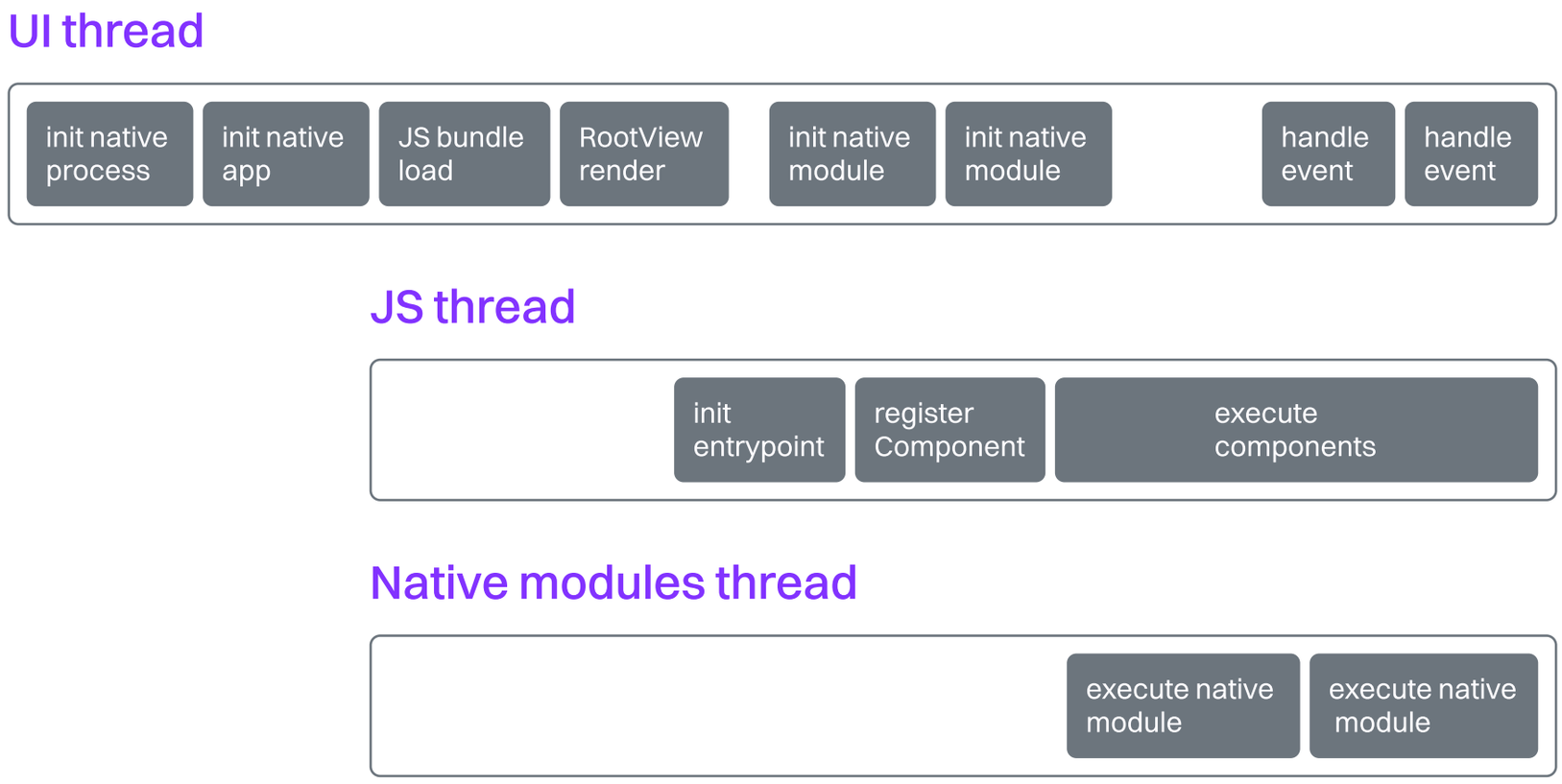
A thread is the smallest unit of execution within a process. It is an independent execution path that allows tasks to run concurrently within a single process, sharing the process's resources

like memory. In mobile app development, threads are used to perform tasks simultaneously, allowing the app to remain responsive.

Mobile developers typically operate between two kinds of threads: the UI thread, used mostly for UI rendering, and background threads that prevent blocking the main thread and are suitable for heavier tasks and the app's business logic.

Web developers are familiar with doing both UI rendering and processing business logic on a single thread and leveraging either CSS hardware acceleration or concurrency mechanisms, such as async functions, Promises, or Timeouts. These APIs leverage the event loop provided by the JS environment: a web browser or server runtime like Node.js. Such an event loop also exists in React Native apps and resides in its cross-platform C++ renderer Fabric, and is accessed by the Hermes JS engine.

As a side note, it's worth appreciating how React Native brings the best of both JS and native worlds into one declarative model. UI and background threading are mostly abstracted away, so React Native engineers can bring their web mental model of writing apps into mobile, with React's native renderer taking care of running appropriate code on the most suitable thread.



A more comprehensive threading model of React Native in New Architecture

React Native initialization

Let's return to the React Native part. Now that the native app is initialized, the native Kotlin code—that comes with the `com.facebook.react:react-android` library loaded during native initialization—can now perform its magic. This code will initialize Hermes, a default JS engine created and optimized for mobile React Native apps. Once that's done, it will load the available JS bundle, typically available in a Hermes Bytecode (HBC) format, into memory. This bytecode allows Hermes to avoid an expensive interpretation step, and once the HBC file is loaded into memory, it can be directly executed on the JS thread (*mqt_v_js*).

Before executing any JavaScript, the JS thread will handle any native modules that are explicitly marked for eager execution, such as C++ TurboModules or TurboModules using

New Architecture's Interop Layer. Once that's done, the JS code is executed, lazily initializing other TurboModules from a dedicated thread (*mqt_v_native*) and scheduling rendering and handling events to the UI Thread, where they're handled by the React Native's cross-platform C++ renderer called Fabric.

See it in action

Wouldn't it be nice if we could inspect these app start stages ourselves? In the following chapter, we'll do that using dedicated platform profilers for both Android and iOS platforms. With these learnings, you'll be able to inspect the app initialization stages yourself and gain a better overview of how things work internally. Be careful; it's pretty addictive to dig deep!

GUIDE

UNDERSTAND PLATFORM DIFFERENCES

As a JavaScript developer, you are most likely familiar with some of the popular IDEs or editors and stick to your favorite one while working on a project.

However, when you start working with React Native, you might feel overwhelmed by the number of new IDEs and tools thrown at you in the setup guide. It's both a blessing and a side effect of React Native being a **native** framework. Because of that, once in a while, you will be working with both Android Studio and Xcode.

Being comfortable with these tools will help you better navigate this guide and make you more productive with React Native. In this guide, we will highlight commonalities across the three platforms, key differences, and productivity tips to enhance your workflow.

This information will help you navigate other chapters in this guide, especially when we refer to Android Studio, Xcode, or different build phases.

If you are using Expo to build your application, you may not need to leave your IDE comfort zone too often. Expo does a great job abstracting away native pieces by using Dev Clients, Config Plugins, and others. The contents of this guide are mostly relevant to projects that rely on React Native Community CLI.

Prerequisites

Before moving on, ensure you have Android Studio and/or Xcode environment ready. If you went through the setup guide on the React Native website, you should already have them installed on your machine. If you haven't done this yet, now is a great time to do it. React Native documentation provides an in-depth explanation, so we will skip this topic in this guide.

While every platform—Web, iOS, Android—is different, they share some similarities. Realizing this will make your job as a React Native developer a little easier.

IDEs

First and foremost, when building an app, you need an editor to write and edit code, at the very least. For JavaScript, you typically use whatever you are the most familiar with—VSCode, WebStorm, Neovim—to name a few. Since these tools are rather unopinionated, it is best practice to have your environment configured to work with TypeScript, ESLint, Prettier, and React.

For Android and iOS, it is generally recommended to work with IDEs provided by Google and Apple, respectively. While you can write Swift or Kotlin from within VSCode, the aforementioned IDEs provide a more predictable and bullet-proof setup overall.

The features that those native IDEs provide out-of-the-box include:

- View Hierarchy Debugger—inspects and debugs UI layouts visually, allowing you to analyze element positioning, constraints, and rendering issues.
- Instrumentation & Performance Profiling—built-in tools for monitoring CPU, memory, battery usage, and UI responsiveness, such as Android Studio's Profiler and Xcode's Instruments.
- Advanced Debugging Tools—LLDB and GDB support in Xcode, Logcat, and Debugger in Android Studio, with deeper insights into stack traces, crash reports, and memory leaks.
- Seamless Gradle (Android) & Build System (Xcode) Integration—manage dependencies, build variants, signing, and packaging without extra setup; integrated workflows for code signing, app validation, and direct publishing to app stores.
- Automated Testing Tools—native unit and UI testing frameworks like XCTest (Xcode) and Espresso/JUnit (Android Studio) with built-in test runners.

We will rely on them as we walk through other guides and best practices in this part.

Dependency management

When working with a web or mobile application, we often rely on third-party modules. In React Native, a library can be a combination of JavaScript, iOS, and Android dependencies. Understanding how dependencies are managed across platforms will help when working directly with native modules.

React Native abstracts most of this complexity with autolinking, which automatically detects and links React Native-specific modules. However, autolinking only works for libraries designed explicitly for React Native. If a dependency has native code for iOS and Android but isn't structured as a React Native module, additional steps are required.

Let's examine how package management works for JavaScript, iOS, and Android. Understanding this process will help you navigate your dependencies and pull in additional native modules when needed.

JavaScript

For JavaScript dependencies, React Native primarily relies on the npm registry, a centralized repository for JavaScript libraries. This registry contains packages that may include only JavaScript code or a combination of JavaScript and native code.

Installing a JavaScript dependency in a React Native project is straightforward using **`npm`**:

```
> npm install react-native-bottom-tabs
```

This command fetches the package from the npm registry, adds it to **`node_modules/`**, and updates **`package.json`**.

Here are some files to look out for:

- **`package.json`**—defines project dependencies, scripts, and metadata.
- **`node_modules`**—stores all installed dependencies locally.
- **`package-lock.json`**—ensures version consistency across environments.

JavaScript developers often use alternative package managers, such as Yarn, PNPM, or Bun.

iOS

For iOS, React Native uses CocoaPods to manage native dependencies.

Unlike a traditional iOS project, where dependencies are fetched from CocoaPods (Podspec repositories) or Swift Package Manager, React Native libraries typically ship their native code to the npm registry, and CocoaPods uses local references to locate and load that code. However, if you need to add a regular iOS-native dependency that is not a React Native library, you will need to fetch it from CocoaPods.

For example, to add **`SDWebImage`** to your project, you have to add the following to your **`Podfile`**:

```
pod 'SDWebImage', '~> 5.0'
```

If you created a new React Native project with React Native Community CLI, the result may look as follows:

```
require_relative '../node_modules/react-native/scripts/react_
native_pods'
require_relative '../node_modules/@react-native-community/cli-
platform-ios/native_modules'

platform :ios, '12.4'
install! 'cocoapods', :deterministic_uuids => false

target 'HelloWorld' do
  # Here goes your native dependency
  pod 'SDWebImage', '~> 5.0'

  # Rest of Podfile
end
```

This is a typical Podfile from a fresh React Native project with some parts removed for readability; you can see the full contents of this Podfile here

The ~> operator in CocoaPods follows a pessimistic version constraint similar to SemVer (Semantic Versioning), but with a slight difference:

- `~> 5.0` allows versions `>= 5.0` but `< 6.0` (patch and minor updates allowed, but no major version updates).
- `~> 5.0.1` allows versions `>= 5.0.1` but `< 5.1` (only patch updates allowed, no minor version updates).

Once you've added your dependencies, go to the `ios` directory and install pods using the `pod install` command:

```
> cd ios && bundle exec pod install
```

Note that we're not running CocoaPods directly, but instead using the `bundle` command. Since CocoaPods is a Ruby project, we can leverage Bundler, Ruby's dependency manager, to have the most up-to-date CocoaPods version required for a given React Native project, specified in a top-level `Gemfile` that comes by default with every new React Native project.

After installing pods, you can quickly launch Xcode using the `xed .` command.

Here are some files to look out for:

- **`Podfile`**—defines dependencies required for the iOS app.
- **`Gemfile`**—defines CocoaPods version and other required plugins for a given React Native app.
- **`Pods`**—contains all installed dependencies fetched via CocoaPods. This folder is generated when you install dependencies and should typically not be committed to version control (it is listed in **`.gitignore`** in all React Native projects by default).
- **`.xcworkspace`**—the workspace file generated by CocoaPods, which must be used instead of **`.xcodeproj`** when opening with Xcode. It ensures that all CocoaPods-managed dependencies are correctly linked in Xcode.

While Apple's Swift Package Manager (SPM) is an alternative for native iOS projects, React Native does not yet support it, making CocoaPods the default choice.

Android

For Android, React Native manages dependencies using **Gradle**, Android's build automation tool. Similar to iOS, React Native-specific modules do not come from external registries like Maven Central or JCenter. Instead, they are referenced locally from **`node_modules/`** and compiled directly into the project.

For an Android-native dependency that is not a React Native module (e.g., **`Glide`** for image loading), you must manually add it to the **`android/app/build.gradle`** file:

```
dependencies {
    implementation 'com.github.bumptech.glide:glide:4.12.0'
}
```

After adding a new dependency in **`build.gradle`**, you need to synchronize the Gradle project to fetch and apply the changes.

While Gradle doesn't use **`~>`**, it offers other flexible versioning options:

- Exact version—**`'com.github.bumptech.glide:glide:4.12.0'`**
- Version range (**`+`** operator)—**`com.github.bumptech.glide:glide:4.+`**. Uses the latest minor or patch update within version **`4.x.x`**, but avoids major updates.
- Dynamic versioning—**`com.github.bumptech.glide:glide:+`**. Uses the latest version available. Generally not recommended, it may break your code unpredictably.
- For production apps, **always use exact versions** to ensure compatibility, just as CocoaPods' **`~>`** prevents breaking changes in iOS projects.

React Native projects come with a Gradle Wrapper (`gradlew`), which ensures all developers use the same Gradle version, avoiding compatibility issues. You can run it like this:

```
> cd android && ./gradlew clean
```

Once completed, you can access a new native dependency in your project.

Additionally, here are some files to look out for:

- `build.gradle` (Project-Level)—located in `android/build.gradle,` this file configures global project settings, including Gradle plugin versions and dependency repositories like Maven Central or JitPack.
- `build.gradle` (App-Level)—located in `android/app/build.gradle`, this file defines app-specific dependencies, build configurations, and compile options.
- `gradle.properties`—contains configuration flags and settings to optimize Gradle builds (e.g., enabling Hermes or ProGuard).
- `gradlew` & `gradlew.bat`—the Gradle Wrapper scripts that come with React Native, ensuring consistent Gradle versions across all development environments. The `.bat` file is used for Windows systems.
- `.gradle/` Directory—stores Gradle's cache, compiled scripts, and temporary build files. It is generated during the build process and should not be committed to version control.

For example, to list available Gradle tasks, you can run `./gradlew tasks`.

Building a project

When working with React Native, it's essential to understand how JavaScript, iOS, and Android handle code execution and compilation. Unlike native platforms that use compiled languages, JavaScript does not have a traditional build system but undergoes a different transformation process.

JavaScript

JavaScript is an interpreted language that does not require a separate compilation step like Swift or Kotlin. Instead, JavaScript code is executed by an engine at runtime. However, due to the multiple ECMAScript versions and evolving JavaScript standards, React Native does require a form of "compilation" known as transpilation to ensure compatibility across different environments.

In React Native, this process is handled by Metro, the JavaScript bundler that transforms modern JavaScript code into a version that the JavaScript engine (e.g., Hermes) can understand. Under the hood, Metro relies on Babel, which rewrites JavaScript syntax to match the latest supported version of the JavaScript engine used in React Native.

Since React Native depends on a JavaScript engine that evolves over time, the exact JavaScript version supported varies based on the Hermes version. React Native preset used by Babel ensures that JavaScript is always transpiled to match the latest engine capabilities used in React Native.

iOS

Unlike JavaScript, Swift and Objective-C are compiled languages. The iOS build system compiles source code into machine code before execution, ensuring optimized performance on Apple devices.

React Native for iOS relies on Xcode's build system, which uses Clang (for Objective-C) and LLVM (for Swift) to convert human-readable source code into an optimized binary (such as `.app` bundle).

Here are the key steps:

- Source Compilation: Swift and Objective-C files are compiled into machine code using Clang/LLVM.
- Linking: All compiled code, system frameworks, and external libraries (from CocoaPods) are linked together.
- Signing: Apple requires apps to be signed with valid certificates and provisioning profiles before they can be run on a device.
- Packaging: The final binary (`.ipa` file) is generated, containing the app and its resources.

Android

Similar to iOS, Android uses a compiled build system, but it relies on Gradle, which automates the process of building, testing, and packaging an Android app.

React Native for Android compiles Java/Kotlin source code into Dalvik Executable (DEX) format, which is optimized for execution by the Android Runtime (ART).

Here are the key steps:

- Source Compilation: Java/Kotlin files are compiled into `.class` bytecode files using the Java/Kotlin compilers.
- DEX Conversion: The compiled `.class` files are converted into `.dex` files, optimized for the Android Runtime.

- Resource Processing: XML layouts, images, and other assets are bundled.
- Linking: External libraries and dependencies (managed by Gradle) are linked.
- Signing: Android requires apps to be signed with a valid keystore file.
- Packaging: The final `.apk` or `.aab` file is generated for installation on devices.

Running the application on the device

Testing is an essential part of the mobile app development workflow. Unlike web applications, which can be tested directly in a browser, mobile apps require **si**mulators (emulators in Android) or physical devices for testing. Simulators allow developers to run and debug apps in a controlled environment that mimics real-world device behavior.

Android

You can interact with Android emulators using the following methods:

- Android Studio AVD Manager: The built-in tool for managing Android emulators. You can create, configure, and launch virtual devices here.
- Command Line (`adb` and `emulator` commands):
  - `emulator -list-avds`—lists all available virtual devices.
  - `emulator @Pixel_6_API_34`—launches a specific device.
  - `adb devices`—lists running emulators and connected physical devices.

CLI tools such as Expo CLI or React Native Community CLI use these commands under the hood to provide a better developer experience.

iOS

You can interact with iOS simulators via the following:

- Xcode: Apple's built-in tool for running iOS apps on a simulated device. You can launch it via Xcode > Open Developer Tool > Simulator.
- Command Line (`xcrun simctl` commands):
  - `xcrun simctl list`—lists available simulators.
  - `xcrun simctl boot "iPhone 15"`—boots a specific simulator.
  - `xcrun simctl shutdown all`—shuts down all running simulators.

Launching and managing simulators is something you will do quite often, and multiple apps are available to help you work with them outside of Android Studio and Xcode. Some that we use daily include:

- MiniSim
- Android iOS Emulator for VSCode
- Expo Orbit
- Shopify Tophat

A screenshot from MiniSim that shows opened simulators (with a tick) and other available simulators too

We hope this chapter, although not strictly correlated with performance optimizations, will give you a better overview of the tooling ecosystem React Native engineers operate within. The tools listed here are essential for further work on optimizing any React Native app.

GUIDE

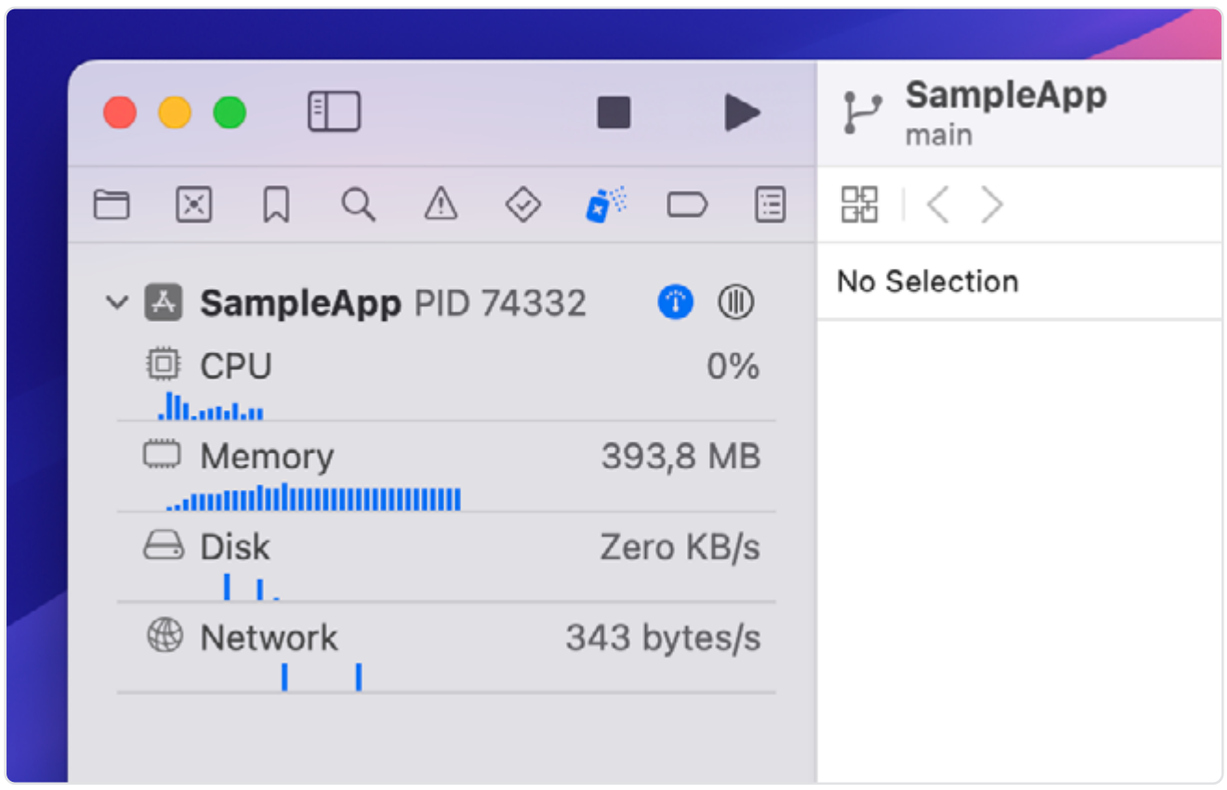
HOW TO PROFILE NATIVE PARTS OF REACT NATIVE

Profiling is essential not only to JavaScript code, which we covered in the How to Profile JS and React Code chapter, but also other languages and platforms, such as iOS or Android. Bottlenecks happen on both sides, and when React Native DevTools is not enough, we need to turn to the platform's native tooling—Android Studio or Xcode.

In addition to CPU, memory, and network profiling, battery usage is equally important for mobile users. It's usually a proxy to the CPU overhead if we remove screen brightness from the equation.

iOS

Xcode provides the Debug Navigator view to inspect CPU, memory, disk, and network load at a glance. It's located in Xcode's side pane under the "Debug Navigator" icon when your app is running and attached to the Xcode debugger:



Xcode's Debug Navigator

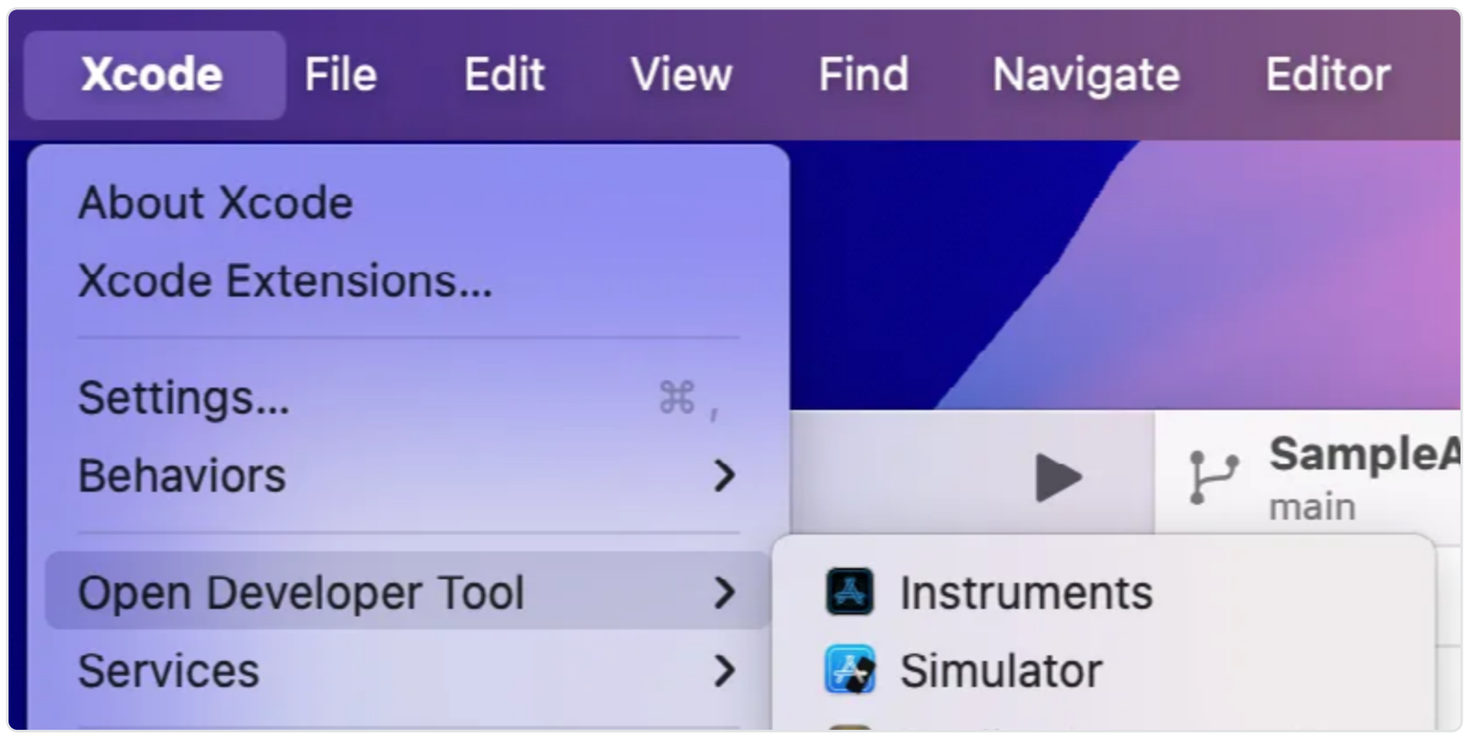
The CPU monitor measures the amount of work done over time. The percentage value shows usage relative to a single core, so it can exceed 100%, especially in React Native apps. The Memory Monitor observes the application's memory usage. All iOS devices use SSD for permanent storage—accessing this data is slower compared to RAM. Disk Monitor understands your app's disk-reading and writing performance. Network Monitor analyzes your iOS app's TCP/IP and UDP/IP connections.

You can tap on each of them to find more information. It also provides an extra monitor that isn't shown by default but can help you inspect your UI—it's the View Hierarchy, which we cover in the Use View Flattening chapter.

Instruments

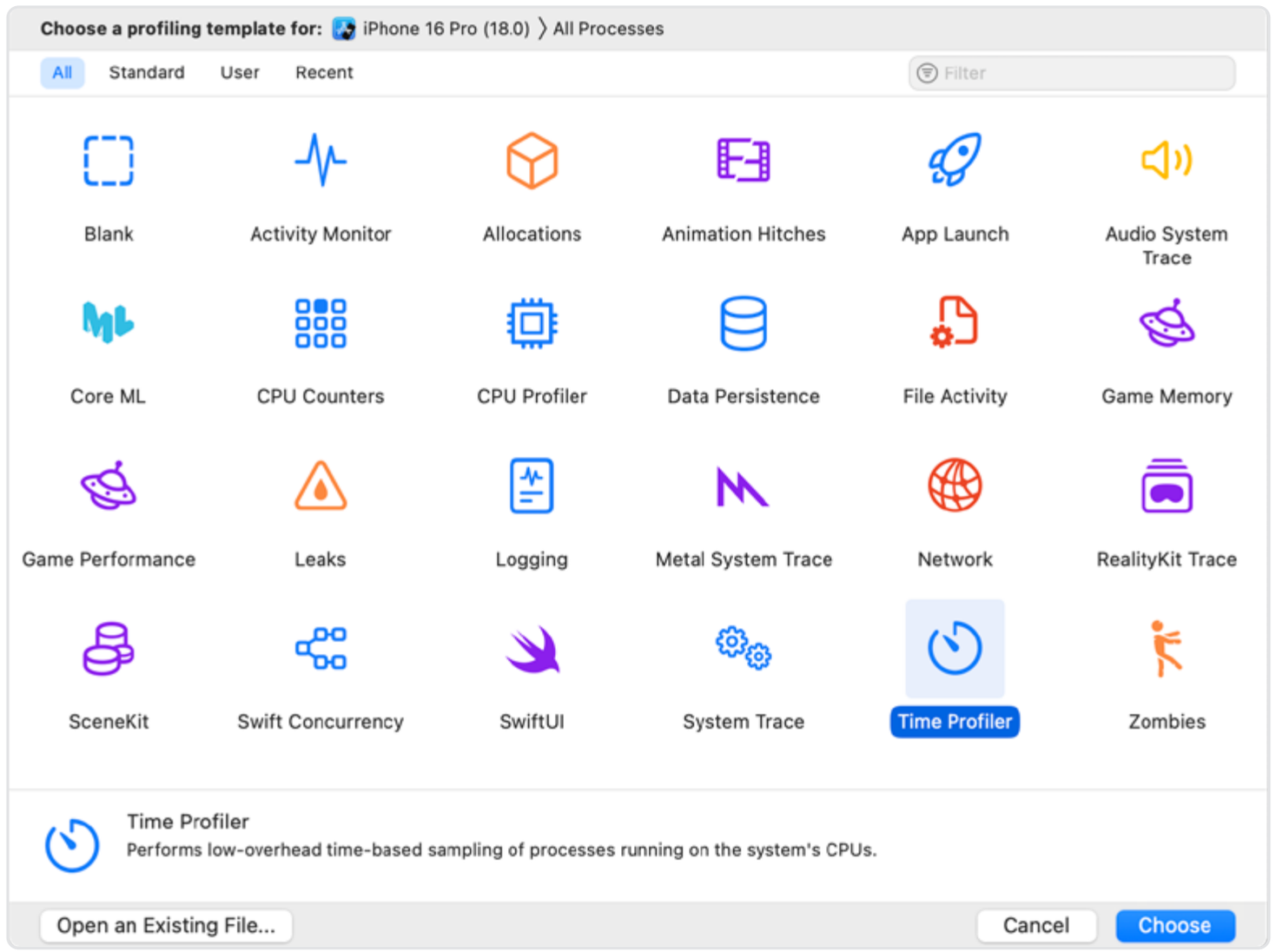
Xcode comes with a pre-packaged debugging and profiling tool called Instruments. It is a suite of inspection tools, each serving a different purpose. You choose from a list of templates, and you choose any of them depending on your goal: improving performance, battery life, or fixing a memory problem. For CPU profiling, we are going to use the Time Profiler. Let's dive into it.

With Xcode open, go to Instruments from Xcode > Open Developer Tool > Instruments menu.



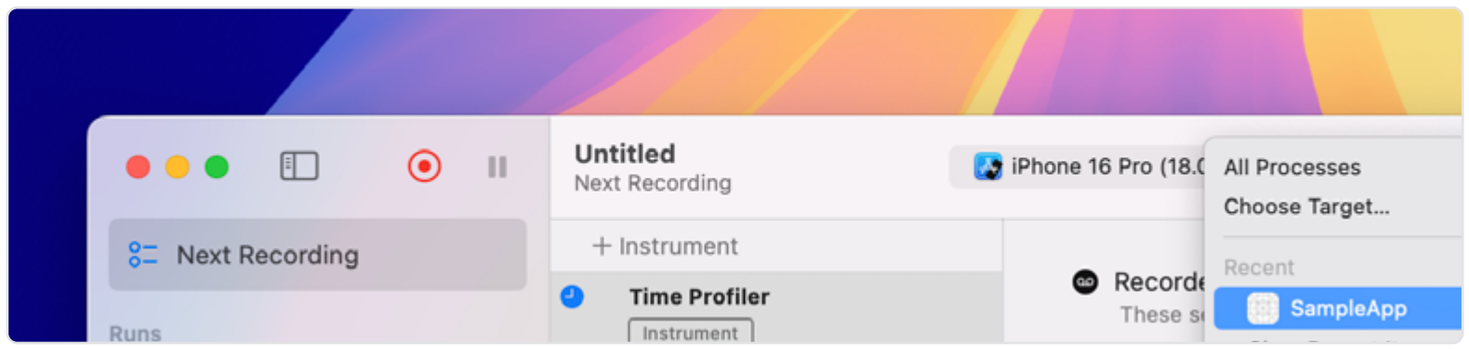
Xcode > Open Developer Tool > Instruments menu

Scroll down to find the Time Profiler tool, which includes CPU Profiler.



Available tools within Xcode's Instruments

It will open a new window. To start profiling your app, click on the dropdown menu next to the target device (here, iPhone 16 Pro), select the app (SampleApp), and hit the recording button (red circle):

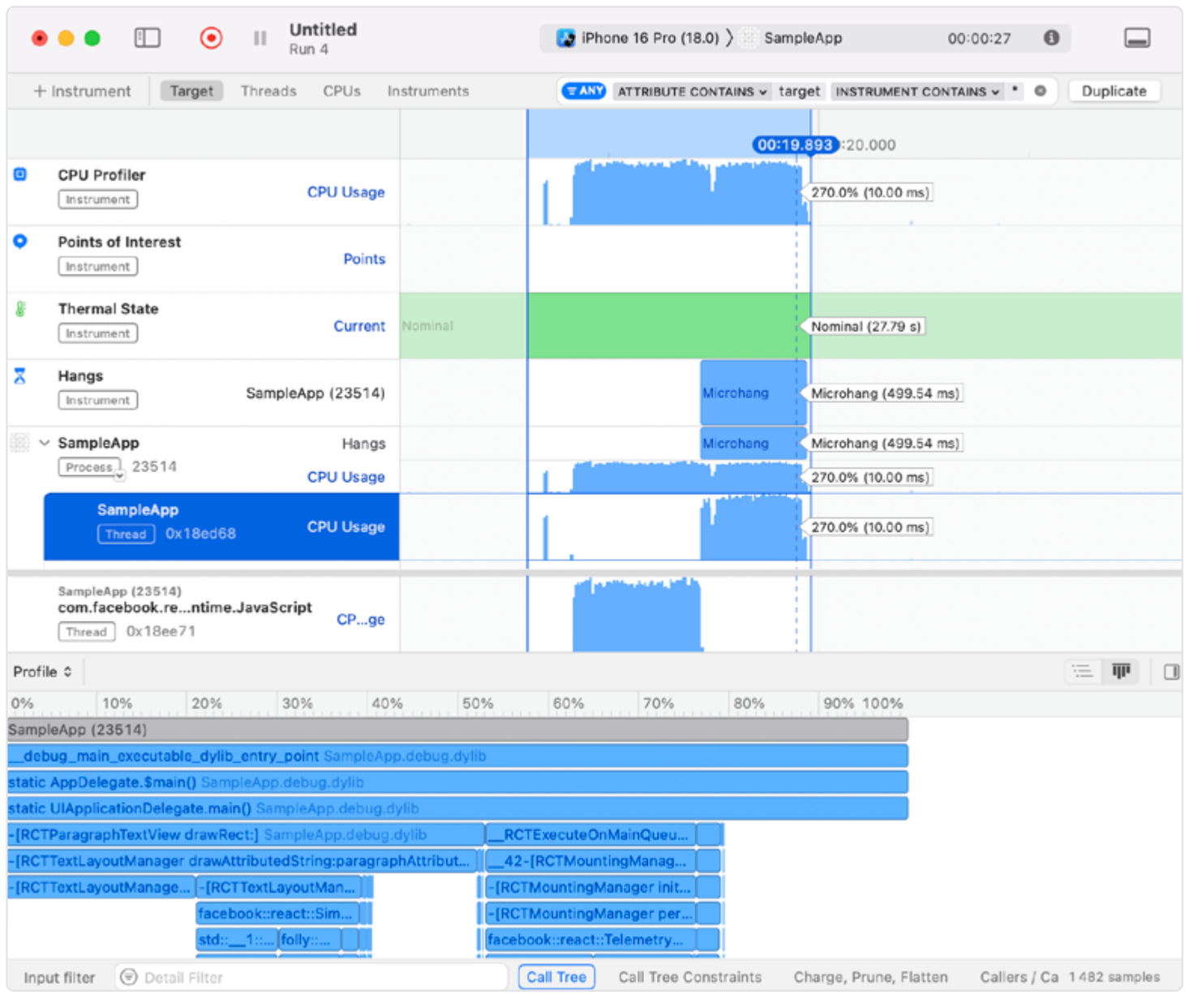


Select profiling target

The profiler will restart your app and start collecting CPU samples for profiling from the start. Now, you can use your app in a way that reproduces a performance regression you or your users found, and you want to investigate. Once you have enough data, stop the recording in the Instruments window.

In our case, we're rendering an application that presents 5,000 views in a `ScrollView` and is re-rendered with every state update when a button is pressed. On top of that, we'll toggle an image that will push the list down, causing a re-layout.

The profile of such interaction looks like this in the Instruments CPU Profiler:



Output of the CPU Profiler on a SampleApp, showing the breakdown of main thread, JS thread, and detected hangs

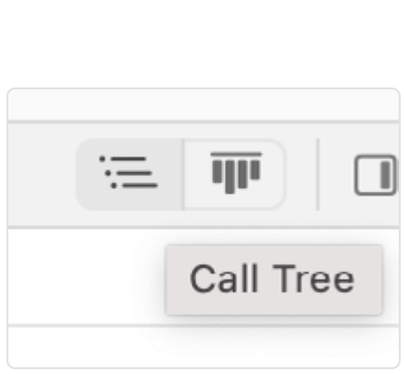
There's a lot to break down here. We've collected 27 seconds of data, but we're only interested in a small portion of that time. Using the `Cmd +` shortcut, we can zoom in on the timeline to our points of interest and horizontally scroll with a mouse or a touchpad.

In our case, we're focusing on ~1.3s of our total time when we press a button and wait for the app to update. You can notice a brief spike on the UI thread (labeled as SampleApp), followed by some intensive CPU work on the JavaScript thread and a lot of work on the UI thread. The profile is displayed as a flame graph on the bottom, showing React Native's internal call sites (we filtered out the system libraries).

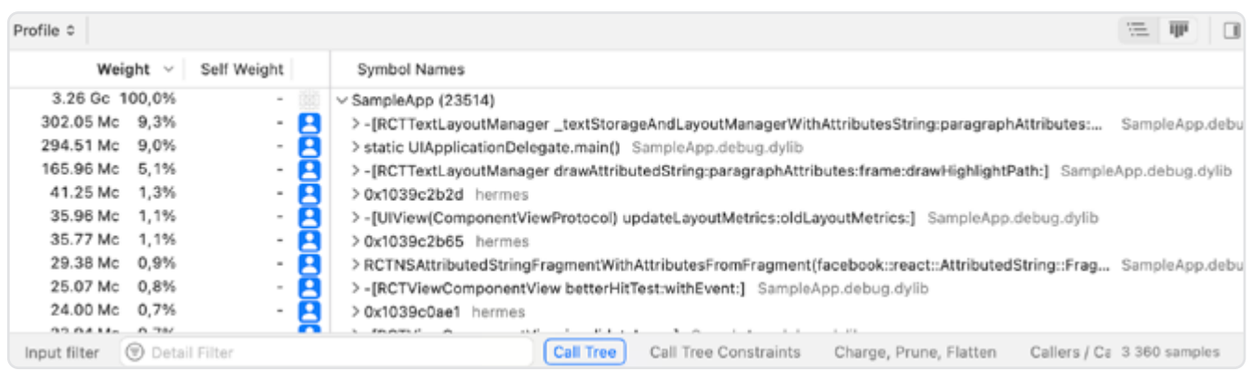
Notice how during the high CPU workload on the UI thread, there's a "Microhang", while a high JS thread load doesn't. This indicates that the UI thread is doing a lot of work, and it won't allow for immediate interactions with our content. It could be worse, though—if the thread was fully blocked, it would be displayed as a "Hang" and should be a high priority for you to investigate. The Hang tool accompanies the Time Profiler and can serve as a quick way to notice problematic parts of our app flow.

You can effectively block the JS thread, and users will still be able to interact with native UI elements of your app, e.g., press a button. It just won't communicate back to the JS thread immediately. It's a nice design implication of React Native that we often take for granted and is worth appreciating.

The CPU Profiler tool allows for quite a few customizations to help you pinpoint the source of the problem. For example, you can filter the Call Tree so it hides system libraries or invert the call tree altogether to access a bottom-up view you may know from other profilers that you can sort by weight expressed in "counts" (where, e.g., Mc is Mega counts, meaning millions of samples collected). It essentially allows you to see the functions where most of the time is spent, regardless of which higher-level function called them. You can also inspect the calls by thread if that helps.

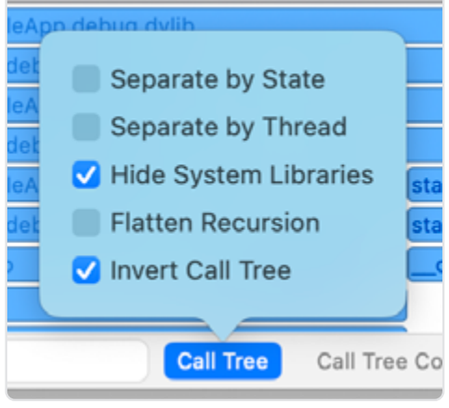


Toggle between Call Tree and Flame Graph view

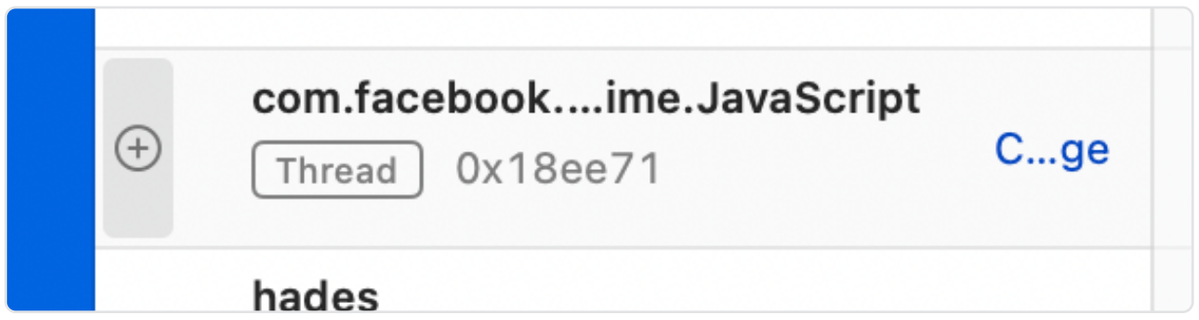


Call Tree view of the samples

You can toggle between a Call Tree and Flame Graph view, which you already may know from React Native DevTools. In our initial screenshot, we pinned the JavaScript thread to see how it correlates with the UI thread and hangs that were detected.



Helpful Call Tree filters you can toggle to quickly find most frequent function calls



Press the plus sign to pin the thread to have easier way comparing it with others on the timeline

Thanks to the CPU profiler, we can conclude that storing 5k elements in a **`ScrollView`** isn't optimal for our app's performance and we need to do something about it.

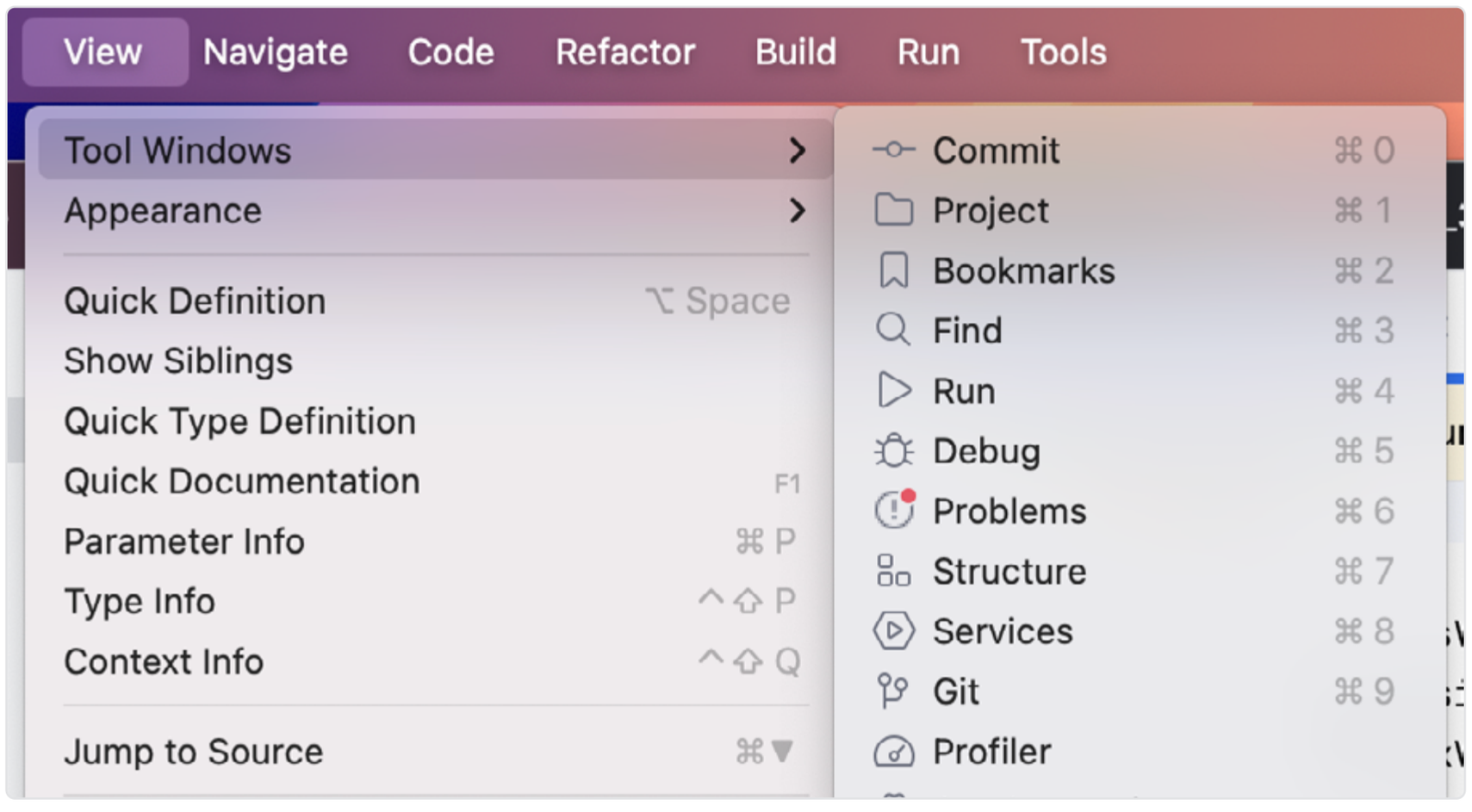
Android

For the Android platform, we can use Android Studio and built-in tools to profile our app's CPU, memory, network, and battery usage.

Android Profiler

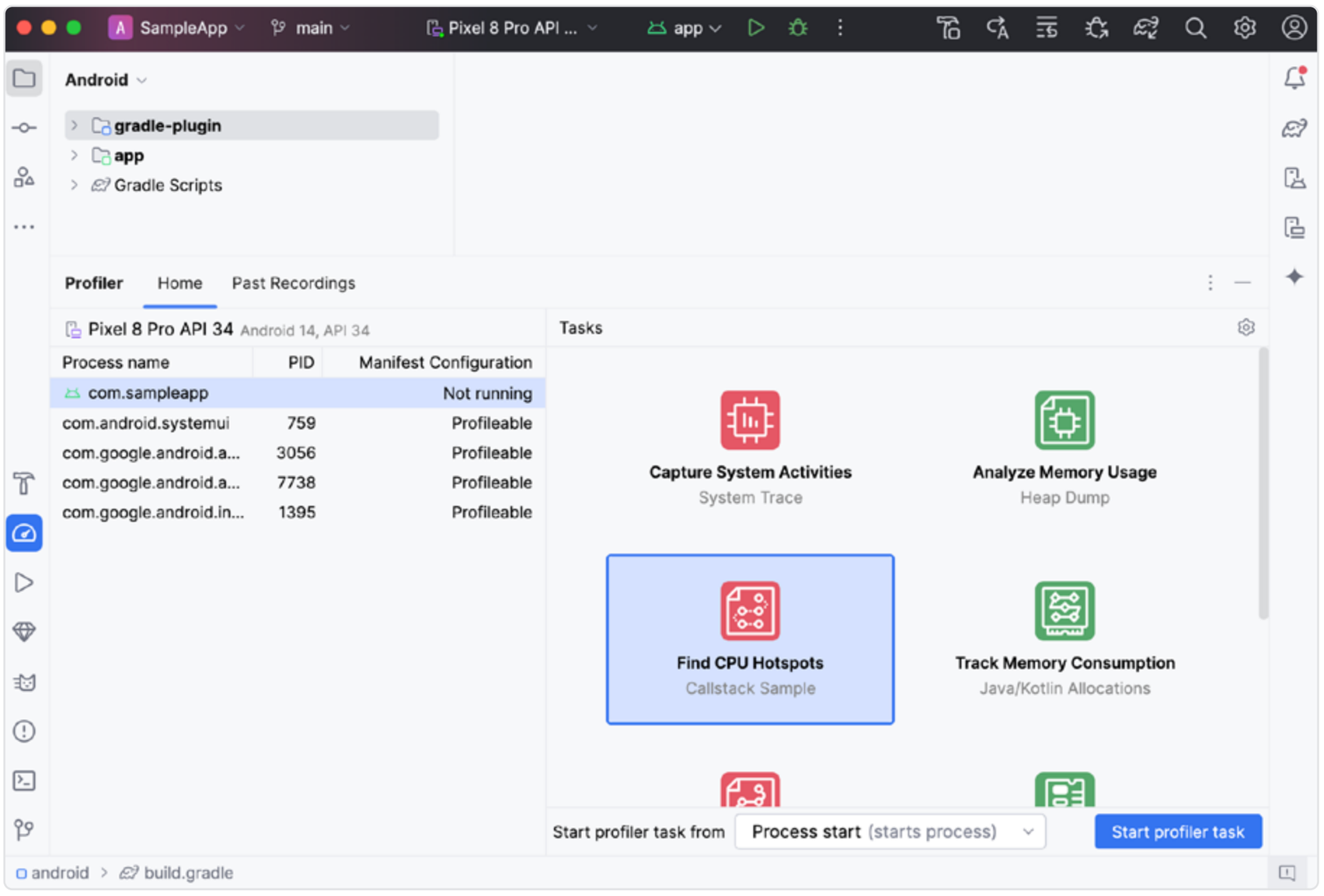
Android Studio is the IDE developed by JetBrains. It is officially supported by Google and an official Android IDE, which can be used to develop any Android app. It is very powerful and contains many functionalities in one place, including Android Profiler.

To open the Profiler, choose View > Tool Windows > Profiler from the Android Studio menu bar:



View > Tool Windows > Profiler menu

Or click "Profiler: Run 'app' as profileable" in the Android Studio's toolbar. If that doesn't work, you'll either need to run a debuggable build or create and run a profileable version of your app yourself using official Android Developer guidelines.



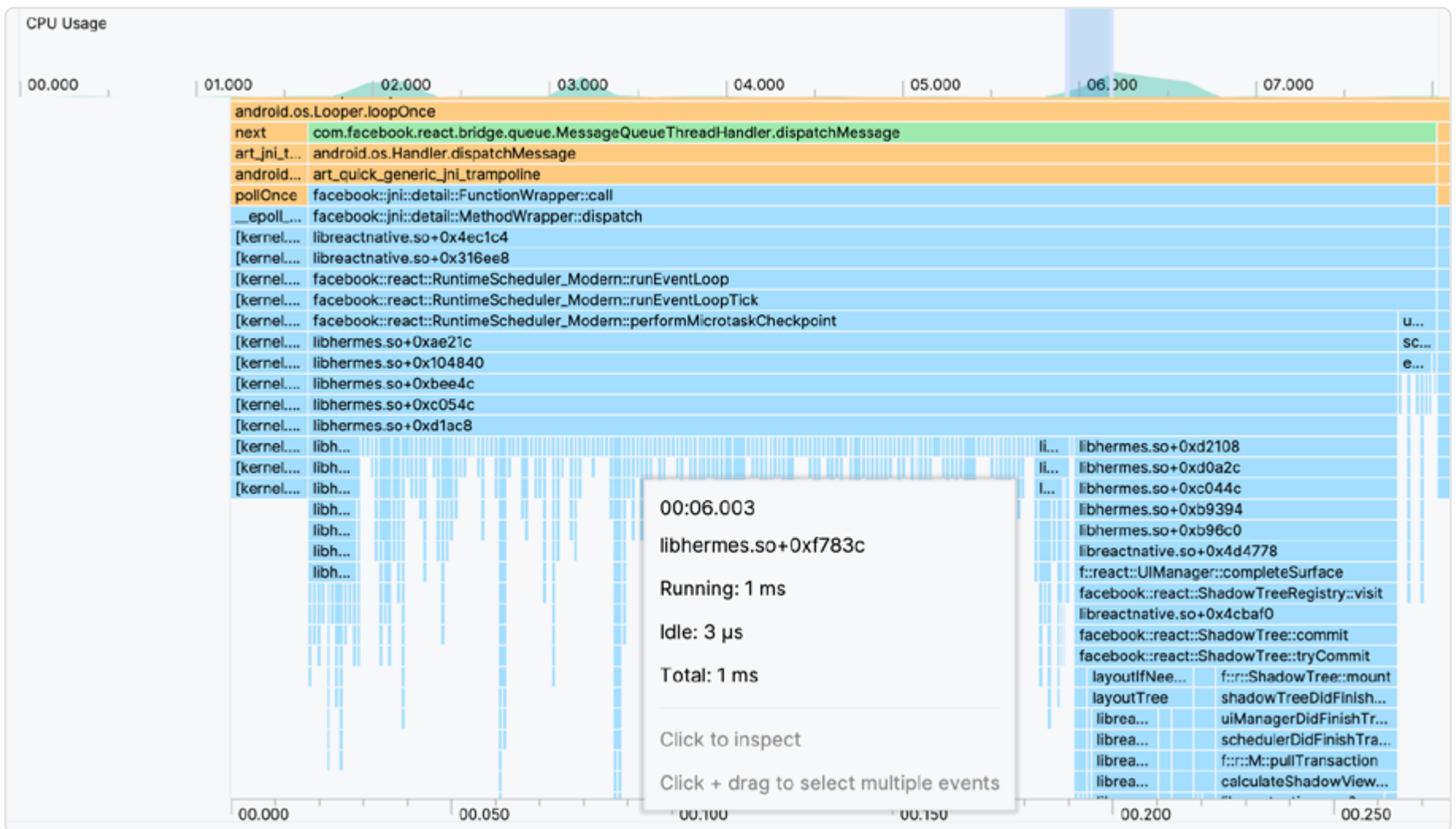
Android Studio Profiler with a handful of tasks like Find CPU Hotspots or Track Memory Consumption

Now, you should be able to connect to a running profileable app on your device. We'll use "Find CPU Hotspots" task and hit that "Start profiler task" button to start collecting samples from our app.

It's worth noting that the Android ecosystem of devices is way more fragmented than iOS. Mobile phones, tablets, TVs, etc., range from low-end through mid-end and high-end. Pick the lowest-end device available or emulator for profiling. Use data from real-time user monitoring if possible.

We'll perform the same scenario as with the iOS app: press our button a few times to cause a re-render of a giant list we have. When you stop profiling, the UI will change to a flame graph showing a breakdown of the work done on the CPU and various threads.

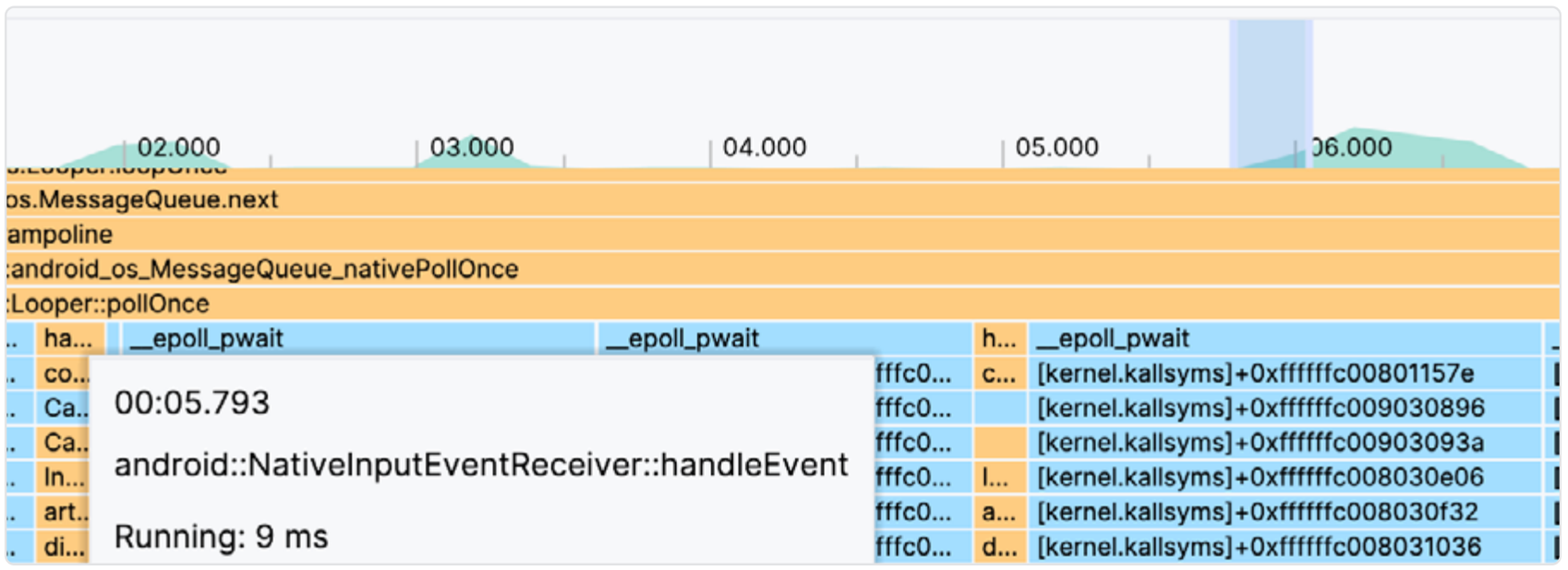
Let's zoom in on that JS thread right after we press the button; we can see an indication of higher CPU usage:



A zoomed-in view of the JS thread showing a lot of work performed by Hermes re-rendering the views

We can observe a lot of tiny 1ms spikes of activity coming from the Hermes JS engine going through all the 5k views we'll need to eventually draw on the screen. All that work sums up to over 240 ms, which is way more than a 16.6 ms budget to get at least 60 FPS. Notice how around a third of the total time is spent in the "commit" phase of the React reconciliation algorithm, which lays out new views with Yoga, to be then mounted and drawn onto the screen.

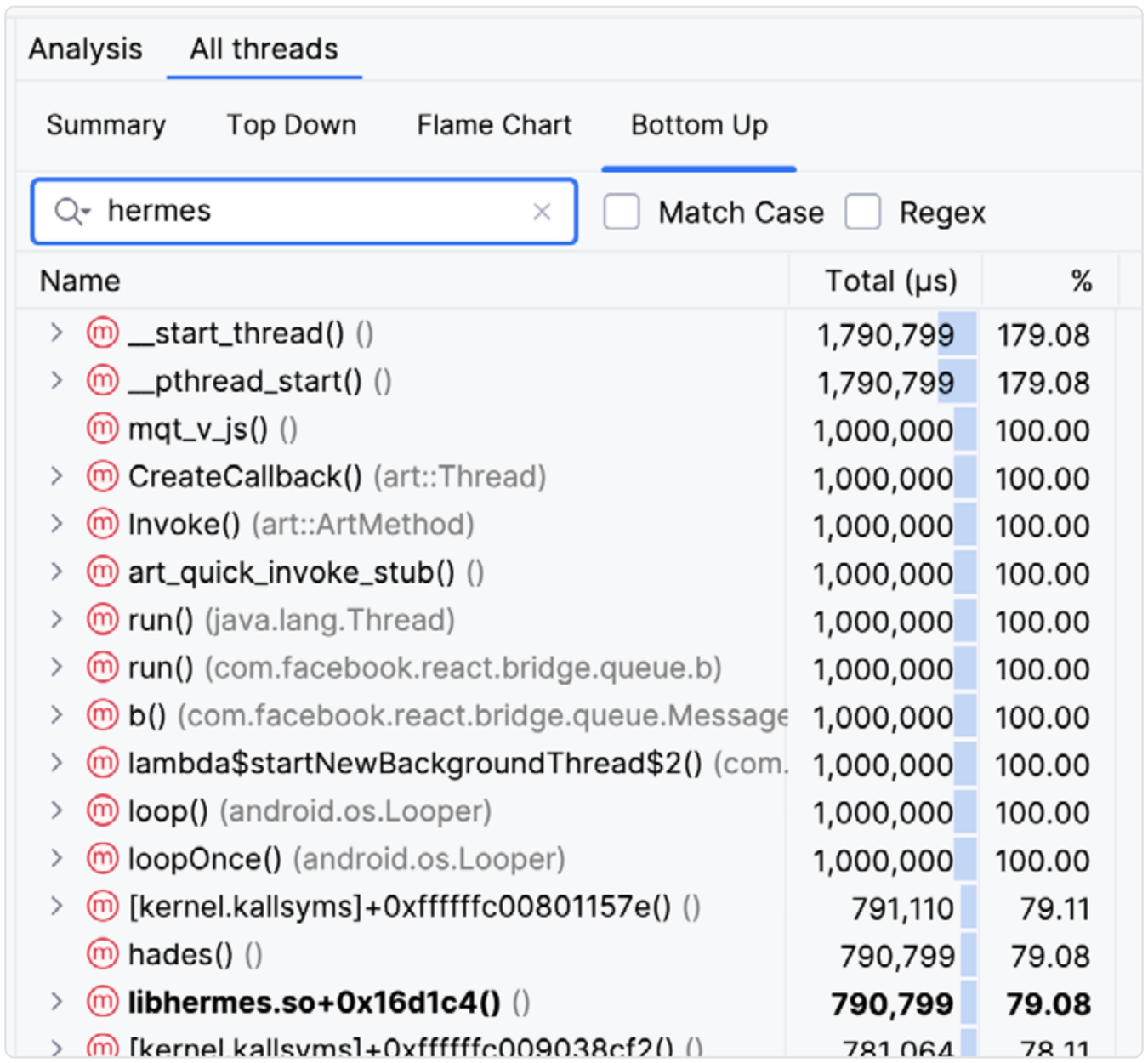
And if we scroll up a little on the flame graph to see the main thread, we can notice an event handler for pressing the button down, followed up by another one when releasing the finger.



Event handler on the main thread that starts the expensive work on JS thread later on

The touch release timing almost perfectly matches the start of the increased work on the JS thread, which is achieved through the sync JSI calls internally by React Native. In the legacy architecture, that information would be serialized and broadcasted by the bridge, adding extra overhead to this interaction.

As with any other profiling tool, we can also access a top-down or bottom-up analysis of the call tree, where we can additionally filter by libraries and calls that interest us.



A bottom up view of call sites filtered by the "hermes" keyword

It's worth remembering that profiling tools such as Android Studio or Chrome JS Profiler allow you to export the trace and load it in a third-party tool.

Perfetto

One such tool that might come in handy is Perfetto, which allows you to do system profiling, app tracing, and trace analysis by loading previously exported traces. It supplies Android Studio Trace Viewer with additional features. You can upload the trace through the online viewer available at ui.perfetto.dev and further inspect it from a different angle.

Android Studio-generated memory trace displayed in the Perfetto web app

Knowing how to use a profiler is one of the most crucial skills you can learn to help yourself fix any kind of performance issue at any codebase. It's especially helpful for the code you haven't even written. But with the right tools, you can understand its performance characteristics and apply optimizations where needed.

GUIDE

HOW TO MEASURE TTI

The app's Time to Interactive (TTI) metric is one of the two most important metrics every app should track. It tells us how much time elapses from touching the app icon to displaying meaningful content that users can interact with through touch, voice, or other input methods. Users expect this process to be as fast as possible, ideally instant.

Operating systems such as iOS even use machine learning-based predictions based on users' past experiences to "prewarm" certain apps so they open up to 40% faster. This helps create the impression that the whole iOS ecosystem is fast and responsive to our intents, even though some apps may be really slow to load.

Time to Interactive describes how quickly users can open and use our app. It's crucial for you to track and optimize, as it significantly affects user experience, satisfaction, retention, and, in effect, the revenue of the app you work on.

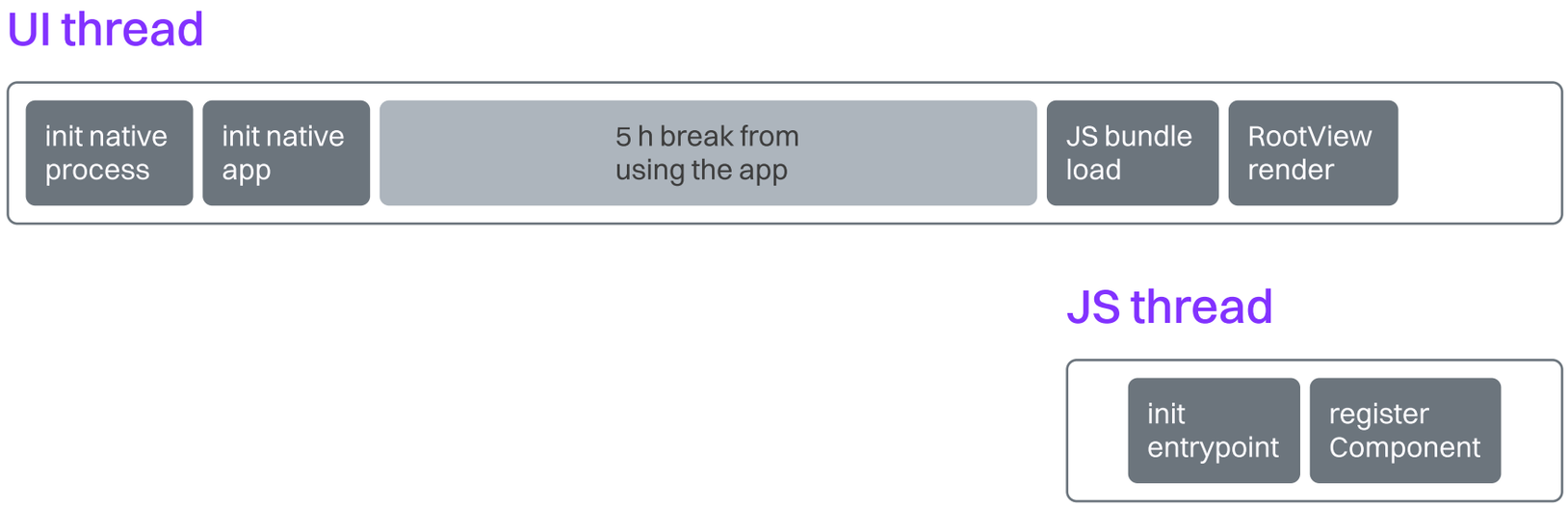
Various reports of mixed quality indicate that users typically expect an app to load in under 2 to 4 seconds; otherwise, they will drop for alternatives (if available). Our experience tells us, however, that you should trust your data and take external reports with a healthy dose of skepticism. The same experience also tells us that the TTI should be as low as possible.

But where to draw the line between the optimal effort and outcome? Is it worth spending months of the engineering team's capacity to get from 2.9s to 2.3s on a reference Android device? Your particular user base is unique, and only observing user behavior can give you an answer. In this chapter, we'll focus on the techniques to get you to the lowest TTI possible.

Measuring TTI reliably

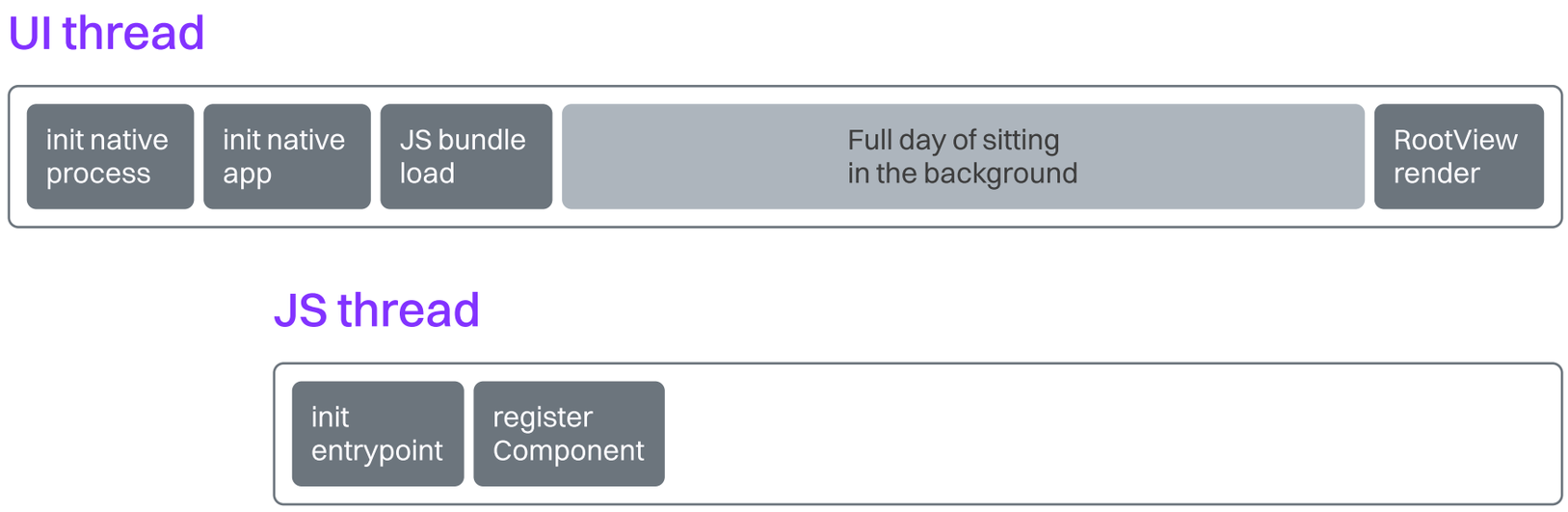
Depending on the underlying OS and its capabilities, our app can perform a cold start, warm or hot start, or it can be prewarmed. This means the same news app you open every day around

8 AM, which normally takes 10s to run on iOS when "cold", may start booting in just under 6 seconds after a few days. That's quite a difference.



A diagram showing how iOS prewarming can influence different stages of React Native app startup pipeline

In another scenario, if we open the app in the morning and come back to it in the evening, it may remain in the background. Now that we're accessing it, it's only going to render the React part, significantly speeding up the whole process compared to a cold start.



A diagram showing how Android putting an app into the background can influence different stages of React Native app startup pipeline

If we naively measured the TTI from every app start to its full interactivity, we would get vastly different measurements that would give us no insights into whether we regressed or improved this metric.

That's why it only makes sense to measure the app in the cold boot state and exclude measurements from all other boot states: prewarm, warm, or hot. Thankfully, iOS and Android creators provide their developers with means to deal with this problem.

Setting up performance markers

Each phase of the React Native app startup pipeline can be instrumented by a performance marker—a piece of code that collects information about the time certain events happen. In React Native applications, we want to focus on at least the following stages of this pipeline:

- Native Process Init—described by **`nativeAppStart`** and **`nativeAppEnd`** markers.
- Native App Init—described by **`appCreationStart`** and **`appCreationEnd`** markers.

- JS Bundle Load—described by **`runJSBundleStart`** and **`runJSBundleEnd`** markers.
- React Native Root View Render—described by **`contentAppeared`** marker.
- React App Render—described by **`screenInteractive`** marker.

To record these markers across iOS, Android, and React code, you will need a native module or a third-party library. The one that we're mostly happy with is `react-native-performance`. It exposes a **`RNPerformance`** class to iOS and Android, allowing you to define custom markers:

```
import ReactNativePerformance

RNPerformance.sharedInstance().mark("myCustomMark")
```

Custom marker on iOS

```
import com.oblador.performance.RNPerformance;

RNPerformance.getInstance().mark("myCustomMark");
```

Custom marker on Android

They are later available to React code using **`performance`** API, just like on the web:

```
import performance from 'react-native-performance';

performance.measure('myCustomMark');
performance.getEntriesByName('myCustomMark');
// returns: [{ name: "myCustomMark", entryType: "myCustomMark",
startTime: 98, duration: 2137 }]
```

Instead of defining custom markers, you can use built-in ones provided by the **`react-native-performance`** library, such as **`nativeLaunchStart`** or **`runJsBundleEnd`**. It's worth noting that the **`nativeLaunchStart`** is measured pre-main, so during prewarming phase, and the rest of the markers after prewarm. You may need to filter it out or create a custom one in **`main()`**.

Native Process Init

First, determine whether it's a cold start. On iOS, you can use **`ProcessInfo`** to get this information:

```
let isColdStart = ProcessInfo.processInfo.environment["ActivePrewarm"]
== "1"
```

On Android, you'll need to tap into the **`onActivityCreated`** lifecycle method to determine that information, so it's a bit more verbose:

```
class MainApplication : Application(), ReactApplication {
  var isColdStart = false
  override fun onCreate() {
    super.onCreate()

    var firstPostEnqueued = true
    Handler().post {
      firstPostEnqueued = false
    }
    registerActivityLifecycleCallbacks(object :
ActivityLifecycleCallbacks {
      override fun onActivityCreated(
        activity: Activity,
        savedInstanceState: Bundle?
      ) {
        unregisterActivityLifecycleCallbacks(this)
        if (firstPostEnqueued && savedInstanceState == null) {
          isColdStart = true
        }
      }
    })
  }
}
```

Since we want to measure TTI only when the app is in the foreground, we'll need to account for that. In **`AppDelegate.swift`** file on iOS, we can hook into the **`didFinishLaunchingWithOptions`** handler:

```
@main
class AppDelegate: RCTAppDelegate {
  var isForegroundProcess = false
    override func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [UIApplication.
LaunchOptionsKey : Any]? = nil) -> Bool {
      if application.applicationState == .active {
        isForegroundProcess = true
      }
      return true
    }
}
```

And in **`MainApplication.kt`** file on Android, we can check that through **`processInfo.importance`** API that we can wrap in a helper **`isForegroundProcess`** function:

```
class MainApplication : Application(), ReactApplication {
  private fun isForegroundProcess(): Boolean {
    val processInfo = ActivityManager.RunningAppProcessInfo()
    ActivityManager.getMyMemoryState(processInfo)
    return processInfo.importance == IMPORTANCE_FOREGROUND
  }
}
```

You can access the app initialization timestamp using the following APIs on iOS and Android to get data for the **`nativeLaunchStart`** metric:

```
var tp = timespec()
clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tp)
```

iOS nativeLaunchStart

```
Process.getElapsedCpuTime()
```

Android nativeLaunchStart

Then, the closest approximation of nativeLaunchEnd would be when a native module is initialized on iOS and when Android Content Provider is created on Android:

```
+ (void) initialize
```

iOS nativeLaunchEnd

```
class StartTimeProvider : ContentProvider() {
    override fun onCreate(): Boolean {}
}
```

Android nativeLaunchEnd

Initializers and other pre-main steps are run preemptively, potentially hours before the iOS app is started and **`main()`** is run. So, take the difference between process start time in pre-main initializer into account.

Native App Init

To hook into the time when the iOS app is created and get the **`appCreationStart`** marker, you can hook into the main method. For Android, you can leverage the Main Application's **`onCreate`** lifecycle method:

```
int main(int argc, char *argv[])
```

iOS appCreationStart

```
class MainApplication : Application(), ReactApplication {
    override fun onCreate() {}
}
```

Android appCreationStart

As prewarming on iOS executes an app's launch sequence up until, but not including, the time when `main()` calls `UIApplicationMain`, it's done until this point and we're safe from its effects on timing.

Then, the native app creation process ends with `didFinishLaunchingWithOptions` on iOS and the `onStart` lifecycle method on Android, where we can hook our `appCreationEnd` logic:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [UIApplication.
LaunchOptionsKey: Any]?) -> Bool
```

iOS appCreationEnd

```
class MyApp : Application() {
    override fun onStart() {}
}
```

Android appCreationEnd

JS Bundle Load

Next, we can finally access a React Native-related metric, `runJSBundleStart`. On iOS, we can tap into the global notification center listening for "`RCTJavaScriptDidLoadNotification`" event from the React Native's core, and on Android, we can import `ReactMarker` from the core to listen for that event:

```
NotificationCenter.default.addObserver(self,
    selector: #selector(emit),
    name: NSNotification.Name("RCTJavaScriptDidLoadNotification"),
    object: nil)
```

iOS runJSBundleStart

```
ReactMarker.addListener { name ->
    when (name) { RUN_JS_BUNDLE_START -> {} }
}
```

Android runJSBundleStart

In a similar fashion, we can listen to the end of the JS bundle loading to get our runJSBundleEnd marker:

```
NotificationCenter.default.addObserver(self,
    selector: #selector(emit),
    name: NSNotification.Name("RCTJavaScriptDidLoadNotification"),
    object: nil)
```

iOS runJSBundleEnd

```
ReactMarker.addListener { name ->
    when (name) { RUN_JS_BUNDLE_END -> {} }
}
```

Android runJSBundleEnd

React Native Root View Render

We can reuse the same React Marker infrastructure to detect when the React Native content appears and mark our **`contentAppeared`** marker:

```
NotificationCenter.default.addObserver(self,
    selector: #selector(emitIfReady),
    name: NSNotification.Name("RCTContentDidAppearNotification"),
    object: nil)
```

iOS contentAppeared

```
ReactMarker.addListener { name ->
    when (name) { CONTENT_APPEARED -> {} }
}
```

Android contentAppeared

React App Render

Our last marker, **`screenInteractive`**, will give us the overall TTI metric. There's no single place we can define this marker. It's specific to each application, and you'll need to decide when the user can safely interact with the contents of your app. As a base, you can put it in the initial

mount of the main `useEffect` of, for example, a home screen. Eventually, find a better spot, such as when a top-level content of your app, that users typically start with, is displayed:

```
export default HomeScreen() {
  useEffect(() => {}, [])
  return <TabNavigator {...} />
}
```

screenInteractive marker in React app

Putting it all together, you'll have a bird's eye view of the whole initialization pipeline of your app—from the native init to the screen, or ideally, multiple different screens, to being interactive. You can send this data to a database or a real-time user metrics platform, observe how your app's performance is affected on various devices, and—what's even more important—how it changes over time. Having this data, you'll be able to make exact correlations between TTI, and your app's customer success metrics. This data can and should influence the priorities of your engineering team, giving you signs that it's either time to optimize or cut back on optimizations that are not effective anymore and move focus somewhere else.

GUIDE

UNDERSTANDING NATIVE MEMORY MANAGEMENT

In native programming languages, developers are usually more or less responsible for managing the memory and resources during the app's execution. The way we do this depends on the language and the platform. Memory management can be split into two main categories: manual and automatic.

When manually managing memory, e.g., in languages such as C or C++, we are responsible for allocating and freeing the memory (and other resources). This approach is generally verbose and error-prone; however, when done right, it can yield the best performance.

In most cases, you should choose automated memory management to make your code easier to understand and maintain. However, if you want to squeeze every nanosecond out of your code and have a deep understanding of how to manage memory, you will be better off with manual management. It's a common practice, for example, in game development.

Knowing common patterns used by programming languages to manage memory is crucial because it allows you to reason about the code you write, making it more efficient and predictable. Let's start by exploring automatic memory management.

Memory management patterns

In this approach, memory is managed automatically using a garbage collector algorithm. We can highlight two common approaches, each with its pros and cons.

Reference counting

In mobile development, this pattern is used by Objective-C and Swift (under the name Automatic Reference Counting, or ARC), but it's also present in Python, PHP, and others.

Whenever we pass a reference to our object, the programming language's reference counter increments the number of its usages. This includes passing the object to another object or as

an argument to a function. When the reference is terminated (e.g., going out of scope), the use count is decremented. If the use count reaches zero, the object is deallocated, and resources are freed.

While it may impact runtime performance, it has the benefit of guaranteed cleanup and freeing the resources exactly when the last reference goes out of scope. Even though it seems like an ideal solution, we must be careful of "Reference Cycles", which happen when two objects hold "strong" references to each other. This essentially locks the reference counter at 1, preventing objects from being deallocated. A common solution for this issue is to use `weak` references, where there is a chance of introducing a reference cycle.

Garbage Collector

This pattern of managing memory is used in JavaScript, Java C#, and others.

Garbage Collector (GC) works independently from what's happening in your code. Think of it as a separate process, e.g., the Hermes engine spawns a separate thread where it runs its Hades GC. It can get invoked periodically or during one of the object allocations. Garbage Collector scans the program's memory and searches for objects that are no longer reachable. The unreachable objects are then freed.

This approach has no problems with reference cycles. Still, the moment when the Garbage Collection process starts is not deterministic, and we have no control over when the objects will be destroyed. Also, if the GC algorithm is not concurrent in any way, our application is paused during the GC process, which can impact perceived user performance in some cases.

Manual management

Lower-level languages, like C or C++, allow us to take full control over allocations and deallocations of objects in memory. Typically, there are two places where you can allocate memory: stack and heap.

The stack is a region of memory that stores data in a last-in-first-out (LIFO) manner, where memory is automatically allocated and deallocated. It's very fast, but limited in size and scope. The heap, on the other hand, is a larger region of memory used for dynamic allocation where data can be accessed globally. When you allocate memory on the heap (for example, using `new` or `malloc`), you are responsible for deallocating it when it's not needed anymore.

This technique is the most efficient when done right; however, if you don't know what you are doing, you can easily shoot yourself in the foot.

Managing memory

Now that you know the theory, let's dive into practice! In this section, we will go over Kotlin, Swift, and C++ memory management to give you a deeper understanding of how to manage memory in each language properly.

C++

In C++ programming language, memory management is explicit and manual. However, long gone are the times when you had to manually call the **`new`** and **`delete`** operators for all memory management. Instead, you can use smart pointers, which the C++ standard library (called **`std`**) provides. This way, you don't have to manually allocate and deallocate memory, resulting in fewer memory leaks in your application.

Further C++ materials assume some knowledge about the concept of pointers in C and C++ languages. If you're not familiar with it, feel free to learn more or skip this part.

There are multiple smart pointers available at your disposal:

- **`std::unique_ptr<T>`**—it cannot be copied to another **`unique_ptr`**, passed by value to a function, or used in any C++ Standard Library algorithm that requires copies to be made. A **`unique_ptr`** can only be moved.
- **`std::shared_ptr<T>`**—reference-counted smart pointer. Use it when you want to assign one raw pointer to multiple owners, for example, when you return a copy of a pointer from a container but want to keep the original.
- **`std::weak_ptr<T>`**—special-case smart pointer for use in conjunction with **`shared_ptr`**. A **`weak_ptr`** provides access to an object owned by one or more **`shared_ptr`** instances but does not participate in reference counting.

The most commonly used smart pointer is a **`std::shared_ptr<T>`**, which is a wrapper object providing a reference counting mechanism for the specified type, a memory management mechanism known from Obj-C and Swift. Let's go over an example usage of smart pointers:

Let's start with **`std::unique_ptr`**:

```
void takeOwnership(std::unique_ptr<std::string> s1) {
    std::cout << *s1;
    // Gets automatically deleted when function ends
}

int main()
{
    auto str1 = std::make_unique<std::string>("Hello World");

    std::cout << *str1;

    // Can only be moved. Doesn't allow for copying
    takeOwnership(std::move(str1));

    // str1 is cleared here
```

```
    return 0;
}
```

In the example above, we create a new unique pointer holding a **`std::string`** using **`make_unique`**. Then, using dereferencing, we can print out the value of the string. Additionally, the example shows the uniqueness of this smart pointer by transferring the ownership to a different function, which can only be done by moving. This is useful when you want to have exclusive ownership over the pointer for example when building immutable data structures.

Now, let's explore **`std::shared_ptr`**. We'll use the same example to make things easier to understand:

```
void takeOwnership(std::shared_ptr<std::string> s1) {
    std::cout << *s1;
    // After function returns, reference count gets back to 1
}

// We accept a reference
void takeReference(const std::shared_ptr<std::string> &s1) {
  std::cout << *s1;
}

int main()
{
    // Create a shared pointer
    auto str1 = std::make_shared<std::string>("Hello World");

    std::cout << *str1;

    // We increase the reference count by 1 and create a copy of
the pointer (not underlying values)
    takeOwnership(str1);

    // Since we didn't move the object it can be still accessed
here
    std::cout << *str1;

    // We don't alter the reference count since we pass a reference
here
    // This doesn't copy anything
    takeReference(str1);

    std::cout << *str1;

    return 0;
}
```

In the example above, we create a new shared pointer using **`std::make_shared()`**, which can be printed out using dereferencing. We pass it to the **`takeOwnership`** function, which passes the shared pointer by value (makes a copy of the pointer, not the underlying value). This operation increases the reference counter.

Since we can copy shared pointers, the value is not cleared, and we can still use it. Then, we pass it to the **`takeReference`** function, which, in contrast, doesn't increment the reference count since we only take the reference to this function. You should pass by reference when the function's use of the **`shared_ptr`** doesn't outlive the caller and pass by value when you need the `shared_ptr` to survive independently of the caller.

Now let's see how to use **`std::weak_ptr`**:

```
void useWeakPtr(std::weak_ptr<std::string> weak) {
    // Try to get access to the object
    // .lock() converts weak to shared_ptr if object exists
    if (auto shared = weak.lock()) {
        std::cout << *shared;          // Safe to use
        // shared_ptr reference count temporarily increases here
    } else {
        std::cout << "Object no longer exists!\n";
    }
}  // shared goes out of scope, count decreases

int main() {
    // Create a shared pointer
    auto str1 = std::make_shared<std::string>("Hello World");

    // Create a weak pointer from shared pointer
    std::weak_ptr<std::string> weak1 = str1;  // Doesn't increase
reference count

    std::cout << "Reference count: " << str1.use_count() << "\n";
// Shows 1

    // Use the weak pointer
    useWeakPtr(weak1);  // Object exists

    // Reset the shared pointer
    str1.reset();  // Reference count becomes 0, object is
destroyed

    // Try to use weak pointer again
    useWeakPtr(weak1);  // Will print "Object no longer exists!"

    return 0;
}
```

We can obtain a `weak_ptr` from a shared one; it doesn't increase the reference count. To use `weak_pt`, we have to use the `.lock()` method, which converts it to a shared pointer when the resource it points to still exists.

Android

On the Android platform, we mainly use Kotlin, which operates on JVM (Java Virtual Machine) that uses Garbage Collection mechanisms to handle working with memory. Similarly to JavaScript (which also uses GC), we can introduce memory leaks by not freeing up resources. An example is forgetting to unregister listeners or cancel a long-running background operation.

One of the few things we can do to instruct GC about managing memory is to use `Weak` types of containers, such as `WeakHashMap` (there is a similar construct in JavaScript called `WeakMap`). These types of containers don't create strong references to their items. Therefore, if nothing else is referencing them, the GC will deallocate those objects in its next pass. Here is an example usage of `WeakHashMap`:

```
fun main() {
    val weakMap = WeakHashMap<String, String>()

    // Add entries inside a scope
    run {
        weakMap[String("temp key")] = "some value"
        println("Map size: ${weakMap.size}") // Prints: 1
        println("Value: ${weakMap[key]}") // Prints: some value
    }

    // Force GC for demonstration
    System.gc()
    Thread.sleep(100)

    println("Map size after GC: ${weakMap.size}") // Prints: 0
}
```

What's important in the case of `WeakHashMap` is that it keeps weak references only to its keys. If you need a weak reference to the values stored in the map, you can use `WeakReference()` and assign the values as follows:

```
weakMap["key"] = WeakReference(LargeObject("1"))
```

iOS

As mentioned before, Swift and Objective-C use reference counting across the whole ecosystem. Let's explore this simple example to give you a better understanding of when the reference count is changed across scopes. We will use a `Person` class for the demonstration purposes:

```
class Person {
    let name: String

    init(name: String) {
        self.name = name
    }
}

do {
    let person1 = Person(name: "John") // Reference Count: 1
        do {
            let person2 = person1  // Reference count: 2
        } // person2 goes out of scope, Reference count: 1

} // person1 goes out of scope, Reference count: 0
```

When we assign **`person1`** to **`person2`**, we increase the reference count (we do not create a copy!). Then, after we go out of the **`do`** block scope, the reference count is set back to 1 and then to 0.

Sources of Memory Leaks

Let's examine some common patterns that cause our app to leak memory.

Forgetting to deallocate in manual management

When using manual memory management, it's easy to introduce a leak. As mentioned before, if you choose manual management, you are on your own. With great power comes great responsibility. To show you how easy it is to introduce a leak, let's consider this example:

```
int main()
{
    std::string *str1 = new std::string {"Hey"};

    std::cout << *str1;

    // Oops! Memory leak, we forgot to delete str1.

    return 0;
}
```

You allocate a new **`std::string`**, print its value, and, at the end, we forgot to delete this string; therefore, we introduced a memory leak. To fix it, you need to call **`delete`** when a resource is not needed:

```
int main()
{
```

```
    std::string *str1 = new std::string {"Hey"};
    // ...
    delete str1;

    return 0;
}
```

Another option would be to allocate this on the stack (dropping the **`new`**), which will automatically deallocate **`str1`** when it goes out of scope:

```
int main()
{
    std::string str1 = std::string {"Hey"};

    cout << str1;

    return 0;
}
```

You should always think twice before putting something on the heap, as it has a significantly bigger overhead than the stack. At the same time, you need to be cautious of how much you allocate on the stack, or else you risk... a stack overflow! But if possible, use stack and resort to allocating on the heap when necessary.

Reference cycles

The most common source of memory leaks when using reference counting are reference cycles, also known as circular references. Let's consider a simple example:

```
class A {
  std::shared_ptr<B> b;
};

class B {
  std::shared_ptr<A> a;
};
```

C++

```
class A {
    var b: B?
}

class B {
  var a: A?
}
```

Swift

In this example, class `A` depends on class `B`, which in turn depends on class `A`, causing a reference cycle.

It's worth noting that the reference cycles will be more complex in real-world scenarios; the cycles can span multiple classes, which makes them harder to find and fix.

The solution is to break the cycle by introducing a weak pointer (`std::weak_ptr<T>`). A weak pointer is an optional dependency that does not block the dependent object from being removed if it's no longer used. Whenever we want to access the dependency, we need to check if it still exists, as it is no longer guaranteed:

```
class A {
  std::shared_ptr<B> b;
};

class B {
  std::weak_ptr<A> a;
};
```

C++

```
class A {
    var b: B?
}

class B {
  weak var a: A?
}
```

Swift

Alternatively, we could introduce a third class, `C`, holding the shared resource that classes `A` and `B` depend on, which would help us avoid the cycle. This refactoring technique is commonly used in JavaScript codebases.

Not freed-up resources

In languages using garbage collector mechanisms, we can often introduce memory leaks by not freeing up resources. This can be observed when using the listener pattern and forgetting to call to remove them. If you come from a JavaScript background, you probably already know it all too well. Let's look at an example that showcases this common issue. First, create a simple `DataManager` class responsible for registering and calling listeners:

```
// Simple data manager class
interface DataListener {
    fun onDataChanged(data: String)
}

class DataManager {
    // List to store all registered listeners
    private val listeners = mutableListOf<DataListener>()

    // Method to register listeners
    fun registerListener(listener: DataListener) {
        listeners.add(listener)
    }

    // Method to unregister listeners
    fun unregisterListener(listener: DataListener) {
        listeners.remove(listener)
    }

    // Method to notify listeners of data changes
    fun notifyDataChanged(data: String) {
        listeners.forEach { it.onDataChanged(data) }
    }
}
```

Then, when we use this class, we have to remember about unregistering the listeners because otherwise, we will introduce memory leaks:

```
class MyClass {
    private val dataManager = DataManager()

    private val listener = object : DataListener {
        override fun onDataChanged(data: String) {
            println("Data changed: $data")
        }
    }

    init {
        // Register listener but never unregister
        dataManager.registerListener(listener)
    }

    // Memory leak: No way to unregister the listener
    // MyClass instance will never be garbage collected
    // as DataManager holds a strong reference to the listener
}
```

Since Kotlin doesn't have a built-in deterministic destructor or deinitializer, we need to use a cleanup lifecycle like **`AutoCloseable`**:

```
class MyClass : AutoCloseable {
    override fun close() {
        // Cleanup code here
        dataManager.unregisterListener(listener)
    }
}
```

We can also use **`WeakReference`** to store callbacks in **`DataManager`** class, which will automatically handle this.

```
class DataManager {
    // List to store all registered listeners
    private val listeners =
mutableListOf<WeakReference<DataListener>>()

    // ...
}
```

Using manual overrides

Languages that provide automated memory management usually also offer a manual way to alter the behavior, mostly for interoperability with external languages. While handy, when misused, this can be a hard-to-find source of memory leaks.

For example, in Swift, we can manually increment or decrement the reference count using the **`Unmanaged`** type. While doing so, you must be extra careful because it can lead to crashes and unexpected behavior. The Unmanaged class exposes a few functions that we can use to manage ARC behavior manually:

- **`passRetained`**—creates an **`Unmanaged`** instance and increments the reference count.
- **`passUnretained`**—creates an **`Unmanaged`** instance without incrementing the reference count.
- **`takeRetainedValue`**—gets the object and decrements the reference count.
- **`takeUnretainedValue`**—gets the object without affecting the reference count.

Let's consider an example using Unmanaged on a Swift class (in most cases, you would use this to interface with C).

```
class MyObject {
    deinit { print("Deallocated") }
}

let obj = MyObject()                          // Reference Count = 1
let unmanaged = Unmanaged.passRetained(obj) // Reference Count = 2
let object1 = unmanaged.takeRetainedValue() // Reference Count = 1
```

However, if you decrease the reference counter below 0, you will get a crash!

```
let obj = MyObject()                            // Reference Count = 1
let unmanaged = Unmanaged.passUnretained(obj) // Reference Count = 1
(no change)
let object1 = unmanaged.takeRetainedValue()   // CRASH!
```

A rule of thumb is to always match your retained and unretained calls. If you call **`passRetained`**, you should call **`takeRetainedValue`** afterward and vice versa. You can also call **`toOpaque()`** on the unmanaged objects to get its raw pointer, which can be later passed to C / C++.

```
unmanagedObject.toOpaque() // 0x000056076df5b1a0
```

Now you know the common memory management patterns used across languages. On top of that, you should now understand how Swift, Kotlin, and C++ manage memory. In the next chapter, we'll discover how to detect memory leaks using built-in profiler tools.

GUIDE

UNDERSTAND THE THREADING MODEL OF TURBO MODULES AND FABRIC

Understanding how React Native executes your code is crucial for building efficient native modules. In this chapter, we will walk you through the lifecycle of a native module, from the initialization phase to calls and de-initialization. By the end of this chapter, you should understand which thread is used under what circumstances.

Threading model

Let's start by exploring what threads are available to us in a mobile React Native app:

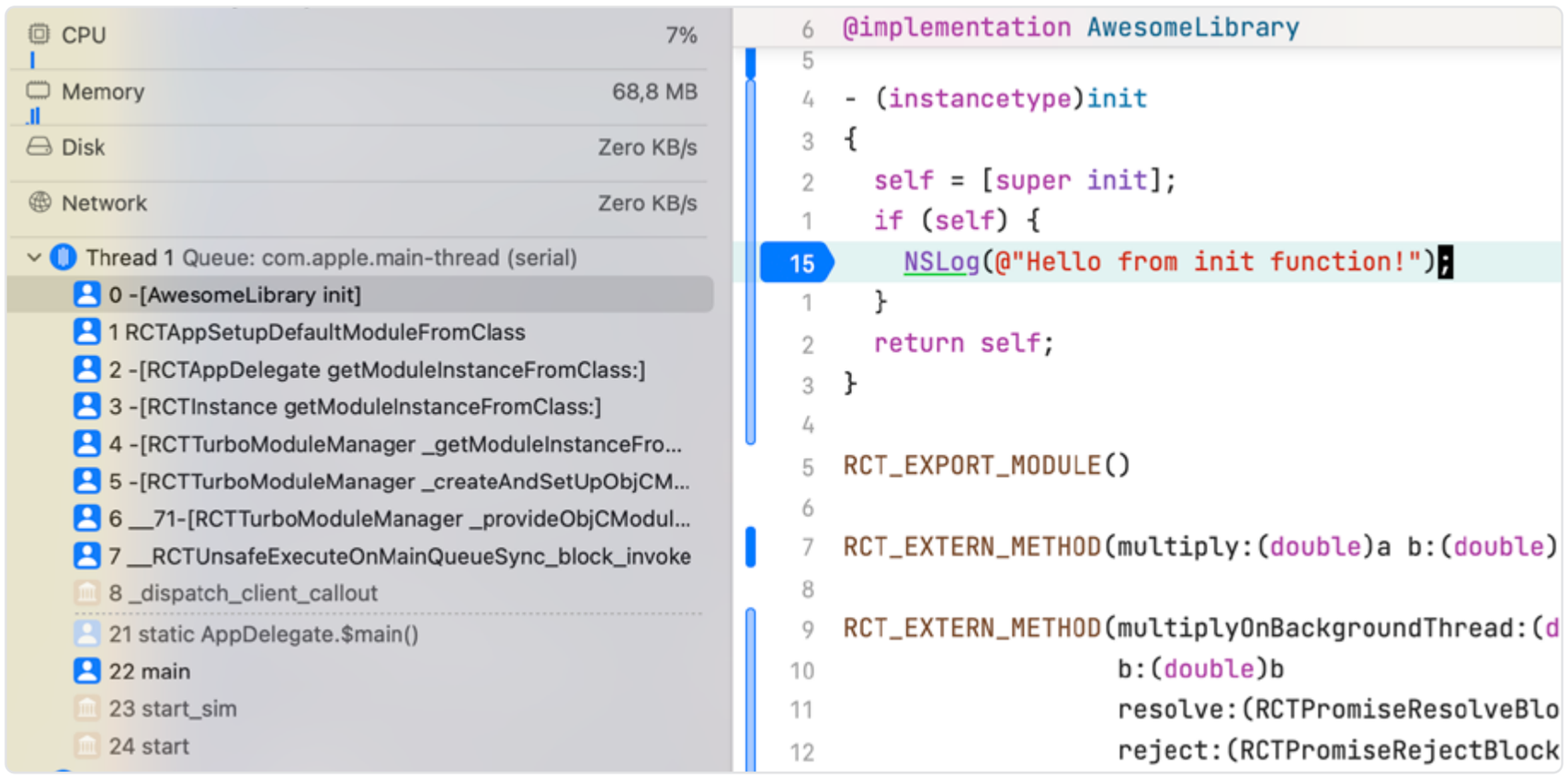
- Main thread / UI thread—whether it's a React Native or plain native app, this thread handles UI operations and maintains a responsive user experience.
- JavaScript thread—as the name suggests, it's used (although not exclusively) to execute JavaScript code).
- Native modules thread—a shared pool allocated by React Native specifically for native modules.

In addition, you, React Native's renderer, and the third-party modules can create additional threads to handle tasks in the background. You can learn more about this topic in the Make Your Native Modules Faster chapter.

Turbo Modules

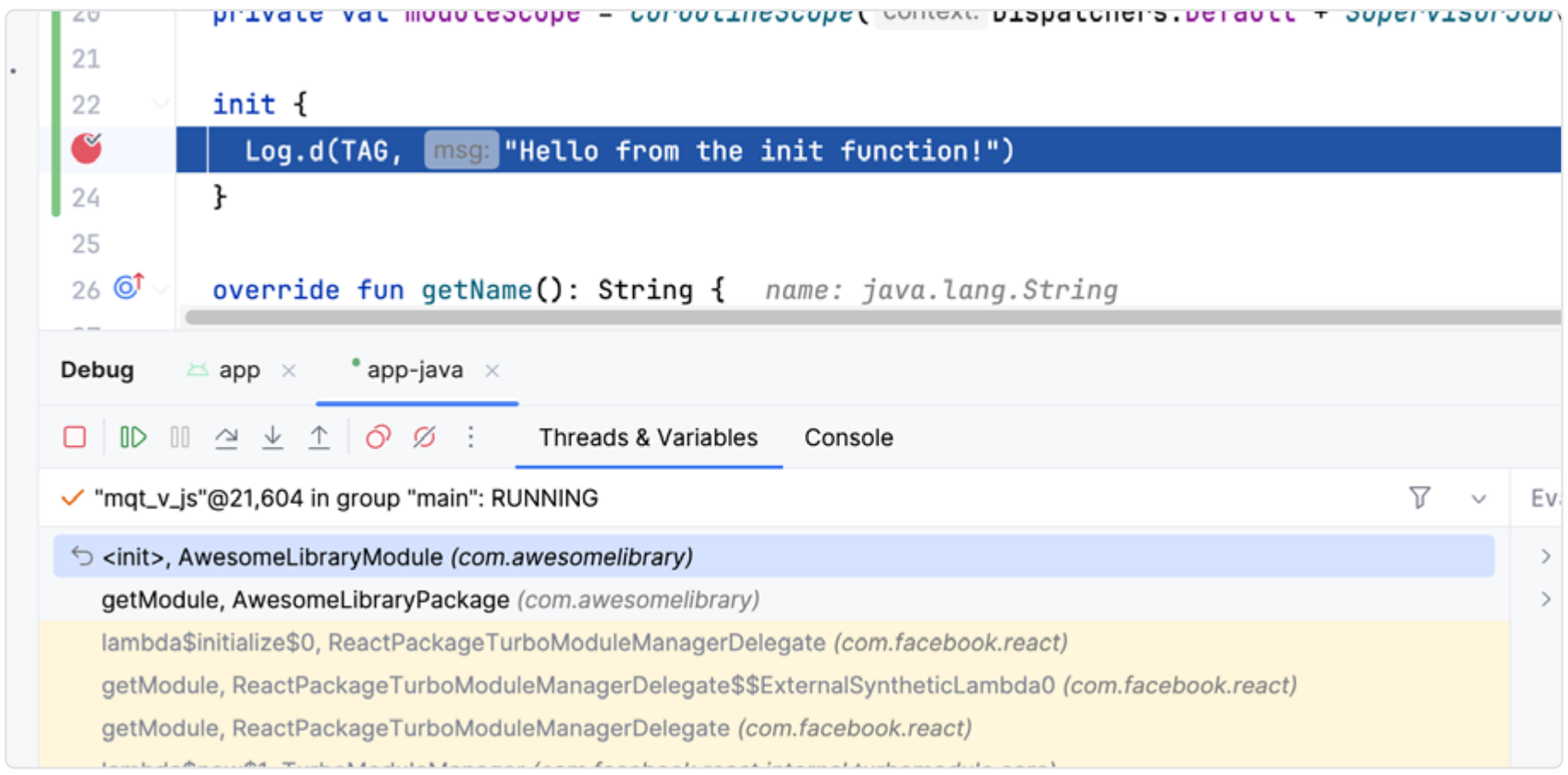
To better understand the threading model of Turbo Modules, let's examine its lifecycle by checking how and on which thread a given method is called.

First, let's check where the **`init`** function will be called on iOS:



A screenshot from Xcode debugger calling `init` on the main thread

and on Android:



A screenshot from Android Studio debugger calling `init` on the JavaScript thread

There is a notable discrepancy between platforms. On Android, the **`init`** method was called on the JavaScript thread, called **`mqt_v_js`**, while iOS used the main thread. The difference comes from an assumption made in React Native about modules overriding the **`init`** method. Whenever we override the init function in an iOS app, React Native assumes that we may access UIKit; therefore, it calls the function on the main queue (UIKit is not thread-safe).

```
/**
 * If a module overrides `init` then we must assume that it expects to be initialized on the main thread, because it
 * may need to access UIKit.
 */
const BOOL hasCustomInit = [moduleClass instanceMethodForSelector:@selector(init)] != objectInitMethod;
```

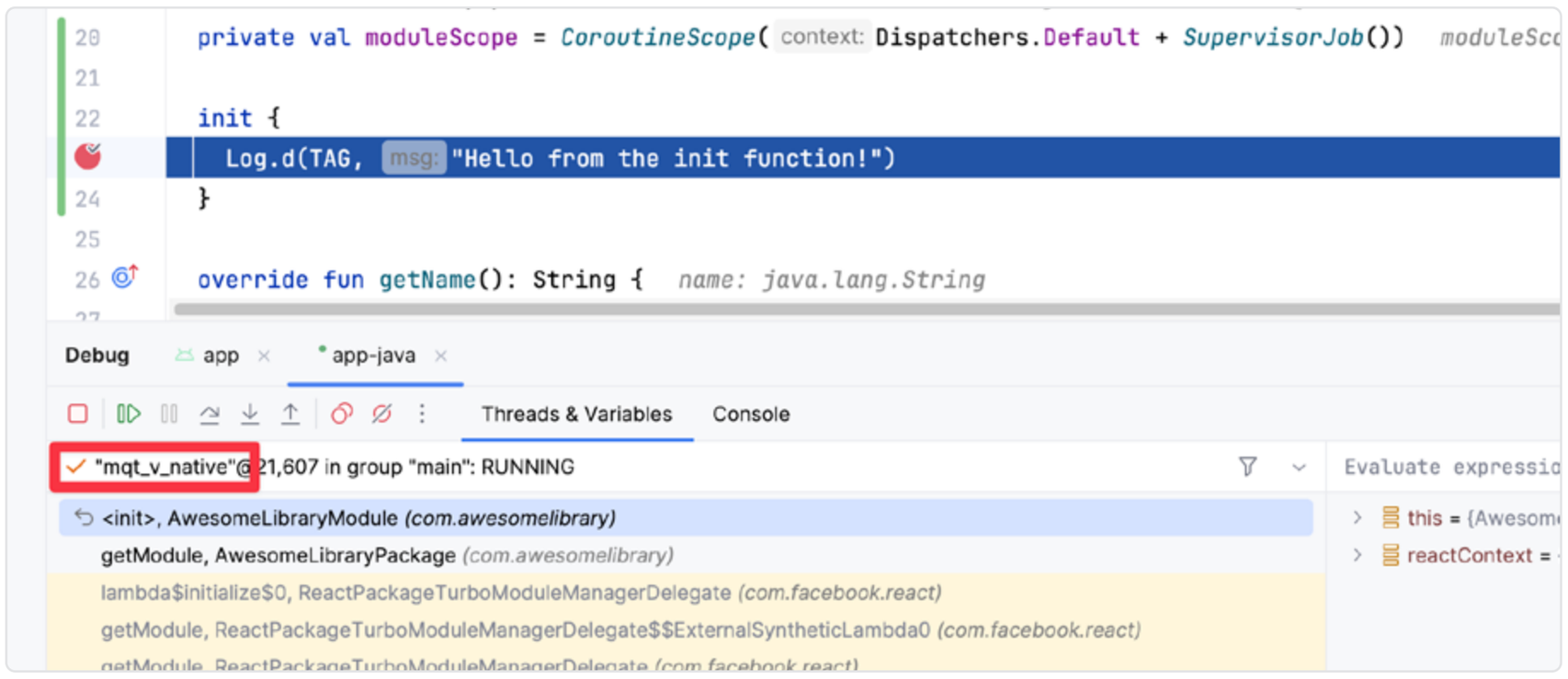
React Native source code explaining running init on the main thread

> After removing this assumption, the module is initialized on the JavaScript thread, which follows the same behavior as on Android. Altering React Native internals requires building it from the source. While this is not recommended for production apps, it can be a valuable learning experience in a development environment.

Next, let's see if opting out of lazy initialization changes the thread our module is using. This feature is available only on Android. To opt in to eager loading, open your library's package file and change the fourth parameter **`needsEagerInit`** to **`true`**:

```
class AwesomeLibraryPackage : BaseReactPackage() {
  // Other code

  override fun getReactModuleInfoProvider(): ReactModuleInfoProvider
{
    return ReactModuleInfoProvider {
      val moduleInfos: MutableMap<String, ReactModuleInfo> =
HashMap()
      moduleInfos[AwesomeLibraryModule.NAME] = ReactModuleInfo(
        AwesomeLibraryModule.NAME,
        AwesomeLibraryModule.NAME,
        false,  // canOverrideExistingModule
        true,   // needsEagerInit <- Change this to true
        false,  // isCxxModule
        true    // isTurboModule
      )
      moduleInfos
    }
  }
}
```

Let's see what happens after changing the flag and re-building the app:

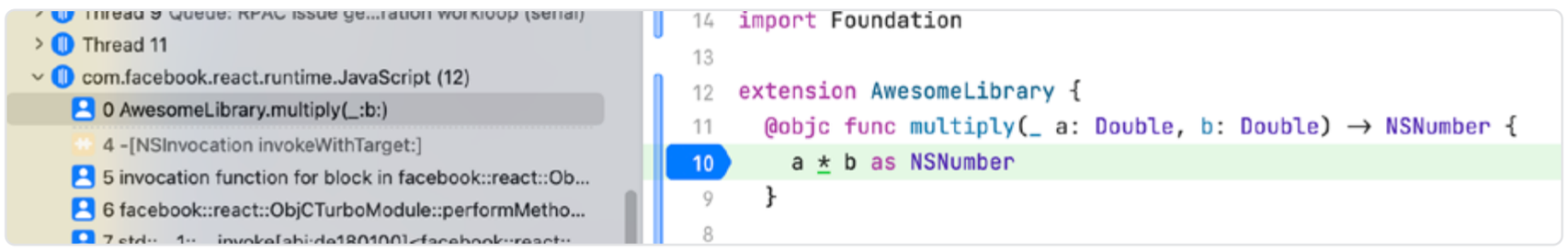


A screenshot from Android Studio debugger calling `init` on the Turbo Modules thread

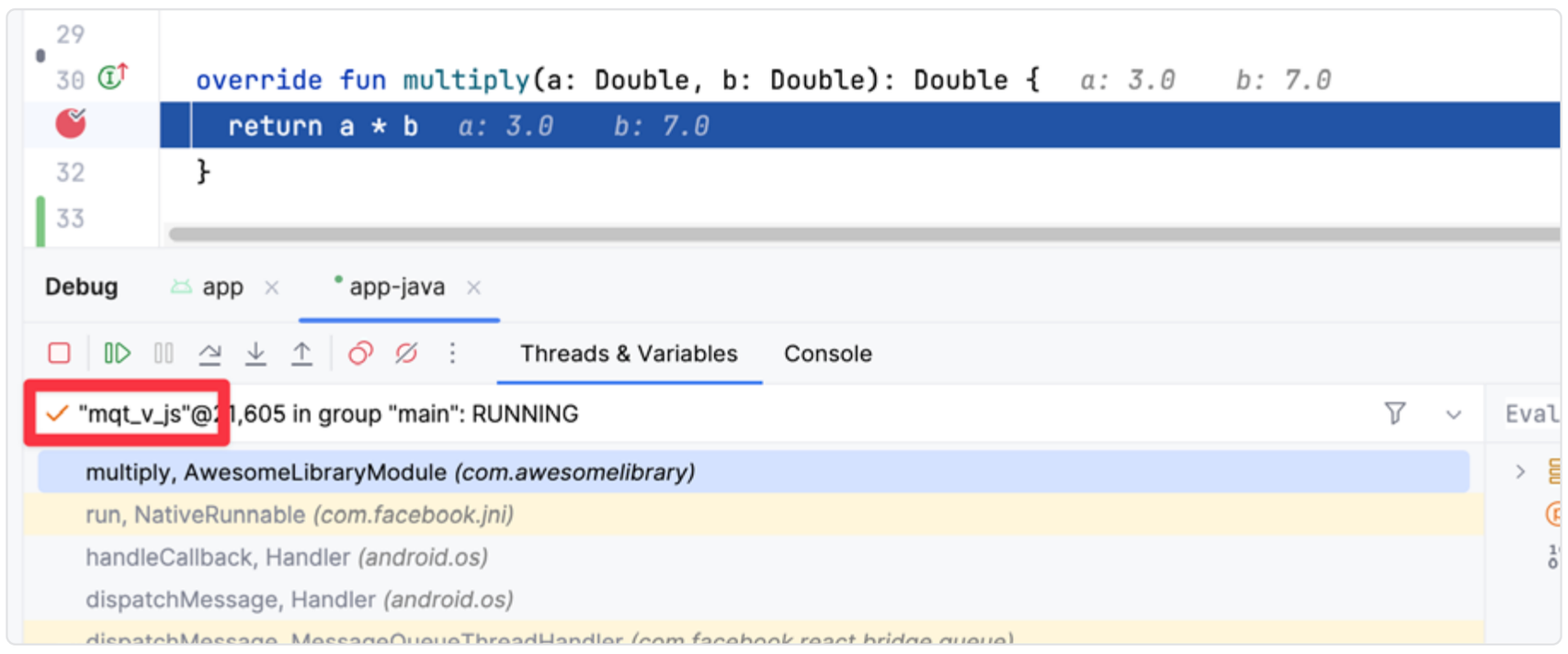
As expected, we are now on a different thread. It's **`mqt_v_native`**—a separate thread for Turbo Modules. It's worth keeping this in mind if you plan to access a JavaScript instance here. In that case, you would need to use JavaScript **`CallInvoker`** to schedule a function call on the JS thread.

Synchronous function calls

With initialization out of the way, let's see how React Native calls native methods. In this case, we can divide them into two categories: synchronous and asynchronous. First, let's try to see what happens if we call a synchronous method, such as **`mutliply`**:



A screenshot from Xcode debugger calling `multiply` on the JavaScript thread



A screenshot from Android Studio debugger calling `multiply` on the JavaScript thread

It gets called on the JavaScript thread. If you think about it, it makes sense, as we want to stop the execution on that thread and wait until this native function returns its value. As you can see, you have to be extra careful with synchronous functions. If you left this innocent-looking function being called on a JavaScript thread by mistake

```
@objc func multiply(_ a: Double, b: Double) -> NSNumber {
    Thread.sleep(forTimeInterval: 20) // Go to sleep
    return a * b as NSNumber
}
```

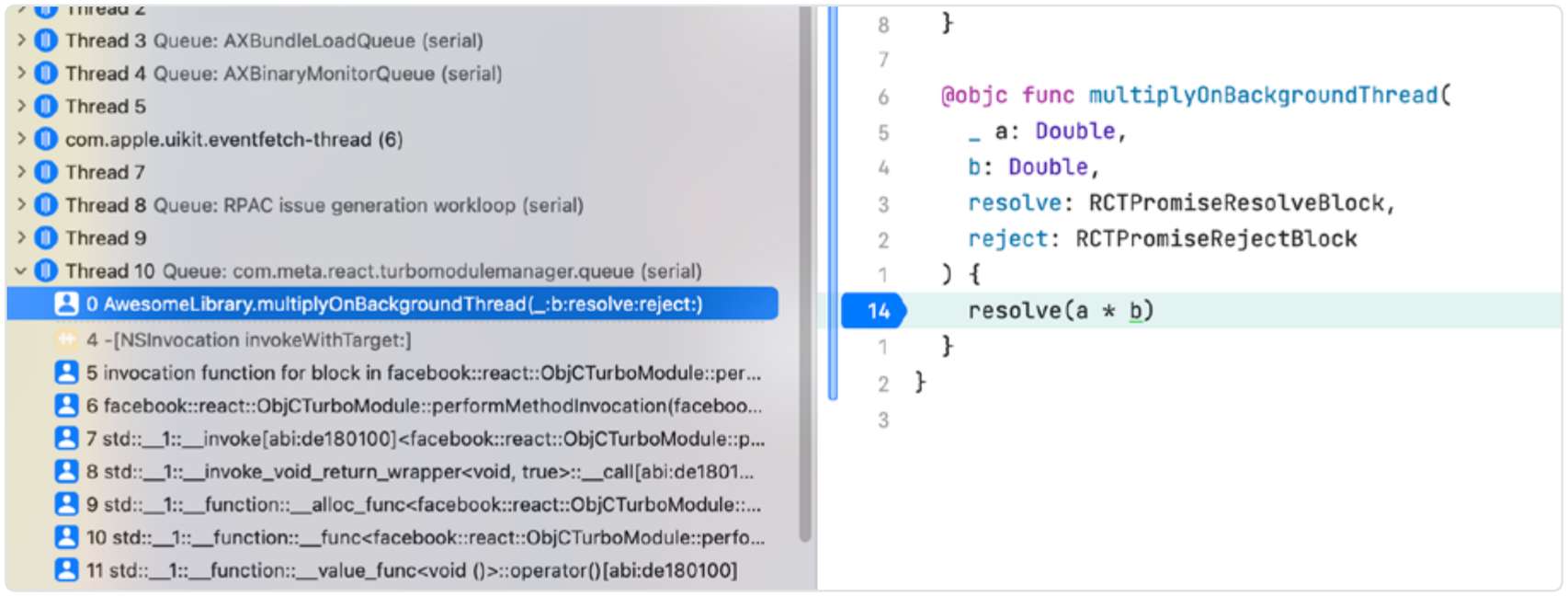
`Thread.sleep` doesn't look quite innocent, though

you would block all interactions on the JavaScript thread for 20 seconds, essentially freezing the whole app.

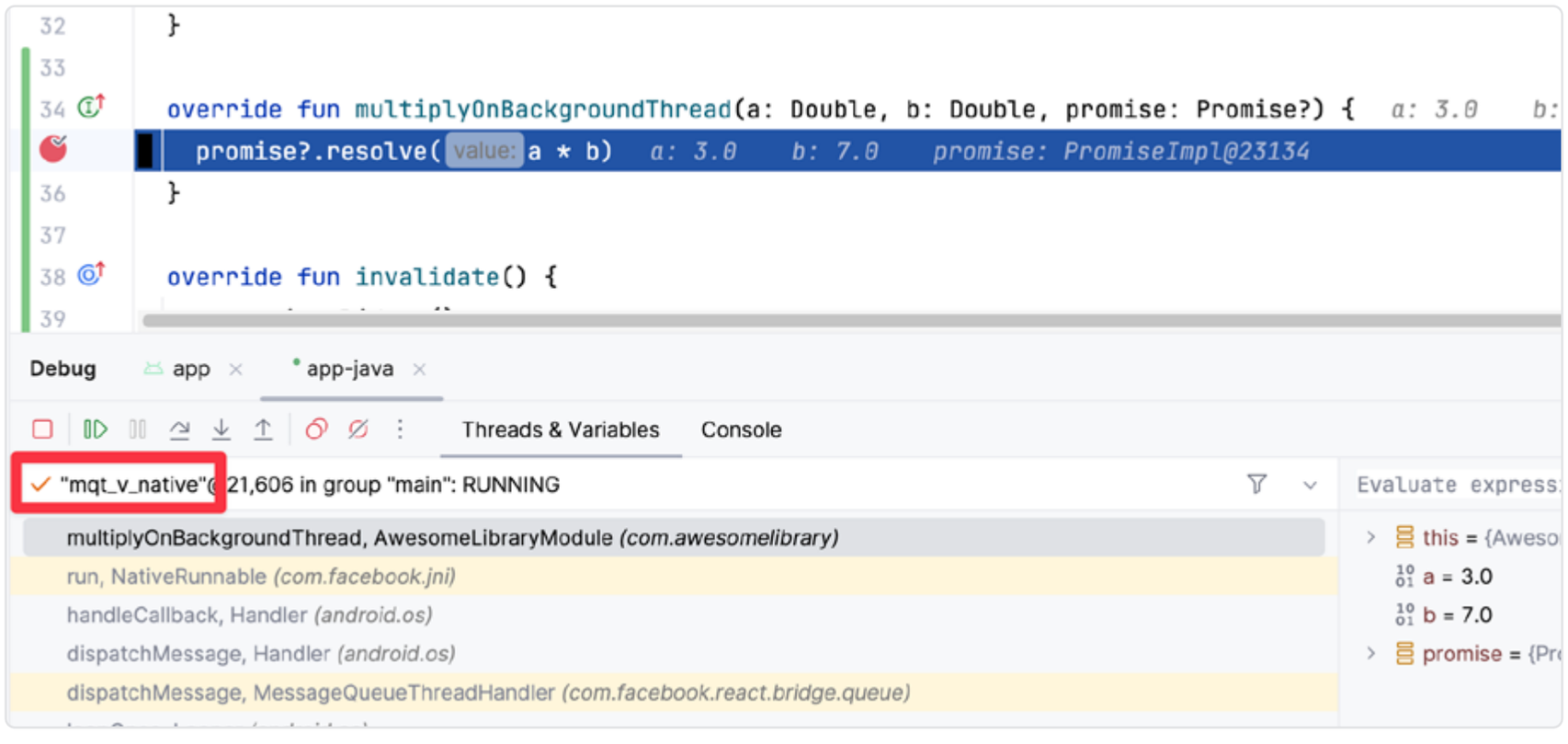
Asynchronous function calls

Now, let's see how asynchronous functions are called. Similarly to web context, when we **`await`** a function—let's say **`fetch`**—the call gets delegated to the browser's Web APIs (the native code layer), which handles this request. The request is handled outside of the main JavaScript execution thread, which in React Native is the native modules thread.

Let's see it in action for an async **`multiplyOnBackgroundThread`** function:



A screenshot from Xcode debugger calling `multiplyOnBackgroundThread` on the Turbo Modules thread

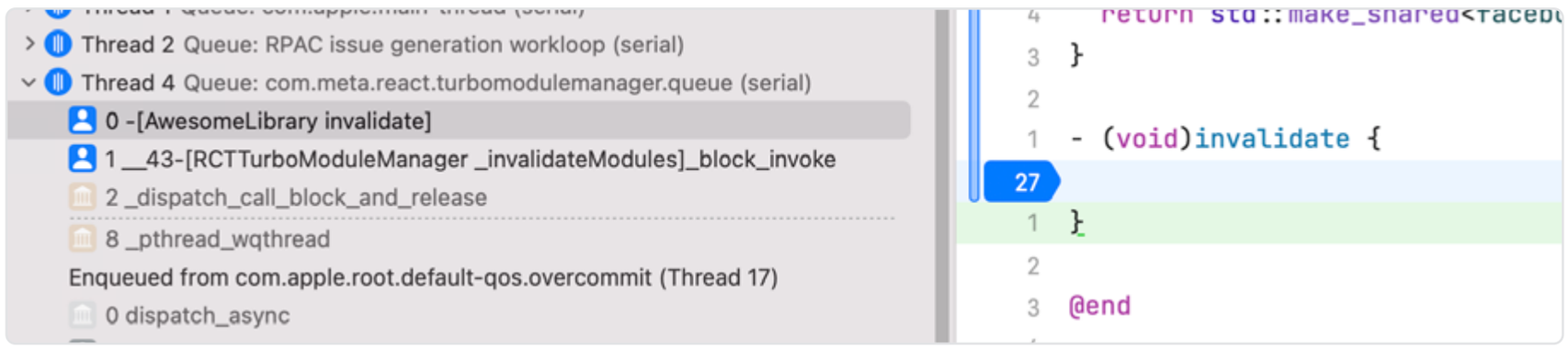


A screenshot from Android Studio debugger calling `multiplyOnBackgroundThread` on the Turbo Modules thread

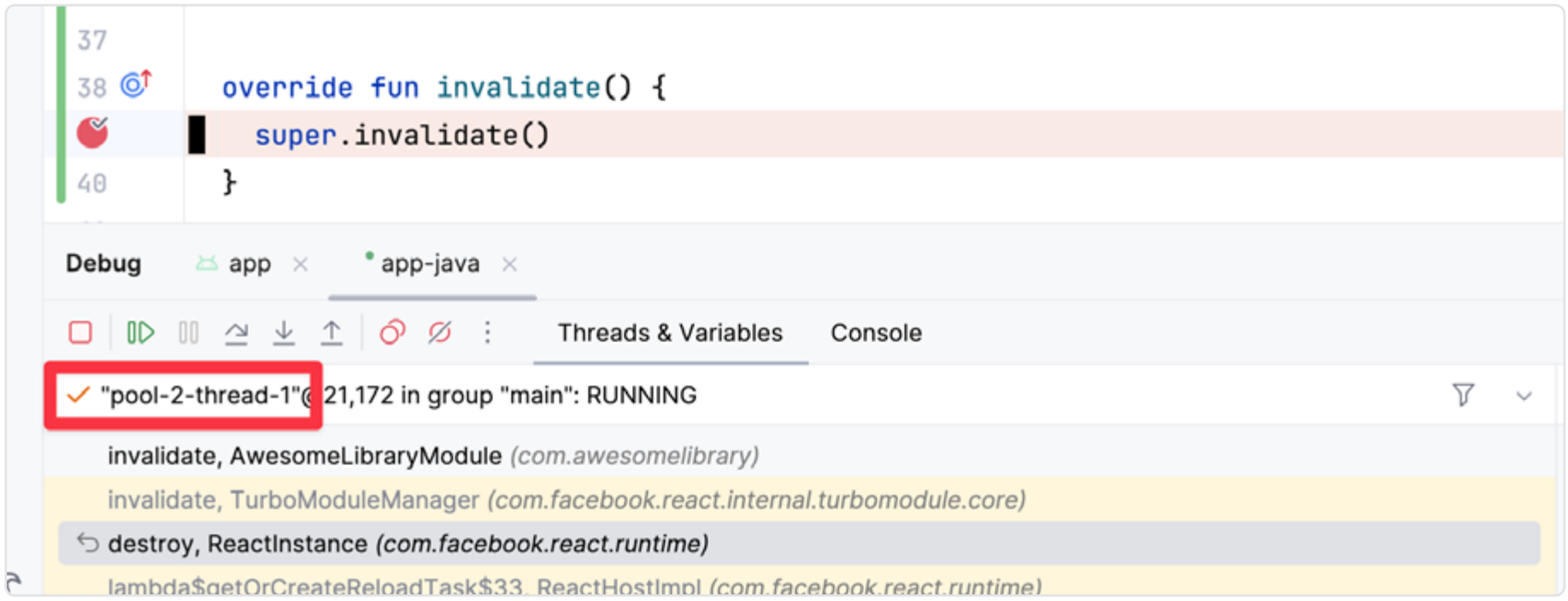
As you can see, React Native offloads the JavaScript thread for us and calls the function on the Turbo Modules thread. However, this thread might be busy as it is shared across all native modules. In this case, you can always create a new thread to mitigate the issue.

The last part is to check how modules are invalidated. In order to support invalidation on iOS, our module has to conform to **`RCTInvalidating`** protocol.

Invalidation happens when the React Native instance is torn down, for example, when we reload the Metro server.



A screenshot from Xcode debugger calling `invalidate` on the Turbo Modules thread



A screenshot from Android Studio debugger calling `invalidate` on `pool-2-thread-1` which is "ReactHost" Thread

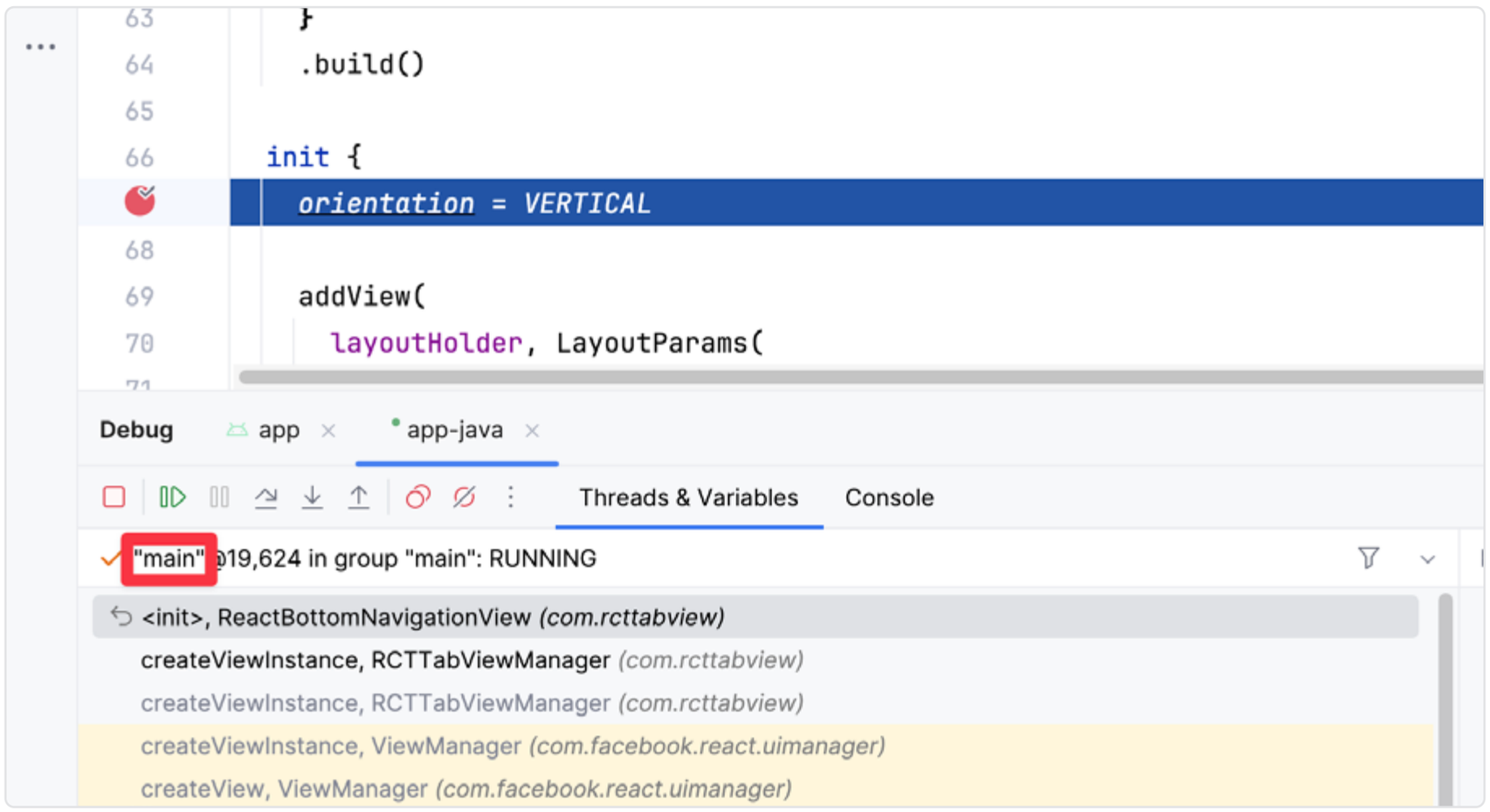
We have another small discrepancy between platforms. On iOS, the function is called on the Turbo Modules thread, and on Android, the thread doesn't have an explicit name; it's probably auto-generated. Going through the source code, you can find that it's a thread spawned by the ReactHost class. Now that you know all about threading in turbo modules, let's look at how Fabric views work.

Native Views

Understanding the threading model of the Native Views layer (internally called Fabric) is a lot easier. As mentioned before, the operating system—whether it's iOS or Android—expects to update its views on the main thread. You also learned that UIKit, an iOS UI Library, is not thread-safe; therefore, almost everything around creating and manipulating views happens on the main thread.

Let's start the exploration. First, let's check how the views are initialized:

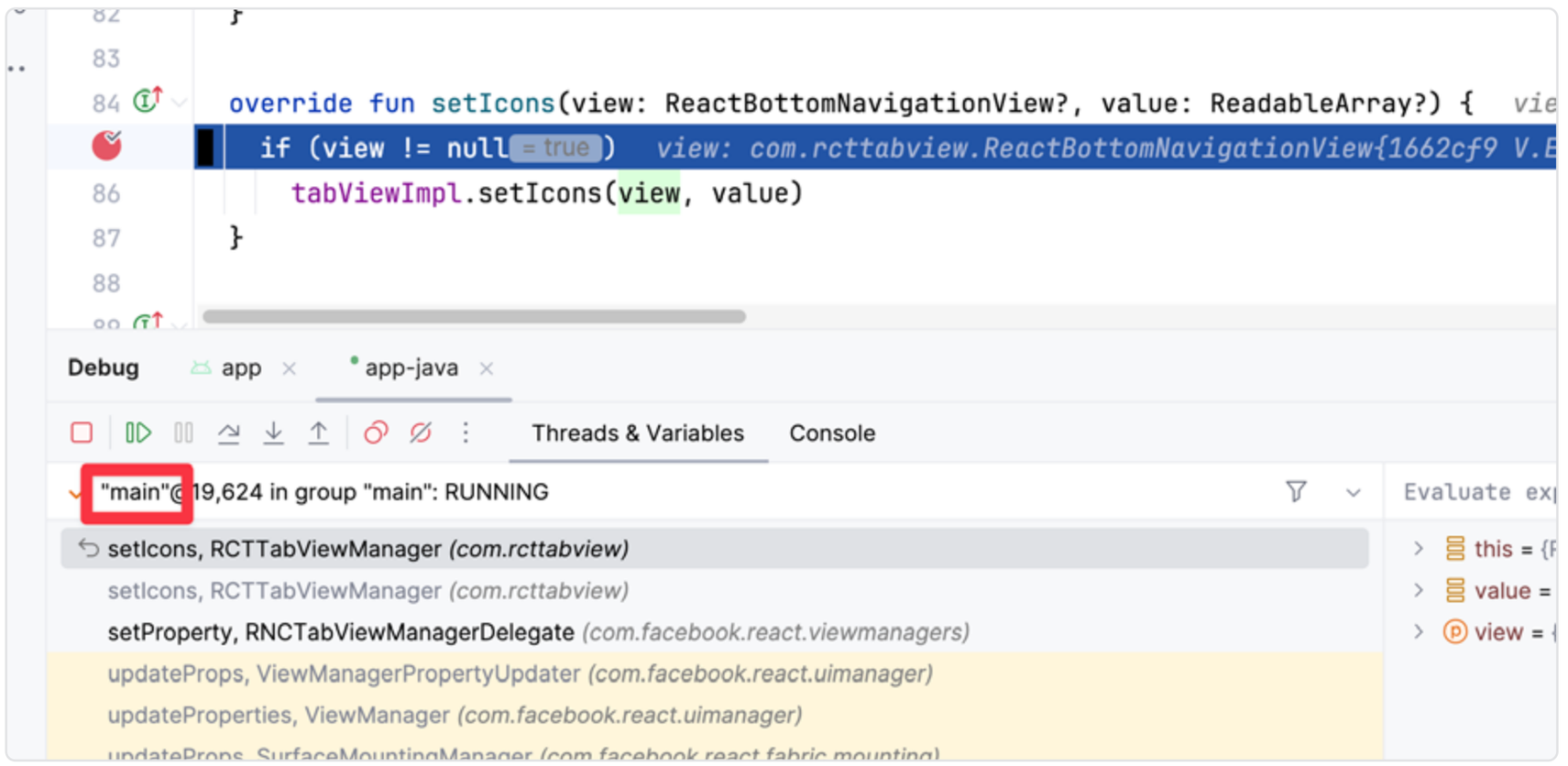
A screenshot from Xcode debugger calling Custom native component init method on the main thread



A screenshot from Android Studio debugger calling Custom native component init method on the main thread

As you can see, the view is initialized on the main thread on both platforms. Now, let's check which thread handles prop updates. On the JavaScript side, we always manipulate props on the JavaScript thread, but let's see what will happen on the native side:

A screenshot from Xcode debugger calling `updateProps` method on the main thread



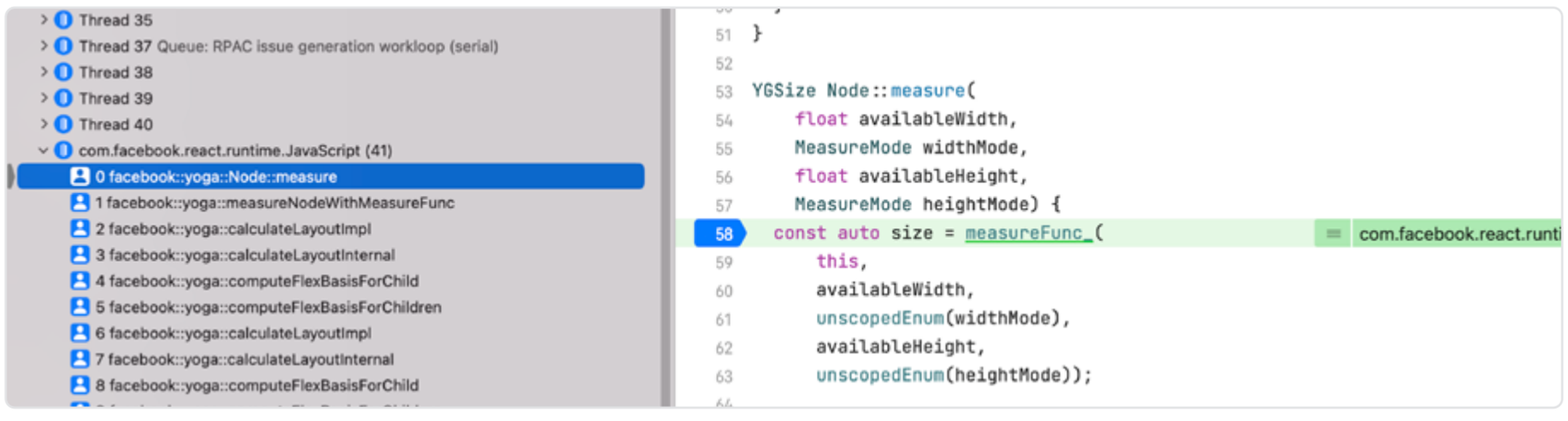
A screenshot from Android Studio debugger calling props setter on the main thread

We are on the main thread again! That's because React Native assumes that the function that updates props will directly manipulate native platform views with the help of Yoga.

Yoga is a vital part of React Native's rendering pipeline. It's a cross-platform layout engine used to calculate layout properties and provide a consistent layout on different platforms based on absolute positioning and the Flexbox layout model. The React Native renderer uses Yoga to calculate layout information for a React Shadow Tree, which is an internal representation of the React component tree in C++.

Yoga is not tied to React Native; it's often used in native apps for the same reasons. In some benchmarks, it's even faster than native view layout systems, such as auto-layout on iOS.

Operations related to the Yoga tree are executed on the JavaScript thread, as you can see below:



A screenshot from Xcode debugger calling `yoga::Node::measure` on the JavaScript thread

Now that you better understand the threading model—what is called on which thread and why—you'll be able to add advanced, thread-safe, and performant native functionalities to your React Native app.

BEST PRACTICE

USE VIEW FLATTENING

React component API is designed to be declarative and reusable through composition. We combine multiple components to create complex layouts, much like LEGO blocks. It works great on the web, where creating `<div/>` is quite cheap. React Native operates on native layout primitives, which are typically more expensive to process than web equivalents. To mitigate this issue, React Native introduced an optimization called "view flattening" to its core renderer. It simplifies the view hierarchy where possible to avoid extra memory pressure and CPU processing.

This kind of optimization was first introduced for the Android React Native renderer. However, as the project progressed with the New Architecture, essentially rewriting core parts to C++, it made sense to move this platform-specific optimization to the shared renderer. Thanks to that, view flattening is now possible on iOS too.

Without going into the details of how this works, the renderer identifies "layout-only" nodes. They only affect the layout of a `View` and don't render anything on the screen. Those elements can be flattened, leading to a depth reduction of the host elements tree. For an in-depth explanation of how the view flattening works, check out the official documentation.

Beware of issues with View Flattening

This optimization technique works great in most cases, but sometimes, we want our view to stay in the hierarchy. A common case for such a requirement is when you are working on a native UI component that accepts children.

```
<MyNativeComponent>
   <Child1/>
   <Child2/>
   <Child3/>
</MyNativeComponent>
```

MyNativeComponent with 3 child views

In the example above, the **`MyNativeComponent`** can expect to receive an array of 3 views on the native side. But what happens if view flattening decides that the **`Child1`**'s container view is "layout-only"? You can unexpectedly get all of the children of this view passed to the native side. This means that if **`Child1`** had 3 child views inside, you would get 5 children passed to your native component instead of 3. To illustrate that:

```
<MyNativeComponent>
  /* Child1 unexpectedly flattened to 3 Views */
   <View/>
   <View/>
   <View/>
   <Child2/>
   <Child3/>
</MyNativeComponent>
```

Greedy view flattening

This is unexpected behavior if you assume the number of JavaScript children will match the children's array on the native side. As you can see, it's not a safe assumption and may prompt you to validate your component logic. But sometimes, you really need to know the children count. To solve it, you can use the **`collapsable`** prop, which controls view flattening behavior. Setting it to false will prevent the view from being merged with others.

```
<MyNativeComponent>
   <Child1 collapsable={false} />
   <Child2 collapsable={false} />
   <Child3 collapsable={false} />
</MyNativeComponent>
```

MyNativeComponent with 3 child views with `collapsable` prop set to false

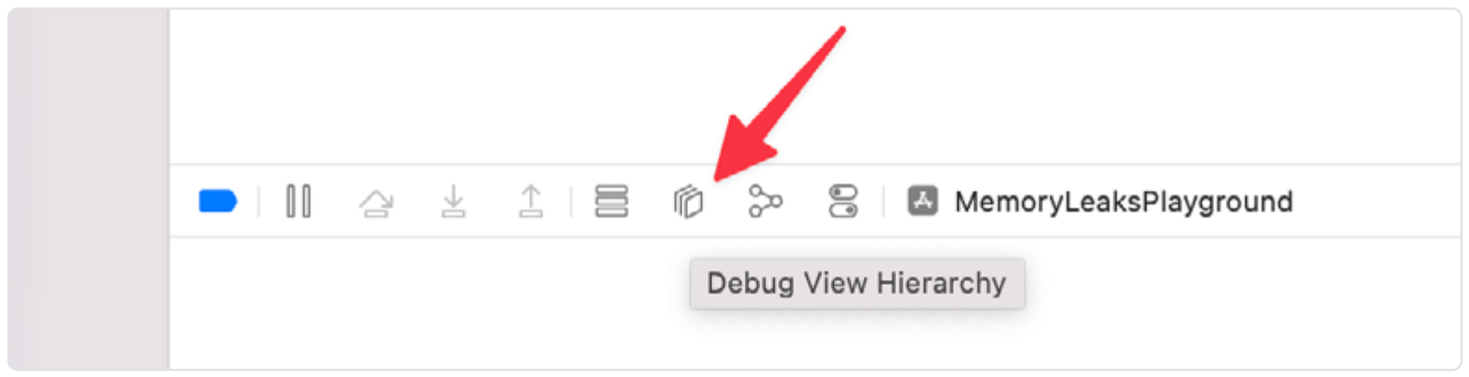
After applying this prop, we can be sure that we will always receive 3 child views in our native component.

Debugging the view hierarchy

While working on view flattening, you might find view hierarchy debugging useful. Thankfully, you can inspect it on both iOS and Android platforms with Xcode and Android Studio, respectively.

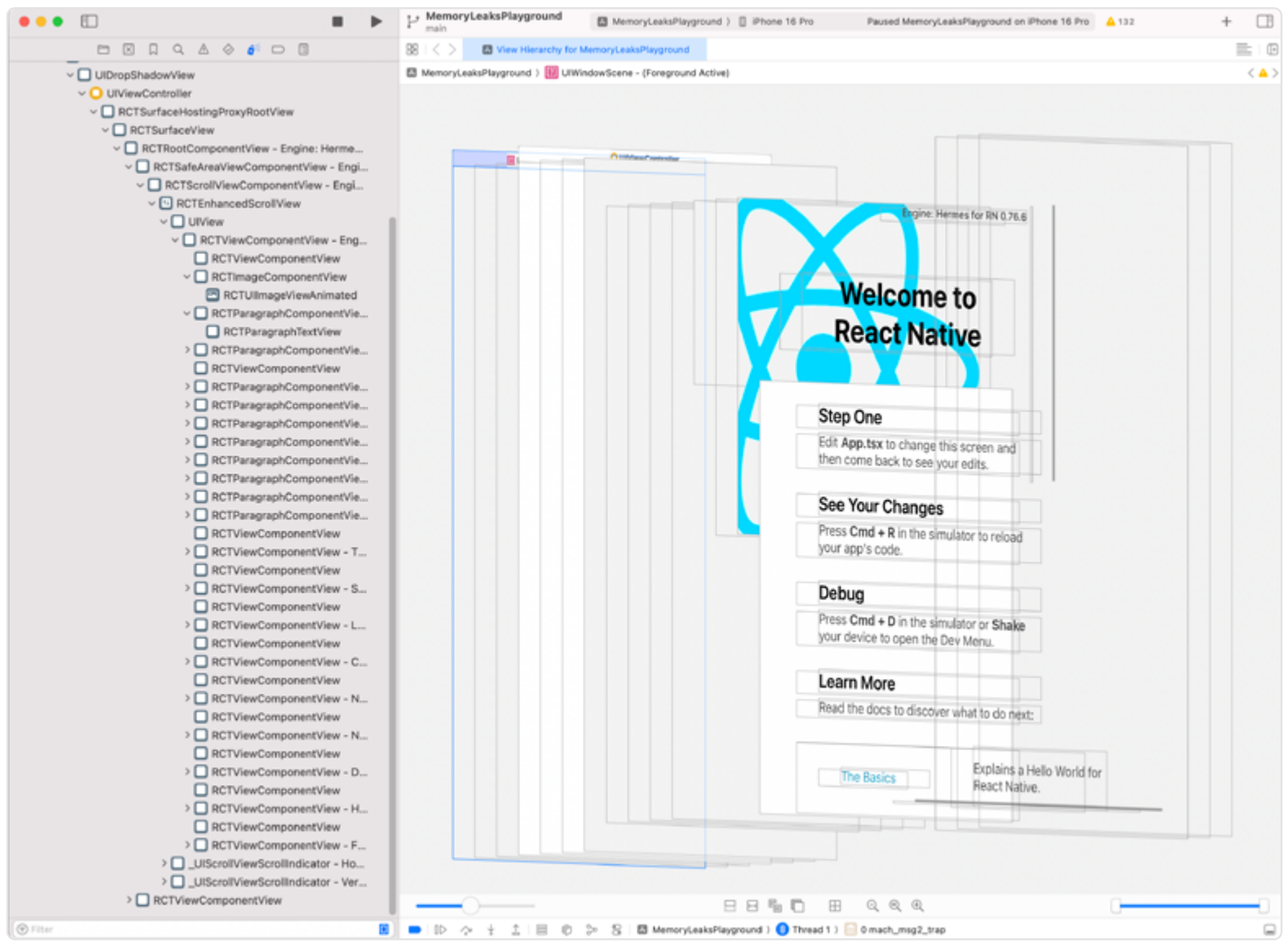
Xcode

When you run your app through Xcode, you'll notice a debugging toolbar which, on top of standard breakpoint debugging, offers Hierarchy inspector. You can launch it by clicking the "Debug View Hierarchy" button:



Debug View Hierarchy button in Xcode

This stops your app, as it was stopped on a breakpoint, but now you can inspect the native view hierarchy:

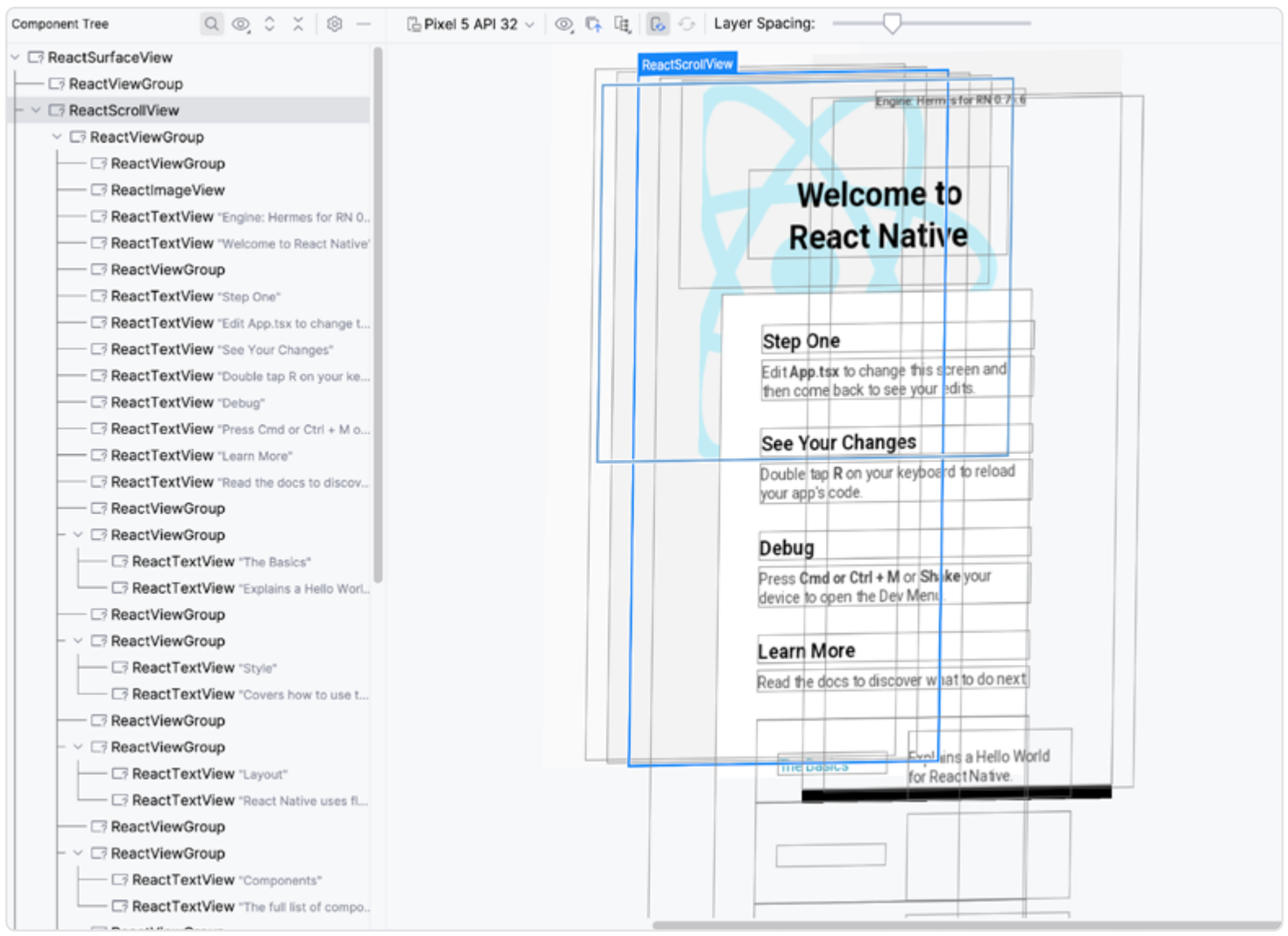


View hierarchy of a sample React Native app; it really is a native app!

Seeing a visual representation of the view hierarchy can be helpful when solving layout-related issues. If you look at the left sidebar, it lists the native views corresponding to JavaScript components, so a JavaScript `<View/>` is `RCTViewComponentView`, the same native building block as inside of fully native apps.

Android Studio

Same as with Xcode, we have a way to inspect the view hierarchy of our app; in Android, it's called "Layout Inspector". To launch it, navigate to the top menu bar and open **`View`** > **`Tool Windows`** > **`Layout Inspector`**.



Layout Inspector in Android Studio

Here, you can see a similar UI to what we have in Xcode. In the sidebar, there is a list of elements currently visible on the screen. You can notice that on this platform, **`<View/>`** from JavaScript corresponds to **`ReactViewGroup`** that holds **`ReactTextView`**s inside.

Being aware of how view flattening works is a useful skill. Most of the time, you shouldn't think about it as the idea behind it was to be fully transparent. Sometimes, however, you might run into issues where this optimization was misapplied, and the layout debugger should help you understand what's happening. And, if necessary, skip flattening of problematic views.

BEST PRACTICE

USE DEDICATED REACT NATIVE SDKS OVER WEB

One of the best things about React Native is that you can use all the good parts of the JavaScript ecosystem in mobile, tablet, and desktop apps. This means reusing some of your React components and managing the business logic state with your favorite library.

While React Native provides web-like functionality for compatibility with the web, you need to understand it's not the same environment. It has its own set of best practices, quick wins, and constraints.

Using a dedicated platform-specific version of the library

The fact that it's often (but not always) possible to run the same JavaScript in a mobile React Native app as in the browser doesn't mean you should do this every time. You must be vigilant, scan your dependencies for usability and keep checking whether they're still necessary or could be replaced with better platform-specific alternatives.

Internationalization polyfills

The `Intl` global object provides language-sensitive string comparison, number formatting, and date and time formatting. It's available in any modern browser and partially in the Hermes engine.

Here is an example of formatting a number to a different locales using `Intl`:

```
const number = 123456.789;
const germanFormat = new Intl.NumberFormat('de-DE');
console.log(germanFormat.format(number)); // 123.456,789
```

In React Native projects, you can often spot a bunch of `Intl` polyfill imports from the Format. JS project, providing JS-only implementation of ECMAScript Internationalization API:

```
import '@formatjs/intl-getcanonicallocales/polyfill';
import '@formatjs/intl-locale/polyfill';
import '@formatjs/intl-numberformat/polyfill';
import '@formatjs/intl-numberformat/locale-data/en';
import '@formatjs/intl-datetimeformat/polyfill';
import '@formatjs/intl-datetimeformat/locale-data/en';
import '@formatjs/intl-pluralrules/polyfill';
import '@formatjs/intl-pluralrules/locale-data/en';
import '@formatjs/intl-relativetimeformat/polyfill';
import '@formatjs/intl-relativetimeformat/locale-data/en';
import '@formatjs/intl-displaynames/polyfill';
```

Importing all possible Intl polyfills is not always necessary

The support for this API has been limited in Hermes, but fortunately, last year, the Hermes team announced they were working on improving it. A year later, some of those polyfills can be removed! At the time of writing this guide (January 2025), here are the available APIs:

Intl API	Hermes support
`Intl.Collator`	☑
`Intl.DateTimeFormat`	☑
`Intl.NumberFormat`	☑
`Intl.getCanonicalLocales()`	☑
`Intl.supportedValuesOf()`	☑
`Intl.ListFormat`	✗
`Intl.DisplayNames`	✗
`Intl.Locale`	✗
`Intl.RelativeTimeFormat`	✗
`Intl.Segmenter`	✗
`Intl.PluralRules`	✗

With that new knowledge, we can reduce the number of polyfills we import into our project:

```
-import '@formatjs/intl-getcanonicallocales/polyfill';
 import '@formatjs/intl-locale/polyfill';
-import '@formatjs/intl-numberformat/polyfill';
-import '@formatjs/intl-numberformat/locale-data/en';
-import '@formatjs/intl-datetimeformat/polyfill';
-import '@formatjs/intl-datetimeformat/locale-data/en';
 import '@formatjs/intl-pluralrules/polyfill';
 import '@formatjs/intl-pluralrules/locale-data/en';
 import '@formatjs/intl-relativetimeformat/polyfill';
 import '@formatjs/intl-relativetimeformat/locale-data/en';
 import '@formatjs/intl-displaynames/polyfill';
```

Importing all possible Intl polyfills is not always necessary

Removing these polyfills alone shaves off over 430 kB from our JS bundle. This code is loaded eagerly at our app's entry point, so it can potentially improve the app's TTI—all by removing unnecessary code.

Crypto libraries

Another example worth considering is replacing libraries related to heavy computation with their native substitutes. One example is a JavaScript implementation of the Node.js `crypto` library called `crypto-js`, which can be replaced with `react-native-quick-crypto` from Margelo. This replacement can yield results up to 58x faster due to using C++ directly.

This is extremely important for projects that rely on random numbers and need to use cryptographically secure pseudorandom number generator (CSPRNG), for example, to generate a Web3 Wallet seed. This function can't be implemented relying on JavaScript's `Math.random()` because it can't guarantee that the result will contain enough entropy to be considered secure.

You can read more about this in this blog post about increasing speed and security with native crypto libraries.

Use native navigation

Most apps need a navigation solution; one of the most popular is React Navigation. It's a robust library allowing for a mix of JS and native navigators, depending on your design needs and performance requirements. While setting up your navigation structure, you might wonder which stack navigator to choose: JS Stack with ultimate flexibility or Native Stack with less flexibility but better performance and native look. It's a similar story if you use tab navigator: should you go with JS Tabs and Native Tabs, which have similar trade-offs as stack navigator?

There are multiple benefits to choosing native-based navigators. First and foremost, they unload the work from the JavaScript thread, making your app use less memory and be smoother overall. There is also a really important factor: native feel. Both Native Stack and Native Tabs give your app extra-level polish by using native platform primitives used by other apps on this platform.

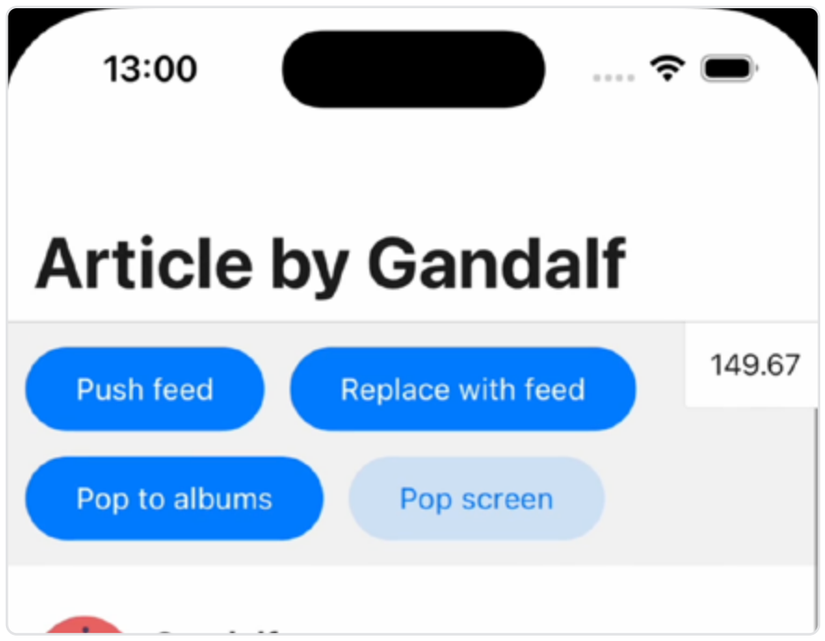
Native Stack Navigator

To use the native stack navigator from `react-native-screens` project with React Navigation, all you need to do is install and use the `@react-navigation/native-stack` package like this:

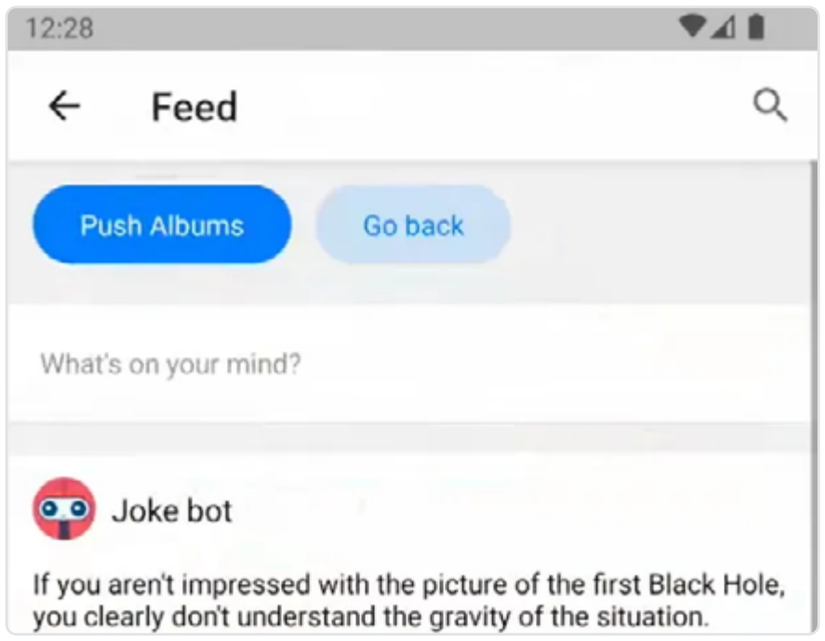
```
import { createNativeStackNavigator } from '@react-navigation/
native-stack';

const MyStack = createNativeStackNavigator({
  screens: {
    Home: HomeScreen,
    Profile: ProfileScreen,
  },
});
```

Initializing native stack navigator

Its API is very similar to the JavaScript stack navigator, so migrating it is often not complicated.



Large iOS header from `react-native-screens` Stack Navigator



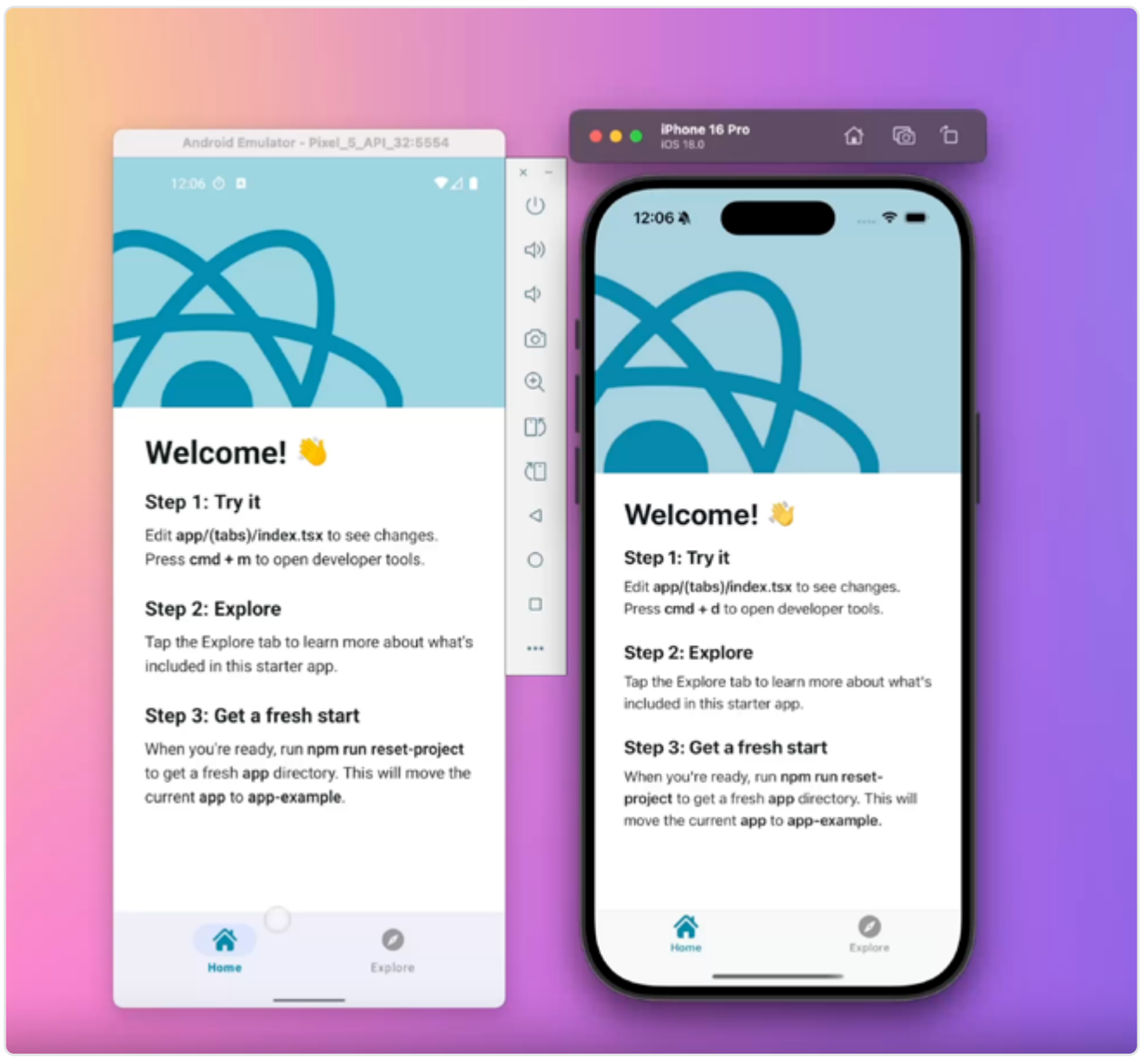
Native Android header with back button from `react-native-screens` Stack Navigator

Native Tabs Navigator

Similarly, instead of using a built-in JS navigator available in React Navigation, you can use the **`react-native-bottom-tabs`** project, which provides a mostly compatible API.

Similarly to Native Stack, you can replace your JavaScript-based navigation and use the same API:

```
import { createNativeBottomTabNavigator } from '@bottom-tabs/react-
navigation';

const MyTabs = createNativeBottomTabNavigator({
  screens: {
    Home: HomeScreen,
    Profile: ProfileScreen,
  },
});
```



Native bottom tabs from `react-native-bottom-tabs`

Prefer using libraries offering native components

Libraries exposing native components should be your first choice. React Native has a wide community of developers working on open-source libraries that expose native components.

Let's go over a few of those:

- React Native Screens—foundation for React Navigation's Native Stack.
- Zeego—beautiful, native menus for React Native + Web, inspired by Radix UI.
- React Native Slider—slider using native platform primitives.
- React Native Date Picker—date picker using native platform primitives.

And there are many more!

Replacing infinitely flexible JavaScript views with platform-native counterparts may not always be possible due to your app's design requirements. If a better, more performant UX alternative exists but requires you to adjust your design choices slightly, you may need to stand up for your users. Great UX is about the dialog between designers' creativity and what's physically possible from an engineering standpoint. Don't be afraid to voice your concerns. After all, we do it all for our users.

BEST PRACTICE

MAKE YOUR NATIVE MODULES FASTER

Native modules unlock endless possibilities in terms of things you can build in a React Native app. When JavaScript is not enough, you can drop into a lower-level language like Swift and Kotlin, which is native to the platform your app runs on. If you need to go even closer to the metal to unlock better performance or code reuse, you can also use C++. Let's review some of the best practices you can incorporate while building your next native module.

Quickly scaffold boilerplate code

Creating a new React Native library involves setting up initial boilerplate code that needs to handle legacy and new architecture implementations, codegen from TypeScript or Flow typings, and infrastructure to run, test, and publish code to npm. Thankfully, there is a great tool that helps you quickly scaffold a new library with everything set up for you—React Native Builder Bob.

To use it, run the `create-react-native-library` command in your terminal window:

```
> npx create-react-native-library@latest library-name
```

It will prompt you for necessary information, such as the package name or whether you build a Turbo module, Fabric view, or JavaScript library. You can also choose which languages you prefer to code in.

```
⌥⌘4                    ~/Developer/tmp

→  npx create-react-native-library@latest AwesomeLibrary
✔ What is the name of the npm package? … awesome-library
✔ What is the description for the package? … Awesome Library
✔ What is the name of package author? … Michał Pierzchała
✔ What is the email address for the package author? … my-email@gmail.com
✔ What is the URL for the package author? … https://github.com/thymikee
✔ What is the URL for the repository? … https://github.com/thymikee/awesome-library
? What type of library do you want to develop? › - Use arrow-keys. Return to submit.
❯   Turbo module - integration for native APIs to JS
    Nitro module
    Fabric view
    Legacy Native module
    Legacy Native view
    JavaScript library
```

A screenshot of `create-react-native-library` initialization flow

After answering a few questions, you will be up and running with your new native module that's ready for publishing to npm as a third-party dependency to use in consumer apps. However, making our code available in the public registry is not always what we want. Sometimes, we just want to spin out a custom module specific to our project. Bob makes it possible with the local modules functionality by passing the **`--local`** flag to the "create" command.

Now, let's explore how to use modern languages for your native modules.

Use modern languages: Swift and Kotlin

Android and iOS have started with different programming languages than they are using today. For Android, it was Java, and for iOS, it was Objective-C.

The successor of Java is Kotlin, created by JetBrains, which took the community by storm and, in 2019, was officially announced by Google as a preferred language for building Android apps. It's fully interoperable with its predecessor. Except for a few configuration changes, you don't need to do anything special to use Kotlin instead of Java. In fact, some parts of React Native were rewritten to Kotlin over the last few years, so a library generated by Builder Bob already uses Kotlin.

On the iOS side, it's similar but a bit different. Swift originated from within Apple as a next-generation proprietary language to replace Objective-C in the long run. In 2014, the Swift toolchain was released to iOS developers through official Xcode support and soon after made it open-source, governed by Apple and Swift community. Swift can easily be used in Objective-C codebases and vice versa, allowing incremental migrations. Ever since, together with SwiftUI, it's a preferred language for writing iOS apps. The tricky part is its story around C++ interoperability. This is important because, with New Architecture, React Native went all in on C++ to ship its core cross-platform functionality. There are, however, ways to mitigate that.

Internally, React Native iOS support still heavily depends on Objective-C and its C++ interoperability named Objective-C++. Swift, on the other hand, has a very recent and still experimental interoperability with C++ but doesn't work the same way as the previous one. What we can do in our React Native libraries is to create a small wrapper to bind our calls from Objective-C to Swift. It's quite cumbersome, but it works. Let's see how we can do that.

After creating the module, first enable Swift support by modifying your library Podspec file that comes from CocoaPods:

```
- s.source_files = "ios/**/*.{h,m,mm,cpp}"
+ s.source_files = "ios/**/*.{h,m,mm,cpp,swift}"
```

Changes in the Podspec necessary for CocoaPods to understand Swift as a source to link

Next, create a new Swift file from within Xcode. Make sure to place it inside your module's iOS folder. Click "yes" when asked if you want to create a Bridging Header.

Next, add these changes inside of your library header file:

```
#import <Foundation/Foundation.h>

+ #if __cplusplus
#import "ReactCodegen/RNAwesomeLibrarySpec/RNAwesomeLibrarySpec.h"
+ #endif


@interface AwesomeLibrary : NSObject
+                                        #if __cplusplus
                                        <NativeAwesomeLibrarySpec>
+                                       #endif

@end
```

A diff of changes necessary for Swift to not fail on unsupported C++ types.

Swift doesn't understand C++ types, so we use **`#if __cplusplus`** macro to import the spec file only if **`__cplusplus`** is defined (which is not for Swift). Next, inside your Bridging Header, import your main library header file:

```
+ #import "AwesomeLibrary.h"
```

Now, we can create an **`extension`** to our AwesomeLibrary class to make it available to Objective-C through **`@objc`** attribute:

```
import Foundation

extension AwesomeLibrary {
  @objc func multiply(_ a: Double, b: Double) -> NSNumber {
    a * b as NSNumber
  }
}
```

AwesomeLibrary extension with `multiply` method exposed to Objective-C

What's left is to use the method we just implemented. To do so, navigate to your library `.mm` file (the one handling Objective-C++ syntax) and use `RCT_EXTERN_METHOD()`:

```
#import "AwesomeLibrary.h"

#if __has_include("awesome_library/awesome_library-Swift.h")
#import "awesome_library/awesome_library-Swift.h"
#else
#import "awesome_library-Swift.h"
#endif

@implementation AwesomeLibrary

RCT_EXPORT_MODULE()

RCT_EXTERN_METHOD(multiply:(double)a b:(double)b);

// rest of the module

@end
```

Let's break down what happens here. First, we import our module. Xcode generates a header file called `<library-name>-Swift.h`. It's prefixed with the library name when we are running in static/dynamic frameworks mode, which is why we check if this import exists. If it doesn't, we fall back to a normal one.

Next, we use `RCT_EXTERN_METHOD`, which remaps the Objective-C function to our Swift call. That's all we need to use Swift inside our module. There are ongoing efforts to support Swift by default, so hopefully, this boilerplate won't be necessary in the future.

Leverage background threads

In a browser context, we typically deal with one thread. We can, of course, offload some JavaScript work to a web worker, but it's a bit different. In React Native, we can leverage the power of devices in the user's hands. By default, synchronous Turbo Module methods get called on the JavaScript thread, as you can see below on iOS:

A screenshot from Xcode stopped on a breakpoint

The same happens on Android:

A screenshot from Android Studio stopped on a breakpoint

> The deep dive into threading in a React Native app is covered in the Understand the Threading Model of Turbo Modules and Fabric chapter.

Now, we want to use that multithreading and schedule some work to be done on a background thread. Let's introduce a new asynchronous method `multiplyOnBackgroundThread`. To keep this guide concise, we will skip the details on adding a new method and focus on offloading code to a background thread.

Let's start with iOS. To run our code on a background thread, we can use a higher-level concept of queues, available in the form of `DispatchQueue` API, which manages the scheduling and execution of tasks, ensuring they're not run on the main thread. The code available in the `DispatchQueue.global()` block will be executed once the global concurrent queue for background work is available. Inside the block, we can call the `resolve()` method to notify our JS function about the result.

```
@objc func multiplyOnBackgroundThread(
    _ a: Double,
    b: Double,
    resolve: @escaping RCTPromiseResolveBlock,
    reject: RCTPromiseRejectBlock
  ) {
    DispatchQueue.global().async {
      resolve(a * b)
    }
  }
```

Example use of `DispatchQueue.global()` in Swift

Let's see how to achieve the same thing on Android. Much like with Swift, we won't access a background thread directly. Instead, we'll use coroutines. They are a part of Kotlin's concurrency framework and provide a way to write asynchronous code that is both readable and maintainable. It's worth noting that coroutines are not bound to a physical thread—thousands of coroutines can run on a small number of threads.

To create a coroutine, we'll use the **`CoroutineScope`** class and instantiate it in the **`moduleScope`** variable:

```
@ReactModule(name = AwesomeLibraryModule.NAME)
class AwesomeLibraryModule(reactContext: ReactApplicationContext) :
  NativeAwesomeLibrarySpec(reactContext) {
  // Other methods..
+  private val moduleScope = CoroutineScope(Dispatchers.Default +
SupervisorJob())

+  override fun invalidate() {
+    super.invalidate()
+    moduleScope.cancel()
+  }
}
```

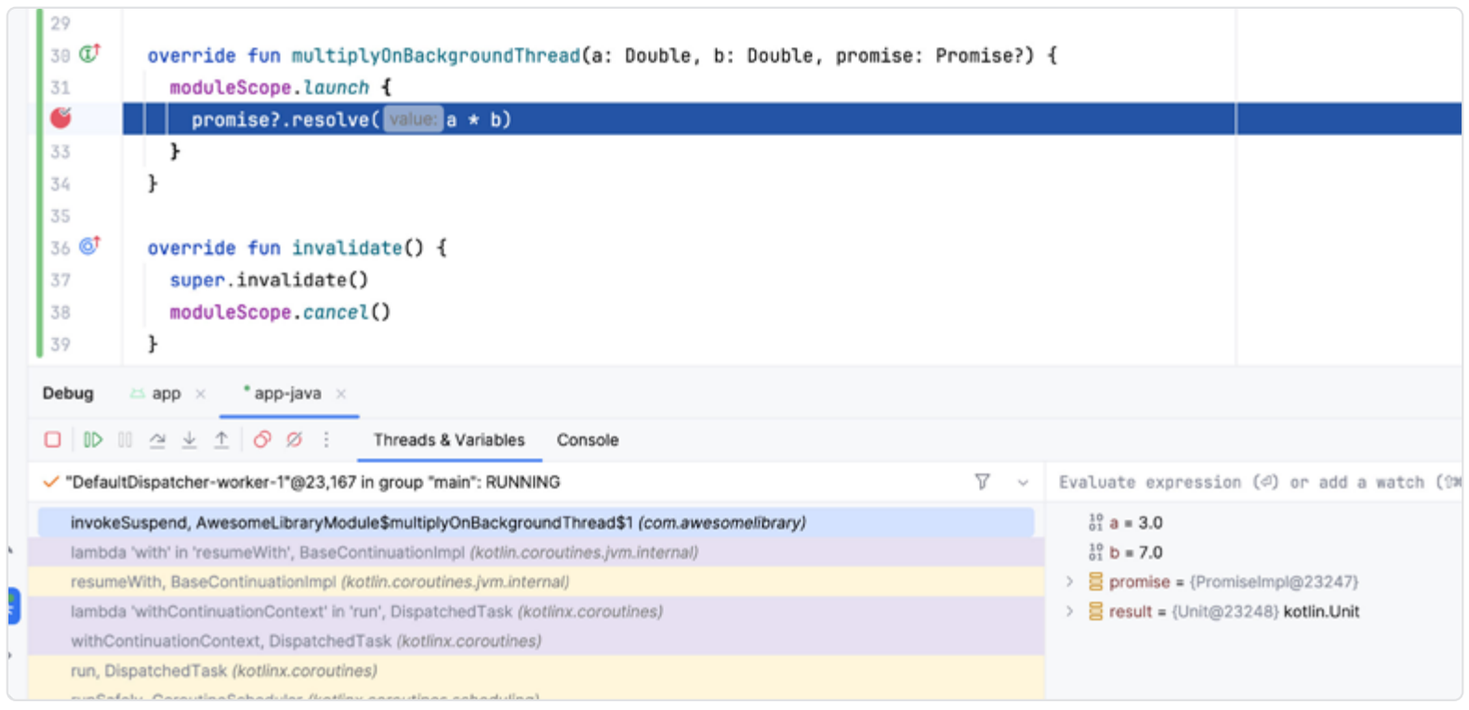
Creating CoroutineScope in a Kotlin file

Remember to cancel the scope when the module gets invalidated. This makes sure that we don't introduce any memory leaks. Now that we have the **`moduleScope`** available, we can offload the work to a background thread using its **`launch`** method:

```
override fun multiplyOnBackgroundThread(a: Double, b: Double,
promise: Promise?) {
  moduleScope.launch {
    promise?.resolve(a * b)
  }
}
```

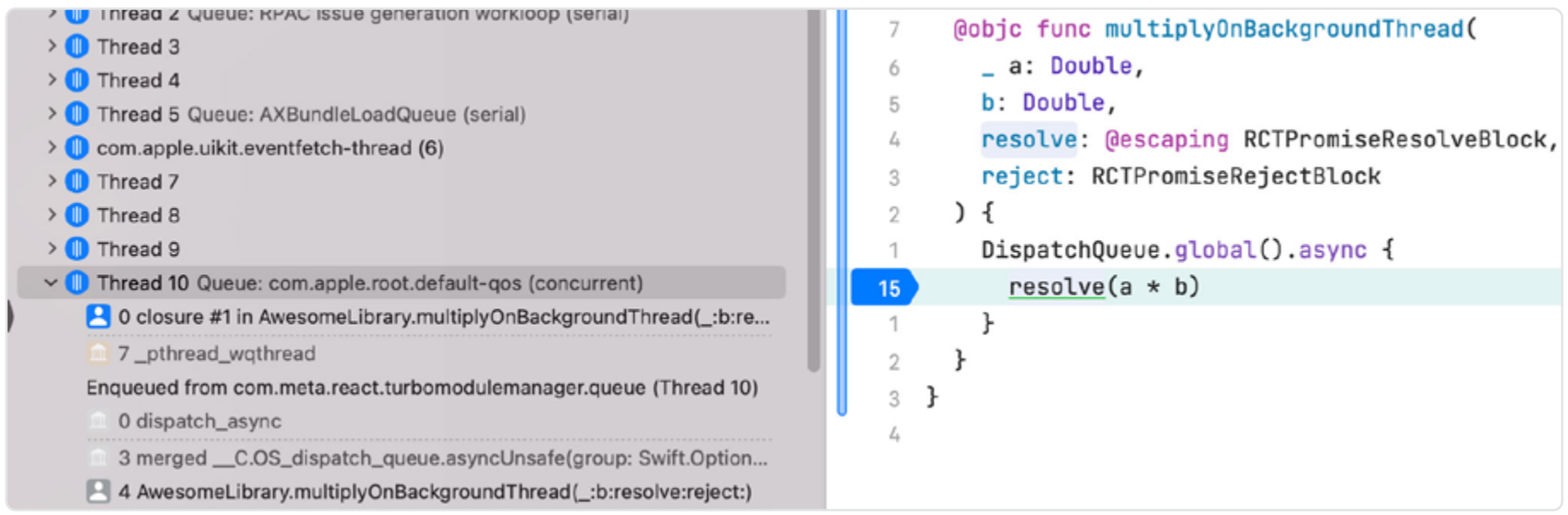
Example use of `moduleScope.launch` in Kotlin

Upon calling this function, once you place a debugger breakpoint, you can observe that the **`multiplyOnBackgroundThread`** is being run on a worker thread instead of the main thread.



A screenshot from Android Studio stopped on a breakpoint

The same thing happens on iOS:



A screenshot from Xcode stopped on a breakpoint

There are a few more caveats about threading, which we discuss in the Understand the Threading Model of Turbo Modules and Fabric chapter.

Replace platform-independent code with C++

If you have platform-independent logic that can be replaced with C++, it can give you a significant performance boost. C++ is supported on iOS, Android, Windows, and virtually everywhere. However, using a lower-level language means you have to be extra careful when dealing with threading and watch out for memory leaks. You can find a guide on building C++ Turbo Modules on the React Native website.

At the time of writing this guide (January 2025), C++ Turbo Modules don't support auto-linking on iOS. To improve the developer experience and spare them modifying their AppDelegate code, you can use the `registerCxxModuleToGlobalModuleMap` method to register your C++ Turbo Module to be available at runtime. To do that, create a small Objective-C class and leverage the `+ load` method, which gets called once a class is added to the Objective-C runtime during the app launch process.

```
#include <ReactCommon/CxxTurboModuleUtils.h>

@implementation YourModule

+ (void)load {
   facebook::react::registerCxxModuleToGlobalModuleMap(
   std::string(facebook::react::YourModule::kModuleName),
   [&](std::shared_ptr<facebook::react::CallInvoker> jsInvoker) {
   return std::make_shared<facebook::react::YourModule>(jsInvoker);
 });
}

@end
```

The hidden cost of interfacing C++

Crossing the boundaries of languages often incurs a hidden cost that we should be aware of when making a deliberate choice to use another native language in our app.

Objective-C++

In iOS, apps—and React Native itself—still commonly use Objective-C, which, in most cases, is slower than Swift due to its dynamic nature. Each Objective-C method call requires a lookup in the method table. On the flip side, the cost of mixing Objective-C with C++ is close to zero since it's handled entirely at compile time. The Objective-C++ compiler treats it as native C++ code. Unfortunately, when building Objective-C Turbo Modules, the method call is executed through a lookup in the method table.

Swift C++ Interoperability

In comparison, Swift uses virtual method tables (vtables) per class similarly to C++. On top of that, Swift's interoperability with C++ also has nearly zero cost when passing between languages. That's one of the reasons why switching to Nitro Modules that skip Objective-C and use C++ interoperability can give you a significant performance boost.

JNI

In the Android ecosystem, the Java Native Interface (JNI) enables interaction between bytecode running in the Java Virtual Machine (JVM) and native code like C++. However, JNI can be costly, as each call requires a function lookup in method tables, adding overhead.

Using JNI directly, for example, bridging each Kotlin call to C++, results in slower calls but can be worthwhile for lengthy operations. With C++ Turbo Modules, JNI is primarily used for module initialization, so you can skip it during runtime. Since JavaScript holds references to C++ functions via JSI (React Native's JavaScript Interface), you can access this code directly without extra overhead.

GUIDE

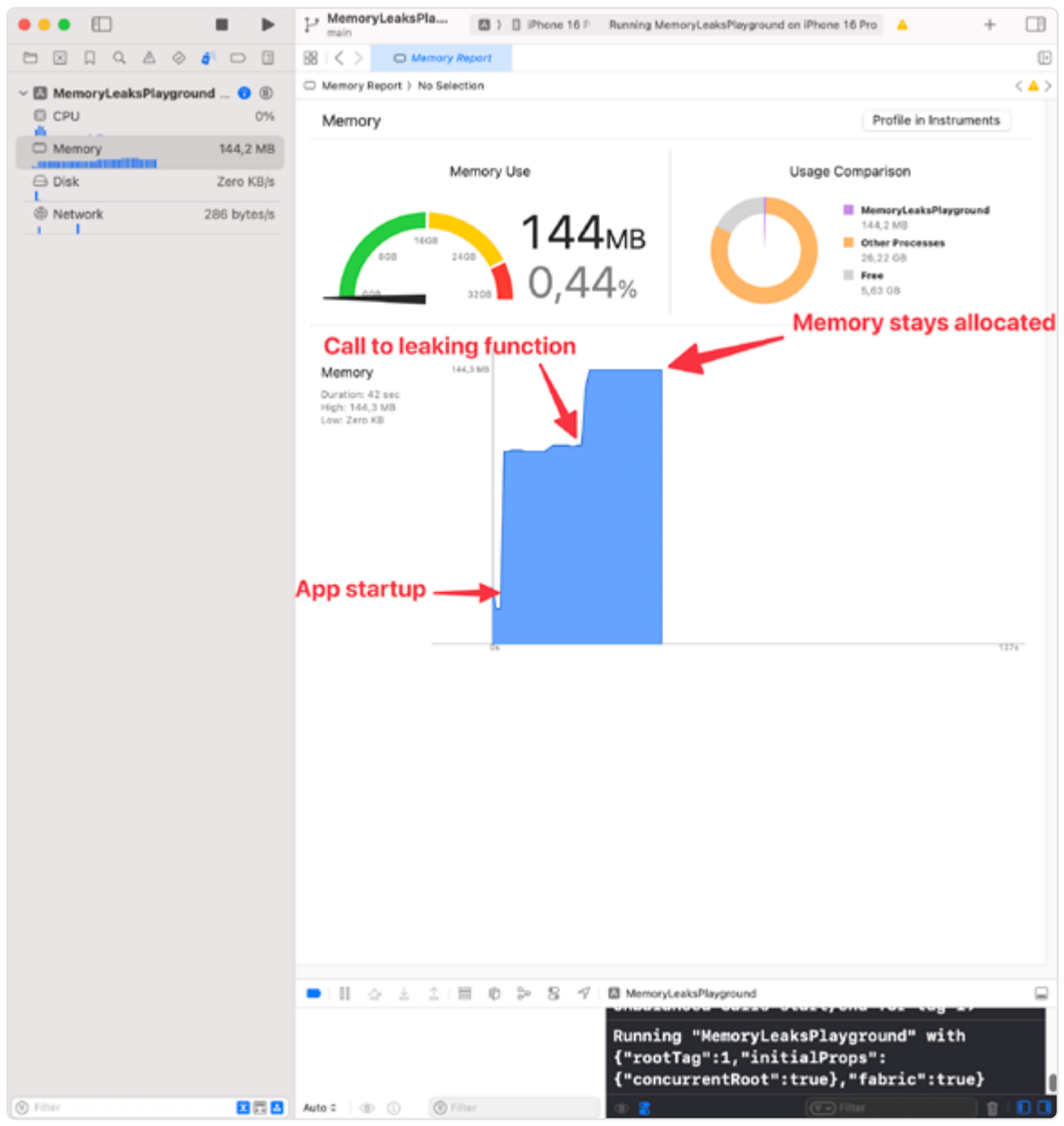
HOW TO HUNT MEMORY LEAKS

It's tough to detect a memory leak without dedicated tools. Thankfully, both iOS and Android have some good tooling that helps us identify the source of the issue, similar to how JS Memory Profiler does, which we covered in the How to Hunt JS Memory Leaks chapter.

Xcode

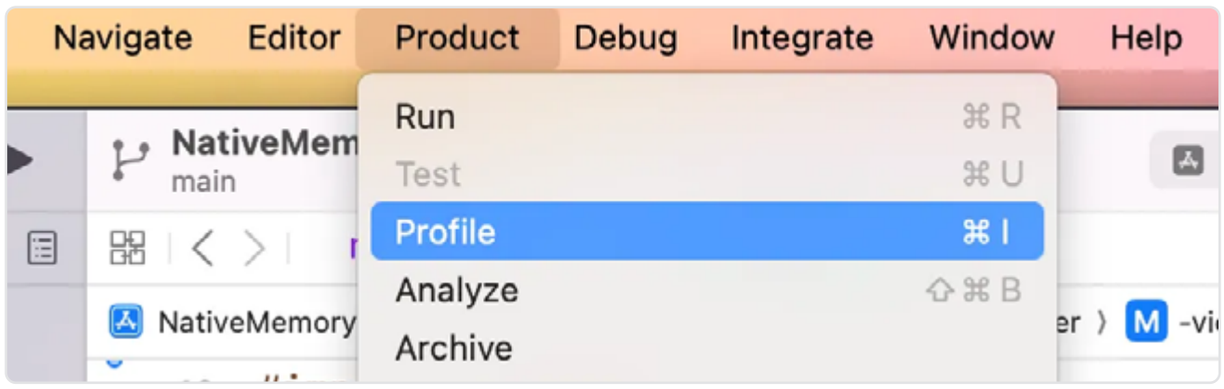
Apple's IDE is a great tool for hunting memory leaks. While profiling your app this way, you can uncover native memory leaks, which we discussed in detail in the Understanding Native Memory Management chapter. To get started, open Xcode and build your app.

The quickest way to examine memory leaks can be to check the "Memory Report" from the Debug Navigator in the side pane when running the app.



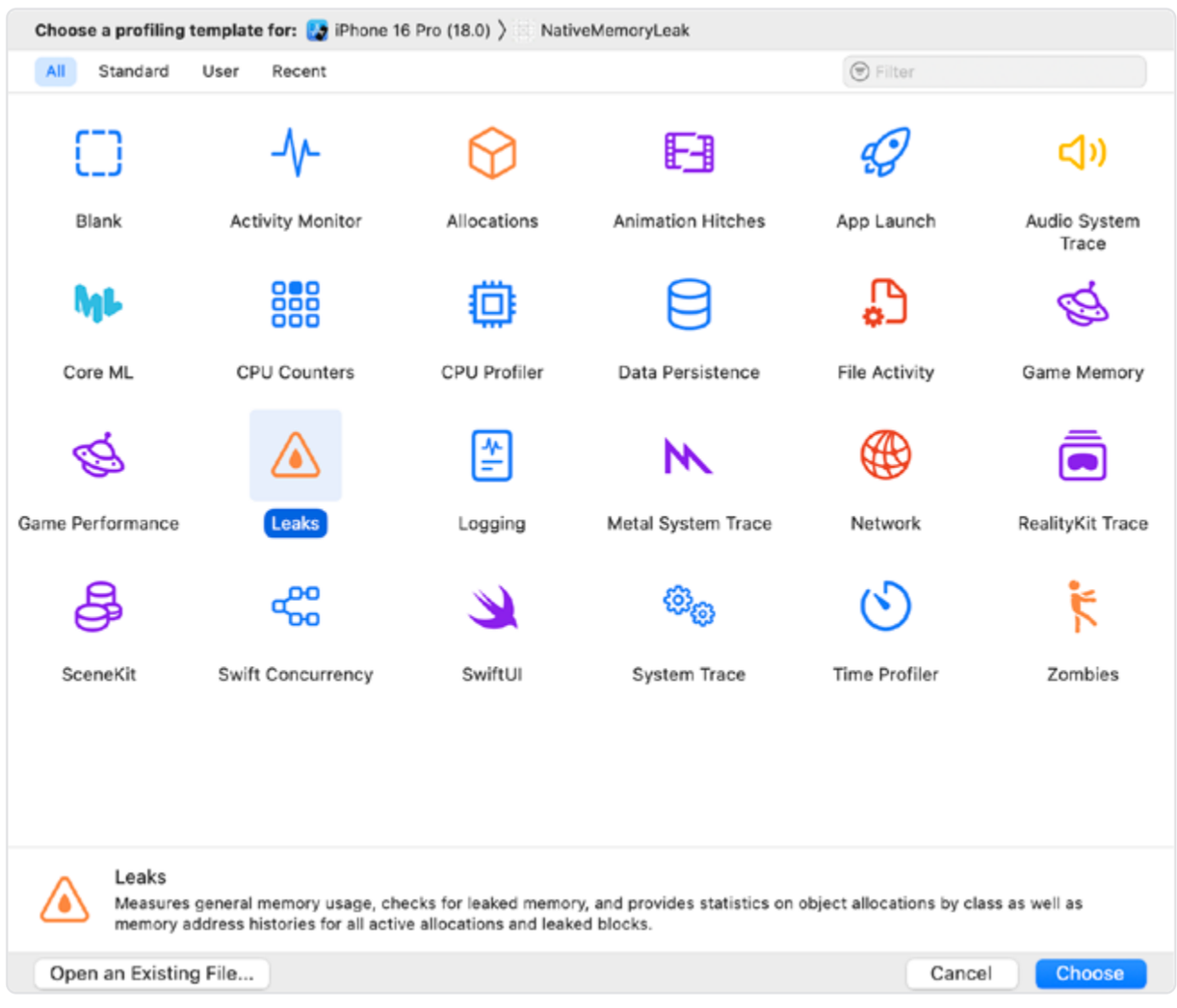
Memory Report presenting a leak in Xcode

It can show you a graph of memory used in your app. It's worth using this profiling technique on the release build of your app. The memory graph is useful for quickly tracking problems with leaking memory—but to check the source of the leak, you will need to use Xcode Instruments, the Leaks profiling template, specifically. To start a new profiling session, navigate to Product > Profile.



Xcode's Product Profile from menu bar

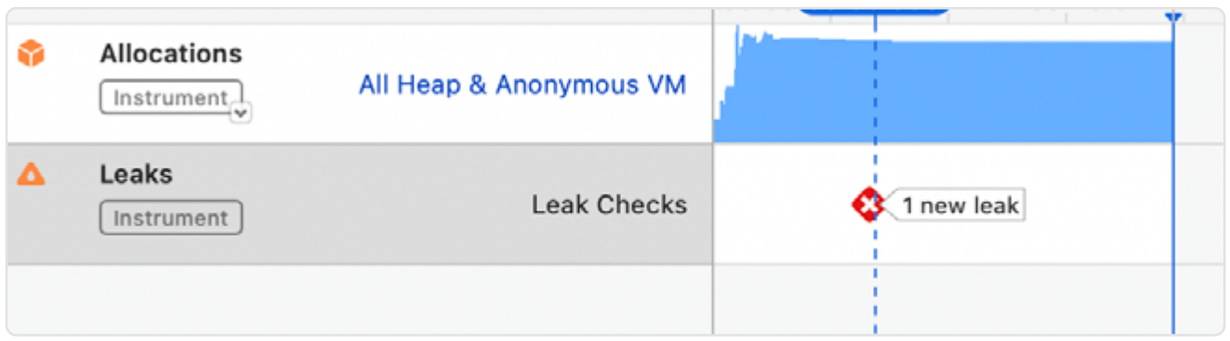
Once the app is built, choose "Leaks" from the menu and press the "Choose" button:



Instruments with Leaks tool selected

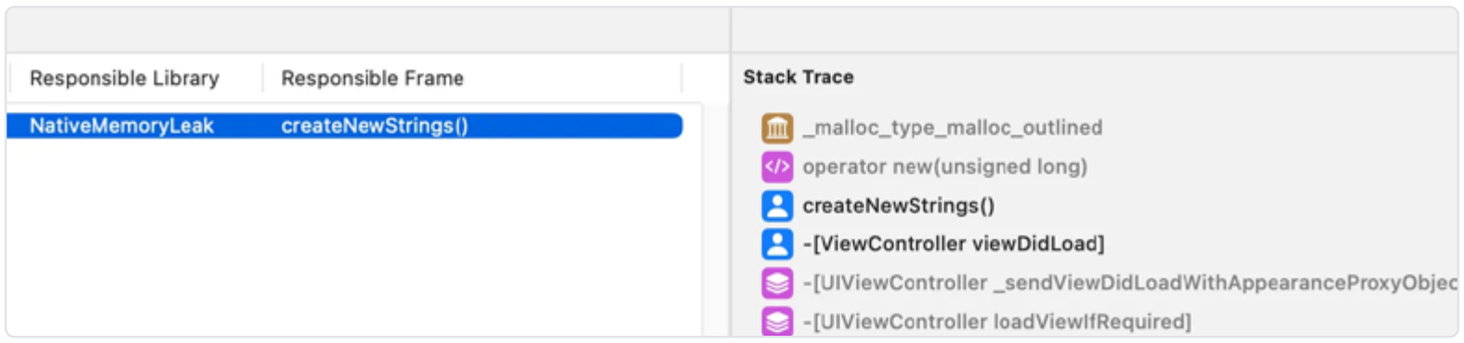
Now, a new window should open. Click on the record button at the top left corner, and on your device, follow the steps that likely cause the memory leak. It might be navigation to a new screen that triggers some native code or a press of a button.

Once recorded, the Leaks tool will show you a red marker indicating a leak.



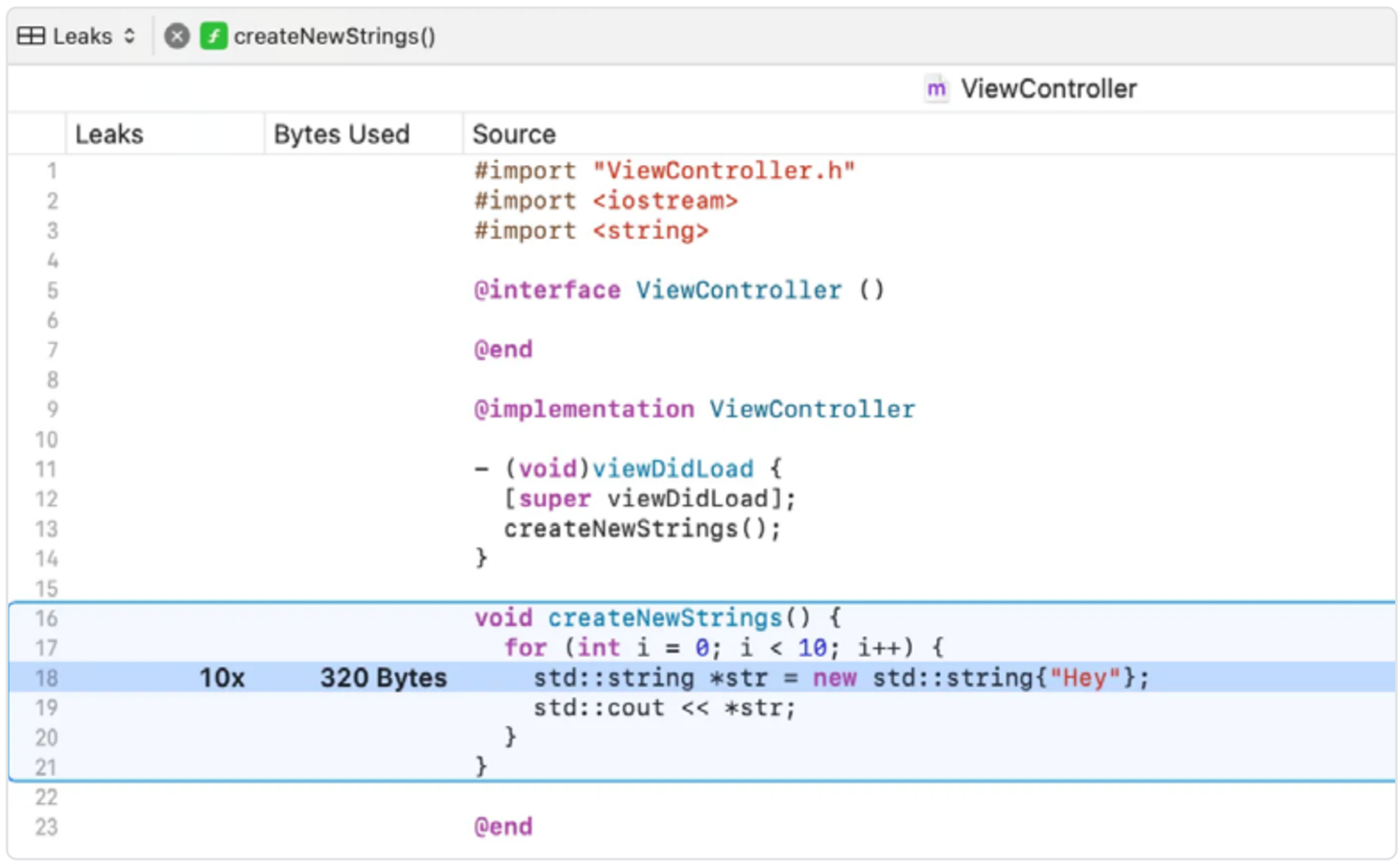
Leaks detected one new leak

After clicking on it, you should see a summary of leaked objects, a responsible library, and a responsible frame. On the right side, there is a stack trace that shows you exactly which function leaked memory.



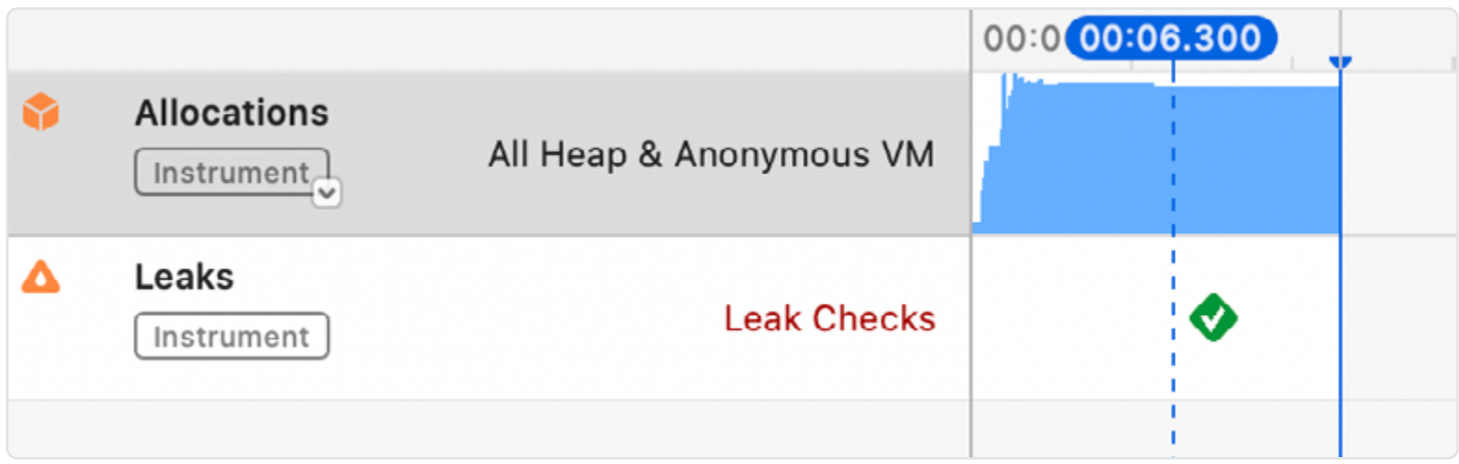
The library and frame responsible for a leak, next to the stack trace pointing to `createNewStrings` method call

In our case, it was **`createNewStrings()`**. By double-clicking on the function name, Xcode shows the source code of this function:



Implementation of a C++ function leaking memory

Ah, we forgot to delete the string allocated on the heap! After explicitly deleting the string, Xcode shows us that the leak is gone.

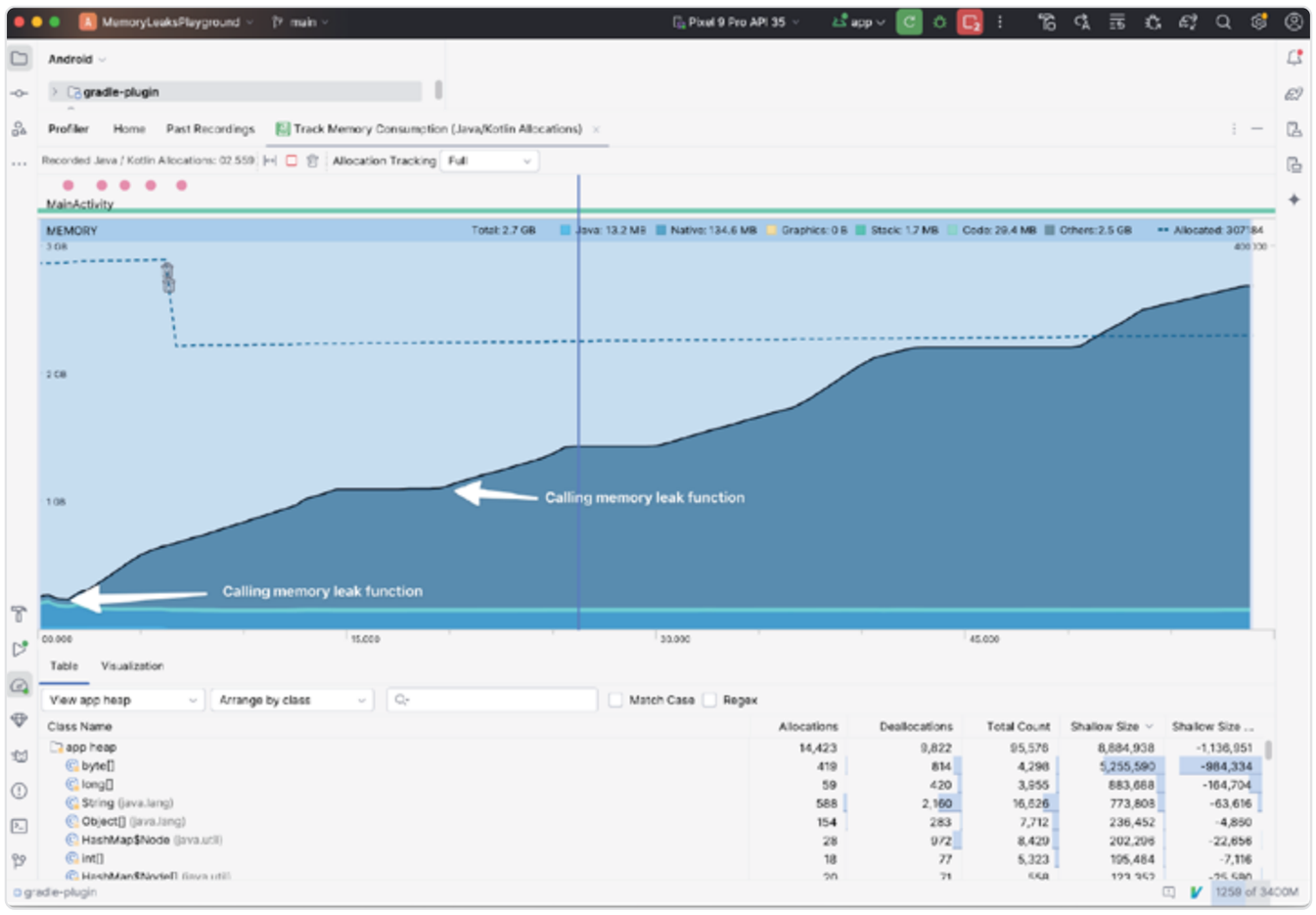


Leaks showing no leaks

Now, let's see how we can hunt memory leaks on Android.

Android Studio

Similarly to Xcode, you can build your app in release mode and examine the memory usage graph. In Android Studio, navigate to the "Profiler" tab, which should be near the bottom left corner, and then choose "Track memory consumption".



Memory Leak Playground in Android Studio

Now, let's hunt down a more complex leak using the Android Studio profiler. If you are familiar with how Android works, you probably know that its default behavior is to recreate its `MainActivity` class when configuration changes. This includes app rotation, dark mode change, and many more.

> React Native opts out of this behavior in `ApplicationManifest.xml` by specifying `android:configChanges`.

However, in native apps, it's common to deal with these configuration changes. Pair the recreation of `MainActivity` with a global singleton implementing the listener pattern, and you have a recipe for a memory leak. Let's see how a seemingly straightforward code can retain our `MainActivity` from deallocation.

Here is the `EventManager` class that registers listeners with the interface for listeners:

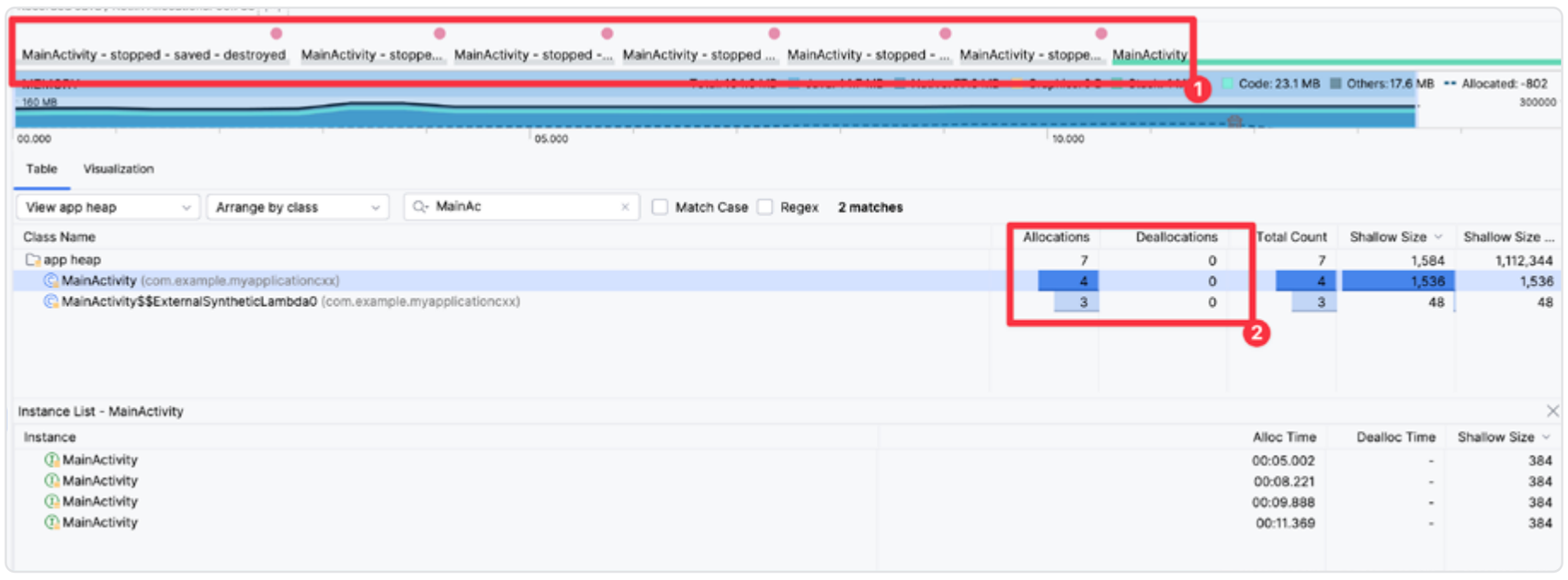
```
object EventManager {
  private val listeners = mutableListOf<Callback>()

  fun addListener(callback: Callback) {
    listeners.add(callback)
  }
```

```
  fun removeListener(callback: Callback) {
    listeners.remove(callback)
  }
}

interface Callback {
  fun onEvent()
}
```

Now let's make our **MainAcitivity** subscribe to **EventManager** changes:

```
class MainActivity : AppCompatActivity(), Callback {
  // ...

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // ...
    EventManager.addListener(this)
    }
  }

  override fun onEvent() {
    Log.d("MAIN_ACTIVITY", "Hey")
  }
```

Next, let's run the Memory profiler by clicking Run > Profile and then choose "Track Memory Consumption (Java/Kotlin allocations)". The app will be launched, and you should see the memory graph updating live. After rotating the device a few times, we can see the **`MainActivity`** lifecycle (highlighted below with number 1). Red dots symbolize touch events. Below, we can see that our activities were re-created after each tap.



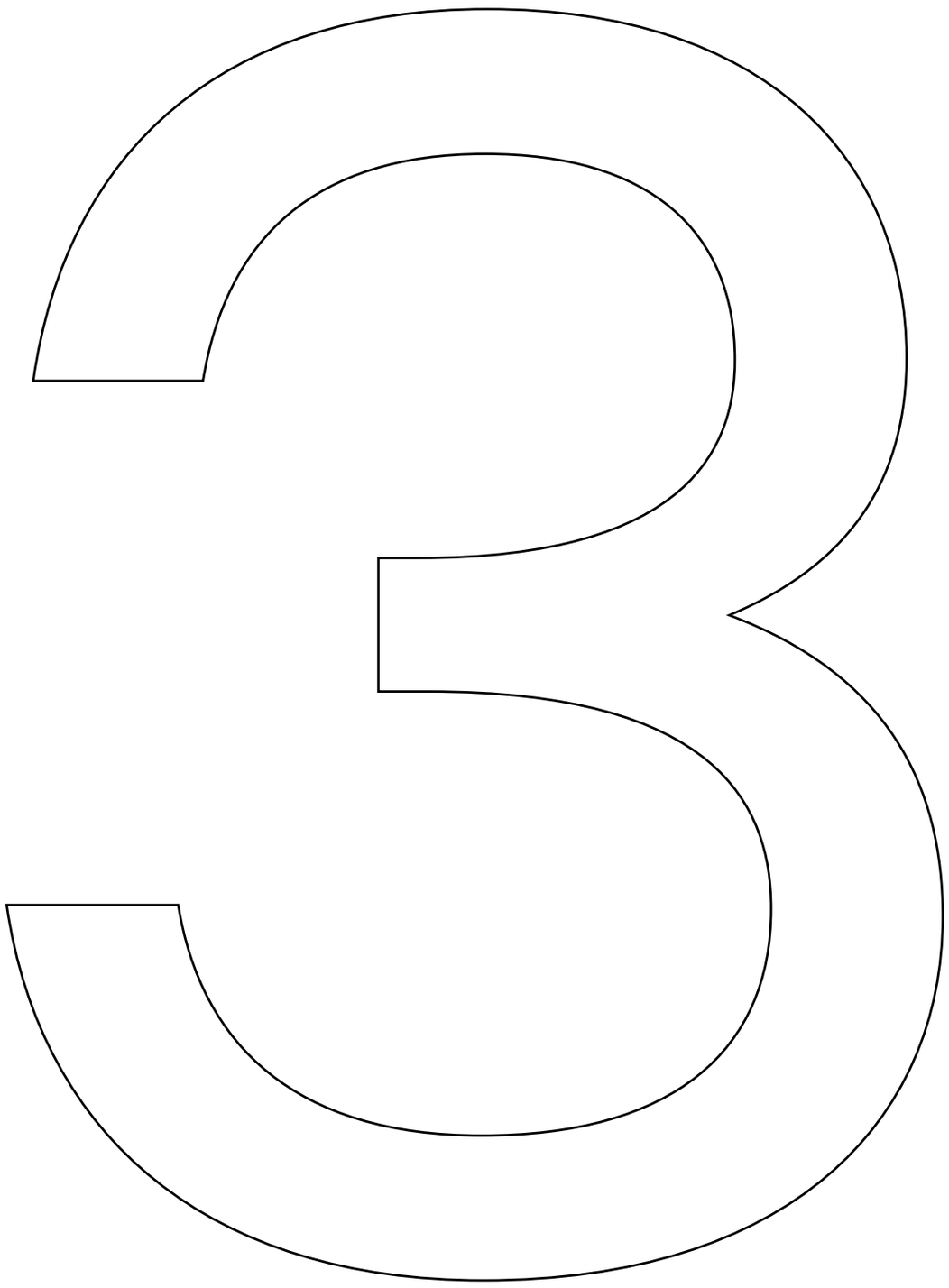
A breakdown of a memory leak from EventManager holding references to MainActivity

Looking at the second highlight, you can see that **`MainActivity`** was allocated four times while deallocated 0 times! This means those previous main activities are still referenced by our

`EventManager`, which prevents them from being deallocated by the Garbage Collector. This may quickly escalate to an app crashing.

In practice, apps don't leak memory that often. But when they do, looking for the source of the leak is dreaded by most developers out there. With the right tools in hand, you'll be better positioned to identify the reasons for the leak and optimize the app to use less memory, which, in turn, may prevent it from being killed by the OS prematurely.

PART



BUNDLING

Guides and techniques to improve TTI by utilizing ahead-of-time compilation and packaging techniques across JavaScript and Native

Introduction

In Parts 1 and 2 of this guide, we explained how to profile CPU and memory, measure key metrics such as Time to Interactive (TTI) and Frames Per Second (FPS), and apply best practices to improve the runtime performance in your app, mostly affecting the FPS metric. But what about the optimizations we can do at build time to make our app initialize faster?

That's where Part 3 comes in. Here, we'll focus on the techniques and tools that can help your React Native app open as quickly as possible, measured by the Time to Interactive (TTI) metric.

As we learned in Part 2, TTI is one of the most important performance metrics for your application. The faster your app loads, the more likely users are to return. Conversely, a slower-loading app increases the chance of users abandoning it for alternatives. Let's explore the key components that influence startup time.

JavaScript bundles and bundlers

Much like on the web, React Native developers use bundlers to produce the final JavaScript artifact read by a JS engine. A bundler is a program that converts multiple source files of different kinds—including TypeScript, HTML, CSS, JPG, MP4, and more—imported through an entry file, into a format understandable by a target JS engine. The result is usually a `.js` file, along with other non-JS assets like `.html`.

By default, React Native projects ship with Metro, a bundler specially created for React Native. It supports bundling for any platform, including the web, though it lacks some key functionalities there, such as code splitting and tree-shaking. While Metro provides an out-of-the-box experience and decent performance, other bundlers like Webpack or Rspack—widely used in web development—can also be integrated into React Native projects with Re.Pack.

In development, your app bundle contains no JavaScript code but rather fetches it from the dev server. This allows for Hot Module Replacement or Fast Refresh with sub-second refreshes of the source files and what ends up on the screen.

When preparing for release, your app bundle will contain a dedicated `.jsbundle` file containing Hermes Bytecode instead of JavaScript and no connection to a dev server. This file will be created by native Xcode and Gradle scripts, which run the `npx react-native bundle` command under the hood, producing the JS artifact and then passing it through the Hermes compiler to make the final bytecode file.

Now, let's explore various packaging strategies and distribution formats related to iOS and Android platforms.

Android app bundles

There are two packaging formats available for Android apps: APK (Android Package Kit) and AAB (Android App Bundle), which encapsulate native code for a variety of devices running different architectures:

- **`armeabi-v7a`**—designed for older and lower-end ARM devices.
- **`arm64-v8a`**—optimized for 64-bit ARM processors, offering a better performance.
- **`x86`**—targeted at Intel-based devices.
- **`x86_64`**—a 64-bit version for Intel devices, providing better performance and capabilities.

In React Native projects, these architectures are configurable through the **`reactNativeArchitectures`** property in **`android/gradle.properties.`**

Building for multiple architectures increases the build time. When developing using React Native Community CLI, you can pass **`--activeArchOnly`** flag to **`run-android`** command to speed up the builds.

APK

APK is a traditional format for distributing and installing Android apps used during development and testing or when distributing apps outside the Play Store. Every **`.apk`** file is, in fact, a ZIP file as well. APKs are not optimized for splitting resources based on device architecture or other configurations and, hence, typically combine multiple device architectures inside so that they can be run on various devices or emulators, increasing the overall size.

AAB

AAB is a newer format introduced by Google to optimize app delivery for app stores. The **`.aab`** file contains files necessary to produce APKs on demand. This format is now required for distribution through the Google Play Store, which generates optimized APKs for each user's device, reducing download size.

Dynamic libraries

Similar to JavaScript, mobile platforms have a concept of a reusable library. Compared to interpreted language like JavaScript, statically compiled languages such as Kotlin or Java powering the Android platform need to link libraries during the compile step. It's important to know there are two formats for such libraries:

- **`.a`**—C/C++ Library files; statically linked.
- **`.aar`**—Android Archive; contains classes, resources like drawables, etc., statically linked.
- **`.so`**—shared Libraries; dynamically linked.

While statically linked libraries typically offer simplicity in dependency management and consistent behavior across devices coming from runtime linking issues, compared to dynamically linked libraries, they produce bigger APK sizes, require more work to rebuild and redeploy, and potentially can increase memory usage due to loading the same code by multiple apps.

A statically linked library is included with the application's code at *compile-time*, creating a single executable file that contains all the code necessary to run the application.

A dynamically linked library is loaded at *runtime*, allowing multiple programs to share the same library, reducing memory usage, and enabling updates without recompiling the application.

iOS app bundles

There are also two packaging formats available for iOS apps: IPA (iOS App Store Package) and APP (Application Bundle), which encapsulate native code for a variety of devices running two different architectures:

- **`arm64`**—used for most modern Apple devices with ARM-based processors.
- **`x86_64`**—used for the iOS Simulator on macOS, providing development and testing support on non-ARM hardware.

IPA

This file format is used to package iOS applications for distribution via the Apple App Store or through ad hoc and enterprise distribution channels. The **`.ipa`** file is an archive containing all the information necessary for an iOS app to be installed on an Apple device.

Fun fact: you can change **`.ipa`** to **`.zip`** and examine what's inside! The same goes for the **`.apk`** file.

The App Store will then use the information available in the IPA file to perform app thinning, which optimizes app installation by tailoring app delivery to specific devices. It includes features like App Slicing (delivering only the necessary app resources for a device's configuration), On-Demand Resources (downloading additional content as needed), and Bitcode (allowing Apple to optimize apps for different devices without requiring a new version from developers).

APP

It's a format used during development, primarily for running the app on a simulator. The **`.app`** is actually a directory that holds all the app resources, binary executables, and metadata, all combined into one. It's not intended for distribution.

Dynamic libraries

iOS is no different from Android in terms of the necessity of a linking step to include external libraries. Available formats are:

- **`.a`**—C/C++ Library; statically linked.
- **`.dylib`**—Dynamic Library; standalone and dynamically linked.

- `.framework`—framework bundle that includes binary, headers, and assets for a single platform; dynamically linked.
- `.xcframework`—XCFramework bundle that can contain multiple `.framework`s for different platforms and architectures; dynamically linked.

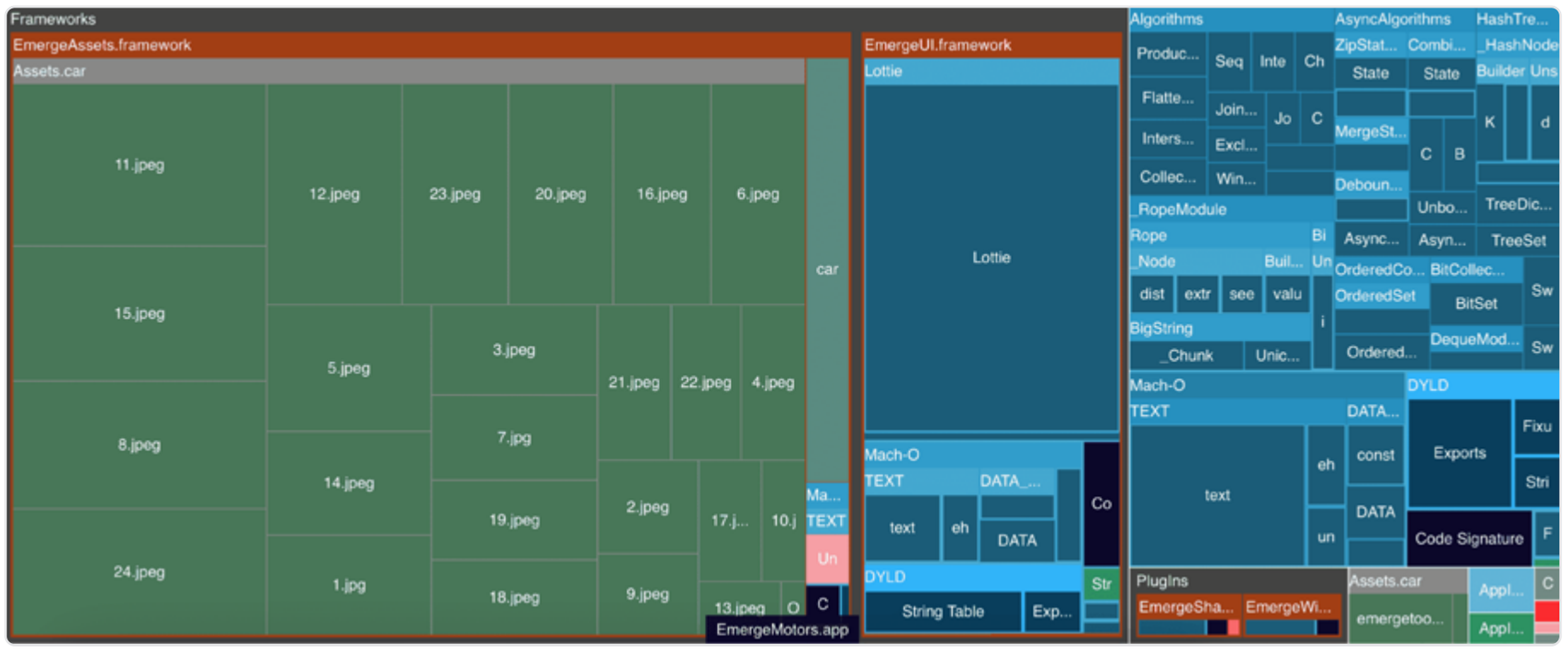
Since iOS natively loads any dynamic libraries on the initial path, Apple advises avoiding using that format *when possible and necessary* to improve the app startup time. However, in a broader scene, it's oftentimes better for a specific library to be dynamically linked and shared across different app targets.

When to switch from static to dynamic

If you use static linking and share one framework among multiple app targets, you will duplicate this framework for each of your targets. If you have three targets (say, an app, a share extension, and a widget extension) you will include this framework in your app three times. This is nicely visualized in a blog post by Emerge Tools:

A screenshot from Emerge Tools before changing linking architecture

As you can see, the `EmergeUI` is included in each target separately, aduplicating it three times. After making the framework link dynamically, they removed that redundancy:



A screenshot from Emerge Tools after changing linking architecture

Explaining how to achieve this linking architecture is beyond the scope of this guide, as it will depend on how a framework is created. We highly recommend you check the abovementioned blog post for more information.

Now that we know a little more about all the different bundles that can be found in mobile React Native apps and how they can be linked, let's analyze their contents and see how we can use that information to further optimize our apps' startup times.

GUIDE

HOW TO ANALYZE JS BUNDLE SIZE

In React Native apps, as with any other JavaScript-powered frameworks, it's important to understand what we actually ship to users as a JS bundle. The final JavaScript file typically consists of our own code, third-party dependencies code, and their own dependencies (which we call "transitive dependencies"). Since the size of that file will more or less influence the app initialization path, it's worth learning how to make our JavaScript smaller. There are many ways to do that, but they all revolve around the same idea: removing the code that's unnecessary at the time. And timing makes all the difference.

```
// This code can be removed in production builds
if (__DEV__) {
  // ...
}

// This code can be removed in Android bundle
if (Platform.OS === 'ios') {
  // ...
}
```

Examples of code that's only necessary at certain circumstances: in development or on iOS

For example, web developers optimize around the size of JS that's initially downloaded by the browser because browser engines need to download, load into memory, parse, and execute this JS. That's a lot of steps, and each one can be optimized. However, it's different in modern React Native apps that run Hermes, a JavaScript engine designed for mobile apps.

Hermes bytecode

If your app has been updated since 2022 and you're running React Native 0.70 or later, chances are that the final app bundle doesn't contain raw JavaScript code like web apps do. You will find there a bytecode file produced by the Hermes JS engine that ships with React Native by default.

> Hermes is the only JS engine that produces usable bytecode in React Native apps. Other engines, such as V8, are technically capable of applying similar ahead-of-time optimization techniques by utilizing a heap snapshot.

The key premise of Hermes, which is an engine designed for lower-end mobile devices, is that it essentially moves the "loading into memory" and "parse" steps to the build time.

This means that the same tricks from web development, most notably code splitting or heavily optimizing for initial bundle size, won't translate directly to React Native development. That's because all React Native can skip the downloading phase (JS is already available in the app bundle), and, thanks to Hermes specifically, they also skip loading JS into memory and parsing steps.

It's still important to pick smaller libraries, remove legacy code, and load only necessary libraries on the initialization path to avoid unnecessarily big bytecode or executing more JS than needed. The bytecode size will influence the app bundle size, making it slower to download and initialize. More JS running on the initialization path will impact the time to interactive (TTI) metric.

What's worse, when this JavaScript loads native modules, it will cause more work on the native side and reduce the benefits of lazy loading of Turbo Modules.

> Turbo Modules are lazily loaded by default. Compared to Native Modules in legacy React Naive architecture, which are eagerly initialized during app start, a Turbo Module will only be executed when it's called from the JavaScript side. This mechanism noticeably improves the TTI in apps that switch to the new architecture.

Imagine importing a component library with hundreds of components, but you use only a dozen. You import these components from a barrel file (export everything from a single file, which you can read more about in the Avoid Barrel Exports chapter) and not use tree shaking or babel plugins optimizing that. On its initialization path, the app will load all of those components, some of which could call their own TurboModules, which will now also start to initialize early, instead of never or later, when they're supposed to be used.

So, how can we inspect our JavaScript for a better overview of what's making it bigger than necessary?

source-map-explorer

We can use tools such as `source-map-explorer` to analyze any JavaScript bundle with a corresponding source map in a web UI. This tool gives us a bird's-eye view of what ends up on users' devices, allowing us to easily identify the outliers that often end up there unintentionally or are leftovers from legacy features.

What's even nicer about this tool is that it will also give you an idea of the size of your own code, where you can find just as many opportunities for JS bundle size reductions as in the case of third-party libraries.

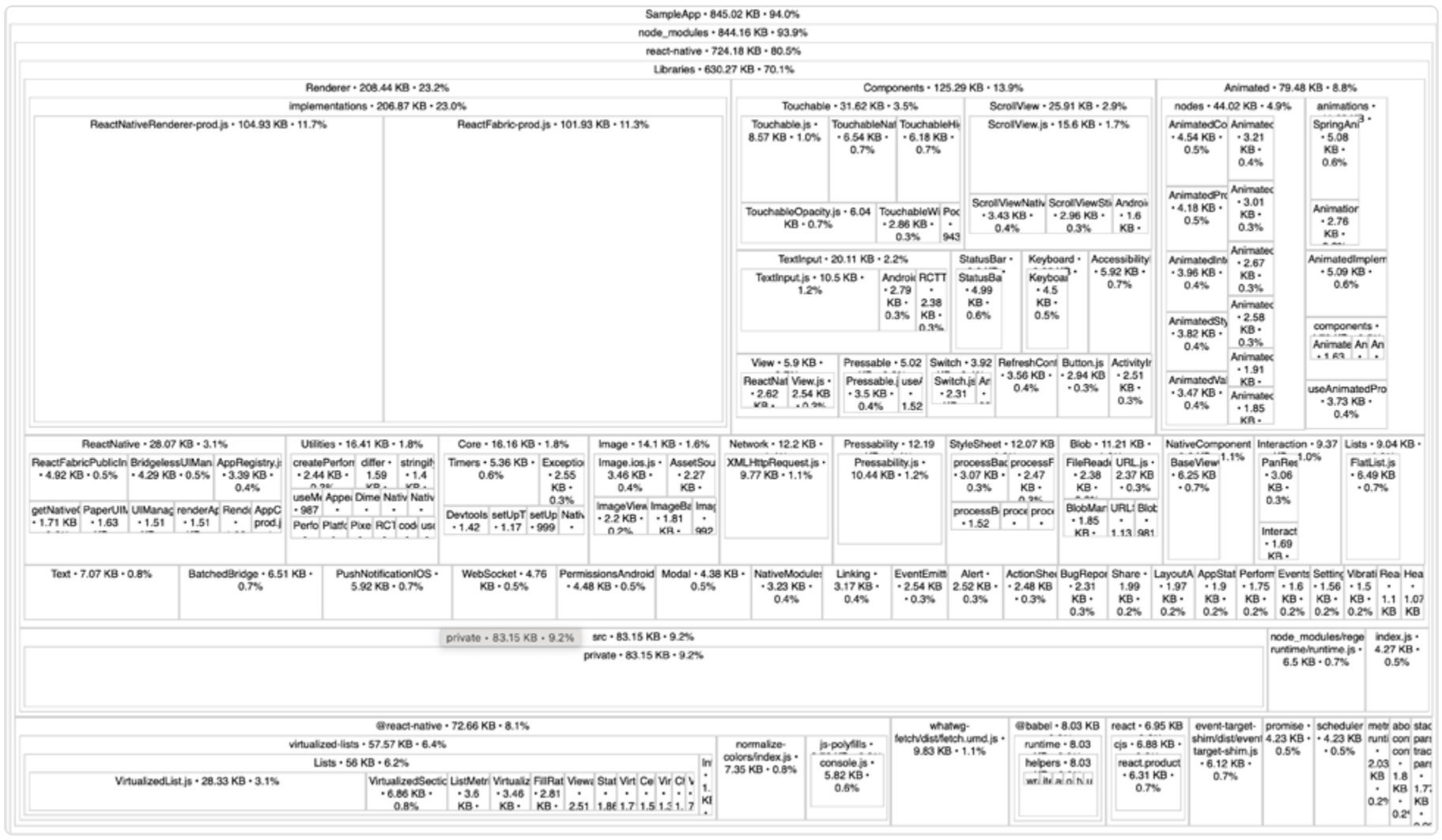
To use it, you'll need to first bundle your JavaScript and emit a source map:

```
> npx react-native bundle \
  --entry-file index.js \
  --bundle-output output.js \
  --platform ios \
  --sourcemap-output output.js.map \
  --dev false \
  --minify true
```

Once the JS is generated, you can run **`source-map-explorer`**:

```
> npx source-map-explorer output.js --no-border-checks
```

Since Metro produces invalid column/line mappings, we need to pass the **`--no-border-checks`** flag to disable validator checks. The good news is that these invalid mappings are not preventing the tool from generating the UI, and once the command finishes, it should open a web browser with the output similar to this:



Web UI of source-map-explorer output

The bad news is that you may lose up to 30% of the information due to invalid mappings, which will hide extra optimization opportunities from you. To get even more information from the bundle, you can use Expo Atlas, which integrates tightly into Metro pipeline.

Expo Atlas

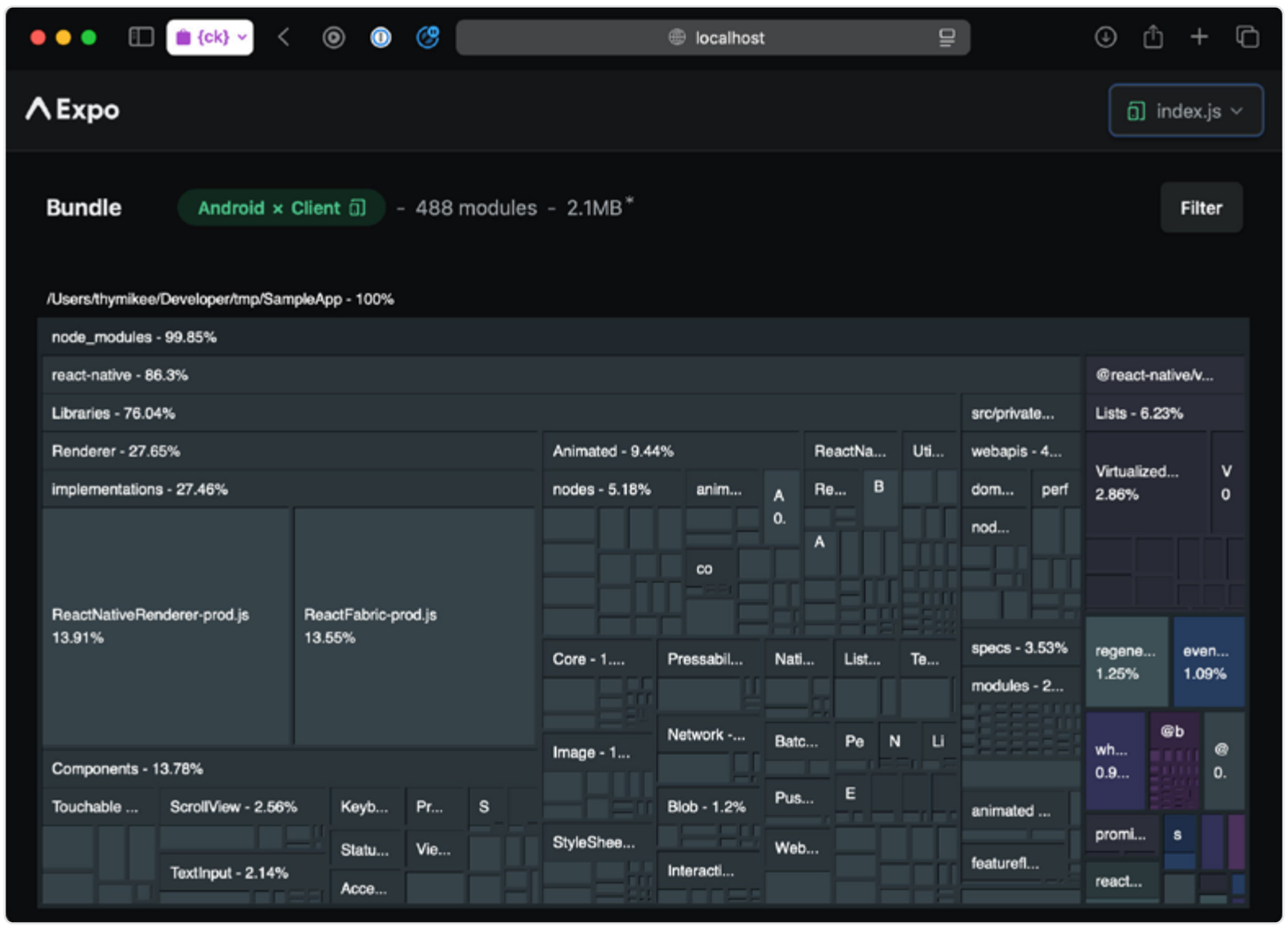
Another tool you can use to visualize the production bundle and identify which libraries contribute to the bundle size is Expo Atlas. This is an option for Expo projects, but there are hacky ways to use it in non-Expo projects. To use it, run Expo CLI with the **`EXPO_UNSTABLE_ATLAS`**:

```
> EXPO_UNSTABLE_ATLAS=true npx expo start --no-dev
# or export
EXPO_UNSTABLE_ATLAS=true npx expo export
```

Then you can run:

```
> npx expo-atlas
```

This will open the UI of Expo Atlas, where you can inspect the bundle.



Web UI of Expo Atlas output; fun fact: this report is generated from a non-expo project

There's a way to run Expo Atlas without Expo CLI, using **`expo-atlas-without-expo`** project.

Rspack bundle analysis

If you're using Re.Pack with Webpack or Rspack as a replacement for Metro, you'll experience a wider variety of higher-quality tools to inspect your JS bundle. We'll focus on Rspack, as it provides superior performance and is mostly on par with the Webpack ecosystem.

All tools we discuss from now on work similarly to the ones already discussed, so we skipped the screenshots for brevity's sake.

webpack-bundle-analyzer

Rspack's CLI supports bundle analysis out-of-box via the **`--analyze`** option. It uses **`webpack-bundle-analyzer`** behind the scenes:

```
> rspack build --analyze
```

bundle-stats and statoscope

You can also generate a **`stats.json`** file for further analysis with other bundle analysis tools like **`bundle-stats`** or **`statoscope`** using Re. Pack's **`bundle`** command with extra **`--json stats.json`** flag:

```
> $ npx react-native bundle \
  --platform android \
  --entry-file index.js \
  --dev false \
  --minify true \
  --json stats.json
```

Then, when using Statoscope, you can pass the generated **`stats.json`** file to the **`bundle-stats`** command:

```
> npx bundle-stats --html --json stats.json
```

And now you're ready to enjoy the rich HTML report in a web app! You can also compare one stats file to another, which should give you insights into how your final bundle changed over time—gaining or reducing size.

Rsdoctor's bundle analysis

Rsdoctor provides the bundle size module, which is mainly used to analyze the information of the outputs of Rspack, including the size of resources, duplicate packages, and module reference relationships. To use it, install **`@rsdoctor/rspack-plugin`** dependency and add it as a plugin to your **`rspack.config.js`**:

```
const { RsdoctorRspackPlugin } = require('@rsdoctor/rspack-
plugin');

module.exports = {
  plugins: [
    process.env.RSDOCTOR &&
      new RsdoctorRspackPlugin({
        // plugin options
      }),
  ].filter(Boolean),
};
```

Next time you run your app with **`RSDOCTOR`** env variable set to true, it will open the Rsdoctor UI, similar to how **`source-map-explorer`** does.

GUIDE

HOW TO ANALYZE APP BUNDLE SIZE

Keeping both the How to Analyze JS Bundle Size and the overall app size low can be crucial for app adoption on certain platforms. Users may opt for a different app based solely on its install size, especially when they're relying on mobile data. Google conducted an experiment measuring the correlation between APK size and the number of downloads. They found that for **every 6 MB increase in an APK's size, the installation conversion rate decreased by 1%.**

A couple of different app sizes affect users in different ways, network and storage-bound. Network-bound metrics would be download size and update size, which describe the amount of compressed data transferred over the internet. Storage-bound metrics, like install size and storage size, describe the uncompressed size of the app and its caches stored on the device.

Measuring Android app size

The Android ecosystem provides several convenient ways to inspect app size. One we particularly like for its ease of use is Ruler, a Gradle plugin by Spotify. It integrates into your existing app and allows you to analyze, debug, and release app bundles for particular architectures.

Installing it requires tinkering with Gradle build settings. First, add the Ruler Gradle plugin to the `buildscript`'s `classpath` in your top-level `build.gradle` file:

```
buildscript {
    // ...
    dependencies {
      // ...
      classpath("com.spotify.ruler:ruler-gradle-plugin:2.0.0-
beta-3")
    }
}
```

Then, in your **`app/build.gradle`**, add the plugin near the top of the file:

```
apply plugin: "com.spotify.ruler"
```

Later, in the same file, add **`ruler`** configuration for the architecture you'd like to inspect, ABI, locale, SDK version and more. You can learn more about that in the official documentation.

ABI stands for Application Binary Interface, which is tied to the device architecture, such as **`arm64-v8a`** or **`x86`**. We covered these architectures in the Introduction.

An example configuration can look like this:

```
ruler {
    abi.set("arm64-v8a")
    locale.set("en")
    screenDensity.set(480)
    sdkVersion.set(34)
}
```

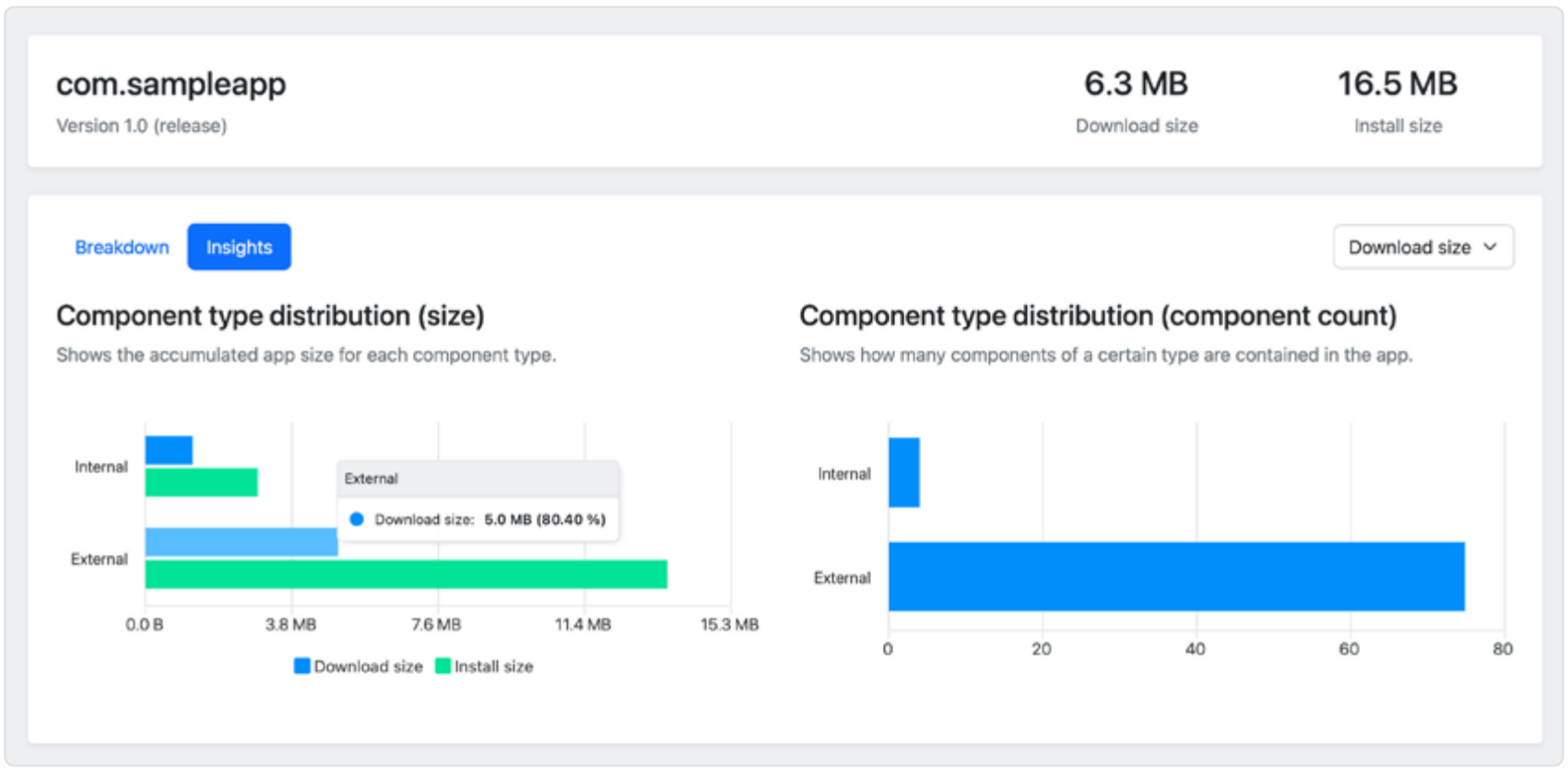
Now, you'll unlock new Gradle tasks: **`analyzeDebugBundle`** and **`analyzeReleaseBundle`**. You can perform these tasks by entering the **`android`** folder and running Gradle Wrapper (**`./gradlew`**) directly in your terminal:

```
> cd android
./gradlew analyzeReleaseBundle
```

If successful, this will generate the following output, pointing to a **`report.html`** file that you can open in your web browser:

```
> Wrote HTML report to file:///<PATH_TO_PROJECT>/android/app/build/
reports/ruler/release/report.html
```

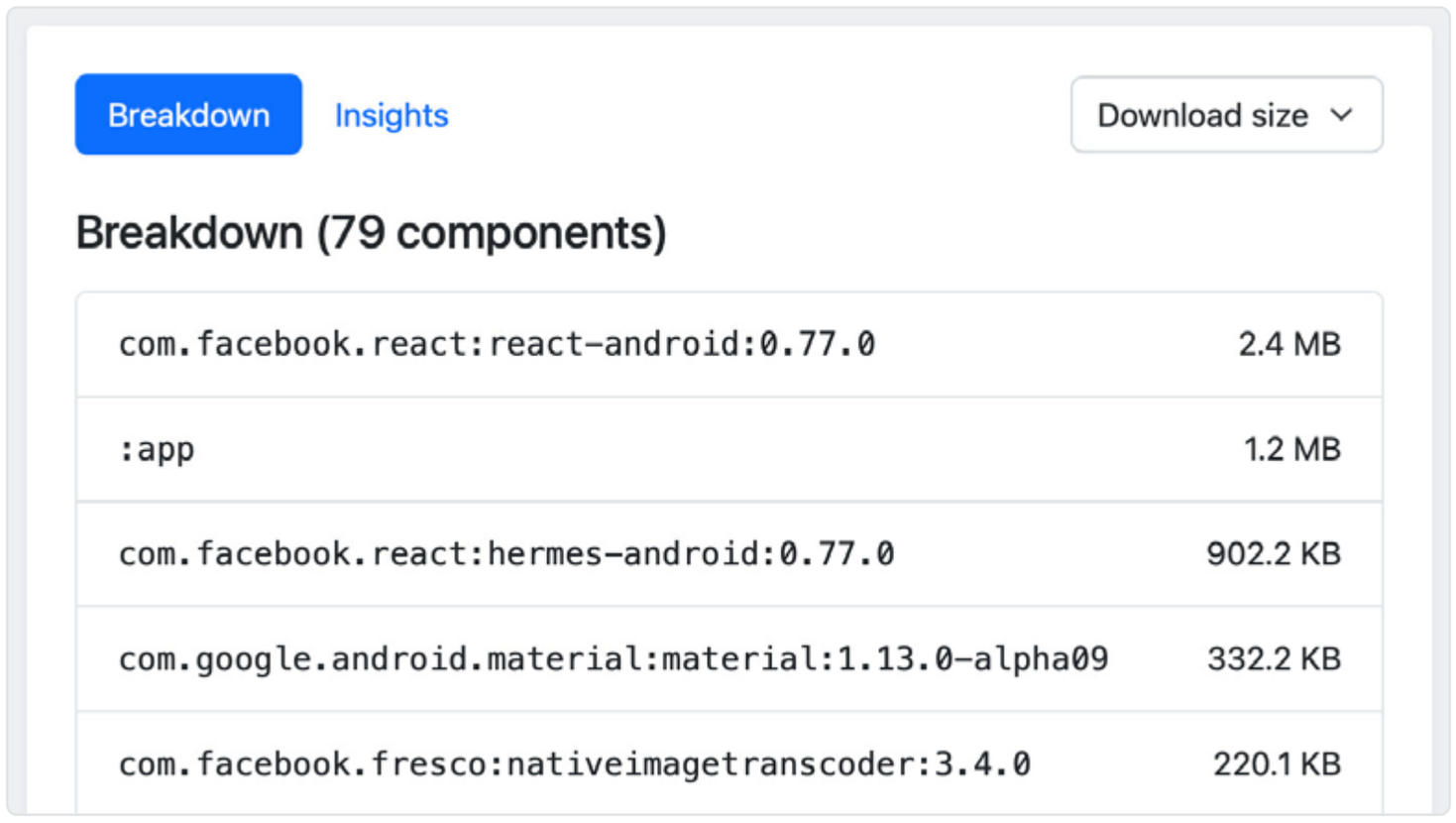
Once opened, you'll have access to the most important metrics: download size and install size.



A sample app size report from Ruler plugin

This sample app is optimized by R8, producing a download size of 6.3 MB. The same app without these build-time optimizations would be 9.5 MB. If you want to learn more about it, refer to the Shrink Code With R8 Android chapter.

Additionally, Ruler allows you to see the top-down breakdown of the internal and external components from the biggest to the smallest one:



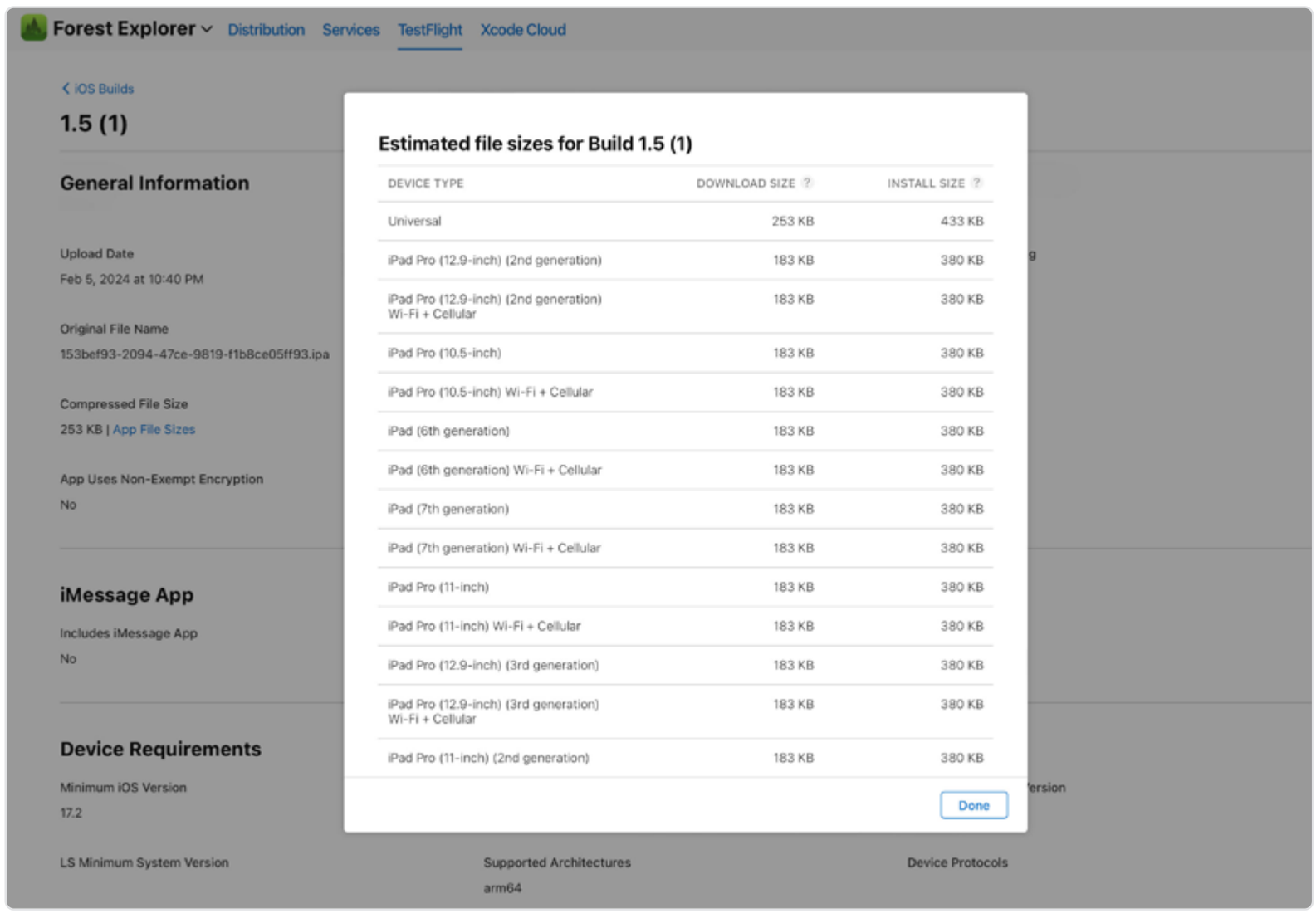
Android app components size breakdown

Ideally, you'd like to add size checks to your CI pipeline to automatically monitor size and, for example, send a GitHub comment under a PR. With Ruler, without any other tooling, you can set up validations on the size and choose to fail the build when the app is too big:

```
ruler {
    // ...
    verification {
        downloadSizeThreshold = 20 * 1024 * 1024 // 20 MB in
bytes
        installSizeThreshold = 50 * 1024 * 1024 // 2 MB in bytes
    }
}
```

Measuring iOS app size

If your app is available through the App Store or the TestFlight app, App Store Connect provides the most accurate size information. It displays the size for each variant of your app and warns you if it exceeds the limit for downloading over a mobile internet connection.

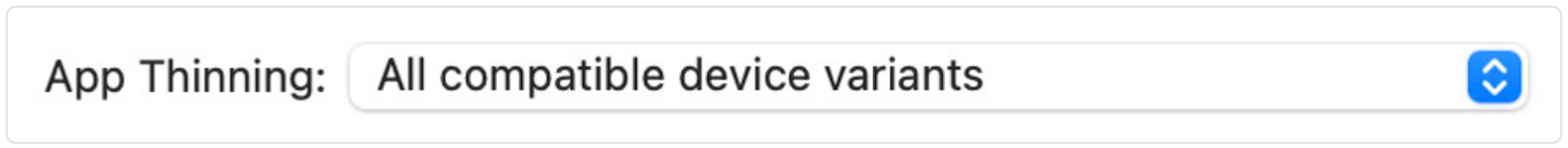


A table showing the different variant sizes available in App Store Connect

TestFlight builds include extra data for testing, making them larger than App Store versions. However, after App Store processing, which includes adding DRM and re-compressing, the final app size may slightly increase compared to the originally uploaded binary. This is normal due to the additional steps taken to secure and optimize the app for distribution.

While App Store Connect offers precise measurements, you can also use Xcode during development to generate an app size report with close estimates for download and installation sizes. To create this report, first generate a signed IPA in Xcode.

Then, in Organizer, after opening the "Distribute App" button, select "Custom." → pick your distribution", and ensure "All compatible device variants" is selected under the App Thinning option.



App Thinning select from app distribution manager

This is an equivalent of this in ExportOptions.plist file, in case you would like to run all this through `xcodebuild`:

```
<key>thinning</key>
<string>&lt;thin-for-all-variants&gt;</string>
```

This process creates a folder with your app's artifacts:

- **A *universal* IPA file** for older devices. This single IPA file contains assets and binaries for all variants of your app.
- ***Thinned* IPA files** for each variant of your app. These files contain assets and binaries for only one variant.

The output folder for your exported app also contains the app size report: a file named **`App Thinning Size Report.txt`**. This report lists the compressed and uncompressed sizes for each of your app's IPA files. The uncompressed size is equivalent to the size of the installed app on the device, and the compressed size is the download size of your app.

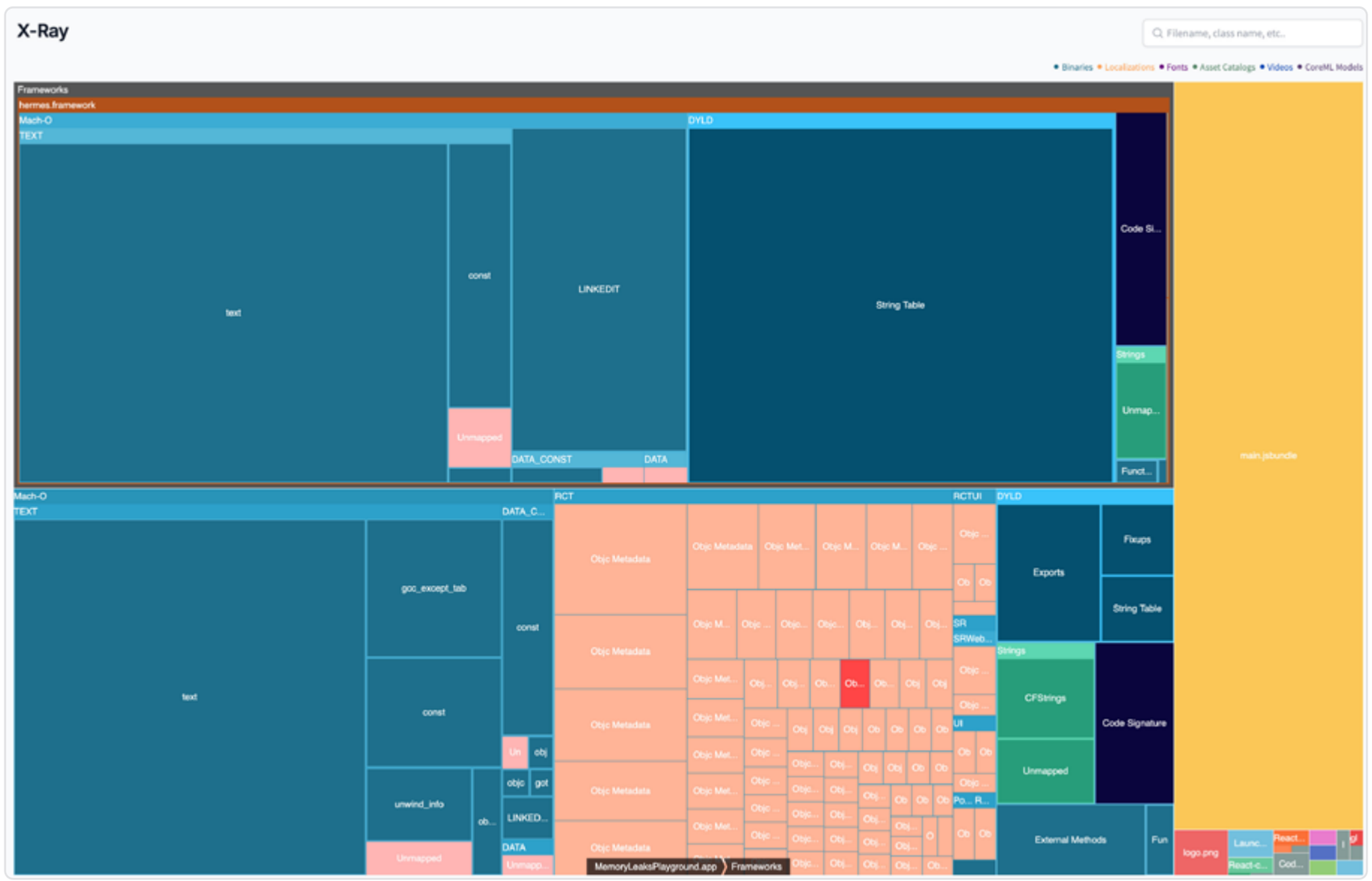
The following shows the beginning of the app size report for a sample app:

```
> App Thinning Size Report for All Variants of SampleApp

Variant: SampleApp-FB829A90-8597-43CA-B6ED-6AB3AEAA1C75.ipa
Supported variant descriptors: ...
App + On Demand Resources size: 3,5 MB compressed, 10,6 MB
uncompressed
App size: 3,5 MB compressed, 10,6 MB uncompressed
On Demand Resources size: Zero KB compressed, Zero KB uncompressed
```

Emerge Tools

The authors of this guide have no financial connection with Emerge Tools, a paid service with a 14-day free trial. However, since we're using their tools and they prove to be valuable when inspecting app size, we'd like to list them here as a viable alternative that works for both iOS and Android apps.

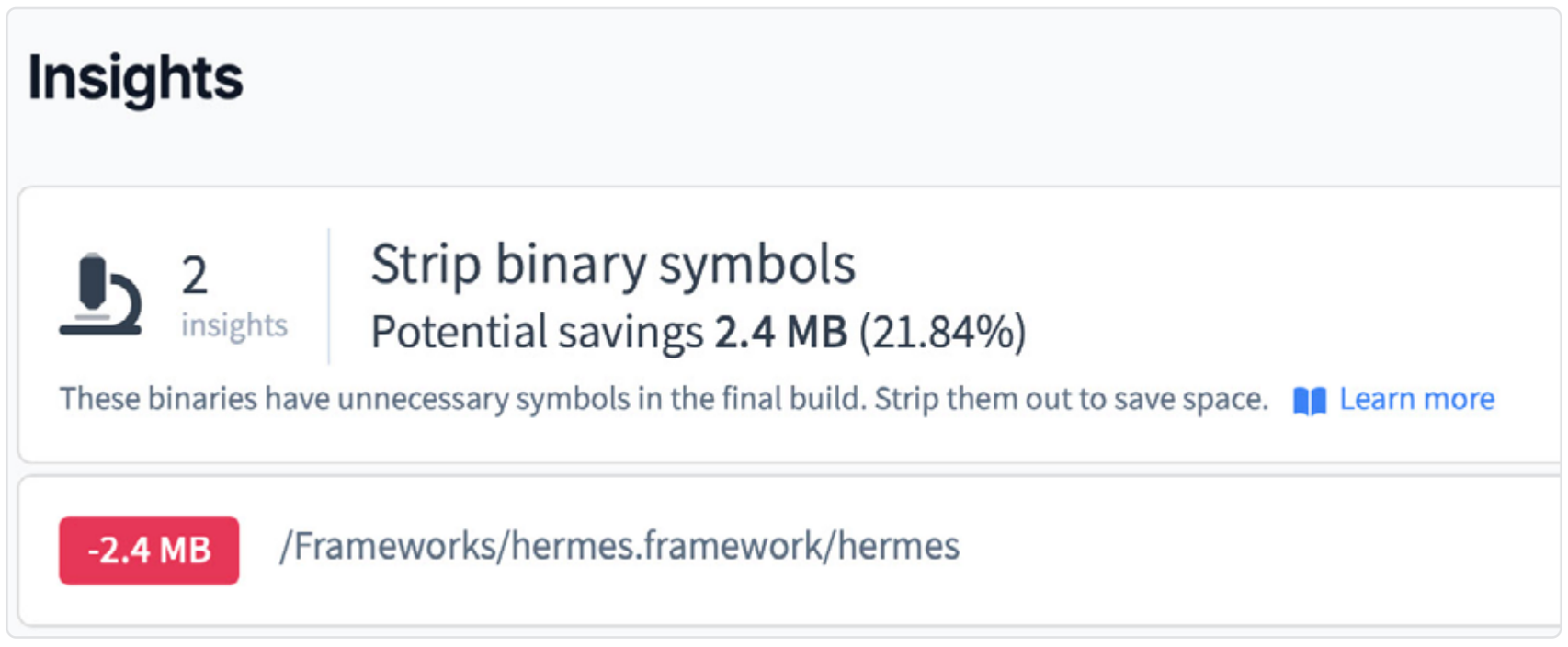
After uploading an IPA, APK, or AAB file, you'll be able to access the X-Ray or Breakdown tools that will present you with a UI like this:



A sample report of an IPA file using X-Ray tool from Emerge Tools

Much like **`source-map-explorer`** presents JS modules and libraries in the form of rectangles that you can click and inspect further, grouped by folders or module boundaries, X-Ray presents binary information about Android and iOS bundles in a similar form.

The tool also offers some insights, but we can't say it's most reliable. For example, it suggests removing the whole Hermes JS engine, which would end up quite badly for your app.



Be careful, because not all insights are worth implementing

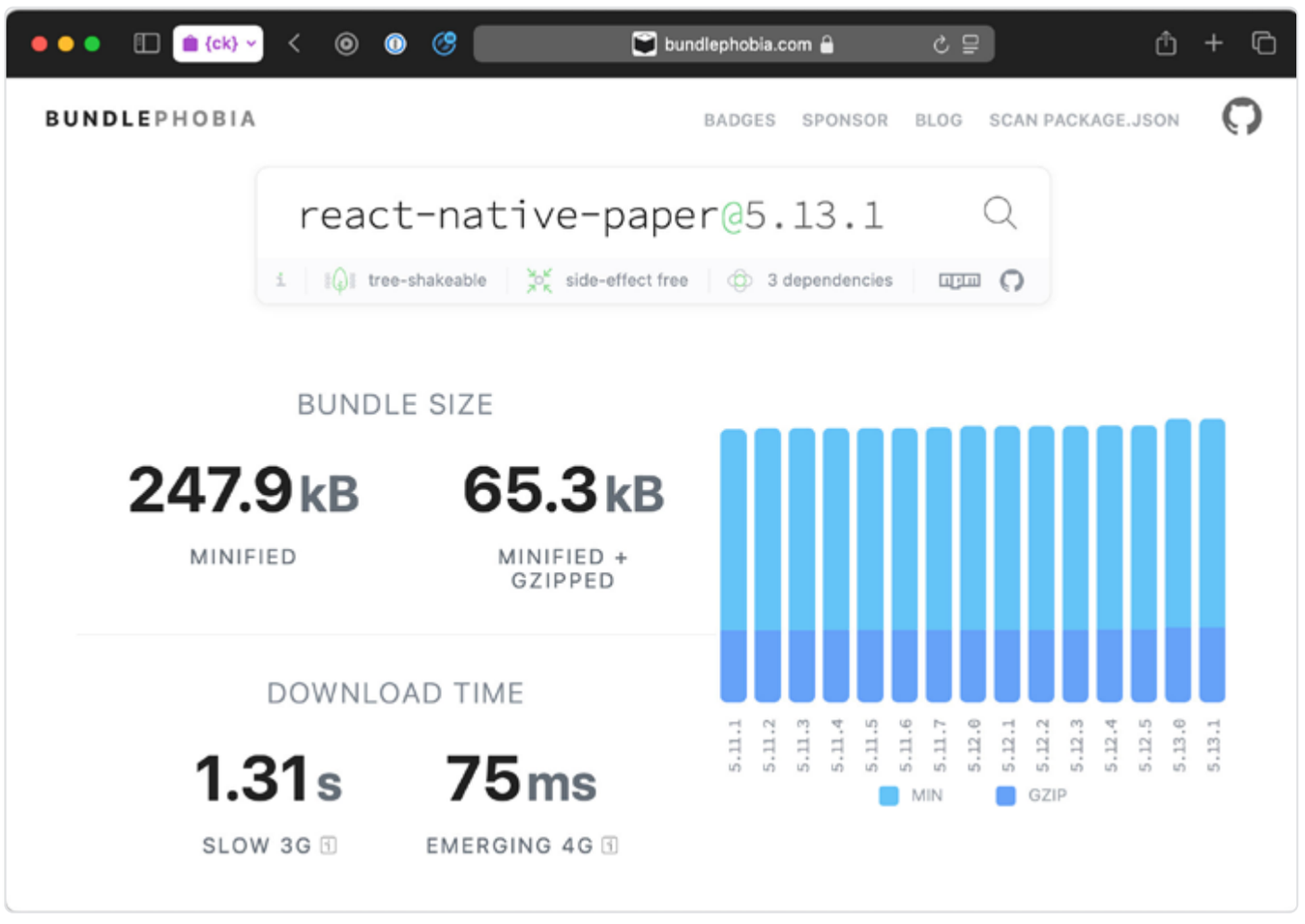
BEST PRACTICE

DETERMINE TRUE SIZE OF THIRD-PARTY LIBRARIES

In the JavaScript ecosystem, we love not doing the same thing twice. Pair that with the open-source culture of the open web platform, and it's no surprise we ended up with npm (Node Package Manager and an online package registry) holding millions of reusable JavaScript libraries that allow us to create apps faster, and often occupy gigabytes in our `node_modules` folders. Thanks to JS bundlers, only a fraction of that ends up in our React Native apps. In this chapter, we'd like to share the tools we use daily to determine the true size of third-party dependencies in our projects.

bundlephobia.com

This is one of our favorite websites. Pass it a package available in npm, and it will show you its minified size, compressed size, and even download time—which is especially useful for web apps.

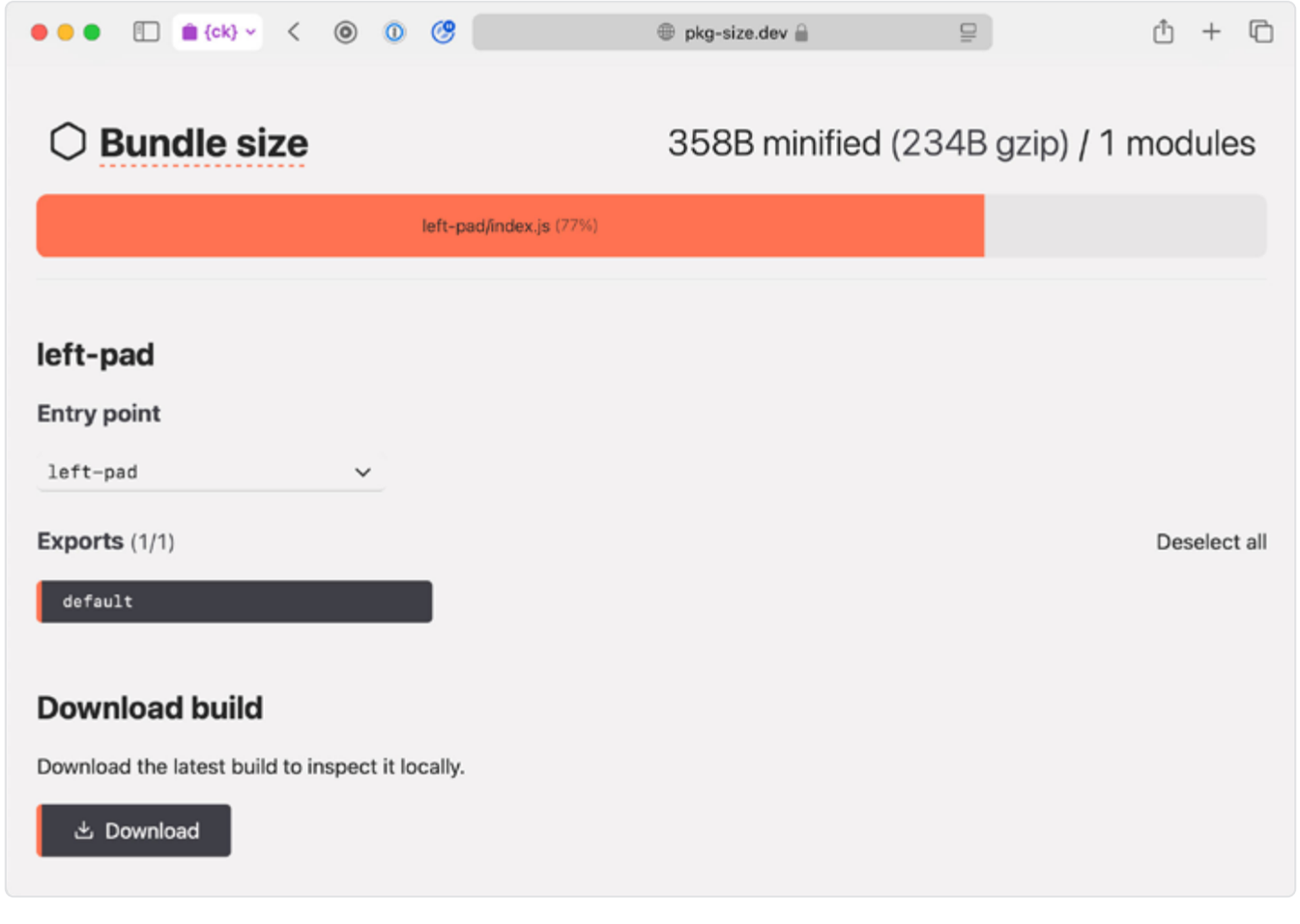


A screenshot of bundlephobia.com website showing react-native-paper dependency

One neat extra feature of Bundlephobia is package and dependency breakdown, similar to `react-native-bundle-visualizer`, which we describe later in this chapter.

pkg-size.dev

Sometimes, Bundlephobia just refuses to work with some packages or experiences an outage. In such cases, we have a backup plan: pkg-size.dev. It's a similar service that installs our package in a web container and presents us with its minified and compressed package size.



A screenshot of pkg-size.dev website showing left-pad dependency

Import Cost VS Code extension

The UX of all the tools described above revolves around occasional, one-time checks. However, you might want more real-time feedback. For such occasions, there's rarely a better way than direct integration with your IDE of choice.

In the case of VS Code or its forks, such as Cursor, you can use the Import Cost extension to achieve just that.

```
import React, {useMemo, memo} from 'react';  6.9k (gzipped: 2.7k)
import {View, Text, StyleSheet, Pressable} from 'react-native';
```

Inline comment showing size of imports from 'react' package

Import Cost works by running your imports through Webpack. Unfortunately, this bundler lacks native support for React Native, so the extension will often fail for packages with native code. Still, it can be useful for the rest of the packages.

You can use Webpack or Rspack as a bundler for your React Native project through Re.Pack.

BEST PRACTICE

AVOID BARREL EXPORTS

A barrel file in JavaScript is a way to group and export multiple modules from a single file. It allows for easier importing of the grouped modules by providing a centralized location to access them. These types of imports usually come from the so-called index files, which group files from multiple directories:

```
// components/index.ts -- an index file
export { Button } from './Button';
export { Card } from './Card';
export { Modal } from './Modal';
```

Later on, these modules can be easily imported from a single entry point:

```
import { Button, Card, Modal, Sidebar } from './components';
```

The problem with barrel exports and imports

While barrel imports might seem convenient, they introduce several significant problems:

Bundle size overhead

Metro bundler will include all modules from the barrel file in your final bundle, even if you only use one. For example, if you import just the **`Button`** component through a barrel file, Metro will still include the code for **`Card`**, **`Modal`**, and all other components exported from that barrel file.

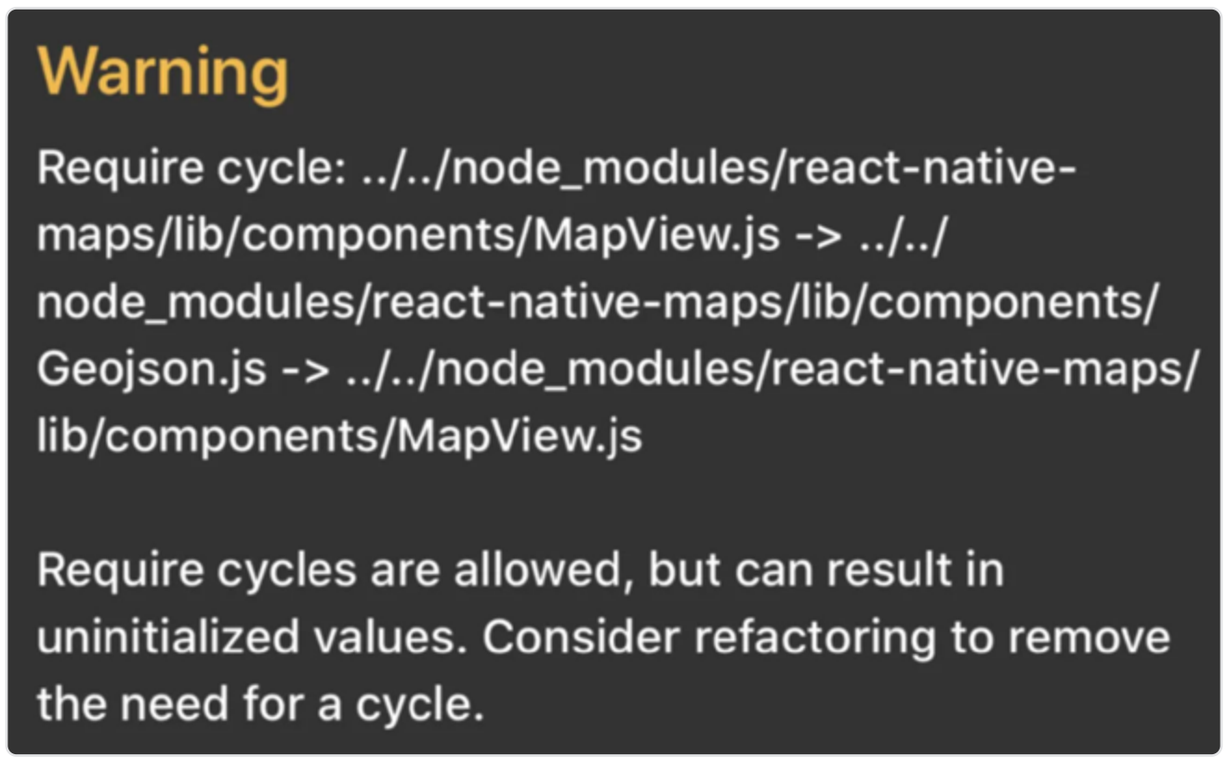
Some third-party libraries, such as React Native Paper, export a dedicated Babel plugin to eliminate unused imports, so you don't have to worry about unused components.

Runtime overhead

When using barrel imports, all modules in the barrel file must be evaluated before returning the requested module. This means that even if you only need one component, JavaScript still needs to process all the other modules in the barrel file, leading to unnecessary runtime overhead affecting the TTI metric.

Circular dependencies

The abstraction provided by barrel files makes it easier to accidentally create circular dependencies between modules. These circular dependencies can be particularly problematic for development workflows, often breaking Hot Module Replacement (HMR) and forcing full reloads whenever a module in the dependency cycle changes.



Example warning emitted by Metro when circular dependency is detected

Avoiding barrel imports altogether

The one-size-fits-all solution to the problems listed above is to avoid barrel imports altogether. Instead of doing this:

```
// importing from a barrel file - avoid!
import { Button, Card, Modal, Sidebar } from './components';
```

do this:

```
// importing from individual files - better!
import Button from './components/Button';
import Card from './components/Card';
import Modal from './components/Modal';
```

If you want to stick to this kind of rules in a project, it's best to add an **`eslint`** plugin to maintain this style of imports. You can use eslint-plugin-no-barrel-imports to do just that.

But let's be realistic—this is not the prettiest solution. There are ways to handle barrel imports automatically. In React Native, you can achieve the same effect as above with one of the following tools:

- **Expo SDK 52** introduces experimental support for tree shaking that can automatically optimize barrel imports—it will only include the modules actually used in the application. Read more about it here.
- **`metro-serializer-esbuild`** from **`rnx-kit`** makes it possible to enable tree shaking in Metro by letting ESBuild, which supports tree shaking as well, take over the bundling. Read more about it here.
- **Webpack and Rspack** ship with advanced tree shaking capabilities that can eliminate unused exports from barrel files. Read more about it here.

Real-world library example: `date-fns`

`date-fns` is a popular JavaScript date utility library that provides comprehensive tooling for manipulating dates. While it's possible to import everything from the main entry point, the library is specifically designed to support submodule imports for better control over what gets included.

So, instead of effectively importing the entire library:

```
// This imports the entire date-fns library
import { format, addDays, isToday } from 'date-fns';
```

we can split the imports to only include the code we need:

```
// These imports only include the code you need
import format from 'date-fns/format';
import addDays from 'date-fns/addDays';
import isToday from 'date-fns/isToday';
```

By using submodule imports, you ensure that only the functions you actually use are included in your final bundle. Make sure to measure the impact on the final JS bundle with the tools described in the How to Analyze JS Bundle Size chapter.

BEST PRACTICE

EXPERIMENT WITH TREE SHAKING

Tree shaking is a dead code elimination technique used in modern JavaScript bundlers to remove unused code from the final bundle. The term "tree shaking" comes from shaking a tree to remove dead leaves, similar to how we remove unused code from our JavaScript modules.

How tree shaking works

Tree shaking identifies and removes code that's never used in your application. It works in two steps:

1. Finding dead code

- Looks for code that's exported but never imported.
- Checks if removing the code is safe.
- Creates a map of what code depends on what.

2. Removing dead code

- Marks unused code for deletion.
- Removes it during build optimization.
- Keeps only the code your app actually uses.

Let's take a look at the following example code:

```
// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// app.js
import { add } from './math';
```

When this code gets tree shaken, **`subtract`** will be removed, as it is not used by the application. This is a fairly trivial example, but bundlers such as Webpack, Rspack, ESBuild, and others deal with more complex dependencies between functions and modules pretty well.

A library can be tree shaken **more effectively** when it's an EcmaScript Module (ESM). Historically, React Native always used CommonJS module system and transformed ESM libraries into CommonJS format by default through its Babel preset. Many libraries in the React Native ecosystem expect this transformation and might not work correctly without it.

In production mode, bundlers perform four main types of optimizations:

1. **Used Exports Check**—finds which exports are actually used and removes unused ones.
2. **Side Effects Check**—identifies code that modifies external state and ensures safe removal of unused code.
3. **Export Tracking**—maps connections between modules and tracks re-exports.
4. **Variable Usage**—follows how variables are used and prevents removing needed code.

Tree shaking in the React Native ecosystem

The support for tree shaking varies across different React Native development environments:

- **Metro**—currently does not support tree shaking, but you can fake it with **`metro-serializer-esbuild`** from **`rnx-kit`**.
- **Expo**—tree shaking is available as an experimental feature starting from Expo SDK 52.
- **Re.Pack**—fully supports tree shaking through its integration with Webpack and Rspack bundlers.

React Native is inherently a multi-platform framework. Through the **`Platform`** module, it's possible to define platform-specific behavior. Expo and Metro utilize this fact to enable a dead code elimination technique called platform-shaking—the removal of code dependent on the target platform (e.g., **`ios`**, **`android`** or **`web`**). It's important to note that this will only work if the **`Platform`** module is imported directly from **`react-native`** package.

Expo

To enable tree shaking in Expo SDK 52, enable the **`experimentalImportSupport`** in your Metro config and ensure your app builds and runs as expected. You may experience issues with required cycles or mixing CommonJS and ESM imports.

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

const config = getDefaultConfig(__dirname);

config.transformer.getTransformOptions = async () => ({
  transform: {
    experimentalImportSupport: true,
  },
});

module.exports = config;
```

metro.config.js file

Experimental import support uses Metro's version of the **`@babel/plugin-transform-modules-commonjs plugin`**, drastically reducing the number of resolutions and simplifying your output bundle. This feature can be used with **`inlineRequires`** (turned off by default in Expo projects) to further optimize your bundle experimentally.

Then edit your **`.env`** file to enable **`EXPO_UNSTABLE_METRO_OPTIMIZE_GRAPH`** and **`EXPO_UNSTABLE_TREE_SHAKING`** variables:

```
EXPO_UNSTABLE_METRO_OPTIMIZE_GRAPH=1
EXPO_UNSTABLE_TREE_SHAKING=1
```

.env file

This will only be used in production mode.

Re.Pack

Re.Pack offers the most battle-tested tree shaking implementation in the React Native ecosystem, thanks to the Webpack bundler introducing the concept for the JavaScript ecosystem and supporting it for years. It's turned on by default, and also works with Rspack, so you don't need to think about it.

What improvements can you expect?

We've created a comprehensive benchmark using Expensify, a large open-source React Native application, to compare bundle sizes between Metro and Re.Pack bundlers. To make the comparison complete, we've included different variants of the bundles produced by Metro and Re.Pack:

- **Development**—debug version with developer tools.
- **Production**—optimized release version.
- **Production Minified**—release version, which is then compressed for smaller size using Terser, the industry standard for JavaScript compression.

- **Production (HBC)**—same as Production, but converted to Hermes bytecode.
- **Production Minified (HBC)**—same as Production Minified, but converted to Hermes bytecode,

Bundle size comparison

You can find the repository with the benchmark here.

Bundle Type	Metro Size (MB)	Re.Pack Size (MB)	Difference
Development	38.49	49.04	+27.41%
Production	35.63	38.48	+8.00%
Production Minified	15.54	13.36	-14.03%
Production (HBC)	21.79	19.35	-11.20%
Production Minified (HBC)	21.62	19.05	-11.89%

We've experimented with tree shaking from Expo on the same codebase and were keen on including it in the benchmark, but the results were inconclusive. We've noticed regressions in the bundle size when compared with Metro, which we believe is due to misconfiguration. This, combined with the fact that the feature is marked as experimental, prevented us from including it in the benchmark results.

The larger size of Re.Pack's production bundle (+8.00%) is surprising at first but expected, as Re.Pack (using Rspack) only marks unused code, which is then removed during minification. Conversion of a bundle to Hermes bytecode includes minification, which is why we see significant improvements in minified bundles (-11% to -14%), even when not using Terser beforehand.

Overall, you can expect about **10-15%** reduction in your final bundle size when using Re.Pack with all optimizations enabled. We didn't measure the impact on the TTI metric, but seeing a reduction in Hermes bytecode size, we should expect a slightly faster app boot time and lower memory footprint.

BEST PRACTICE

LOAD CODE REMOTELY WHEN NEEDED

On the web, it's native to developers to load their code from a remote server. After all, this is how users access their apps—through clients (such as browsers) loading HTML, CSS, JS, and other assets from a server. The situation is slightly different on mobile, where this pattern is harder to execute. Users download the apps from app stores, and our JS code is already there, available offline. It's possible, however, to leverage remote code loading with React Native. You'll need to adopt a different mindset when using it compared to the web, though.

When to consider remote code loading

The effectiveness of dynamically loading code from a remote, or code-splitting, varies significantly depending on your JavaScript engine choice. With Hermes (React Native's default engine), the benefits of loading code on demand are minimal due to its memory mapping technique that already efficiently reads only necessary parts of the bundle's bytecode. However, if you're using JavaScriptCore or V8, code-splitting can still meaningfully improve initial loading times by loading chunks on demand.

Therefore, there are very few cases a modern React Native application would benefit from code-splitting—but still, there are some. You can consider trying code-splitting when:

- You're not using Hermes JS engine.
- You have a specific need for remote code loading.
- Your app size is significantly impacting user experience.
- Your app size is too big for Google PlayStore (200 MB).
- You're building a microfrontend architecture using Module Federation.
- You've exhausted other optimization techniques.

Moving to Re.Pack

The default Metro bundler does not support code-splitting out of the box for production builds. To implement this feature, you'll need to switch to Re.Pack—a tool built on top of Webpack that enables bundling React Native. While this transition requires some additional work with migrating the existing configuration, it may be worth it when you've exhausted other options for optimizations.

To get started with Re.Pack, the best way is to use its initialization tool:

```
> npx @callstack/repack-init
```

The command will apply necessary changes to your project files and guide you through that process. Once that's done, the next time you run **`npm start`**, your JS will be served by a Re.Pack dev server with Webpack or Rspack as a bundler instead of Metro. As much as Re.Pack tries to be an almost drop-in replacement for Metro, you may hit some incompatibilities, make sure to follow the official migration guide.

Re.Pack is a tool developed at Callstack. Recently, we've added support for Rspack which drastically improves build times, making it even faster than Metro in most cases.

Splitting the bundle

The most common way to split your code is to use dynamic imports along with React's **`lazy`** function and **`Suspense`** component. Introducing a splitting point typically looks like this:

```
// Before - regular import
import FeatureComponent from './FeatureComponent';

// After - creating a split point
const FeatureComponent = React.lazy(() =>
  import(/* webpackChunkName: "feature" */ './FeatureComponent')
);
```

Introducing code-splitting through dynamic import

These changes will cause the **`FeatureComponent`** to be placed in a separate file when bundling the app, called **`feature.chunk.bundle`**.

Here's how it looks in practice when we want to implement code splitting for a screen in an app:

```
-import React from 'react';
+import React, { Suspense } from 'react';
-import SettingsScreen from './screens/SettingsScreen';

+const SettingsScreen = React.lazy(() =>
+  import(/* webpackChunkName: "settings" */ './screens/
SettingsScreen')
+);

 const App = () => {
   return (
+    <Suspense fallback={<LoadingSpinner />}>
       <SettingsScreen />
+    </Suspense>
   );
 };
```

Conversion from a component that is imported synchronously to one imported dynamically

Lazy loading code from a remote location

The comments marked as **`webpackChunkName`** allow us to name the chunk, which is useful in the last step—telling Re.Pack how to load that chunk, which is no longer part of the main bundle and that we can download from a remote server:

```
// index.js
import { ScriptManager, Script } from '@callstack/repack/client';

// Tell Re.Pack where to load chunks from
ScriptManager.shared.addResolver((scriptId) => ({
  url: __DEV__
    // In development, load from the dev server
    ? Script.getDevServerURL(scriptId)
    // In production, load from a CDN
    : `https://my-cdn.com/assets/${scriptId}`,
}));

// Then, run the app just like before
AppRegistry.registerComponent(appName, () => App);
```

Using ScriptManager to tell Re.Pack how to obtain the asynchronous chunks in an app

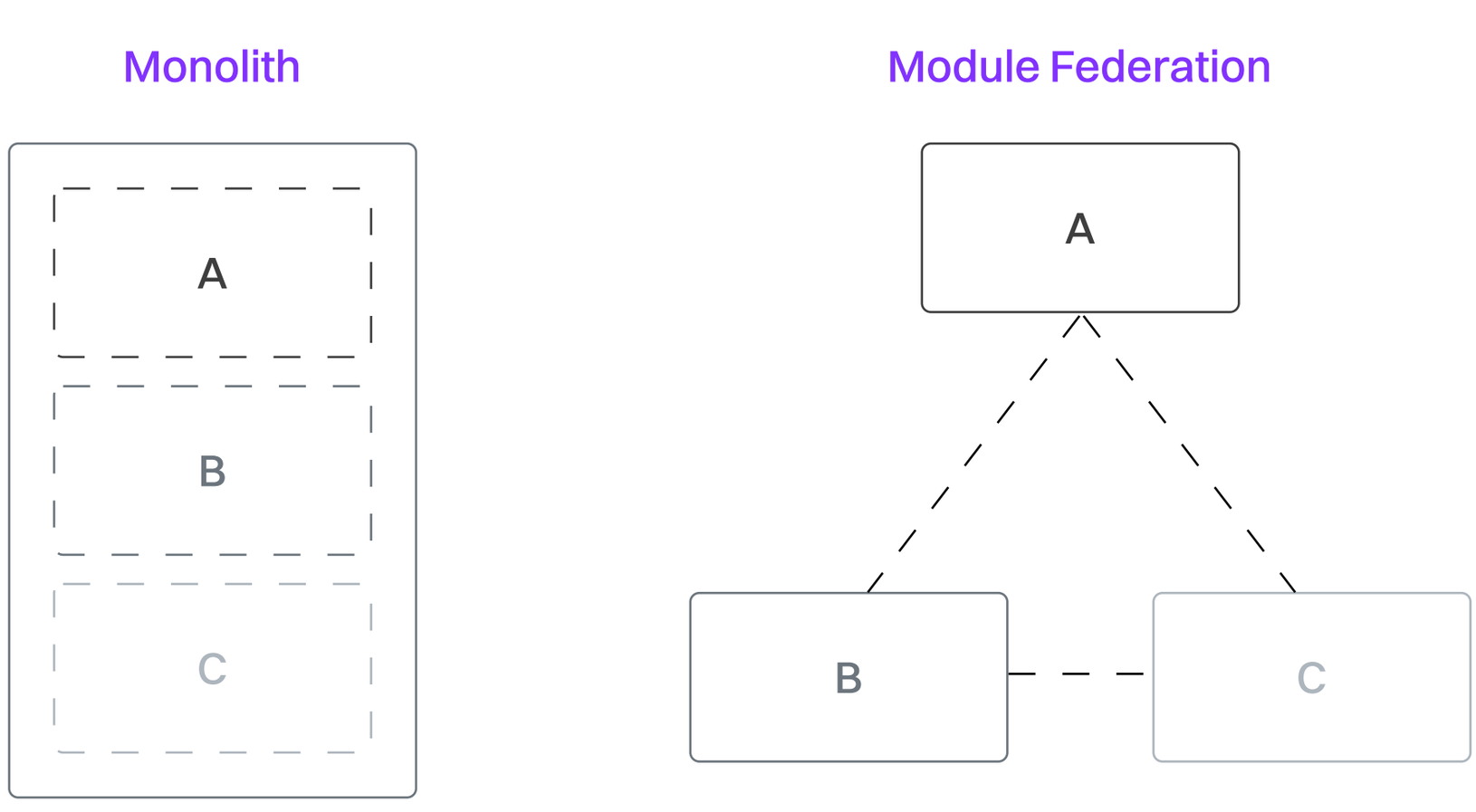
`ScriptManager` is a runtime client of Re.Pack, which you can use to configure the behavior of finding and loading chunks other than the main bundle. These steps are all that is required for the most basic case of code-splitting.

Many more elements come into play when using code-splitting, such as caching or deploying to a production CDN. Refer to the official Re.Pack documentation for more information on how to achieve the desired effect with code-splitting.

Module Federation

One of the more compelling use cases for code-splitting is the ability to load only the features that specific users need and not load them for those who don't. For example, in a large enterprise application, different user roles might require access to entirely different functionality sets. Instead of bundling everything together, you can load features on-demand based on user permissions or preferences. This approach optimizes the initial load and provides a more tailored experience.

For more complex scenarios where you need to manage multiple isolated teams working on different parts of the application or want to give teams more autonomy in development and deployment, you might consider a microfrontends architecture with Module Federation. It's a popular and battle-tested solution for microfrontends on the web and mobile thanks to Re.Pack.



Monolith architecture vs decentralised one with Module Federation

However, it's important to note that microfrontends should not be chosen solely for bundle size optimization. This architecture comes with its own complexity and should be considered only when organizational and development workflow benefits outweigh the additional overhead.

One of the complexities of microfrontends is version management. It's challenging to handle on your own and can be quite cumbersome based on our experience. If you're open to third-party services, we recommend Zephyr Cloud, which simplifies this problem, allows for sub-second deploys, and officially integrates with Re.Pack.

BEST PRACTICE

SHRINK CODE WITH R8 ANDROID

As mentioned in the How to Analyze App Bundle Size chapter, app size is an important metric that you should keep an eye on. In the previous chapters, we discussed how you can optimize the size of your JavaScript bundle. However, in some cases, native code can affect the app size too. Using tools described in this section can help you shave off additional megabytes from the size of your app and make it safer by using obfuscation techniques.

Enable R8

You may have heard of ProGuard. It's a tool in Android Gradle that shrinks, optimizes, and obfuscates your app's code to reduce APK size. However, if your project is using React Native v0.60 or higher, the Android Gradle Plugin, which is a core dependency for most Android projects, doesn't optimize your code with ProGuard anymore. It's using R8.

R8 is a tool in Android that replaces ProGuard to shrink, optimize, and obfuscate APKs, offering improved performance and compatibility with modern features, such as Kotlin, which ProGuard didn't support. It's also faster and compatible with ProGuard tooling, such as its configuration files, so you may still see some "proguard" references in your Android files, and that's fine.

To enable R8 in a React Native project, edit the **`app/build.gradle`** file:

```
def enableProguardInReleaseBuilds = true
```

Enabling proguard in android/app/build gradle

In the default React Native template, this variable will set the **`minifyEnabled`** setting to **`true`** in the release build variant, which tells Gradle to apply R8 optimizations to your apps when bundling for distribution.

In the How to Analyze App Bundle Size chapter, we tested how much enabling R8 with default ProGuard rules can shrink the app size. The sample app mentioned there was 9.5 MB without

optimizations. After enabling R8, we got 6.3 MB, a 33% difference. Larger projects should expect slightly smaller improvements, but it is worth doing so.

ProGuard rules and configuration for R8

As we already mentioned, R8 is configured with "ProGuard rules". In a dedicated file, you can control which classes, methods, and fields should be kept (preserved) or discarded during the build optimization process. Additionally, the rules let you configure code obfuscation patterns (like class/method renaming) to make reverse engineering more difficult. In a React Native app, these rules are defined in **`app/android/proguard-rules.pro`**. This file is empty by default.

In most cases, you don't need to define any rules manually. However, if you run into issues with your app not working correctly after dead code elimination, you may need to add classes to exclude. This may be necessary for certain dependencies making assumptions about class names. For example, if you are running into issues using Firebase, you can tell R8 to keep those classes (prevent removal):

```
# Firebase
-keep class io.invertase.firebase.** { *; }
-dontwarn io.invertase.firebase.**
```

proguard-rules.pro file

Obfuscation is enabled by default when specifying **`minifyEnabled`**. If, for some reason, you want to disable it, use **`-dontobfuscate`** setting in your **`proguard-rules.pro`** file:

```
-dontobfuscate
```

proguard-rules.pro file

You can read more about defining ProGuard rules in the official documentation.

Shrinking resources

Once you have R8 enabled in your release build variants, you can unlock further optimizations thanks to Gradle's resource minification. This can reduce the app bundle even further. When enabled, it performs several optimizations:

1. Merges duplicate resources.
2. Optimizes PNG files by reducing their color depth and applying lossless compression.
3. Processes vector drawables.

It can be enabled inside of **app/build.gradle** file using **shrinkResources** setting:

```
android {
    buildTypes {
        release {
            // Enables code shrinking, obfuscation, and
optimization for only
            // your project's release build type. Make sure to
use a build
            // variant with `debuggable false`.
            minifyEnabled true

            // Enables resource shrinking, which is performed by
the
            // Android Gradle plugin.
            // Make sure `minifyEnabled` is set to `true`
            shrinkResources true
        }
    }
}
```

app/build.gradle file

Now you know how to make your app smaller by enabling dead code elimination. Remember you can use the Ruler tool to verify the difference between your app's downloaded and installed size. Make sure to give your app a good round of testing before shipping it to the store with this technique enabled.

BEST PRACTICE

USE NATIVE ASSETS FOLDER

A fair chunk of the app size typically comes from assets, such as images, sounds, or videos bundled into it. Both iOS and Android have ways to optimize the delivery of such assets based on the device's screen size or pixel density. For example, it's a good practice to load larger images on a mobile phone so that they appear crisp and clear. Mobile devices typically have large resolutions packed into a small screen, which translates to high pixel density. With more pixels on the screen, we can show more; hence, larger images are preferred. When used correctly, these platform optimization mechanisms can help us load only necessary image resolutions when the user downloads the app from the application store.

Remember to also optimize your assets using dedicated tools to reduce their size.

Using size suffixes

When dealing with assets inside a React Native application, it is generally recommended to create multiple size variants of the same image: a base size (1x), twice the size (2x), and thrice the size (3x). The convention is to use size suffixes in file names, such as `@2x` or `@3x`.

```
assets/
├── image.jpg      // 1x resolution
├── image@2x.jpg   // 2x resolution
└── image@3x.jpg   // 3x resolution
```

Directory structure for logo.png with size density extensions

When you import such an image with standard **`require`**, React Native selects the best one, depending on the current device's screen density.

Native assets folders

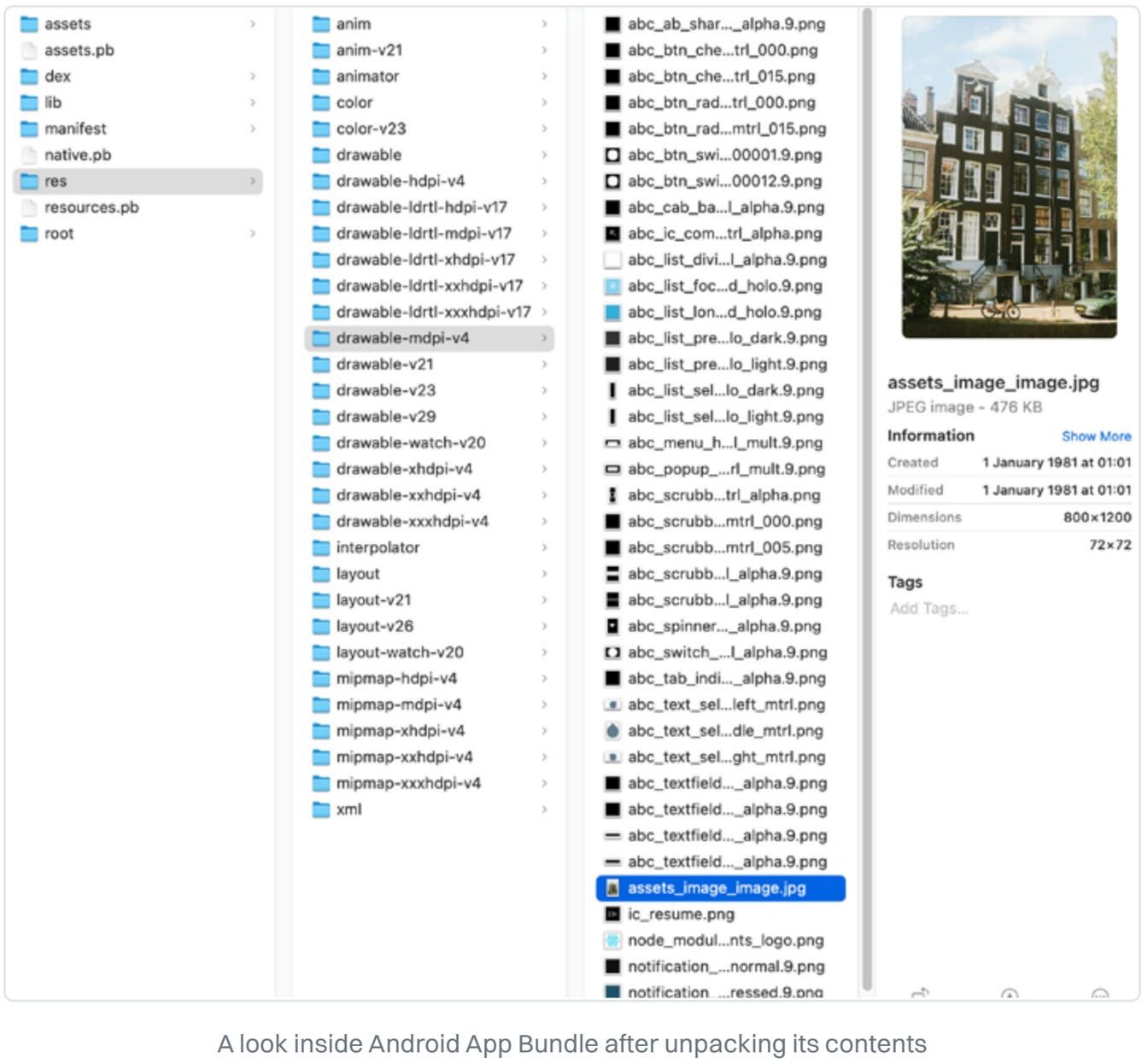
As long as your images have size suffixes, Android will apply this optimization automatically. Let's use this as an opportunity to look under the hood and better understand what is going on.

To do so, we will first create an AAB file by running the following command:

```
> ./gradlew bundleRelease
```

Running this command will produce an Android App Bundle that the Play Store can use to produce application binaries on-demand, optimized for the user's device architecture. If you want to learn more about this and other formats, head back to the Introduction.

Once done, head over to **`android/app/build/outputs/bundle/release/app-release`** folder. In order to inspect the contents of the bundle, we will unpack it by changing its extension to **`.zip`** first (after all, AAB is just a ZIP archive).



A look inside Android App Bundle after unpacking its contents

After heading to the resources catalog, you will see multiple folders with names corresponding to Android screen densities, such as **`drawable-xhdpi-v4`**, and **`drawable-xxhdpi-v4`**. Looking at the images inside, you will notice your suffixes, such as **`@2x`** are gone. Instead, images were located in each folder based on the best density available.

When a user downloads the application from Google Play, the on-demand application for their respective device will contain assets for that density only, without everything else.

Discussing the process of on-demand compilation is beyond the scope of this guide. However, if you want to learn more about this topic, you may find the following resource(s) interesting.

Xcode Asset Catalog

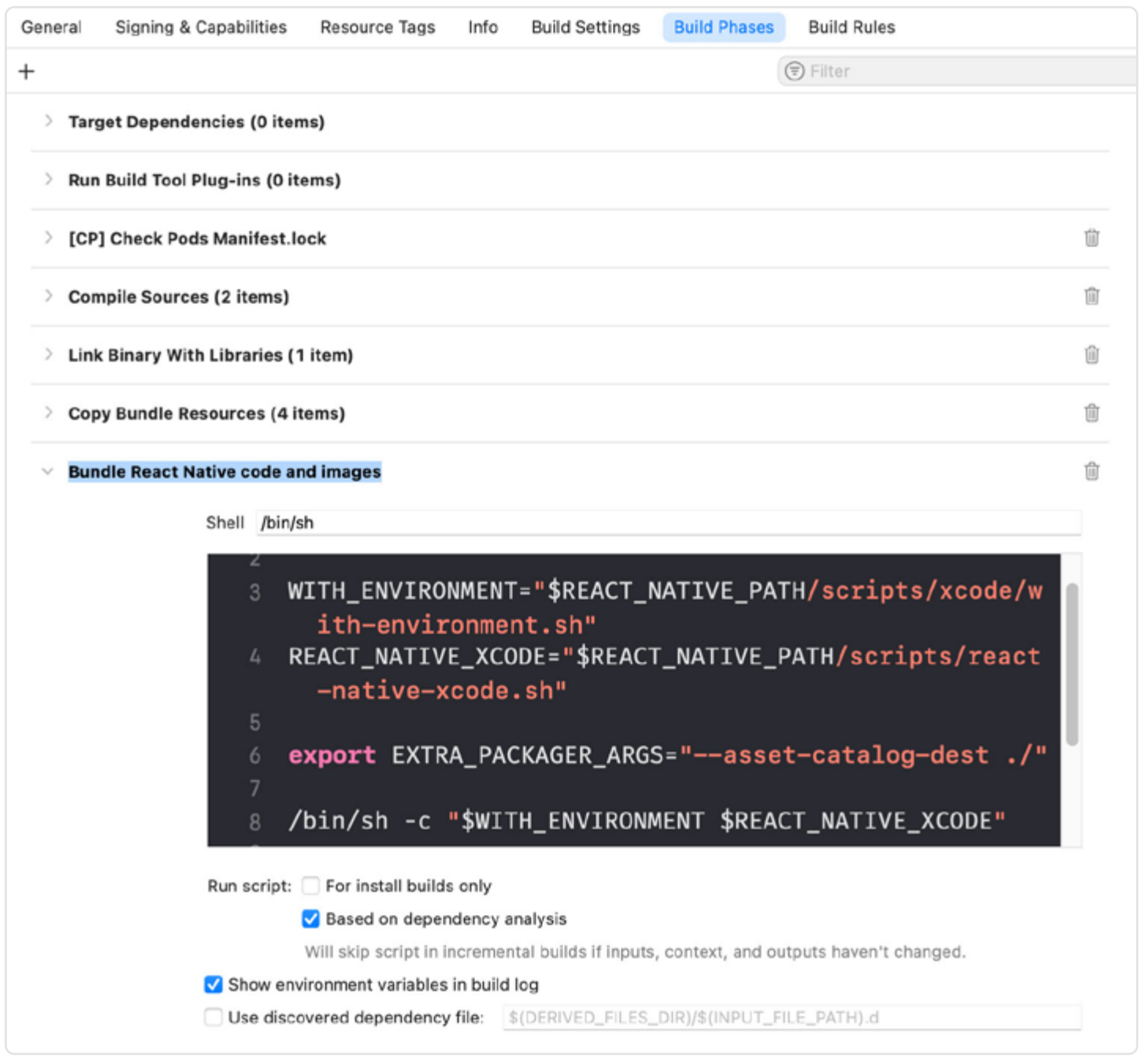
We can achieve similar results on iOS. However, unlike on Android, it is not done automatically by React Native and Metro while creating the bundle. In order to benefit from similar optimization, we must explicitly create an *Xcode Asset Catalog* (**`.xcassets`**) and configure it as part of our compilation pipeline.

To get started, let's create the **`RNAssets.xcassets`** file and place it inside the **`ios`** folder.

This folder has to be named **`RNAssets.xcassets`** due to the hard-coded path inside React Native bundle scripts. Hopefully, this constraint will be dropped in the future.

After creating the folder, navigate to **`Build Phases`** from within your Projects Settings and modify the **`Bundle React Native code and images`** build phase with **`EXTRA_PACKAGER_ARGS`** variable above line 8:

```
export EXTRA_PACKAGER_ARGS="--asset-catalog-dest ./"
```



Bundling code and images happens as one of build phases

As a result, when that build phase gets executed, Metro will place assets inside **`RNAssets.xcassets`** catalog. After running the build for the first time, your asset folder should look like this:

A look into Xcode Asset Catalog with sample images

You can also trigger the bundle command manually if you run into issues or want to test if everything works (notice that **`--asset-catalog-dest`** now points to **`ios`** because we run this script from the root of the React Native project):

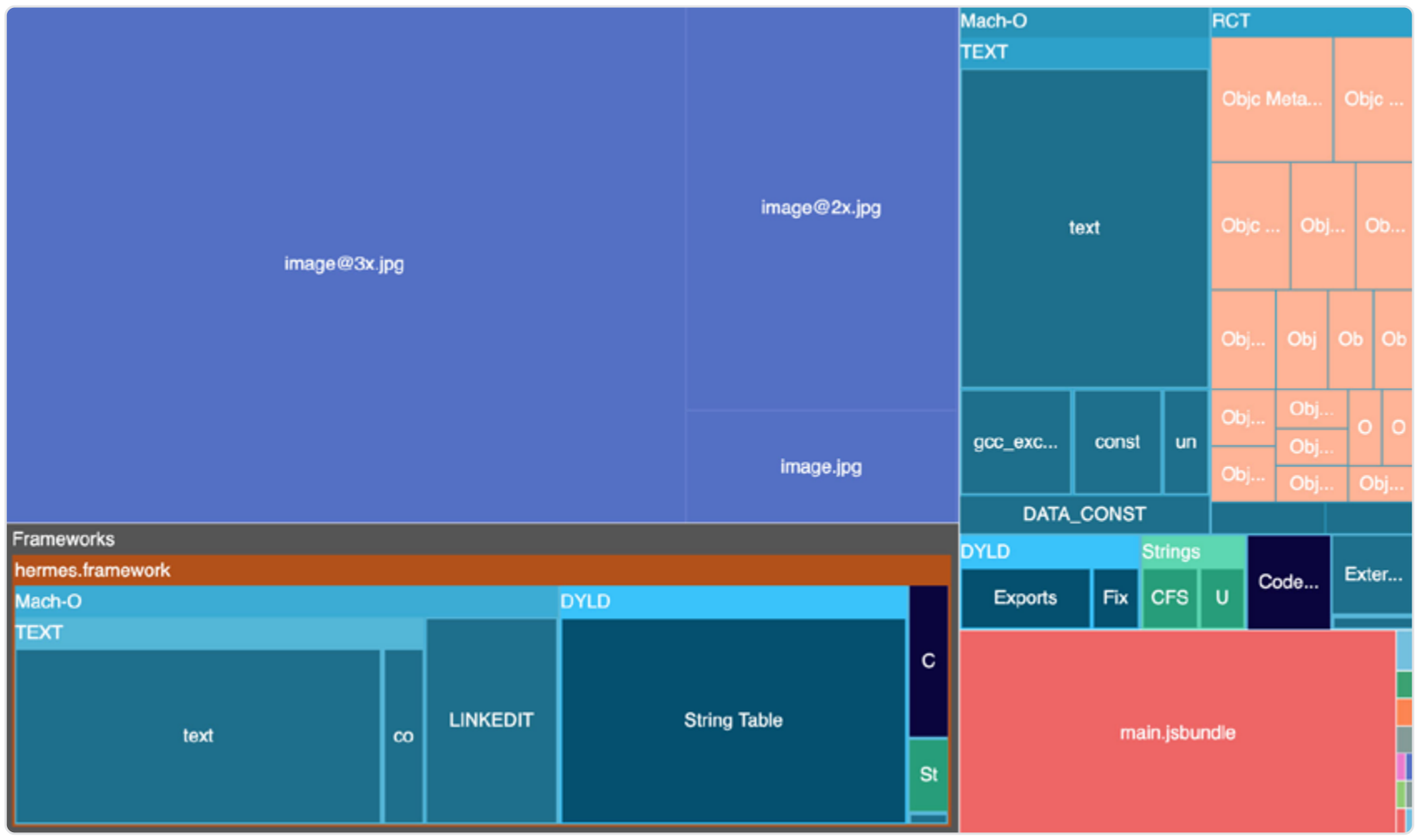
```
> npx react-native bundle \
  --entry-file index.js \
  --bundle-output output.js \
  --platform ios \
  --dev false \
  --minify true \
  --asset-catalog-dest ios \
  --assets-dest <your-assets-folder>
```

Now, after uploading your app to the App Store, the app thinning mechanism will kick in and choose proper image sizes based on the user's device!

On iOS, "App Thinning" optimizes app delivery by creating device-specific versions, including only required assets. Android achieves similar results through "Android App Bundles" (AAB), which generate optimized APKs for each device configuration, breaking apps into modular components that can be downloaded on demand.

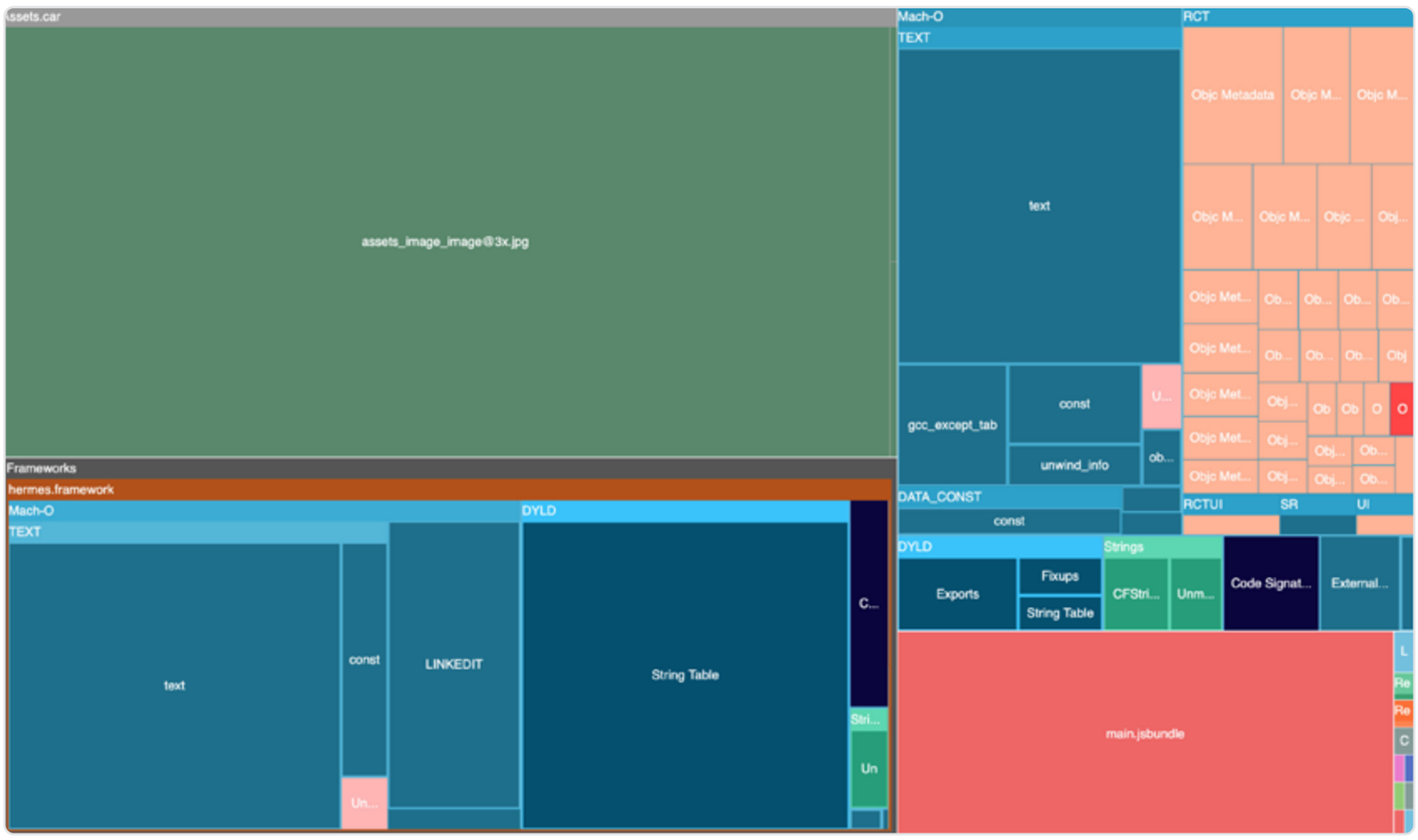
Testing it out

Let's examine the real difference on iOS, before and after applying this optimization. Without it, your bundle will include all three images, which is a waste because your device will only use one size.



A screenshot from Emerge Tools before optimization of assets catalog on iOS

After applying the abovementioned technique and using the assets catalog, you will see that only the relevant image (`assets_image_image@3x.jpg`) is present in the bundle.



A screenshot from Emerge Tools after enabling app thinning

It's a great optimization technique that ensures your app includes only necessary images for a given platform. Overall, it has a positive impact on the app size metric.

At the time of writing this chapter (January 2025), Assets Catalog is not enabled by default; however, there are ongoing efforts to make it the default choice.

BEST PRACTICE

DISABLE JS BUNDLE COMPRESSION

We already learned that APK and Android App Bundles are compressed archives. The Android build system applies compression to files during packaging primarily to reduce the overall size of the APK or AAB, which can be beneficial for reducing download sizes and storage usage on devices. It does so without necessarily distinguishing between file formats, beyond generally skipping already efficiently compressed formats like image files (e.g., **`.png`**, **`.jpg`**).

In an Android React Native app using Hermes, the final JS bundle is packaged into the **`index.android.bundle`** file. Although binary, it might be treated like other resources packaged in the app because the build system applies compression to most resource files by default unless explicitly told not to.

What's intriguing is that—as a compressed file—it will be skipped during the memory mapping (mmap) procedure, as investigated in this Pull Request to the React Native core. This prevents one of the most important Hermes optimizations from working properly, negatively affecting the TTI metric.

As of writing this note (February 17, 2025), the React Native Core Team is including this behavior by default. It's most likely going to be available in React Native 0.79.

To opt out of this behavior, regardless of the React Native version used, and fully leverage Hermes potential, you can manually tell the Android build system to disable compression for files with **`bundle`** extension by setting **`androidResources`** in your **`app/build.gradle`** file:

```
android {
+    androidResources {
```

```
+         noCompress += ["bundle"]
+     }
  }
```

app/build.gradle file with compression disabled for .bundle files

As a result, **`the index.android.bundle`** file won't be compressed, effectively increasing the overall app bundle size. This size increase will affect the install size of your app but not the download size. The most important bit is that your app should load significantly faster now due to Hermes being able to leverage memory mapping and load the JS bundle faster. In our initial testing, for an APK of 75.9 MB install size, it was a 6.1 MB size increase (+8%), while the TTI dropped by 450 ms (-16%).

It's possible, but not very likely, that your app won't grow in size for various reasons. Measure the outcomes and observe your users' behavior to ensure this change is a net positive rather than a net negative. Most likely, though, the benefits will outweigh the cost.

ABOUT THE AUTHORS

The Ultimate Guide to React Native Optimization was brought to you by the Callstack team

Mike Grabowski, CTO and Founder at Callstack

Passionate about cross-platform technologies. When not working, find him on a race track.

@grabbou @grabbou

Oskar Kwaśniewski, Senior Software Engineer

Passionate about bridging the gap between Native and JavaScript development. React Native Core Contributor, creator of React Native visionOS and React Native Bottom Tabs. In free time, he enjoys doing outdoor sports & reading.

@o_kwasniewski @okwasniewski

Robert Pasiński, Senior Software Engineer

Passionate about all things development, low-level coding enthusiast.

@rob_pasinski @robik

Michał Pierzchała, Principal Engineer

Passionate about building mobile and web tooling in the Open Source. Core React Native Community contributor.

@thymikee @thymikee

Jakub Romańczyk, Senior Software Engineer

JS Infra nerd. Passionate about React Native, OSS & JS tooling. Souls-like games enthusiast.

@_jbroma @jbroma

Szymon Rybczak, Software Engineer

18-year-old, passionate about open-source technologies around universal apps.

@szymonrybczak @szymonrybczak

Piotr Tomczewski, Expert Software Engineer

Father of two cats, a foodie, will code for LEGO.

@Piotrski @piotrski

LIBRARIES AND TOOLS MENTIONED IN THIS GUIDE

Animation and performance optimization

- React Native Reanimated—a high-performance animation library for React Native.
- React Navigation—a navigation library for React Native supporting animated transitions.

React Compiler

- Jotai—a minimalistic atomic state management library for React.
- Zustand—a scalable state management library for React, simpler than Redux.
- Recoil—a state management library by Meta with fine-grained reactivity.

Lists and scrolling performance

- FlashList—a high-performance replacement for FlatList, developed by Shopify.
- Legend List—a new alternative to FlatList, optimized for React Native's New Architecture.
- RecyclerListView—a performant list library for React Native, used by FlashList.s.

Profiling and debugging tools

- React Native DevTools—a built-in debugging and profiling tool for React Native.
- Chrome DevTools—a browser-based tool for profiling memory and performance issues.
- Flashlight—a tool for measuring JS FPS and collecting performance metrics on Android.

- Lighthouse—Google's web performance auditing tool, referenced for benchmarks.
- Xcode Instruments—a macOS tool for profiling memory usage, CPU, and performance bottlenecks.
- Android Studio Profiler—a built-in tool for tracking CPU, memory, network, and battery usage in Android applications.
- Perfetto—a system tracing tool for deep profiling of Android applications.

JavaScript engines and native execution

- Hermes—a lightweight, high-performance JS engine optimized for React Native.
- Yoga—a cross-platform layout engine used by React Native for calculating component layouts.

Code optimization and shrinking

- R8—Android's default tool for shrinking, optimizing, and obfuscating APKs, replacing ProGuard.

Code splitting and remote loading

- Re.Pack—a Webpack-based bundler for React Native that enables code-splitting and remote module loading.
- Module Federation—a Webpack feature that enables microfrontend architectures and dynamic code loading.
- Zephyr Cloud—a platform that simplifies managing and deploying microfrontend architectures with module federation..

Tree shaking

- Rspack—a high-performance bundler similar to Webpack, optimized for tree shaking in React Native.
- Terser—a JavaScript minification tool used to reduce bundle size and optimize performance.
- metro-serializer-esbuild—a plugin that enables tree shaking in Metro by integrating Esbuild.

JS and app bundle size analysis

- source-map-explorer—a tool to analyze JavaScript bundles and identify unused dependencies.
- Expo Atlas—a tool for visualizing JS bundle sizes in Expo projects.
- webpack-bundle-analyzer—a Webpack plugin for interactive visualization of bundle sizes.
- bundle-stats—a CLI tool for generating JS bundle analysis reports.

- Statoscope—a tool for deep analysis of Webpack and Rspack bundles.
- Rsdoctor—a tool for analyzing React Native bundles built with Rspack.
- Emerge Tools—a paid service for analyzing and optimizing Android and iOS app sizes.
- Ruler—a Gradle plugin by Spotify for analyzing APK and AAB sizes.

Third-party library size analysis

- Bundlephobia—a tool that shows the minified and gzipped size of npm packages.
- pkg-size.dev—a web-based tool for checking package sizes, similar to Bundlephobia.
- Import Cost (VS Code Extension)—a VS Code extension that shows the size of imported modules inline.

Native development tools and build systems

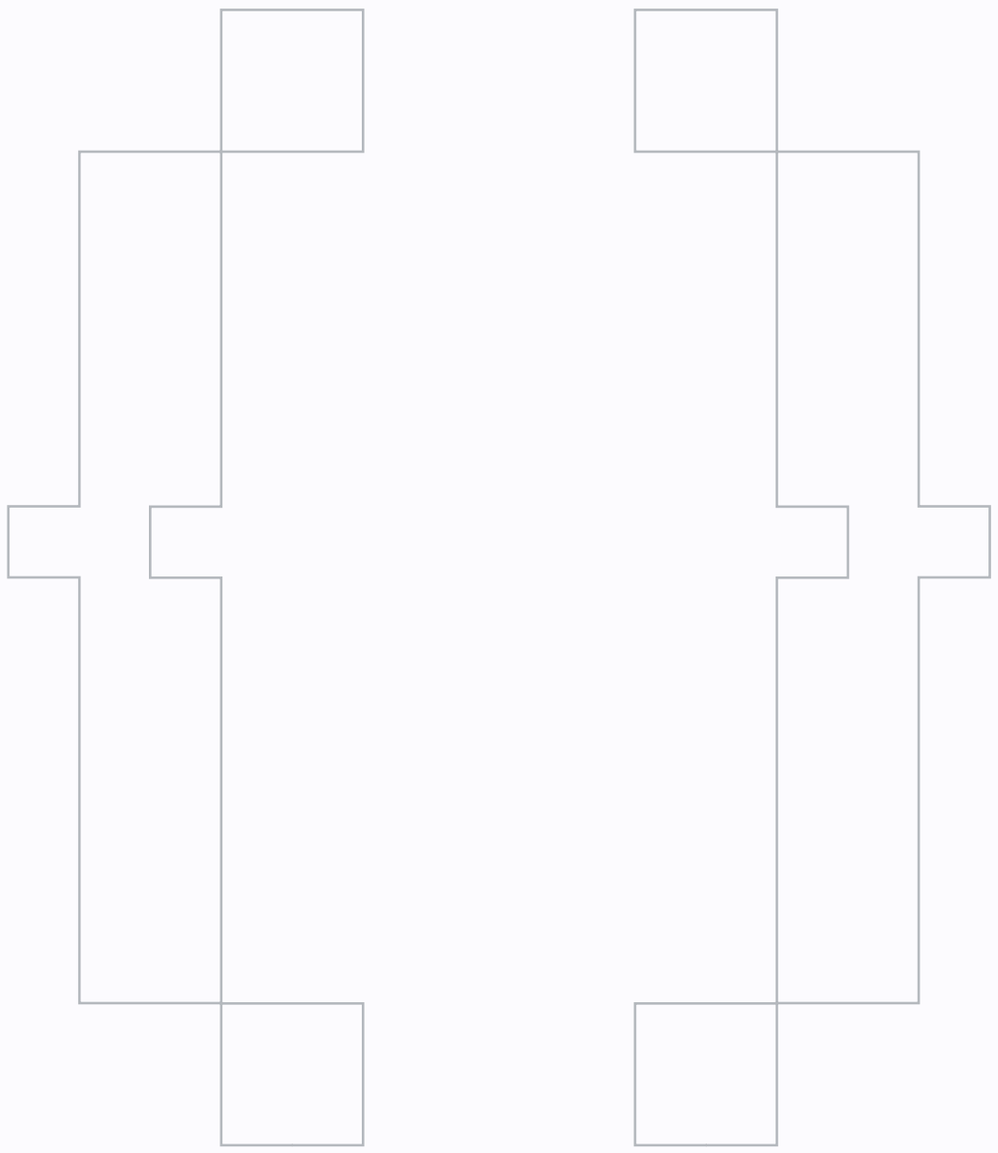
- React Native Builder Bob—a tool for scaffolding new React Native libraries with support for the new architecture.
- Ninja—a fast build system commonly used for compiling native code in React Native projects.

Measuring app performance with Time to Interactive

- react-native-performance—a tool for tracking React Native app performance markers like TTI.

{callstack}

Callstack is a team of React Native experts, core contributors, and Meta partners helping developers build high-performance, scalable applications through Open Source.



ABOUT THIS GUIDE

Optimizing React Native apps requires more than just intuition—it calls for a deep understanding of how JavaScript, native code, and platform-specific build processes work together. This guide provides a systematic approach to diagnosing and resolving performance bottlenecks, helping you develop apps that boot fast and are always responsive to user input.

Learn how to measure, interpret, and optimize Time to Interactive (TTI) and Frames Per Second (FPS)—the key indicators of app responsiveness and fluidity. Discover techniques to speed up startup times, eliminate janky animations, and ensure smooth UI interactions. With practical guidance on profiling, understanding platform differences, and implementing best practices, this guide will equip you with the skills to make data-driven optimizations and keep your React Native app running at peak performance.