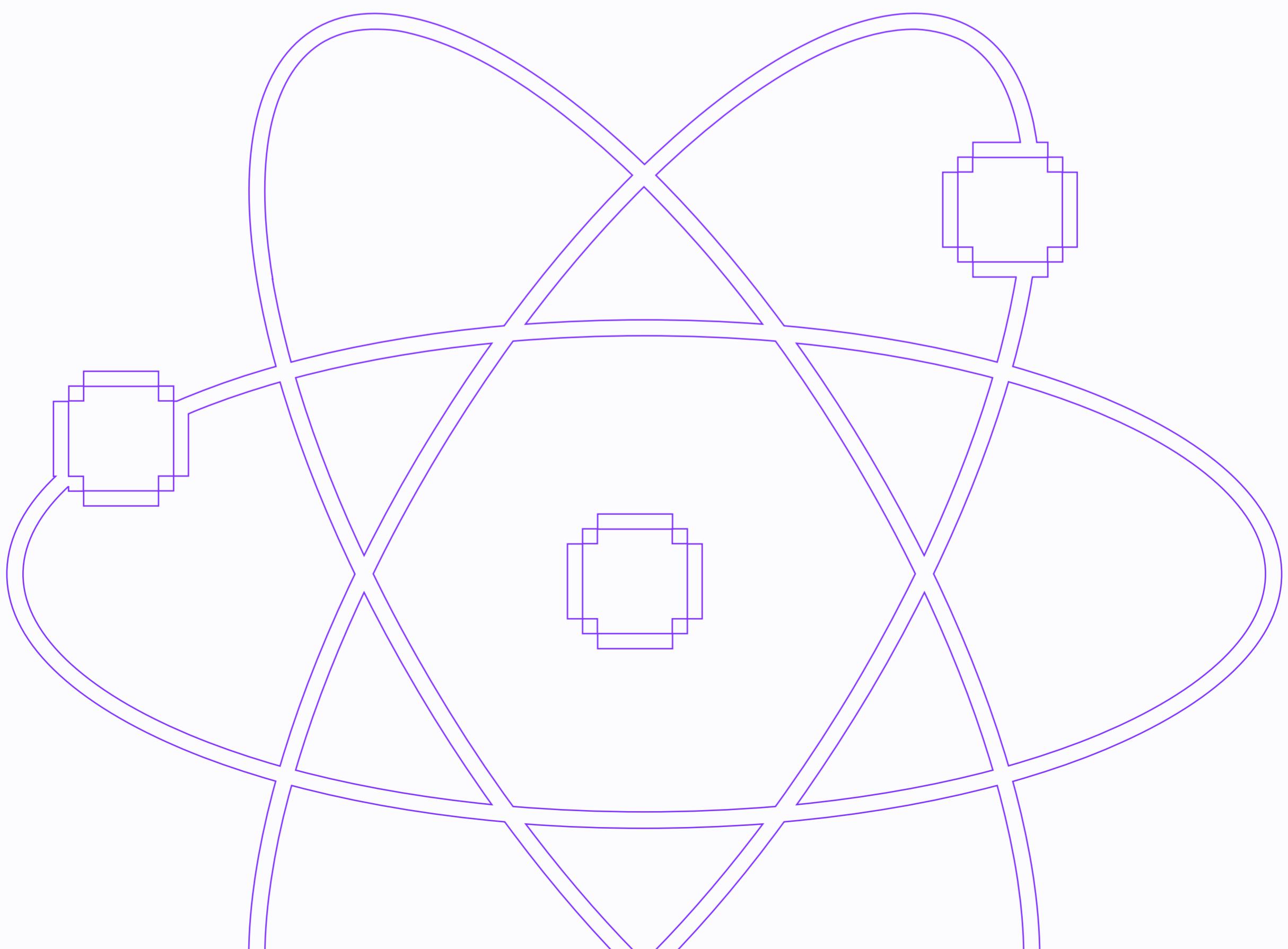


{callstack}

React Native Optimization

THE ULTIMATE GUIDE



목차

소개

머리말	4
이 책을 읽는 방법	5
성능이 중요한 이유	7

Part 1: JavaScript

JS, React 코드를 프로파일링하는 방법	14
JS FPS를 측정하는 방법	21
JS에서 메모리 누수를 잡는 방법	24
비제어 컴포넌트	30
HoC(Higher-Order Specialized Components)	34
아토믹 상태 관리	42
React 동시성	46
React 컴파일러	52
프레임 드롭없이 고성능 애니메이션 구현하기	59

Part 2: Native

플랫폼간 차이 이해하기	67
React Native의 Native 부분 프로파일링하기	76
TTI 측정하기	85
Native의 메모리 관리 이해하기	93
Turbo Modules, Fabric의 스레딩 모델 이해하기	105
View Flattening 사용하기	113
웹을 통해서 React Native 전용 SDK들을 사용하기	117
Native Module을 더 빠르게 만들기	122
Native에서 메모리 누수를 잡는 방법	130

Part 3: Bundling

JS 번들 사이즈 분석하는 방법	142
앱 번들사이즈 분석하는 방법	148
서드파티 라이브러리의 진짜 size 측정하는 방법	154
Barrel Exports 피하기	156
Tree Shaking 실험하기	159
필요할때 원격으로 코드 로드하기	163
R8 Android로 코드 줄이기	167
Native Assets 폴더 사용하기	170
JS Bundle 압축 비활성화 하기	175
감사의 말	
저자들에 관하여	177
이 가이드에 언급된 라이브러리 도구들	179

머리말

React Native는 먼 길을 걸어왔습니다. 초반에는 크로스 플랫폼 분야에서 유망한 실험으로 시작되었고, 세계 유수의 기업들이 도입하기까지 눈부신 진화를 해왔습니다.

React Native 프레임워크는 성숙해지면서 대규모 어플리케이션 요구사항을 충족하는 동시에 기술적 경계를 확장하고 개선해 나가는 역량을 입증했습니다.

수년 동안 우리는 React Native가 종종 획기적인 새로운 기능을 도입하는 것을 보았습니다. 그리고 오늘날에 모든 release에서 우리는 성능, 안정성, 확장성을 향상시키는 꾸준하고 저수준의 개선을 목격하고 있습니다.

이런 개선은 React Native를 더욱 광범위하게 사람들이 채택하고 있음을 보여주고 생태계가 성숙해지고 있음을 보여줍니다. 그리고 기업 환경에 적합한 프레임워크로서의 역할도 강화되고 있습니다.

이 여정에서 중요한 포인트는 뉴 아키텍처(New Architecture)의 도입이었습니다. React Native 프레임워크의 핵심적인 여러 측면을 재정의했는데, 여기에는 개발 전반에 대한 우리의 성능 최적화에 대한 생각이 포함됩니다. 새로운 기능과 모범 사례를 제공해서 개발자가 월씬 더 효율적이고 확장 가능한 애플리케이션을 구축할 수 있도록 했습니다.

그래서 우리는 최적화에 대한 가이드를 완전히 점검해야 될 때라고 결정했습니다. 강산도 10년 이면 변하듯이 우리의 접근방식도 바뀌어야 합니다. 한때 필수로 여겨졌던 일부 기술들은 더 이상 관련이 없습니다. 그리고 다른 새로운 것들이 중요성을 갖게 되었습니다. 이 새로운 버전에서는 개발자가 자신의 인사이트 도구 및 전략을 습득하여 React Native의 진화하는 기능을 최대한 활용할 수 있도록 하는 것을 목표로 합니다.

경험이 풍부한 React Native 엔지니어든 이전 시작하는 엔지니어든 이 가이드는 다음과 같습니다. 성능 최적화 Best Practice 그 이상입니다. 또한 React Native의 전반적인 이해에 도움이 되는 필수 지식과 내부 작동 방식을 제공합니다. 저희는 이러한 지식을 바탕으로 더 좋고 효율적이며 확장 가능한 어플리케이션을 개발하고 있으며 이 지식이 여러분도 같은 목표를 달성하는데 도움이 되기를 바랍니다

— Mike Grabowski, Callstack's Founder & CTO

이 책을 읽는 방법

이 책은 다양한 수준의 경험을 가진 리액트 네이티브 개발자를 대상으로 합니다. 웹 개발이나 네이티브 앱 개발 경력에 관계없이 리액트 네이티브를 처음 접하는 개발자부터 숙련 등 개발자까지, 누구나 자신의 앱에서 가치 있고 유용한 정보를 얻을 수 있을 것이라고 생각합니다.

특정 시점에 제시된 모든 최적화에 관심이 없을 수도 있다는 점을 인정합니다. 이 책에서는 React 리렌더링 최적화와 같은 특정 영역에 집중하는 경우가 많습니다. 따라서 책 전체를 순차적으로 읽어도 되지만 각 장을 쉽게 펼쳐 관심 있는 주제로 바로 이동할 수 있도록 구성했습니다.

어떤 내용이 기다리고 있나요?

모든 사람이 관련 정보를 더 쉽게 찾을 수 있도록 이 가이드를 세 부분으로 나누었습니다. 특히 관심을 가질 만한 다양한 최적화 유형 즉 React 측면, Native 측면, 그리고 전반적인 빌드 시간 최적화에 초점을 맞춘 파트들이 있습니다. 세 파트 모두 주요 주제에 대한 소개, 가이드 및 앱의 성능을 설명하는 가장 중요한 매트릭인 FPS(Frame per Second) 및 TTI(Time to Interactive)를 개선하는 데 도움이 되는 모범 사례가 포함이 되어 있습니다.

각 부분에서 기대할 수 있는 사항은 다음과 같습니다.

- 소개: 주요 주제 용어 및 주요 주제와 관련된 아이디어를 제시합니다.
- 가이드: 전문적인 도구를 사용하고 중요한 것을 측정하는 방법을 설명합니다.
- 모범 사례: 앱을 더 빨리 실행하고 초기화하는 방법을 보여드립니다.

이 책에서 사용되는 컨벤션.

이 책에 제시된 아이디어를 더 잘 설명하기 위해서 많은 짧은 코드 스니펫을 포함했습니다. 우리가 사용하는 도구의 스크린샷, 디어그램들도 포함되어있습니다. 공식 문서 또는 라이브러리와 같은 외부 리소스뿐만 아니라 이 문서의 다른 장에도 링크가 걸려있습니다. 선형적 독해를 위한 책임으로 읽을 때 건너뛰지 않는 것이 좋습니다

Callstack 팀 소개

Callstack은 React Native를 발전시키고 개발자가 다음을 수행할 수 있도록 최선을 다하고 있습니다.

- 고성능 어플리케이션을 구축합니다.
- 핵심 기여자이자 Meta의 파트너로서 저희는 커뮤니티와 긴밀하게 협력하여 제안서를 구체화하고 주요 모듈을 유지 관리하고 프레임워크의 발전을 주도합니다.
- 우리 팀은 React Native의 월간 릴리즈에 적극적으로 기여하면 개발자가 최신 개선사항에 액세스할 수 있도록 합니다.
- 전문 지식 도구 및 모범 사례를 공유함으로써 팀이 성능 문제를 해결하고 앱을 최적화하고 리액트 네이티브로 가능한 것의 경계를 넓힐 수 있도록 지원합니다.

연락 주세요

React Native는 커뮤니티로 인해 번성하고 있으며 여러분도 그 일부가 될 수 있습니다.

자신의 앱을 최적화하거나, 오픈 소스 프로젝트에 기여하거나, 최신 발전 사항에 대한 정보를 유지함으로써 여러분의 참여는 생태계를 발전시키는 데 도움이 됩니다.

참여하는 방법은 다음과 같습니다.

- **최신 상태 유지:** 뉴스레터를 구독하고 최신 인사이트를 따르세요.
- **대화에 참여:** 디스코드 서버에 가입하여 제안에 대해 논의하고, 아이디어를 공유해주시면, React Native의 미래를 형성하는 데 도움이 됩니다.
- **참여:** 오픈 소스 프로젝트를 탐색하고 협업해주세요.

연락을 원하시면 언제든지 저희에게 연락해주세요. 함께 더 빠르고 더 나은 React Native를 구축해봅시다!

왜 성능이 중요한가?

모바일 앱을 개발할 때 성능은 단순히 기술적인 측면의 문제가 아니라, 사용자 경험(UX)과 직결되는 최우선 과제입니다. 빠르고 매끄럽게 반응하는 앱은 사용자에게 만족감을 주지만, 그렇지 못한 앱은 사용자가 외면하게 만들 수 있습니다. 그렇다면 여기서 '빠르다'는 것은 정확히 무엇을 의미할까요? 단순히 수치로 측정되는 속도를 말하는 걸까요, 아니면 사용자가 주관적으로 느끼는 빠르다는 '인식'을 의미하는 걸까요?

사용자 관점에서의 성능

'체감 성능'이란 사용자에게 앱이 얼마나 빠르게 느껴지는지에 대한 문제입니다. 이는 단순히 측정된 수치나 벤치마크 결과가 아니라, 사용자에게 빠르다는 '인식' 혹은 '느낌'을 어떻게 만들어 주느냐에 달려 있습니다.

1940년대 뉴욕의 한 오피스 빌딩에서는 엘리베이터가 너무 느리다는 불만이 제기되었습니다. 이 문제를 해결하기 위해 건물 관리자는 엘리베이터 속도를 개선하는 대신, 엘리베이터 옆에 거울을 설치했습니다. 이 간단한 조치는 사람들이 거울을 보며 기다리는 동안 주의를 다른 곳으로 돌리게 만들었고, 그 결과 사용자가 느끼는 대기 시간이 줄어 문제가 효과적으로 해결된 것입니다. 이 이야기가 도시 전설에 가까울 수도 있지만, 그 바탕에 깔린 개념, 즉 '인지된 시간'과 '주의 분산'이 특정 경험의 속도감에 영향을 미친다는 점은 타당한 심리학적 원리에 근거합니다.

모바일 앱이 로딩되는 몇 초의 시간 동안 스플래시 화면이나 스켈레톤 UI(화면 윤곽만 먼저 보여주는 방식), 혹은 간단한 미니 게임 같은 것을 보여준다면, 사용자는 앱이 멈추지 않고 응답하고 있으며 곧 사용할 준비가 될 것이라고 느끼게 됩니다. 이것이 바로 사용자 만족도 측면에서 실제 성능 수치보다 체감 성능이 종종 더 중요하게 여겨지는 이유입니다.

하지만 체감 성능에만 오롯이 집중하는 것은 자칫 본질을 놓치게 만들 수 있습니다. 애니메이션 효과, 플레이스 홀더(내용이 들어갈 자리를 미리 보여주는 것), 또는 콘텐츠 선(先) 로딩과 같은 기법들이 사용자의 체감 속도를 개선할 수는 있겠지만, 앱의 근본적인 성능 저하 문제를 해결하는 것은 아닙니다. 바로 이 지점에서 TTI(Time to Interactive, 상호작용까지 걸리는 시간)나 FPS(Frames Per Second, 초당 프레임 수)와 같은 실제 측정 가능한 지표들의 중요성이 부각됩니다.

핵심 지표: TTI와 FPS

측정하고 관리할 수 있는 수많은 성능 지표 중에서, 사용자가 앱의 속도를 체감하는 방식에 가장 큰 영향을 미치는 두 가지 핵심 지표가 있습니다. 바로 TTI(Time To Interactive)가 설명하는 '사용자가 얼마나 빨리 앱과 상호작용을 시작할 수 있는지', 그리고 FPS(Frames Per Second)가 설명하는 '앱과 상호작용하는 동안 얼마나 부드럽게 느껴지는지'입니다.

TTI (Time to Interactive): 상호작용 가능 시간

TTI 지표는 앱의 초기 구동(부팅) 성능을 측정합니다. 즉, 앱이 실행된 후 사용자가 자연이나 버벅임(jank) 없이 실제로 앱과 상호작용을 시작할 수 있는 상태가 되기까지 얼마나 시간이 걸리는지를 나타냅니다.

하지만 TTI는 단순히 앱의 초기 로딩 시간만을 의미하지는 않습니다. 기업들은 종종 이 지표를 변형하여 홈 화면이 로딩되기까지 걸리는 시간(Time to Home)이나 특정 핵심 기능 화면으로 이동하는 데 걸리는 시간(Time to Specific Screen) 등 다양한 형태로 추적하기도 합니다. 이러한 세부 지표들은 앱 내 이동 경로(navigation flow)가 복잡하거나 초기에 많은 데이터를 불러와야 하는 앱의 경우 특히 중요하게 관리됩니다.

만약 앱이 사용 가능 상태가 되기까지 너무 오랜 시간이 걸린다면(즉, TTI가 길다면), 사용자들은 앱의 기능을 제대로 경험해보기도 전에 앱 사용을 포기할 수 있습니다. 또한 앱 내에서 어떤 작업을 시도할 때마다 느리다고 느끼게 되면, 사용자는 불쾌감을 느끼고 결국에는 앱 자체, 심지어는 그 브랜드에 대해서까지 부정적인 인식을 갖게 될 수 있습니다.

FPS (Frames Per Second): 초당 프레임 수

앱이 일단 실행되고 나면, FPS는 런타임(실행 중) 성능의 핵심 지표가 됩니다. FPS는 스크롤, 스와이프, 버튼 탭과 같은 사용자의 상호작용에 앱이 얼마나 부드럽게 반응하는지를 측정합니다. 높은 FPS(이상적으로 초당 60프레임 이상)는 애니메이션과 화면 전환이 매끄럽고 자연스럽게 느껴지도록 보장합니다. 반면 FPS가 떨어지면 사용자 경험에 지연(랙)이 발생하고 애니메이션이 끊기기(버벅거리기) 시작하며, 이는 앱이 다듬어지지 않았거나 반응이 느리다는 인상을 주게 됩니다. 이는 특히 시각적으로 풍부한 콘텐츠를 다루거나 복잡한 상호작용이 많은 앱에서 매우 중요합니다.

그렇다면 '빠르다'는 것은 대체 무엇을 의미할까요?

우리가 가설 검증을 위해 데이터와 과학적 접근 방식을 중요하게 생각하지만, 모바일 사용자에게 '빠르다'는 것이 구체적으로 무엇을 의미하는지에 대한 독립적이고 신뢰성 높은 연구는 아직 부족한 것이 현실입니다. 연구가 있다 해도, 대개는 거대 기술 기업들이 발표하는 전환율 개선 사례나 설문조사에 기반한 유료 보고서 형태에 그치는 경우가 많습니다.

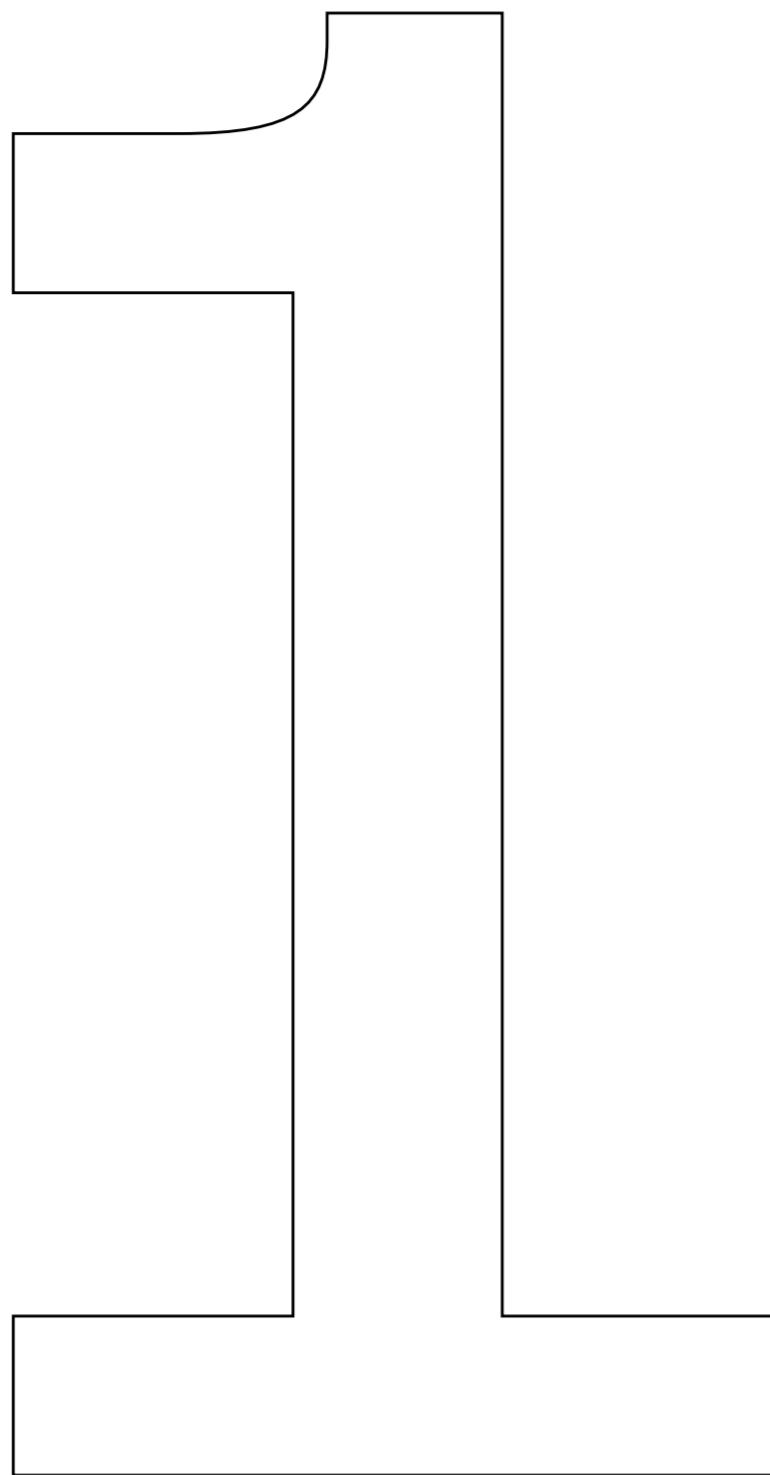
이보다 더욱 중요한 점은, 속도에 대한 사용자들의 '체감' 기준은 계속해서 변한다는 사실입니다. 고성능 기기가 보편화되고 스마트폰 하나로 수많은 앱을 사용하는 경험이 쌓이면서 사용자들의 눈높이는 끊임없이 높아지고 있습니다. 또한, 이러한 기대치는 사용자층(demographic)에 따라서도 크게 달라진다는 점을 기억해야 합니다. 예를 들어, 젊은 세대는 일반적으로 (과거의 느린 환경에 더 익숙한) 부모나 조부모 세대보다 앱 로딩과 반응 속도가 훨씬 빠르기를 기대합니다. 앱 성능 개선에 대한 기대치와 동기 부여 요인 역시, 일반 소비자와 기업 내부 사용자 간에 차이가 있을 수 있습니다.

자신의 앱이 '충분히 빠른가'를 판단하는 가장 신뢰할 만한 방법은 결국 '자신의 사용자'와 '경쟁 환경'을 제대로 파악하는 것입니다. 수집된 사용자 행동 분석 데이터는 사용자들이 주로 어떤 지점에서 이탈하는지에 대한 귀중한 정보를 제공합니다. 한편, 경쟁 앱들과 객관적인 속도를 비교 분석해보면 성능 문제가 사용자 이탈의 잠재적인 원인인지 아닌지에 대한 단서를 얻을 수 있습니다.

이 책의 1부와 2부에서는 자바스크립트, 리액트, 그리고 네이티브 코드 최적화를 통해 앱의 런타임 성능을 개선하는 구체적인 방법들에 초점을 맞출 것입니다. FPS에 영향을 미치는 요인들을 더 깊이 이해하고 문제 지점을 찾아 해결하는 데 필요한 모든 도구와 기법들을 상세히 안내해 드리겠습니다.

3부에서는 앱의 번들링(bundling) 과정에 초점을 맞춥니다. 이 번들링 과정은 자바스크립트와 네이티브 코드 양쪽 모두에서 진행되며, 앱의 초기 구동(부팅) 성능과 매우 밀접한 관련이 있습니다. 앞선 1부, 2부와 마찬가지로, 3부에서도 무엇이 앱의 로딩 속도를 느리게 만드는 원인인지 명확하게 파악하고, 이를 개선하여 속도를 높이는 데 필요한 모든 내용을 다룰 것입니다.

PART



JAVASCRIPT

**자바스크립트와 React 최적화를 통한
React Native FPS 개선 방법과 가이드**

서론

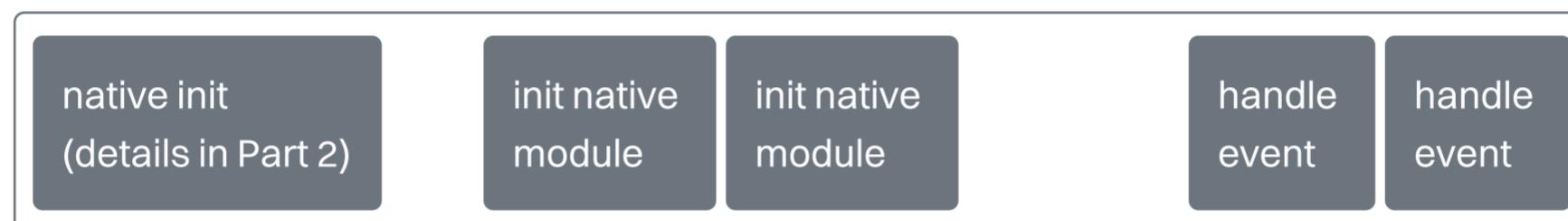
이 가이드의 1부에서는 **React Native** 생태계의 자바스크립트(**JavaScript**) 부분에 초점을 맞춥니다. 아마 이미 알고 계시겠지만, **React Native**는 네이티브 플랫폼의 기본 기능(**primitives**)을 활용하며 다양한 기술들을 결합하여 사용합니다. 안드로이드(Android) 개발에는 코틀린(Kotlin)이나 자바(Java), iOS 개발에는 스위프트(Swift)나 오브젝티브-씨(Objective-C), **React Native**의 핵심 런타임(core runtime)에는 C++, 그리고 자바스크립트(**JavaScript**) 실행을 위해서는 헤르메스(Hermes) 엔진을 사용하는 식입니다. 알아야 할 기술이 많으므로, 설명을 단순화하기 위해 우선 안드로이드(Android) 플랫폼을 중심으로 자세히 살펴보겠습니다.

사용자가 일반적인 네이티브 안드로이드 앱을 실행하면, 앱은 보통 다음과 같은 초기화 단계를 거칩니다. 먼저 메인 스레드(main thread)에서 앱이 시작되고, 코틀린(Kotlin), 자바(Java) 또는 C++ 등으로 작성된 네이티브 코드를 메모리에 불러온(load) 후, 이 코드를 실행하여 사용자에게 UI를 표시합니다.

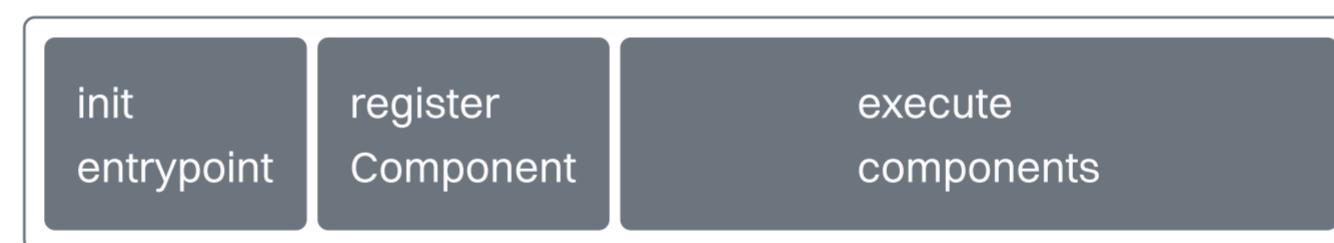
React Native로 만든 안드로이드 앱의 경우에도 이 초기화 과정은 크게 다르지 않습니다. 결국 **React Native** 앱도 본질적으로는 네이티브 앱이기 때문입니다. 다만 한 가지 특징적인 차이점은, 실행되는 네이티브 코드 중 일부가 **React Native** 프레임워크 자체로부터 온다는 것입니다.

자바스크립트 초기화 과정

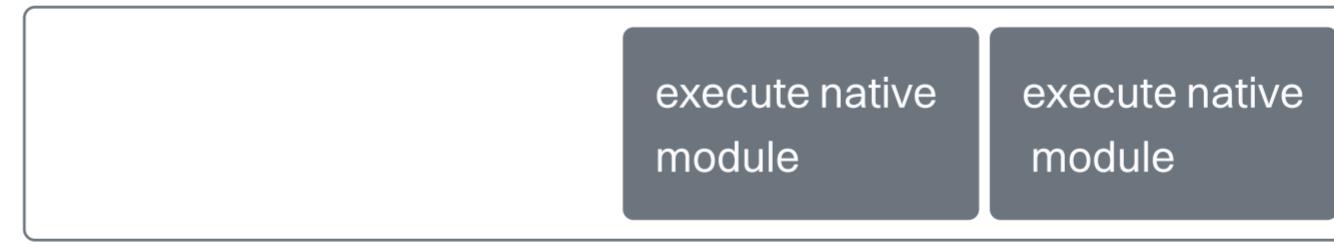
UI thread



JS thread



Native modules thread



JS 실행 환경을 보여주는 **React Native New Architecture**의 간소화된 스레딩 모델

React Native 초기화 코드는 몇 가지 핵심 메커니즘을 처리합니다:

- 메인 스레드에서 **React Native** 내부 요소(크로스 플랫폼 C++ React 렌더러, JSI(JavaScript Interface), Hermes JS 엔진, 레이아웃 엔진(Yoga) 등) 초기화.
- JS를 실행하고 JSI를 통해 메인 스레드와 양방향 통신하는 자바스크립트 스레드 초기화.
- lazy loading 되는 터보 모듈(Turbo Modules)을 실행하는 네이티브 모듈 스레드 초기화

렌더링 로직(Kotlin, C++)과 소위 '비즈니스 로직'이 설계상 각각 별도의 스레드(메인 스레드 대 JS 스레드)에서 실행됨을 주목하세요. 이 파트는 자바스크립트와 React에 관한 것이므로, React가 전용 JS 스레드에서 실행될 수 있게 하는 자바스크립트 측면과 이 모델이 React Native 앱 성능에 미치는 영향에 초점을 맞춥니다.

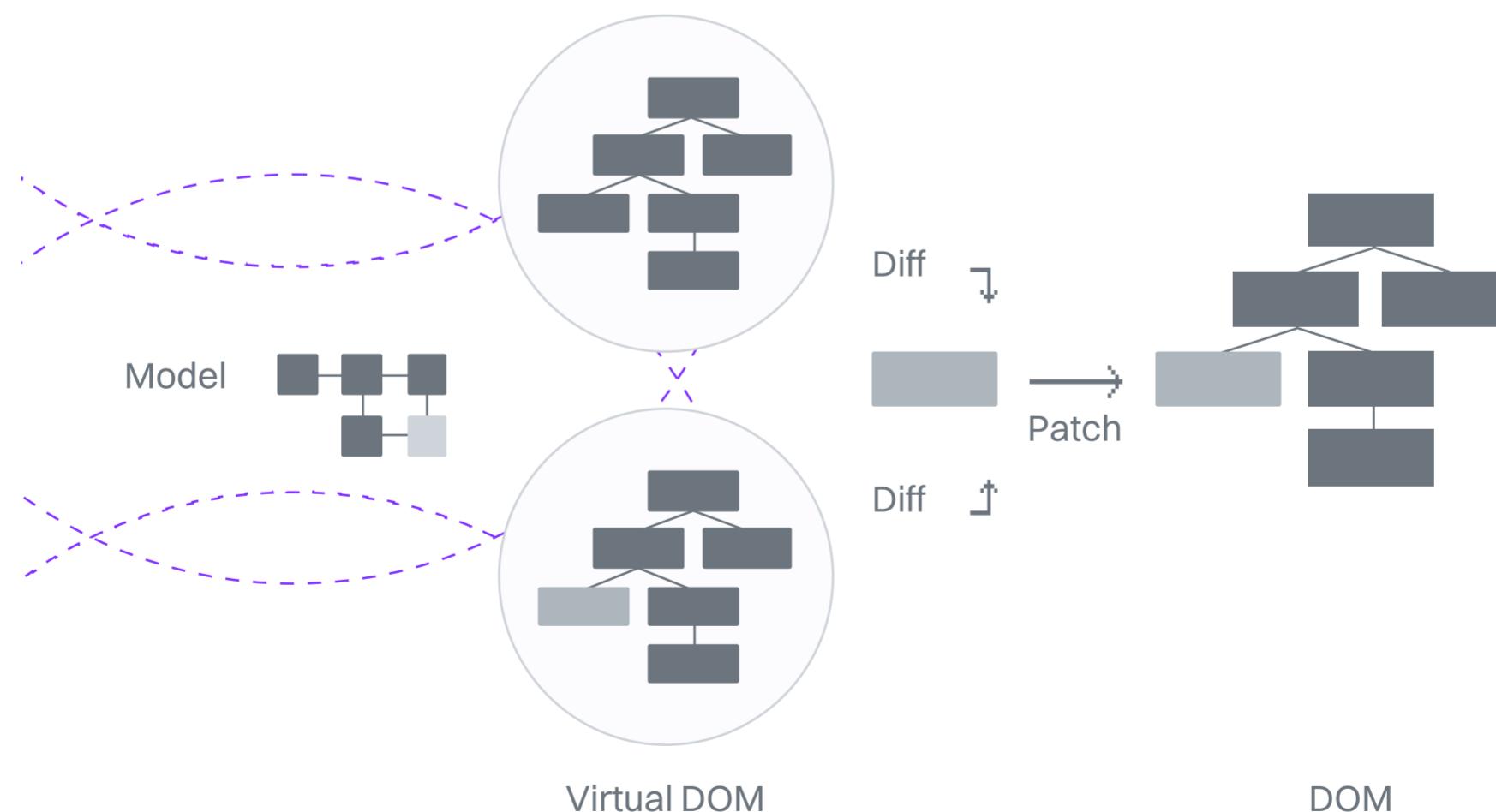
Callstack 내부의 React Native 개발자 100명을 대상으로 한 설문조사에 따르면, 모바일, TV, 데스크톱 React Native 앱에서 겪는 성능 문제의 80%가 자바스크립트 측에서 발생한다고 합니다. 이 표본은 크기가 작아 실제를 대표하지 못할 수도 있습니다. 그러나 React Native 커뮤니티가 겪는 어려움을 볼 때, 대부분의 성능 문제는 자바스크립트와 관련이 있다고 가정하는 것이 타당합니다. 따라서 '2부: 네이티브'에서 설명할 고급 네이티브 최적화로 넘어가기 전에 이러한 문제에 먼저 집중하는 것이 현명합니다.

React 리렌더링 모델

React는 환경(플랫폼)에 관계없이(웹, iOS, Android 등) 상태(state)에 기반하여 애플리케이션 UI 렌더링 및 업데이트를 처리합니다. react 라이브러리 자체는 작으며, 주로 공개 API 정의, 일부 크로스 플랫폼 기능, 그리고 재조정(reconciliation) 알고리즘으로 구성됩니다. 이 알고리즘은 컴포넌트 상태 변화를 반영하도록 효율적으로 UI를 업데이트하며, 실질적으로 '무엇을 할지(what to do)'를 기술합니다.

하지만 이 모델의 강점은 '무엇을 할지(what)'와 '어떻게 할지(how)'를 분리하고, 후자('어떻게')는 각기 다른 환경에 업데이트를 적용하는 방법을 관리하는 특화된 렌더러(renderer)에 위임한다는 점입니다. 예를 들어 웹용으로는 react-dom, iOS/Android용으로는 react-native, 윈도우(Windows)용으로는 react-native-windows 등이 있습니다.

이 모델 덕분에, 지원되는 모든 플랫폼용 앱 컴포넌트를 한 번에 보편적으로 정의하고, 이런 작은 빌딩 블록(building block)들로 최종 인터페이스를 구성할 수 있습니다. 이런 접근 방식에서는 개발자가 애플리케이션 렌더링 생명주기(lifecycle)를 직접 제어하지 않고, React와 해당 렌더러가 그 역할을 대신 수행합니다.



React 재조정 알고리즘 시각적 설명

즉, 화면을 언제 다시 그릴지(repaint) 결정하는 것은 전적으로 React의 역할이며, '어떻게' 그릴지는 렌더러(renderer)의 역할입니다. React는 컴포넌트에 발생한 변경 사항을 감지하고 비교한 후, 설계에 따라 필요한 최소한의 실제 업데이트만 수행합니다.



React 컴포넌트는 다음과 같은 경우에 리렌더링됩니다:

- 부모 컴포넌트가 리렌더링될 때
- 상태(state, 흑(hook) 포함)가 변경될 때
- Props가 변경될 때
- Context가 변경될 때
- 강제 업데이트(force update) 시 (예외적 방법)

앱의 특정 사용 사례(use case)에 따라 적용 여부가 달라질 수 있는 여러 기법들을 살펴보기 전에, 먼저 어떤 앱을 개발하든 항상 유용하게 활용될 프로파일링(profiling)과 측정(measuring)부터 시작하겠습니다.

GUIDE

JS 및 React 코드 프로파일링 방법

성능 최적화 시에는 정확히 어느 코드 경로(code path)가 속도 저하의 원인인지 알아야 합니다. 경험이 있고 코드베이스와 일반적인 성능 문제 지점(pitfall)을 안다면, 개발자는 직관적으로 문제를 파악하고 앱 속도 개선을 기대하기도 합니다. 하지만 현실적으로 앱은 매우 복잡하여 누구도 전체 시스템을 속속들이 알기는 어렵습니다. 이런 상황에서 하는 일반적인 '감(感)에 의존한 최적화(blind optimization)'는 전체 성능에 거의 영향을 미치지 못합니다. 따라서 정확한 개선 작업을 위해서는 반드시 데이터에 기반한 결정이 필요합니다.

이러한 데이터는 특화된 도구를 사용한 성능 측정에서 나옵니다. 프로그램 실행 '프로파일(profile)'을 생성하기 위해 프로그램을 분석하는 이 과정을 흔히 '프로파일링(profiling)'이라고 부릅니다. 이 프로파일에는 보통 프로그램의 어느 부분이 CPU 시간이나 메모리 같은 자원을 가장 많이 사용하는지에 대한 정보가 포함되어, 성능 병목 지점(bottleneck)을 찾는 데 도움을 줍니다.

React Native DevTools로 프로파일링하기

React Native 앱에서는 React 자체를 프로파일링하여 불필요한 리렌더링을 검사하는 것이 유용합니다. 모바일 앱 내 React를 프로파일링하는 가장 좋은 도구는 **React Native DevTools**이며, 이 가이드에서는 바로 이 도구에 집중할 것입니다.

React Profiler는 **React Native DevTools**에 기본 플러그인으로 통합되어 있으며, 프로파일링 결과로 React 렌더링 파이프라인의 플레임 그래프(flame graph)를 생성할 수 있습니다. 우리는 이 데이터를 활용하여 앱의 리렌더링 문제를 분석할 수 있습니다.

다음은 우리가 프로파일링해 볼 코드입니다:

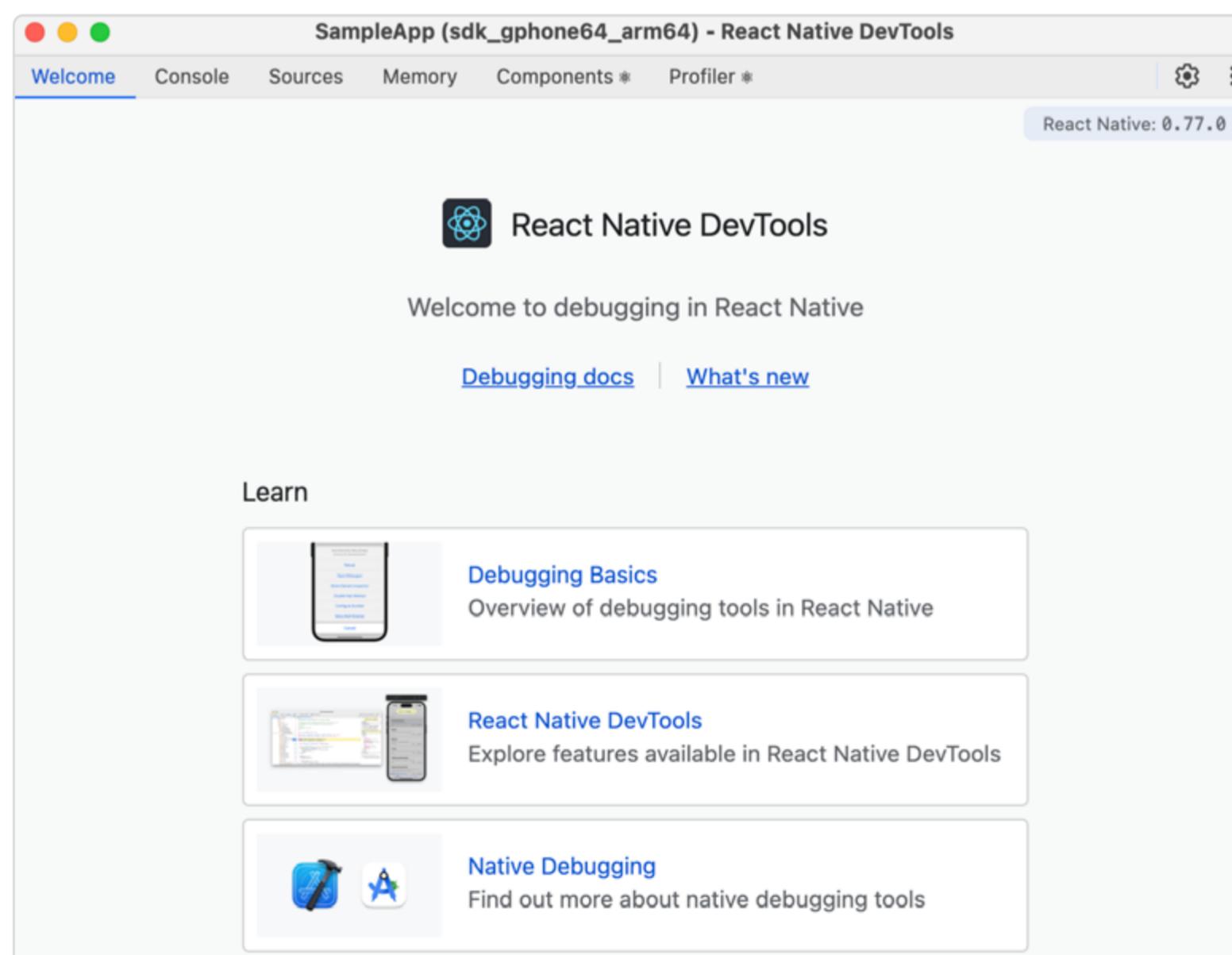
```
export const App = () => {
  const [count, setCount] = React.useState(0);
  const [second, setSecond] = React.useState(0);
  return (
    <View style={styles.container}>
```

```
<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>
  );
};
```

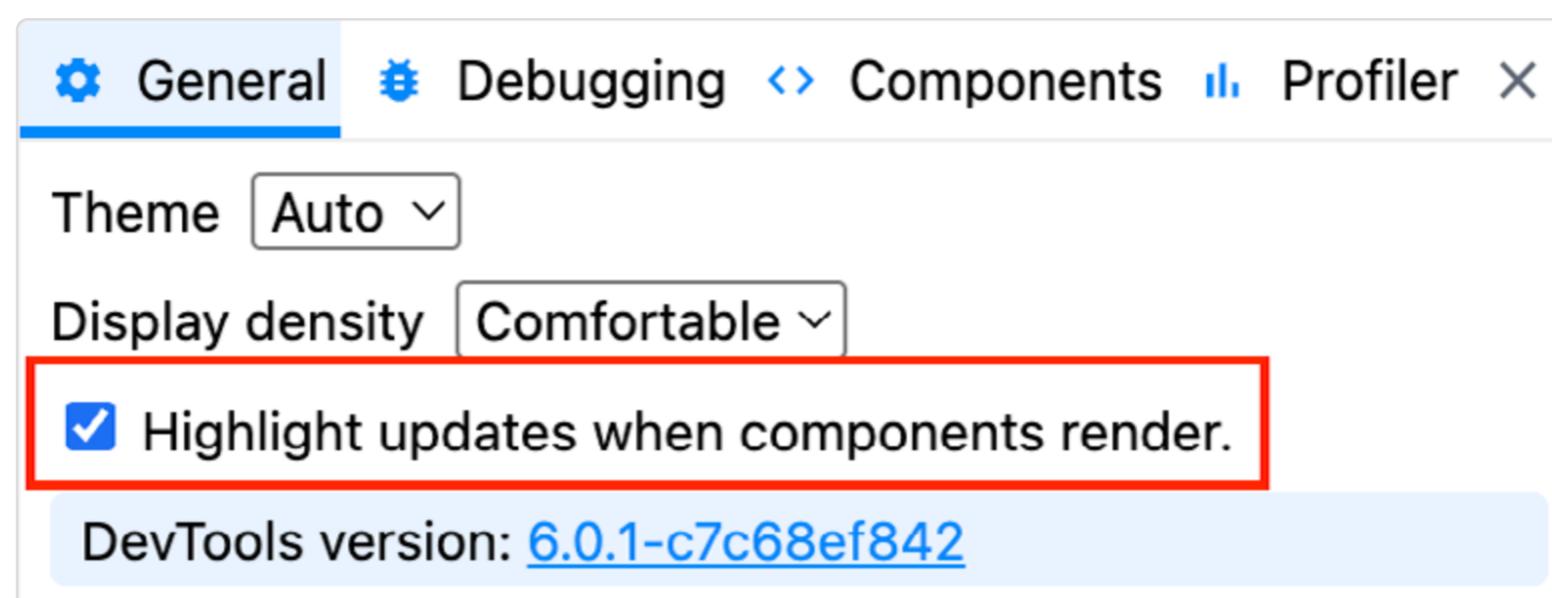
<App /> 컴포넌트가 렌더링되면, 환영 메시지와 함께 각각 별도의 상태 핸들러(state handler)로 제어되는 카운터와 버튼 두 쌍이 표시됩니다. 이제 React Native DevTools로 이 React 코드를 프로파일링할 준비가 되었습니다.

React Native DevTools는 메트로(Metro) 개발 서버에서 'j' 키를 누르거나, 앱 내 React Native 개발 메뉴(기기 흔들기 제스처 또는 안드로이드 Cmd+M / iOS Cmd+D로 접근)에서 'Open DevTools'를 선택하여 열 수 있습니다. 그러면 디버깅 기초, DevTools 기능, 네이티브 디버깅에 대한 학습 자료로 연결되는 시작 화면이 표시됩니다. 이 자료 중 일부는 본 가이드 내용과 중복되지만, 미리 숙지해두시기를 적극 권장합니다.



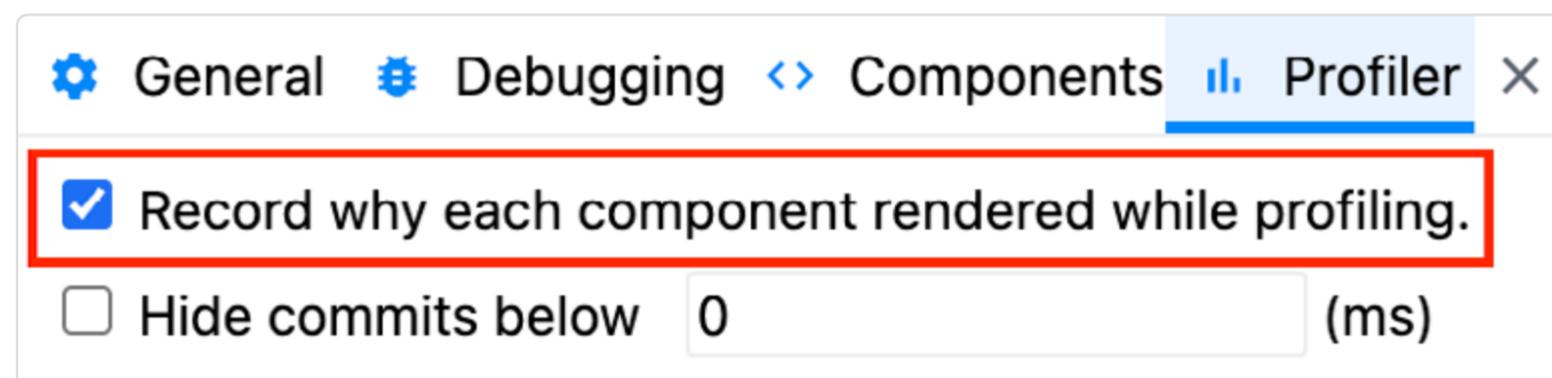
React Native DevTools welcome screen

이제 React Native DevTools를 엽니다. 상단 Profiler 탭으로 이동하세요. '설정(gear)' 버튼을 눌러 'Highlight updates when components render' 설정을 겁니다.



React Native DevTools의 Profiler 탭 내 'View Settings' 아이콘을 통해 접근하는 일반 프로파일러 설정

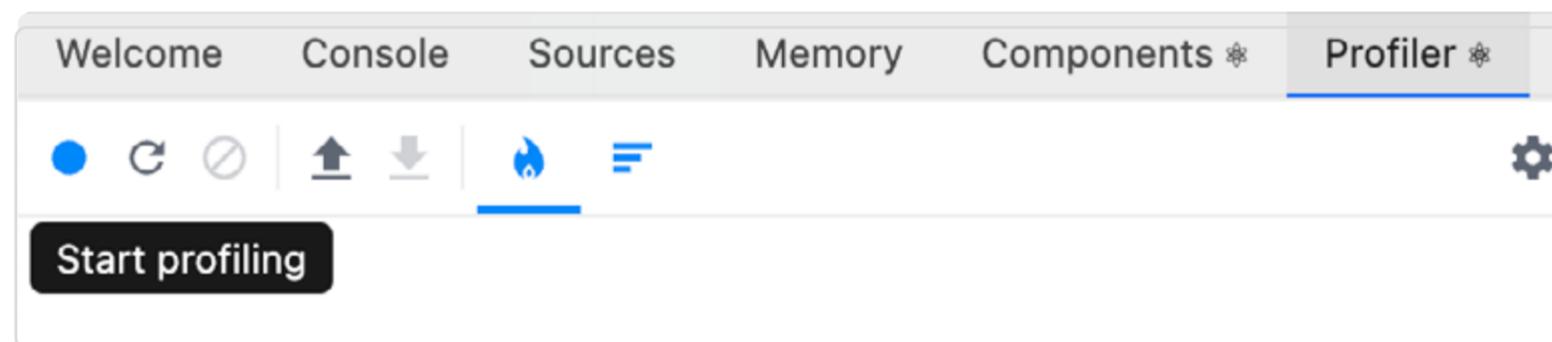
그리고 '프로파일링 중 각 컴포넌트가 렌더링된 이유 기록(Record why each component rendered while profiling)' 설정도 겁니다.



React Native DevTools Profiler 탭 'View Settings' 아이콘에서 접근하는 프로파일러 설정

마지막으로, 화면 좌측 상단의 'Start profiling' 또는 'Reload and start profiling' 버튼을 눌러 프로파일링을 시작합니다.

이번 사용 사례에서는 사용자 입력 반응을 살펴보는 것이므로 어느 버튼을 사용해도 괜찮습니다. 하지만 앱 시작 과정을 프로파일링할 때는 'Reload and start profiling' 버튼이 매우 유용합니다.

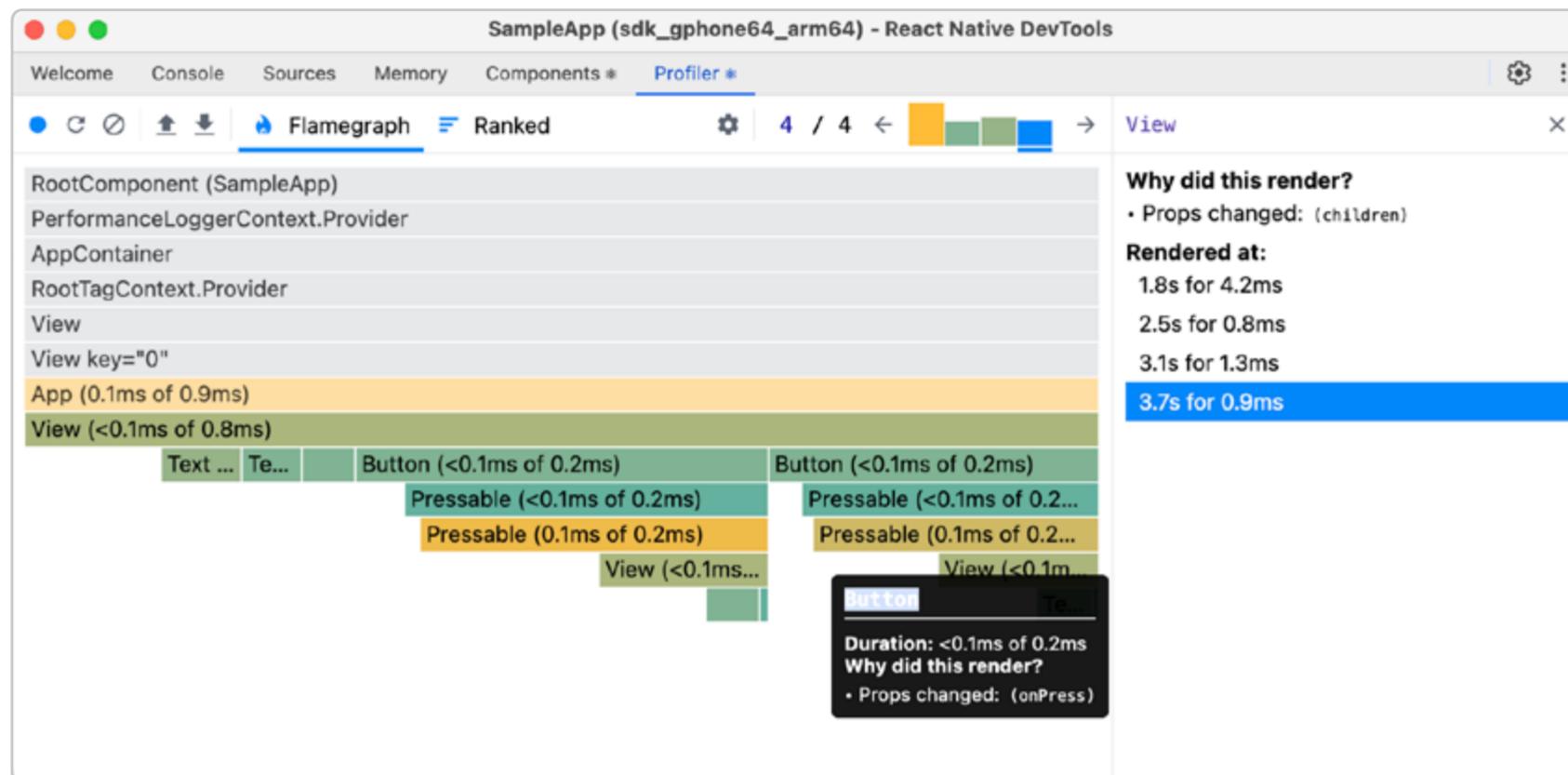


"Start profiling" 버튼

샘플 앱에서 두 버튼을 몇 번 클릭하면서, DevTools에서 리렌더링되는 컴포넌트가 강조 표시되는 실시간 피드백을 관찰하세요.

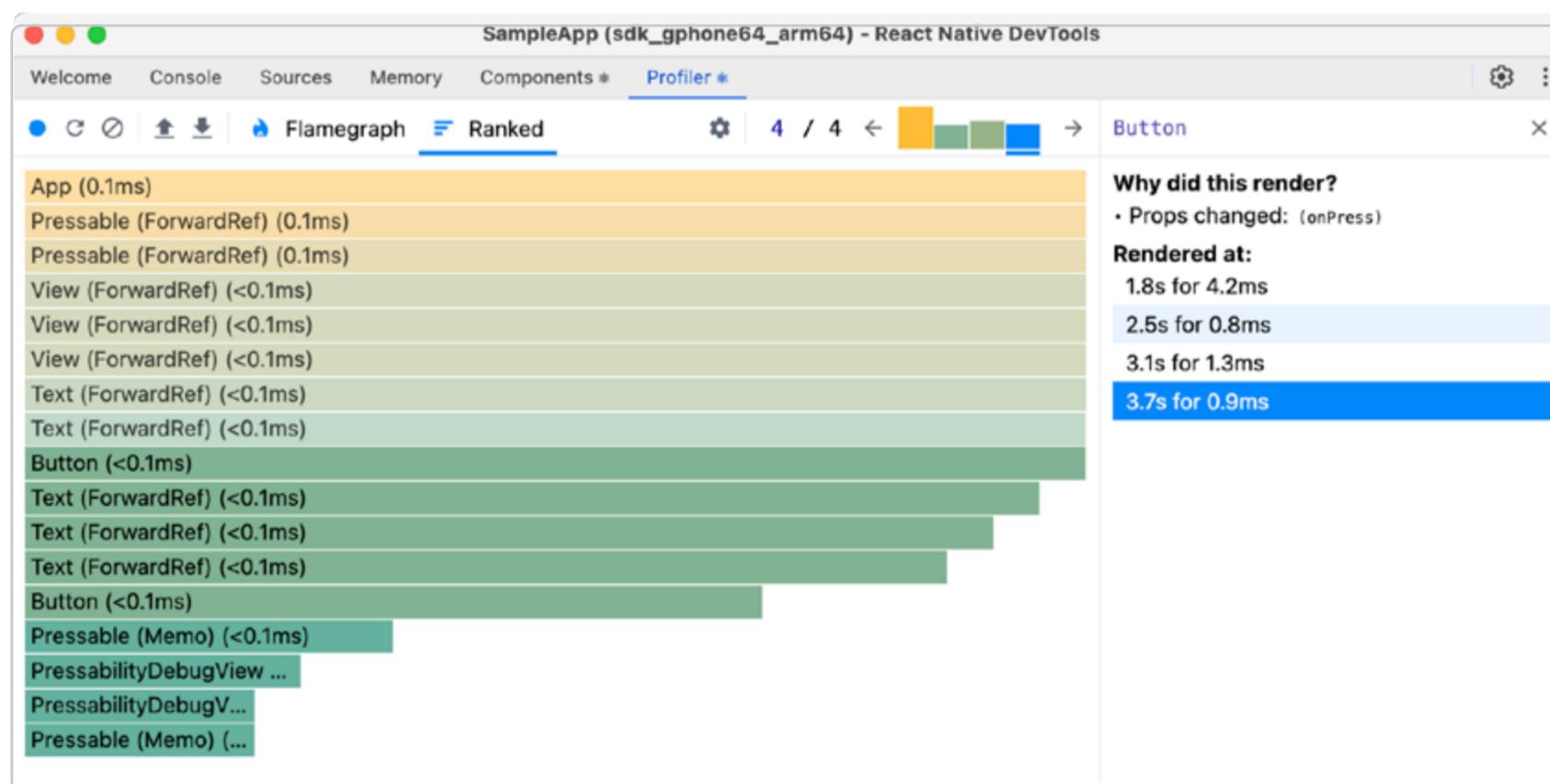
(이어지는) 스크린샷은 'Press one'을 두 번, 이어서 'Press two'를 두 번 눌러 React 렌더러가 네 번의 '커밋(commit)'을 생성한 후의 프로파일을 보여줍니다.

💡 React 렌더(render) 및 커밋(commit) 과정에 대한 자세한 내용은 [공식 문서를](#) 참조하세요.



플레임 그래프(Flame graph)에서 녹색 View 컴포넌트를 눌러 하위 자식 컴포넌트들로 '줌 인(zoom in)' 할 수 있습니다. 이는 Profiler에 구현된 핵심적인 UX 방식으로, 여러분도 곧 익숙하게 사용하시게 될 것입니다. 이 예시에서는 모든 커밋(commit)마다 두 개의 Button 컴포넌트가 모두 렌더링되는 것을 볼 수 있는데, Props가 변경되지 않았으므로 불필요해 보입니다. 정말 그럴까요?

상황을 다른 관점에서 파악하려면, 렌더링이 가장 느린 컴포넌트(노란색)부터 가장 빠른 컴포넌트(녹색) 순으로 보여주는 'Ranked' 뷰를 이용할 수 있습니다. 이는 Chrome DevTools나 네이티브 프로파일러 같은 다른 프로파일링 도구의 'Bottom-up' 뷰와 상당히 유사합니다.

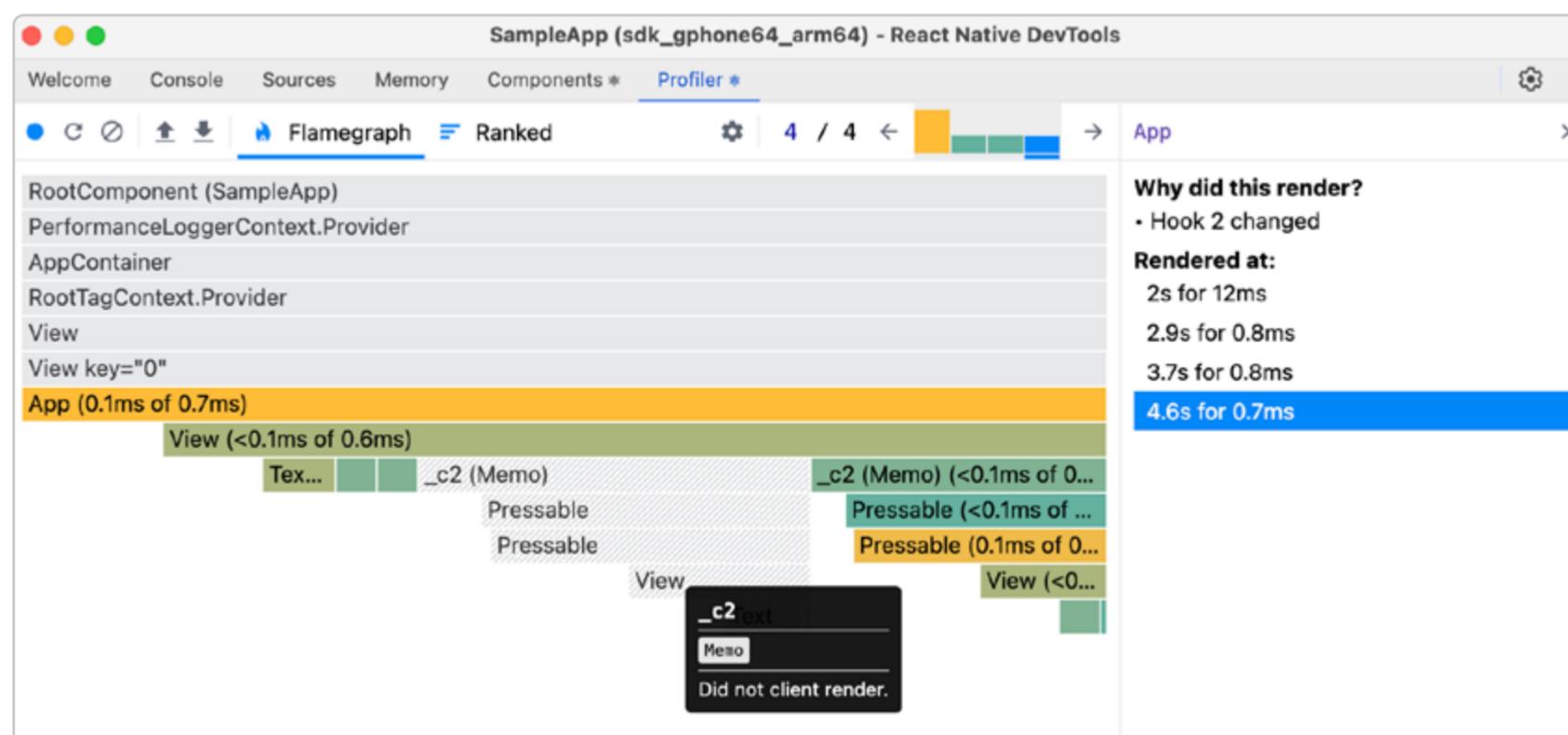


두 뷰(Flame 및 Ranked) 모두에서 Button 컴포넌트가 두 번씩 렌더링되는 것을 확인할 수 있는데, 이는 상태(state)가 업데이트될 때마다 두 개의 버튼이 모두 리렌더링됨을 보여줍니다.

코드를 보면 이는 예상된 동작입니다. 단, React Compiler를 사용하는 경우는 예외인데, 이 컴파일러는 자동으로 코드를 메모아이즈(memoize)하여 리렌더링 횟수를 줄여주기 때문입니다. 그러나 여기서는 수동으로 메모아이즈를 적용해 보겠습니다.

```
// 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}) => {
  // ...
});
```

Button 컴포넌트를 `React.memo`로, `onPress` 핸들러들을 `React.useCallback`으로 감싼 후에는 프로파일러(Profiler)에서 다른 결과를 볼 수 있습니다. React 커밋(commit) 횟수는 4번으로 동일하게 표시되지만, 이제는 실제로 눌린 버튼만 리렌더링되고, 다른 버튼은 (자동 생성된 `_c2` 이름과 함께) 메모아이즈(memoize)됩니다. 이는 해당 컴포넌트와 그 자식 요소들이 녹색으로 표시되는 것을 통해 확인할 수 있습니다.



이제 여러분은 웹 앱이든 모바일 앱이든 모든 React 애플리케이션의 프로파일링을 시작할 준비를 마쳤습니다. 지금 잠시 읽기를 멈추고, 평소 개발하는 앱에 직접 적용해보시기를 강력히 권장합니다.

실제 React 앱은 훨씬 복잡하기 때문에 플레임 그래프(flame graph) 결과가 훨씬 더 어지럽게(noisier) 보일 것입니다.

 React가 가장 많은 시간을 소비하는 부분인 노란색 표시 컴포넌트에 집중하세요.
그리고 'Why did this render?' 정보를 활용하세요.

자바스크립트(JavaScript) 코드 프로파일링

React Native DevTools는 React뿐만 아니라 자바스크립트(JavaScript) 코드 프로파일링을 위한 훌륭한 유ти리티도 제공합니다. 결국 이는 Chrome DevTools에 기반하며 그 백엔드/backend)를 재사용하기 때문입니다.

이 섹션에서는 **CPU 프로파일링**, **콜 스택(call stack)** 내용 검토, 그리고 오래 실행되는 연산(**long-running operation**) 탐지 방법에 초점을 맞출 것입니다. 메모리 관련 문제를 확인하고 싶다면, '**JS 메모리 누수 추적 방법(How to Hunt JS Memory Leaks)**' 챕터를 참고하세요.

React Native DevTools를 실행한 후, '**JavaScript Profiler**' 탭으로 이동하여 자바스크립트 **CPU 프로파일(JavaScript CPU Profile)** 기록을 시작할 수 있습니다.



'**JavaScript Profiler**' 탭이 보이지 않는다면, 우측 상단의 설정(gear) 아이콘을 클릭하여 설정으로 이동한 다음, **experiments** 섹션에서 **JavaScript Profiler**를 활성화하세요.

좌측 상단에서 원하는 뷰(view)를 선택할 수 있습니다. '**Chart**', '**Heavy (Bottom-Up)**', 또 는 '**Tree**' 중에서 선택 가능하며, 분석 목표에 따라 각 뷰를 활용해보세요.

다음의 오래 실행되는 함수(**long-running function**)로 실험해 보겠습니다:

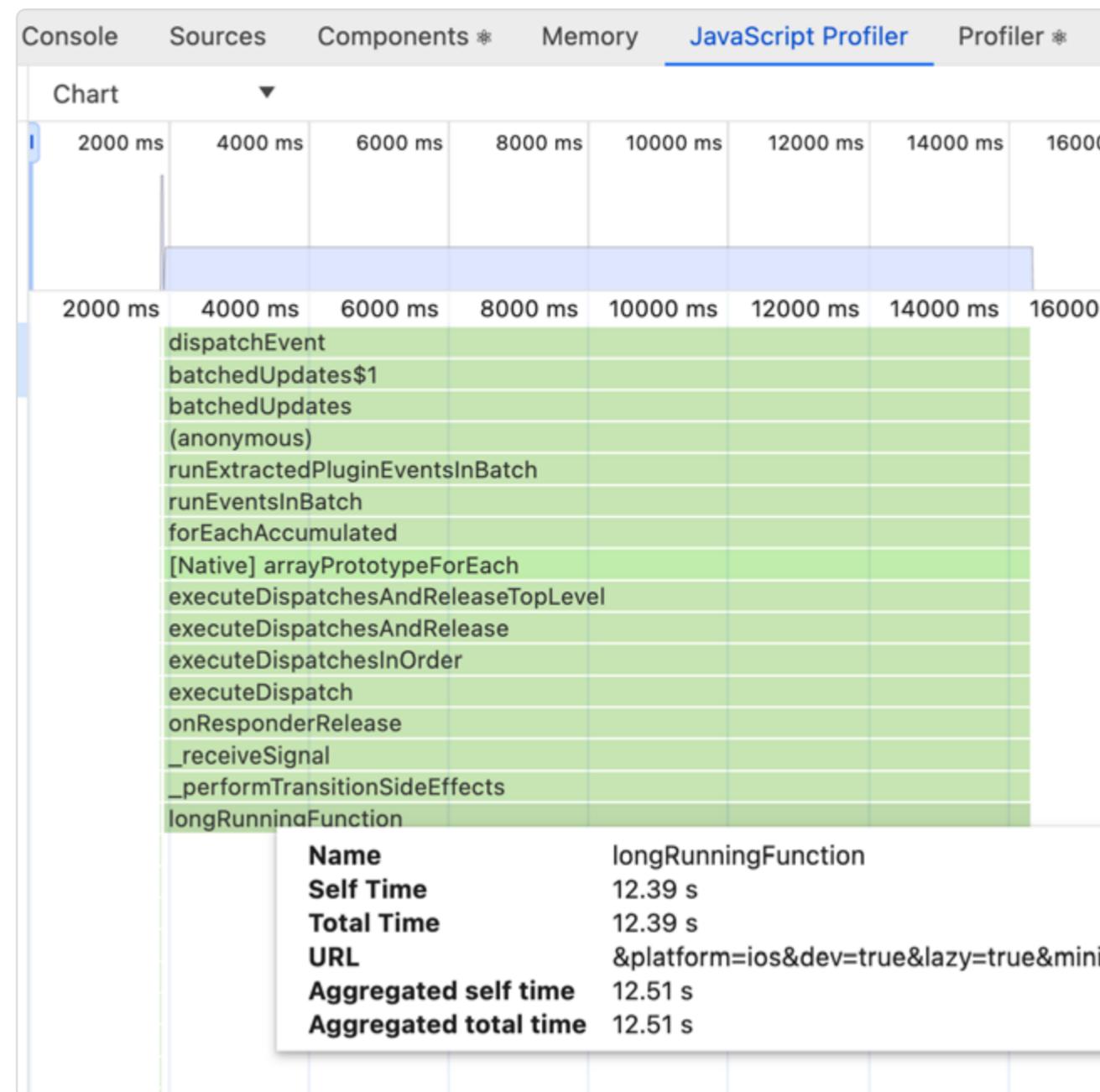
```
const longRunningFunction = () => {
  let i = 0;
  while (i < 1000000000) {
    i++;
  }
};
```

프로파일 기록을 시작하고 (함수 실행이) 끝나면 중지합니다.

프로파일이 생성되면, **longRunningFunction** 함수 실행에 12초 이상 걸렸으며, 이 함수 호출로 이어진 전체 콜 스택(call stack) 정보도 함께 볼 수 있습니다. 이는 실질적으로 해당 12초 동안 앱의 JS 부분을 응답 없음 상태로 만들지만, **React Native**의 스레딩(threading) 모델 덕분에 네이티브 UI 부분은 여전히 응답성을 유지할 것입니다.



모바일 앱에서 **60 FPS**를 달성하려면, 함수가 JS 스레드를 **16ms** 이상 차단(block)하지 않아야 합니다. 또한, 점점 보편화되는 **120 FPS**를 위해서는 이상적으로 **8ms**를 넘지 않아야 합니다.
FPS에 대한 자세한 내용은 '**JS FPS 측정 방법(How to Measure JS FPS)**' 챕터를 참고하세요.



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.

The screenshot shows the React DevTools JavaScript Profiler with the 'Heavy (Bottom Up)' view selected. This view lists the longest running functions in descending order of execution time. The first row shows 'longRunningFunction' with a self time of 2512.9 ms (52.02%) and a total time of 2512.9 ms (52.02%). Subsequent rows show nested functions: (anonymous), [Host Function] unstable_now, updateSlot, getPooledWarningPropertyDefinition, performConcurrentWorkOnRoot, dispatchEvent, renderRootSync, batchedUpdates\$1, workLoopSync, batchedUpdates, batchedUpdates, performUnitOfWork, resetRenderTimer, (anonymous), and beginWork. Each row includes columns for Self Time, Total Time, and Function name.

Self Time	Total Time	Function
2512.9 ms	52.02 %	▶ longRunningFunction
31.4 ms	0.13 %	▶ (anonymous)
15.7 ms	0.07 %	▶ [Host Function] unstable_now
15.7 ms	0.07 %	▶ updateSlot
15.7 ms	0.07 %	▶ getPooledWarningPropertyDefinition
0.0 ms	0.00 %	▶ performConcurrentWorkOnRoot
0.0 ms	0.00 %	▶ dispatchEvent
0.0 ms	0.00 %	▶ renderRootSync
0.0 ms	0.00 %	▶ batchedUpdates\$1
0.0 ms	0.00 %	▶ workLoopSync
0.0 ms	0.00 %	▶ batchedUpdates
0.0 ms	0.00 %	▶ batchedUpdates
0.0 ms	0.00 %	▶ performUnitOfWork
0.0 ms	0.00 %	▶ resetRenderTimer
0.0 ms	0.00 %	▶ (anonymous)
0.0 ms	0.00 %	▶ beginWork

프로파일러(Profiler) 출력: longRunningFunction 함수 실행에 12초 이상 소요된 플레임 차트(flame chart)

프로파일러(Profiler)는 예상보다 오래 실행되는 함수를 쉽게 찾아내고, 함수가 호출되는 위치까지 매우 정밀하게 추적하여 파악할 수 있게 해줍니다. 소프트웨어 엔지니어로서, 이 강력한 기능은 일상 업무에서 충분히 활용할 가치가 있습니다.

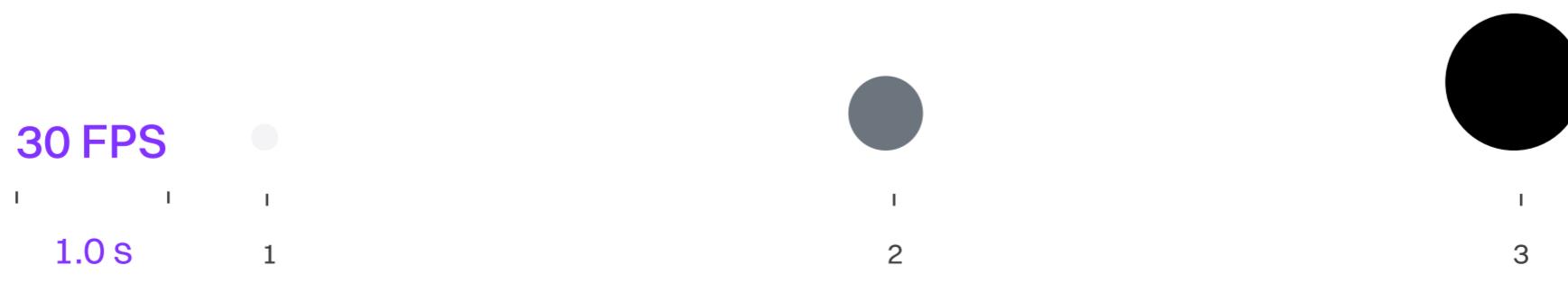
GUIDE

JS FPS 측정하기

모바일 사용자는 즉각적으로 반응하고 명확한 시각적 피드백을 제공하는, 부드럽고 잘 디자인된 인터페이스를 기대합니다. 그 결과, 애플리케이션에는 데이터 가져오기(data fetching)나 상태 관리(state management) 같은 다른 프로세스와 함께 실행되어야 하는 수많은 애니메이션이 포함되는 경우가 많습니다. 만약 앱이 반응성 좋은 인터페이스를 제공하지 못하거나, 사용자 입력이 지연되거나, 때때로 '버벅거리는(janky)' 느낌을 준다면, 이는 UI가 '프레임을 건너뛰고(dropping frames)' 있다는 신호입니다. 사용자는 이를 매우 싫어합니다.

FPS란 무엇인가

코드가 화면에 무언가를 표시하기까지는, 코드를 실행 가능한 형식으로 컴파일하고 다양한 기술을 사용하여 화면에 픽셀을 그리는 과정이 포함됩니다. 이렇게 한 번 화면을 그리는 것을 '프레임(frame)'이라고 합니다. 정적인 UI를 원치 않는다면 (대부분 그렇듯이), 앱은 매초 많은 프레임을 그려야 합니다. 대부분의 모바일 기기는 60Hz의 주사율(refresh rate)을 가지는데, 이는 초당 60개의 프레임, 즉 60 FPS(Frames Per Second)를 표시할 수 있음을 의미합니다. 이것이 우리가 사용할 지표입니다.



30 FPS와 90 FPS 비교

대부분의 사람들은 60 FPS를 부드러운 움직임으로 인식합니다. 하지만 기술이 발전하면서, 사용자들은 초당 120번 또는 심지어 240번까지 새로 고침되는 화면(120Hz, 240Hz)을 접하게 되었습니다. 일단 사용자가 120Hz 화면의 부드러움을 경험하면, 60Hz는 '끊기거나(choppy)', '느리다(laggy/slow)'고 여길 가능성이 높습니다. 이는 프레임당 16.6ms(초당 60프레임 기준)라는 우리의 목표치가 변하고 있으며, 이제는 프레임당 8.3ms(초당 120프레임 기준)를 목표로 해야 함을 의미합니다.

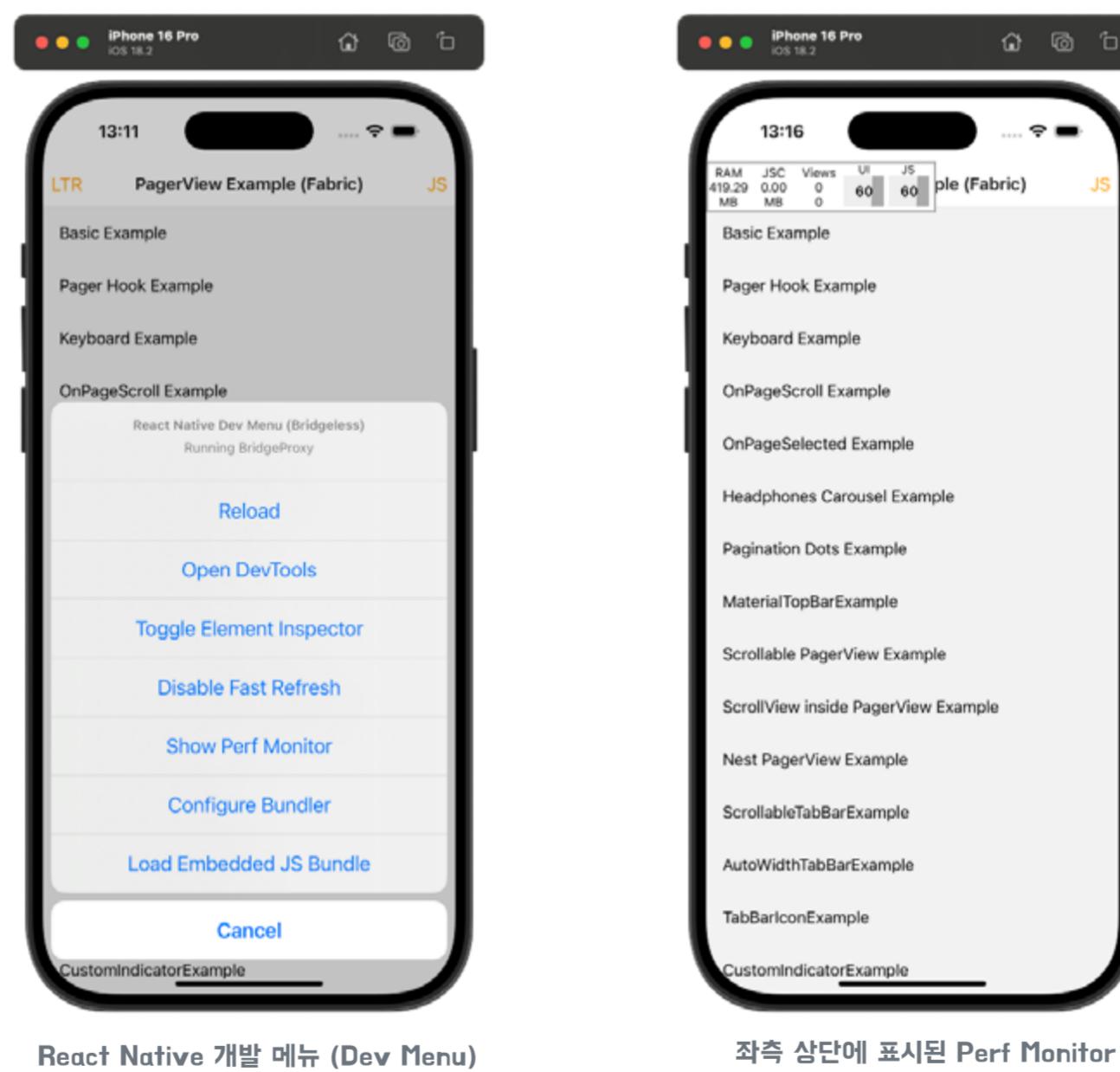
💡 앱 성능에 대해 충분히 고민하고 문제 해결을 위한 적절한 도구를 선택하지 않는다면, 조만간 프레임 드롭(frame drop)을 경험하기 십상입니다.

React Perf Monitor

다행히도 React Native에는 React Perf Monitor라는 유용한 도구가 내장되어 있는데, 이 도구는 앱 위에 오버레이(overlay) 형태로 실시간 FPS를 표시할 수 있습니다. React Native 개발 메뉴(Dev Menu)에서 이 도구를 선택하여 열 수 있습니다.

💡 개발 메뉴(DevMenu)는 기기를 흔들거나 전용 키보드 단축키를 사용하여 열 수 있습니다:

- iOS 시뮬레이터: Ctrl + Cmd ⌘ + Z (또는 메뉴의 Device > Shake 선택).
- Android 에뮬레이터: (Cmd ⌘ / Ctrl) + M.



그러면 애플리케이션 좌측 상단에 작은 창(pane)이 열립니다. React 개발 메뉴(DevMenu)에 다시 접근하여 'Hide Perf Monitor' 옵션을 선택하면 이 창을 숨길 수 있습니다.

이를 통해 메모리 사용량과 FPS 저하 시점을 모니터링할 수 있습니다. 실제로는 UI (또는 메인) 스레드용과 JS 스레드용, 이렇게 두 개의 FPS 모니터가 있습니다. 이를 통해 특정 상호작용(interaction)의 속도 저하 원인이 자바스크립트(JavaScript)인지 네이티브(native)인지 빠르게 파악할 수 있습니다.



성능 측정 시에는 항상 개발 모드(development mode)를 비활성화하세요.

Android에서는 개발 메뉴(Dev Menu)의 'Settings > JS Dev Mode'에서 설정할 수 있습니다. iOS에서는 개발 모드가 비활성화되도록 앱을 빌드/실행해야 합니다.

'JS 및 React 코드 프로파일링 방법(How to Profile JS and React Code)' 챕터에서 배웠듯이, 애니메이션이나 특정 작업 중 FPS가 떨어질 때는 프로파일러(profiler)를 사용하여 원인을 좁혀나갈 수 있습니다.

Flashlight

개발 메뉴(Dev Menu)의 FPS는 보기에는 쉽지만 추적하고 측정하기는 어렵습니다. 특정 상호작용(interaction)이나 화면 로딩 시의 평균 FPS 같은 값을 알 수 있다면 매우 편리할 것입니다. 바로 이런 지점에서 Flashlight 같은 도구가 유용합니다. 단, Android에서만 사용할 수 있습니다.

시작하려면, 먼저 Flashlight가 설치되어 터미널에서 사용 가능한지 확인하세요. 측정하려는 Android 애플리케이션을 특정 Android 에뮬레이터에서 열어둔 다음, 아래 명령어를 실행합니다:

```
▶ flashlight measure
```

애플리케이션과 상호작용하는 동안, 다양한 성능 지표(performance metrics)가 포함된 다이어그램(diagram)이 실시간으로 표시되는 것을 볼 수 있습니다.



평균 FPS를 얻는 것은 이 도구(Flashlight)가 생성하는 [Lighthouse](#)와 유사한 보고서(lighthouse-like report)에서 제공하는 여러 기능 중 하나일 뿐입니다. 이 보고서에는 측정 중 수집된 지표들(예: 평균 FPS 수치, 평균 CPU 및 RAM 사용량 등)에 기반한 앱의 전반적인 성능 점수도 포함됩니다.

수집된 모든 측정값은 JSON 파일에 저장되어, 코드 버전 간 비교를 위해 저장하고 활용될 수 있습니다. 이는 자동화된 로컬 성능 모니터링과 FPS가 확실히 개선되었음을 증명하는 데 좋은 해결책입니다.

GUIDE

JS 메모리 누수 찾기

프로그램이 대상 기기나 가상 머신에서 실행될 때, 해당 프로그램의 올바른 실행을 위해 특별히 할당된 기기의 RAM(Random Access Memory) 공간 일부를 항상 차지합니다. 이 할당된 메모리가 예상치 못하게 해제되지 않고 유지될 때, 우리는 이것이 '새고 있다(leaking)'고 말하며, 이것이 바로 메모리 누수(memory leak)입니다.

메모리 누수는 적절한 도구 없이는 추적하기 어려울 수 있습니다. 대부분의 경우 프로그래머의 실수로 발생합니다. 만약 앱이 메모리를 해제하지 않고 지속적으로 점점 더 많이 사용한다면, 어딘가에 누수가 있을 가능성이 매우 높습니다. 무엇이 누수를 일으키는지 추측해볼 수도 있겠지만, 누수를 찾는 데 도움이 되는 적절한 도구로 앱을 프로파일링하는 것이 훨씬 쉽습니다. 먼저 메모리 누수가 무엇인지 정의하는 것부터 시작하겠습니다.

자바스크립트(JavaScript) 앱에서의 메모리 누수 구조

Hermes와 같은 자바스크립트(JavaScript) 엔진 및 가상 머신을 활용하여 실행되는 다른 인터프리터 언어들은 '가비지 컬렉터(garbage collector, GC)'라는 특별한 프로그램을 구현합니다. 이름에서 알 수 있듯이, 이는 '쓰레기(garbage)'를 수집하는데, 여기서 '쓰레기'는 애플리케이션 실행에 더 이상 필요하지 않아 다른 프로그램이 사용할 수 있도록 해제될 수 있는 메모리 주소를 의미합니다. GC는 주기적으로 할당된 객체들을 스캔하여 더 이상 필요하지 않은 객체인지 판단합니다. (재미있는 사실: Hermes의 가비지 컬렉터 이름은 Hades입니다.)

가비지 컬렉터는 어떤 메모리를 해제할지 안정적이고 안전하게 결정하기 위해 고급 기술을 사용 하지만, 경우에 따라서는 코드가 GC를 속여서 실제로는 필요 없는 객체를 계속 할당된 상태로 유지하게 만들 수도 있습니다. 이것이 바로 전용 도구 없이는 추적하기 어려운 메모리 누수가 발생하는 방식입니다.



요컨대, 메모리 누수는 프로그램이 더 이상 필요하지 않은 메모리를 해제하지 못할 때 발생합니다.

이해를 돋기 위해, 메모리 누수를 일으키는 코드의 몇 가지 예시를 들어보겠습니다.

1. 리스닝(listening)을 멈추지 않는 리스너(Listener):

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

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

2. 카운팅(counting)을 멈추지 않는 타이머(Timer):

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

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

3. 큰 객체 참조(reference)를 계속 유지하는 클로저(Closure):

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

  createLeakyFunction(): () => number {
    // 클로저(Closure) 때문에 전체 largeData 배열이 메모리에서 해제되지 않음
    return () => this.largeData.length;
  }
}

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

  createEfficientFunction(): () => number {
    // 전체 배열이 아닌, length(길이) 값만 참조하기
    const length = this.largeData.length;
    return () => length;
  }
}
```

이제 일반적인 메모리 누수가 어떤 형태인지 알았으니, 이를 쉽게 찾아낼 수 있는 도구들을 살펴보겠습니다.

React Native DevTools로 메모리 누수 추적하기

새로운 **React Native DevTools**는 [Chrome DevTools](#) 기반으로 만들어졌으므로, 웹 디버깅에 익숙하다면 매우 친숙하게 느껴질 것입니다. [Chrome DevTools](#)에 대해 더 알고 싶다면 공식 문서를 확인해보세요.

React Native DevTools는 앱의 메모리를 프로파일링하는 세 가지 방법을 제공합니다:

- **힙 스냅샷(Heap snapshot)** 페이지의 자바스크립트(JavaScript) 객체와 관련 DOM 객체 간의 메모리 분포를 보여줍니다.
- **타임라인의 할당 계측(Allocation instrumentation on the timeline)** 시간 경과에 따른 계측된 자바스크립트 메모리 할당을 보여줍니다. 프로파일 기록 후, 특정 시간 간격을 선택하여 해당 간격 내에 할당되었고 기록 종료 시점까지 여전히 활성 상태인 객체를 볼 수 있습니다. 메모리 누수를 분리(isolate)할 때 이 프로파일 유형을 사용합니다.
- **할당 샘플링(Allocation sampling)** 샘플링(sampling) 방식을 사용하여 메모리 할당을 기록합니다. 이 프로파일 유형은 성능 오버헤드가 최소화되어 오래 실행되는 작업에 사용할 수 있습니다. 자바스크립트 실행 스택(execution stack)별로 분류된 할당량의 좋은 근사치를 제공합니다.

우리는 할당이 해제되지 않는 경우를 찾는 데 도움이 되는 '타임라인의 할당 계측(Allocation instrumentation on the timeline)'에 집중할 것입니다. 다음 코드 조각에 이 방법을 실행해 보겠습니다:

```
// 클로저 저장용 전역 변수
let leakyClosures: Function[] = [];

// 더미 데이터 생성
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: '메모리 소모가 큰 중첩 데이터',
      }),
    },
  }));
};

const createClosure = () => {
  const newDataLeak = generateLargeData();

  return () => {
```

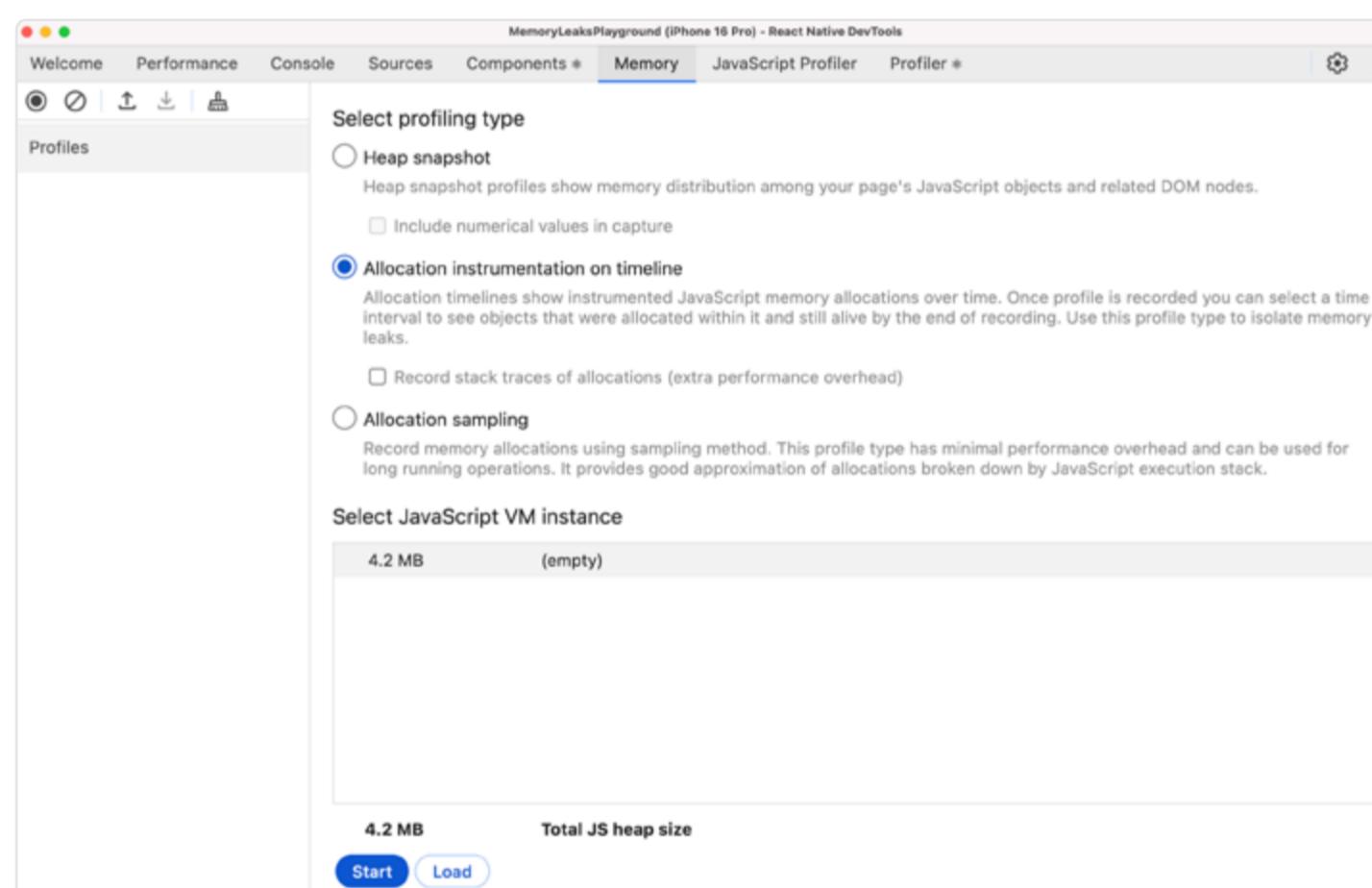
```
newDataLeak.forEach((data) => data.id); // 클로저(Closure)가
'newDataLeak' 참조 유지
};

// 각각 자신만의 큰 데이터를 참조하는 많은 클로저(Closure) 생성
const createManyLeaks = () => {
  for (let i = 0; i < 10; i++) {
    const leakyClosure = createClosure();
    leakyClosures.push(leakyClosure); // GC(가비지 컬렉션) 방지를
    위해 참조(reference) 저장
  }
}

// 모든 클로저(Closure)를 실행하여 데이터를 '활성(active)' 상태로 유지
leakyClosures.forEach(closure => closure());
```

앞 섹션의 세 번째 문제(큰 객체 참조를 계속 유지하는 클로저)를 보여주는 코드

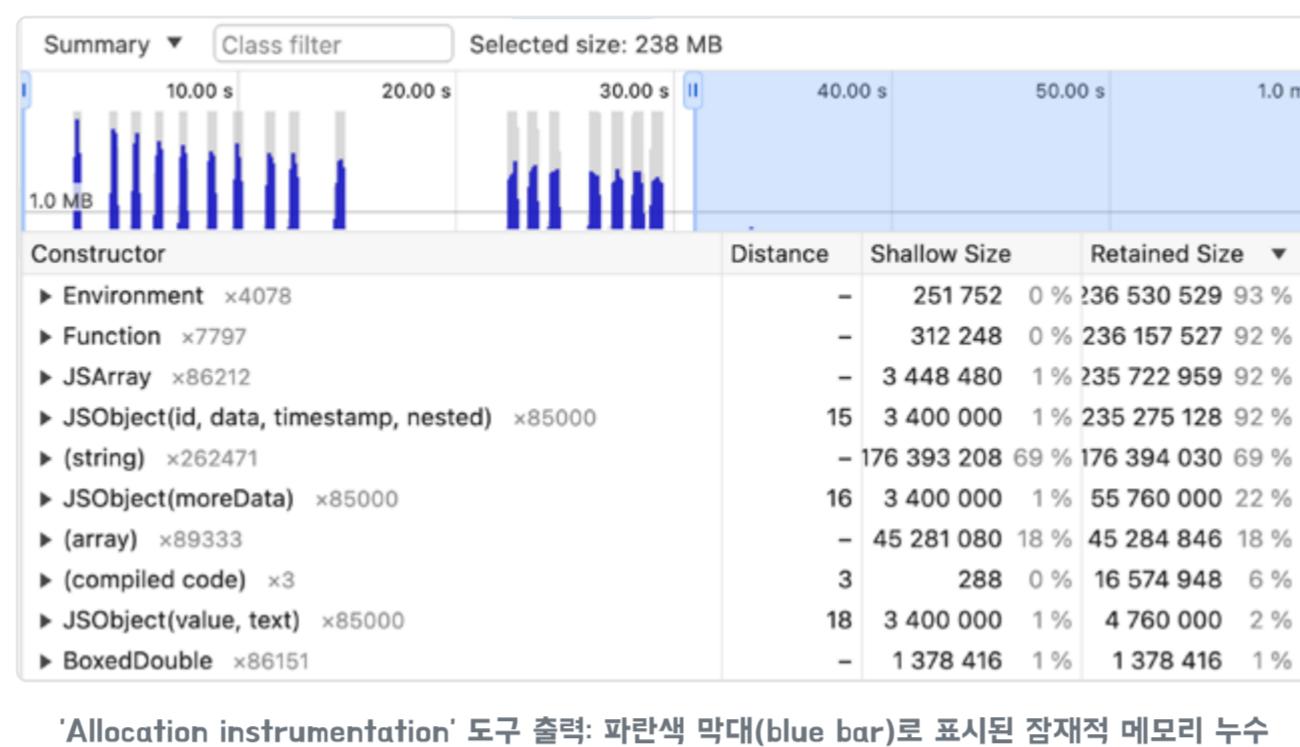
DevTools를 실행한 후, 'Memory' 탭을 찾아 'Allocation instrumentation on timeline'을 선택합니다. 그런 다음 'Start' 버튼을 클릭합니다.



React Native DevTools의 Memory 패널

프로파일링을 마쳤는데도 객체(object)가 여전히 메모리에 할당되어 있다면, 누수가 있는 것입니다! 하지만 앞서 언급했듯이, GC(가비지 컬렉터)는 주기적으로 실행되므로 객체가 즉시 할당 해제(deallocate)되지 않을 수도 있습니다. 기존 객체의 할당 해제를 유발(trigger)하기 위해 다른 것을 할당해야 할 수도 있습니다.

스냅샷(snapshot)을 캡처한 후 프로파일러(Profiler)는 다음과 같이 보입니다:

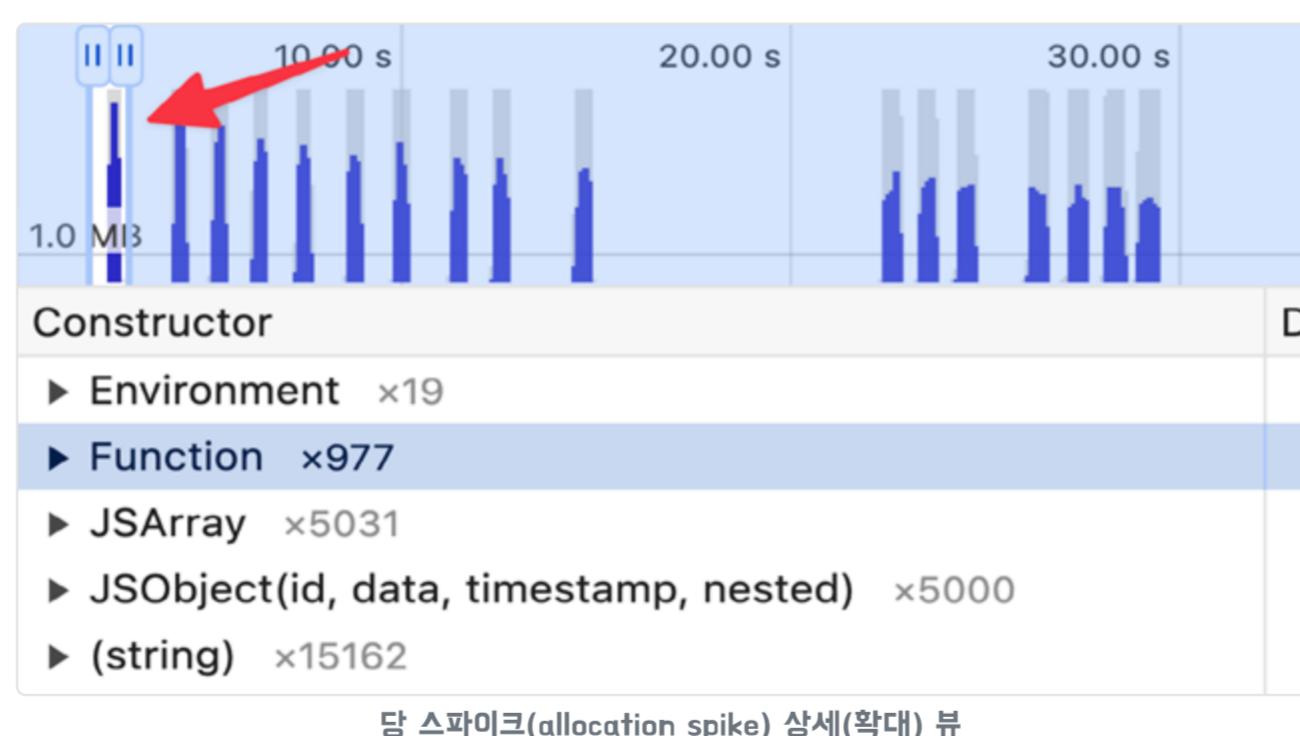


'Allocation instrumentation' 도구에 대해 몇 가지 기억해야 할 점이 있습니다:

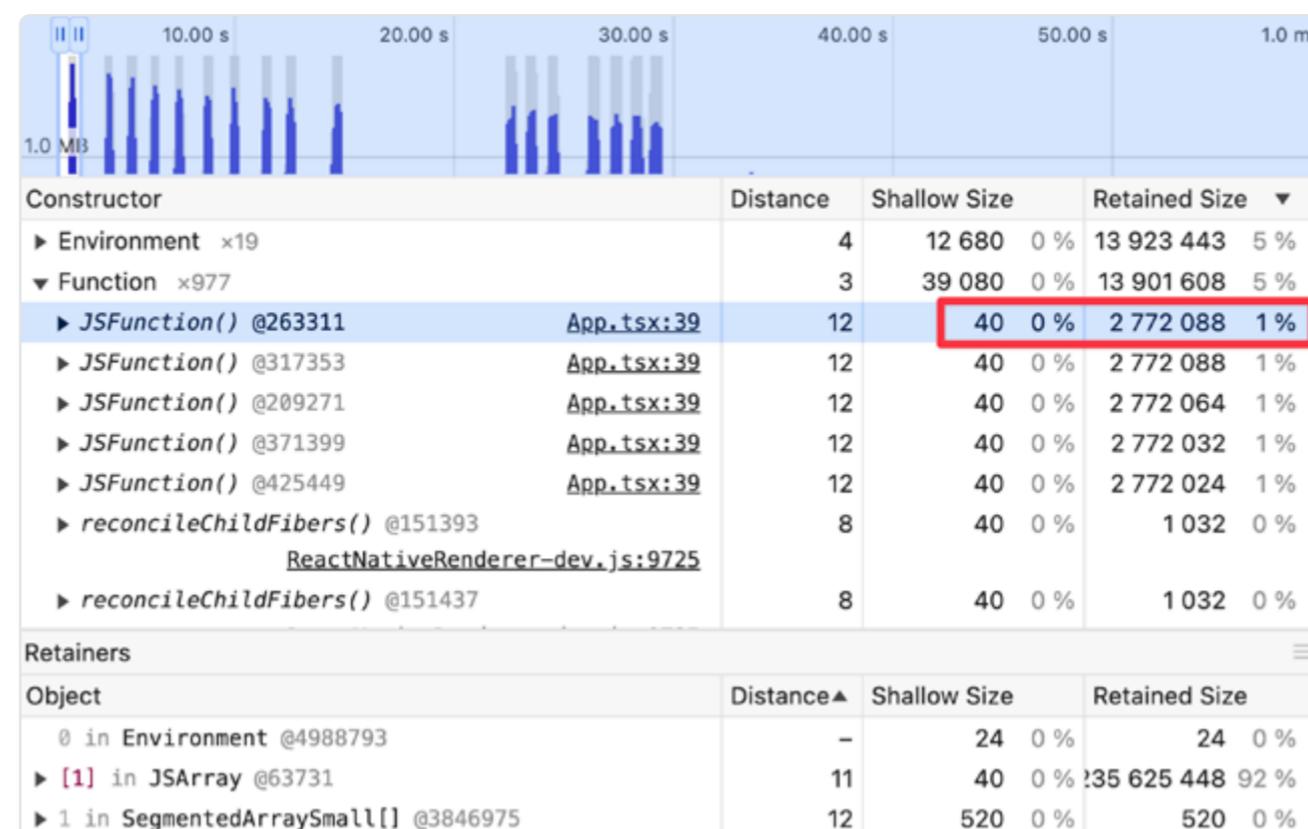
- 타임라인의 파란색 막대(blue bar)는 메모리 할당을 나타냅니다.
- 파란색 막대가 회색으로 바뀌면 해당 객체가 할당 해제(deallocate)된 것입니다.
- 아래쪽에는 호출된 생성자(constructor) 목록이 표시되며, 그 바로 옆에는 다음 두 가지 측정 항목(metrics)이 있습니다:
 - 얕은 크기(Shallow size) 객체 자체가 보유한 메모리 크기.
 - 유지된 크기(Retained size) 해당 객체가 삭제된 후 해제될 메모리 크기.

보시다시피, 30초가 넘는 시간 동안 `createManyLeaks` 함수가 여러 번 호출되었지만, GC(가비지 컬렉터)는 메모리에서 일부 객체만 정리했습니다(그래프에 회색으로 표시됨). 하지만 파란색 부분은 여전히 활성 상태(alive)입니다. 이 그래프는 여러 누수가 발생하고 있음을 명확히 보여줍니다("screams").

첫 번째 스파이크(spike)에 집중하여 원인이 무엇인지 조사해 보겠습니다.



이제 그래프가 첫 번째 스파이크(spike)에 맞춰졌으므로, 어떤 생성자(constructor)가 호출되고 있는지 볼 수 있습니다. `Function` 생성자를 확장하여 무엇이 있는지 살펴보겠습니다.



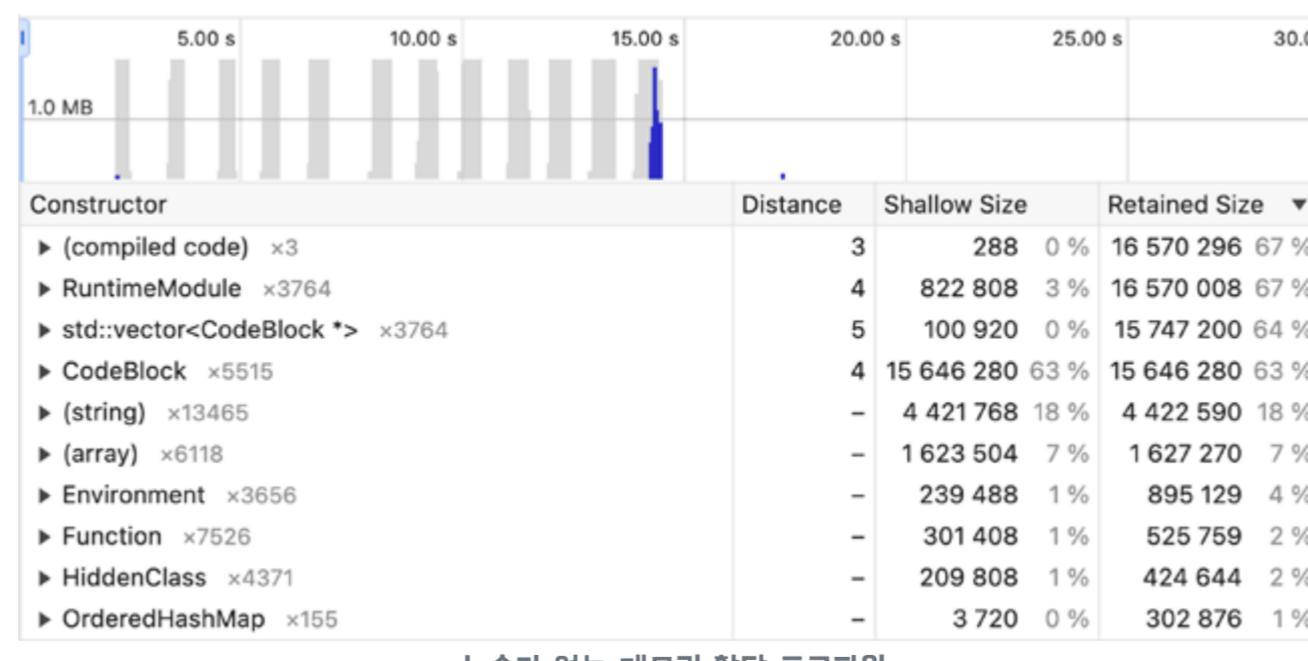
JSFunction의 얕은 크기(Shallow size)와 유지된 크기(Retained size) 간의 차이

보시다시피, JSFunction() 생성자 호출의 얕은 크기(shallow size)는 40 바이트(bytes)이지만 유지된 크기(retained size)는 무려 2,772,088 바이트(!)입니다. 이것이 바로 우리가 찾던 누수입니다! 이 클로저(closure)는 실제로는 더 이상 필요 없는 객체들을 여전히 필요한 것처럼 GC(가비지 컬렉터)를 속이고 있는 것입니다. DevTools는 심지어 이 생성자가 호출된 정확한 코드 라인까지 보여줍니다.

누수를 찾았다면 수정 후 다시 테스트해야 합니다. createManyLeaks 함수를 변경하여, 클로저(closure)가 유지(retain)되지 않도록 배열에 추가(push)하는 부분을 주석 처리해 보겠습니다.

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

보시다시피, 이제 (아직 할당 해제(deallocate)되지 않은 마지막 막대를 제외하고) 모든 막대(bar)가 회색입니다. 이는 누수가 없다는 의미입니다!



누수가 없는 메모리 할당 프로파일

이제 메모리 누수를 추적하고 수정하는 방법을 알게 되었습니다! 다음 챕터에서는 '제어 컴포넌트(Controlled Components)'에 대해 집중적으로 알아보겠습니다.

BEST PRACTICE

UNCONTROLLED COMPONENTS

React 프로그래밍 모델은 상태(state) 변경이 있을 때마다 (컴포넌트 트리(tree)로 표현되는) 전체 앱을 리렌더링한다는 개념을 중심으로 합니다. 이 모델은 대부분의 UI 프로그래밍 사용 사례에 적합하지만, 이는 추상화(abstraction)일 뿐입니다. 다른 모든 추상화와 마찬가지로, 이는 현실을 단순화하여 이해하기 쉽게 만들지만, 정밀함(correctness/precision)을 어느 정도 희생합니다. 좋은 추상화는 '비상 탈출구(escape hatch)'를 포함하는데, 이는 프로그래머로서 목표 달성을 위해 때로는 그 추상화에서 벗어나도 된다는 제작자의 허락과 같습니다.

React도 다르지 않아서, refs와 같은 다양한 '비상 탈출구'를 제공합니다. 이는 React의 리렌더링 로직을 우회하고 상태 업데이트에 의해 구동되지 않는 컴포넌트를 가능하게 합니다. 우리는 이를 '비제어 컴포넌트(uncontrolled components)'라고 부르며, 이 챕터에서는 특히 React Native의 레거시 비동기 아키텍처(legacy asynchronous architecture) 맥락에서 이 강력한 패턴을 탐색할 것입니다.

레거시 아키텍처에서의 제어 TextInput(Controlled TextInput)

거의 모든 React Native 애플리케이션에는 텍스트나 음성 같은 어떤 형태의 입력(input)이 포함됩니다. 여기서는 훨씬 더 일반적인 텍스트 입력에 초점을 맞추겠습니다. React Native 앱에서 텍스트 입력은 TextInput 컴포넌트로 표현됩니다. 이 컴포넌트는 React props를 통해 제어되거나(controlled), refs 및 콜백(callback) props를 활용하여 React 모델과 다시 통신하는 비제어(uncontrolled) 방식일 수 있습니다.

먼저 이러한 제어되는 텍스트 입력(controlled text input)의 예시를 살펴보겠습니다:

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

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

  return (
    <TextInput
```

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

위 코드 예시는 사용자가 텍스트를 입력할 때마다 React 모델을 통해 네이티브 텍스트 입력(**native text input**) 값이 업데이트되도록 합니다. 텍스트 입력 속도는 매우 빠르거나 자동 완성(autocomplete) 기능의 도움을 받는 경우도 많습니다. 여기에 저사양 Android 기기 환경이거나 사용자가 입력하는 동안 앱이 무거운 연산(heavy computing)을 수행하는 상황이 더해지면, 사용자는 지연(lag)이나 프레임 드롭(frame drop)을 경험할 수 있습니다. 심지어 레거시 비동기 아키텍처(legacy asynchronous architecture)에서는 React 상태(state)가 네이티브 입력(**native input**) 상태와 동기화되지 않아 예기치 않은 동작을 유발할 수도 있습니다.

 **New Architecture**에서는 제어되는 **TextInput**(controlled **TextInput**)의 동기화 해제(de-synchronization) 문제가 발생하지 않을 것입니다.

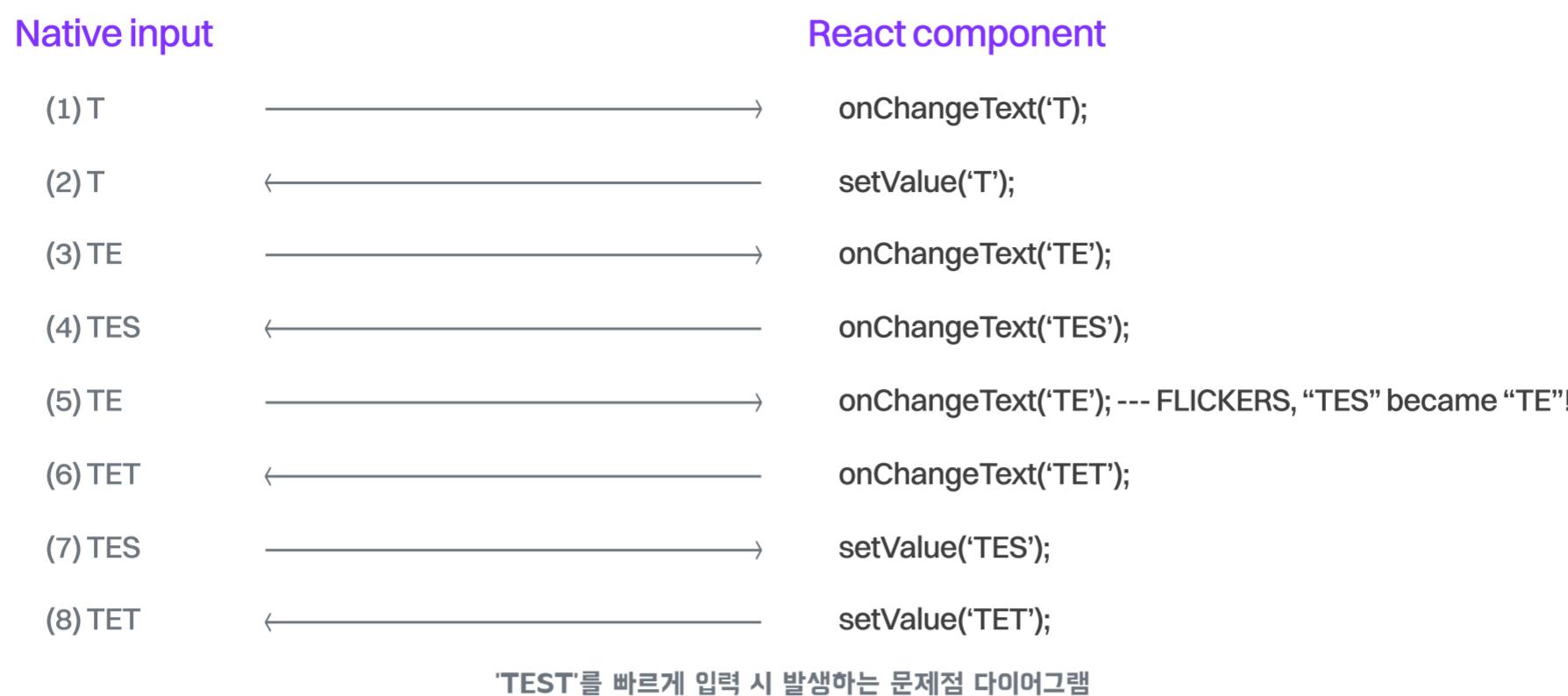
무슨 일이 일어나는지 더 잘 이해하려면, 사용자가 `<TextInput />`에 문자를 입력할 때 일반적으로 발생하는 작업 순서를 먼저 봅시다.



사용자가 네이티브 입력(**native input**)에 새 문자를 입력하는 즉시, `onChangeText` prop을 통해 React Native로 업데이트가 전송됩니다(위 다이어그램의 작업 1). React는 이 정보를 처리하고 `setState`를 호출하여 상태(state)를 그에 맞게 업데이트합니다. 다음으로, 제어 컴포넌트(controlled component)는 자신의 자바스크립트(JavaScript) 값을 네이티브 컴포넌트 값과 동기화합니다(위 다이어그램의 작업 2).

이러한 접근 방식에는 장점이 있습니다. React가 입력 값(`input value`)을 결정하는 '신뢰할 수 있는 단일 출처(source of truth)' 역할을 합니다. 이 기술을 사용하면 유효성 검사(validation) 수행, 마스킹(masking) 처리, 또는 완전히 수정하는 방식으로 사용자가 입력하는 대로 내용을 변경할 수 있습니다.

불행히도, 이 방식은 궁극적으로 더 깔끔하고 React 작동 방식에 더 부합하지만, 한 가지 단점이 있습니다. 이는 가용 자원(resource)이 제한적이거나 사용자가 매우 빠른 속도로 타이핑할 때 가장 두드러집니다.



onChangeText를 통한 업데이트가 React Native가 각각을 다시 동기화하기 전에 도착하면, 인터페이스가 깜빡이기(flickering) 시작합니다. 첫 번째 업데이트(작업 1 및 작업 2)는 사용자가 'T'를 입력하기 시작할 때 문제없이 수행됩니다.

다음으로 작업 3이 도착하고 연이어 작업 4가 도착합니다. 사용자는 React Native가 다른 작업으로 바쁜 동안 'E'와 'S'를 입력했고, 이로 인해 문자 'E'의 동기화(작업 5)가 지연됩니다. 그 결과, 네이티브 입력(native input) 값은 일시적으로 'TES'에서 'TE'로 다시 변경됩니다.

이제 사용자는 텍스트 입력 값이 잠시 'TE'로 설정되었을 때 실제로 다른 문자를 입력할 만큼 충분히 빠르게 타이핑했습니다. 그 결과, 'TET' 값을 가진 또 다른 업데이트(작업 6)가 도착했습니다. 이는 의도된 것이 아니며, 사용자는 입력 값이 'TES'에서 'TE'로 변경될 것이라고 예상하지 않았습니다.

마지막으로 작업 7은 몇 글자 전에 사용자로부터 받은 올바른 입력('TES', 작업 4에서 알려줌)으로 입력을 다시 동기화했습니다. 불행히도, 이 값은 즉시 다른 업데이트(작업 8)에 의해 덮어쓰여 최종 입력 값인 'TET'으로 동기화되었습니다.

이 상황의 근본 원인은 작업 순서에 있습니다. 만약 작업 5가 작업 4보다 먼저 실행되었다면 모든 것이 순조롭게 진행되었을 것입니다. 또한, 만약 사용자가 값이 'TES' 대신 'TE'였을 때 'T'를 입력하지 않았다면, 인터페이스는 깜빡였겠지만 입력 값 자체는 올바르게 유지되었을 것입니다.

비제어 TextInput(Uncontrolled TextInput)

이 동기화 문제에 대한 한 가지 해결책은 TextInput에서 value prop을 완전히 제거하여 비제어 컴포넌트(uncontrolled component)로 만드는 것입니다.

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

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

```
    setValue(text);
};

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

그 결과, 데이터는 iOS의 네이티브 `RCTSingleTextInputView` 또는 Android의 `AndroidTextInputNativeComponent`에서 자바스크립트(JavaScript) 측으로만 단방향으로 흐르게 됩니다. 네이티브 컴포넌트는 `onChangeText` 이벤트를 발생시키고 내부 입력 상태(`internal input state`)를 자체적으로 제어하므로, 앞서 설명한 동기화 문제가 제거됩니다.

BEST PRACTICE

HIGHER-ORDER SPECIALIZED COMPONENTS

React Native 애플리케이션에서는 거의 모든 것이 컴포넌트입니다. 컴포넌트 계층(hierarchy)의 가장 끝에는 소위 '기본(primitive)' 컴포넌트인 `Text`, `View`, `TextInput` 등이 있습니다. 이 컴포넌트들은 React Native에 의해 구현되며, 대상 플랫폼에서 가장 기본적인 사용자 상호작용(interaction)을 지원하기 위해 제공됩니다.

우리는 애플리케이션을 만들 때 더 작은 빌딩 블록(building block)들을 조합합니다. 이를 위해 기본(primitive) 컴포넌트를 사용합니다. 예를 들어, 로그인 화면을 만들려면 여러 개의 `TextInput` 컴포넌트를 사용하여 사용자 정보를 등록하고 `Pressable` 컴포넌트로 사용자 상호작용을 처리할 것입니다.

이러한 접근 방식은 애플리케이션 내에서 만드는 가장 첫 컴포넌트부터 최종 개발 단계까지 유효하게 적용됩니다.

기본(primitive) 컴포넌트 외에도, `react-native` 패키지 및 서드파티(third-party) 라이브러리들은 특정 목적을 위해 설계되고 최적화된 다양한 고차 컴포넌트(higher-order components)를 함께 제공합니다. 이를 사용하지 않으면, 특히 실제 운영 환경의 데이터로 상태(state)를 채울 때 애플리케이션 성능에 영향을 미칠 수 있습니다.

리스트(목록) 표시하기

리스트(list)를 예로 들어 보겠습니다. 모든 애플리케이션에는 어떤 형태로든 리스트가 포함됩니다. 웹 개발에서 흔히 아는 기본 방식은 `<ScrollView />` 내부에 `<View />` 컴포넌트들을 조합하여 요소 목록을 만드는 것입니다:

```
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;
```

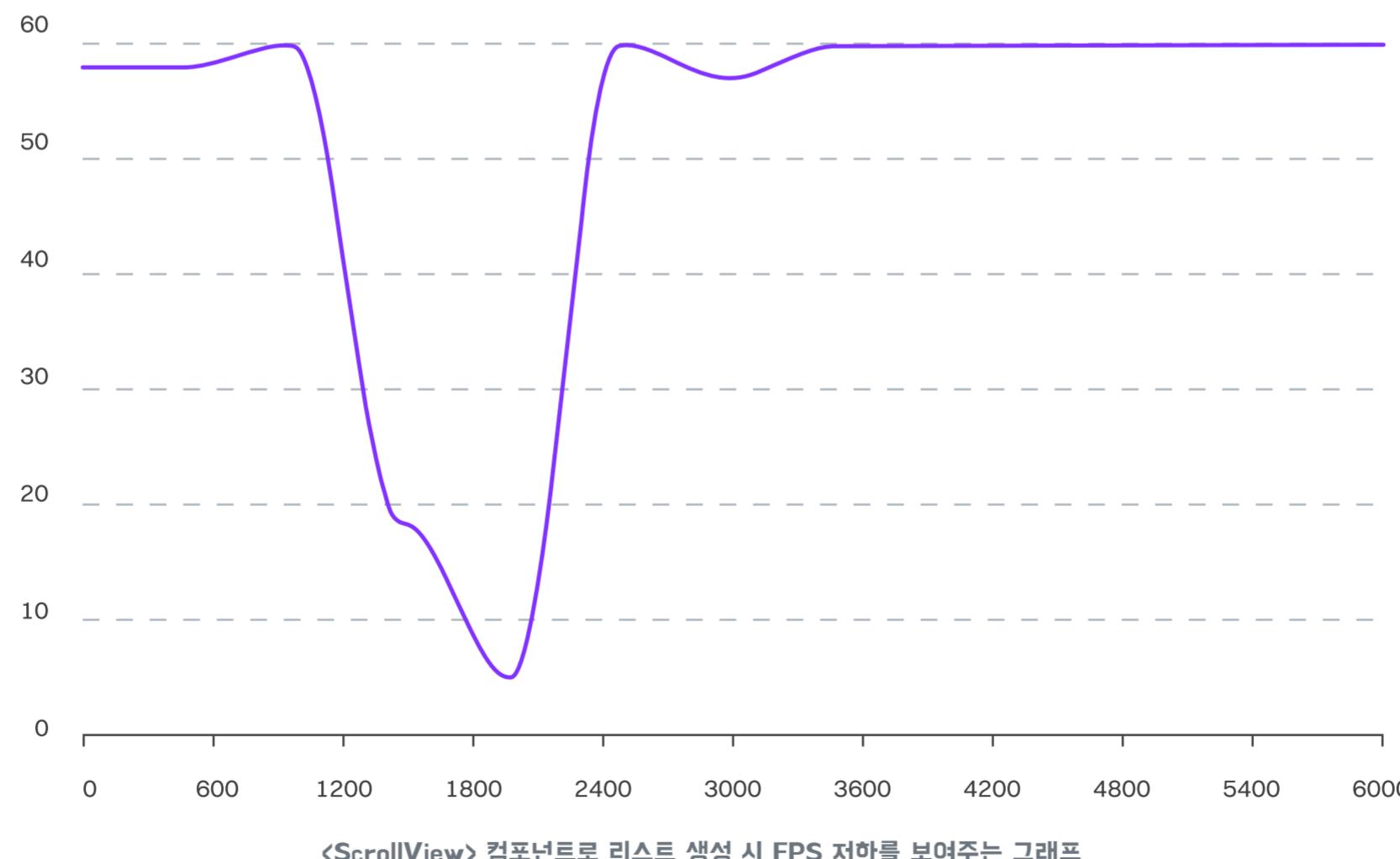
Source: <https://snack.expo.dev/@callstack-snack/9374a7>

이러한 방식은 애플리케이션 개발 초기에는 잘 작동하지만, 금방 문제가 발생합니다. 쿼리(query) 결과로 이 리스트에 5000개의 항목(item)이 반환될 때 어떤 일이 발생하는지 살펴보겠습니다:

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

아래 그래프에서 볼 수 있듯이, FPS로 측정한 성능은 애플리케이션이 약 1초간 응답하지 않을 정도로 크게 떨어졌습니다.

Frame rate (FPS)



FlatList로 전환하기

앞서 언급한 문제는 기본적인 해결책(primitive solution)에서 벗어나 특화된 FlatList 컴포넌트로 전환함으로써 빠르게 해결될 수 있습니다. FlatList는 React Native에 포함되어 있으며, 리스트(list)에 많은 수의 요소를 효율적으로 표시하는 목적으로 특화되어 설계되었습니다.

ScrollView를 FlatList로 교체하고 성능에 어떤 영향을 미치는지 살펴보겠습니다:

```
-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}`);

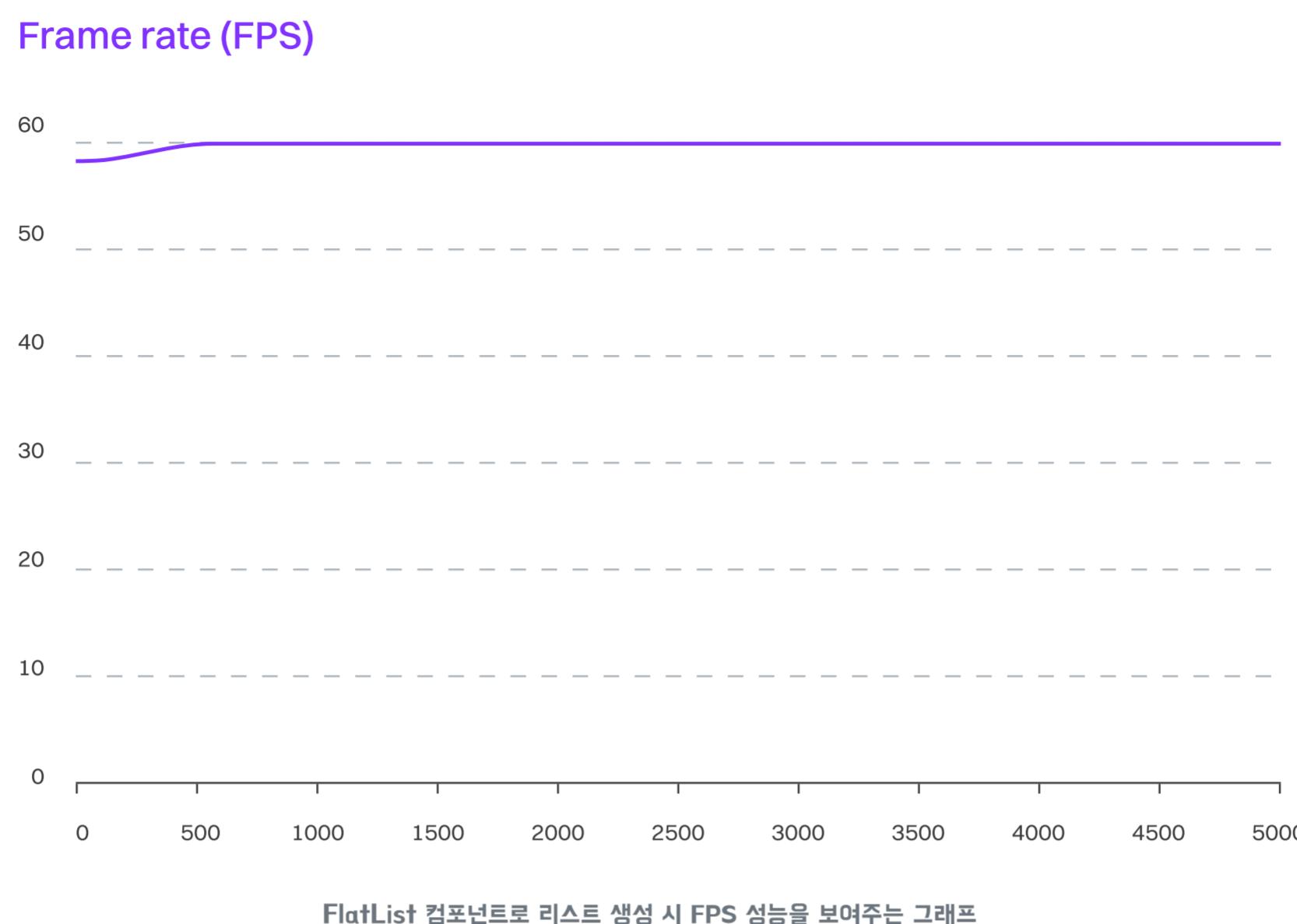
  + const renderItem = ({ item }) => (
    +   <View>
    +     <Text>{item}</Text>
    +   </View>
    + );

  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>

이제 예제는 스크롤할 때 프레임 드롭(frame drop) 없이 실행됩니다.



그 차이는 매우 뚜렷합니다. 무엇이 이렇게 큰 성능 향상을 가져오는지 궁금하실 것입니다. 결국에는 <FlatList />도 내부적으로는(under the hood) 동일한 기본(primitive) 컴포넌트를 사용하며, 결과적으로는 ScrollView 안에 View들을 표시하는 것처럼 보일 수 있습니다.

FlatList가 ScrollView보다 빠른 이유

핵심은 <FlatList /> 컴포넌트 내부에 추상화된(abstracted away) 로직에 있습니다. 여기에는 화면에 데이터를 표시하는 동안 발생하는 불필요한 렌더링(extraneous rendering) 횟수를 줄이고, 스크롤 경험(scrolling experience)이 항상 60 FPS로 실행되도록 보장하기 위한 많은 헤리스틱스(heuristics)과 고급 자바스크립트(JavaScript) 연산이 포함되어 있습니다.

FlatList는 내부적으로 VirtualizedList를 사용하는데, 이는 '윈도잉(windowing)'을 구현합니다. 이 기법은 현재 뷰포트(viewport)에 보이는 항목과 약간의 버퍼(buffer)만 렌더링하고 마운트(mount)합니다. 스크롤하면, 뷰에서 벗어나는 항목은 동적으로 마운트 해제(unmount)하고 뷰 안으로 들어오는 새 항목을 마운트하여, 활성 항목의 유한한 렌더 창(render window)을 유지합니다. 이는 대부분의 시나리오에서 메모리 사용량을 크게 줄이고 부드러운 스크롤 성능을 보장합니다.

FlatList에서 복잡한 요소 렌더링하기

하지만 경우에 따라서는 FlatList만 사용하는 것만으로는 충분하지 않을 수 있습니다. FlatList의 성능 최적화는 현재 화면 밖에 있는 요소를 렌더링하지 않는 것에 의존하기 때문입니다.

그렇긴 하지만, 전체 과정에서 비용이 가장 많이 드는 부분은 레이아웃 측정(layout measurement)입니다.

FlatList는 앞으로 표시될 요소들을 위해 스크롤 영역에 얼마만큼의 공간을 예약해야 할지 결정하기 위해 레이아웃을 측정해야 합니다. 리스트 요소가 복잡할 경우, 컴포넌트가 레이아웃을 측정하기 위해 모든 항목이 렌더링될 때까지 기다려야 하므로 상호작용(interaction) 속도가 느려질 수 있습니다.

이 문제를 완화하려면, `getItemLayout()` 함수를 구현하여 요소 크기(높이 또는 너비)를 미리 지정해주면 됩니다. 이는 레이아웃 측정 단계를 건너뛰어 성능을 크게 향상시킵니다.

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

const NUMBER_OF_ITEMS = 10;
+ const ITEM_HEIGHT = 50; // 리스트 아이템 컴포넌트 높이 정의/계산하기

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>

높이가 고정되지 않은 항목(item)의 경우 이는 간단하지 않습니다. 하지만 텍스트 줄 수나 다른 레이아웃 제약 조건(layout constraint)을 기반으로 값을 계산할 수 있습니다.

FlatList의 대안으로서의 FlashList

이미 논의했듯이, FlatList는 ScrollView에 비해 거대한 리스트의 성능을 크게 향상시킵니다. 하지만 FlatList 역시 성능 좋은 해결책임에도 불구하고, 스크롤 중 빈 공간이 렌더링되거나, 스크롤이 느리고(laggy), 리스트 반응이 즉각적이지 않은(snappy) 것과 같은 몇 가지 문제점 (caveat)을 가집니다.

또한, `FlatList`는 특정 요소들을 메모리에 유지하도록 설계되었는데, 이는 기기에 오버헤드 (`overhead`)를 추가하고 결과적으로 리스트 속도를 저하시킬 수 있습니다.



공식 문서의 팁(tip)들을 따르면 앞서 언급된 증상들(빈 공간, 느린 스크롤 등)의 발생 빈도를 어느 정도는 줄일 수 있습니다. 하지만 대부분의 경우, 우리는 추가적인 (최적화) 작업 없이도 더 부드럽고 즉각적으로 반응하는(`snappy`) 리스트를 원합니다.

`FlatList`를 사용하면 JS 스레드가 대부분의 시간 동안 바쁘기 때문에, 리스트 스크롤 시 JS 스레드에서도 항상 60 FPS를 달성하기를 원하게 됩니다.

그렇다면 이런 문제에 어떻게 접근해야 할까요? 다행히도, Shopify에서 개발한 매우 훌륭한 대체재(`drop-in replacement`)인 `FlashList`가 있습니다.

```
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;
```

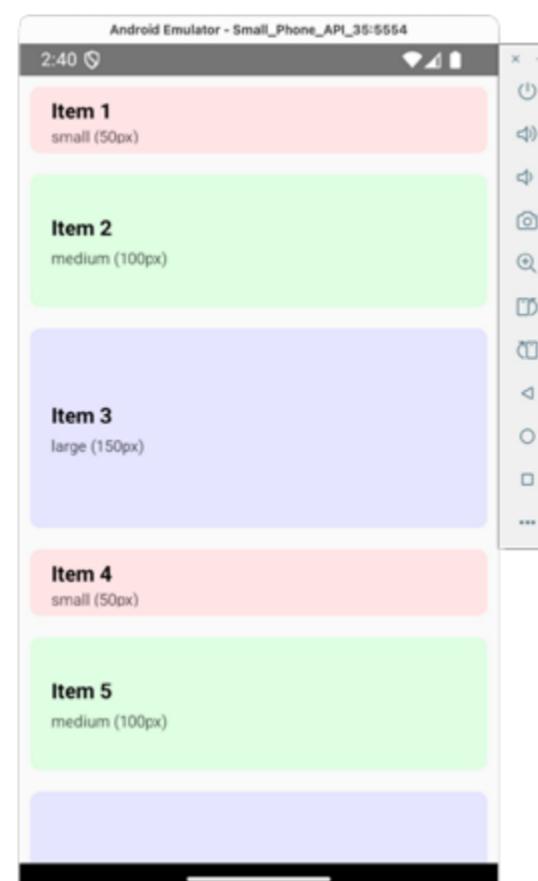
`FlashList` 라이브러리는 `RecyclerListView`를 기반으로 작동하며, 그 재활용 (`recycling`) 기능을 활용합니다. 동시에 복잡한 API, 동적 높이를 가진 셀(cell) 사용, 첫 렌더링 시 레이아웃 불일치와 같은 일반적인 문제점(`pain point`)들을 해결합니다.

`FlashList`는 뷰포트(`viewport`) 밖의 뷰(`view`)들을 재활용하여 다른 항목(item)들을 위해 재사용합니다. 리스트에 서로 다른 유형의 항목들이 있다면, 타입(`type`)에 따라 항목을 재사용하기 위해 재활용 폴(`recycle pool`)을 사용합니다. 리스트를 유지하는 것이 중요합니다.

그리고 아이템(item)들을 부수 효과(side effect) 없이 가능한 한 가볍게 유지하는 것이 중요합니다. 그렇지 않으면 리스트 성능이 저하될 것입니다.

예상 아이템 크기(Estimated item size)

FlatList 작업에서 이미 알고 있는 data 및 renderItem props 외에도, 추가적으로 매우 중요한 prop이 하나 더 있는데, 바로 estimatedItemSize입니다. 이것은 리스트 아이템(item)의 대략적인 크기(approximate size)이며, FlashList가 초기 로드(initial load) 전과 스크롤하는 동안 몇 개의 아이템을 렌더링할지 결정하는 데 사용됩니다.

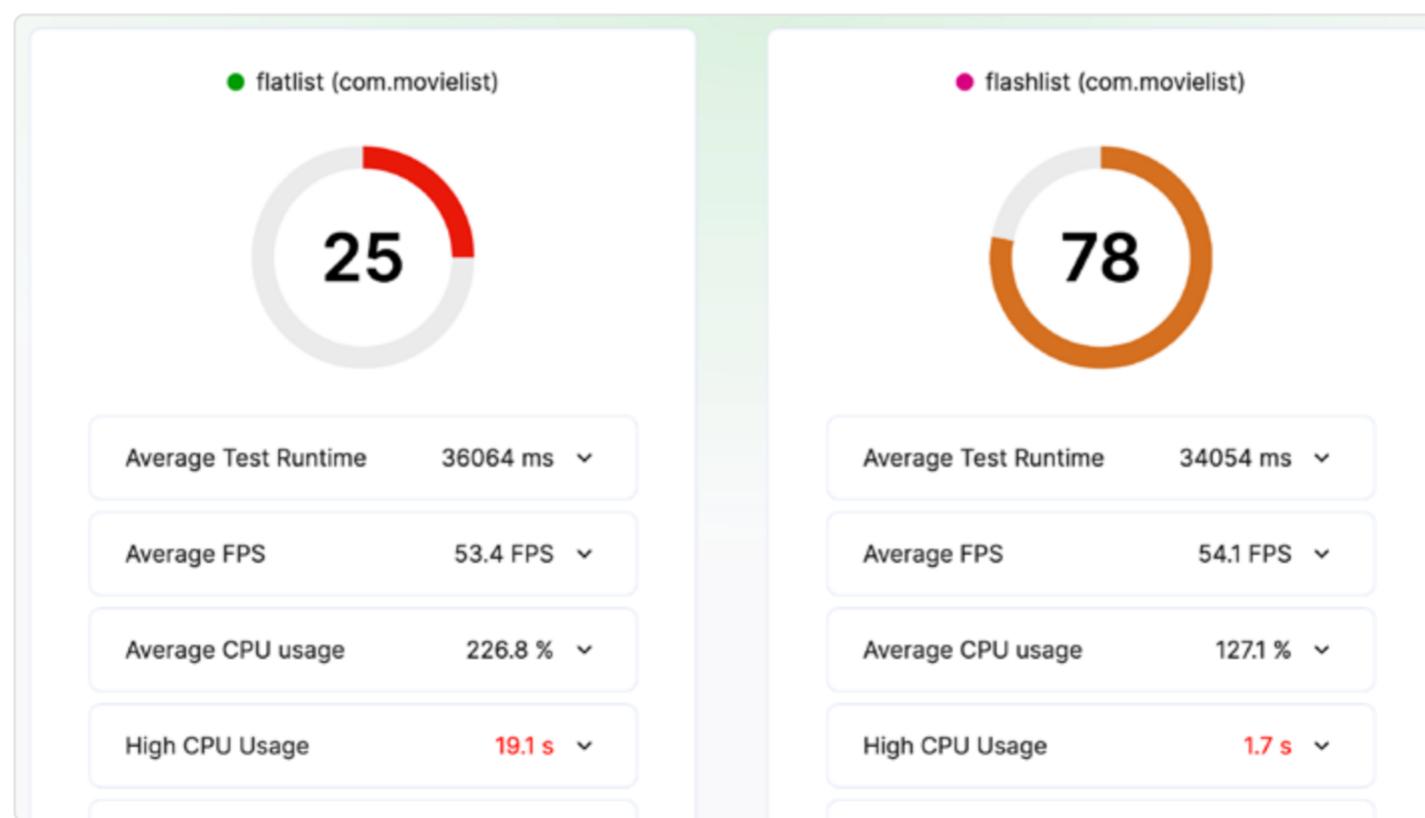


다양한 크기의 아이템들 예시

리스트에 다양한 크기의 아이템들이 포함되어 있다면, 아이템들의 평균 크기를 계산하여 estimatedItemSize 값을 얻을 수 있습니다. (예: 50px, 100px, 150px 크기 아이템들의 평균 = $(50px + 100px + 150px) / 3 = 100px$).

이 값(estimatedItemSize)을 제공하지 않으면, 첫 렌더링 시 리스트(list)에서 경고(warning) 메시지를 통해 (권장) 값을 알려줄 것입니다. 사용자에게 리스트를 배포하기 전에 이 경고를 무시하지 말고 해당 prop을 명시적으로 정의하는 것이 좋습니다.

얼마나 더 빠를까요?



성능 비교: FlatList vs FlashList 예제 실행

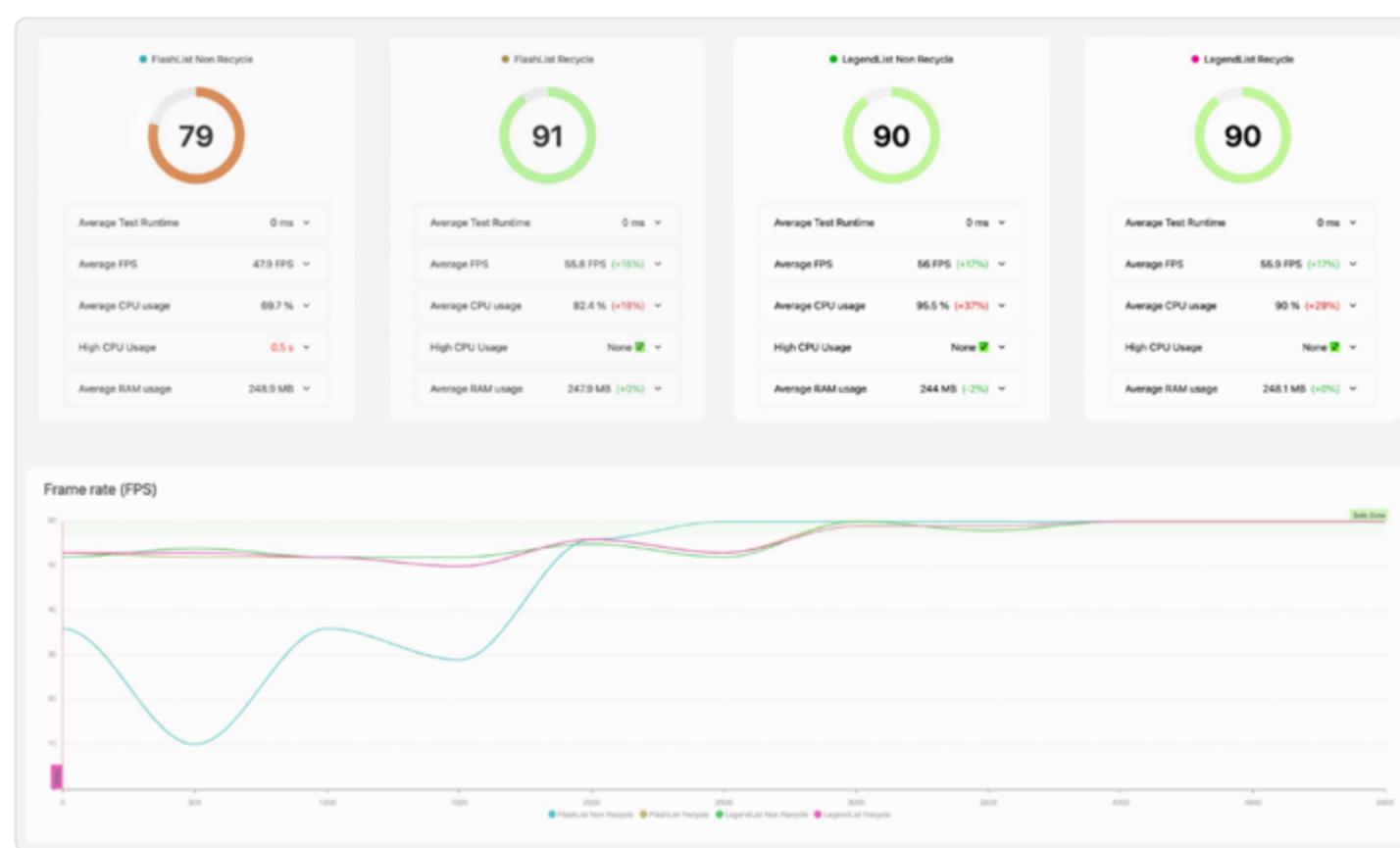
보고서에서 볼 수 있듯이, FlashList는 성능 점수 68/100 대 25/100으로 FlatList보다 훨씬 뛰어난 성능을 보여주며, 더 빠른 런타임(runtime)과 낮은 자원 소모량(resource consumption)을 포함한 모든 지표(metric)에서 우수한 효율성을 입증합니다. 프레임 속도(frame rate) 그래프는 약 56 FPS에서 더 안정적인 성능을 보여주며 FlashList의 우위를 더욱 확실히 뒷받침합니다. 여전히 개선의 여지는 있는 것 같습니다.



또한 FlashList의 콜백(callback) 함수들을 활용하여 성능과 렌더링 시간을 측정할 수도 있습니다. 사용 가능한 헬퍼(helper) 기능들에 대한 더 자세한 정보는 공식 문서의 '[Metrics](#)' 섹션을 참고하세요.

Legend List 주목하기

2025년 초 현재, Jay Meistrich가 FlatList의 새로운 대안인 [Legend List](#)를 집중적으로 개발해 왔습니다. 현재 버전은 1.0-beta이며 FlashList와 매우 유사한 성능을 보여줍니다. 이 라이브러리는 New Architecture의 가능성을 활용하며 전체가 자바스크립트(JavaScript)로 구현되었습니다. 아직 운영 환경(production)에서 사용하기에는 이르지만, React Native 커뮤니티가 분명 주목해야 할 대상입니다.



성능 비교: LegendList vs FlashList

특화된 컴포넌트를 사용하는 것이 항상 가능한 가장 빠른 결과를 보장하지는 않을 수도 있지만, FlatList, FlashList, LegendList의 예시에서 보듯이 이들은 쉽게 교체 가능하여 선택하고 (성능을) 검증할 수 있는 더 많은 옵션을 제공합니다. 그리고 다른 서드파티(third-party) 의존성(dependency)들과 마찬가지로, 새로운 버전이 나오면서 추가 비용 없이 더 최적화된 코드를 얻을 수도 있지만, 아닐 수도 있습니다! 항상 (성능을) 측정하세요.

BEST PRACTICE

아토믹(Atomic) 상태 관리

React 앱은 상태(state), props, 또는 컨텍스트(context) 형태로 받는 입력에 기반하여 매번 자신을 (개념적으로) 처음부터 다시 만듭니다. 모든 React 컴포넌트는 이러한 입력과 부모 컴포넌트가 업데이트될 때 자신을 리렌더링합니다. 이것이 바로 개발자로서 우리가 매우 좋아하는 React의 약속(promise)인데, 이는 변경 사항이 항상 전체 앱과 동기화된다는 예측 가능성과 확실성을 제공하기 때문입니다.

하지만 바로 이 메커니즘이, 신중하게 최적화되지 않으면 컴포넌트 트리(component tree) 아래로 계속 전파되는 불필요한 리렌더링 때문에 종종 성능 문제를 일으킵니다. 적어도 React Compiler를 사용하지 않는 한 그렇습니다.

컴포넌트 트리 상위에서 리렌더링이 발생할수록, 변경 사항이 리프(leaf) 컴포넌트까지 불필요하게 전파될 가능성이 커집니다. 이는 거의 모든 React 앱에서 발견되는 전역 앱 상태(global app state)의 경우에 해당합니다. 최적화되지 않으면, 전역 스토어(global store)의 변경 사항이 해당 변경의 영향을 전혀 받지 않는 컴포넌트들까지 리렌더링하게 만듭니다.

우리는 React의 Context나 Redux와 같은 외부 라이브러리에 크게 의존하는 코드베이스에서 이러한 현상을 종종 관찰합니다.

필터(filter)와 할 일 목록(todos) 상태를 보유하는 App 컴포넌트와, 이 상태들을 사용하지만 메모아이즈(memoize)되지 않은 FilterMenuItem 및 TodoItemList 컴포넌트를 보여주는 간단한 코드 조각을 살펴보겠습니다.

```
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}
                />
            ))}
        </View>
    );
}

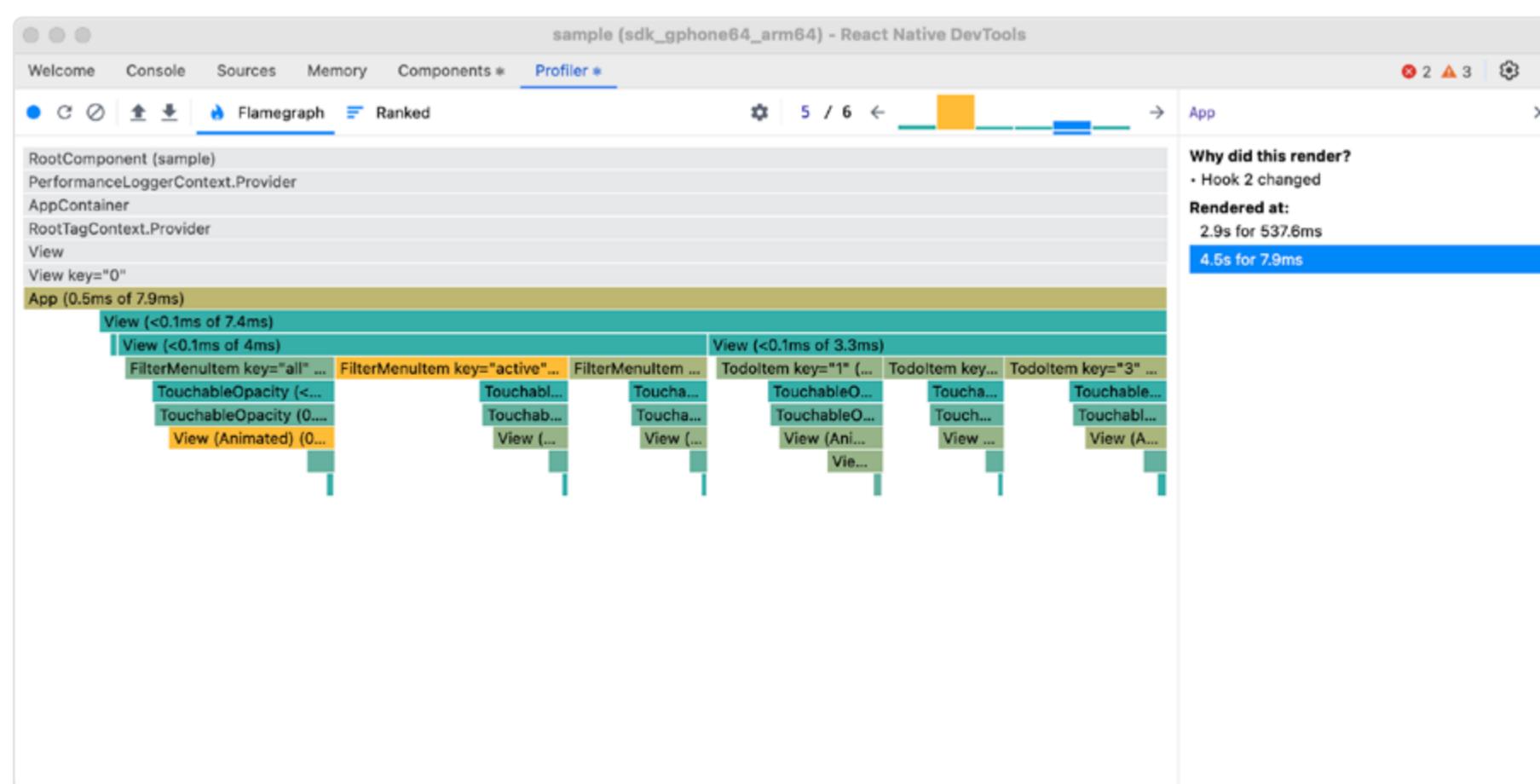
/* 할 일 목록 섹션: 필터 변경 시 전체 리렌더링 문제 */
{filteredTodos.map((todo, index) => (
    <TodoItem key={index} item={todo} onChange={setTodos} />
))
</View>
);
};

export default App;
```

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

filter 또는 todos 상태(state) 변경 결과로, 모든 컴포넌트(App, FilterMenuItem, TodoItem)가 리렌더링됩니다. TodoItem을 업데이트하면 todos 상태에 의존하지 않는 FilterMenuItem까지 리렌더링됩니다.

이는 이상적이지 않지만, **React Native DevTools**에서 컴포넌트를 검사해보면 이러한 리렌더링이 흑(hook) 변경 또는 부모 컴포넌트의 리렌더링 때문에 발생함을 알 수 있습니다.



플레임 그래프: TodoItem 상태 토글 시 전체 UI 트리 리렌더링

불필요한 리렌더링을 피하기 위해, `React.memo`, `useMemo`, `useCallback` 흑(hook)들을 사용하여 수동으로 메모아이즈(memoize)하거나, `React Compiler`를 켜서 자동으로 처리하게 할 수 있습니다. 또는 세 번째 방법으로, `React` 모델에서 벗어나 아토믹(atomic) 또는 시그널 기반(signal-based) 상태 관리 라이브러리를 활용할 수도 있습니다.

아토믹(Atomic) 상태 관리

아토믹 상태(atomic state) 방식에 초점을 맞춰 봅시다. 이는 상태를 '아톰(atom)'이라 불리는 작고 독립적인 단위로 나누는 상향식(bottom-up) 접근법입니다. 크고 중앙화된 스토어(centralized store)를 유지하는 대신, 각 아톰은 `React` 렌더링 모델 외부에서 개별적으로 업데이트하고 구독(subscribe)할 수 있는 최소한의 상태 조각을 나타냅니다. 이를 통해 더 세분화된 제어(granular control)가 가능하며, 목표 지정된(targeted) 리렌더링을 통해 더 나은 성능을 얻을 수 있습니다.

Zustand, Recoil, 또는 Jotai와 같이 아토믹, 상향식 상태 관리를 제공하는 라이브러리는 많습니다. 다음 예시에서는 Jotai를 사용할 것입니다.



Jotai는 일본어로 '상태(状態, jōtai)'를 의미하며, Zustand는 독일어로 '상태(state)'를 의미합니다.

Jotai에서는 상태 조각(piece of state)을 나타내는 아톰(atom)을 정의합니다. 이 아톰에는 원시값(primitive)이나 더 복잡한 데이터 구조의 초기값(initial value)만 지정해주면 됩니다:

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

그런 다음 `useAtom` 흑(hook)을 사용하여 필터(filter) 아톰(atom)의 `getter`(값)와 `setter`(업데이트 함수)에 접근할 수 있습니다:

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

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

혹은 `useSetAtom` 흑으로 `todos` 아톰의 `setter`(업데이트 함수)만 사용할 수도 있습니다:

```
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
));
}

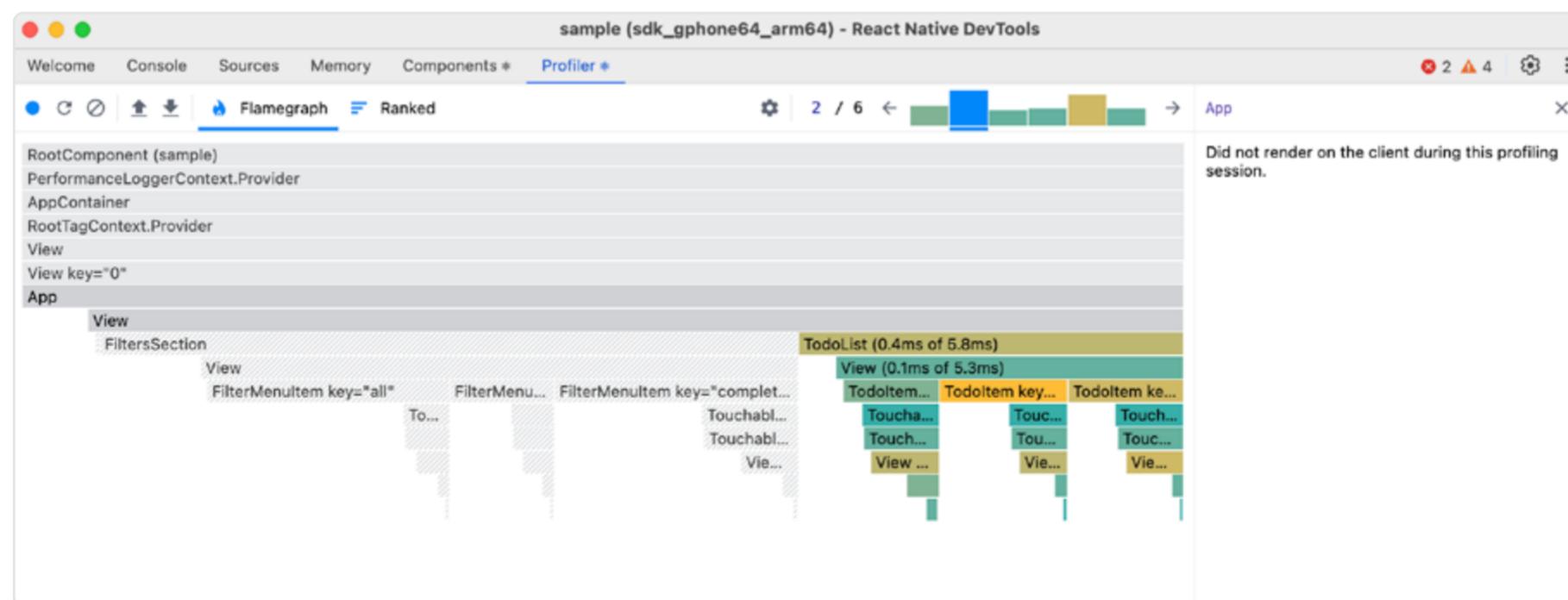
>
{ /* ... */
</TouchableOpacity>
);
};

}
```

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

상태의 atomic 방식 덕분에, memoization과 같은 어떠한 최적화 기법도 사용하지 않고 관련 없는 컴포넌트들의 상태 변경을 막을 수 있습니다.

우리의 간단한 시나리오에서는, filter를 변경하면 해당 filter atom으로부터 값을 읽어오는 모든 컴포넌트가 리렌더링됩니다. 여기에는 (filter 상태를 각각 구독하는) 모든 FilterMenuItemss와 (필터링된 결과에 의존하는) TodoItems를 가진 루트 컴포넌트가 포함됩니다. 하지만 todo의 완료 상태를 토글할 때는, todos atom을 구독하는 컴포넌트만 리렌더링됩니다. FilterMenuItemss는 todos 상태에 의존하지 않기 때문에 변경되지 않고 그대로 있습니다.



아이템 상태를 토글했을 때, 유일하게 영향을 받은 TodoList component의 리렌더링을 나타내는 flame graph

다시 한번 말하지만, React 프로그래밍 모델에서 벗어나는 것이 우리 앱의 전체 성능에 이점을 줄 수 있습니다. 수년간 atomic 상태 라이브러리는 내부적으로 hooks와 useSyncExternalStore 같은 React API를 활용하면서 전체 리렌더링 횟수를 줄이는 데 도움이 되었습니다. 이제 우리가 React Compiler를 사용할 수 있게 되면서, 컴파일러가 우리 대신 힘든 작업을 처리해 줄 수 있을 때 성능 때문에 앱 전체를 새로운 라이브러리로 마이그레이션하는 것은 현명하지 못한 판단일 수 있습니다. 이에 대한 더 자세한 내용은 [React Compiler](#) 챕터에서 읽어볼 수 있습니다.

BEST PRACTICE

React 동시성

앱이 로드되고 사용자 인터랙션에 응답하는 방식에 대한 인식은 때때로 실제 성능보다 더 중요합니다. 특히 많은 앱에서 가장 느린 부분인 원격 서버에서 리소스를 가져올 때 더욱 그렇습니다. 앱을 물리적으로 더 빠르게 실행되도록 만들 수는 없지만, 사용자에게 얼마나 빠르게 느껴지는지를 개선할 수 있습니다. 상대적인 인식에 관한 것이므로 이를 **인지 성능**이라고 부릅니다.

인지 성능을 개선하기 위한 좋은 일반적인 지침은, 작업이 완전히 완료될 때까지 사용자를 기다리게 하는 것보다 빠른 응답과 정기적인 상태 업데이트를 제공하는 것이 일반적으로 더 좋다는 것입니다. 이것이 **동시성 React의 핵심 아이디어**입니다.

React의 렌더링 모델에 대한 근본적인 업데이트로서, **동시성 React**는 React 18에 도입되었고 React Native에는 버전 0.69에 적용되었습니다. 단일 중단 없는 프로세스로 업데이트를 완료하는 대신, 동시성 React는 필요할 때 업데이트를 일시 중지하고 우선순위를 재조정하거나 심지어 중단할 수도 있게 합니다. 이를 통해 사용자 입력과 같은 핵심적인 인터랙션이 자연 없이 최우선으로 처리되도록 보장하여 전반적인 사용자 경험을 개선합니다.

동시성은 이면의 메커니즘이지만, 여기에는 중단 가능한 렌더링, 타임 슬라이싱, 자동 배치를 포함한 몇 가지 강력한 기능을 가능하게 합니다. 이러한 기능은 React Native가 UI 업데이트를 준비하고 관리하는 것을 더 효율적으로 가능하게 하며, 특히 대규모 데이터셋을 렌더링하거나 복잡한 인터랙션을 관리하는 것과 같은 리소스 집약적인 시나리오에서 더욱 그렇습니다.



React Native에서 데이터 페칭을 위한 `useTransition`, `useDeferredValue`, 자동 배치, `Suspense`와 같은 동시성 React 기능을 사용하려면, React Native 0.76부터 새로운 프로젝트의 기본값인 New Architecture로 마이그레이션해야 합니다.

느린 컴포넌트 처리하기

많은 애플리케이션에서 특정 컴포넌트는 렌더링 비용이 많이 들고 최적화하기 어렵습니다. 예를 들어, 복잡한 계산을 수행하거나 서드파티 라이브러리를 통합할 때 그렇습니다. 이러한 무거운 컴포넌트가 UI를 차단하는 대신, `useDeferredValue hook`을 사용하여 우선순위가 높은 작업에 간섭하지 못하도록 방지할 수 있습니다.

`useDeferredValue hook`은 앱의 업데이트에 대한 버퍼처럼 작동합니다. 친구가 어깨 너머로 읽고 있는 동안 메모를 작성하고 있다고 상상해 보세요. 만약 그 친구가 작성 중인 내용에 대해 코멘트하기 위해 당신을 방해한다면, 메모 작성은 잠시 중단해야 할 수 있습니다. 유사하게, `useDeferredValue`는 React가 덜 중요한 업데이트의 렌더링을 '일시 중지'하도록 허용하여 타이핑이나 클릭과 같은 사용자 인터랙션이 부드럽게 유지되도록 보장합니다.

실제로는, 자주 변경되는 값에 의존하는 컴포넌트가 앱 속도를 늦추지 않도록 보장합니다. 대신에, React는 (사용자가 타이핑할 때 디스플레이를 업데이트하는 것과 같이) 주어진 시점에 가장 중요한 UI의 특정 부분을 렌더링하는 것을 우선순위로 둡니다. 백그라운드에서 더 무거운 계산이나 업데이트는 자연시키면서 말이죠.

```
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;
```

위 예시에서, `useDeferredValue hook`은 `CounterNumber`는 반응성을 유지하는 동안 `SlowComponent`에 대한 업데이트가 자연되도록 보장합니다. `ActivityIndicator`는 자연된 값이 즉각적인 값보다 뒤처질 때마다 나타나며, 사용자에게 명확한 시각적 피드백을 제공합니다. 이 패턴은 렌더링 비용이 많이 드는 컴포넌트를 렌더링하거나 시간이 오래 걸리는 계산을 수행할 때 특히 유용합니다.



부모 컴포넌트의 리렌더링으로 인해 발생하는 불필요한 리렌더링을 방지하기 위해, 지연된 값을 전달하는 계산이 많은 컴포넌트들을 `React.memo()`로 감싸야 합니다.

업데이트를 기다리는 동안 이전 값 보여주기

데이터 페칭과 같이 비동기 작업을 다룰 때, 이는 모든 앱에서 발생하는 패턴인데, 작업이 완료되기 를 기다리는 동안 로딩 상태를 보여주는 것이 매우 흔합니다. 하지만 이는 콘텐츠가 사라지고 로딩 인디케이터로 자주 대체되는 매끄럽지 못한 사용자 경험으로 이어질 수 있습니다.

`useDeferredValue hook`은 업데이트가 아직 완료되지 않은 동안 이전 값을 보여줄 수 있게 함으로써 더 정교한 접근 방식을 제공합니다.

```
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>
  );
}
```

초기 렌더링 중에 `query`와 `deferredQuery`의 값이 같습니다. 사용자가 `TextInput` 안에서 타이핑을 시작할 때, `query` 속성은 즉시 업데이트되지만, `deferredQuery`는 업데이트(검색 작업)가 완료될 때까지 이전 값을 유지합니다. 이 경험을 더 잘 표현하기 위해 시각적 피드백을 개선 할 수 있습니다. 이 예시에서 검색 결과가 이전 값이라면 값을 유지하는 속성을 도입하여 말이죠:

```
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>
  );
}
```

업데이트를 기다릴 때, 검색 결과는 투명도가 감소할 것입니다. 업데이트가 완료되면 최신 결과가 표시될 것이고, `deferredQuery`는 `query`와 같아질 것입니다.

중요하지 않은 업데이트를 위해 트랜지션 사용하기

트랜지션은 개발자가 보조적인 상태 변경과 같은 중요하지 않은 UI 업데이트를 낮은 우선순위로 표시함으로써 처리할 수 있도록 허용합니다. 이는 사용자 입력과 같은 긴급한 인터랙션이 먼저 처리되도록 보장합니다. 특정 값의 렌더링을 지연시키는 `useDeferredValue`와 달리, `useTransition`은 전체 업데이트 프로세스를 지연시키며, 상태 변경 관리를 더 부드럽게 보장합니다.

```
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);
  }

  const handleSlowCount = () => {
    setSlowCount((prevSlowCount) => prevSlowCount + 1);
  }
}

export default TransitionScreen;
```

```
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;
```

중요한 업데이트와 중요하지 않은 업데이트를 분리함으로써, UI는 반응성을 유지하고 부드러운 사용자 경험을 제공합니다. `useTransition hook`은 `slowCount` 상태와 같은 긴급하지 않은 업데이트를 처리합니다. 이는 `CounterNumber`의 `count` 상태와 같은 핵심 업데이트가 즉시 처리되도록 보장하면서, 지연된 업데이트는 백그라운드에서 수행되는 동안 말이죠.

useDeferredValue와 useTransition 중 선택 방법

`useDeferredValue`는 단일 값의 렌더링을 지연시키는 데 가장 잘 사용됩니다. 이는 그 값에 의존하는 컴포넌트가 핵심 업데이트를 차단하지 않도록 보장합니다. 예를 들어, 자주 변경되는 입력에 기반한 계산량이 많은 컴포넌트를 렌더링할 때, `useDeferredValue`는 나머지 UI가 반응성을 유지하도록 보장합니다.

반면에, `useTransition`은 전체 업데이트 또는 렌더링 작업을 낮은 우선순위로 표시함으로써 지연시킵니다. 여러 상태나 컴포넌트가 백그라운드에서 업데이트되어야 할 때, (뷰 간에) 트랜지션 할 때와 같은 경우 또는 UI의 더 넓은 섹션을 업데이트할 때 가장 효과적입니다. React Native 앱에서 뷰 트랜지션은 일반적으로 (React Navigation과 같은) 네이티브 내비게이션에 의해 처리됩니다. 네이티브 스택 내비게이터와 함께 말이죠. 하지만 일부 내비게이터는 JS 기반이며 이 최적화 기법의 대상이 될 수 있습니다.

둘 중에서 결정하려면, 업데이트의 범위를 고려하세요. 단일 `prop` 또는 값을 관리하는 경우 `useDeferredValue`가 더 나은 선택일 가능성이 높습니다. 여러 업데이트 또는 더 넓은 UI 변경이 포함된 트랜지션을 관리하는 경우 `useTransition`이 더 적합합니다.

자동 배치

React 18은 자동 배치를 도입했습니다. 이는 React가 더 나은 성능을 위해 여러 상태 업데이트를 단일 리렌더링으로 그룹화하는 프로세스입니다. 이전에는, React는 React 이벤트 핸들러 내부의 업데이트만 배치했습니다. `Promise`, `setTimeout`, 네이티브 이벤트 핸들러 또는 다른 어떤 곳 내부의 업데이트는 React에 의해 기본적으로 배치되지 않았습니다.

```
// Before: React 이벤트만 배치되었습니다.  
setTimeout(() => {  
    setProductQuantity(quantity => quantity + 1);  
    setIsCartOpen(isOpen => !isOpen);  
    // React는 각 상태 업데이트마다 한 번씩, 총 두 번 렌더링될 것입니다 (배치되지 않습니  
다).  
}, 1000);  
  
// After: setTimeout, Promise 등 내부의 업데이트  
setTimeout(() => {  
    setProductQuantity(quantity => quantity + 1);  
    setIsCartOpen(isOpen => !isOpen);  
    // React는 마지막에 한 번만 리렌더링될 것입니다 (배치 덕분에!).  
}, 1000);
```

자동 배치는 컴포넌트가 한 가지 상태만 업데이트되는 '절반만 완료된' 상태가 되는 것을 방지하여, 예상치 못한 시각적 결함을 유발할 수 있는 상황을 막아줍니다.



이 주제에 대해 더 자세히 알아보고 싶으시다면, 공식 React 문서 내부에 있는 [이 페이지](#)를 읽어보시는 것을 강력히 추천합니다.

`useDeferredValue`, `useTransition`, 그리고 자동 배치를 포함한 동시성 React 기능은 렌더링 작업과 상태 업데이트를 지능적으로 관리함으로써 UI를 차단하지 않고 더 나은 인지 성능을 가능하게 합니다. 이러한 강력한 기능은 앱에 점진적으로 도입될 수 있으며, 특히 리소스 집약적인 시나리오와 같이 가장 필요한 곳에서 성능과 반응성을 향상시킬 수 있도록 허용합니다.

BEST PRACTICE

React 컴파일러

이미 알고 계시겠지만, React Native 앱에서 많은 성능 문제는 React 컴포넌트가 필요 이상으로 자주 리렌더링되는 것에서 비롯됩니다. 이러한 과도한 렌더링을 해결하기 위해, 개발자는 다양한 메모이제이션 기법을 사용하거나 특정 사용 사례에 대해 React 렌더링 모델에서 벗어날 수 있습니다(전역 상태 관리와 같은). 하지만 직접 작성하는 메모이제이션의 문제는 코드를 읽고 이해하기 더 어렵게 만든다는 것입니다. 더 나은 방법이 있어야 합니다. 우리 함수와 컴포넌트를 자동으로 메모이제이션해 줄 어떤 스마트한 프로그램 말이죠. 그리고 그런 것이 있습니다 - 바로 React 컴파일러입니다.

React 컴파일러는 React 코어 팀에서 개발한 새로운 도구입니다. React 애플리케이션을 빌드 타임에 자동으로 최적화하도록 설계되었습니다. 이는 컴포넌트 구조를 분석하고 메모이제이션 기법을 적용하여 불필요한 리렌더링을 줄입니다. `React.memo`, `useMemo`, 또는 `useCallback`을 사용한 수동 최적화와 달리, 컴파일러는 이 프로세스를 자동화함으로써 개발자의 추가적인 노력 없이 최적의 성능을 더 쉽게 달성할 수 있도록 만듭니다.



이 가이드를 작성할 시점인 (2025년 1월)에는, React 컴파일러는 아직 베타 상태입니다. Meta와 같은 회사들은 이미 프로덕션 환경에서 사용하고 있지만, 사용할 수 있는지 여부는 [React의 규칙을](#) 코드가 얼마나 잘 따르는지에 달려 있습니다. 만약 사용해 보고 싶으시다면, npm에서 `beta` 태그가 붙은 베타 버전을 설치하거나 `experimental` 태그가 붙은 데일리 빌드를 실험해 볼 수 있습니다.

React 컴파일러를 위한 코드베이스 준비하기

코드베이스에 React 컴파일러를 추가하기 전에, 새로운 도구를 위해 준비하는 것이 좋습니다. 부담 없는 ESLint plugin을 설치함으로써 말이죠. React Compiler ESLint plugin은 유용한 도구입니다. 실시간으로 잠재적인 문제를 감지하고, [React 규칙 위반을](#) 표시하며, 최적화 방해 요소에 대해 경고합니다.

아직 컴파일러를 사용하고 있지 않더라도, 이 플러그인을 활성화하는 것은 코드 품질을 향상시키고 컴파일러를 도입하기로 결정했을 때 더 부드러운 전환을 보장합니다.



React 컴파일러는 React 17+ 애플리케이션 및 라이브러리와 호환됩니다. 하지만, 그 효과성은 React의 모범 사례를 준수하는지에 달려 있습니다. 클래스 컴포넌트, 오래된 패턴, 또는 React의 규칙을 어기는 컴포넌트는 최적화하지 않습니다.

플러그인을 설치하려면, eslint-plugin-react-compiler를 beta 태그에서 개발 종속성에 추가하세요:

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

그런 다음, ESLint가 컴파일러 규칙을 적용하도록 설정하세요:

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

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

eslint.config.js

이제 코드베이스에서 React 규칙의 위반 사항을 수정할 준비가 되었습니다.

컴파일러 실행하기

이제 린터 설정이 모두 끝났으니, 본론으로 들어갈 시간입니다. 바로 컴파일러를 포함하는 Babel plugin입니다:

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

Babel 설정에서 다음과 같이 설정하세요:

```
const ReactCompilerConfig = {
```

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

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

babel.config.js

React Native 0.78보다 낮은 버전인 (React 19가 함께 제공되는) 프로젝트에서 React 컴파일러를 사용하고 싶다면, 한 가지 추가 패키지인 `react-compiler-runtime@beta`를 추가하고 `ReactCompilerConfig`에서 그에 맞게 타겟을 "18"로 설정해야 합니다. 이제 모든 준비가 끝났습니다! 또는 거의 끝났습니다.

아마 컴파일러가 특정 파일에서 실패할 수 있다는 것을 알게 되실 것입니다. 그런 경우를 위해, Babel plugin은 `sources` 설정을 통해 어느 정도 수준의 커스터마이제이션을 허용합니다. 여기서 일부 컴포넌트를 건너뛰거나 심지어 전체 디렉토리도 건너뛸 수 있는 설정이죠:

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

babel.config.js

이런 식으로, 프로젝트에 React 컴파일러를 점진적으로 추가할 수 있습니다. 베타 소프트웨어의 버그로 인해 발생하는 회귀의 위험을 낮추면서 말이죠. `TextInput`의 `onChangeText` 콜백이 있는 짧은 예시와 함께 컴파일러가 우리 소스 코드에 어떻게 영향을 미치는지 봅시다:

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

컴파일러 실행하기 전 소스 코드

```
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;
}
```

컴파일러 실행한 후: 코드는 더 많아지지만, 리렌더링은 줄어듭니다.

React 컴파일러가 우리코드를 어떻게 변환했는지 주목하세요. 이는 `react/compiler-runtime`에서 `c` 함수를 임포트하고 그것을 사용하여 상태에 따라 무엇을 렌더링할지 결정합니다. `c(n)` 함수는 React 19의 `useMemoCache(n)`의 폴리필입니다. 이전 버전의 React에서 동일한 동작을 가능하게 해주는 것이죠. 이는 렌더링 간에 유지되는 배열을 생성하는데, `useRef`와 유사합니다. 만약 `useRef`를 사용하여 구현한다면, 다음과 같을 것입니다:

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

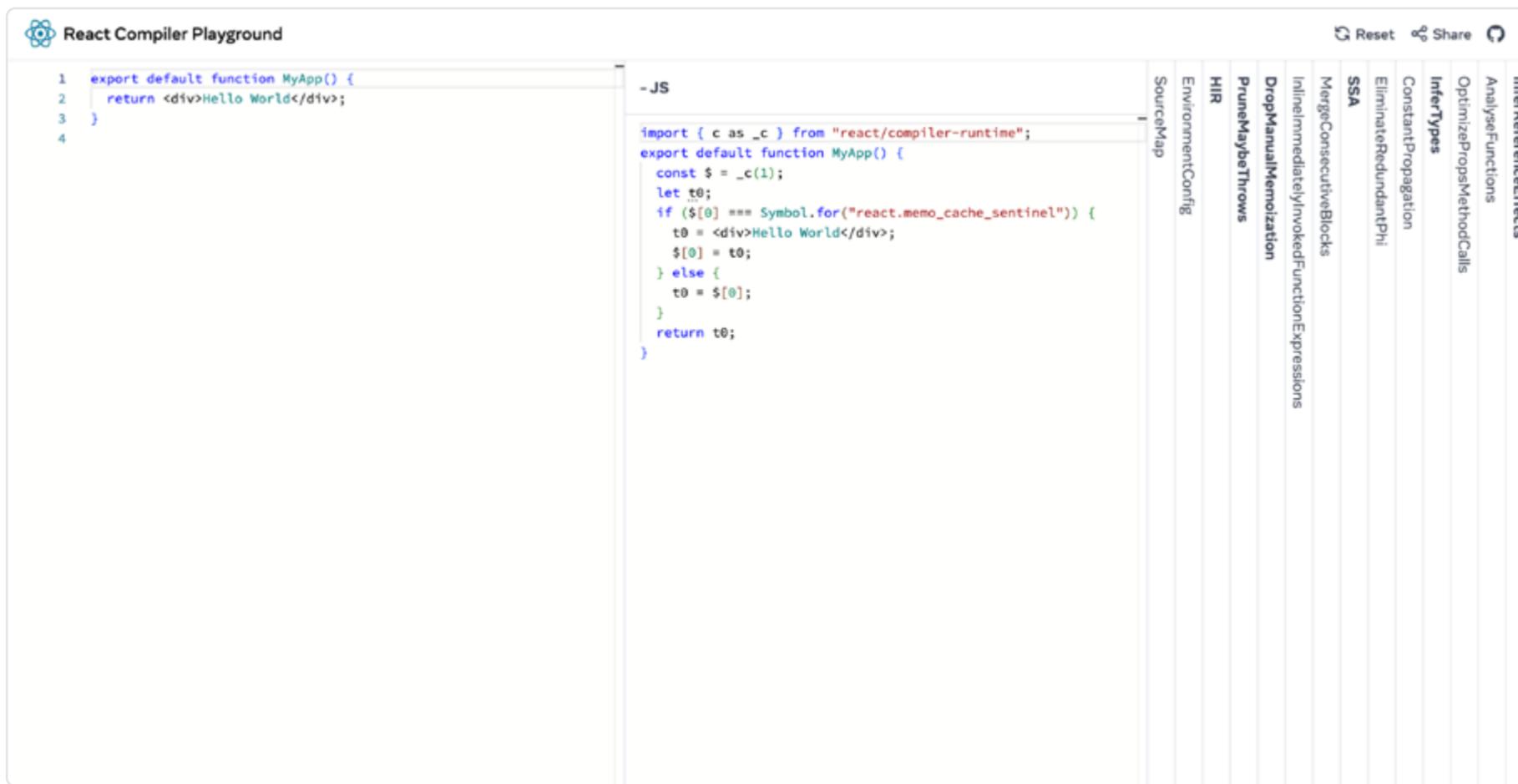
`useMemoCache`는 렌더링 간에 유지되는 `n`개의 슬롯을 가진 배열을 생성합니다.

위 예시에서, `const $ = _c(2)`는 `useMemoCache(2)`처럼 작동하며, 두 개의 슬롯을 가진 배열을 반환합니다. `[0]`은 이전 값을 보유하며, `[1]`에는 마지막으로 렌더링된 `TextInput`이 저장됩니다. `value`가 변경될 때, 새로운 `TextInput`이 `[1]`에 저장됩니다. 만약 `value`가 동일하게 유지된다면, React는 캐시된 컴포넌트를 재사용합니다. 그것을 리렌더링하는 대신 말이죠.

 React 컴파일러는 `React.memo`와 `useMemo`와 마찬가지로 얇은 비교를 사용합니다. 그러므로 `props`로 객체나 배열을 사용할 때 주의하세요. 만약 그들의 참조가 변경된다면, 새로운 값으로 처리될 것입니다.

React Compiler Playground

만약 React 컴파일러가 코드를 어떻게 변환하는지 이해하고 싶다면, [React Compiler Playground](#)는 좋은 도구입니다. 이는 당신이 컴파일러가 컴포넌트를 어떻게 최적화하는지 검사하고, 다른 구조를 테스트하며, 컴파일러 출력을 디버그할 수 있도록 허용합니다. 이 인터랙티브한 환경은 컴파일러가 어떤 변경을 하는지 확인하고 컴포넌트에서 발생하는 예상치 못한 동작을 식별하는데 도움이 됩니다.

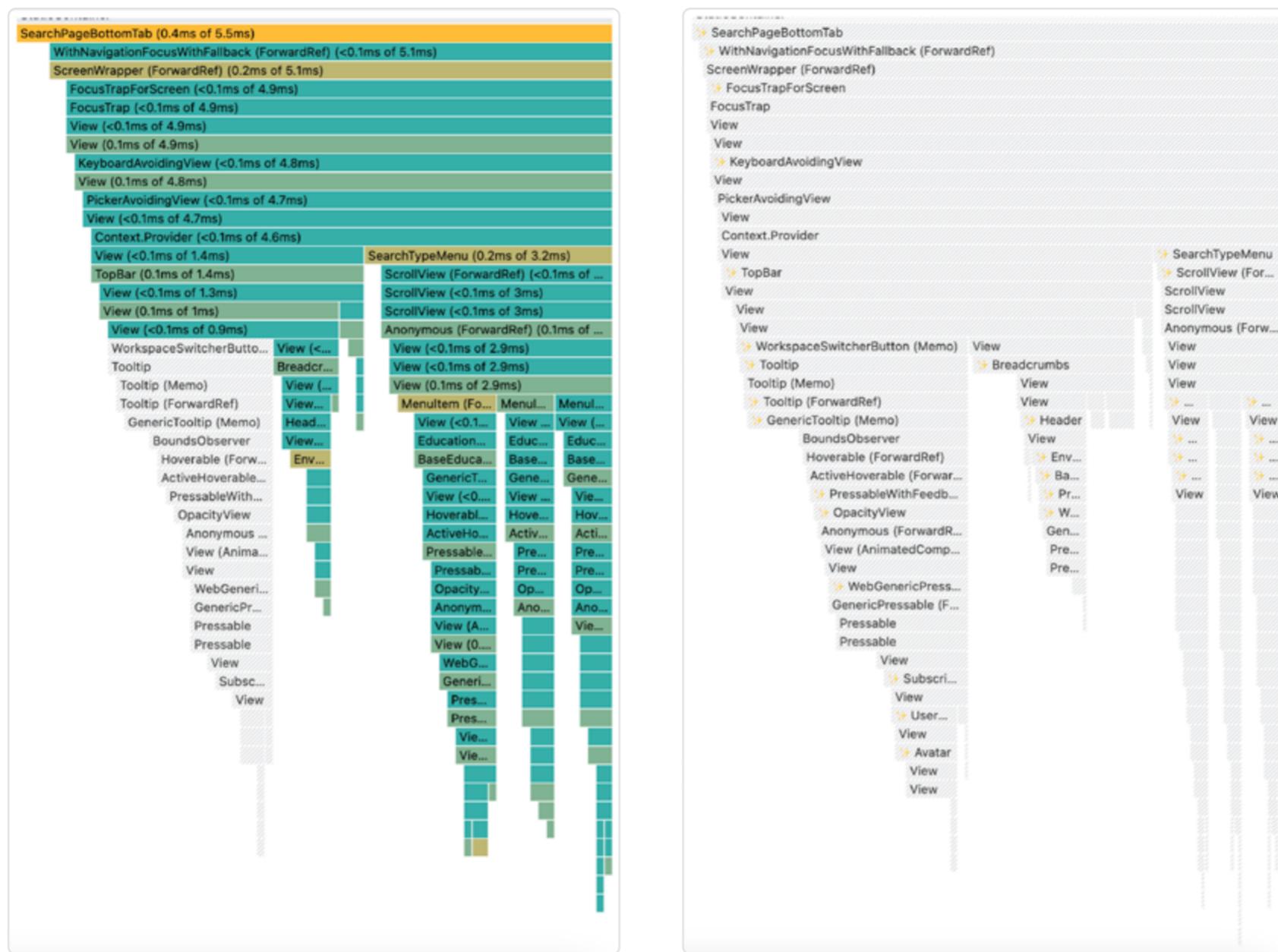


수동 메모이제이션을 제거할 때인가요?

아직은 아닙니다. React 컴파일러가 메모이제이션을 자동화하지만, `React.memo`, `useMemo`, 또는 `useCallback`을 모든 경우에 완전히 대체하지는 않습니다. 만약 코드가 이러한 최적화에 크게 의존한다면, 컴파일러가 안정 버전(stable release)에 도달하고 React 팀의 공식 권장 사항이 수동 메모이제이션이 더 이상 필요 없다는 것을 확인할 때까지 그대로 유지해야 합니다. 어떤 최적화가 불필요한지에 대해서는 린터가 안내해 줄 것으로 예상합니다.

어떤 성능 향상을 기대할 수 있나요?

React 컴파일러는 불필요한 리렌더링을 줄이고 수동 최적화의 필요성을 최소화하는 것을 목표로 합니다. 컴포넌트 계산을 자동으로 메모이제이션함으로써, 연쇄적인 업데이트를 방지하고 특히 깊은 컴포넌트 트리를 가진 대규모 애플리케이션에서 성능을 향상시킵니다. [Expensify app](#)에서 React 컴파일러를 테스트했을 때 성능에서 측정 가능한 향상을 보여주었습니다. 예를 들어, 가장 중요한 지표 중 하나인 "Chat Finder Page Time to Interactive"(실질적으로 TTI)가 4.3% 향상되었습니다. React 컴파일러가 불필요한 리렌더링을 크게 줄여주지만, 수동 메모이제이션 기법으로 이미 최적화된 애플리케이션은 미미한 향상만 볼 수 있습니다.



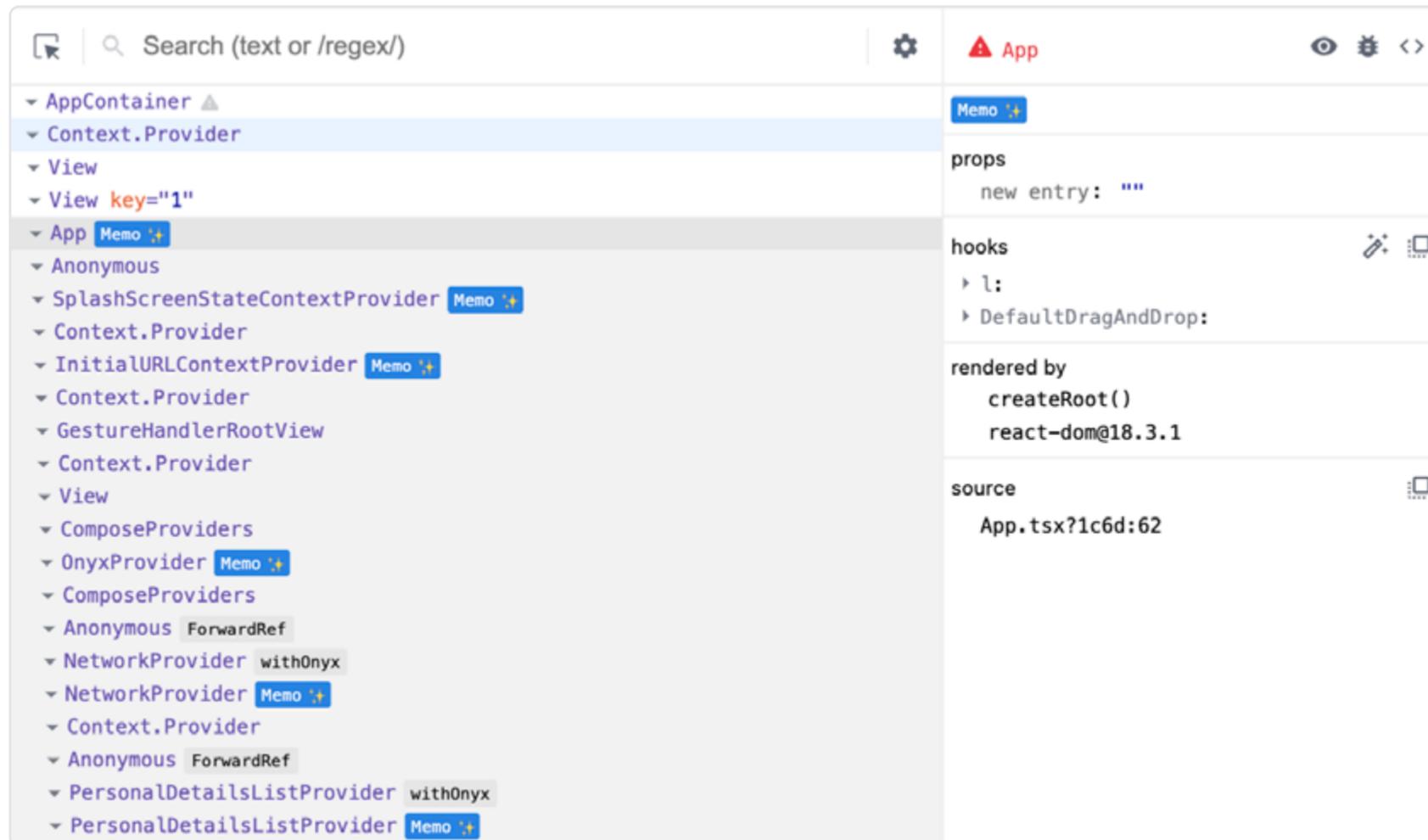
SearchPageBottomTab에서 단일 업데이트가 발생하면, 깊이 중첩된 컴포넌트들 전반에 걸쳐 연쇄적인 리렌더링을 유발하여 렌더링 시간을 증가시킵니다.

React 컴파일러는 불필요한 업데이트를 차단하고, 리렌더링 범위를 줄이며, 실제로 영향을 받은 컴포넌트만 업데이트되도록 보장합니다.

최적화를 확인하는 방법

React DevTools를 열어 React 컴파일러에 의해 어떤 컴포넌트가 최적화되었는지 확인해 보세요.

최적화된 컴포넌트는 'Memo' 배지로 표시됩니다. 이는 컴파일러가 최적화를 적용했음을 보여줍니다.





React 컴파일러는 유니버설 React를 위해 빌드되었지만 주로 웹 환경에서 테스트 되었습니다. React Native에서 컴파일러를 활성화하는 것은 'Memo' 배지가 올바르게 표시되도록 보장하려면 추가적인 단계가 필요할 수 있습니다.

React Native가 자체 버전의 react-devtools를 포함하고 있기 때문에, package.json에서 이를 오버라이드하고 버전 6.0.1 이상으로 업데이트되었는지 확인해야 할 수 있습니다. 이렇게 하지 않으면, 'Memo' 배지가 표시되지 않을 수 있습니다.

React DevTools 백엔드 버전을 확인하려면, DevTools를 열고 설정 아이콘을 클릭하세요.

BEST PRACTICE

고성능 애니메이션 - 프레임 드롭 없이

애니메이션은 매력적인 모바일 경험을 만드는 데 중요한 역할을 합니다. React Native에서 초당 최소 60프레임(FPS)으로 실행되는 부드러운 애니메이션을 보장하는 것은 필수적이지만, 특히 부하가 높은 작업을 처리할 때 때로는 어렵습니다. 애니메이션이 프레임을 떨어뜨리거나 끊겨 보일 때, 사용자 경험에 심각하게 영향을 미치며 앱이 매끄럽지 않게 느껴지게 할 수 있습니다. 이 챕터에서는 애니메이션이 예상대로 실행되도록 보장하기 위해 어떤 도구를 사용할 수 있는지, 그 한계, 그리고 따라야 할 모범 사례를 설명해 드릴 것입니다.

React Native에서 메인 스레드 이해하기

먼저, 메인 스레드가 무엇인지 이해해 봅시다. React Native에서, 메인 스레드(UI 스레드라고도 불리는)는 UI 렌더링과 사용자 인터랙션을 처리하는 주요 스레드입니다. 이는 모든 시각적 업데이트가 발생하는 스레드이며 사용자 인터페이스가 터치 이벤트에 응답하는 곳입니다.



React Native에서, 'main thread'와 'UI thread'라는 용어는 서로 바꿔가며 사용됩니다. 개발자들이 어느 하나에 대해 이야기할 때, 같은 것을 지칭합니다.

이것은 대부분의 React Native 코드가 실행되는 곳인 자바스크립트 스레드와는 다릅니다. 이는 서론에서 더 광범위하게 설명했던 부분입니다.

React Native Reanimated

React Native 앱에 애니메이션을 추가하고 싶을 때, React Native Reanimated를 사용하는 것이 좋습니다. 이는 강력한 라이브러리이며 고성능 애니메이션을 위한 일급 지원을 제공합니다.



React Native Reanimated는 애니메이션을 만드는 데 있어서 업계 표준입니다. 2025년 1월 기준으로, 이는 검증된 솔루션이며 주간 다운로드 수가 거의 100만 건에 달합니다.

Reanimated를 사용하면, **worklets**이라는 개념 덕분에 애니메이션을 UI 스레드에서 직접 쉽게 실행할 수 있습니다. Worklets은 UI 스레드에서 실행될 수 있는 짧게 실행되는 자바스크립트 함수입니다. 이는 또한 코드에서 함수를 실행하는 것과 마찬가지로 자바스크립트 스레드에서도 실행될 수 있습니다.



Reanimated에 의해 개척된 worklets은 **VisionCamera** 또는 **LiveMarkdown**과 같은 다른 커뮤니티 라이브러리로 확산된 강력한 개념입니다. 이러한 라이브러리는 **createWorkletRuntime API**를 사용하여 새로운 JS runtime을 생성할 수 있으며, 이는 worklets를 JS 또는 UI 스레드와는 다른 스레드에서 실행하는 데 사용될 수 있습니다.

대부분의 경우, Reanimated를 사용하여 작업할 때, 코드는 UI 스레드에서 기본적으로 실행됩니다. **useAnimatedStyle hook**의 경우와 같이 말이죠:

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

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

Reanimated는 다양한 헬퍼와 함께 제공됩니다. 그중 특별히 주목하고 싶은 하나는 **runOnUI**입니다. 이는 UI 스레드에서 어떤 코드든 수동으로 실행할 수 있도록 허용합니다. 예를 들어, **useEffect**와 함께 사용하여 컴포넌트 마운트/언마운트 시 애니메이션을 시작하거나, UI 스레드에서만 구현체가 있는 **measure** 및 **scrollTo** 함수와 함께 사용될 수 있습니다. 다음과 같이 사용할 수 있습니다:

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

또 다른 유틸리티 함수인 **runOnJS**도 있습니다. 이는 worklets이 아니며 UI 스레드에서 실행될 수 없는 함수들(예: 외부 라이브러리에서 온 함수들)을 실행하기 위해 당신이 활용할 수 있습니다.

```
// JS 스레드에서 실행되어야 하는 함수
const notifyCompletion = () => {
  console.log('Animation completed!');
  // 애널리틱스 트래킹, 상태 업데이트 등일 수 있습니다.
};
```

```
const animatedStyle = useAnimatedStyle(() => {
  if (progress.value >= 1) {
    // UI 스레드에서 JS 스레드 함수 호출하기
    runOnJS(notifyCompletion)();
  }

  return {
    opacity: progress.value,
    transform: [{ scale: withSpring(progress.value) }],
  };
});
```

코드를 어떤 스레드에서 실행할지에 대한 선택은 여러 요인에 따라 달라집니다:

UI 스레드 (메인 스레드)

- 시각적 애니메이션, 트랜스폼, 및 레이아웃 변경에 가장 적합합니다.
- UI 요소에 대한 직접적인 조작이 필요할 때 사용하세요.

자바스크립트 스레드

- 복잡한 계산, 데이터 처리, 및 상태 관리에 가장 적합합니다.
- React 상태나 props에 접근하거나 비즈니스 로직을 수행할 때 사용하세요.

InteractionManager

React Native는 InteractionManager API를 제공합니다. 이는 잠시간 실행되는 작업이 모든 인터랙션/애니메이션이 완료된 후에 스케줄링되도록 허용합니다. 이는 자바스크립트 애니메이션이 부드럽게 실행되도록 하여, 끊김(jank)과 프레임 드롭을 피하는 데 도움을 줍니다.

사용자가 앱과 인터랙션할 때(예: 버튼 탭 또는 제스처 처리), 자바스크립트 스레드에서 UI 업데이트와 동시에 무거운 작업을 실행하는 것은 성능 문제와 원치 않는 동작으로 이어질 수 있습니다. InteractionManager를 사용하면, 중요하지 않은 작업을 모든 현재 인터랙션이 완료된 후에 실행되도록 스케줄링할 수 있습니다:

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

터치 처리 시스템은 하나 이상의 활성 터치를 '인터랙션'으로 간주하며, 모든 터치가 종료되거나 취소될 때까지 runAfterInteractions() 콜백을 지연시킵니다. 또한 InteractionManager.createInteractionHandle 함수를 사용하여 애니메이션을 등록할 수도 있습니다. 이는 인터랙션 후 수동으로 클리어될 때, 스케줄링된 모든 액션을 실행할 것입니다.

```
const handle = InteractionManager.createInteractionHandle();
// 애니메이션 실행... (runAfterInteractions 작업이 대기열에 추가됩니다)
// 나중에, 애니메이션 완료 시:
InteractionManager.clearInteractionHandle(handle);
// 만약 모든 핸들이 클리어되었다면, 대기열에 추가된 작업이 실행됩니다.
```

React Navigation과 연동하기

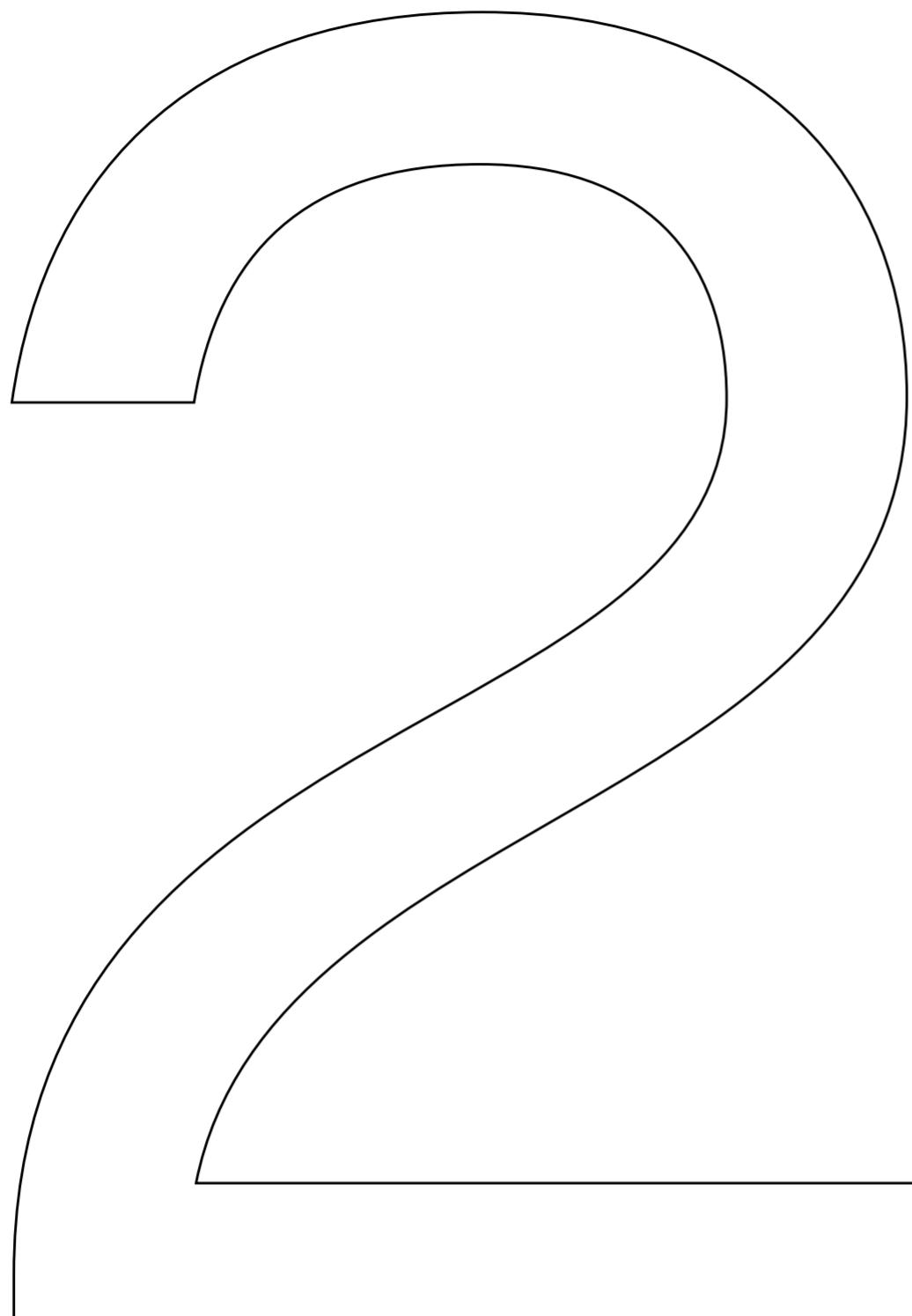
InteractionManager를 사용하고 싶은 일반적인 시나리오는 React Navigation의 `useFocusEffect`와 함께 사용할 때입니다. 이 hook은 화면이 포커스될 때 즉시 효과를 실행합니다. 이는 화면 변경을 위한 애니메이션이 있다면 아직 완료되지 않았을 수 있다는 것을 종종 의미합니다.

React Navigation은 UI 스레드에서 애니메이션을 실행하므로, 많은 경우에 문제가 되지 않습니다. 하지만 만약 효과가 UI를 업데이트하거나 비용이 많이 드는 것을 렌더링한다면, 애니메이션 성능에 영향을 미칠 수 있습니다. 그런 경우에, InteractionManager를 사용하여 애니메이션이나 제스처가 완료될 때까지 우리의 작업을 지연시킬 수 있습니다:

```
useFocusEffect(
  React.useCallback(() => {
    const task = InteractionManager.runAfterInteractions(() => {
      // 비용이 많이 드는 작업 - 결국 UI를 업데이트하는
    });
    return () => task.cancel();
  }, [])
);
```

React Native 전용API인 InteractionManager의 대안으로, 당신은 또한 `startTransition` hook과 같은 React의 최신 추가 기능을 활용하여 중요하지 않은 업데이트를 나중에 실행되도록 이동시켜 UI 업데이트를 차단하지 않도록 할 수 있습니다. 이에 대해 더 자세한 내용은 '[Using transitions for non-critical updates](#)' 챕터에서 읽어볼 수 있습니다.

PART



NATIVE

**iOS, Android, 그리고 C++에서 React Native의
네이티브 측을 최적화하여 FPS를 향상시키기 위한
가이드 및 기법**

서론

1부에서는 Android 또는 iOS에서의 React Native 애플리케이션을 위한 간소화된 앱 시작 모델을 살펴보았습니다. 또한 성능 문제의 최대 80%가 JavaScript 스레드에서 비롯된다는 것을 알게 되었습니다. 이는 그런 문제의 최소 20%가 네이티브 측과 관련이 있다는 결론으로 이어집니다. 이는 개선 할 여지가 상당함을 의미합니다. 따라서 2부에서는 타입라인의 네이티브 측에 초점을 맞출 것입니다. 먼저 앱 시작에 영향을 미치는 가장 중요한 부분부터 시작할 것입니다. 바로 JS 스레드가 초기화되기 전에 실행되는 코드입니다. 그런 다음, 앱이 시작된 후에 더 나은 성능을 달성하는 데 도움이 될 수 있는 런타임 최적화에 대해 다룰 것입니다.

프로세스 (사전) 초기화

앱이 화면에 UI를 표시하도록 하는 것은 그리 간단하지 않습니다. 먼저, 운영 체제(이 예시를 위해 Android에 초점을 맞출 것입니다)가 앱을 부트스트랩할 프로세스를 초기화해야 합니다. 사실, 다양한 메커니즘을 통해 OS는 그러한 초기화를 완전히 또는 부분적으로 수행하기로 결정할 수 있습니다. 심지어 사용자가 앱을 열기 전에도 말이죠.



이 기법은 Android에서 월 또는 핫 스타트업이라고 불립니다. 그리고 이에 대해 더 자세한 내용은 [Android 개발자 포털](#)에서 읽어볼 수 있습니다. iOS는 [prewarming](#)이라고 불리는 유사한 최적화 기법을 지원합니다.

이러한 메커니즘을 이해하는 것은 앱 시작을 안정적으로 측정할 때 특히 중요합니다. 여기서 우리는 항상 콜드 스타트업에 초점을 맞춰야 합니다. 초기화 중에, 운영 체제는 또한 앱의 인-메모리 데이터를 위해 안전한 샌드박스를 생성하고 이에 대해 다른 시스템 서비스에 알립니다.

앱 시작 및 재시작

일단 프로세스가 초기화되면, OS는 Application.onCreate()를 사용하여 애플리케이션 초기화 단계로 진입하고 Activity 및 그 Activity.onCreate() 메서드를 호출합니다. 이것이 네이티브 엔지니어가 코드를 초기화할 수 있는 지점이며 마침내 React Native 여정이 시작될 수 있는 곳입니다. OS는 그런 다음 Activity.onStart()와 같은 다른 앱 라이프사이클 메서드를 호출하고 UI 렌더링 단계로 진입하여, UI 스레드가 레이아웃 및 드로잉 작업을 처리하고, 뷰를 측정하며, 화면에 표시할 것입니다.

React Native의 스레딩 모델

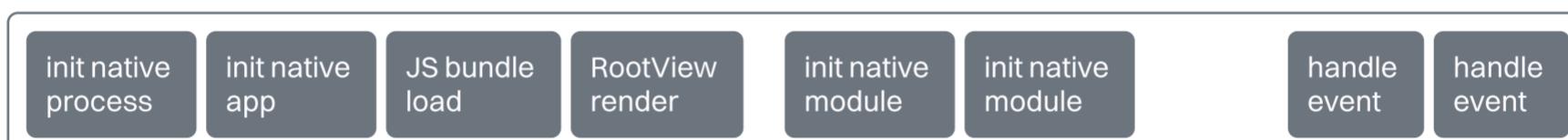
잠깐, UI 스레드가 뭐죠? 만약 자바스크립트 배경 지식이 있다면, 스레드라는 주제와 스레딩 모델에 위축될 수 있습니다. 결국, JS는 단일 스레드 언어이며, 대신 동시성에 의존하고 스레드 언어가 가져오는 모든 함정(및 이점)을 피합니다. 스레드란 프로세스 내에서 실행의 가장 작은 단위입니다. 이는 단일 프로세스 내에서 작업이 동시에 실행되도록 허용하는 독립적인 실행 경로이며, 메모리와 같은 프로세스의 리소스를 공유합니다.

모바일 앱 개발에서, 스레드는 작업을 동시에 수행하기 위해 사용되며, 앱이 반응성을 유지하도록 허용합니다. 모바일 개발자는 일반적으로 두 종류의 스레드 사이에서 작업합니다. UI 스레드(주로 UI 렌더링에 사용되는)와 백그라운드 스레드(메인 스레드 차단을 방지하고 더 무거운 작업 및 앱의 비즈니스 로직에 적합한)입니다.

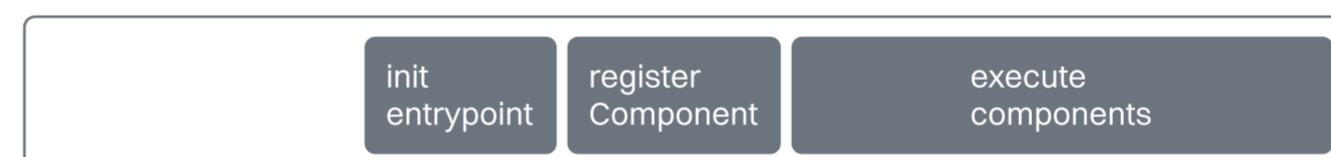
웹 개발자는 단일 스레드에서 UI 렌더링과 비즈니스 로직 처리를 둘 다 수행하는 것에 익숙합니다. 그리고 CSS 하드웨어 가속 또는 `async` 함수, Promises, Timeouts와 같은 동시성 메커니즘을 활용합니다. 이러한 API는 웹 브라우저 또는 Node.js와 같은 서버 런타임 등 JS 환경이 제공하는 이벤트 루프를 활용합니다. 그러한 이벤트 루프도 React Native 앱에도 존재하며, 크로스 플랫폼 C++ 렌더러인 Fabric에 있으며 Hermes JS engine에 의해 접근됩니다.

참고로, React Native가 JS 및 네이티브 세계 둘 다의 최고를 하나의 선언적 모델로 가져오는 방식을 높이 평가할 만합니다. UI 및 백그라운드 스레딩은 대부분 추상화되어 있어서, React Native 엔지니어는 앱 작성에 대한 웹 개발자의 멘탈 모델을 모바일로 가져올 수 있습니다. React의 네이티브 렌더러가 가장 적합한 스레드에서 적절한 코드를 실행하는 것을 처리하면서 말이죠.

UI thread



JS thread



Native modules thread



New Architecture에서의 React Native의 더 포괄적인 스레딩 모델

React Native 초기화

React Native 부분으로 돌아가 봅시다. 이제 네이티브 앱이 초기화되었으니, 네이티브 초기화 중에 로드된 `com.facebook.react:react-native` 라이브러리와 함께 제공되는 native Kotlin code가 이제 그 마법을 수행할 수 있습니다. 이 코드는 모바일 React Native 앱을 위해 생성되고 최적화된 기본 JS engine인 Hermes를 초기화할 것입니다. 그것이 완료되면, 사용 가능한 JS bundle을 메모리로 로드할 것입니다(일반적으로 Hermes Bytecode (HBC) 형식으로 제공되는). 이 바이트코드는 Hermes가 비용이 많이 드는 해석 단계를 피하도록 허용하며, 그리고 HBC file이 메모리로 로드되면 JS 스레드(`mqt_v_js`)에서 직접 실행될 수 있습니다.

어떤 JavaScript 코드든 실행하기 전에, JS 스레드는 즉시 실행을 위해 명시적으로 표시된 네이티브 모듈(C++ TurboModules과 같은, 또는 New Architecture의 Interop Layer를 사용하는 TurboModules)을 처리할 것입니다.

New Architecture의 Interop Layer를 사용하는 TurboModules)을 처리할 것입니다. 그것이 완료되면, JS 코드가 실행됩니다. 전용 스레드(mqt_v_native)에서 다른 TurboModules를 지연 초기화하고 렌더링 및 이벤트 처리를 UI 스레드로 스케줄링하며, 여기서 그것들이 React Native의 크로스 플랫폼 C++ 렌더러인 Fabric에 의해 처리됩니다.

작동 모습 살펴보기

이러한 앱 시작 단계를 직접 검사할 수 있다면 좋지 않을까요? 다음 챕터에서는 Android 및 iOS 플랫폼 모두를 위한 전용 플랫폼 프로파일러를 사용하여 그렇게 할 것입니다. 이러한 학습을 통해, 앱 초기화 단계를 직접 검사할 수 있을 것이며 내부적으로 어떻게 작동하는지에 대한 더 나은 개요를 얻을 수 있을 것입니다. 주의하세요. 깊이 파고드는 것은 꽤 중독성이 있습니다!

GUIDE

플랫폼 차이점 이해하기

JavaScript 개발자로서, 당신은 아마 인기 있는 IDE 또는 에디터 중 일부에 익숙하고 프로젝트에서 작업하는 동안 가장 좋아하는 것을 고수할 것입니다. 하지만, React Native로 작업하기 시작할 때, 수많은 새로운 IDE 및 도구에 의해 압도당하는 느낌을 받을 수 있습니다 (설정 가이드에서 제시되는). 이는 축복이기도 하지만 React Native가 네이티브 프레임워크이기 때문에 발생하는 부작용이기도 합니다. 그 때문에, 가끔은 Android Studio와 Xcode 둘 다로 작업하게 될 것입니다.

이러한 도구에 익숙해지는 것은 이 가이드를 더 잘 탐색하는 데 도움이 될 것이며 React Native로 더 생산적으로 작업할 수 있게 해 줄 것입니다. 이 가이드에서는 세 가지 플랫폼 전반의 공통점, 주요 차이점, 그리고 워크플로우를 향상시키기 위한 생산성 팁을 강조할 것입니다. 이 정보는 이 가이드의 다른 챕터를 탐색하는 데 도움이 될 것입니다. 특히 우리가 Android Studio, Xcode, 또는 다른 빌드 단계를 언급할 때 말이죠.



만약 [Expo](#)를 사용하여 애플리케이션을 빌드하고 있다면, 당신의 IDE comfort zone을 너무 자주 벗어날 필요가 없을 수도 있습니다. Expo는 Dev Clients, Config Plugins 등과 같은 기능들을 사용하여 네이티브 부분들을 잘 추상화합니다. 이 가이드의 내용은 주로 React Native Community CLI에 의존하는 프로젝트에 관련 있습니다.

필수 사항

계속 진행하기 전에, Android Studio 및/또는 Xcode 환경을 준비해 두세요.

만약 [React Native 웹사이트](#)의 설정 가이드를 거쳤다면, 당신은 이미 당신의 머신에 그것들이 설치되어 있을 것입니다. 만약 아직 이것을 하지 않았다면, 지금이 하기에 좋은 시기입니다. React Native 문서는 심층적인 설명을 제공하므로, 우리는 이 주제를 이 가이드에서 건너뛸 것입니다.



모든 플랫폼(Web, iOS, Android)이 다르긴 하지만, 몇 가지 유사점을 공유합니다. 이것을 깨닫는 것은 React Native 개발자로서 당신의 일을 조금 더 쉽게 만들어 줄 것입니다.

IDEs

무엇보다 먼저, 앱을 빌드할 때, 코드를 작성하고 편집하기 위해 최소한 에디터가 필요합니다. JavaScript의 경우, VSCode, WebStorm, Neovim 등 몇 가지만 들자면, 일반적으로 가장 익숙한 것을 사용합니다. 이러한 도구들은 비교적 제한이 없으므로, TypeScript, ESLint, Prettier, 그리고 React와 함께 작동하도록 환경을 설정하는 것이 모범 사례입니다.

Android 및 iOS의 경우, 각각 Google 및 Apple에서 제공하는 IDE로 작업하는 것이 일반적으로 권장됩니다. VSCode 내에서 Swift 또는 Kotlin 코드를 작성할 수 있지만, 앞서 언급된 IDE는 전반적으로 더 예측 가능하고 안정적인 설정을 제공합니다.

해당 네이티브 IDE가 기본적으로 제공하는 기능은 다음과 같습니다.

- **View Hierarchy Debugger** UI 레이아웃을 시각적으로 검사하고 디버그하며, 요소 위치 지정, 제약 조건 및 렌더링 문제를 분석할 수 있도록 허용합니다.
- **Instrumentation & Performance Profiling** Android Studio의 Profiler 및 Xcode의 Instruments와 같은 CPU, 메모리, 배터리 사용량 및 UI 반응성 모니터링을 위한 내장 도구.
- **고급 디버깅 도구** Xcode의 LLDB 및 GDB 지원, Android Studio의 Logcat 및 Debugger, 스택 트레이스, 충돌 보고서 및 메모리 누수에 대한 더 깊은 통찰력.
- **원활한 Gradle (Android) & 빌드 시스템 (Xcode) 통합** 추가 설정 없이 종속성 관리, 빌드 변형, 서명 및 패키징: 코드 서명, 앱 검증 및 앱 스토어로 직접 게시를 위한 통합된 워크플로우.
- **자동화된 테스트 도구** 내장된 테스트 러너와 함께 XCTest (Xcode) 및 Espresso/JUnit (Android Studio)와 같은 네이티브 단위 및 UI 테스트 프레임워크.

우리는 이 부분에서 다른 가이드들과 모범 사례들을 따라가면서 그것들에 의존할 것입니다.

종속성 관리

웹 또는 모바일 애플리케이션으로 작업할 때, 우리는 종종 서드파티 모듈에 의존합니다. React Native에서는 라이브러리가 JavaScript, iOS, 그리고 Android 종속성의 조합일 수 있습니다. 플랫폼 전반에서 종속성이 어떻게 관리되는지 이해하는 것은 네이티브 모듈과 직접 작업할 때 도움이 될 것입니다. React Native는 autolinking으로 이러한 복잡성의 대부분을 추상화합니다. 이는 React Native 전용 모듈을 자동으로 감지하고 연결합니다. 하지만 autolinking은 React Native를 위해 명시적으로 설계된 라이브러리에만 작동합니다. 만약 종속성에 iOS 및 Android를 위한 네이티브 코드가 있지만, React Native 모듈로 구조화되어 있지 않다면, 추가적인 단계가 필요합니다.

JavaScript, iOS, 그리고 Android에서 패키지 관리가 어떻게 작동하는지 살펴봅시다. 이 프로세스를 이해하는 것은 당신이 종속성을 탐색하고 필요할 때 추가적인 네이티브 모듈을 가져오는 데 도움이 될 것입니다.

JavaScript

JavaScript 종속성의 경우, React Native는 주로 npm 레지스트리에 의존합니다. 이는 JavaScript 라이브러리를 위한 중앙 집중식 리포지토리입니다. 이 레지스트리는 JavaScript 코드만 포함하거나 또는 JavaScript 및 네이티브 코드의 조합을 포함할 수 있는 패키지를 포함합니다.

React Native 프로젝트에서 JavaScript 종속성을 설치하는 것은 npm을 사용하여 간단합니다:

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

이 명령어는 npm 레지스트리에서 패키지를 가져오고, node_modules/에 추가하며, package.json을 업데이트합니다.

주요 파일:

- package.json 프로젝트 종속성, 스크립트, 및 메타데이터를 정의합니다.
- node_modules 설치된 모든 종속성을 로컬에 저장합니다.
- package-lock.json 환경 전반에서 버전 일관성을 보장합니다.

 JavaScript 개발자는 Yarn, PNPM, 또는 Bun과 같은 다른 패키지 관리자를 종종 사용합니다.

iOS

iOS의 경우, React Native는 CocoaPods를 사용하여 네이티브 종속성을 관리합니다.

전통적인 iOS 프로젝트와 달리, 여기서 종속성이 CocoaPods (Podspec 리포지토리) 또는 Swift Package Manager에서 가져와지는 것과 다르게, React Native 라이브러리는 일반적으로 네이티브 코드를 npm 레지스트리로 배포합니다. 그리고 CocoaPods는 로컬 참조를 사용하여 해당 코드를 찾아 로드합니다. 하지만, 만약 React Native 라이브러리가 아닌 일반적인 iOS 네이티브 종속성을 추가해야 한다면, CocoaPods에서 가져와야 할 것입니다.

예를 들어, SDWebImage를 당신의 프로젝트에 추가하기 위해, 당신은 다음 내용을 당신의 Podfile에 추가해야 합니다:

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

만약 새로운 React Native 프로젝트를 React Native Community CLI로 생성했다면, 결과는 다음과 같을 수 있습니다:

```
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

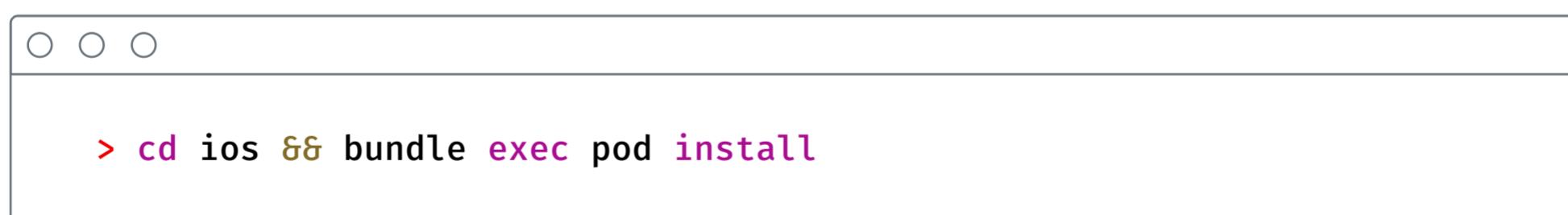
target 'HelloWorld' do
  # Here goes your native dependency
  pod 'SDWebImage', '~> 5.0'

  # Rest of Podfile
end
```

이것은 새 React Native 프로젝트에서 가져온 전형적인 Podfile이며, 가독성을 위해 일부 부분이 제거되었습니다. 이 Podfile의 전체 내용은 [여기](#)에서 확인할 수 있습니다.

-  CocoaPods에서 `~>` 연산자는 pessimistic version constraint를 따르는데, 이는 SemVer (Semantic Versioning)와 유사하지만 약간의 차이가 있습니다.
- `~> 5.0`은 ≥ 5.0 이지만 < 6.0 인 버전을 허용합니다 (패치 및 마이너 업데이트는 허용되지만, 메이저 버전 업데이트는 허용되지 않음).
 - `~> 5.0.1`은 $\geq 5.0.1$ 이지만 < 5.1 인 버전을 허용합니다 (패치 업데이트만 허용되고, 마이너 버전 업데이트는 허용되지 않음).

일단 당신의 종속성을 추가했다면, ios 디렉터리로 이동하여 `pod install` 명령어를 사용하여 pods를 설치하세요:



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

우리가 CocoaPods를 직접 실행하는 것이 아니라, 대신에 `bundle` 명령어를 사용하고 있다는 점에 주목하세요. CocoaPods는 Ruby 프로젝트이므로, 우리는 Bundler(Ruby의 종속성 관리자)를 활용하여 주어진 React Native 프로젝트에 필요한 가장 최신 CocoaPods 버전을 사용할 수 있습니다. 이는 모든 새로운 React Native 프로젝트와 함께 기본적으로 제공되는 최상위 Gemfile에 명시되어 있습니다.

-  pods를 설치한 후, 당신은 `xed .` 명령어를 사용하여 Xcode를 빠르게 실행할 수 있습니다.

주요 파일:

- Podfile iOS 앱에 필요한 종속성을 정의합니다.
- Gemfile 주어진 React Native 앱을 위해 CocoaPods 버전과 필요한 다른 플러그인을 정의합니다.
- Pods CocoaPods를 통해 가져와진 설치된 모든 종속성을 포함하는 폴더입니다. 종속성을 설치할 때 생성되며 일반적으로 버전 컨트롤에 커밋되어서는 안 됩니다 (.gitignore에 기본적으로 목록화되어 있습니다).
- .xcworkspace CocoaPods에 의해 생성된 워크스페이스 파일로, Xcode로 열 때 .xcodeproj 대신 사용되어야 합니다. 이는 CocoaPods가 관리하는 모든 종속성이 Xcode에 올바르게 연결되도록 보장합니다.



Apple의 Swift Package Manager (SPM)는 네이티브 iOS 프로젝트를 위한 대안이지만, React Native는 아직 그것을 지원하지 않습니다. 따라서 CocoaPods가 기본 선택지가 되었습니다.

Android

Android의 경우, React Native는 Gradle을 사용하여 종속성을 관리합니다 (이는 Android의 빌드 자동화 도구입니다). iOS와 유사하게, React Native 전용 모듈은 Maven Central 또는 JCenter와 같은 외부 레지스트리에서 오지 않습니다. 대신에, 그것들은 node_modules/에서 로컬로 참조되며 프로젝트로 직접 컴파일됩니다.

React Native 모듈이 아닌 Android 네이티브 종속성(예: 이미지 로딩을 위한 Glide)의 경우, 당신은 android/app/build.gradle 파일에 수동으로 추가해야 합니다:

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

build.gradle에 새로운 종속성을 추가한 후, 당신은 Gradle 프로젝트를 동기화하여 변경 사항을 가져와 적용해야 합니다.

Gradle은 ~>를 사용하지 않지만, 다른 유연한 버전 관리 옵션을 제공합니다:

- 정확한 버전 'com.github.bumptech.glide:glide:4.12.0'
- 버전 범위(+ 연산자) com.github.bumptech.glide:glide:4.+. 버전 4.x.x 내에서 가장 최신 마이너 또는 패치 업데이트를 사용하지만, 메이저 업데이트는 피합니다.
- 동적 버전 관리 com.github.bumptech.glide:glide:+. 사용 가능한 가장 최신 버전을 사용합니다. 일반적으로 권장되지 않으며, 코드를 예측 불가능하게 손상시킬 수 있습니다.
- 프로덕션 앱의 경우, CocoaPods의 ~>가 iOS 프로젝트에서 breaking changes를 방지하는 것과 마찬가지로, 호환성을 보장하기 위해 항상 정확한 버전을 사용하세요.

React Native 프로젝트에는 Gradle Wrapper(gradlew)가 함께 제공됩니다. 이는 모든 개발자가 동일한 Gradle version을 사용하도록 보장하여 호환성 문제를 피하게 해 줍니다. 당신은 그것을 다음과 같이 실행할 수 있습니다:

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

완료되면, 새로운 네이티브 종속성을 당신의 프로젝트에서 접근할 수 있습니다.

추가적으로 살펴볼 파일들:

- build.gradle (프로젝트 수준) android/build.gradle에 위치하며, Gradle plugin versions과 Maven Central 또는 JitPack과 같은 종속성 리포지토리를 포함한 전역 프로젝트 설정을 설정합니다.
- build.gradle (앱 수준) android/app/build.gradle에 위치하며, 앱별 종속성, 빌드 구성 및 컴파일 옵션을 정의합니다.
- gradle.properties Gradle builds를 최적화하기 위한 구성 플래그 및 설정을 포함합니다(예: Hermes 또는 ProGuard 활성화).
- gradlew 및 gradlew.bat React Native와 함께 제공되는 Gradle Wrapper 스크립트로, 모든 개발 환경 전반에서 일관된 Gradle versions을 보장합니다. .bat 파일은 Windows 시스템에서 사용됩니다.
- .gradle/ 디렉터리 Gradle의 캐시, 컴파일된 스크립트, 및 임시 빌드 파일을 저장합니다. 이는 빌드 프로세스 중에 생성되며 버전 컨트롤에 커밋되어서는 안 됩니다.



예를 들어, 사용 가능한 Gradle 작업을 나열하려면, 당신은 ./gradlew tasks를 실행할 수 있습니다.

프로젝트 빌드하기

React Native로 작업할 때, JavaScript, iOS, 그리고 Android가 코드 실행 및 컴파일을 어떻게 처리하는지 이해하는 것이 필수적입니다. 컴파일된 언어를 사용하는 네이티브 플랫폼과 달리, JavaScript는 전통적인 빌드 시스템을 가지고 있지 않습니다. 하지만 다른 변환 프로세스를 거칩니다.

JavaScript

JavaScript는 해석 언어이며, Swift 또는 Kotlin처럼 별도의 컴파일 단계를 필요로 하지 않습니다. 대신에, JavaScript 코드는 런타임에 엔진에 의해 실행됩니다. 하지만, 여러 ECMAScript 버전과 진화하는 JavaScript 표준 때문에, React Native는 다른 환경 전반에서 호환성을 보장하기 위해 transpilation으로 알려진 "컴파일"의 한 형태를 요구합니다.

React Native에서, 이 프로세스는 Metro에 의해 처리됩니다. Metro는 JavaScript engine(예: Hermes)이 이해할 수 있는 버전으로 변환하는 JavaScript 번들러입니다. 내부적으로, Metro는 Babel에 의존합니다. 이는 React Native에서 사용되는 JavaScript engine의 가장 최신 지원 버전에 맞게 JavaScript 문법을 다시 작성합니다.



React Native는 시간이 지남에 따라 진화하는 JavaScript engine에 의존하기 때문에, 지원되는 정확한 JavaScript version은 Hermes version에 따라 달라집니다. Babel이 사용하는 React Native preset은 JavaScript가 항상 transpiled되어 React Native에서 사용되는 가장 최신 engine capabilities에 맞게 변환되도록 보장합니다.

iOS

JavaScript와 달리, Swift 및 Objective-C는 컴파일 언어입니다. iOS 빌드 시스템은 실행 전에 소스 코드를 머신 코드로 컴파일하여, Apple devices에서 최적화된 성능을 보장합니다.

React Native for iOS는 Xcode's build system에 의존합니다. 이는 Clang (Objective-C용) 및 LLVM (Swift용)을 사용하여 사람이 읽을 수 있는 소스 코드를 최적화된 바이너리(.app bundle과 같은)로 변환합니다.

핵심 단계는 다음과 같습니다:

- **Source Compilation(소스 컴파일):** Swift 및 Objective-C 파일은 Clang/LLVM을 사용하여 머신 코드로 컴파일됩니다.
- **Linking(링킹):** 컴파일된 모든 코드, 시스템 프레임워크, 및 외부 라이브러리(CocoaPods에서 가져온)가 함께 링크됩니다.
- **Signing(서명):** Apple은 앱이 장치에서 실행될 수 있기 전에 유효한 인증서 및 프로비저닝 파일로 서명되도록 요구합니다.
- **Packaging(패키징):** 최종 바이너리(.ipa file)가 생성되며, 앱 및 그 리소스를 포함합니다.

Android

iOS와 유사하게, Android는 컴파일된 빌드 시스템을 사용하지만, Android 앱 빌드, 테스트, 및 패키징 프로세스를 자동화하는 Gradle에 의존합니다.

React Native for Android는 Java/Kotlin 소스 코드를 Android Runtime (ART)에 의한 실행을 위해 최적화된 Dalvik Executable (DEX) 형식으로 컴파일합니다.

핵심 단계는 다음과 같습니다:

- **Source Compilation(소스 컴파일):** Java/Kotlin 파일은 Java/Kotlin 컴파일러를 사용하여 .class 바이트코드 파일로 컴파일됩니다.
- **DEX Conversion(DEX 변환):** 컴파일된 .class 파일은 Android Runtime을 위해 최적화된 .dex 파일로 변환됩니다.

- **Resource Processing(리소스 처리)**: XML 레이아웃, 이미지, 및 기타 에셋이 번들링됩니다.
- **Linking(링킹)**: Gradle에 의해 관리되는 외부 라이브러리 및 종속성이 링크됩니다.
- **Signing(서명)**: Android는 앱이 유효한 키스토어 파일로 서명되도록 요구합니다.
- **Packaging(패키징)**: 최종 .apk 또는 .aab 파일이 장치에 설치하기 위해 생성됩니다.

애플리케이션 실행하기 - 장치에서

테스트는 모바일 앱 개발 워크플로우의 필수적인 부분입니다. 브라우저에서 직접 테스트될 수 있는 웹 애플리케이션과 달리, 모바일 앱은 테스트를 위해 시뮬레이터(Android에서는 에뮬레이터) 또는 물리적 장치를 필요로 합니다. 시뮬레이터는 개발자가 실제 장치 동작을 모방하는 통제된 환경에서 앱을 실행하고 디버그하는 것을 할 수 있도록 허용합니다.

Android

Android 에뮬레이터와 상호작용할 수 있는 다음 방법을 사용하여:

- **Android Studio AVD Manager**: Android 에뮬레이터를 관리하기 위한 내장 도구입니다. 여기서 가상 장치를 생성하고, 구성하며, 실행할 수 있습니다.
- **커맨드 라인 (adb 및 emulator 명령어)**:
 - `emulator -list-avds` 사용 가능한 모든 가상 장치를 나열합니다.
 - `emulator @Pixel_6_API_34` 특정 장치를 실행합니다.
 - `adb devices` 실행 중인 에뮬레이터와 연결된 물리적 장치를 나열합니다.



Expo CLI 또는 React Native Community CLI와 같은 CLI 도구들은 더 나은 개발자 경험을 제공하기 위해 이러한 명령어들을 내부적으로 사용합니다.

iOS

iOS 시뮬레이터와 상호작용할 수 있는 다음 방법:

- **Xcode**: Apple의 내장 도구로, 시뮬레이션된 장치에서 iOS 앱을 실행하기 위한 것입니다. Xcode > Open Developer Tool > Simulator 경로를 통해 실행할 수 있습니다.
- **커맨드 라인 (xcrun simctl 명령어)**:
 - `xcrun simctl list` 사용 가능한 시뮬레이터를 나열합니다.
 - `xcrun simctl boot "iPhone 15"` 특정 시뮬레이터를 부팅합니다.
 - `xcrun simctl shutdown all` 실행 중인 모든 시뮬레이터를 종료합니다.



시뮬레이터 실행 및 관리는 꽤 자주 하게 될 일입니다. 그리고 **Android Studio** 및 **Xcode** 외부에서 그것들을 다루는 데 도움이 되는 여러 앱을 사용할 수 있습니다. 우리가 매일 사용하는 몇 가지는 다음과 같습니다:

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



MiniSim에서 썩은 스크린샷으로, 열려 있는 시뮬레이터(체크 표시와 함께)와 사용 가능한 다른 시뮬레이터도 함께 보여줍니다.

우리는 이 챕터가 성능 최적화와 엄격하게 상관이 없더라도, **React Native** 엔지니어가 작업하는 툴링 생태계에 대한 더 나은 개요를 제공해 줄 것이라고 바랍니다. 여기에 나열된 도구들은 어떤 **React Native** 앱이든 최적화하는 데 더 나아간 작업을 위해 필수적입니다.

GUIDE

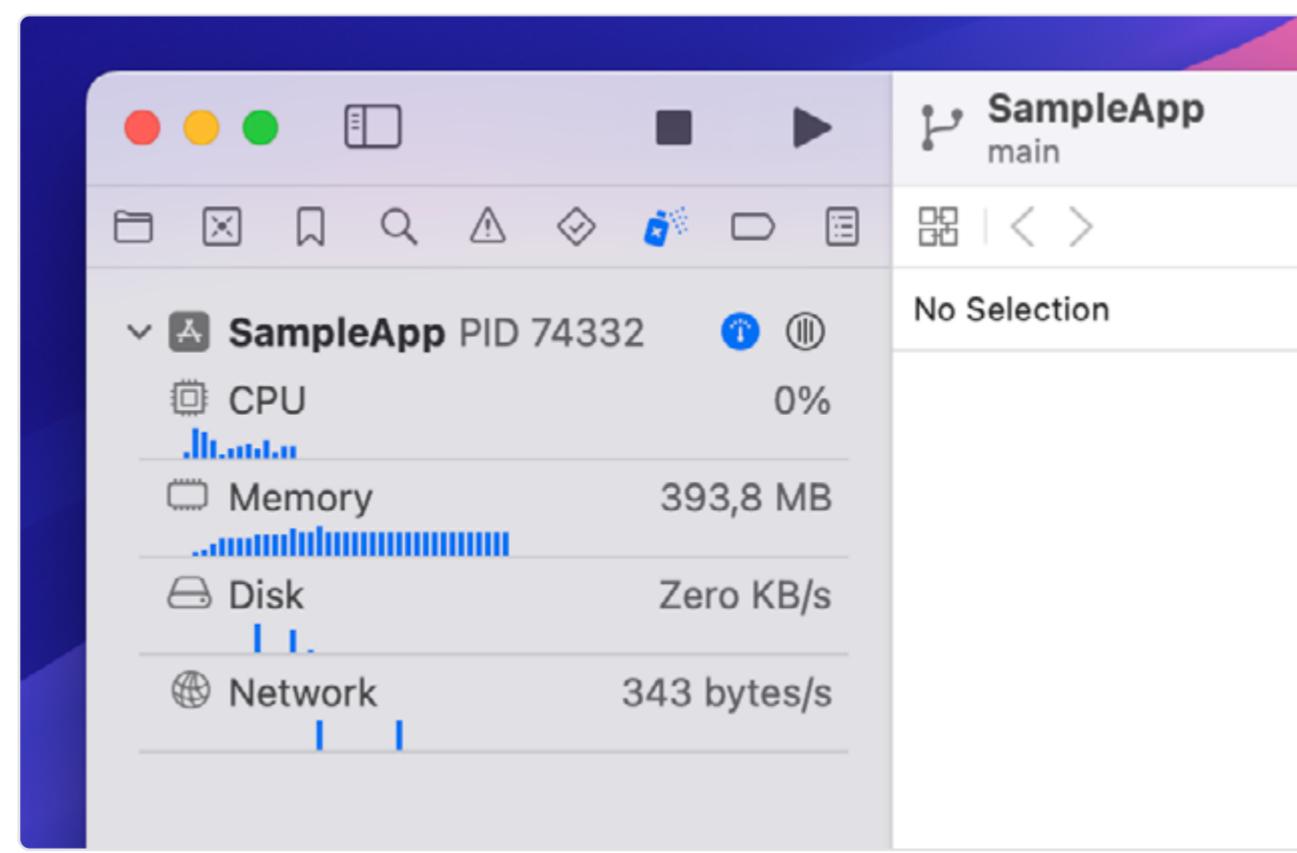
React Native의 네이티브 부분을 프로파일링하는 방법

프로파일링은 '[How to Profile JS and React Code](#)' 챕터에서 다루었던 JavaScript 코드뿐만 아니라, iOS 또는 Android와 같은 다른 언어 및 플랫폼에도 필수적입니다. 병목 현상은 양쪽 모두에서 발생하며, React Native DevTools가 충분하지 않을 때에는 Android Studio 또는 Xcode와 같은 플랫폼의 네이티브 툴링을 사용해야 합니다.

CPU, 메모리, 및 네트워크 프로파일링 외에도, 배터리 사용량은 모바일 사용자에게 마찬가지로 중요합니다. 이는 화면 밝기를 제외한다면 일반적으로 CPU 오버헤드의 대리 지표입니다.

iOS

Xcode는 CPU, 메모리, 디스크, 및 네트워크 부하를 한눈에 검사하기 위한 Debug Navigator view를 제공합니다. 이는 앱이 실행 중이고 Xcode Debugger에 연결되어 있을 때 Xcode의 사이드 패널에 있는 "Debug Navigator" 아이콘 아래 위치합니다:



Xcode 디버그 네비게이터

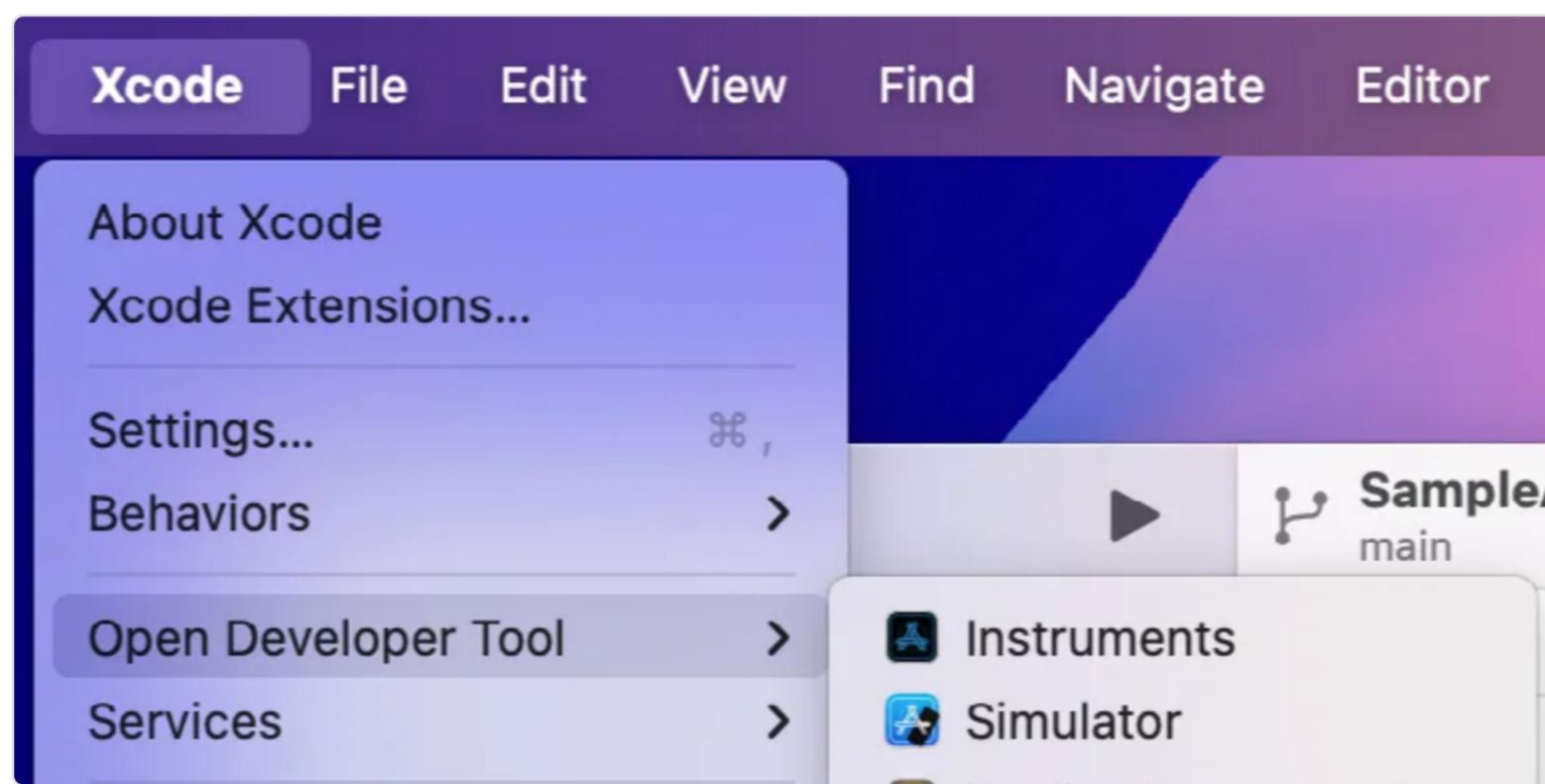
CPU 모니터는 시간 경과에 따른 작업량을 측정합니다. 퍼센트 값은 단일 코어에 대한 사용량을 보여주므로, 특히 React Native 앱에서 100%를 초과할 수 있습니다. Memory 모니터는 애플리케이션의 메모리 사용량을 관찰합니다. 모든 iOS 장치는 영구 저장을 위해 SSD를 사용하며, RAM과 비교하여 이 데이터에 접근하는 것은 더 느립니다. Disk 모니터는 앱의 디스크 읽기 및 쓰기 성능을 파악합니다. Network 모니터는 당신의 iOS 앱의 TCP/IP 및 UDP/IP 연결을 분석합니다.

각각을 탭하여 더 많은 정보를 확인할 수 있습니다. 이는 또한 기본적으로 표시되지 않지만 UI를 검사하는 데 도움이 될 수 있는 추가 모니터를 제공하는데, 바로 View Hierarchy입니다. 이는 우리가 'Use View Flattening' 챕터에서 다루는 것입니다.

Instruments

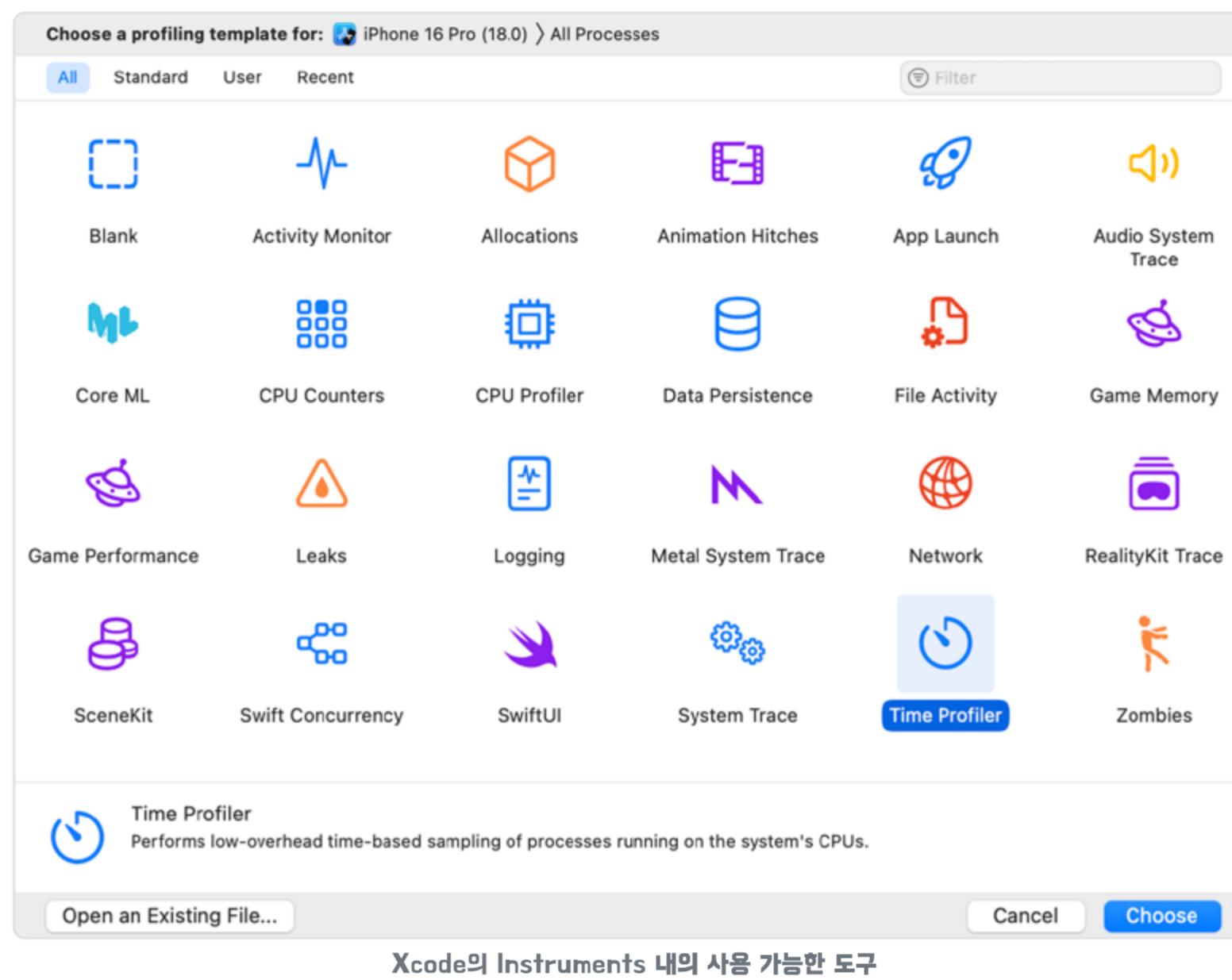
Xcode는 사전 패키지된 디버깅 및 프로파일링 도구인 Instruments와 함께 제공됩니다. 이는 검사 도구 모음이며, 각각 다른 목적에 사용됩니다. 당신은 templates 목록에서 선택하며, 성능 향상, 배터리 수명, 또는 메모리 문제 해결 등 당신의 목표에 따라 그중 어떤 것인든 선택할 수 있습니다. CPU 프로파일링을 위해, 우리는 Time Profiler를 사용할 것입니다. 그것을 자세히 살펴봅시다.

Xcode가 열려 있는 상태에서, Xcode > Open Developer Tool > Instruments 메뉴에서 Instruments로 이동하세요.

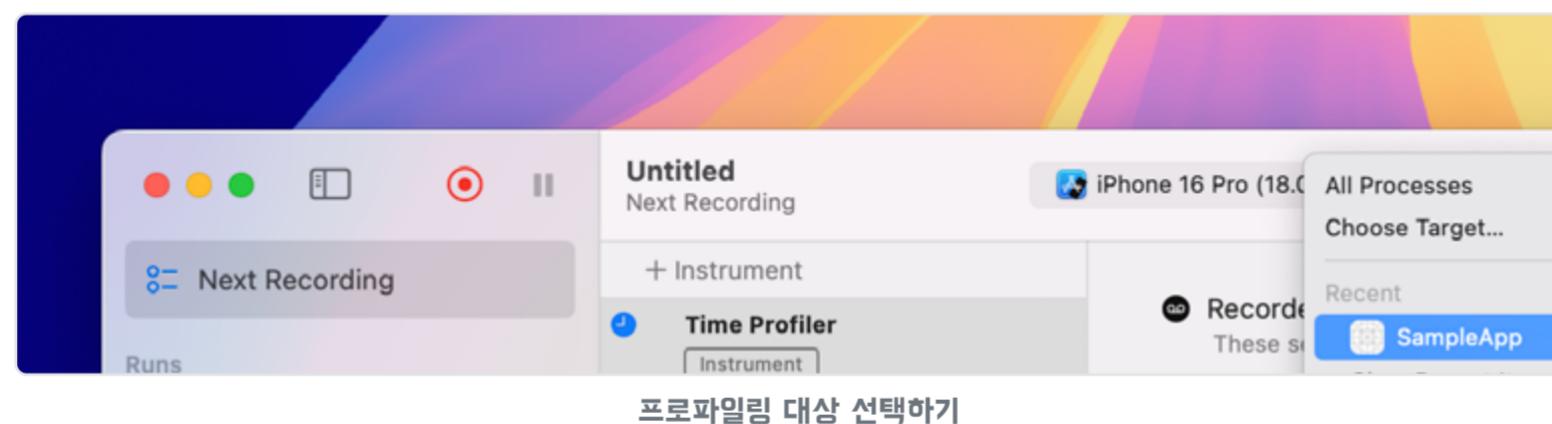


Xcode > 개발자 도구 열기 > Instruments 메뉴

아래로 스크롤하여 Time Profiler 도구를 찾으세요. 이는 CPU Profiler를 포함합니다.



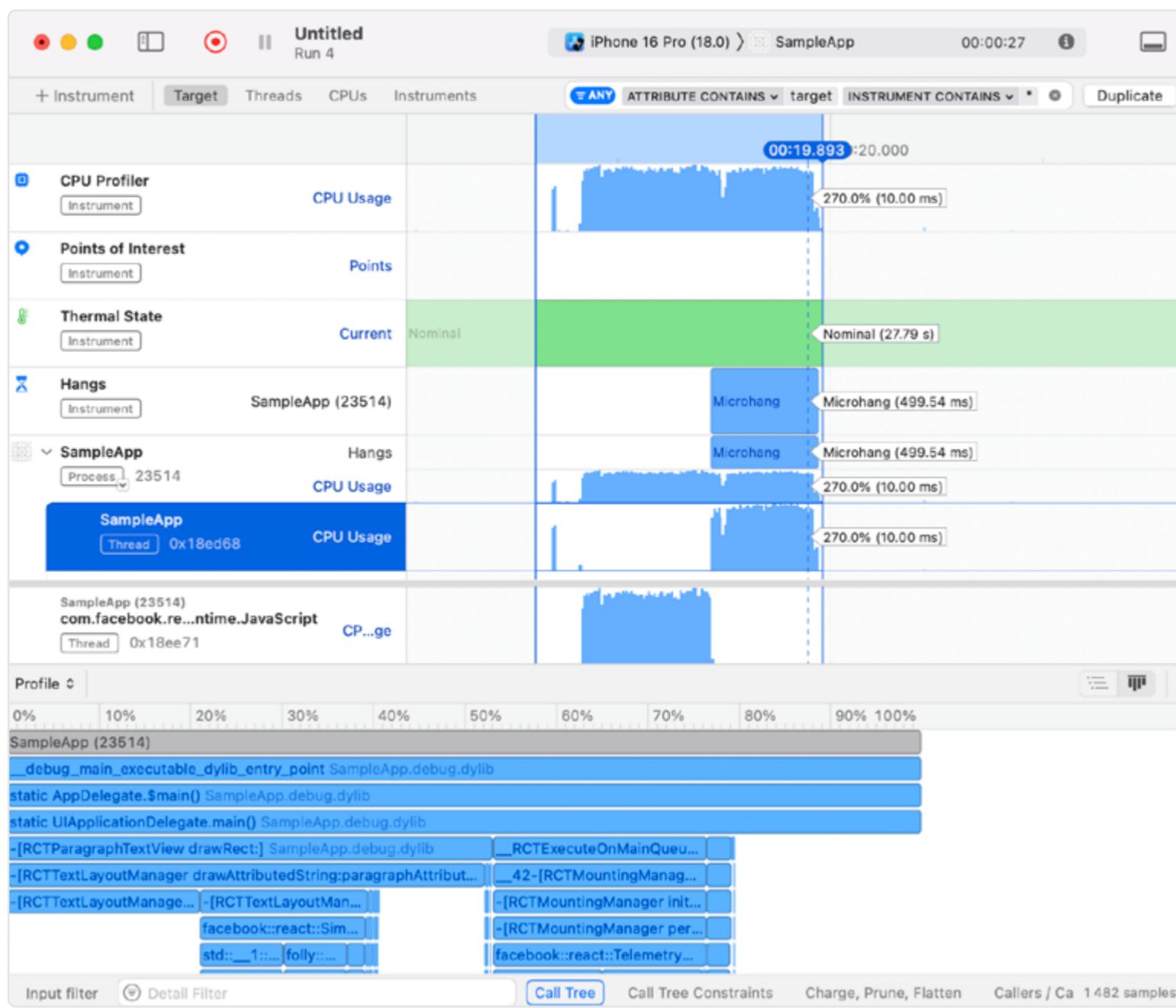
새 창이 열릴 것입니다. 앱 프로파일링을 시작하려면, 대상 장치 옆에 있는 드롭다운 메뉴를 클릭하고 (여기서는 iPhone 16 Pro), 앱(SampleApp)을 선택한 다음, 기록 버튼(빨간 원)을 누르세요:



프로파일러는 당신의 앱을 재시작하고 처음부터 프로파일링을 위해 CPU samples 수집을 시작할 것입니다. 이제, 당신이나 당신의 사용자가 발견했고 당신이 조사하고 싶은 성능 회귀를 재현하는 방식으로 당신의 앱을 사용할 수 있습니다. 충분한 데이터가 있다면, Instruments window에서 기록을 중지하세요.

우리의 경우, 우리는 ScrollView 내에서 5,000개의 뷰를 표시하는 애플리케이션을 렌더링하고 있습니다. 그리고 버튼이 눌렸을 때 모든 상태 업데이트 시 리렌더링됩니다. 그에 더해, 우리는 목록을 아래로 밀어낼 이미지를 토글하여, re-layout을 유발할 것입니다.

Instruments CPU Profiler에서 그러한 인터랙션의 프로파일 결과는 다음과 같습니다:



SampleApp에서 CPU 프로파일러의 결과로, main thread, JS thread, 그리고 감지된 행(hangs)의 세부 내용을 보여줍니다

여기서 분석할 내용이 많습니다. 우리는 27초의 데이터를 수집했지만, 그 시간 중 작은 부분에만 관심이 있습니다. Cmd + 단축키를 사용하여 타입라인을 우리가 관심 있는 지점으로 확대하고 마우스나 터치패드로 가로 스크롤을 할 수 있습니다.

우리의 경우, 우리는 버튼을 누르고 앱이 업데이트되기를 기다리는 전체 시간 중 약 ~1.3s에 초점을 맞추고 있습니다. 당신은 UI 스레드(SampleApp으로 레이블링된)에 짧은 스파이크가 나타나는 것을 확인할 수 있습니다. 그에 이어 JavaScript 스레드에 집중적인 CPU work이 있고, UI 스레드에도 많은 작업이 있습니다. 프로파일은 하단에 flame graph 형태로 표시되며, React Native의 내부 호출 사이트를 보여줍니다 (시스템 라이브러리는 필터링됨).

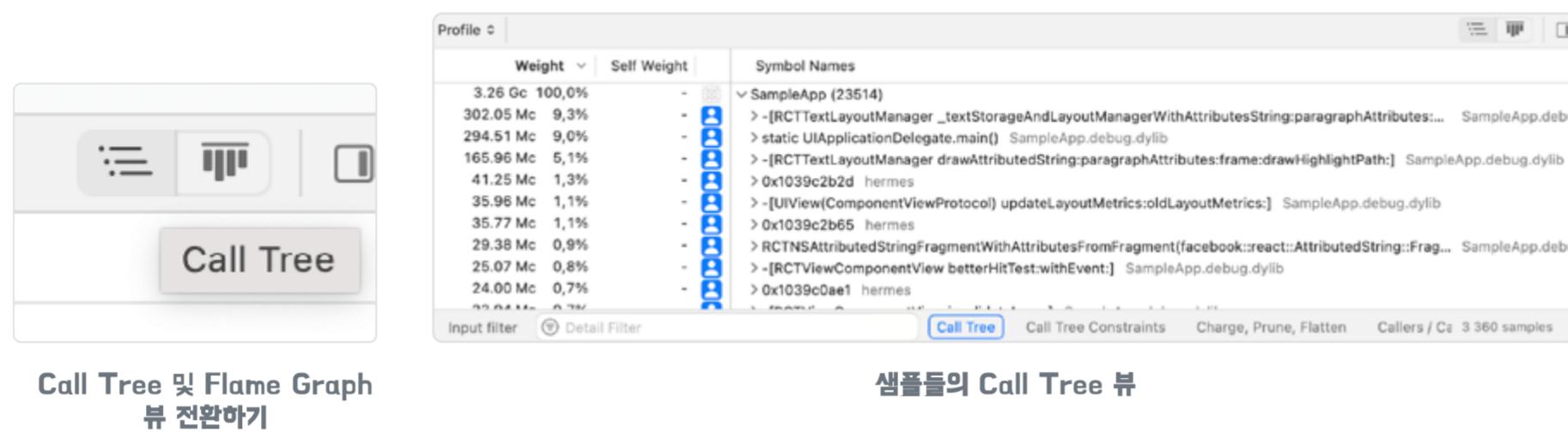
UI 스레드에서 높은 CPU workload 동안 "Microhang"가 발생하는 반면, 높은 JS thread load에서는 그렇지 않다는 점에 주목하세요. 이는 UI 스레드가 많은 작업을 하고 있으며, 우리 콘텐츠와의 즉각적인 인터랙션을 허용하지 않을 것임을 나타냅니다. 하지만 더 나쁠 수도 있습니다 만약 스레드가 완전히 차단되었다면, "Hang"으로 표시될 것이며, 당신이 조사해야 할 높은 우선순위가 되어야 합니다. Hang tool은 Time Profiler와 함께 제공되며, 우리 앱 플로우의 문제가 있는 부분을 빠르게 알아차릴 수 있는 방법으로 사용될 수 있습니다.



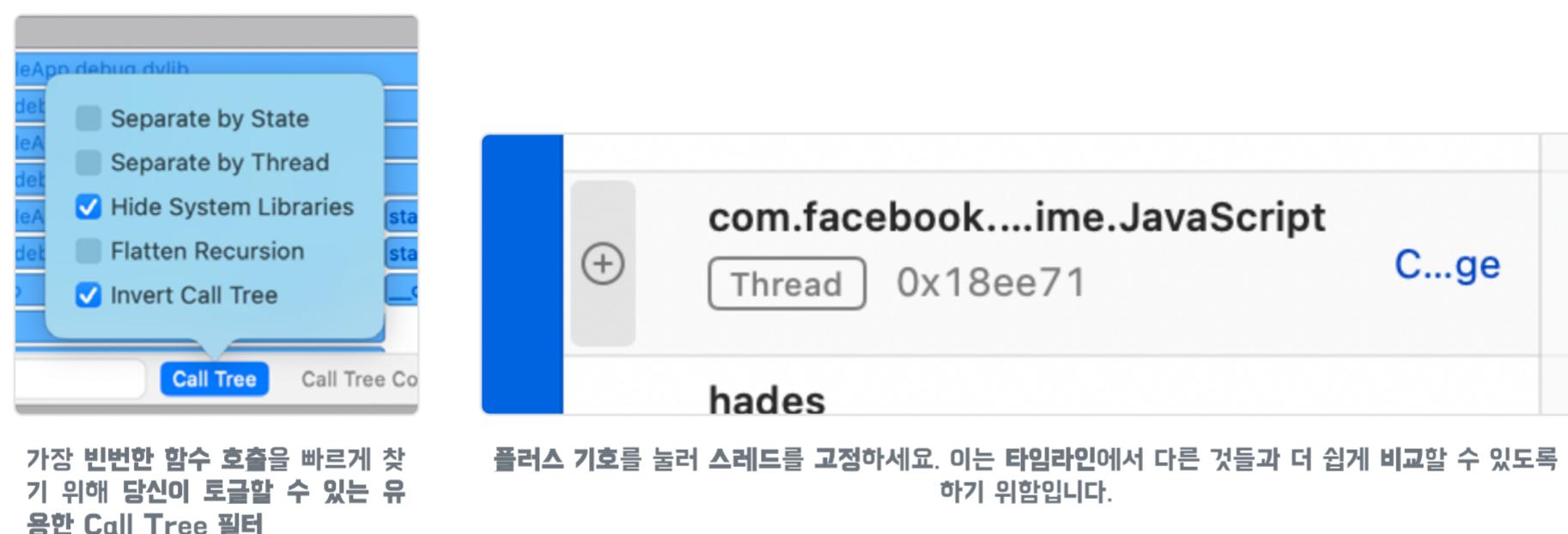
당신은 JS 스레드를 효과적으로 차단할 수 있으며, 사용자는 여전히 당신의 앱의 네이티브 UI 요소와 상호작용할 수 있을 것입니다(예: 버튼을 누르는 것과 같이 말이죠). 그것은 그냥 즉시 JS 스레드로 다시 통신하지 않을 뿐입니다. 이는 우리가 종종 당연하게 여기지만 감사할 만한 가치가 있는 React Native의 멋진 설계 특징입니다.

CPU Profiler 도구는 문제의 원인을 정확히 파악하는 데 도움이 되도록 꽤 많은 커스터마이제이션을 허용합니다. 예를 들어, 당신은 Call Tree를 필터링하여 시스템 라이브러리를 숨기거나 Call Tree를 완전히 반전시켜서 다른 프로파일러에서 알고 있을 수도 있는 bottom-up view에 접근할 수 있습니다. 이는 'counts'(여기서 예를 들어 Mc는 Mega counts, 즉 수백만 개의 수집된 샘플을 의미함)로 표현되는 가중치로 정렬할 수 있습니다.

이는 본질적으로 어떤 상위 레벨 함수가 그것들을 호출했는지와 상관없이 대부분의 시간이 소비되는 함수를 볼 수 있도록 허용합니다. 만약 그것이 도움이 된다면, 스레드별 호출을 검사할 수도 있습니다.



당신은 Call Tree와 Flame Graph 뷰 사이를 전환할 수 있습니다. 이는 React Native DevTools에서 이미 알고 있을 수도 있는 기능입니다. 우리의 초기 스크린샷에서는, JavaScript 스레드를 고정했습니다. 이는 UI 스레드 및 감지된 행(hangs)과 어떻게 상관관계가 있는지 보기 위함입니다.



CPU 프로파일러 덕분에, 우리는 ScrollView에 5,000개의 요소를 저장하는 것이 우리 앱의 성능에 최적이지 않다는 것, 그리고 이에 대해 조치해야 한다는 것을 결론 내릴 수 있습니다.

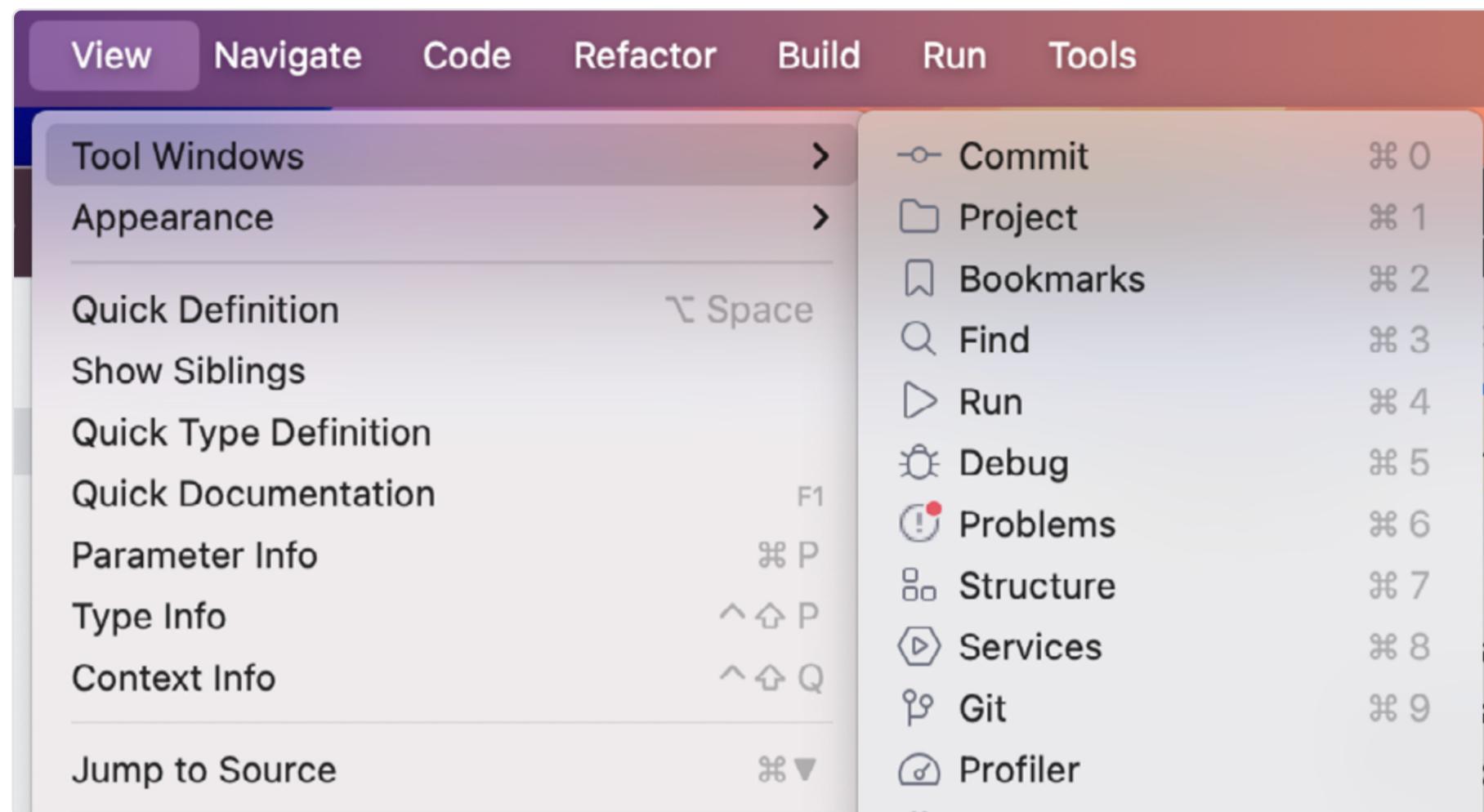
Android

Android 플랫폼의 경우, 우리는 Android Studio와 내장 도구를 사용하여 우리 앱의 CPU, 메모리, 네트워크, 그리고 배터리 사용량을 프로파일링할 수 있습니다.

Android Profiler

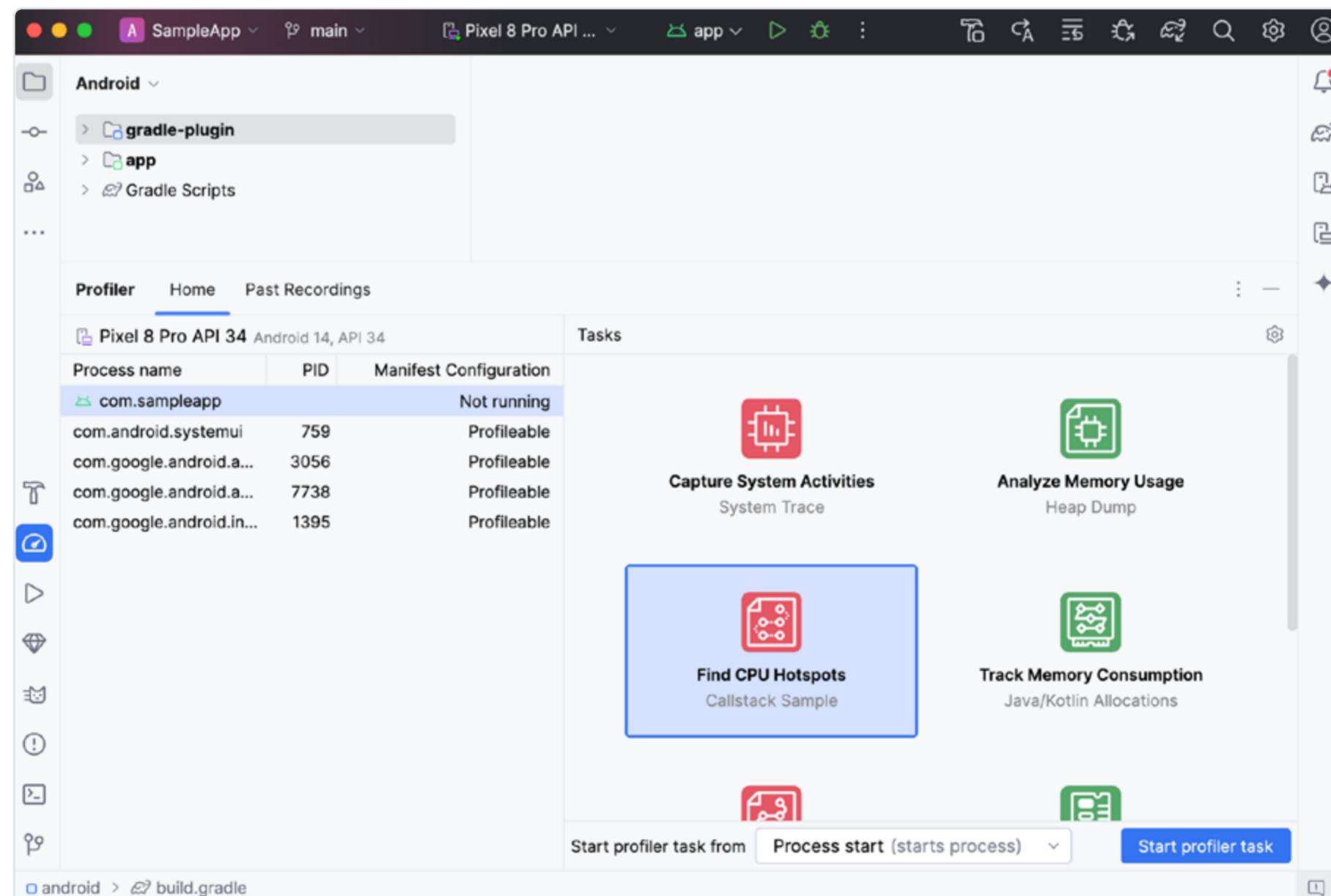
Android Studio는 JetBrains에 의해 개발된 IDE이며, 이는 Google에 의해 공식적으로 지원되며, 어떤 Android 앱이든 개발하는 데 사용될 수 있는 공식 Android IDE입니다. 이는 매우 강력하고 Android Profiler를 포함하여 많은 기능을 한곳에 포함하고 있습니다.

프로파일러를 열려면, **Android Studio 메뉴 바에서 View > Tool Windows > Profiler**를 선택하세요:



View > Tool Windows > Profiler menu

또는 **Android Studio의 툴바에서 "Profiler: Run 'app' as profileable"**을 클릭하세요. 만약 그것이 작동하지 않는다면, 당신은 **debuggable build**를 실행하거나 **공식 Android 개발자 가이드라인**을 사용하여 당신의 앱의 **profileable version**을 직접 생성하고 실행해야 할 것입니다.



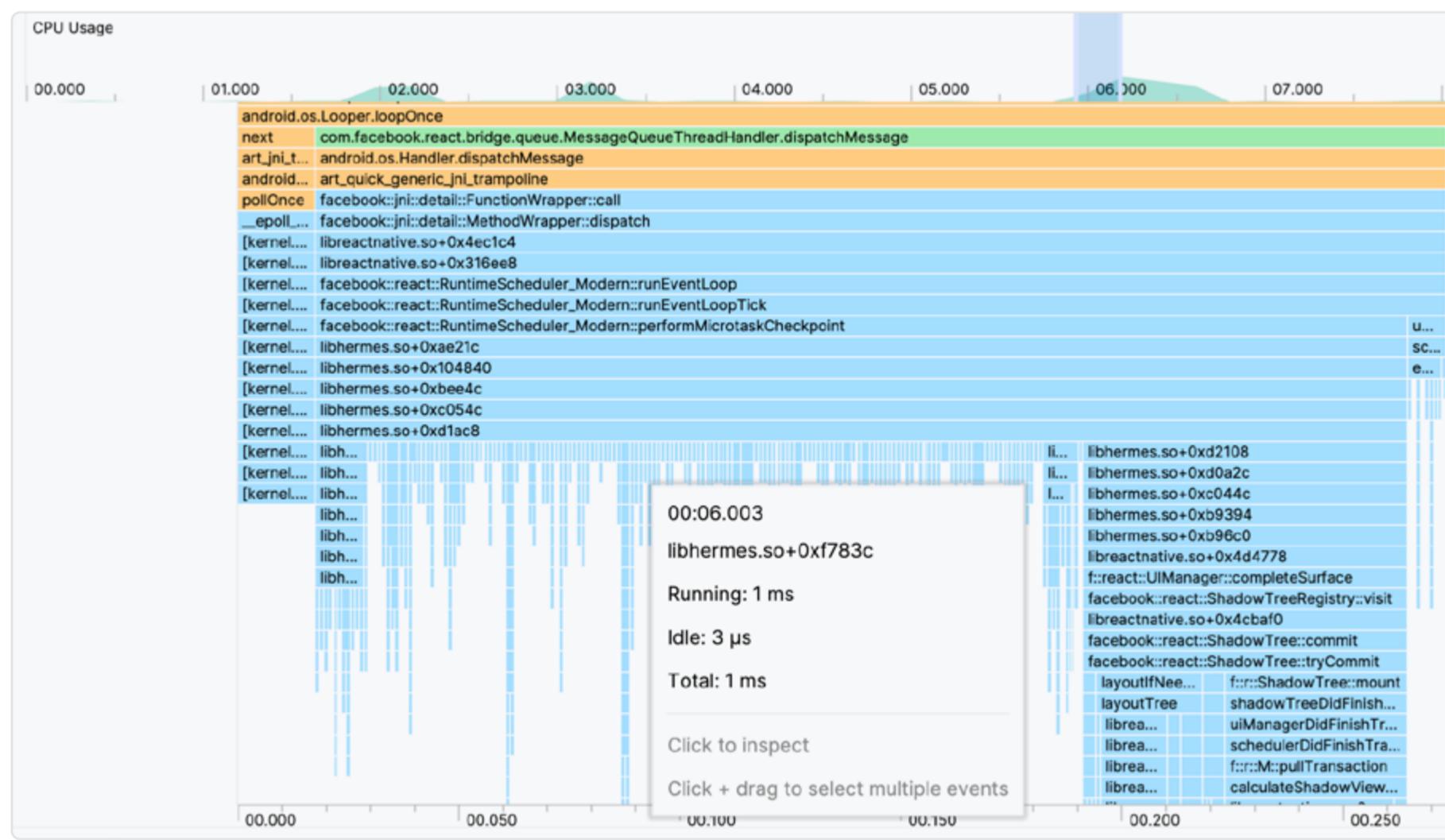
Android Studio Profiler에 포함된 Find CPU Hotspots, Track Memory Consumption 등의 분석 기능

이제, 당신은 당신의 장치에서 실행 중인 profileable 앱에 연결할 수 있을 것입니다. 우리는 "Find CPU Hotspots" 작업을 사용하고 "Start profiler task" 버튼을 눌러 우리 앱에서 샘플 수집을 시작할 것입니다.



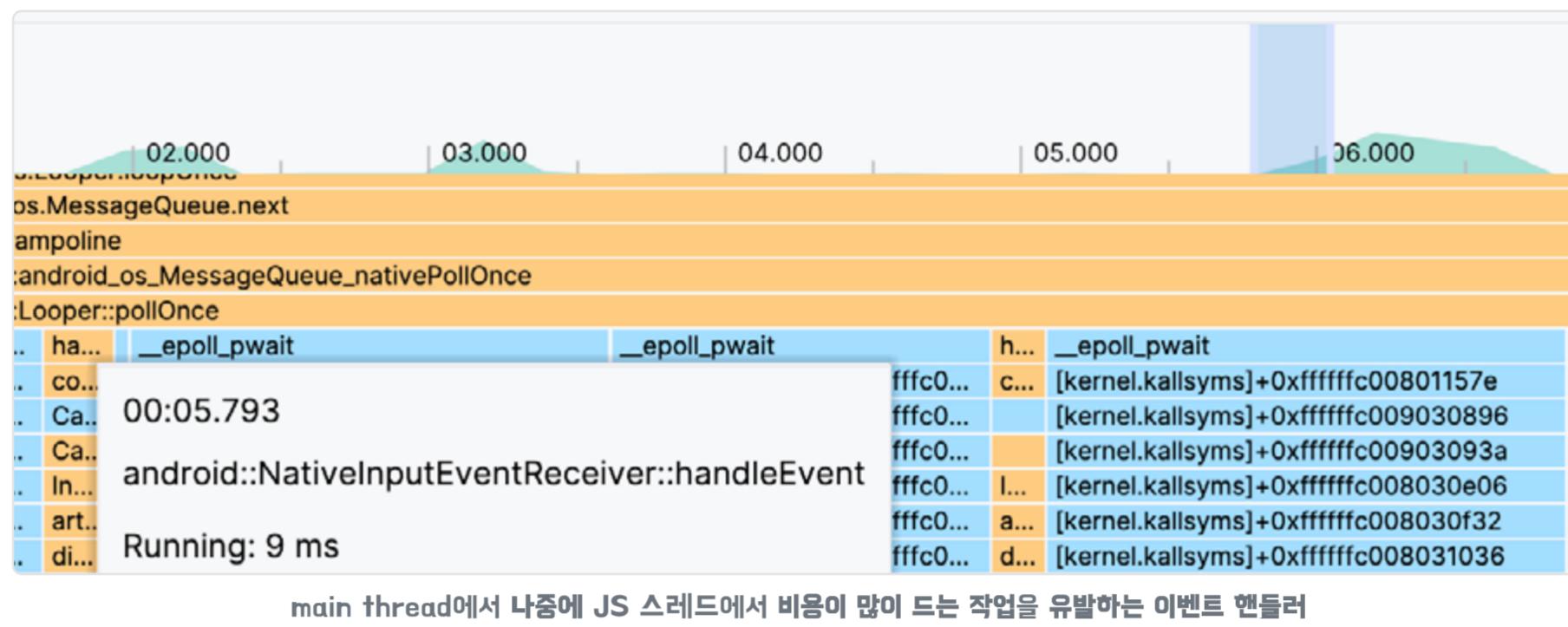
Android 기기 생태계가 iOS보다 훨씬 더 파편화되어 있다는 점에 유의할 필요가 있습니다. 휴대폰, 태블릿, TV 등, 저가형부터 중저가형, 그리고 고가형까지 다양하게 존재합니다. 프로파일링을 위해 사용 가능한 가장 저가형 기기나 emulator를 선택하세요. 가능하다면 실시간 사용자 모니터링에서 얻은 데이터를 사용하세요.

우리는 iOS 앱에서와 동일한 시나리오를 수행할 것입니다: 우리가 가진 거대한 목록의 리렌더링을 유발하기 위해 우리의 버튼을 몇 번 누르는 것입니다. 프로파일링을 중지하면, UI는 CPU와 다양한 스레드에서 수행된 작업의 세부 내용을 보여주는 flame graph로 변경될 것입니다. 버튼을 누른 직후의 해당 JS 스레드를 확대해 봅시다. 더 높은 CPU usage의 징후를 볼 수 있습니다:



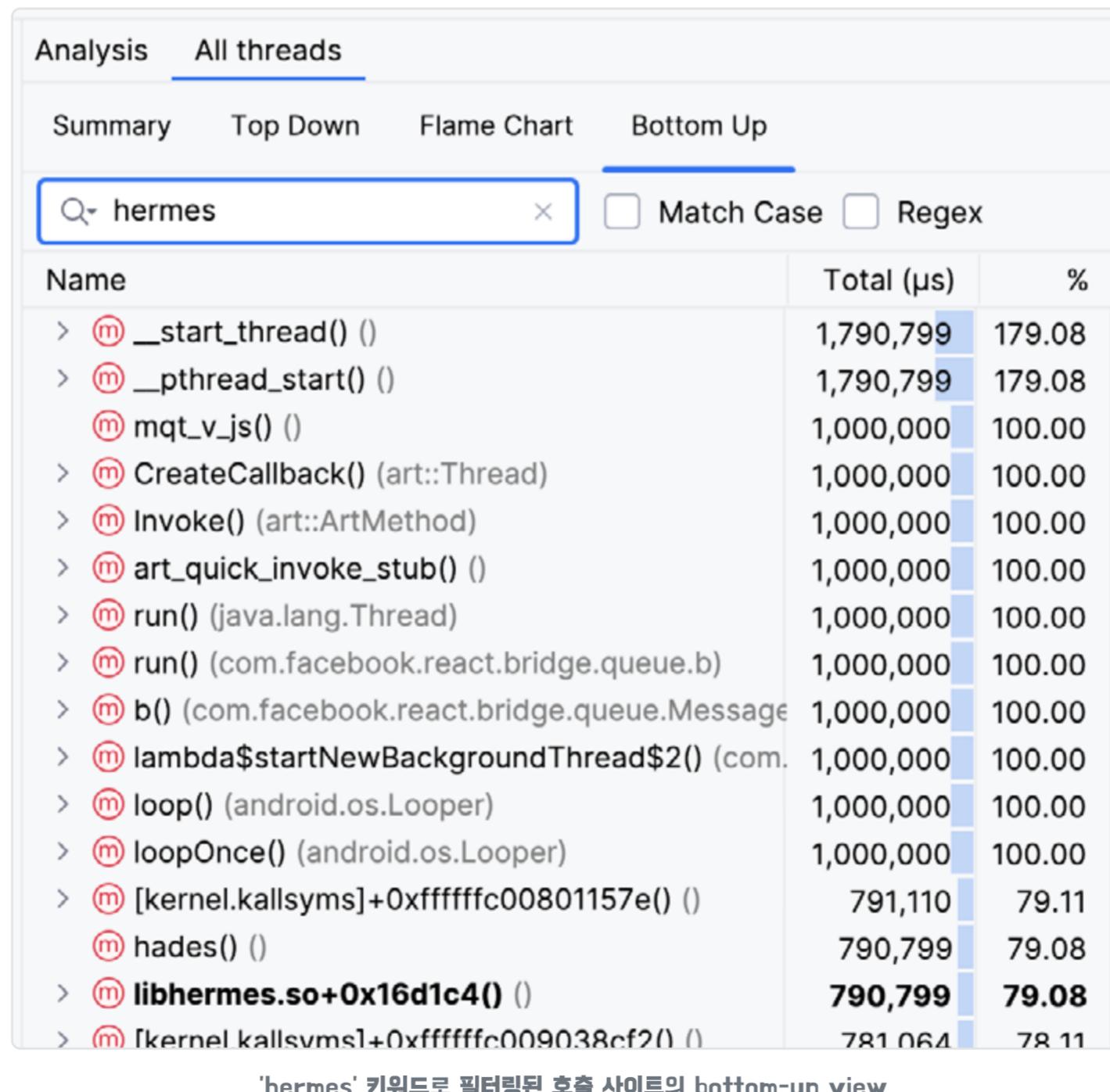
우리는 Hermes JS engine에서 오는 많은 미세한 1ms 스파이크 활동을 관찰할 수 있습니다. 이는 우리가 결국 화면에 그려야 할 5k 뷰를 거치는 것입니다. 그 모든 작업은 240ms 이상에 달합니다. 이는 최소 60 FPS를 달성하기 위한 16.6 ms 예산보다 훨씬 많습니다. 어떻게 전체 시간의 약 3분의 1이 React reconciliation algorithm의 "commit" 단계에서 소비되는지 주목하세요. 이 단계는 Yoga를 사용하여 새로운 뷰를 배치하고, 그런 다음 마운트되고 화면에 그려지는 뷰입니다.

그리고 우리가 flame graph에서 main thread를 보기 위해 조금 위로 스크롤하면, 우리는 버튼을 눌렀을 때의 이벤트 핸들러, 그리고 손가락을 뗐을 때의 다른 하나가 뒤따르는 것을 확인할 수 있습니다.



터치 릴리즈 타이밍은 JS 스레드에서의 증가된 작업 시작과 거의 완벽하게 일치합니다. 이는 React Native에 의해 내부적으로 sync JSI calls을 통해 달성됩니다. 레거시 아키텍처에서는, 그 정보가 브릿지에 의해 직렬화되고 브로드캐스트되어, 이 인터랙션에 추가 오버헤드를 추가했습니다.

다른 어떤 프로파일링 도구와 마찬가지로, 우리는 또한 Call Tree의 탑다운 또는 바텀-업 분석에 접근할 수 있습니다. 여기서 우리는 추가적으로 라이브러리별 그리고 우리가 관심 있는 호출별로 필터링할 수 있습니다.

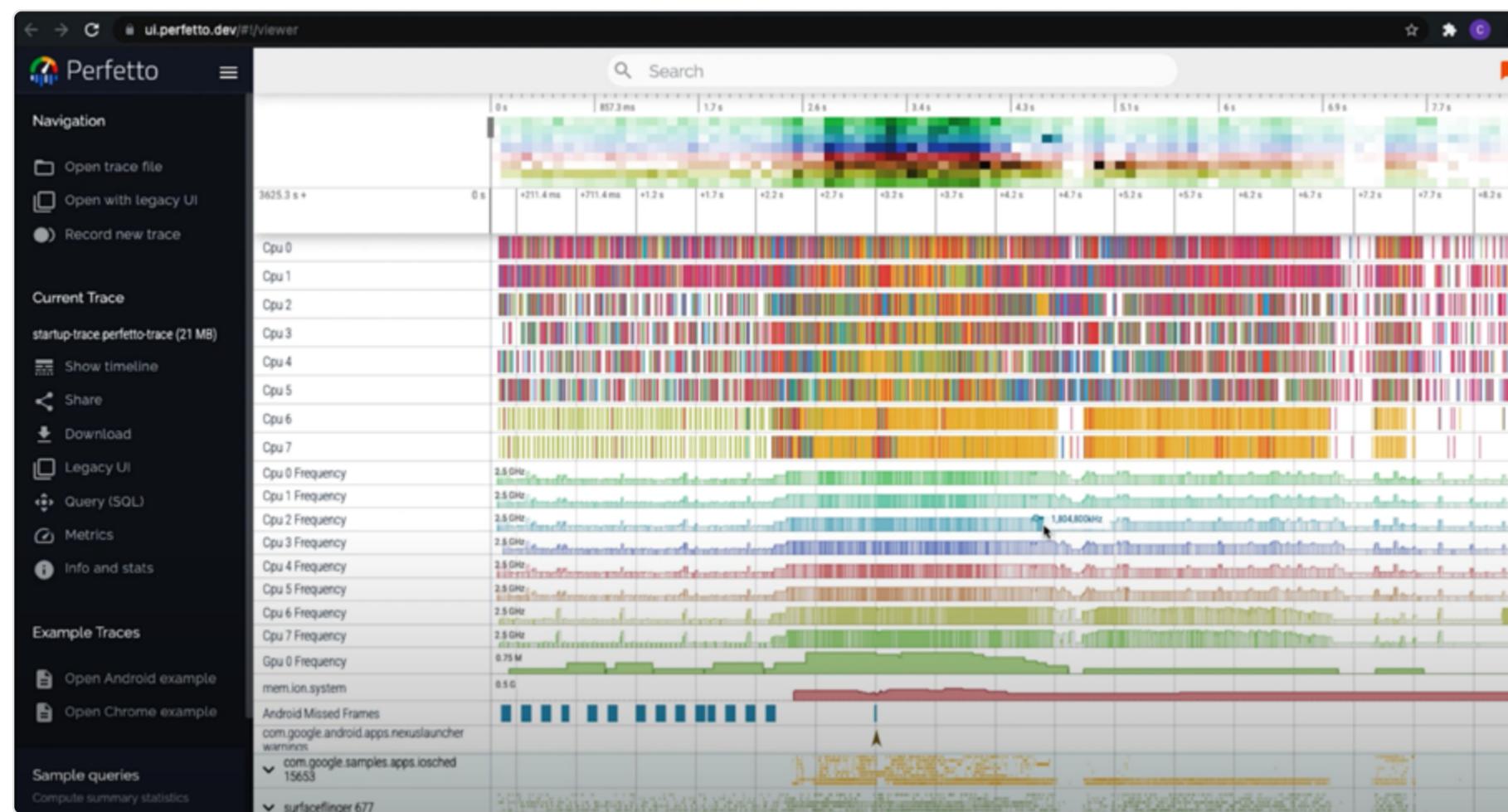


'hermes' 키워드로 필터링된 호출 사이트의 bottom-up view

Android Studio 또는 Chrome JS Profiler와 같은 프로파일링 도구는 트레이스를 내보내고 그것을 서드파티 도구에서 로드할 수 있도록 허용한다는 점을 기억할 가치가 있습니다.

Perfetto

유용한 도구 중 하나로 Perfetto가 있습니다. 이 도구는 시스템 프로파일링, 앱 추적, 그리고 이전에 저장한 추적 데이터를 불러와 분석할 수 있도록 지원합니다. Perfetto는 **Android Studio Trace Viewer**에 추가 기능을 제공합니다. ui.perfetto.dev에서 제공하는 온라인 뷰어를 통해 추적 데이터를 업로드하면 다양한 관점에서 자세히 살펴볼 수 있습니다.



Perfetto 웹 앱에서의 Android Studio 메모리 추적

프로파일러를 사용하는 방법을 아는 것은 코드베이스에서 발생하는 성능 문제를 해결하는 데 가장 중요한 기술 중 하나입니다. 특히, 자신이 작성하지 않은 코드에서도 매우 유용합니다. 적절한 도구를 활용하면 코드의 성능 특성을 파악하고 필요한 부분에 최적화를 적용할 수 있습니다.

GUIDE

TTI 측정 방법

앱의 TTI(Time to Interactive, 상호작용 가능 시간) 메트릭은 모든 앱이 추적해야 할 두 가지 가장 중요한 메트릭 중 하나입니다. 이 메트릭은 사용자가 앱 아이콘을 터치한 순간부터 터치, 음성, 또는 기타 입력 방식을 통해 상호작용할 수 있는 의미 있는 콘텐츠가 표시되기까지 걸리는 시간을 알려줍니다. 사용자는 이 과정이 가능한 한 빠르게, 이상적으로는 즉시 이루어지기를 기대합니다.

iOS와 같은 운영 체제는 사용자의 과거 경험을 기반으로 머신 러닝 예측을 활용해 특정 앱을 "미리 준비"함으로써 앱이 최대 40% 더 빠르게 열리도록 합니다. 이를 통해 일부 앱이 실제로는 로드 속도가 느리더라도 전체 iOS 생태계가 빠르고 사용자의 의도에 즉각 반응하는 듯한 인상을 줍니다.



TTI(Time to Interactive, 상호작용 가능 시간)는 사용자가 앱을 열고 사용할 수 있게 되는 속도를 나타냅니다. 이는 사용자 경험, 만족도, 유지율, 그리고 결과적으로 앱의 수익에 큰 영향을 미치므로 추적하고 최적화하는 것이 매우 중요합니다.

다양한 품질의 보고서에 따르면, 사용자는 일반적으로 앱이 2~4초 이내에 로드되기를 기대하며, 그렇지 않을 경우 대체 앱(사용 가능한 경우)으로 전환한다고 합니다. 하지만 우리의 경험에 따르면, 외부 보고서를 맹신하기보다는 자신의 데이터를 신뢰하고 건강한 회의적인 태도를 유지해야 합니다. 또한 경험을 통해 TTI는 가능한 한 낮아야 한다는 것을 알 수 있습니다.

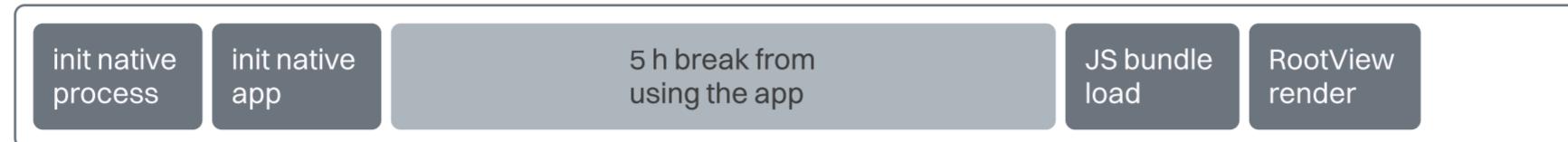
하지만 최적의 노력과 결과 사이에서 어디에 선을 그어야 할까요? 엔지니어링 팀이 수개월 동안 노력해서 기준 Android 기기에서 TTI를 2.9초에서 2.3초로 줄이는 것이 정말 가치 있을까요? 각 사용자층은 고유하며, 사용자 행동을 관찰하는 것만이 답을 줄 수 있습니다. 이 장에서는 TTI를 최대한 낮추기 위한 기술에 초점을 맞출 것입니다.

TTI를 안정적으로 측정하기

기반 운영 체제와 그 기능에 따라, 앱은 콜드 스타트, 웜 스타트, 핫 스타트 또는 사전 준비(prewarmed) 상태로 실행될 수 있습니다. 이는 매일 같은 시간대에 여는 뉴스 앱이라도 상황에 따라 다르게 동작할 수 있다는 뜻입니다.

매일 아침 8시에 여는 앱이 iOS에서 “콜드” 상태일 때는 보통 10초가 걸리지만, 며칠이 지나면 6초 미만으로 부팅이 시작될 수 있습니다. 이는 상당한 차이입니다.

UI thread



JS thread



iOS 사전 준비(prewarming)가 React Native 앱 시작 파이프라인의 다양한 단계에 어떤 영향을 미치는지를 보여주는 다이어그램

다른 상황에서는 아침에 앱을 열었다가 저녁에 다시 돌아왔을 때 앱이 백그라운드에 남아 있을 수 있습니다. 이 경우 앱에 다시 접근하면 React 부분만 렌더링하므로 콜드 스타트에 비해 전체 프로세스가 훨씬 빨라집니다.

UI thread



JS thread



Android에서 앱을 백그라운드로 전환하는 것이 React Native 앱 시작 파이프라인의 다양한 단계에 어떤 영향을 미치는지를 보여주는 다이어그램

만약 우리가 앱 시작부터 완전한 상호작용 가능 상태까지의 TTI를 단순히 측정한다면, 매우 다른 측정값을 얻게 되어 이 메트릭이 개선되었는지 퇴화했는지에 대한 통찰을 얻을 수 없을 것입니다.

따라서 앱을 콜드 부트 상태에서만 측정하고, 사전 준비(prewarm), 월, 핫 부트 상태에서의 측정은 제외하는 것이 합리적입니다. 다행히 iOS와 Android 개발자들은 이 문제를 다룰 수 있는 방법을 개발자들에게 제공합니다.

성능 마커 설정

React Native 앱 시작 파이프라인의 각 단계는 성능 마커 특정 이벤트가 발생한 시간을 수집하는 코드 조각을 통해 계측할 수 있습니다. React Native 애플리케이션에서는 최소한 다음 파이프라인 단계에 집중해야 합니다:

- 네이티브 프로세스 초기화 `nativeAppStart`와 `nativeAppEnd` 마커로 설명됩니다.
- 네이티브 앱 초기화 `appCreationStart`와 `appCreationEnd` 마커로 설명됩니다.

■ JS 번들 로드: `runJSBundleStart`와 `runJSBundleEnd` 마커로 표시됩니다.

■ 리액트 네이티브 루트 뷰 렌더링: `contentAppeared` 마커로 표시됩니다.

■ 리액트 앱 렌더링: `screenInteractive` 마커로 표시됩니다.

이러한 마커를 iOS, Android, 그리고 리액트 코드 전반에 걸쳐 기록하려면 네이티브 모듈이나 서드파티 라이브러리가 필요합니다. 우리가 가장 만족스럽게 사용하고 있는 것은 `react-native-performance`입니다. 이 라이브러리는 iOS와 Android에서 `RNPerformance` 클래스를 제공하여 사용자 정의 마커를 설정할 수 있게 해줍니다.

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

iOS에서 사용자 정의 마커

```
import com.oblador.performance.RNPerformance;  
  
RNPerformance.getInstance().mark("myCustomMark");
```

Android에서 사용자 정의 마커

이후 웹에서와 마찬가지로 성능 API를 사용해 React 코드에서 이 마커들을 이용할 수 있습니다.

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

💡 사용자 정의 마커를 정의하는 대신, `react-native-performance` 라이브러리에서 제공하는 기본 마커(예: `nativeLaunchStart`, `runJsBundleEnd`)를 사용할 수 있습니다. `nativeLaunchStart`는 메인 함수 실행 전, 즉 프리워밍 단계에서 측정되며, 나머지 마커들은 프리워밍 이후에 측정된다는 점에 유의해야 합니다. 필요에 따라 이를 필터링하거나 `main()`에서 사용자 정의 마커를 생성해야 할 수도 있습니다.

네이티브 프로세스 초기화

먼저, 콜드 스타트인지 확인합니다. iOS에서는 `ProcessInfo`를 사용해 이 정보를 얻을 수 있습니다:

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

Android에서는 `onActivityCreated` 라이프사이클 메서드를 활용해 해당 정보를 확인해야 하므로, 코드가 다소 길어질 수 있습니다:

```
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
                }
            }
        })
    }
}
```

앱이 포그라운드에 있을 때만 TTI를 측정하고 싶으므로, 이를 고려해야 합니다. iOS의 `AppDelegate.swift` 파일에서 `didFinishLaunchingWithOptions` 핸들러에 후크를 추가할 수 있습니다:

```
@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
    }
}
```

Android의 `MainApplication.kt` 파일에서는 `processInfo.importance` API를 통해 이를 확인할 수 있으며, 이를 `isForegroundProcess`라는 헬퍼 함수로 래핑할 수 있습니다:

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

iOS와 Android에서 아래 API를 사용해 앱 초기화 타임스탬프에 접근하여 nativeLaunchStart 메트릭 데이터를 얻을 수 있습니다:

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

iOS nativeLaunchStart

```
Process.getElapsedCpuTime()
```

Android nativeLaunchStart

nativeLaunchEnd의 가장 근접한 추정치는 iOS에서는 네이티브 모듈이 초기화되는 시점, Android에서는 Android Content Provider가 생성되는 시점입니다:

```
+ (void) initialize
```

iOS nativeLaunchEnd

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

Android nativeLaunchEnd

초기화 및 기타 프리메인 단계는 iOS 앱이 시작되고 main()이 실행되기 몇 시간 전부터 선제적으로 실행될 수 있습니다. 따라서 프리메인 초기화 시점의 프로세스 시작 시간과의 차이를 고려해야 합니다.

네이티브 앱 초기화

iOS 앱이 생성되는 시점에 후크를 걸어 appCreationStart 마커를 얻으려면 main 메서드에 후크를 추가할 수 있습니다. Android의 경우 MainApplication의 onCreate 라이프사이클 메서드를 활용할 수 있습니다:

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

iOS appCreationStart

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

Android appCreationStart

iOS의 프리워밍은 `main()`이 `UIApplicationMain`을 호출하기 직전까지 앱의 시작 시퀀스를 실행하므로, 이 시점까지 진행되며 타이밍에 미치는 영향에서 안전합니다.

이후 네이티브 앱 생성 프로세스는 iOS에서는 `didFinishLaunchingWithOptions`, Android에서는 `onStart` 라이프사이클 메서드에서 끝나며, 여기서 `appCreationEnd` 로직을 후크할 수 있습니다:

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

iOS appCreationEnd

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

Android appCreationEnd

JS 번들 로드

이제 React Native 관련 메트릭인 `runJSBundleStart`에 접근할 수 있습니다. iOS에서는 React Native 코어에서 발생하는 "RCTJavaScriptDidLoadNotification" 이벤트를 글로벌 알림 센터를 통해 감지할 수 있으며, Android에서는 코어에서 `ReactMarker`를 가져와 해당 이벤트를 감지할 수 있습니다:

```
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

비슷한 방식으로 JS 번들 로딩의 종료를 감지하여 runJSBundleEnd 마커를 얻을 수 있습니다:

```
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 루트 뷰 렌더링

동일한 React Marker 인프라를 재사용하여 React Native 콘텐츠가 나타나는 시점을 감지하고 contentAppeared 마커를 설정할 수 있습니다:

```
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 앱 렌더링

마지막 마커인 screenInteractive는 전체 TTI 메트릭을 제공합니다. 이 마커를 정의할 수 있는 단일 지점은 없습니다. 이는 각 애플리케이션에 따라 다르며, 사용자가 앱 콘텐츠와 안전하게 상호 작용할 수 있는 시점을 결정해야 합니다. 기본적으로, 예를 들어 홈 화면의 주요 useEffect의 초기 마운트 시점에 이를 설정할 수 있습니다.

결국, 사용자가 일반적으로 시작하는 앱의 최상위 콘텐츠가 표시되는 시점과 같이 더 적합한 지점을 찾아야 합니다:

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

React 앱에서 screenInteractive 마커

모든 것을 종합하면, 앱의 전체 초기화 파이프라인 네이티브 초기화부터 화면, 이상적으로는 여러 화면에 걸쳐 상호작용 가능 상태에 이르기까지를 한눈에 파악할 수 있습니다.

이 데이터를 데이터베이스나 실시간 사용자 메트릭 플랫폼으로 전송하여 다양한 디바이스에서 앱 성능이 어떻게 영향을 받는지, 그리고 더 중요한 점으로 시간이 지남에 따라 어떻게 변하는지를 관찰할 수 있습니다.

이 데이터를 통해 TTI와 앱의 고객 성공 메트릭 간의 정확한 상관관계를 파악할 수 있습니다.

이 데이터는 엔지니어링 팀의 우선순위를 결정하는 데 영향을 미쳐야 하며, 최적화가 필요한 시점인지, 혹은 더 이상 효과적이지 않은 최적화를 줄이고 다른 곳에 집중해야 할지를 알려주는 신호를 제공합니다.

GUIDE

네이티브 메모리 관리 이해하기

네이티브 프로그래밍 언어에서는 개발자가 앱 실행 중 메모리와 리소스를 어느 정도 직접 관리해야 합니다. 이 방식은 사용하는 언어와 플랫폼에 따라 다릅니다. 메모리 관리는 크게 두 가지로 나눌 수 있습니다: 수동 관리와 자동 관리.

수동 메모리 관리에서는, 예를 들어 C나 C++와 같은 언어에서, 메모리 할당과 해제를 개발자가 직접 책임져야 합니다. 이 방식은 일반적으로 코드가 장황하고 오류가 발생하기 쉽지만, 제대로 수행하면 최고의 성능을 얻을 수 있습니다.

대부분의 경우, 코드의 가독성과 유지보수를 쉽게 하기 위해 자동 메모리 관리를 선택하는 것이 좋습니다. 하지만 코드에서 매 나노초 단위의 성능을 극대화하고 메모리 관리에 대한 깊은 이해가 있다면, 수동 관리가 더 나을 수 있습니다. 이는 예를 들어 게임 개발에서 흔히 사용되는 방식입니다.

프로그래밍 언어가 메모리를 관리하는 일반적인 패턴을 아는 것은 중요합니다. 이를 통해 작성하는 코드의 효율성과 예측 가능성을 높일 수 있습니다. 먼저 자동 메모리 관리부터 살펴보겠습니다.

메모리 관리 패턴

이 방식에서는 가비지 컬렉터 알고리즘을 사용해 메모리를 자동으로 관리합니다. 여기서는 두 가지 일반적인 접근 방식을 강조하며, 각각의 장단점을 알아보겠습니다.

참조 카운팅

모바일 개발에서 이 패턴은 Objective-C와 Swift(자동 참조 카운팅, ARC라는 이름으로)에서 사용되며, Python, PHP 등에서도 적용됩니다. 객체에 대한 참조를 전달할 때마다 프로그래밍 언어의 참조 카운터는 해당 객체의 사용 횟수를 증가시킵니다. 여기에는 객체를 다른 객체로 전달하거나 함수의 인자로 전달하는 경우가 포함됩니다. 참조가 종료되면(예: 스코프를 벗어날 때) 사용 횟수가 감소합니다. 사용 횟수가 0에 도달하면 객체는 할당 해제되고 리소스가 해제됩니다.

이 방식은 런타임 성능에 영향을 줄 수 있지만, 마지막 참조가 스코프를 벗어나는 즉시 리소스를 확실히 정리하고 해제한다는 장점이 있습니다. 이상적인 해결책처럼 보일 수 있지만, "참조 사이클(Reference Cycles)"에 주의해야 합니다. 이는 두 객체가 서로 "강한(strong)" 참조를 유지할 때 발생하며, 참조 카운터가 1에서 멈춰 객체가 해제되지 않는 상황을 만듭니다. 이 문제를 해결하는 일반적인 방법은 참조 사이클이 발생할 가능성이 있는 곳에서 약한(weak) 참조를 사용하는 것입니다.

가비지 컬렉터

이 메모리 관리 패턴은 JavaScript, Java, C# 등에서 사용됩니다. 가비지 컬렉터(GC)는 코드에서 발생하는 작업과 독립적으로 작동합니다. 이를 별도의 프로세스로 생각할 수 있습니다. 예를 들어, Hermes 엔진은 Hades GC를 실행하는 별도의 스레드를 생성합니다. GC는 주기적으로 호출되거나 객체 할당 중에 실행될 수 있습니다. 가비지 컬렉터는 프로그램의 메모리를 스캔하여 더 이상 도달할 수 없는 객체를 찾아냅니다. 이러한 도달 불가능한 객체는 이후 해제됩니다.

이 방식은 참조 사이클 문제가 없습니다. 하지만 가비지 컬렉션 프로세스가 시작되는 시점은 비결정적이며, 객체가 언제 파괴될지에 대한 제어권이 없습니다. 또한, GC 알고리즘이 동시성을 지원하지 않는 경우, GC 프로세스 중에는 애플리케이션이 일시 중지되어 사용자 경험에 영향을 줄 수 있습니다.

수동 관리

C나 C++와 같은 저수준 언어에서는 메모리의 객체 할당과 해제를 완전히 제어할 수 있습니다. 일반적으로 메모리 할당은 두 가지 영역에서 이루어집니다: 스택과 힙.

스택은 데이터를 후입선출(LIFO) 방식으로 저장하는 메모리 영역으로, 메모리 할당과 해제가 자동으로 이루어집니다. 매우 빠르지만 크기와 범위가 제한적입니다. 반면 힙은 동적 할당에 사용되는 더 큰 메모리 영역으로, 데이터에 전역적으로 접근할 수 있습니다. 힙에 메모리를 할당할 때(예: new 또는 malloc 사용), 더 이상 필요하지 않을 때 이를 해제하는 책임은 개발자에게 있습니다.

이 기법은 제대로 수행하면 가장 효율적이지만, 제대로 알지 못하고 사용하면 쉽게 문제를 일으킬 수 있습니다.

메모리 관리

이제 이론을 알았으니 실습으로 들어가 봅시다! 이 섹션에서는 Kotlin, Swift, C++의 메모리 관리 방법을 살펴보며 각 언어에서 메모리를 적절히 관리하는 방법에 대해 더 깊이 이해해 보겠습니다.

C++

C++ 프로그래밍 언어에서는 메모리 관리가 명시적이고 수동적입니다. 하지만 모든 메모리 관리를 위해 `new`와 `delete` 연산자를 수동으로 호출해야 했던 시절은 오래전에 끝났습니다. 이제는 C++ 표준 라이브러리(`std`)에서 제공하는 스마트 포인터를 사용할 수 있습니다. 이를 통해 메모리 할당과 해제를 수동으로 관리할 필요가 없으며, 애플리케이션에서 메모리 누수를 줄일 수 있습니다.



추가적인 C++ 자료는 C 및 C++ 언어의 포인터 개념에 대한 약간의 지식을 전제합니다. 이에 익숙하지 않다면, 더 배우거나 이 부분을 건너뛰어도 됩니다.

여러 가지 스마트 포인터를 사용할 수 있습니다:

- `std::unique_ptr<T>`

다른 `unique_ptr`로 복사하거나, 값으로 함수에 전달하거나, 복사가 필요한 C++ 표준 라이브러리 알고리즘에 사용할 수 없습니다. `unique_ptr`는 오직 이동(move)만 가능합니다.

- `std::shared_ptr<T>`

참조 카운팅 방식의 스마트 포인터입니다. 하나의 원시 포인터를 여러 소유자에게 할당하고 싶을 때 사용합니다. 예를 들어, 컨테이너에서 포인터의 복사본을 반환하면서 원본을 유지하고 싶을 때 유용합니다.

- `std::weak_ptr<T>`

`shared_ptr`와 함께 사용하기 위한 특수한 스마트 포인터입니다. `weak_ptr`는 하나 이상의 `shared_ptr` 인스턴스가 소유한 객체에 접근할 수 있게 하지만, 참조 카운팅에는 참여하지 않습니다.

가장 흔히 사용되는 스마트 포인터는 `**std::unique_ptr<T>**`로, 지정된 타입에 대해 참조 카운팅 메커니즘을 제공하는 래퍼 객체이며, 이는 Objective-C와 Swift에서 알려진 메모리 관리 메커니즘과 유사합니다. 스마트 포인터의 사용 예제를 살펴보겠습니다:

먼저 `std::unique_ptr`부터 시작합시다:

```
void takeOwnership(std::unique_ptr<std::string> s1) {
    std::cout << *s1;
    // 함수가 종료되면 자동으로 삭제됩니다.
}

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

    std::cout << *str1;

    // 오직 이동(move)만 가능하며, 복사는 허용되지 않습니다.
    takeOwnership(std::move(str1));

    // 여기서 str1이 지워집니다.
```

```
    return 0;  
}
```

위 예제에서는 `make_unique`를 사용해 `std::string`을 보유하는 새로운 `unique` 포인터를 생성합니다. 그런 다음, 역참조(dereferencing)를 통해 문자열 값을 출력할 수 있습니다. 또한, 이 예제는 스마트 포인터의 고유성을 보여주며, 소유권을 다른 함수로 이전할 때는 오직 이동(move)을 통해서만 가능합니다. 이는 예를 들어 불변 데이터 구조를 구축할 때 포인터에 대한 독점적 소유권을 유지하고자 할 때 유용합니다.

이제 `std::shared_ptr`를 살펴보겠습니다. 이해를 쉽게 하기 위해 동일한 예제를 사용하겠습니다:

```
void takeOwnership(std::shared_ptr<std::string> s1) {  
    std::cout << *s1;  
    // 함수가 반환된 후 참조 카운트는 다시 1로 돌아갑니다.  
}  
  
// 참조를 받아들입니다.  
void takeReference(const std::shared_ptr<std::string> &s1) {  
    std::cout << *s1;  
}  
  
int main()  
{  
    // 공유 포인터를 생성합니다.  
    auto str1 = std::make_shared<std::string>("Hello World");  
  
    std::cout << *str1;  
  
    // 참조 카운트를 1 증가시키고 포인터의 복사본을 생성합니다(기본 값이 아닌 포인터 자체).  
    takeOwnership(str1);  
  
    // 객체를 이동시키지 않았기 때문에 여기서도 여전히 접근할 수 있습니다.  
    std::cout << *str1;  
  
    // 여기서는 참조를 전달하므로 참조 카운트를 변경하지 않습니다.  
    // 이는 아무것도 복사하지 않습니다.  
    takeReference(str1);  
  
    std::cout << *str1;  
  
    return 0;  
}
```

위 예제에서는 `std::make_shared()`를 사용해 새로운 공유 포인터를 생성하며, 이를 역참조 (dereferencing)를 통해 출력할 수 있습니다. 우리는 이를 `takeOwnership` 함수에 전달하는데, 이 함수는 공유 포인터를 값으로 전달받아(포인터 자체를 복사하며, 기본 값은 복사하지 않음) 참조 카운터를 증가시킵니다.

공유 포인터는 복사가 가능하므로 값이 지워지지 않고, 이후에도 계속 사용할 수 있습니다. 다음으로, `takeReference` 함수에 전달하는데, 이 함수는 참조만 받기 때문에 참조 카운트를 증가시키지 않습니다. 함수가 호출자를 초과하여 `shared_ptr`을 사용할 필요가 없는 경우에는 참조로 전달하고, `shared_ptr`이 호출자와 독립적으로 유지되어야 하는 경우에는 값으로 전달해야 합니다.

이제 `std::weak_ptr`의 사용 방법을 살펴보겠습니다:

```
void useWeakPtr(std::weak_ptr<std::string> weak) {
    // 객체에 접근하려고 시도합니다.
    // .lock()은 객체가 존재할 경우 약한 포인터(weak_ptr)를 공유 포인터(shared_ptr)로 변환합니다.
    if (auto shared = weak.lock()) {
        std::cout << *shared;           // 안전하게 사용할 수 있습니다.
        // 여기서 shared_ptr의 참조 카운트가 일시적으로 증가합니다.
    } else {
        std::cout << "Object no longer exists!\n";
    }
} // 공유 객체가 스코프 밖으로 나가면, 카운트가 감소합니다.

int main() {
    // 공유 포인터를 생성
    auto str1 = std::make_shared<std::string>("Hello World");

    // 공유 포인터로부터 weak 포인터 생성
    std::weak_ptr<std::string> weak1 = str1; // 참조 카운트를 증가시키지 않습니다.

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

    // weak 포인터를 사용합니다.
    useWeakPtr(weak1); // 객체가 존재합니다.

    // 공유 포인터를 재설정합니다.
    str1.reset(); // 참조 카운트가 0이 되어, 객체가 소멸됩니다.

    // weak 포인터를 다시 사용해 봅니다.
    useWeakPtr(weak1); // "객체가 더 이상 존재하지 않습니다!"를 출력할 것입니다.

    return 0;
}
```

공유 포인터에서 `weak_ptr`을 얻을 수 있으며, 이는 참조 카운트를 증가시키지 않습니다. `weak_ptr`을 사용하려면 `.lock()` 메서드를 사용해야 하는데, 이 메서드는 `weak_ptr`이 가리키는 리소스가 여전히 존재할 때 이를 공유 포인터로 변환합니다.

Android

Android 플랫폼에서 우리는 주로 Kotlin을 사용하며, 이는 메모리 관리를 위해 가비지 컬렉션 메커니즘을 사용하는 JVM(Java Virtual Machine) 위에서 동작합니다. 가비지 컬렉션을 사용하는 JavaScript와 마찬가지로, 리소스를 해제하지 않으면 메모리 누수가 발생할 수 있습니다. 예를 들어, 리스트나 등록 해제나 장기 실행 백그라운드 작업 취소를 잊는 경우가 있습니다.

메모리 관리에 대해 GC에 지시할 수 있는 몇 안 되는 방법 중 하나는 `WeakHashMap`과 같은 Weak 타입의 컨테이너를 사용하는 것입니다 (JavaScript에도 `WeakMap`이라는 유사한 구조가 있습니다). 이러한 타입의 컨테이너는 아이템에 대한 강력한 참조를 생성하지 않습니다. 따라서 다른 어떤 것도 해당 아이템을 참조하지 않으면, GC는 다음 실행 시에 해당 객체들을 할당 해제할 것입니다. 다음은 `WeakHashMap`의 사용 예시입니다.

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

    // 스코프 내부에 항목을 추가합니다.
    run {
        weakMap[String("temp key")] = "some value"
        println("Map size: ${weakMap.size}") // Prints: 1
        println("Value: ${weakMap[key]}") // Prints: some value
    }

    // 데모를 위해 GC를 강제로 실행합니다.
    System.gc()
    Thread.sleep(100)

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

`WeakHashMap`의 중요한 점은 키에 대해서만 약한 참조를 유지한다는 것입니다. 맵에 저장된 값에 대한 약한 참조가 필요한 경우, `WeakReference()`를 사용하여 다음과 같이 값을 할당할 수 있습니다.

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

iOS

iOS에서는 앞서 언급한 바와 같이 Swift와 Objective-C가 전체 생태계에서 참조 카운팅을 사용합니다. 스코프 전반에 걸쳐 참조 카운트가 언제 변경되는지 더 잘 이해할 수 있도록 간단한 예시를 살펴보겠습니다. 데모를 위해 Person 클래스를 사용하겠습니다.

```
class Person {  
    let name: String  
  
    init(name: String) {  
        self.name = name  
    }  
}  
  
do {  
    let person1 = Person(name: "John") // 참조 카운트: 1  
  
    do {  
        let person2 = person1 // 참조 카운트: 2  
    } // person2가 스코프 밖으로 나가고, 참조 카운트: 1  
  
} // person1이 스코프 밖으로 나가고, 참조 카운트: 0
```

person1을 person2에 할당하면 참조 카운트가 증가합니다 (복사본을 생성하는 것이 아닙니다!). 그런 다음 do 블록 스코프 밖으로 나가면 참조 카운트가 다시 1로 설정된 다음 0으로 설정됩니다.

메모리 누수의 원인

앱에서 메모리 누수를 일으키는 몇 가지 일반적인 패턴을 살펴보겠습니다.

수동 메모리 관리 시 할당 해제 누락

수동 메모리 관리를 사용할 때 누수를 발생시키기 쉽습니다. 앞서 언급했듯이, 수동 관리를 선택하면 모든 책임은 사용자에게 있습니다. 큰 힘에는 큰 책임이 따르는 법입니다. 얼마나 쉽게 누수가 발생하는지 보여드리기 위해 다음 예시를 살펴보겠습니다.

```
int main()  
{  
    std::string *str1 = new std::string {"Hey"};  
  
    std::cout << *str1;  
  
    // 앗! 메모리 누수입니다. str1을 삭제하는 것을 잊었습니다.  
  
    return 0;  
}
```

새로운 std::string을 할당하고, 그 값을 출력한 다음, 마지막에 이 문자열을 삭제하는 것을 잊었습니다. 따라서 메모리 누수가 발생했습니다. 이를 해결하려면 더 이상 필요하지 않은 리소스에 대해 delete를 호출해야 합니다.

```
int main()  
{
```

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

    return 0;
}
```

또 다른 방법은 스택에 할당하는 것입니다 (`new`를 사용하지 않음). 이렇게 하면 `str1`이 스코프 밖으로 나갈 때 자동으로 할당 해제됩니다.

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

    cout << str1;

    return 0;
}
```

힙에 무언가를 할당하기 전에 항상 두 번 생각해야 합니다. 힙은 스택보다 훨씬 큰 오버헤드를 갖기 때문입니다. 동시에 스택에 얼마나 많은 것을 할당하는지 주의해야 합니다. 그렇지 않으면 스택 오버플로 위험이 있습니다! 하지만 가능하다면 스택을 사용하고, 필요한 경우에만 힙에 할당하는 것이 좋습니다.

참조 순환

참조 카운팅을 사용할 때 가장 흔한 메모리 누수 원인은 참조 순환, 즉 순환 참조입니다. 간단한 예시를 살펴보겠습니다.

```
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

이 예시에서 클래스 A는 클래스 B에 의존하고, 클래스 B는 다시 클래스 A에 의존하여 참조 순환이 발생합니다.



실제 시나리오에서는 참조 순환이 훨씬 더 복잡할 수 있다는 점에 유의해야 합니다. 순환은 여러 클래스에 걸쳐 발생할 수 있으므로 찾고 수정하기가 더 어렵습니다.

해결책은 **weak 포인터**(`std::weak_ptr<It;T>`)를 도입하여 순환을 끊는 것입니다. **weak 포인터**는 종속된 객체가 더 이상 사용되지 않을 때 제거되는 것을 막지 않는 선택적 의존성입니다. 의존성에 접근하려면 더 이상 존재가 보장되지 않으므로 항상 존재하는지 확인해야 합니다.

```
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

또 다른 방법은 클래스 A와 B가 의존하는 공유 리소스를 보유하는 세 번째 클래스 C를 도입하여 순환을 피하는 것입니다. 이 리팩토링 기술은 JavaScript 코드베이스에서 일반적으로 사용됩니다.

해제되지 않은 리소스

가비지 컬렉터 메커니즘을 사용하는 언어에서는 리소스를 해제하지 않음으로써 종종 메모리 누수를 발생시킬 수 있습니다. 이는 리스너 패턴을 사용하고 제거 호출을 잊었을 때 관찰할 수 있습니다. JavaScript를 사용해 본 적이 있다면 이미 잘 알고 있을 것입니다. 이 일반적인 문제를 보여주는 예시를 살펴보겠습니다. 먼저, 리스너를 등록하고 호출하는 역할을 하는 간단한 **DataManager** 클래스를 만듭니다.

```
// 간단한 데이터 매니저 클래스입니다
interface DataListener {
    fun onDataChanged(data: String)
}

class DataManager {
    // 등록된 모든 리스너를 저장하는 리스트입니다.
    private val listeners = mutableListOf<DataListener>()

    // 리스너를 등록하는 메서드입니다.
    fun registerListener(listener: DataListener) {
        listeners.add(listener)
    }

    // 리스너를 등록 해제하는 메서드입니다.
    fun unregisterListener(listener: DataListener) {
        listeners.remove(listener)
    }

    // 데이터 변경 사항을 리스너에게 알리는 메서드입니다.
    fun notifyDataChanged(data: String) {
        listeners.forEach { it.onDataChanged(data) }
    }
}
```

그런 다음 이 클래스를 사용할 때 리스너 등록 해제를 잊지 않도록 주의해야 합니다. 그렇지 않으면 메모리 누수가 발생할 것입니다.

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

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

    init {
        // 리스너를 등록했지만 등록 해제를 하지 않았습니다.
        dataManager.registerListener(listener)
    }

    // 메모리 누수: 리스너를 등록 해제할 방법이 없습니다.
    // MyClass 인스턴스는 영원히 가비지 컬렉션되지 않을 것입니다.
    // DataManager가 리스너에 대한 강력한 참조를 보유하고 있기 때문입니다.
}
```

Kotlin에는 결정적인 소멸자나 deinitializer가 내장되어 있지 않으므로, AutoCloseable과 같은 정리 생명 주기를 사용해야 합니다.

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

DataManager 클래스에서 콜백을 저장하기 위해 WeakReference를 사용할 수도 있으며, 이는 이 문제를 자동으로 처리해 줄 것입니다.

```
class DataManager {  
    // 등록된 모든 리스너를 저장하는 리스트입니다.  
    private val listeners = mutableListOf<WeakReference<DataListener>>()  
  
    // ...  
}
```

수동 오버라이드 사용

자동 메모리 관리를 제공하는 언어는 일반적으로 외부 언어와의 상호 운용성을 위해 동작을 수동으로 변경하는 방법도 제공합니다. 편리하지만 잘못 사용하면 찾기 어려운 메모리 누수 원인이 될 수 있습니다.

예를 들어 Swift에서는 Unmanaged 타입을 사용하여 참조 카운트를 수동으로 증가시키거나 감소시킬 수 있습니다. 이 작업을 수행할 때는 충돌 및 예기치 않은 동작으로 이어질 수 있으므로 각별히 주의해야 합니다. Unmanaged 클래스는 ARC 동작을 수동으로 관리하는 데 사용할 수 있는 몇 가지 함수를 제공합니다.

- **passRetained** Unmanaged 인스턴스를 생성하고 참조 카운트를 증가시킵니다.
- **passUnretained** 참조 카운트를 증가시키지 않고 Unmanaged 인스턴스를 생성합니다.
- **takeRetainedValue** 객체를 가져오고 참조 카운트를 감소시킵니다.
- **takeUnretainedValue** 참조 카운트에 영향을 주지 않고 객체를 가져옵니다.

Swift 클래스에서 Unmanaged를 사용하는 예시를 살펴보겠습니다 (대부분의 경우 C와의 인터페이스에 사용됩니다).

```
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
```

하지만 참조 카운터를 0 아래로 감소시키면 충돌이 발생합니다!

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

경험적으로 볼 때, `passRetained`를 호출했다면 나중에 반드시 `takeRetainedValue`를 호출해야 하고, 그 반대의 경우도 마찬가지입니다. 이렇게 짹을 맞춰서 호출하는 것이 좋습니다. 또한 관리되지 않는 객체에 대해 `toOpaque()`를 호출하여 원시 포인터를 얻을 수 있으며, 이 포인터는 나중에 C/C++로 전달될 수 있습니다.

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

이제 여러 언어에서 사용되는 일반적인 메모리 관리 패턴을 알게 되었습니다. 또한 Swift, Kotlin 및 C++이 메모리를 어떻게 관리하는지 이해하셨을 것입니다. 다음 장에서는 내장 프로파일러 도구를 사용하여 메모리 누수를 감지하는 방법을 알아보겠습니다.

GUIDE

터보 모듈과 Fabric의 스레딩 모델 이해

React Native가 여러분의 코드를 어떻게 실행하는지 이해하는 것은 효율적인 네이티브 모듈을 구축하는 데 매우 중요합니다. 이 장에서는 네이티브 모듈의 생명 주기를 초기화 단계부터 호출 및 소멸 단계까지 자세히 살펴보겠습니다. 이 장의 끝부분에서는 어떤 상황에서 어떤 스레드가 사용되는지 이해하게 될 것입니다.

스레딩 모델

모바일 React Native 앱에서 사용할 수 있는 스레드부터 살펴보겠습니다.

- **메인 스레드 / UI 스레드** - React Native 앱이든 일반 네이티브 앱이든, 이 스레드는 UI 작업을 처리하고 반응성이 뛰어난 사용자 경험을 유지합니다.
- **JavaScript 스레드** - 이름에서 알 수 있듯이 (전적으로는 아니지만) JavaScript 코드를 실행하는 데 사용됩니다.
- **네이티브 모듈 스레드** - React Native가 네이티브 모듈을 위해 특별히 할당한 공유 풀입니다.

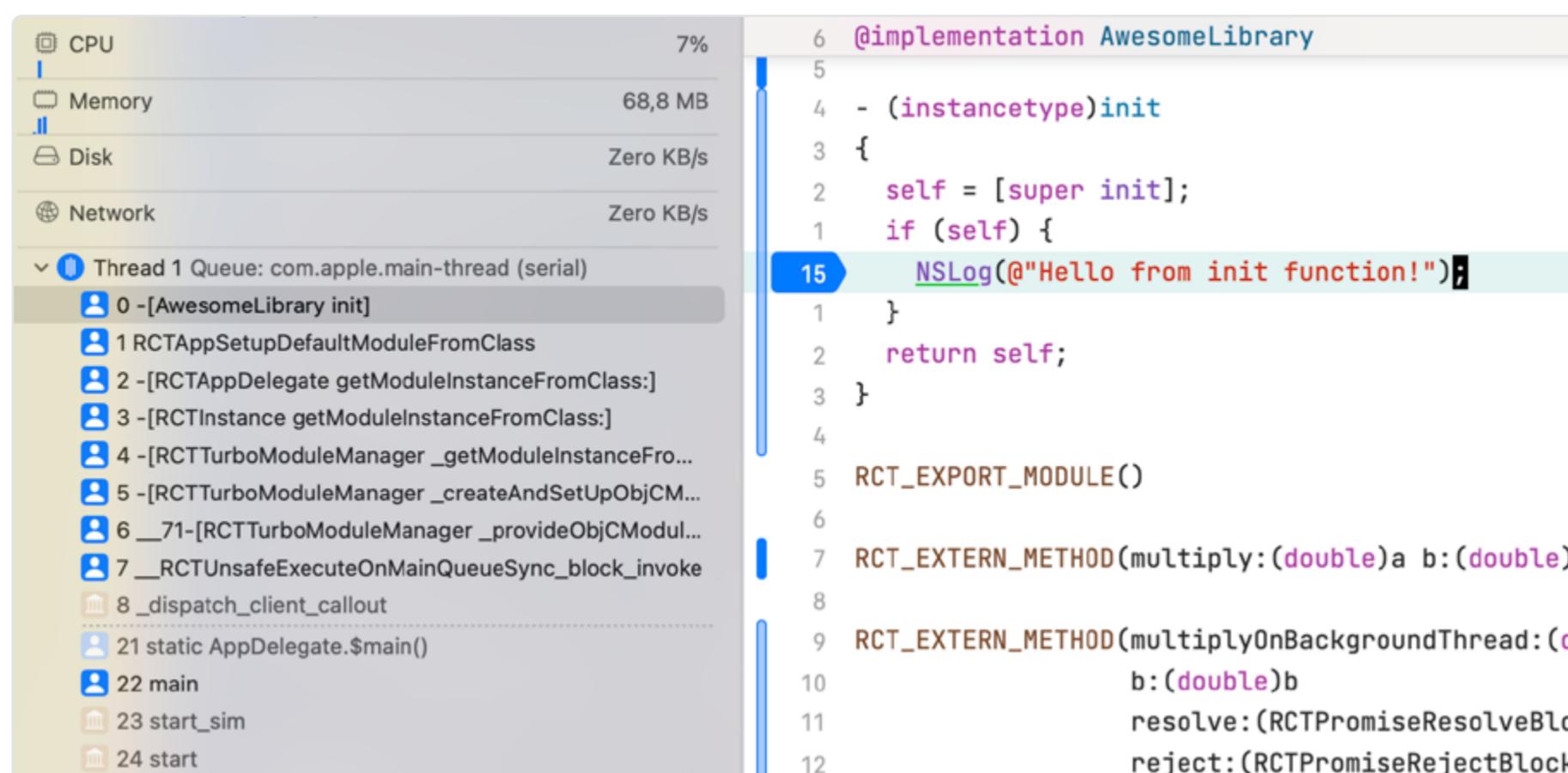
추가적으로 여러분, React Native의 렌더러 및 서드파티 모듈은 백그라운드에서 작업을 처리하기 위해 추가 스레드를 생성할 수 있습니다.

이 주제에 대한 자세한 내용은 ["네이티브 모듈 더 빠르게 만들기"](#) 장에서 확인할 수 있습니다.

터보 모듈

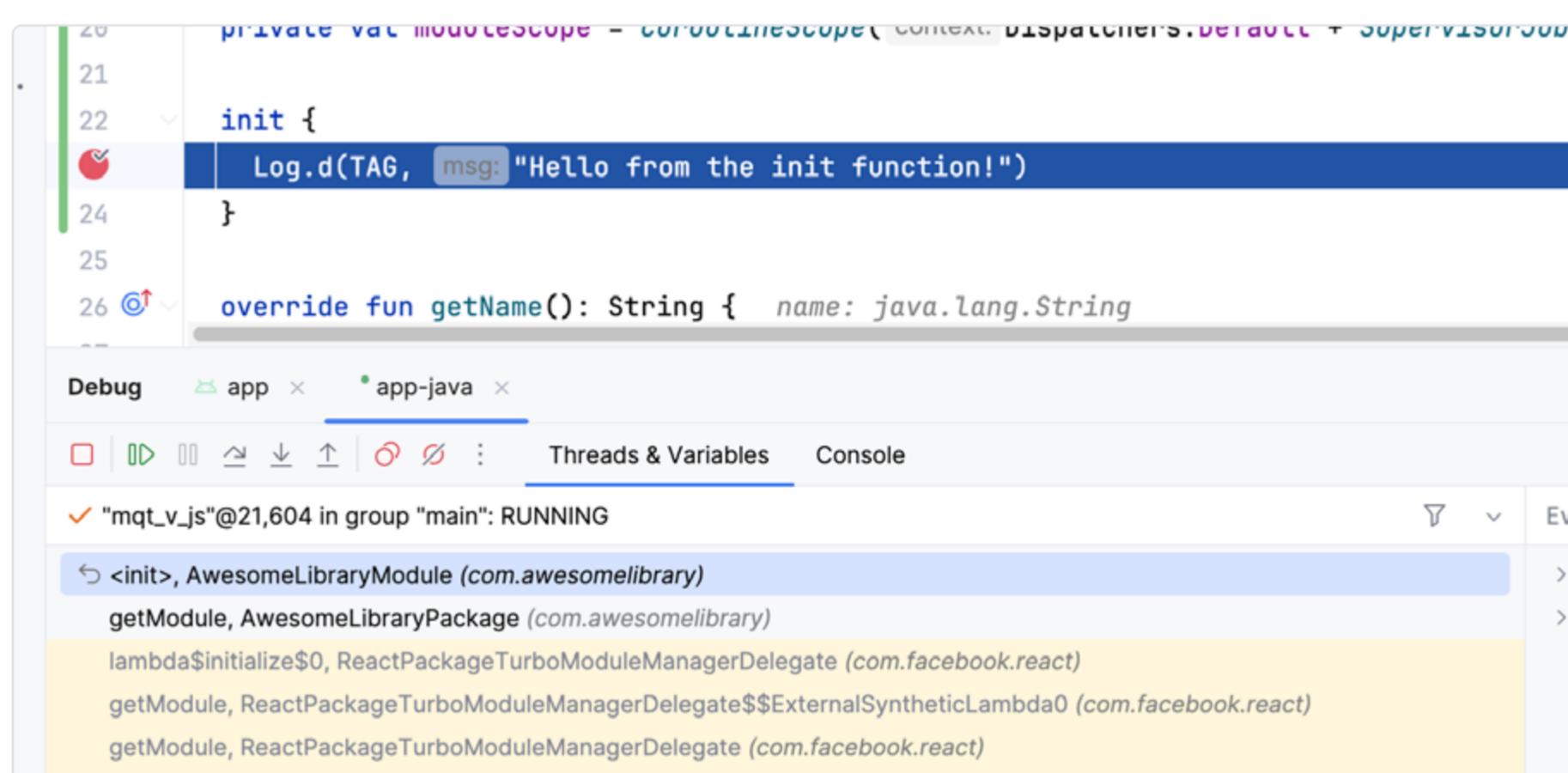
터보 모듈의 스레딩 모델을 더 잘 이해하기 위해 특정 메서드가 어떻게 그리고 어떤 스레드에서 호출되는지 확인하여 터보 모듈의 생명 주기를 살펴보겠습니다.

먼저 iOS에서 init 함수가 어떤 스레드에서 호출되는지 확인해 보겠습니다.



Xcode 디버거에서 init이 메인 스레드에서 호출되는 스크린샷

그리고 Android에서는 다음과 같습니다.



Android Studio 디버거에서 init이 JavaScript 스레드에서 호출되는 스크린샷

플랫폼 간에 주목할 만한 차이가 있습니다. Android에서는 init 메서드가 `mqt_v_js`라는 JavaScript 스레드에서 호출된 반면, iOS는 메인 스레드를 사용했습니다. 이러한 차이는 React Native가 모듈이 init 메서드를 오버라이드할 것이라는 가정에서 비롯됩니다. iOS 앱에서 init 함수를 오버라이드할 때마다 React Native는 UIKit에 접근할 수 있다고 가정합니다. 따라서 (UIKit은 스레드 안전하지 않으므로) 메인 큐에서 해당 함수를 호출합니다.

```
/*
 * 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;
```

메인 스레드에서 init을 실행하는 것에 대한 React Native 소스 코드 설명



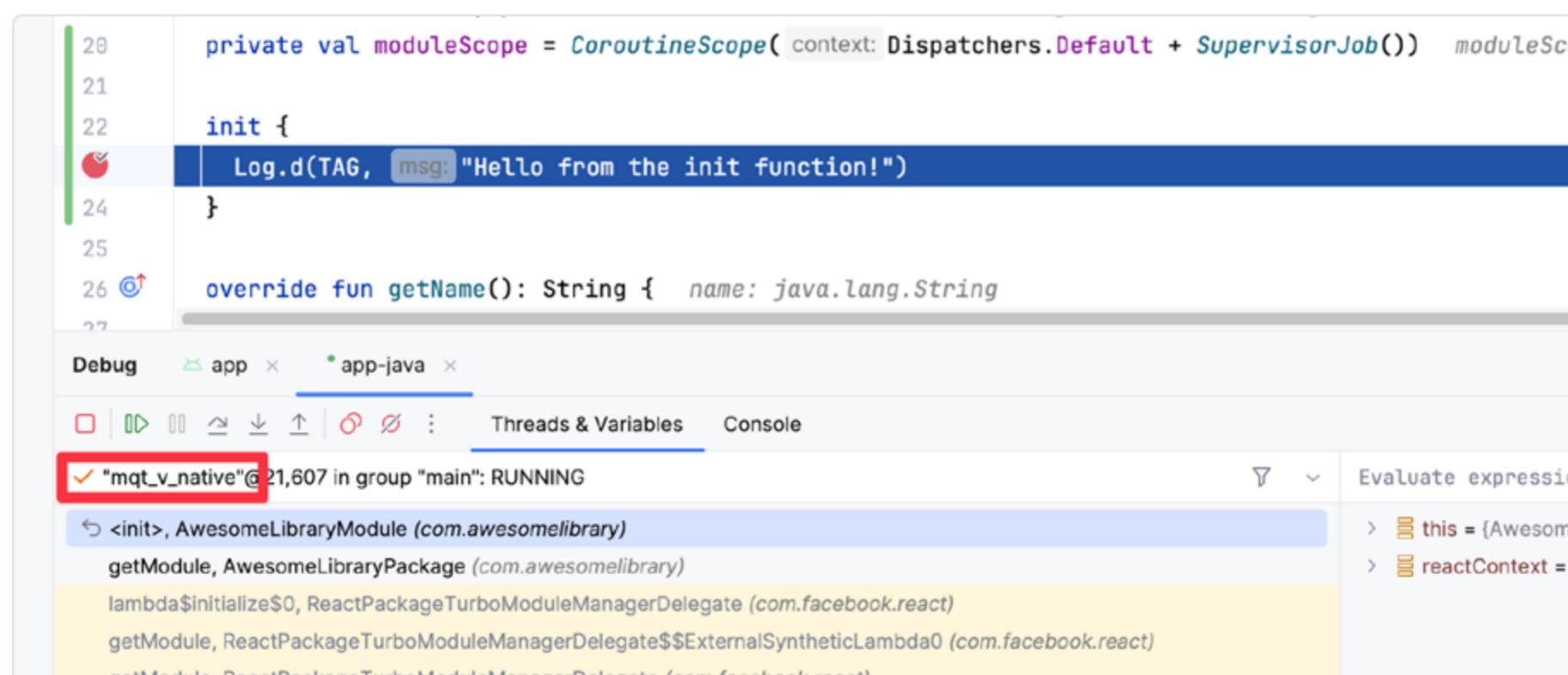
이 가정을 제거하면 모듈은 Android와 동일한 동작 방식으로 JavaScript 스레드에서 초기화됩니다. React Native 내부를 변경하려면 소스에서 빌드해야 합니다. 프로덕션 앱에는 권장되지 않지만, 개발 환경에서는 귀중한 학습 경험이 될 수 있습니다.

다음으로, 자연 초기화를 해제하는 것이 우리 모듈이 사용하는 스레드를 변경하는지 살펴보겠습니다. 이 기능은 Android에서만 사용할 수 있습니다. 즉시 로딩을 선택하려면 라이브러리의 패키지 파일을 열고 네 번째 매개변수인 `needsEagerInit`을 `true`로 변경하세요.

```
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 <- true로 바꾸기
                false, // isCxxModule
                true // isTurboModule
            )
            moduleInfos
        }
    }
}
```

플래그를 변경하고 앱을 다시 빌드한 후 어떤 일이 발생하는지 살펴보겠습니다.

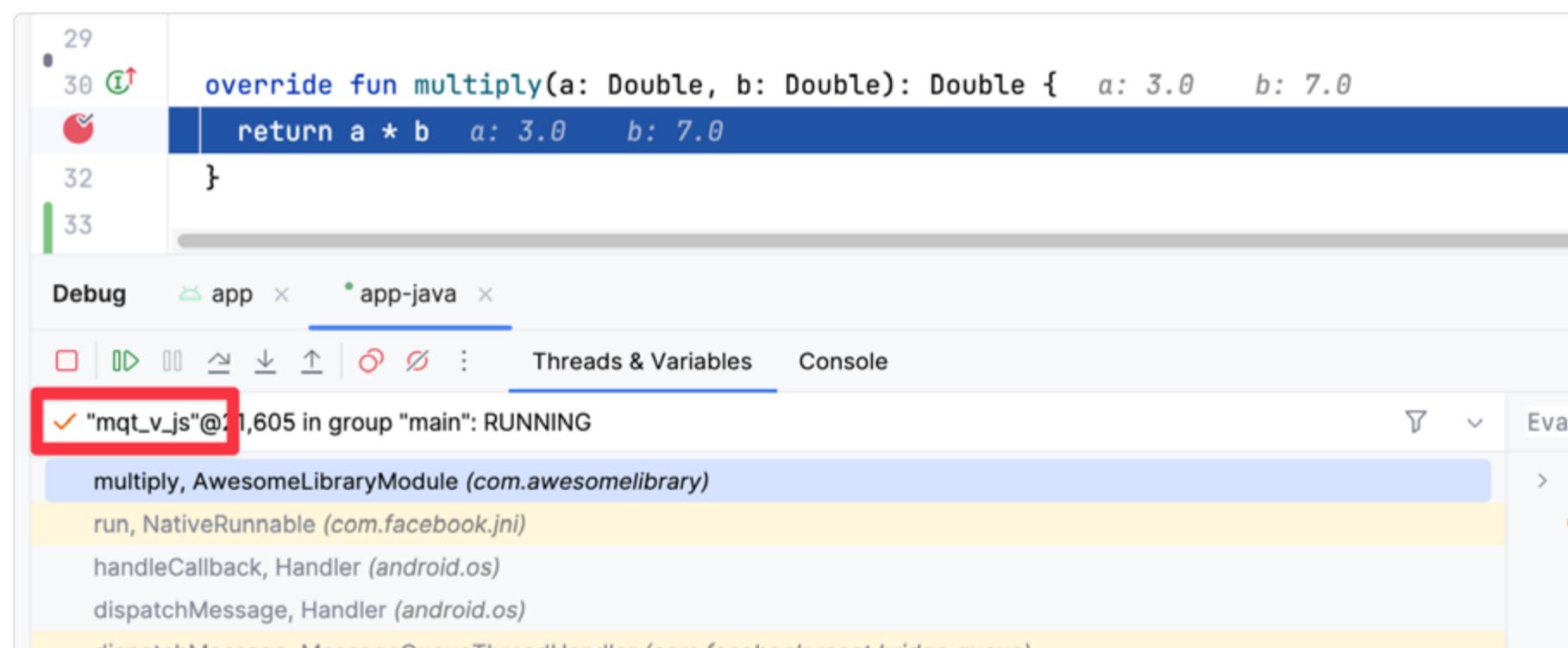


Android Studio 디버거에서 init이 터보 모듈 스레드에서 호출되는 스크린샷

예상대로 이제 다른 스레드에 있습니다. `mqt_v_native`라는 터보 모듈 전용의 별도 스레드입니다. 여기서 JavaScript 인스턴스에 접근할 계획이라면 이 점을 염두에 두는 것이 좋습니다. 이 경우 JavaScript 스레드에서 함수 호출을 예약하기 위해 JavaScript CallInvoker를 사용해야 합니다.

동기 함수 호출

초기화 과정이 끝났으니, React Native가 네이티브 메서드를 어떻게 호출하는지 살펴보겠습니다. 이 경우 동기 방식과 비동기 방식의 두 가지 범주로 나눌 수 있습니다. 먼저 `multiply`와 같은 동기 메서드를 호출하면 어떤 일이 발생하는지 확인해 보겠습니다.



JavaScript 스레드에서 호출되는 것을 확인할 수 있습니다. 생각해보면 이치에 맞습니다. 네이티브 함수의 결과 값을 받을 때까지 해당 스레드의 실행을 멈추고 기다려야 하기 때문입니다. 보시다시피 동기 함수를 사용할 때는 각별한 주의가 필요합니다. 만약 여러분이 무심코 JavaScript 스레드에서 이 별것 아닌 것처럼 보이는 함수를 호출하도록 내버려 둔다면...

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

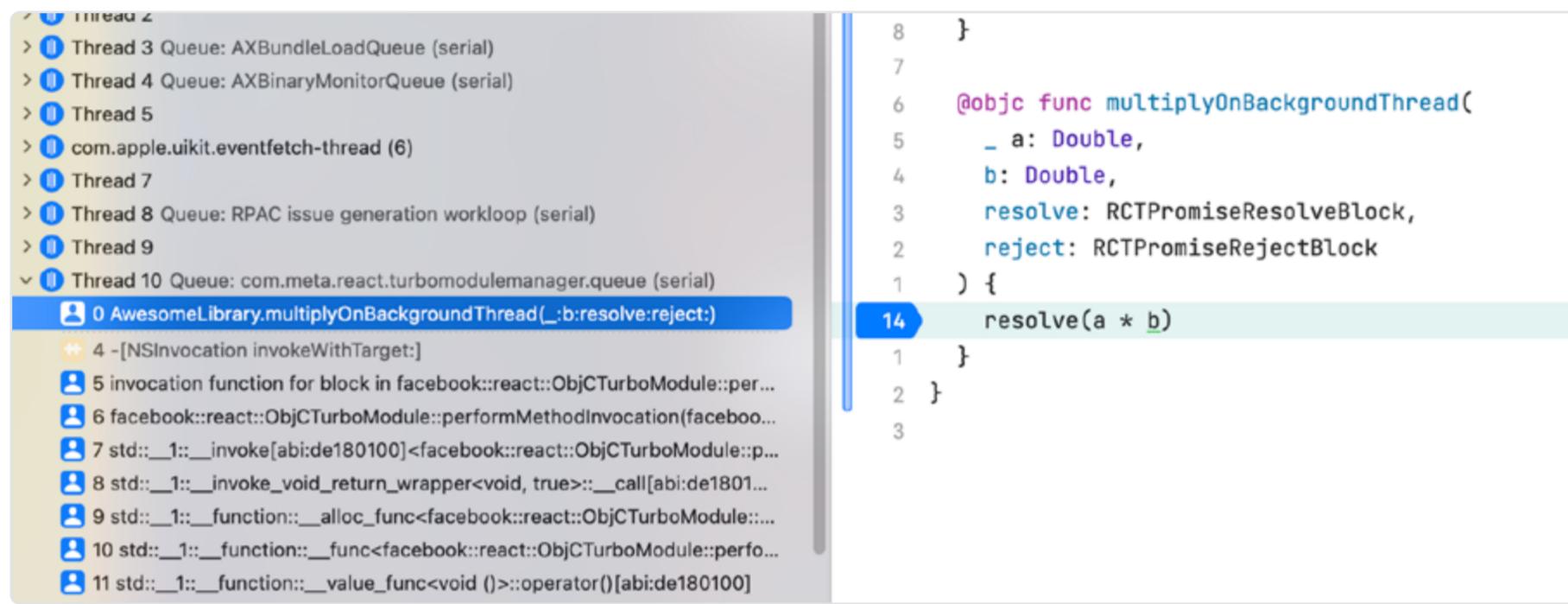
Thread.sleep은 그다지 순진해 보이지 않네요.

20초 동안 JavaScript 스레드의 모든 상호 작용을 차단하여 사실상 앱 전체를 멈추게 될 것입니다.

비동기 함수 호출

이제 비동기 함수가 어떻게 호출되는지 살펴보겠습니다. 웹 환경과 마찬가지로, `fetch`와 같은 함수를 `await`하면 해당 호출은 브라우저의 Web API (네이티브 코드 레이어)로 위임되어 요청을 처리합니다. 이 요청은 React Native에서는 네이티브 모듈 스레드인 메인 JavaScript 실행 스레드 외부에서 처리됩니다.

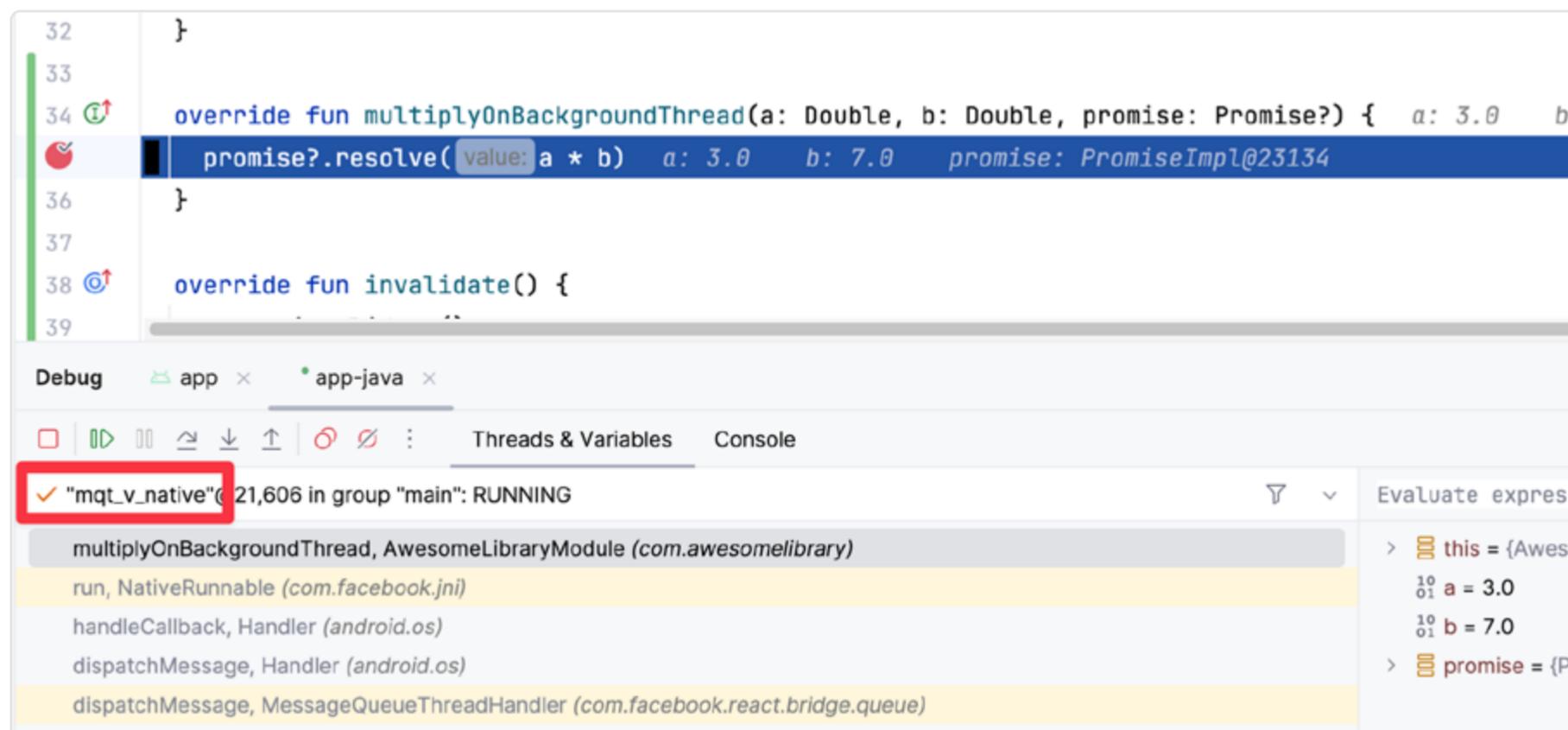
`multiplyOnBackgroundThread`라는 비동기 함수의 실제 동작을 살펴보겠습니다.



The screenshot shows the Xcode debugger's call stack for the `multiplyOnBackgroundThread` function. The stack trace is as follows:

```
Thread 2
> Thread 3 Queue: AXBundleLoadQueue (serial)
> Thread 4 Queue: AXBinaryMonitorQueue (serial)
> Thread 5
> com.apple.uikit.eventfetch-thread (6)
> Thread 7
> Thread 8 Queue: RPAC issue generation workloop (serial)
> Thread 9
> Thread 10 Queue: com.meta.react.turbomodulemanager.queue (serial)
    0 AwesomeLibrary.multiplyOnBackgroundThread(_:b:resolve:reject:)
        4 -[NSInvocation invokeWithTarget:]
        5 invocation function for block in facebook::react::ObjCTurboModule::per...
        6 facebook::react::ObjCTurboModule::performMethodInInvocation(faceboo...
        7 std::__1::__invoke[abi::de180100]<facebook::react::ObjCTurboModule::p...
        8 std::__1::__invoke_void_return_wrapper<void, true>::__call[abi::de1801...
        9 std::__1::__function::__alloc_func<facebook::react::ObjCTurboModule::...
        10 std::__1::__function::__func<facebook::react::ObjCTurboModule::perfo...
        11 std::__1::__function::__value_func<void ()>::operator() [abi::de180100]
```

Xcode 디버거에서 `multiplyOnBackgroundThread`가 터보 모듈 스레드에서 호출되는 스크린샷



The screenshot shows the Android Studio debugger's call stack for the `multiplyOnBackgroundThread` function. The stack trace is as follows:

```
32 }
33
34 ⚡ override fun multiplyOnBackgroundThread(a: Double, b: Double, promise: Promise?) { a: 3.0 b:
35     promise?.resolve(value: a * b) a: 3.0 b: 7.0 promise: PromiseImpl@23134
36 }
37
38 ⚡ override fun invalidate() {
39 }
```

The "Threads & Variables" tab is selected, showing the current thread: "mqt_v_native" (21,606 in group "main": RUNNING). The "Evaluate expression" panel shows the variables: `this = {AwesomeLibraryModule}`, `a = 3.0`, `b = 7.0`, and `promise = {PromiseImpl@23134}`.

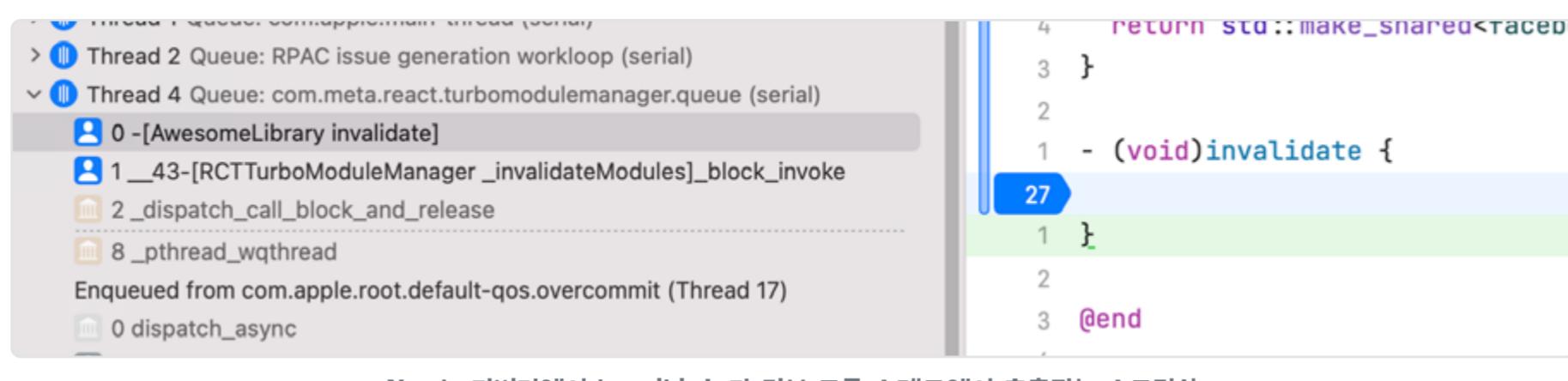
Android Studio 디버거에서 `multiplyOnBackgroundThread`가 터보 모듈 스레드에서 호출되는 스크린샷

보시다시피 React Native는 JavaScript 스레드의 부담을 덜어주고 터보 모듈 스레드에서 함수를 호출합니다. 그러나 이 스레드는 모든 네이티브 모듈에서 공유되므로 바쁠 수 있습니다. 이 경우 문제를 완화하기 위해 항상 새 스레드를 생성할 수 있습니다.

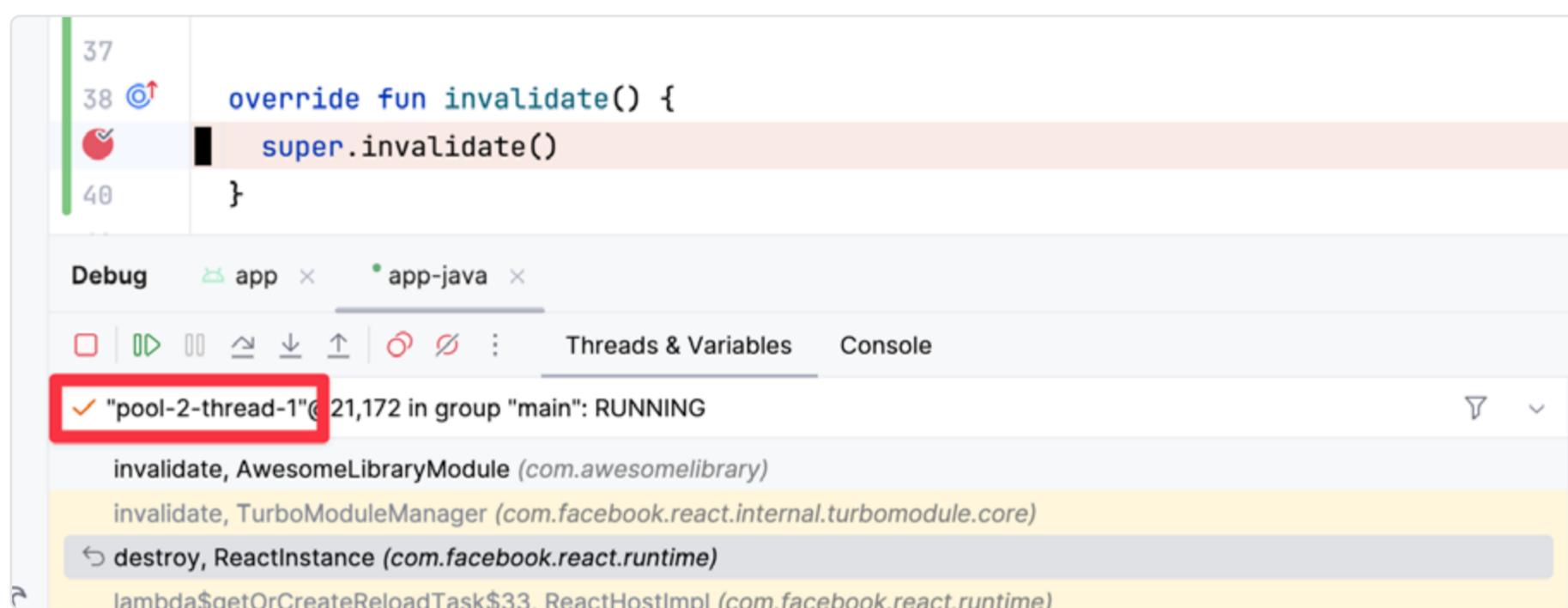
마지막으로 모듈이 어떻게 무효화되는지 확인해 보겠습니다. iOS에서 무효화를 지원하려면 모듈이 `RCTInvalidating` 프로토콜을 준수해야 합니다.



무효화는 React Native 인스턴스가 해체될 때 발생합니다. 예를 들어 Metro 서버를 다시 로드할 때 그렇습니다.



Xcode 디버거에서 invalidate가 터보 모듈 스레드에서 호출되는 스크린샷



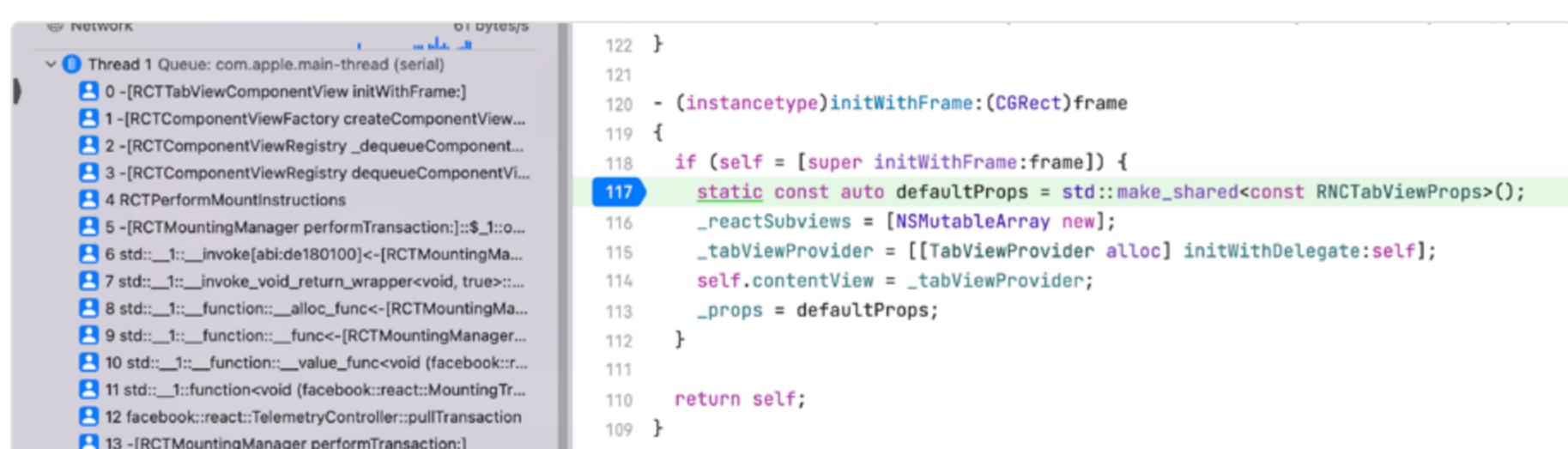
Android Studio 디버거에서 "ReactHost" 스레드인 pool-2-thread-1에서 invalidate가 호출되는 스크린샷

플랫폼 간에 또 다른 작은 차이가 있습니다. iOS에서는 함수가 터보 모듈 스레드에서 호출되고, Android에서는 스레드에 명시적인 이름이 없고 자동으로 생성된 것으로 보입니다. 소스 코드를 살펴보면 `ReactHost` 클래스에서 생성한 스레드임을 알 수 있습니다. 이제 터보 모듈의 스레딩에 대해 모두 알았으니, Fabric 뷰가 어떻게 작동하는지 살펴보겠습니다.

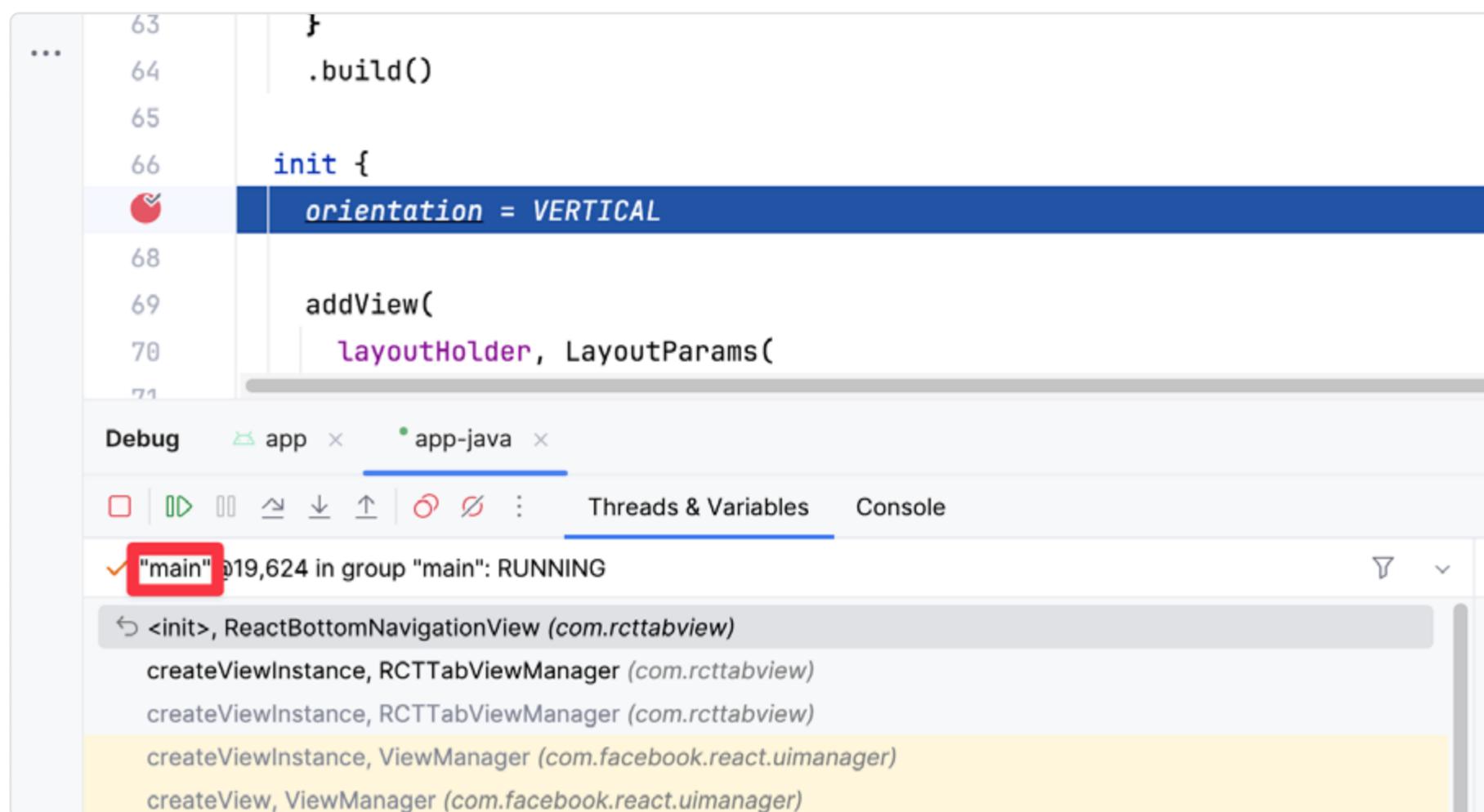
네이티브 뷰

네이티브 뷰 레이어(내부적으로 Fabric이라고 함)의 스레딩 모델을 이해하는 것은 훨씬 쉽습니다. 앞서 언급했듯이, 운영 체제(iOS든 Android든)는 메인 스레드에서 뷰를 업데이트할 것으로 예상합니다. 또한 iOS UI 라이브러리인 UIKit은 스레드 안전하지 않으므로, 뷰를 생성하고 조작하는 거의 모든 작업이 메인 스레드에서 이루어진다는 것을 배웠습니다.

탐색을 시작해 보겠습니다. 먼저 뷰가 어떻게 초기화되는지 확인해 보겠습니다.



Xcode 디버거에서 사용자 지정 네이티브 컴포넌트의 init 메서드가 메인 스레드에서 호출되는 스크린샷



The screenshot shows the Android Studio debugger interface. The code editor at the top has several lines of Java code, with the line containing 'orientation = VERTICAL' highlighted in blue. Below the code editor is a toolbar with tabs for 'Debug', 'app', and 'app-java'. The 'Threads & Variables' tab is selected. A list of threads is shown, with the 'main' thread highlighted and a red box drawn around it. The stack trace for the main thread is displayed below, showing the call chain from 'ReactBottomNavigationView' down to 'ViewManager'.

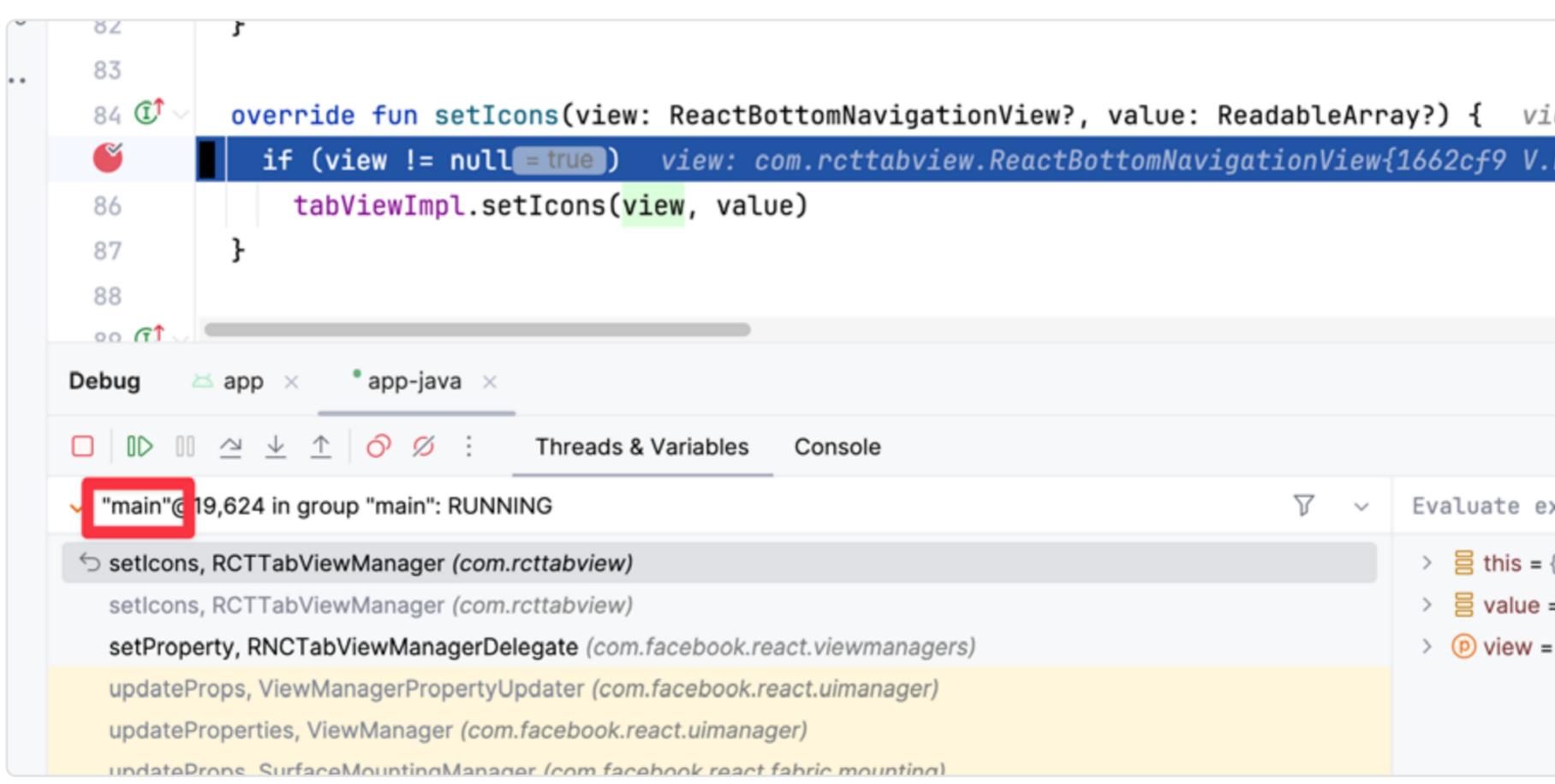
Android Studio 디버거에서 사용자 지정 네이티브 컴포넌트의 init 메서드가 메인 스레드에서 호출되는 스크린샷

보시다시피 양쪽 플랫폼 모두에서 뷰가 메인 스레드에서 초기화됩니다. 이제 props 업데이트를 처리하는 스레드를 확인해 보겠습니다. JavaScript 측에서는 항상 JavaScript 스레드에서 props를 조작하지만, 네이티브 측에서는 어떤 일이 발생하는지 살펴보겠습니다.



The screenshot shows the Xcode debugger interface. On the left, there's a network log with a single entry for 'Thread 1'. On the right, the code editor shows a C++ file with several lines of code. The line containing 'const auto &oldViewProps = *std::static_pointer_cast<RNCTabViewProps const>(_props);' is highlighted in green. Below the code editor is a toolbar with tabs for 'Network', 'Threads & Variables', and 'Console'. The 'Threads & Variables' tab is selected. A list of threads is shown, with the 'main' thread highlighted and a red box drawn around it. The stack trace for the main thread is displayed below, showing the call chain from 'updateProps' down to '_tabViewProvider.itemsData'.

Xcode 디버거에서 updateProps 메서드가 메인 스레드에서 호출되는 스크린샷



The screenshot shows the Android Studio debugger interface. The code editor at the top has several lines of Java code, with the line containing 'if (view != null) view: com.rcttabview.ReactBottomNavigationView[1662cf9]' highlighted in blue. Below the code editor is a toolbar with tabs for 'Debug', 'app', and 'app-java'. The 'Threads & Variables' tab is selected. A list of threads is shown, with the 'main' thread highlighted and a red box drawn around it. The stack trace for the main thread is displayed below, showing the call chain from 'setIcons' down to 'updateProps'.

Android Studio 디버거에서 props 설정자가 메인 스레드에서 호출되는 스크린샷

다시 메인 스레드에 있습니다! React Native는 props를 업데이트하는 함수가 Yoga를 사용하여 네이티브 플랫폼 뷰를 직접 조작할 것이라고 가정하기 때문입니다.

Yoga는 **React Native** 렌더링 파이프라인의 핵심적인 부분입니다. 절대 위치 지정 및 Flexbox 레이아웃 모델을 기반으로 다양한 플랫폼에서 일관된 레이아웃을 계산하고 제공하는 크로스 플랫폼 레이아웃 엔진입니다. **React Native** 렌더러는 **Yoga**를 사용하여 C++로 작성된 **React** 컴포넌트 트리의 내부 표현인 **React Shadow Tree**의 레이아웃 정보를 계산합니다.



Yoga는 **React Native**에만 국한된 것이 아니라, 동일한 이유로 네이티브 앱에서도 자주 사용됩니다. 일부 벤치마크에서는 iOS의 Auto Layout과 같은 네이티브 뷰 레이아웃 시스템보다 빠르기도 합니다.

보시다시피 **Yoga** 트리에 관련된 작업은 **JavaScript** 스레드에서 실행됩니다.

```
Thread 35
> Thread 37 Queue: RPAC issue generation workloop (serial)
> Thread 38
> Thread 39
> Thread 40
> com.facebook.react.runtime.JS (41)
  0 facebook::yoga::Node::measure
    1 facebook::yoga::measureNodeWithMeasureFunc
    2 facebook::yoga::calculateLayoutImpl
    3 facebook::yoga::calculateLayoutInternal
    4 facebook::yoga::computeFlexBasisForChild
    5 facebook::yoga::computeFlexBasisForChildren
    6 facebook::yoga::calculateLayoutImpl
    7 facebook::yoga::calculateLayoutInternal
    8 facebook::yoga::computeFlexBasisForChild

51 }
52
53 YGSize Node::measure(
54   float availableWidth,
55   MeasureMode widthMode,
56   float availableHeight,
57   MeasureMode heightMode) {
58   const auto size = measureFunc_(
      this,
      availableWidth,
      unscopedEnum(widthMode),
      availableHeight,
      unscopedEnum(heightMode));
59
60
61
62
63
64
```

Xcode 디버거에서 `yoga::Node::measure`가 JavaScript 스레드에서 호출되는 스크린샷

이제 어떤 것이 어떤 스레드에서 호출되는지, 그리고 그 이유는 무엇인지 스레딩 모델을 더 잘 이해하셨으므로, **React Native** 앱에 고급스럽고 스레드 안전하며 성능이 뛰어난 네이티브 기능을 추가할 수 있을 것입니다.

BEST PRACTICE

뷰 평면화를 통한 최적화

React 컴포넌트 API는 선언적이고 컴포지션을 통해 재사용 가능하도록 설계되었습니다. 레고 블록처럼 여러 컴포넌트를 결합하여 복잡한 레이아웃을 만듭니다. 웹에서는 `<div>`를 만드는 것이 매우 저렴하기 때문에 이는 훌륭하게 작동합니다. 하지만 React Native는 네이티브 레이아웃 기본 요소 위에서 작동하며, 이는 일반적으로 웹의 동등 요소보다 처리 비용이 더 많이 듭니다. 이러한 문제를 완화하기 위해 React Native는 핵심 렌더러에 "뷰 평면화"라는 최적화 기술을 도입했습니다. 이는 가능한 경우 뷰 계층 구조를 단순화하여 불필요한 메모리 부담과 CPU 처리를 줄입니다.



이러한 종류의 최적화는 Android React Native 렌더러에 처음 도입되었습니다. 그러나 새로운 아키텍처로 프로젝트가 진행되면서 핵심 부분을 C++로 재작성하게 되었고, 이 플랫폼별 최적화를 공유 렌더러로 이동하는 것이 합리적이었습니다. 덕분에 이제 iOS에서도 뷰 평면화가 가능합니다.

작동 방식에 대한 자세한 설명은 생략하고, 렌더러는 "레이아웃 전용" 노드를 식별합니다. 이 노드들은 View의 레이아웃에만 영향을 미치고 화면에 아무것도 렌더링하지 않습니다. 이러한 요소들은 평면화될 수 있으며, 이는 호스트 요소 트리의 깊이를 줄이는 결과를 가져옵니다. 뷰 평면화가 어떻게 작동하는지에 대한 자세한 설명은 [공식 문서를 참조하십시오](#).

뷰 평면화 시 주의 사항

이 최적화 기술은 대부분의 경우에 훌륭하게 작동하지만, 때로는 뷰가 계층 구조에 유지되기를 원할 때가 있습니다. 이러한 요구 사항의 일반적인 경우는 자식을 허용하는 네이티브 UI 컴포넌트에서 작업할 때입니다.

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

3개의 자식 뷰를 가진 MyNativeComponent

위 예시에서 **MyNativeComponent**는 네이티브 측에서 3개의 뷰 배열을 받을 것으로 예상할 수 있습니다. 하지만 뷰 평면화가 **Child1**의 컨테이너 뷰가 "레이아웃 전용"이라고 판단하면 어떻게 될까요? 예상치 않게 이 뷰의 모든 자식들이 네이티브 측으로 전달될 수 있습니다. 이는 만약 **Child1** 내부에 3개의 자식 뷰가 있었다면, 네이티브 컴포넌트로 3개가 아닌 5개의 자식이 전달된다는 의미입니다. 이를 설명하기 위해 다음을 살펴보겠습니다.

```
<MyNativeComponent>
  /* Child1이 예상치 않게 3개의 뷰로 평면화됨 */
  <View/>
  <View/>
  <View/>
  <Child2/>
  <Child3/>
</MyNativeComponent>
```

탐욕적(greedy) 뷰 평면화

이는 JavaScript 자식의 수가 네이티브 측의 자식 배열과 일치할 것이라고 가정한다면 예상치 못한 동작입니다. 보시다시피 이는 안전한 가정이 아니며 컴포넌트 로직을 검증하도록 유도할 수 있습니다. 하지만 때로는 자식의 개수를 정말로 알아야 할 때가 있습니다. 이를 해결하기 위해 뷰 평면화 동작을 제어하는 **collapsible prop**을 사용할 수 있습니다. 이를 **false**로 설정하면 해당 뷰가 다른 뷰와 병합되는 것을 방지합니다.

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

collapsible prop이 **false**로 설정된 3개의 자식 뷰를 가진 MyNativeComponent

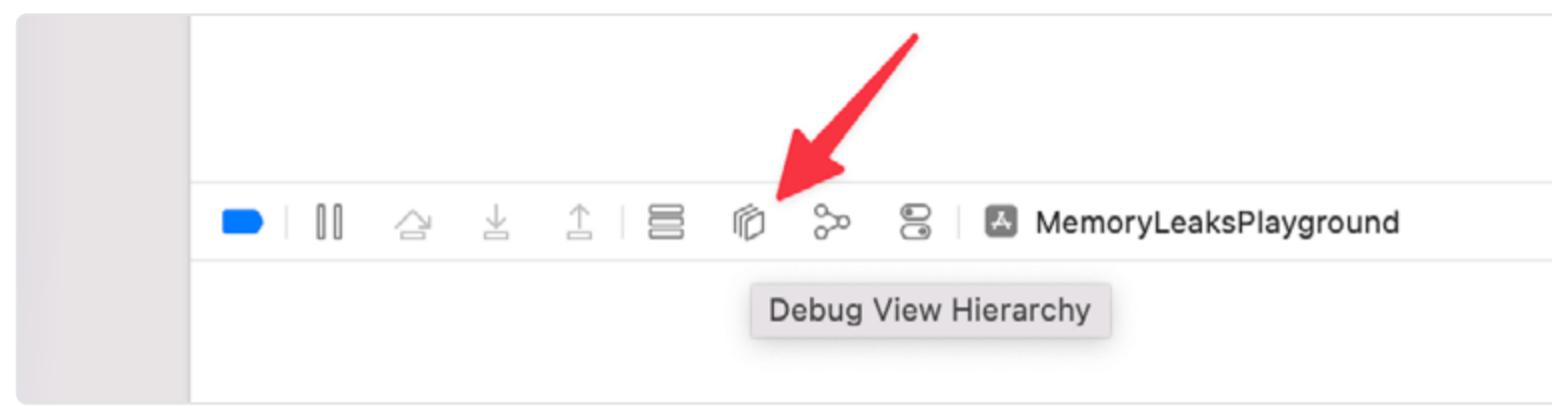
이 **prop**을 적용하면 네이티브 컴포넌트에서 항상 3개의 자식 뷰를 받게 될 것이라고 확신할 수 있습니다.

뷰 계층 구조 디버깅

뷰 평면화 작업을 할 때 뷰 계층 구조 디버깅이 유용할 수 있습니다. 다행히도 iOS와 Android 플랫폼 모두에서 각각 Xcode와 Android Studio를 사용하여 이를 검사할 수 있습니다.

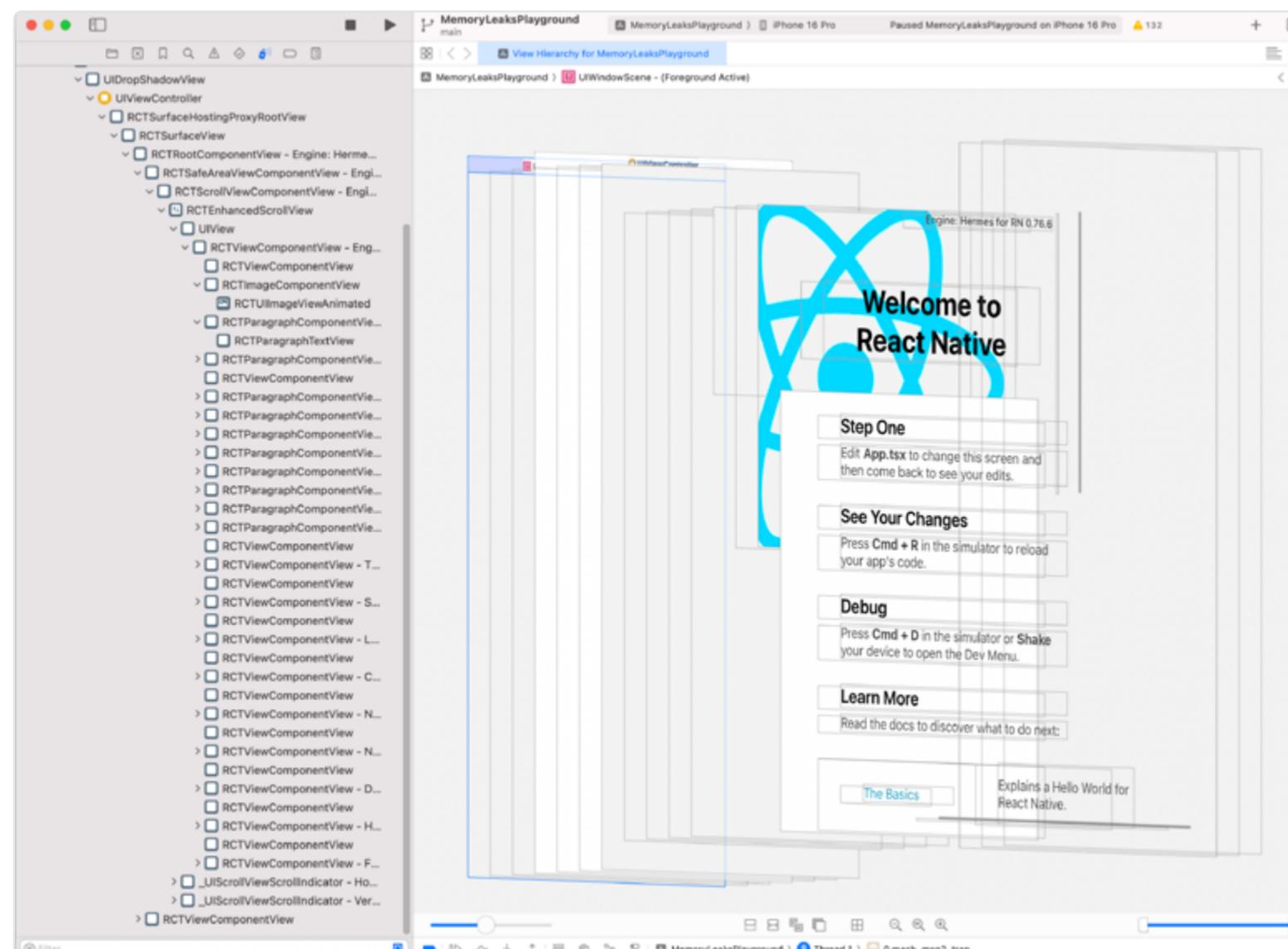
Xcode

Xcode를 통해 앱을 실행하면 표준 중단점 디버깅 외에도 계층 구조 검사기(Hierarchy Inspector)를 제공하는 디버깅 툴바를 볼 수 있습니다. "Debug View Hierarchy" 버튼을 클릭하여 실행할 수 있습니다.



Xcode의 "Debug View Hierarchy" 버튼

앱은 중단점에서 멈춘 것처럼 멈추지만, 이제 네이티브 뷰 계층 구조를 검사할 수 있습니다.

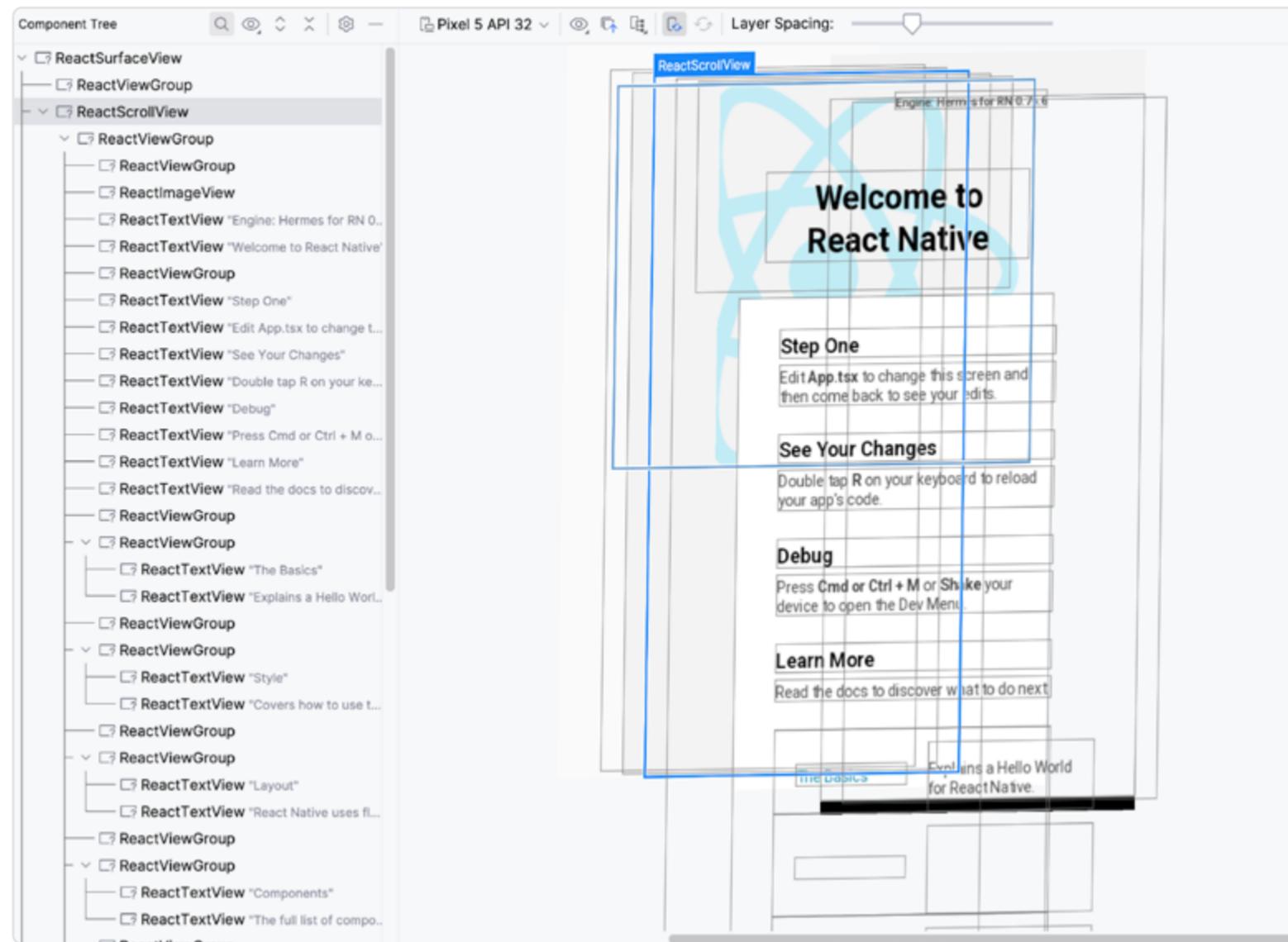


샘플 React Native 앱의 뷰 계층 구조: 정말 네이티브 앱이군요!

뷰 계층 구조를 시각적으로 표현한 것을 보는 것은 레이아웃 관련 문제를 해결하는 데 도움이 될 수 있습니다. 왼쪽 사이드바를 보면 JavaScript 컴포넌트에 해당하는 네이티브 뷰 목록이 표시됩니다. 따라서 JavaScript의 `<View/>`는 완전한 네이티브 앱 내부와 동일한 네이티브 기본 구성 요소인 `RCTViewComponentView`에 해당합니다.

Android Studio

Xcode와 마찬가지로 Android에서도 앱의 뷰 계층 구조를 검사하는 방법이 있습니다. Android에서는 이를 "Layout Inspector"라고 부릅니다. 실행하려면 상단 메뉴 바에서 View > Tool Windows > Layout Inspector를 선택하세요.



Android Studio의 레이아웃 검사기

여기 Xcode와 유사한 UI를 볼 수 있습니다. 사이드바에는 현재 화면에 보이는 요소 목록이 있습니다. 이 플랫폼에서는 JavaScript의 <View/>가 내부에 ReactTextview를 포함하는 ReactViewGroup에 해당한다는 것을 알 수 있습니다.

뷰 평면화가 어떻게 작동하는지 이해하는 것은 유용한 기술입니다. 대부분의 경우 이 기능은 완전히 투명하게 작동하도록 설계되었으므로 특별히 신경 쓸 필요가 없습니다. 하지만 때로는 이 최적화가 잘못 적용되어 문제가 발생할 수 있으며, 이때 레이아웃 디버거가 상황을 이해하는데 도움을 줄 것입니다. 그리고 필요한 경우 문제가 되는 뷰의 평면화를 건너뛸 수도 있습니다.

BEST PRACTICE

웹 대신 전용 React Native SDK를 사용하세요

React Native의 가장 큰 장점 중 하나는 JavaScript 생태계의 좋은 부분을 모바일, 태블릿, 데스크톱 앱에서 모두 사용할 수 있다는 것입니다. 이는 일부 React 컴포넌트를 재사용하고 즐겨 사용하는 라이브러리로 비즈니스 로직 상태를 관리할 수 있음을 의미합니다.

React Native는 웹과의 호환성을 위해 웹과 유사한 기능을 제공하지만, 동일한 환경이 아니라 는 것을 이해해야 합니다. React Native에는 고유한 모범 사례, 손쉬운 개선 방법 및 제약 조건이 있습니다.

전용 플랫폼별 라이브러리 버전 사용

모바일 React Native 앱에서 브라우저에서와 동일한 JavaScript를 실행하는 것이 종종 (항상 그런 것은 아니지만) 가능하다는 사실이 매번 그렇게 해야 한다는 의미는 아닙니다. 사용 편의성을 위해 종속성을 주의 깊게 살펴보고, 여전히 필요한지 또는 더 나은 플랫폼별 대안으로 대체할 수 있는지 지속적으로 확인해야 합니다.

국제화 Polyfill

Intl 객체는 언어에 민감한 문자열 비교, 숫자 형식 지정, 날짜 및 시간 형식 지정을 제공합니다. 모든 최신 브라우저에서 사용할 수 있으며, Hermes 엔진에서는 부분적으로 지원됩니다.

다음은 Intl을 사용하여 숫자를 여러 로케일로 형식 지정하는 예입니다.

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

React Native 프로젝트에서는 ECMAScript 국제화 API의 JavaScript 전용 구현을 제공하는 Format.JS 프로젝트에서 가져온 많은 Intl 폴리필 import를 종종 볼 수 있습니다.

```
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';
```

모든 가능한 Intl 폴리필을 가져오는 것이 항상 필요한 것은 아닙니다.

이 API에 대한 Hermes의 지원은 제한적이었지만, 다행히 작년에 Hermes 팀에서 개선 작업을 진행 중이라고 발표했습니다. 1년 후인 지금은 일부 폴리필을 제거할 수 있습니다! 이 가이드 작성 시점(2025년 1월) 기준으로 사용 가능한 API는 다음과 같습니다.

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	✗

이러한 새로운 정보를 바탕으로 프로젝트에 import하는 폴리필의 수를 줄일 수 있습니다.

```
-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';
```

모든 가능한 Intl 폴리필을 가져오는 것이 항상 필요한 것은 아닙니다.

이러한 폴리필만 제거해도 JS 번들 크기를 430kB 이상 줄일 수 있습니다. 이 코드는 앱의 진입 점에서 즉시 로드되므로 불필요한 코드를 제거하는 것만으로도 앱의 TTI(Time To Interactive)를 잠재적으로 개선할 수 있습니다.

암호화 라이브러리

고부하 연산과 관련된 라이브러리를 네이티브 대체재로 교체하는 것도 고려해 볼 만한 또 다른 예입니다. 한 가지 예로 Node.js crypto 라이브러리의 JavaScript 구현인 [crypto-js](#)를 Margelo의 [react-native-quick-crypto](#)로 대체할 수 있습니다. 이 교체는 C++을 직접 사용하여 최대 58배 빠른 결과를 얻을 수 있습니다.

이는 예를 들어 Web3 지갑 시드를 생성하기 위해 암호학적으로 안전한 의사 난수 생성기 (CSPRNG)를 사용해야 하고 난수에 의존하는 프로젝트에 매우 중요합니다. JavaScript의 `Math.random()`은 결과가 안전하다고 간주될 만큼 충분한 엔트로피를 포함하는지 보장할 수 없기 때문에 이 함수는 JavaScript에 의존하여 구현할 수 없습니다.

네이티브 암호화 라이브러리를 사용하여 속도와 보안을 향상시키는 방법에 대한 자세한 내용은 이 블로그 게시물을 참조하십시오.

네이티브 탐색 사용

대부분의 앱에는 탐색 솔루션이 필요하며, 가장 인기 있는 솔루션 중 하나는 [React Navigation](#)입니다. 이는 디자인 요구 사항 및 성능 요구 사항에 따라 JS 및 네이티브 탐색기를 혼합하여 사용할 수 있는 강력한 라이브러리입니다. 탐색 구조를 설정하는 동안 어떤 스택 탐색기를 선택해야 할지 궁금할 수 있습니다. 최고의 유연성을 제공하는 JS 스택일까요, 아니면 유연성은 떨어지지만 더 나은 성능과 네이티브 모양을 제공하는 네이티브 스택일까요? 탭 탐색기를 사용하는 경우에도 마찬가지입니다. 스택 탐색기와 유사한 장단점을 가진 JS 탭과 네이티브 탭 중 어떤 것을 선택해야 할까요?

네이티브 기반 탐색기를 선택하면 여러 가지 이점이 있습니다. 무엇보다도 JavaScript 스레드의 작업을 덜어 앱의 메모리 사용량을 줄이고 전반적으로 더 부드럽게 만듭니다. 또한 매우 중요한 요소인 네이티브 느낌도 있습니다. 네이티브 스택과 네이티브 탭 모두 해당 플랫폼의 다른 앱에서 사용되는 네이티브 플랫폼 기본 요소를 사용하여 앱에 추가적인 수준의 완성도를 제공합니다.

네이티브 스택 탐색기

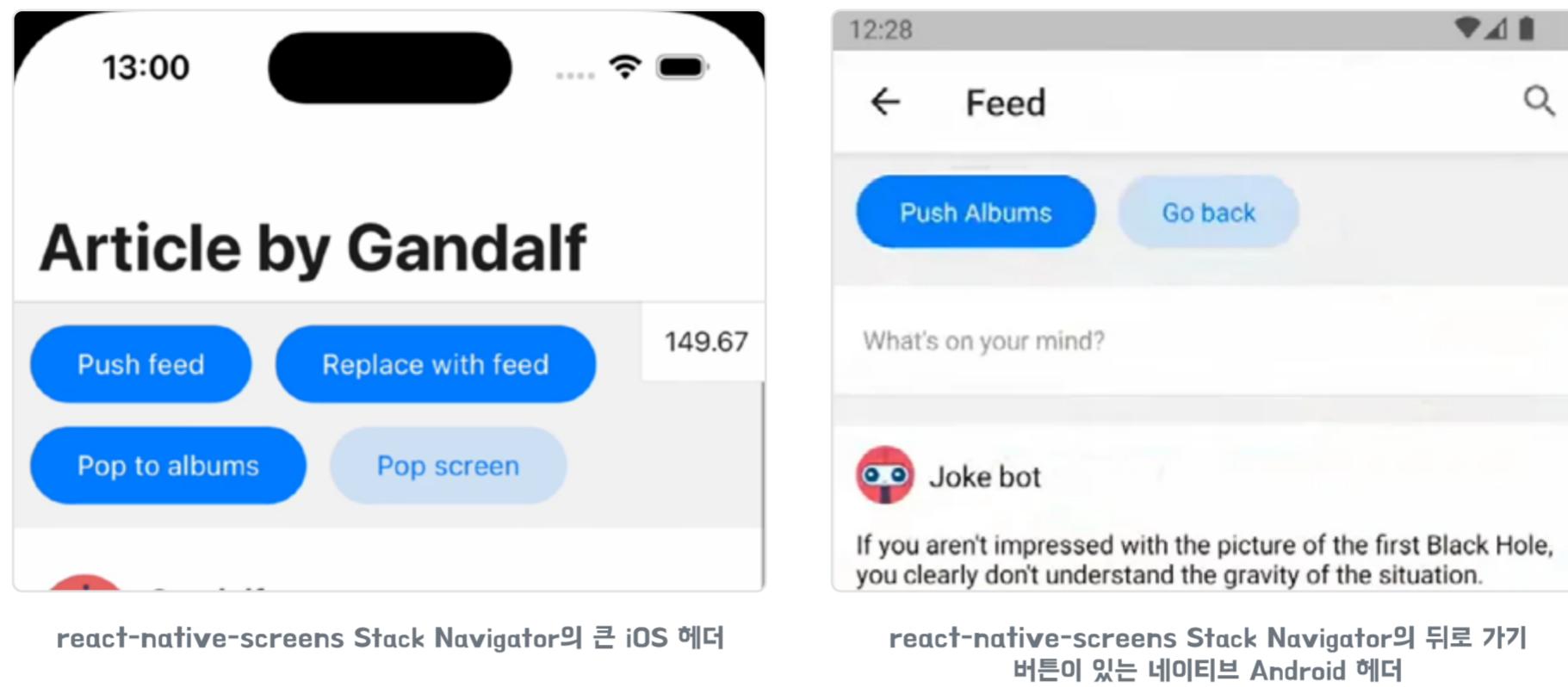
[react-native-screens](#) 프로젝트의 네이티브 스택 탐색기를 [React Navigation](#)과 함께 사용하려면 [@react-navigation/native-stack](#) 패키지를 설치하고 다음과 같이 사용하기만 하면 됩니다.

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

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

native 스택 내비게이터 초기화

해당 API는 JavaScript 스택 내비게이터와 매우 유사하므로 마이그레이션하는 데 복잡한 경우가 많지 않습니다.



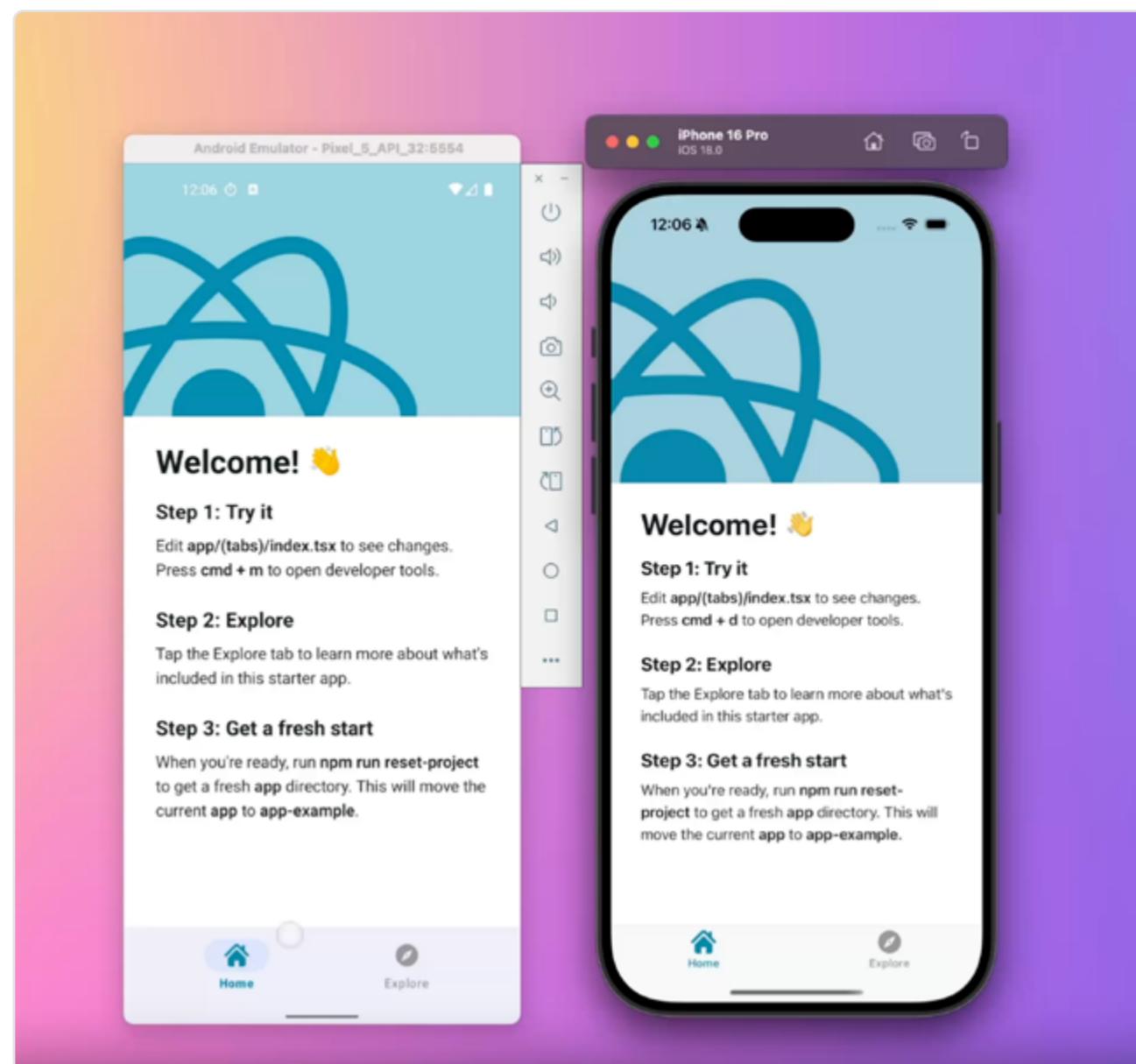
네이티브 탭 내비게이터

마찬가지로 React Navigation에서 제공하는 기본 JS 내비게이터 대신, 대부분 호환 가능한 API를 제공하는 react-native-bottom-tabs 프로젝트를 사용할 수 있습니다.

네이티브 스택과 마찬가지로 JavaScript 기반 내비게이션을 대체하고 동일한 API를 사용할 수 있습니다.

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

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



react-native-bottom-tabs의 네이티브 하단 탭

네이티브 컴포넌트를 제공하는 라이브러리 우선 사용

네이티브 컴포넌트를 노출하는 라이브러리를 최우선으로 선택해야 합니다. React Native는 네이티브 컴포넌트를 노출하는 오픈 소스 라이브러리를 개발하는 광범위한 개발자 커뮤니티를 보유하고 있습니다. 몇 가지를 살펴보겠습니다.

- React Native Screens - React Navigation의 Native Stack 기반입니다.
- Zeego - Radix UI에서 영감을 받은 React Native + Web용의 아름다운 네이티브 메뉴입니다.
- React Native Slider - 네이티브 플랫폼 기본 요소를 사용하는 슬라이더입니다.
- React Native Date Picker - 네이티브 플랫폼 기본 요소를 사용하는 날짜 선택기입니다.

이 외에도 훨씬 더 많은 라이브러리가 있습니다!

앱의 디자인 요구 사항 때문에 무한히 유연한 JavaScript 뷰를 플랫폼 네이티브 대응 요소로 항상 대체할 수 있는 것은 아닐 수 있습니다. 더 나은 성능의 UX 대안이 존재하지만 디자인 선택을 약간 조정해야 하는 경우, 사용자를 위해 목소리를 높여야 할 수도 있습니다. 훌륭한 UX는 디자이너의 창의성과 엔지니어링 관점에서 물리적으로 가능한 것 사이의 대화에 관한 것입니다. 우려 사항을 표현하는 것을 두려워하지 마세요. 결국 우리는 모두 사용자를 위해 이 모든 것을 하는 것입니다.

BEST PRACTICE

네이티브 모듈 더 빠르게 만들기

네이티브 모듈은 React Native 앱에서 구축할 수 있는 무한한 가능성을 열어줍니다. JavaScript만으로는 충분하지 않을 때 앱이 실행되는 플랫폼의 네이티브 언어인 Swift 및 Kotlin과 같은 하위 수준 언어를 사용할 수 있습니다. 더 나은 성능 또는 코드 재사용을 위해 더 낮은 수준까지 내려가야 하는 경우 C++을 사용할 수도 있습니다. 다음 네이티브 모듈을 구축하는 동안 통합할 수 있는 몇 가지 모범 사례를 살펴보겠습니다.

보일러플레이트 코드 빠르게 구성하기

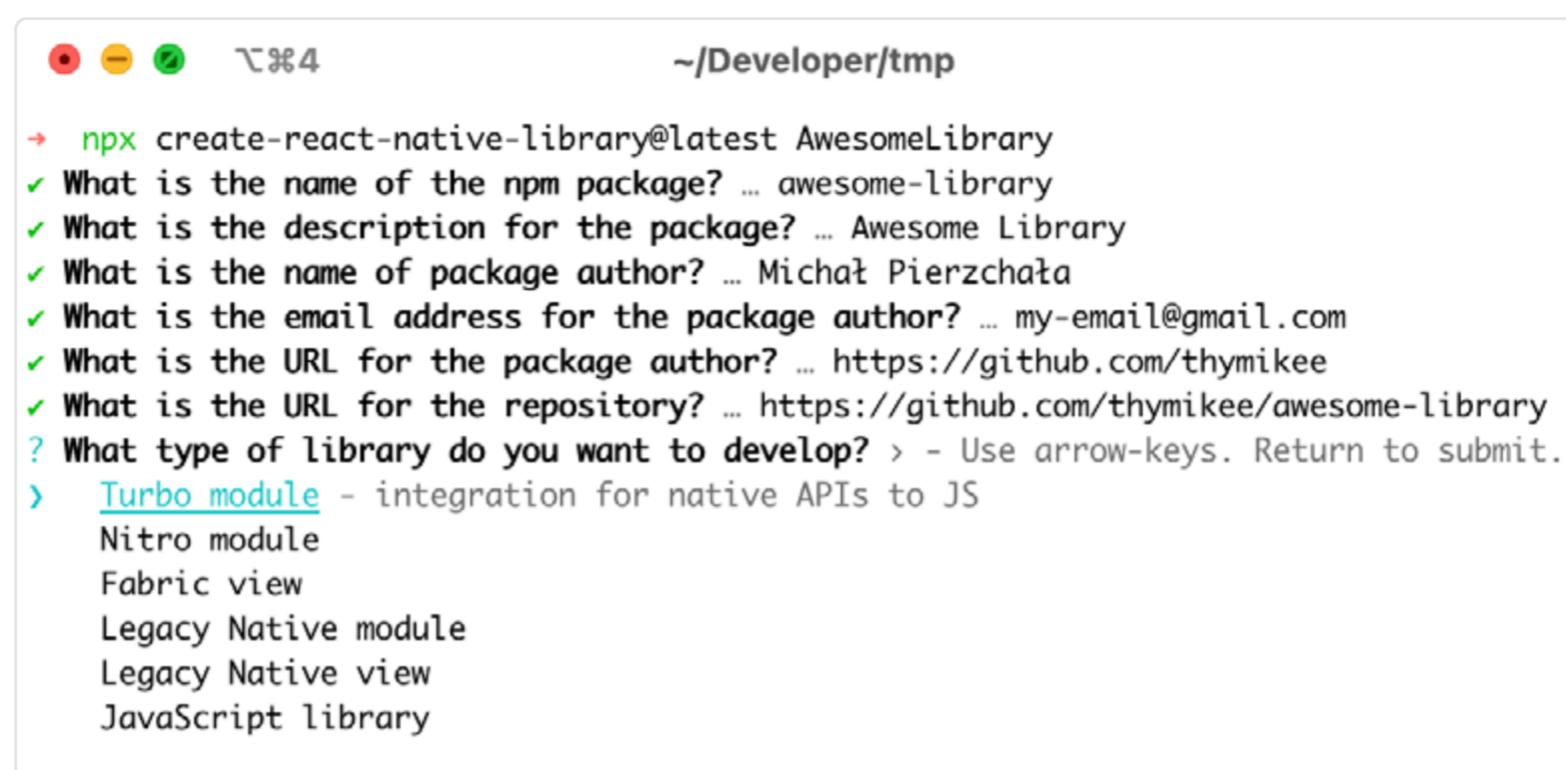
새로운 React Native 라이브러리를 만드는 작업에는 레거시 및 새로운 아키텍처 구현, TypeScript 또는 Flow 타이핑에서 코드 생성, 코드를 실행, 테스트 및 npm에 게시하기 위한 인프라를 처리해야 하는 초기 보일러플레이트 코드를 설정하는 작업이 포함됩니다.

다행히 모든 것이 설정된 새로운 라이브러리를 빠르게 구성하는 데 도움이 되는 훌륭한 도구인 **React Native Builder Bob**이 있습니다.

이를 사용하려면 터미널 창에서 `create-react-native-library` 명령을 실행하십시오.

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

그러면 패키지 이름, 터보 모듈, Fabric 뷰 또는 JavaScript 라이브러리 구축 여부와 같은 필요한 정보를 묻는 메시지가 표시됩니다. 또한 선호하는 코딩 언어를 선택할 수도 있습니다.



```
~/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
```

create-react-native-library 초기화 흐름 스크린샷

몇 가지 질문에 답하고 나면, 소비자 앱에서 사용할 수 있도록 타사 종속성으로 npm에 게시할 준비가 된 새로운 네이티브 모듈을 바로 사용할 수 있게 됩니다. 그러나 코드를 공개 레지스트리에서 사용할 수 있도록 하는 것이 항상 원하는 것은 아닙니다. 때로는 프로젝트에 특정한 사용자 정의 모듈을 빠르게 만들고 싶을 뿐입니다. Bob은 "create" 명령에 --local 플래그를 전달하여 로컬 모듈 기능을 통해 이를 가능하게 합니다.

이제 네이티브 모듈에 최신 언어를 사용하는 방법을 살펴보겠습니다.

최신 언어 사용: Swift 및 Kotlin

Android와 iOS는 오늘날 사용하는 프로그래밍 언어와 다른 언어로 시작했습니다. Android의 경우 Java였고, iOS의 경우 Objective-C였습니다.

Java의 후속 언어는 JetBrains에서 만든 Kotlin으로, 커뮤니티를 강타했고 2019년에는 Google에서 Android 앱 개발의 기본 언어로 공식 발표했습니다. 이전 버전과 완전히 상호 운용 가능합니다. 몇 가지 구성 변경 사항을 제외하고는 Java 대신 Kotlin을 사용하기 위해 특별히 해야 할 일은 없습니다. 실제로 지난 몇 년 동안 React Native의 일부가 Kotlin으로 재작성되었으므로 Builder Bob이 생성한 라이브러리는 이미 Kotlin을 사용합니다.

iOS의 경우 비슷하지만 약간 다릅니다. Swift는 Apple 내부에서 Objective-C를 장기적으로 대체하기 위한 차세대 독점 언어로 탄생했습니다. 2014년에는 공식 Xcode 지원을 통해 Swift 툴체인이 iOS 개발자에게 공개되었고, 곧 Apple과 Swift 커뮤니티가 관리하는 오픈 소스가 되었습니다. Swift는 Objective-C 코드베이스에서 쉽게 사용할 수 있으며 그 반대도 가능하여 점진적인 마이그레이션을 허용합니다. 그 이후로 SwiftUI와 함께 iOS 앱을 작성하는데 선호되는 언어입니다. 까다로운 부분은 C++ 상호 운용성입니다. 이는 새로운 아키텍처를 통해 React Native가 핵심 크로스 플랫폼 기능을 제공하기 위해 C++에 전념했기 때문에 중요합니다. 그러나 이를 완화할 방법이 있습니다.

내부적으로 React Native iOS 지원은 Objective-C와 Objective-C++라는 C++ 상호 운용성에 여전히 크게 의존합니다. 반면에 Swift는 매우 최근의 아직 실험적인 C++ 상호 운용성을 가지고 있지만 이전 방식과 동일하게 작동하지 않습니다. React Native 라이브러리에서 할 수 있는 것은 Objective-C에서 Swift로의 호출을 바인딩하는 작은 래퍼를 만드는 것입니다. 꽤 번거롭지만 작동합니다. 어떻게 할 수 있는지 살펴 보겠습니다.

모듈을 생성한 후 먼저 CocoaPods에서 제공하는 라이브러리 Podspec 파일을 수정하여 Swift 지원을 활성화합니다.

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

CocoaPods가 Swift를 연결할 스스로 이해하는 데 필요한 Podspec 변경 사항

다음으로 Xcode 내에서 새 Swift 파일을 만듭니다. 모듈의 iOS 폴더 안에 위치시켜야 합니다. 브리징 헤더를 생성할지 묻는 메시지가 나타나면 "예"를 클릭합니다.

다음으로 라이브러리 헤더 파일 내부에 다음 변경 사항을 추가합니다.

```
#import <Foundation/Foundation.h>  
  
+ #if __cplusplus  
#import "ReactCodegen/RNAwesomeLibrarySpec/RNAwesomeLibrarySpec.h"  
+ #endif
```

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

지원되지 않는 C++ 타입에서 Swift가 실패하지 않도록 필요한 변경 사항 diff

Swift는 C++ 타입을 이해하지 못하므로, `#if __cplusplus` 매크로를 사용하여 `__cplusplus`가 정의된 경우에만 (Swift에서는 정의되지 않습니다) 스펙 파일을 가져옵니다. 다음으로 브리징 헤더 내부에 기본 라이브러리 헤더 파일을 가져옵니다.

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

이제 `@objc` 속성을 통해 `AwesomeLibrary` 클래스를 Objective-C에서 사용할 수 있도록 확장 기능을 만들 수 있습니다.

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

multiply 메서드가 Objective-C에 노출된 AwesomeLibrary 확장

이제 방금 구현한 메서드를 사용하는 일만 남았습니다. 그렇게 하려면 라이브러리의 .mm 파일 (Objective-C++ 구문을 처리하는 파일)로 이동하여 RCT_EXTERN_METHOD()를 사용하십시오.

```
#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

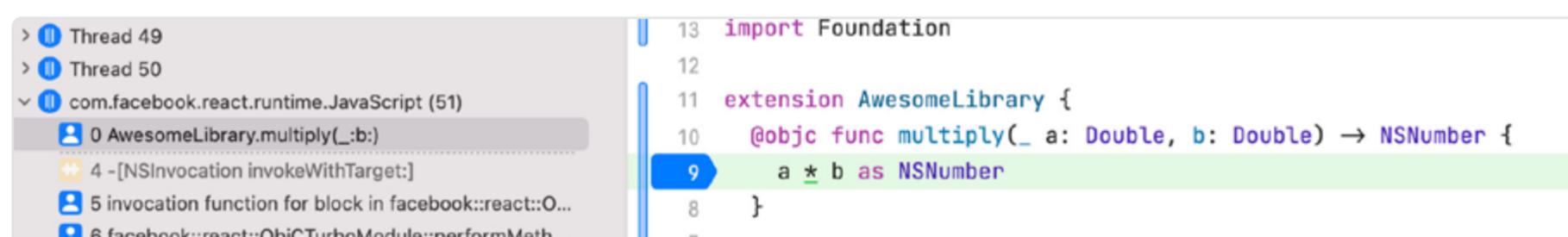
@end
```

여기서 무슨 일이 일어나는지 자세히 살펴보겠습니다. 먼저 모듈을 가져옵니다. Xcode는 <라이브러리 이름>-Swift.h라는 헤더 파일을 생성합니다. 정적/동적 프레임워크 모드에서 실행할 때는 라이브러리 이름이 접두사로 붙으므로 이 import가 존재하는지 확인합니다. 존재하지 않으면 일반적인 import로 대체합니다.

다음으로 RCT_EXTERN_METHOD를 사용하여 Objective-C 함수를 Swift 호출로 다시 매핑합니다. 이것이 모듈 내에서 Swift를 사용하는 데 필요한 전부입니다. Swift를 기본적으로 지원하기 위한 노력이 진행 중이므로, 앞으로는 이 보일러플레이트 코드가 필요하지 않기를 바랍니다.

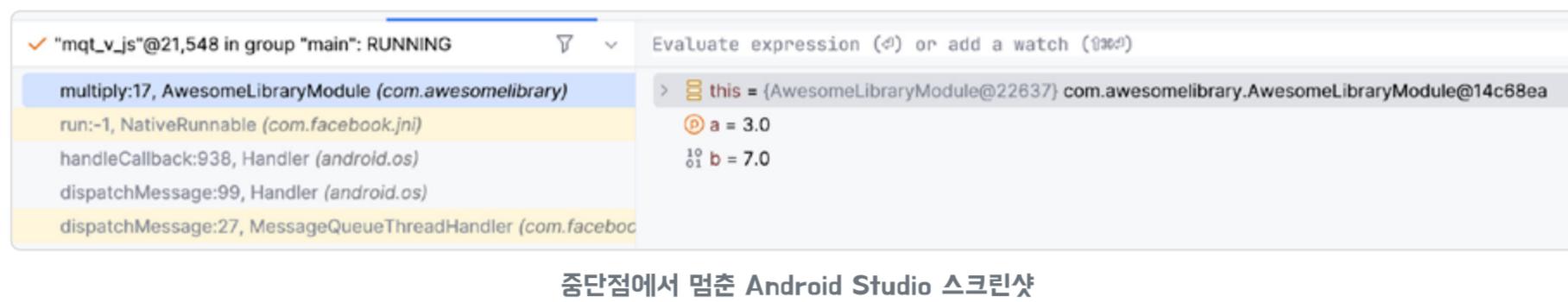
백그라운드 스레드 활용

브라우저 환경에서는 일반적으로 하나의 스레드를 처리합니다. 물론 일부 JavaScript 작업을 웹 워커로 오프로드할 수 있지만 약간 다릅니다. React Native에서는 사용자의 손에 있는 장치의 성능을 활용할 수 있습니다. 기본적으로 동기식 터보 모듈 메서드는 iOS에서 아래에서 볼 수 있듯이 JavaScript 스레드에서 호출됩니다.



중단점에서 멈춘 Xcode 스크린샷

Android에서도 마찬가지입니다.



**React Native 앱의 스레딩에 대한 자세한 내용은
"터보 모듈과 Fabric의 스레딩 모델 이해" 장에서 다룹니다.**

이제 멀티스레딩을 활용하여 일부 작업을 백그라운드 스레드에서 수행하도록 예약하려고 합니다. 새로운 비동기 메서드 `multiplyOnBackgroundThread`를 도입해 보겠습니다. 이 가이드의 간결성을 유지하기 위해 새 메서드를 추가하는 자세한 내용은 건너뛰고, 코드를 백그라운드 스레드로 오프로드하는 데 중점을 두겠습니다.

iOS부터 시작하겠습니다. 백그라운드 스레드에서 코드를 실행하려면 작업의 스케줄링 및 실행을 관리하여 메인 스레드에서 실행되지 않도록 보장하는 `DispatchQueue API` 형태의 고수준 큐 개념을 사용할 수 있습니다. `DispatchQueue.global()` 블록에서 사용할 수 있는 코드는 백그라운드 작업을 위한 글로벌 동시 큐가 사용 가능해지면 실행됩니다. 블록 내부에서 `resolve()` 메서드를 호출하여 결과에 대해 JS 함수에 알릴 수 있습니다.

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

Swift에서 `DispatchQueue.global()` 사용 예시

Android에서 동일한 작업을 수행하는 방법을 살펴보겠습니다. Swift와 마찬가지로 백그라운드 스레드에 직접 접근하지 않습니다. 대신 코루틴을 사용합니다. 코루틴은 Kotlin의 동시성 프레임워크의 일부이며, 읽기 쉽고 유지 관리하기 쉬운 비동기 코드를 작성하는 방법을 제공합니다. 코루틴은 물리적 스레드에 바인딩되지 않으며, 수천 개의 코루틴이 적은 수의 스레드에서 실행될 수 있다는 점에 유의해야 합니다.

코루틴을 생성하기 위해 `CoroutineScope` 클래스를 사용하고 이를 `moduleScope` 변수에 인스턴스화합니다.

```
@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()
    }
}
```

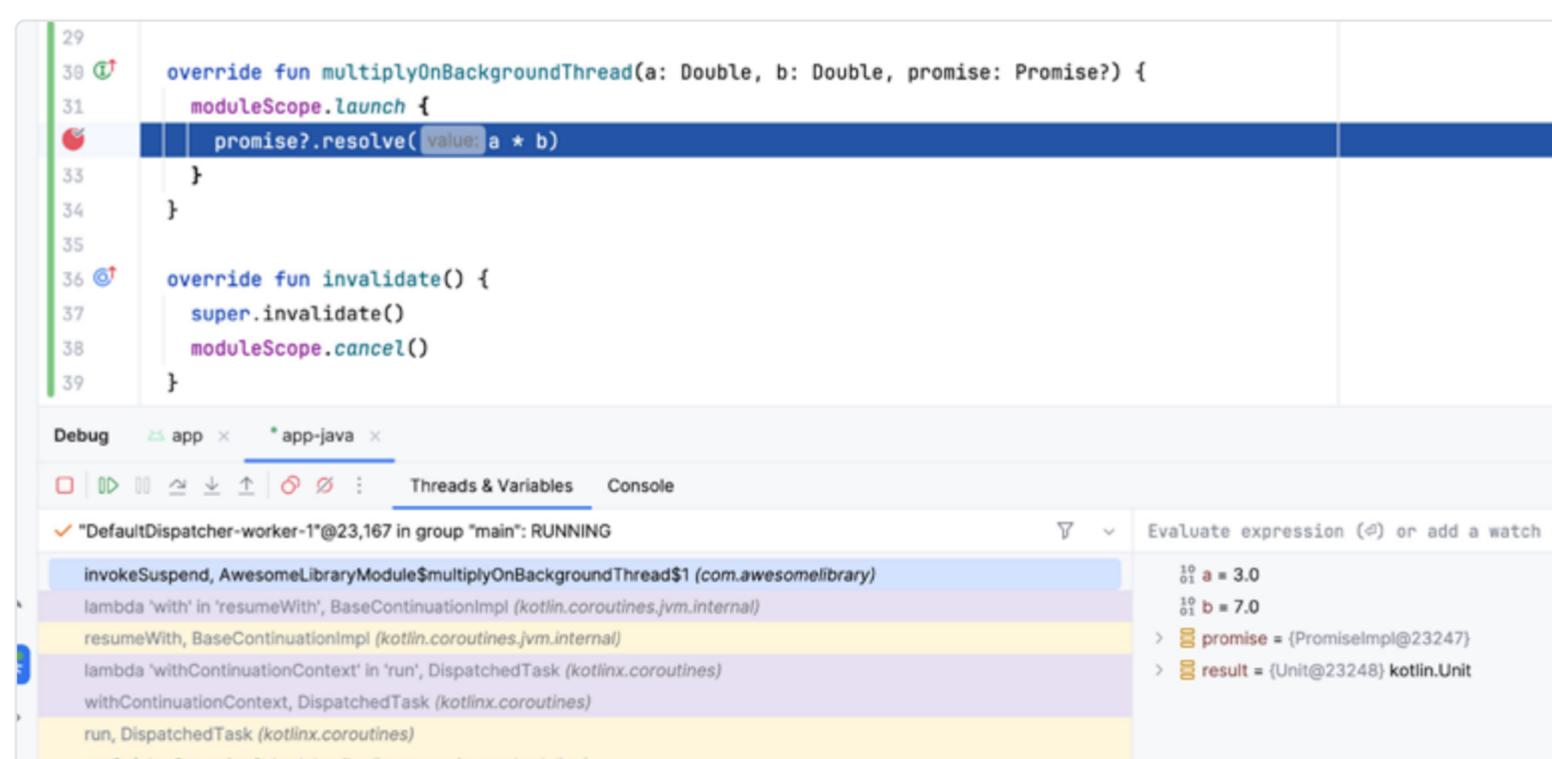
Kotlin 파일에서 `CoroutineScope` 생성하기

모듈이 무효화될 때 스코프를 취소하여 메모리 누수를 방지하는 것을 잊지 마세요. 이제 `moduleScope`를 사용할 수 있으므로, `launch` 메서드를 사용하여 작업을 백그라운드 스레드로 오프로드할 수 있습니다.

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

Kotlin에서 `moduleScope.launch` 사용 예시

이 함수를 호출한 후 디버거 중단점을 설정하면 `multiplyOnBackgroundThread`가 메인 스레드 대신 작업자 스레드에서 실행되는 것을 확인할 수 있습니다.



중단점에서 멈춘 Android Studio 스크린샷

iOS에서도 마찬가지입니다.

The screenshot shows a comparison between a thread dump in Xcode and the corresponding C++ Turbo Module code. On the left, a thread dump lists various threads, including Thread 10 which is executing the `multiplyOnBackgroundThread` function. On the right, the source code for this function is shown:

```
7 @objc func multiplyOnBackgroundThread(
6   _ a: Double,
5   b: Double,
4   resolve: @escaping RCTPromiseResolveBlock,
3   reject: RCTPromiseRejectBlock
2 ) {
1   DispatchQueue.global().async {
15     resolve(a * b)
1   }
2 }
3 }
```

The line `15 resolve(a * b)` corresponds to the line in the thread dump where the closure is being resolved on the background thread.

Xcode에서 중단점에 멈춘 스크린샷



"터보모듈과 Fabric의 스레딩 모델 이해하기 장"에서 설명하는 스레딩과 관련된 몇 가지 더 주의해야 할 점이 있습니다.

플랫폼 독립적인 코드를 C++로 대체하십시오.

만약 C++로 대체할 수 있는 플랫폼 독립적인 로직이 있다면, 상당한 성능 향상을 얻을 수 있습니다. C++은 iOS, Android, Windows는 물론 사실상 모든 환경에서 지원됩니다. 하지만, 더 낮은 수준의 언어를 사용하는 것은 스레딩을 다룰 때 더욱 신중해야 하고 메모리 누수를 주의해야 함을 의미합니다. [React Native 웹사이트](#)에서 C++ Turbo Module 구축에 대한 가이드를 찾을 수 있습니다.

본 가이드 작성 시점 (2025년 1월)에는 C++ Turbo Module이 iOS에서 자동 링크를 지원하지 않습니다. 개발자 경험을 개선하고 AppDelegate 코드를 수정하는 번거로움을 덜어주기 위해, `registerCxxModuleToGlobalModuleMap` 메서드를 사용하여 C++ Turbo Module을 등록하여 런타임에 사용 가능하도록 만들 수 있습니다. 이를 위해 작은 Objective-C 클래스를 만들고, 앱 실행 과정에서 Objective-C 런타임에 클래스가 추가될 때 한 번 호출되는 `+ load` 메서드를 활용하십시오.

```
#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
```

C++ 인터페이스의 숨겨진 비용

언어 간 경계를 넘나드는 것은 종종 숨겨진 비용을 수반하며, 앱에서 다른 네이티브 언어를 사용하기로 신중하게 결정할 때 이러한 비용을 인지해야 합니다.

Objective-C++

iOS에서 앱과 React Native 자체는 여전히 Objective-C를 흔히 사용하는데, 이는 대부분의 경우 동적인 특성 때문에 Swift보다 느립니다. 각 Objective-C 메서드 호출은 메서드 테이블에서 조회가 필요합니다. 반면에, Objective-C와 C++을 혼용하는 비용은 컴파일 시간에 완전히 처리되므로 거의 0에 가깝습니다. Objective-C++ 컴파일러는 이를 네이티브 C++ 코드로 취급합니다. 안타깝게도, Objective-C Turbo Module을 빌드할 때는 메서드 호출이 메서드 테이블 조회를 통해 실행됩니다.

Swift C++ 상호 운용성

이와 비교하여 Swift는 C++과 유사하게 클래스당 가상 메서드 테이블(vtables)을 사용합니다. 게다가 Swift와 C++ 간의 상호 운용성은 언어 간 전달 시 비용이 거의 들지 않습니다. 이것이 Objective-C를 건너뛰고 C++ 상호 운용성을 사용하는 Nitro Module로 전환하면 상당한 성능 향상을 얻을 수 있는 이유 중 하나입니다.

JNI

Android 생태계에서 Java Native Interface (JNI)는 Java Virtual Machine (JVM)에서 실행되는 바이트코드와 C++와 같은 네이티브 코드 간의 상호 작용을 가능하게 합니다. 그러나 각 호출마다 메서드 테이블에서 함수 조회가 필요하므로 JNI는 비용이 많이 들 수 있습니다.

예를 들어 각 Kotlin 호출을 C++로 연결하는 것처럼 JNI를 직접 사용하면 호출 속도가 느려지지만 시간이 오래 걸리는 작업에는 가치가 있을 수 있습니다. C++ Turbo Module을 사용하면 JNI는 주로 모듈 초기화에 사용되므로 런타임 중에 건너뛸 수 있습니다. JavaScript는 JSI (React Native의 JavaScript 인터페이스)를 통해 C++ 함수에 대한 참조를 보유하므로 추가 오버헤드 없이 이 코드에 직접 액세스 할 수 있습니다.

GUIDE

메모리 누수 추적 방법

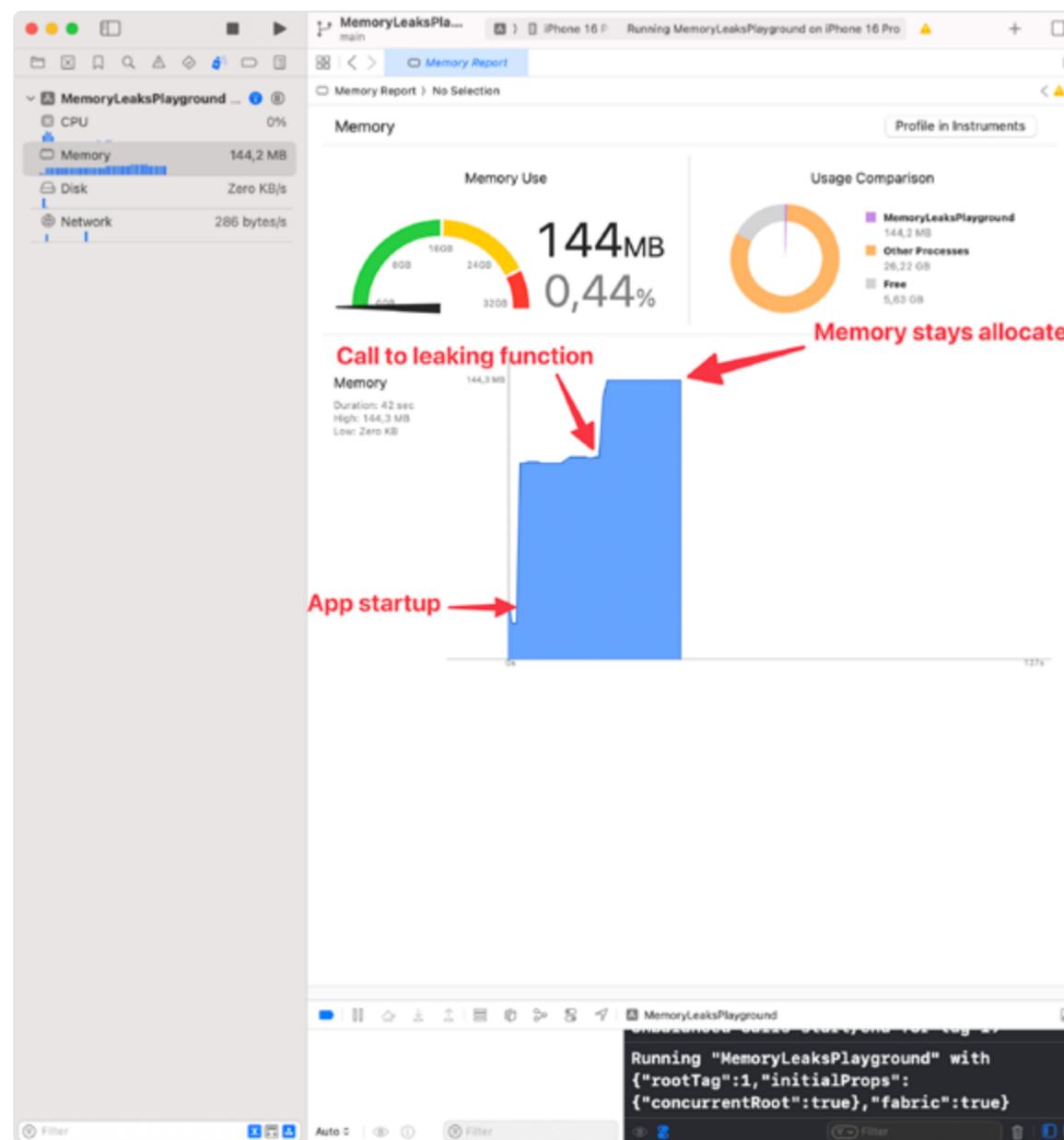
전용 도구 없이 메모리 누수를 감지하기는 어렵습니다.

다행히 iOS와 Android 모두 JS 메모리 누수 추적 방법 챕터에서 다룬 JS 메모리 프로파일러와 유사하게 문제의 근원을 식별하는 데 도움이 되는 유용한 도구를 갖추고 있습니다.

Xcode

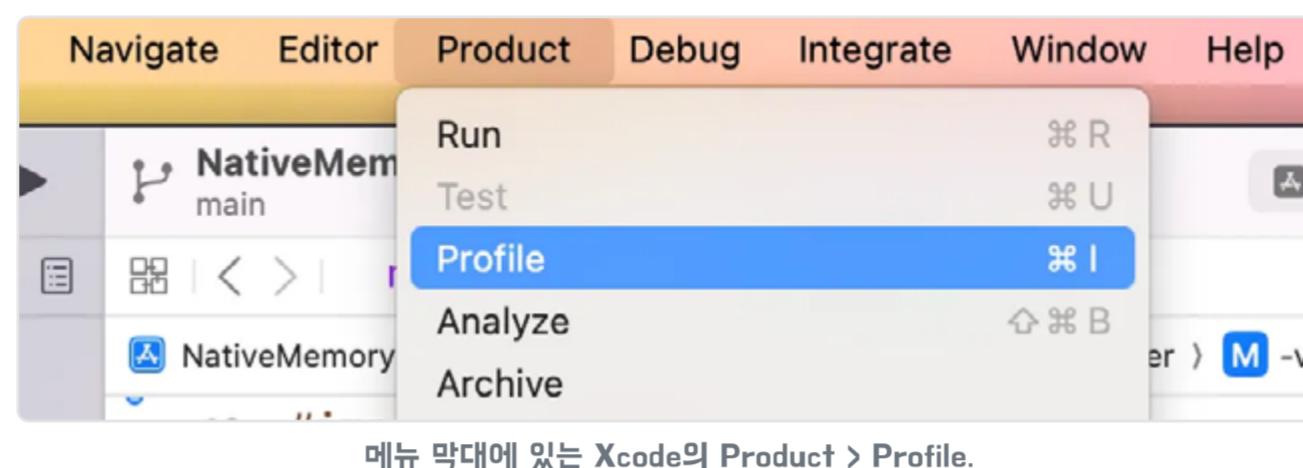
Apple의 IDE는 메모리 누수 추적에 유용한 도구입니다. 이러한 방식으로 앱을 프로파일링하는 동안 네이티브 메모리 관리 이해 챕터에서 자세히 논의한 네이티브 메모리 누수를 발견할 수 있습니다. 시작하려면 Xcode를 열고 앱을 빌드합니다.

메모리 누수를 가장 빠르게 확인하는 방법은 앱을 실행할 때 측면 창의 디버그 네비게이터에서 "Memory Report"를 확인하는 것입니다.

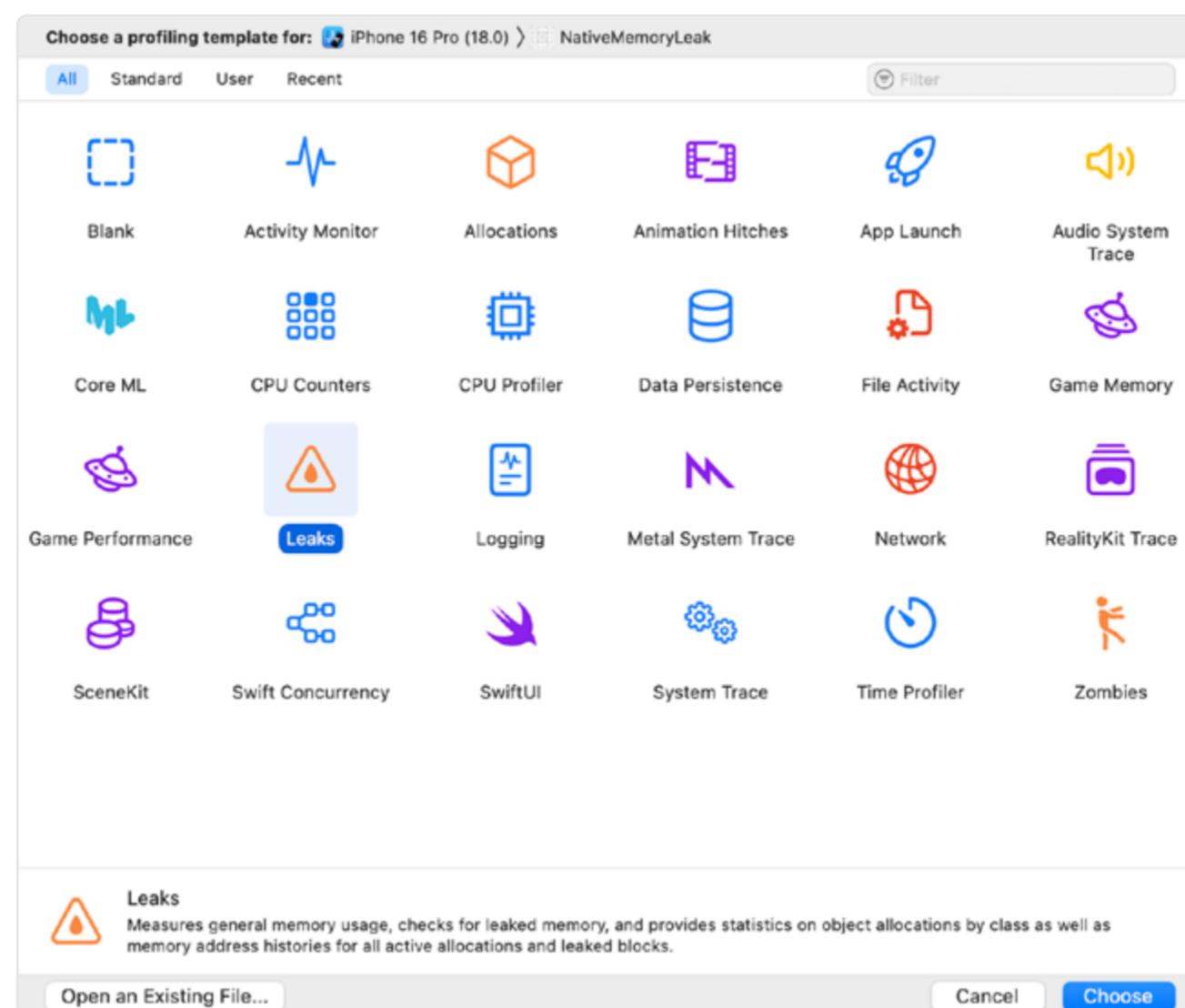


Xcode에서 누수를 보여주는 메모리 보고서.

앱에서 사용된 메모리 그래프를 보여줍니다. 앱의 릴리스 빌드에서 이 프로파일링 기술을 사용하는 것이 좋습니다. 메모리 그래프는 메모리 누수 문제를 빠르게 추적하는 데 유용하지만 누수의 소스를 확인하려면 Xcode Instruments, 특히 Leaks 프로파일링 템플릿을 사용해야 합니다. 새 프로파일링 세션을 시작하려면 Product > Profile로 이동합니다.



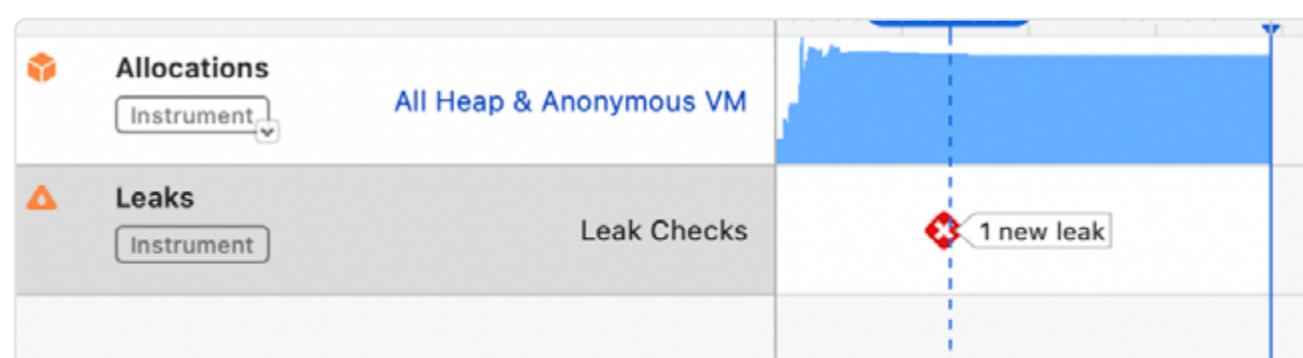
앱이 빌드되면 메뉴에서 "Leaks"를 선택하고 "Choose" 버튼을 누릅니다.



Leaks 도구가 선택된 Instruments.

이제 새 창이 열립니다. 왼쪽 상단에 있는 녹음 버튼을 클릭하고, 장치에서 메모리 누수를 일으킬 가능성이 있는 단계를 따르세요. 이는 일부 네이티브 코드를 트리거하는 새 화면으로의 탐색이나 버튼 누름일 수 있습니다.

녹음이 완료되면 Leaks 도구가 누수를 나타내는 빨간색 마커를 표시합니다.



Leaks에서 새로운 누수가 감지되었습니다.

그것을 클릭하면 누수된 객체, 책임 라이브러리 및 책임 프레임의 요약이 표시되어야 합니다. 오른쪽에는 메모리를 누출한 정확한 함수를 보여주는 스택 추적이 있습니다.

The screenshot shows the 'Leaks' instrument's stack trace. On the left, under 'Responsible Library' and 'Responsible Frame', 'NativeMemoryLeak' is selected, and 'createNewStrings()' is highlighted. On the right, the 'Stack Trace' pane displays the following call stack:

```
_malloc_type_malloc_outlined
operator new(unsigned long)
createNewStrings()
-[ViewController viewDidLoad]
-[UIViewController _sendViewDidLoadWithAppearanceProxyObject]
-[UIViewController loadViewIfRequired]
```

`createNewStrings` 메서드 호출을 가리키는 스택 추적 옆에 누수의 원인이 되는 라이브러리와 프레임입니다.

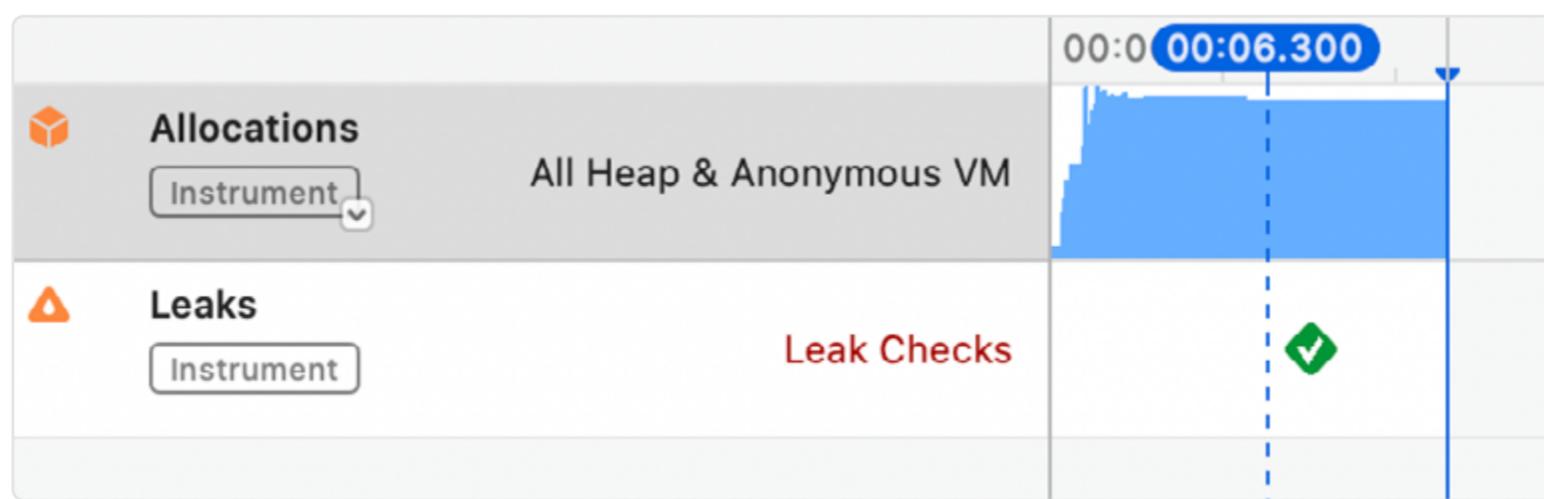
그러한 경우에, 누수의 원인은 `createNewStrings()`입니다. 함수 이름을 더블 클릭하면 Xcode는 이 함수의 소스 코드를 보여줍니다.

The screenshot shows the Xcode editor with the file `ViewController.m` open. The cursor is positioned on the line of code where the memory leak was detected. The code is as follows:

```
1 #import "ViewController.h"
2 #import <iostream>
3 #import <string>
4
5 @interface ViewController : UIViewController
6
7 @implementation ViewController
8
9 - (void)viewDidLoad {
10     [super viewDidLoad];
11     createNewStrings();
12 }
13
14 void createNewStrings() {
15     for (int i = 0; i < 10; i++) {
16         std::string *str = new std::string("Hey");
17         std::cout << *str;
18     }
19 }
20
21 @end
```

메모리 누수를 일으키는 C++ 함수 구현

아, 힙에 할당된 문자열을 삭제하는 것을 잊었군요! 문자열을 명시적으로 삭제했더니 Xcode에서 누수가 사라졌다고 표시해줍니다.

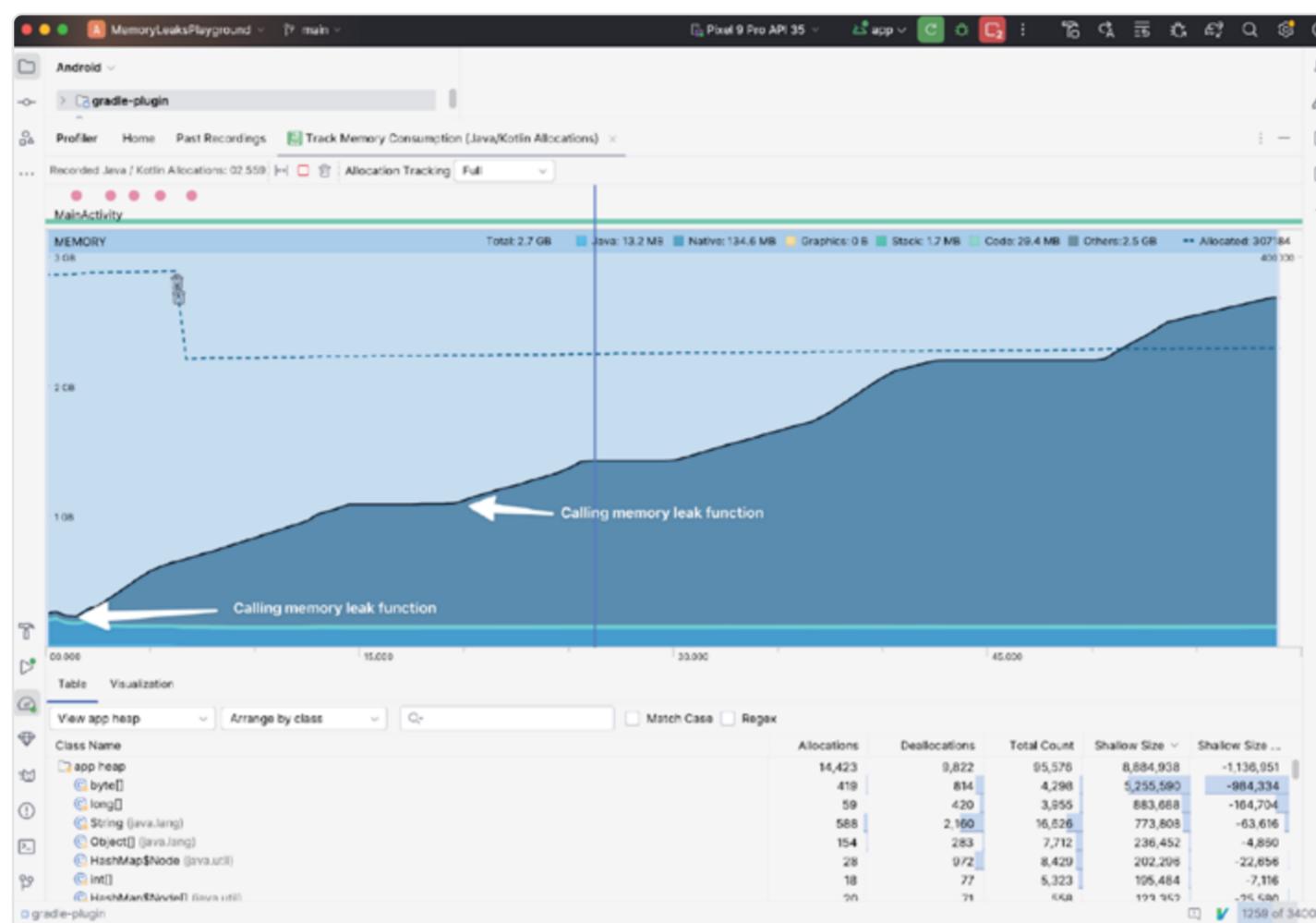


Leaks 도구에 누수가 표시되지 않음.

자, 이제 Android에서 메모리 누수를 어떻게 찾아낼 수 있는지 알아봅시다.

Android Studio

Xcode와 마찬가지로, 릴리스 모드로 앱을 빌드하고 메모리 사용량 그래프를 검사할 수 있습니다. Android Studio에서는 왼쪽 하단 모서리 근처에 있는 "Profiler" 탭으로 이동한 다음 "메모리 소비 추적"을 선택하십시오.



Android Studio의 메모리 누수 실험 환경

이제 Android Studio 프로파일러를 사용하여 더 복잡한 누수를 찾아보겠습니다. Android 작동 방식에 익숙하다면, 구성이 변경될 때 Android의 기본 동작은 `MainActivity` 클래스를 다시 생성하는 것임을 알 것입니다. 여기에는 앱 회전, 다크 모드 변경 등 많은 것이 포함됩니다.



React Native는 `ApplicationManifest.xml`에서 `android:configChanges`를 지정하여 이 동작을 선택 해제합니다.

그러나 네이티브 앱에서는 이러한 구성 변경을 처리하는 것이 일반적입니다. `MainActivity`의 재생성을 리스너 패턴을 구현하는 전역 싱글톤과 결합하면 메모리 누수를 일으키는 레시피가 됩니다. 겉으로는 간단해 보이는 코드가 `MainActivity`의 할당 해제를 어떻게 유지하는지 살펴봅시다.

다음은 리스너 인터페이스로 리스너를 등록하는 `EventManager` 클래스입니다.

```
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()
    }
```

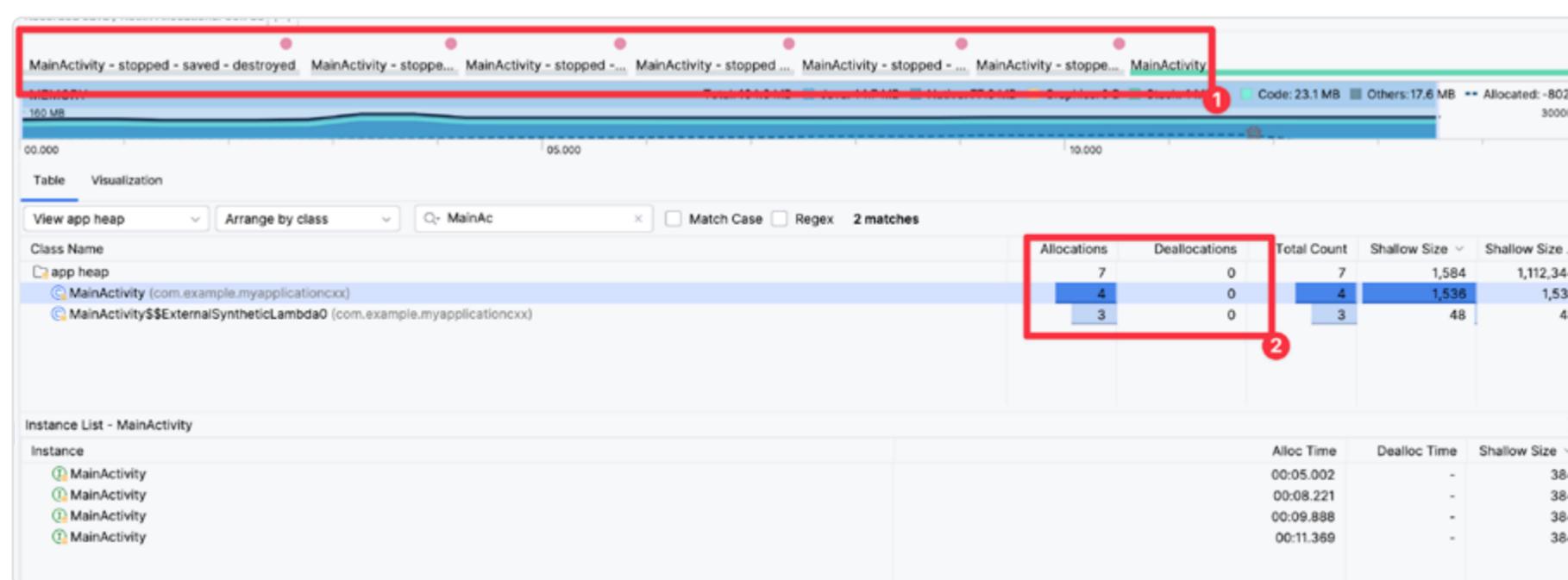
이제 `MainActivity`를 `EventManager` 변경 사항에 구독시켜 보겠습니다.

```
class MainActivity : AppCompatActivity(), Callback {
    // ...

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // ...
        EventManager.addListener(this)
    }
}

override fun onEvent() {
    Log.d("MAIN_ACTIVITY", "Hey")
}
```

다음으로 실행 > 프로필을 클릭한 다음 “메모리 소비 추적(Java/Kotlin 할당)”을 선택하여 메모리 프로파일러를 실행합니다. 앱이 실행되고 메모리 그래프가 실시간으로 업데이트되는 것을 볼 수 있습니다. 장치를 여러 번 회전한 후 `MainActivity` 수명 주기(아래 숫자 1로 강조 표시됨)를 볼 수 있습니다. 빨간색 점은 터치 이벤트를 상징합니다. 아래에서 각 탭 후 활동이 다시 생성되었음을 알 수 있습니다.



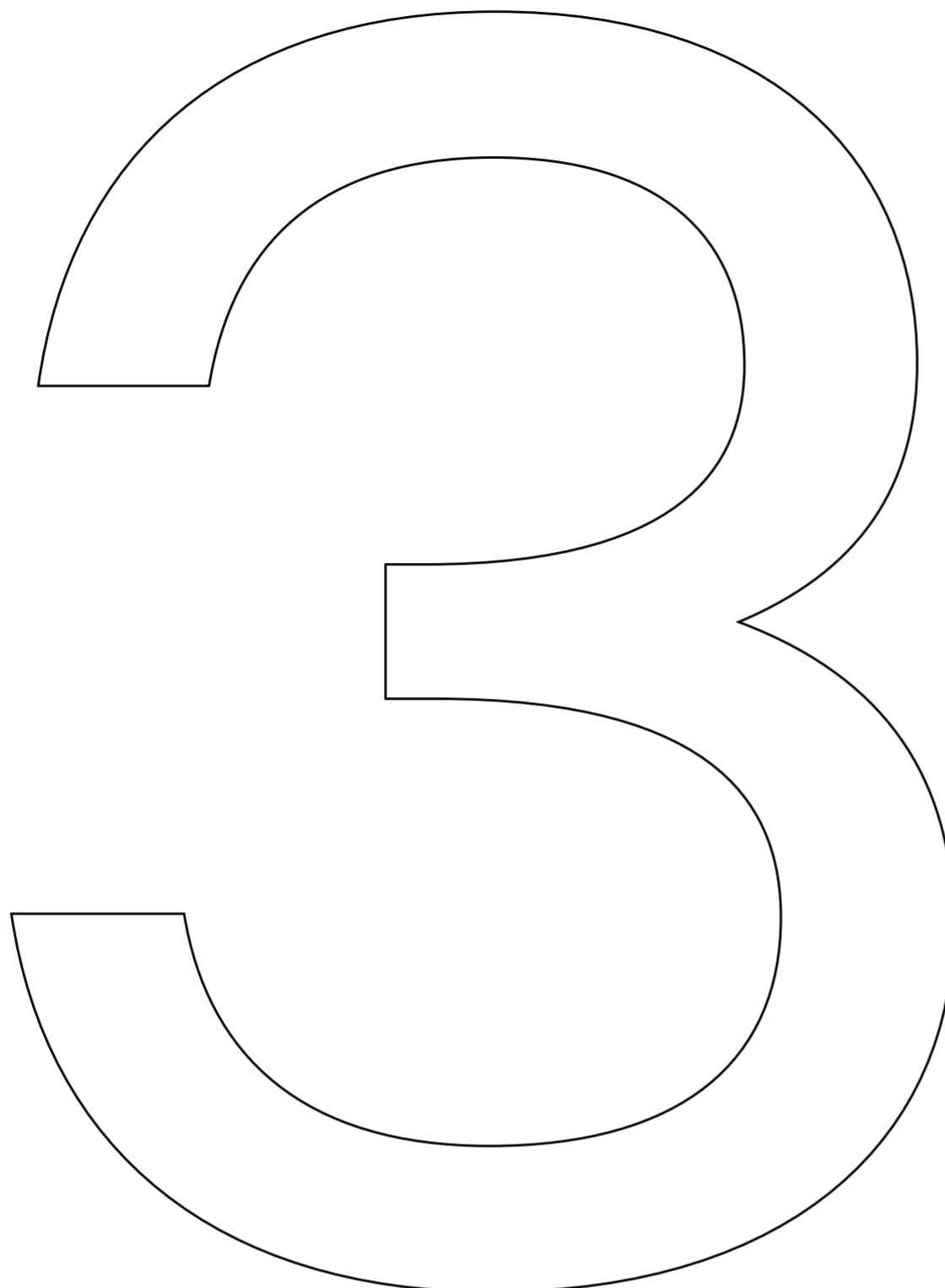
EventManager가 MainActivity에 대한 참조를 보유하여 발생하는 메모리 누수의 분석.

두 번째 강조 표시를 보면 `MainActivity`가 4번 할당되었지만 해제된 횟수는 0번입니다! 이는 이전의 Main Activity가 여전히 `EventManager`에 의해 참조되고 있기 때문에 Garbage Collector에 의해 해제되지 못하고 있음을 의미합니다. 이는 빠르게 앱 충돌로 이어질 수 있습니다.

실제로 앱에서 메모리 누수가 자주 발생하는 것은 아닙니다.

하지만 발생하는 경우 메모리 누수의 원인을 찾는 것은 대부분의 개발자가 두려워하는 일입니다
. 올바른 도구를 사용하면 누수의 원인을 식별하고 앱을 최적화하여 메모리 사용량을 줄일 수 있
으며, 결과적으로 OS에 의해 조기에 종료되는 것을 방지할 수 있습니다.

PART



번들링

이 문서는 JavaScript와 네이티브 코드 모두에서 사전 컴파일(AOT) 및 패키징 기술을 활용하여 TTI(Time To Interactive, 상호 작용 가능 시점)를 개선하기 위한 가이드와 기술을 제공합니다.

Introduction

이 가이드의 1부와 2부에서는 CPU 및 메모리 프로파일링 방법, TTI(Time to Interactive) 및 FPS(Frames Per Second)와 같은 주요 지표 측정 방법, 그리고 앱의 런타임 성능을 개선하기 위한 모범 사례 적용 방법 (주로 FPS 지표에 영향을 미침)에 대해 설명했습니다. 하지만 앱 초기화 속도를 높이기 위해 빌드 시점에 할 수 있는 최적화는 무엇일까요?

3부에서는 바로 이 부분을 다룹니다. 여기서는 TTI(Time to Interactive) 지표로 측정되는 React Native 앱이 가능한 한 빨리 열리도록 돋는 기술 및 도구에 초점을 맞출 것입니다.

2부에서 배운 것처럼 TTI는 애플리케이션의 가장 중요한 성능 지표 중 하나입니다. 앱 로딩 속도가 빠를수록 사용자가 돌아올 가능성이 높습니다. 반대로 앱 로딩 속도가 느리면 사용자가 대안을 찾아서 떠날 가능성이 높아집니다. 시작 시간에 영향을 미치는 주요 구성 요소를 살펴보겠습니다.

JavaScript 번들과 번들러

웹과 마찬가지로 React Native 개발자는 번들러를 사용하여 JS 엔진에서 읽을 수 있는 최종 JavaScript 결과물을 생성합니다. 번들러는 TypeScript, HTML, CSS, JPG, MP4 등 다양한 종류의 여러 소스 파일을 하나의 시작 파일을 통해 가져와서 대상 JS 엔진이 이해할 수 있는 형식으로 변환하는 프로그램입니다. 일반적으로 결과는 .js 파일과 .html과 같은 다른 비 JS 자산입니다.

기본적으로 React Native 프로젝트에는 React Native용으로 특별히 제작된 번들러인 Metro가 함께 제공됩니다. Metro는 웹을 포함한 모든 플랫폼용 번들링을 지원하지만, 코드 분할 및 트리 쉐이킹과 같은 몇 가지 주요 기능은 부족합니다. Metro는 즉시 사용 가능한 환경과 괜찮은 성능을 제공하지만, 웹 개발에서 널리 사용되는 Webpack 또는 Rspack과 같은 다른 번들러도 RePack을 사용하여 React Native 프로젝트에 통합할 수 있습니다.

개발 환경에서 앱 번들은 JavaScript 코드를 포함하지 않고 개발 서버에서 코드를 가져옵니다. 이를 통해 Hot Module Replacement 또는 Fast Refresh를 통해 소스 파일이 변경될 때 화면에 나타나는 내용이 1초 이내로 빠르게 새로 고침될 수 있습니다.

릴리스를 준비할 때 앱 번들에는 JavaScript 대신 Hermes Bytecode가 포함된 전용 .jsbundle 파일이 포함되며 개발 서버에 연결되지 않습니다. 이 파일은 기본 Xcode 및 Gradle 스크립트에 의해 생성되며, 내부적으로 'npx react-native bundle' 명령을 실행하여 JS 결과물을 생성한 다음 Hermes 컴파일러를 통과시켜 최종 바이트코드 파일을 만듭니다.

이제 iOS 및 Android 플랫폼과 관련된 다양한 패키징 전략 및 배포 형식을 살펴보겠습니다.

Android 앱 번들

Android 앱에 사용할 수 있는 패키징 형식에는 두 가지가 있습니다. APK(Android Package Kit)와 AAB(Android App Bundle)가 있으며, 이들은 서로 다른 아키텍처를 실행하는 다양한 장치에 대한 네이티브 코드를 캡슐화합니다.

- **armeabi-v7a** 구형 및 저가형 ARM 장치용으로 설계되었습니다.
- **arm64-v8a** 64비트 ARM 프로세서에 최적화되어 더 나은 성능을 제공합니다.
- **x86** Intel 기반 장치를 대상으로 합니다.
- **x86_64** Intel 장치용 64비트 버전으로 더 나은 성능과 기능을 제공합니다.

React Native 프로젝트에서 이러한 아키텍처는 `android/gradle.properties` 파일의 `reactNativeArchitectures` 속성을 통해 구성할 수 있습니다.

 여러 아키텍처를 빌드하면 빌드 시간이 늘어납니다. **React Native Community CLI**를 사용하여 개발하는 경우 '`--activeArchOnly`' 플래그를 '`run-android`' 명령에 전달하여 빌드 속도를 높일 수 있습니다.

APK

APK는 Android 앱을 배포하고 설치하는 데 사용되는 전통적인 형식으로, 개발 및 테스트 중 또는 Play Store 외부에서 앱을 배포할 때 사용됩니다. 모든 .apk 파일은 사실 ZIP 파일이기도 합니다. APK는 장치 아키텍처 또는 기타 구성을 기반으로 리소스를 분할하도록 최적화되어 있지 않으므로 일반적으로 여러 장치 아키텍처를 내부에 결합하여 다양한 장치 또는 에뮬레이터에서 실행될 수 있도록 하여 전반적인 크기를 늘립니다.

AAB

AAB는 앱 스토어에 대한 앱 배포를 최적화하기 위해 Google에서 도입한 최신 형식입니다. .aab 파일에는 필요에 따라 APK를 생성하는 데 필요한 파일이 포함되어 있습니다. 이 형식은 이제 Google Play Store를 통해 배포하는 데 필수이며, 각 사용자 장치에 최적화된 APK를 생성하여 다운로드 크기를 줄입니다.

동적 라이브러리

JavaScript와 유사하게 모바일 플랫폼에는 재사용 가능한 라이브러리라는 개념이 있습니다. JavaScript와 같은 해석 언어와 비교하여 Android 플랫폼을 구동하는 Kotlin 또는 Java와 같은 정적으로 컴파일된 언어는 컴파일 단계에서 라이브러리를 연결해야 합니다. 이러한 라이브러리에는 다음과 같은 두 가지 형식이 있다는 것을 아는 것이 중요합니다.

- **.a** C/C++ 라이브러리 파일; 정적으로 연결됨.
- **.aar** Android Archive; 클래스, 드로어블과 같은 리소스 등을 포함하며 정적으로 연결됨.
- **.so** 공유 라이브러리; 동적으로 연결됨.

정적으로 연결된 라이브러리는 런타임 링크 문제로부터 발생하는 장치 간의 종속성 관리 및 일관된 동작 측면에서 일반적으로 단순성을 제공하지만, 동적으로 연결된 라이브러리와 비교하여 APK 크기가 더 커지고 재빌드 및 재배포하는 데 더 많은 작업이 필요하며 여러 앱에서 동일한 코드를 로드하여 잠재적으로 메모리 사용량이 증가할 수 있습니다.



정적으로 연결된 라이브러리는 컴파일 시간에 애플리케이션 코드와 함께 포함되어 애플리케이션을 실행하는 데 필요한 모든 코드가 포함된 단일 실행 파일을 만듭니다.

동적으로 연결된 라이브러리는 런타임에 로드되어 여러 프로그램이 동일한 라이브러리를 공유할 수 있도록 하여 메모리 사용량을 줄이고 애플리케이션을 다시 컴파일하지 않고도 업데이트를 적용할 수 있습니다.

iOS 앱 번들

iOS 앱에 사용할 수 있는 패키징 형식에는 두 가지가 있습니다. IPA(iOS App Store Package)와 APP(Application Bundle)가 있으며, 이들은 서로 다른 아키텍처를 실행하는 다양한 장치에 대한 네이티브 코드를 캡슐화합니다.

- **arm64** ARM 기반 프로세서를 사용하는 대부분의 최신 Apple 장치에 사용됩니다.
- **x86_64** macOS의 iOS 시뮬레이터에 사용되며, ARM 기반이 아닌 하드웨어에서 개발 및 테스트를 지원합니다.

IPA

이 파일 형식은 Apple App Store를 통해 또는 임시 및 엔터프라이즈 배포 채널을 통해 배포하기 위해 iOS 애플리케이션을 패키징하는 데 사용됩니다. .ipa 파일은 iOS 앱이 Apple 장치에 설치되는 데 필요한 모든 정보가 포함된 아카이브입니다.



재미있는 사실: .ipa를 .zip으로 변경하고 내용물을 검사할 수 있습니다! .apk 파일도 마찬가지입니다.

App Store는 IPA 파일에서 사용 가능한 정보를 사용하여 앱 얇게 만들기를 수행하고, 특정 장치에 앱 배포를 맞춤화하여 앱 설치를 최적화합니다. 여기에는 App Slicing(장치 구성에 필요한 앱 리소스만 제공), On-Demand Resources(필요에 따라 추가 콘텐츠 다운로드) 및 Bitcode(Apple이 개발자에게 새 버전을 요구하지 않고도 다양한 장치에 맞게 앱을 최적화할 수 있도록 허용)와 같은 기능이 포함됩니다.

APP

이것은 주로 시뮬레이터에서 앱을 실행하기 위해 개발 중에 사용되는 형식입니다. .app은 실제로 앱의 모든 리소스, 바이너리 실행 파일 및 메타데이터를 하나로 결합한 디렉토리입니다. 배포용으로는 사용되지 않습니다.

동적 라이브러리

iOS는 외부 라이브러리를 포함하기 위한 연결 단계의 필요성에 있어 Android와 다르지 않습니다. 사용 가능한 형식은 다음과 같습니다.

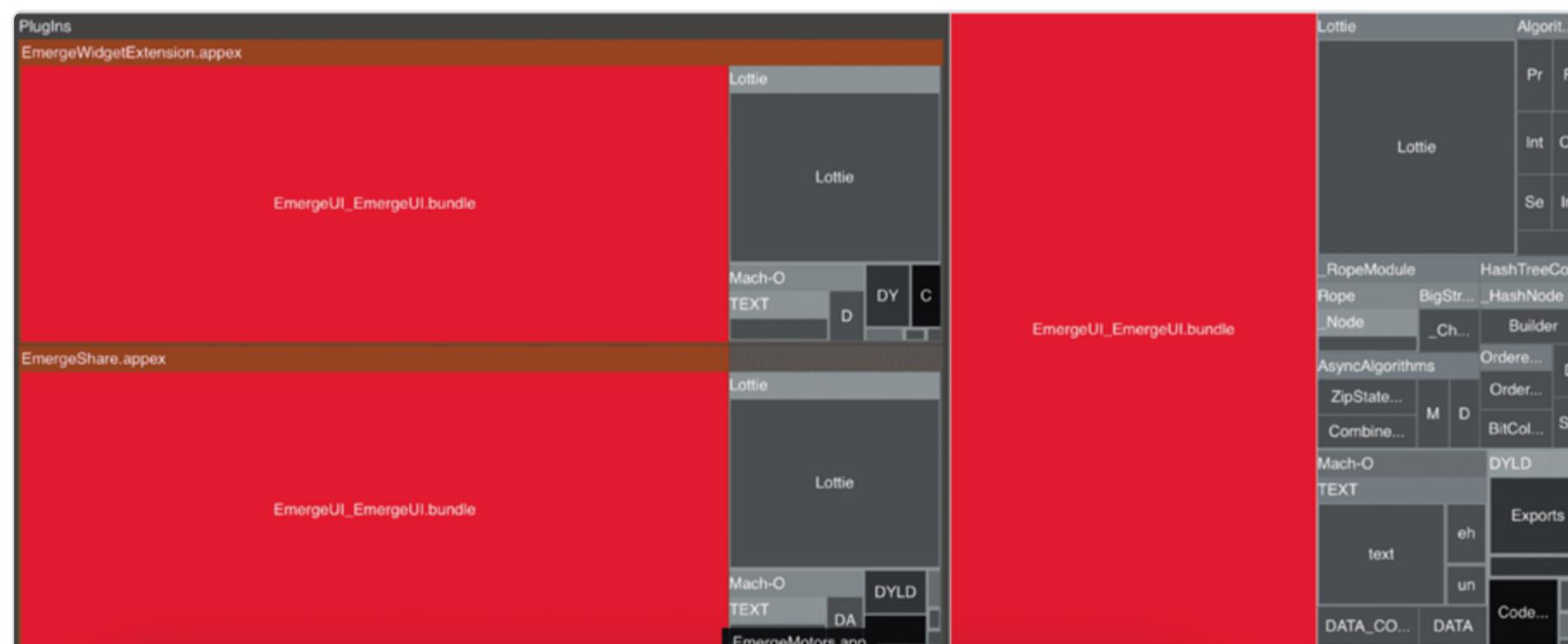
- **.a** C/C++ 라이브러리; 정적으로 연결됨.
- **.dylib** 동적 라이브러리; 독립 실행형이며 동적으로 연결됨.

- **.framework** 단일 플랫폼에 대한 바이너리, 헤더 및 자산을 포함하는 프레임워크 번들; 동적으로 연결됨.
- **.xcframework** 여러 플랫폼 및 아키텍처에 대한 여러 .framework를 포함할 수 있는 XCFramework 번들; 동적으로 연결됨.

iOS는 기본적으로 초기 경로에 있는 모든 동적 라이브러리를 로드하므로 Apple은 가능한 경우 앱 시작 시간을 개선하기 위해 해당 형식을 사용하지 않는 것이 좋다고 권장합니다. 그러나 더 넓은 관점에서 볼 때 특정 라이브러리가 동적으로 연결되어 다양한 앱 대상에서 공유되는 것이 더 나은 경우가 많습니다.

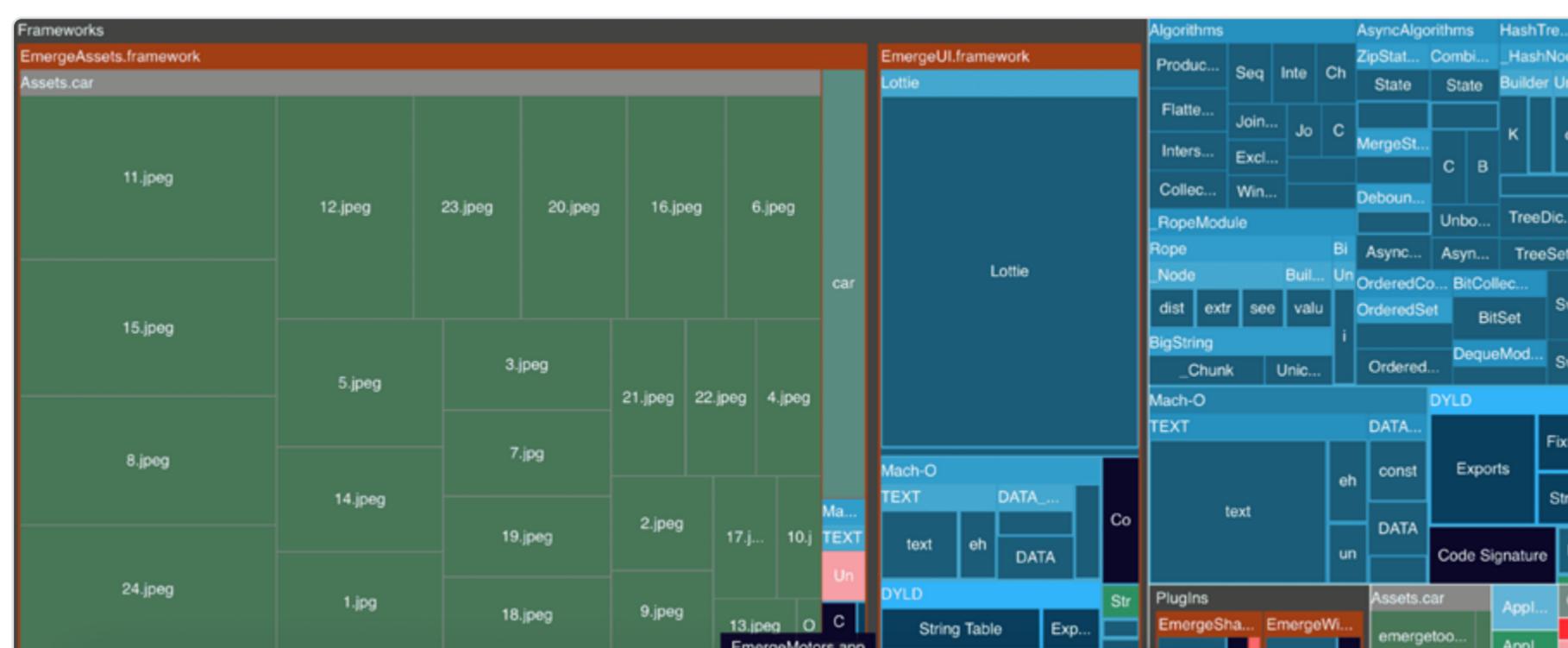
정적 연결에서 동적 연결로 전환해야 하는 경우

정적 연결을 사용하고 여러 앱 대상 간에 하나의 프레임워크를 공유하는 경우 각 대상에 대해 이 프레임워크를 복제하게 됩니다. 세 개의 대상(예: 앱, 공유 확장 프로그램 및 위젯 확장 프로그램)이 있는 경우 이 프레임워크를 앱에 세 번 포함하게 됩니다. 이것은 [Emerge Tools의 블로그 게시물](#)에서 깔끔하게 시각화됩니다.



연결 아키텍처를 변경하기 전의 Emerge Tools 스크린샷

보시다시피 EmergeUI는 각 대상에 개별적으로 포함되어 세 번 복제됩니다. 프레임워크 링크를 동적으로 만든 후에는 해당 중복성을 제거했습니다.



연결 아키텍처를 변경한 후의 Emerge Tools 스크린샷

이러한 연결 아키텍처를 달성하는 방법은 이 가이드의 범위를 벗어납니다. 프레임워크가 생성되는 방식에 따라 달라지기 때문입니다. 더 자세한 내용은 위에서 언급한 블로그 게시물을 확인하는 것이 좋습니다.

이제 모바일 React Native 앱에서 찾을 수 있는 다양한 번들과 연결 방법에 대해 조금 더 알게 되었으므로 번들의 내용을 분석하고 해당 정보를 사용하여 앱 시작 시간을 더욱 최적화하는 방법을 살펴보겠습니다.

GUIDE

JS 번들 크기를 분석하는 방법

React Native 앱에서는 다른 JavaScript 기반 프레임워크와 마찬가지로 실제로 JS 번들로서 사용자에게 무엇을 제공하는지 이해하는 것이 중요합니다. 최종 JavaScript 파일은 일반적으로 자체 코드, 타사 종속성 코드 및 해당 종속성(이를 "전이 종속성"이라고 함)으로 구성됩니다. 해당 파일의 크기가 앱 초기화 경로에 다소간 영향을 미치므로 JavaScript를 더 작게 만드는 방법을 배우는 것이 좋습니다. 그렇게 하는 방법은 여러 가지가 있지만 모두 동일한 아이디어를 중심으로 이루어집니다. 즉, 해당 시점에 필요하지 않은 코드를 제거하는 것입니다. 그리고 타이밍이 모든 차이를 만들습니다.

```
// 이 코드는 프로덕션 빌드에서 제거할 수 있습니다.  
if (__DEV__){  
    // ...  
}
```

```
// 이 코드는 Android 번들에서 제거할 수 있습니다.  
if (Platform.OS === 'ios') {  
    // ...  
}
```

특정 상황에서만 필요한 코드의 예: 개발 중이거나 iOS에서만 필요한 경우

예를 들어, 웹 개발자는 브라우저 엔진이 JS를 다운로드, 메모리에 로드, 파싱 및 실행해야 하기 때문에 브라우저에서 초기에 다운로드되는 JS의 크기를 중심으로 최적화합니다. 이는 많은 단계이며, 각 단계를 최적화할 수 있습니다. 그러나 모바일 앱을 위해 설계된 JavaScript 엔진인 Hermes를 실행하는 최신 React Native 앱에서는 다릅니다.

Hermes 바이트코드

2022년 이후에 앱을 업데이트했고 React Native 0.70 이상을 실행 중이라면, 최종 앱 번들에는 웹 앱처럼 원시 JavaScript 코드가 포함되어 있지 않을 가능성이 높습니다. React Native에 기본적으로 포함된 Hermes JS 엔진에서 생성된 바이트코드 파일을 찾을 수 있을 것입니다.



Hermes는 React Native 앱에서 사용 가능한 바이트코드를 생성하는 유일한 JS 엔진입니다. V8과 같은 다른 엔진은 힙 스냅샷을 활용하여 유사한 사전 최적화 기술을 기술적으로 적용 할 수 있습니다.

저사양 모바일 장치를 위해 설계된 엔진인 Hermes의 핵심 전제는 기본적으로 "메모리에 로드" 및 "파싱" 단계를 빌드 시간으로 옮기는 것입니다.

이는 웹 개발에서 사용되는 동일한 트릭, 특히 코드 분할 또는 초기 번들 크기를 위해 집중적으로 최적화하는 것이 React Native 개발로 직접적으로 이어지지 않는다는 것을 의미합니다. 그 이유는 React Native는 다운로드 단계를 건너뛸 수 있고 (JS가 이미 앱 번들에 있음), 특히 Hermes 덕분에 JS를 메모리에 로드하고 파싱하는 단계도 건너뛰기 때문입니다.

바이트코드가 불필요하게 커지거나 필요 이상으로 더 많은 JS를 실행하는 것을 방지하기 위해 더 작은 라이브러리를 선택하고, 레거시 코드를 제거하고, 초기화 경로에서 필요한 라이브러리만 로드하는 것은 여전히 중요합니다. 바이트코드 크기는 앱 번들 크기에 영향을 미쳐 다운로드 및 초기화 속도가 느려집니다. 초기화 경로에서 더 많은 JS가 실행되면 TTI(Time To Interactive) 지표에 영향을 미칩니다.

더 나쁜 것은 이 JavaScript가 네이티브 모듈을 로드할 때 네이티브 측에서 더 많은 작업을 발생시키고 터보 모듈의 자연 로딩의 이점을 줄인다는 것입니다.



터보 모듈은 기본적으로 자연 로드됩니다. 앱 시작 시 적극적으로 초기화되는 레거시 React Native 아키텍처의 네이티브 모듈과 비교하여 터보 모듈은 JavaScript 측에서 호출될 때만 실행됩니다. 이 메커니즘은 새로운 아키텍처로 전환하는 앱의 TTI를 눈에 띄게 향상시킵니다.

수백 개의 컴포넌트가 있는 컴포넌트 라이브러리를 임포트했지만, 실제로는 12개 정도만 사용한다고 상상해 보세요.

이 컴포넌트들을 배럴 파일 (단일 파일에서 모든 것을 익스포트, ["Barrel Export 피하기" 챕터](#)에서 자세히 알아볼 수 있습니다)로부터 임포트하고, Tree Shaking이나 Babel 플러그인을 통한 최적화도 적용하지 않았다고 가정합니다. 앱의 초기화 경로에서, 앱은 그 모든 컴포넌트들을 로드할 것이고, 그 중 일부는 자체 Turbo Modules를 호출할 수도 있습니다. 그러면 그 Turbo Modules들이 사용되어야 할 시점인 나중이나 예전 사용되지 않는 대신, 지금부터 초기화를 시작해 버릴 것입니다.

그렇다면, JavaScript가 필요 이상으로 커지게 만드는 요소를 더 잘 파악하기 위해 JavaScript를 어떻게 검사할 수 있을까요?

source-map-explorer

source-map-explorer 같은 도구를 사용하여 해당 소스 맵을 가진 JavaScript 번들을 웹 UI에서 분석할 수 있습니다. 이 도구는 사용자 장치에 최종적으로 어떤 것들이 포함되는지 한눈에 보여주어, 의도치 않게 포함되었거나 레거시 기능의 잔재인 이상치들을 쉽게 식별할 수 있도록 해 줍니다.

이 도구의 훨씬 더 좋은 점은, 여러분 자신의 코드 크기에 대한 아이디어를 얻을 수 있다는 점입니다. 여기서 타사 라이브러리의 경우만큼이나 JS 번들 크기를 줄일 수 있는 기회를 많이 찾아 낼 수 있습니다.

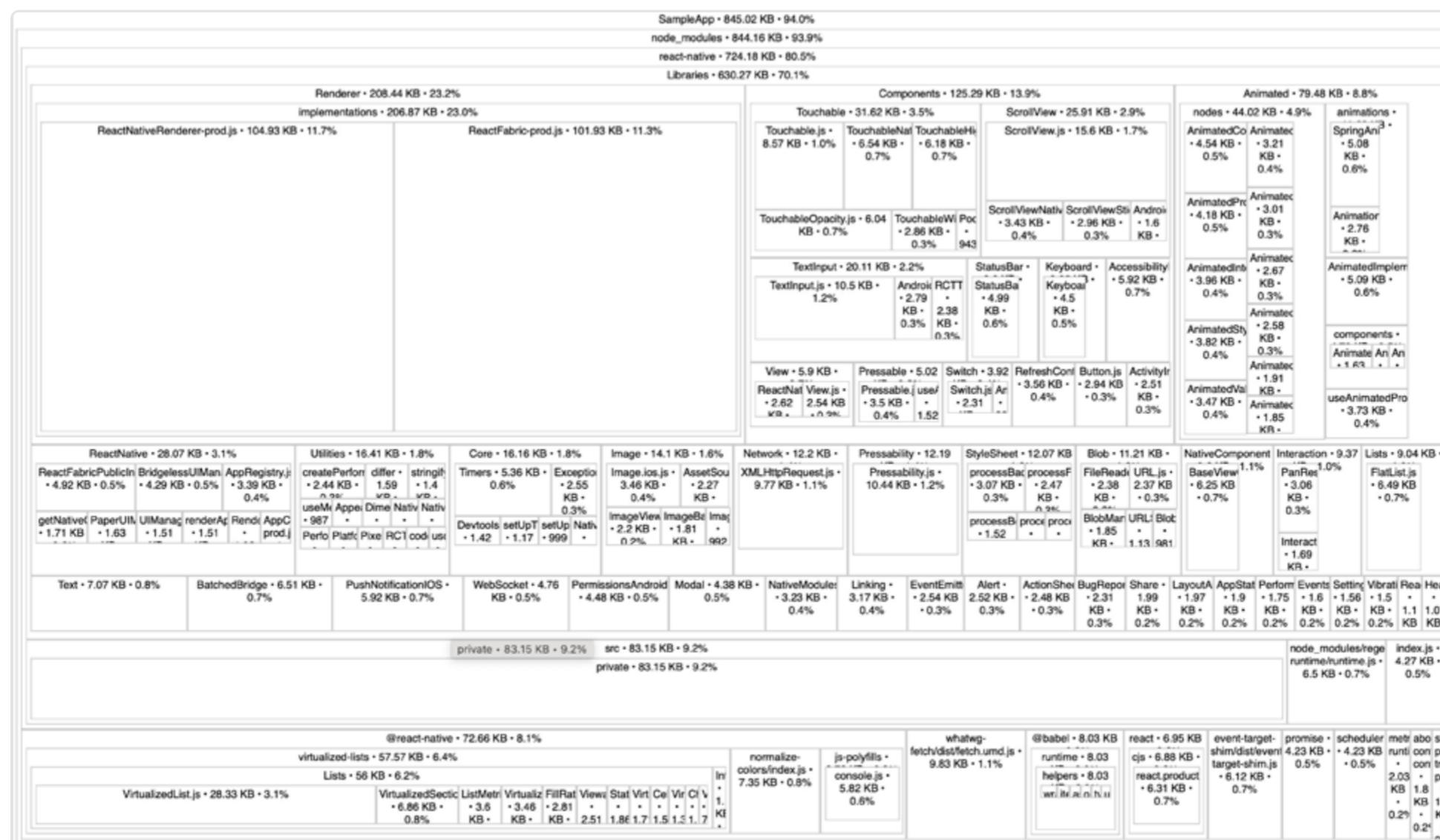
이 도구를 사용하려면 먼저 JavaScript를 번들링하고 소스 맵을 생성해야 합니다.

```
> npx react-native bundle \
  --entry-file index.js \
  --bundle-output output.js \
  --platform ios \
  --sourcemap-output output.js.map \
  --dev false \
  --minify true
```

JavaScript가 생성되면 source-map-explorer 를 실행할 수 있습니다.

```
> npx source-map-explorer output.js --no-border-checks
```

Metro는 유효하지 않은 컬럼/라인 매핑을 생성하므로, `--no-border-checks` 플래그를 전달하여 유효성 검사 체크를 비활성화해야 합니다. 다행히도, 이러한 유효하지 않은 매핑이 도구가 UI를 생성하는 것을 막지는 않습니다. 명령이 완료되면 다음과 유사한 출력을 가진 웹 브라우저가 열릴 것입니다:



source-map-explorer 출력의 웹 UI

나쁜 소식은 유효하지 않은 매핑으로 인해 최대 30%의 정보를 잃을 수 있으며, 이는 추가 최적화 기회를 숨길 수 있다는 것입니다. 번들에서 훨씬 더 많은 정보를 얻으려면 Metro 파이프라인에 긴밀하게 통합되는 Expo Atlas를 사용할 수 있습니다.

Expo Atlas

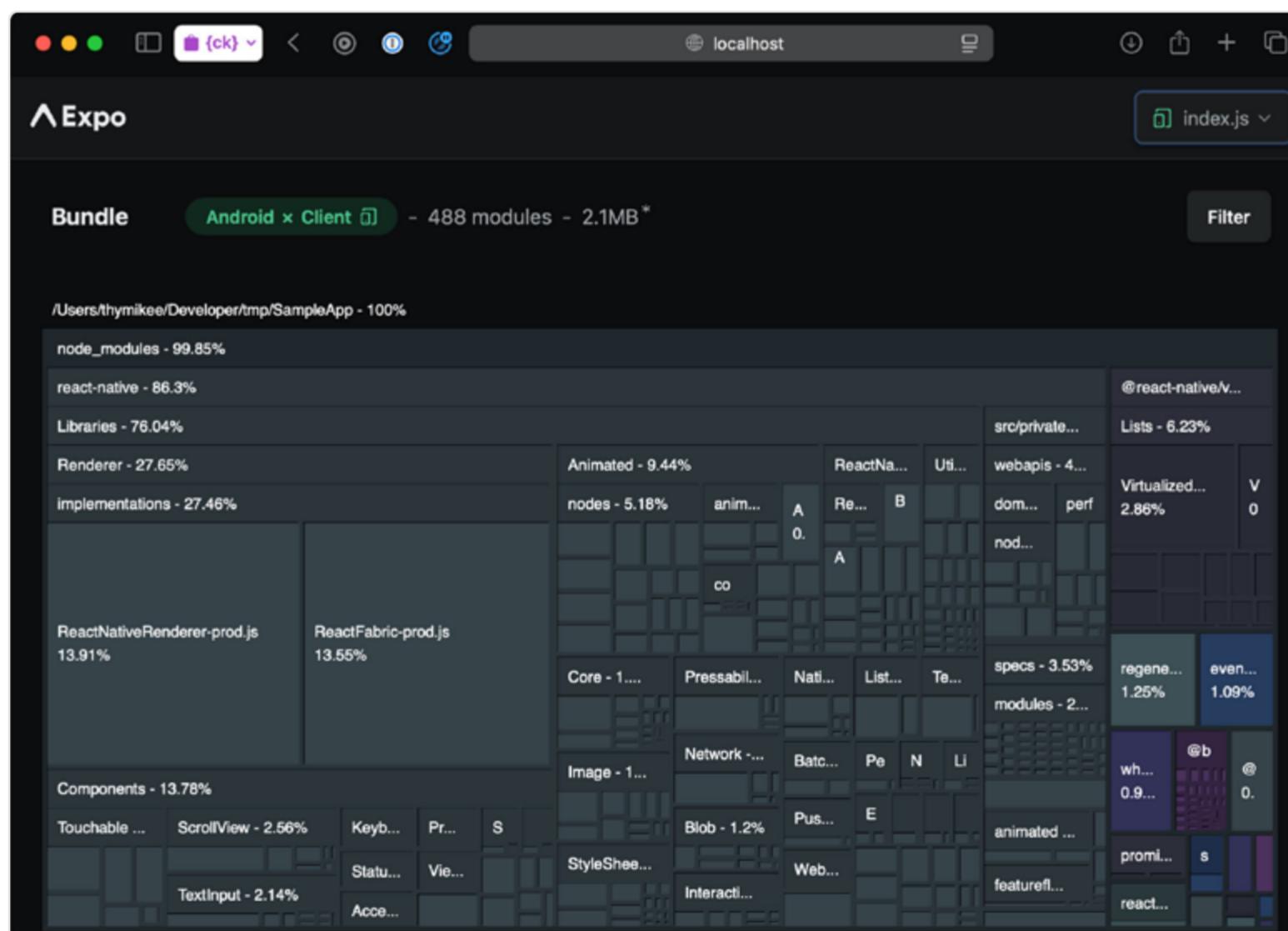
프로덕션 번들을 시각화하고 어떤 라이브러리가 번들 크기에 기여하는지 식별하는 데 사용할 수 있는 또 다른 도구는 Expo Atlas입니다. 이것은 Expo 프로젝트를 위한 선택 사항이지만, Expo가 아닌 프로젝트에서도 사용할 수 있는 약간의 편법적인 방법들이 있습니다. 이를 사용하려면 EXPO_UNSTABLE_ATLAS 환경 변수를 설정하여 Expo CLI를 실행하십시오:

```
> EXPO_UNSTABLE_ATLAS=true npx expo start --no-dev
# or
EXPO_UNSTABLE_ATLAS=true npx expo export
```

그런 다음 다음 명령을 실행할 수 있습니다:

```
> npx expo-atlas
```

이렇게 하면 번들을 검사할 수 있는 Expo Atlas의 UI가 열립니다.



다음은 Expo Atlas 출력의 웹 UI입니다. 재미있는 사실은, 이 보고서는 Expo 프로젝트가 아닌 프로젝트에서 생성되었다는 것입니다.

- 💡 **expo-atlas-without-expo** 프로젝트를 사용하여 Expo CLI 없이도 Expo Atlas를 실행하는 방법이 있습니다.

Rspack 번들 분석

Metro를 대체하여 Webpack 또는 Rspack으로 Re.Pack을 사용하는 경우, JS 번들을 검사하기 위한 더 광범위하고 고품질의 도구를 경험할 수 있습니다. Webpack 생태계와 거의 동등하고 우수한 성능을 제공하는 Rspack에 집중할 것입니다.

- 💡 지금부터 논의하는 모든 도구는 이미 논의된 도구와 유사하게 작동하므로 간결성을 위해 스크린샷을 생략했습니다.

webpack-bundle-analyzer

Rspack의 CLI는 `--analyze` 옵션을 통해 즉시 번들 분석을 지원합니다. 이는 내부적으로 `webpack-bundle-analyzer`를 사용합니다.

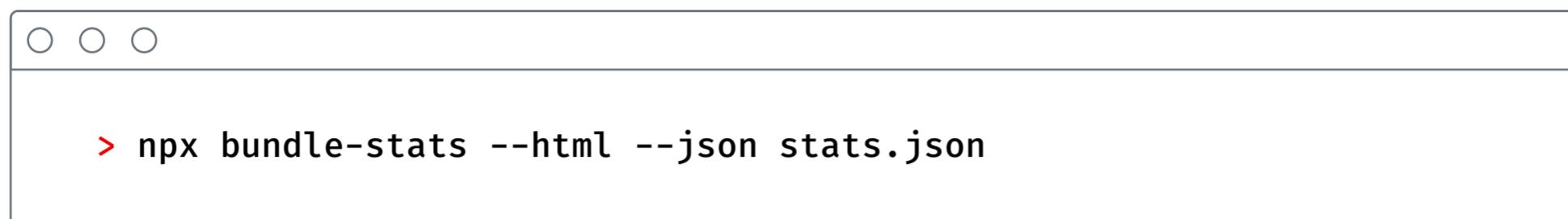
```
○ ○ ○  
➤ rspack build --analyze
```

bundle-stats 및 statoscope

추가적인 분석을 위해 `bundle-stats` 또는 `statoscope`와 같은 다른 번들 분석 도구와 함께 `stats.json` 파일을 생성 할 수도 있습니다. 이를 위해 Re.Pack의 번들 명령에 `--json stats.json` 플래그를 추가하십시오:

```
○ ○ ○  
➤ $ npx react-native bundle \  
  --platform android \  
  --entry-file index.js \  
  --dev false \  
  --minify true \  
  --json stats.json
```

그런 다음, **Statoscope**을 사용할 때, 생성된 **stats.json** 파일을 **bundle-stats** 명령에 전달 할 수 있습니다.



이제 웹 앱에서 풍부한 HTML 보고서를 즐길 준비가 되었습니다! 또한 하나의 통계 파일을 다른 통계 파일과 비교하여 최종 번들이 시간이 지남에 따라 크기가 증가 또는 감소했는지에 대한 통찰력을 얻을 수 있습니다.

Rsdoctor의 번들 분석

Rsdoctor는 리소스 크기, 중복 패키지 및 모듈 참조 관계를 포함하여 **Rspack** 출력 정보를 분석하는 데 주로 사용되는 번들 크기 모듈을 제공합니다. 이를 사용하려면 [@rsdoctor/rspack-plugin](#) 종속성을 설치하고 이를 **rspack.config.js**에 플러그인으로 추가하십시오.

```
const { RsdoctorRspackPlugin } = require('@rsdoctor/rspack-plugin');

module.exports = {
  plugins: [
    process.env.RSDOCTOR &&
      new RsdoctorRspackPlugin({
        // plugin options
      }),
    ].filter(Boolean),
};
```

다음에 **RSDOCTOR** 환경 변수를 **true**로 설정하여 앱을 실행하면, **source-map-explorer** 와 유사하게 **Rsdoctor** UI가 열립니다.

GUIDE

앱 번들 크기 분석 방법

앱의 크기를 작게 유지하는 것은 특정 플랫폼에서 앱 성공에 매우 중요할 수 있습니다. 사용자는 특히 모바일 데이터에 의존할 때 설치 크기만으로 다른 앱을 선택할 수 있습니다. Google은 APK 크기와 다운로드 수의 상관 관계를 측정하는 실험을 수행했으며, APK 크기가 6MB 증가 할 때마다 설치 전환율은 1% 감소한다는 것을 발견했습니다.

앱 크기는 네트워크 및 저장 공간 제한과 같이 다른 방식으로 사용자에게 영향을 미칩니다. 네트워크 관련 메트릭은 다운로드 크기와 업데이트 크기이며, 이는 인터넷을 통해 전송되는 앱 축 데이터의 양을 나타냅니다. 저장 공간 관련 메트릭(예: 설치 크기 및 저장 크기)은 장치에 저장된 앱 및 캐시의 압축되지 않은 크기를 나타냅니다.

Android 앱 크기 측정

Android 생태계는 앱 크기를 검사하는 여러 편리한 방법을 제공합니다. 특히 사용하기 쉽기 때문에 Spotify의 Gradle 플러그인인 Ruler를 선호합니다. 이 플러그인은 기존 앱에 통합되어 특정 아키텍처에서 앱 번들을 분석, 디버깅 및 배포할 수 있습니다.

설치에는 Gradle 빌드 설정 조정이 필요합니다. 먼저 상위 레벨 build.gradle 파일의 빌드스크립트 classpath에 Ruler Gradle 플러그인을 추가하세요.

```
buildscript {  
    // ...  
    dependencies {  
        // ...  
        classpath("com.spotify.ruler:ruler-gradle-plugin:2.0.0-  
beta-3")  
    }  
}
```

그 다음, 앱/build.gradle 파일 상단부분에 플러그인을 추가하십시오.

```
apply plugin: "com.spotify.ruler"
```

나중에, 동일한 파일에서 원하는 아키텍처, ABI, 언어 설정, SDK 버전 등을 검사할 Ruler 구성 값을 추가하십시오. 자세한 내용은 [공식 문서를 참조하십시오](#).



ABI는 Application Binary Interface의 약자로, 장치 아키텍처와 연결되어 있으며, arm64-v8a 또는 x86과 같은 것이 있습니다. 우리는 [소개](#)에서 이러한 아키텍처에 대해 다루었습니다.

예시 구성은 다음과 같습니다:

```
ruler {  
    abi.set("arm64-v8a")  
    locale.set("en")  
    screenDensity.set(480)  
    sdkVersion.set(34)  
}
```

이제 새로운 Gradle 작업이 잠금 해제됩니다:

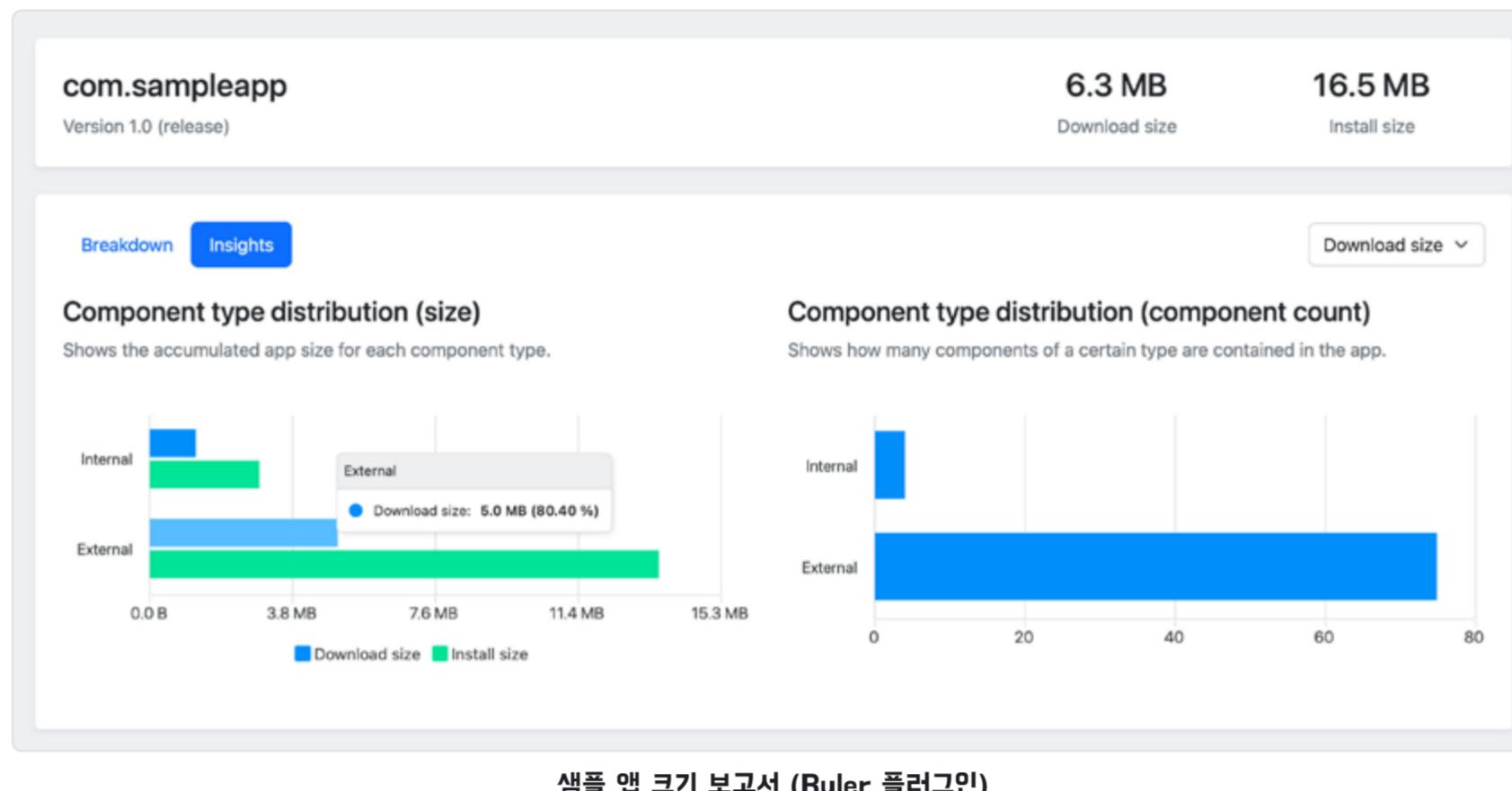
analyzeDebugBundle 및 analyzeReleaseBundle. 이러한 작업을 수행하려면 터미널에서 Android 폴더에 들어가 Gradle Wrapper (`./gradlew`)를 직접 실행하면 됩니다.

```
○ ○ ○  
> cd android  
.gradlew analyzeReleaseBundle
```

성공적으로 완료되면 다음 출력이 생성되며, 웹 브라우저에서 열 수 있는 report.html 파일이 가리키는 위치가 나타납니다.

```
○ ○ ○  
> 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.

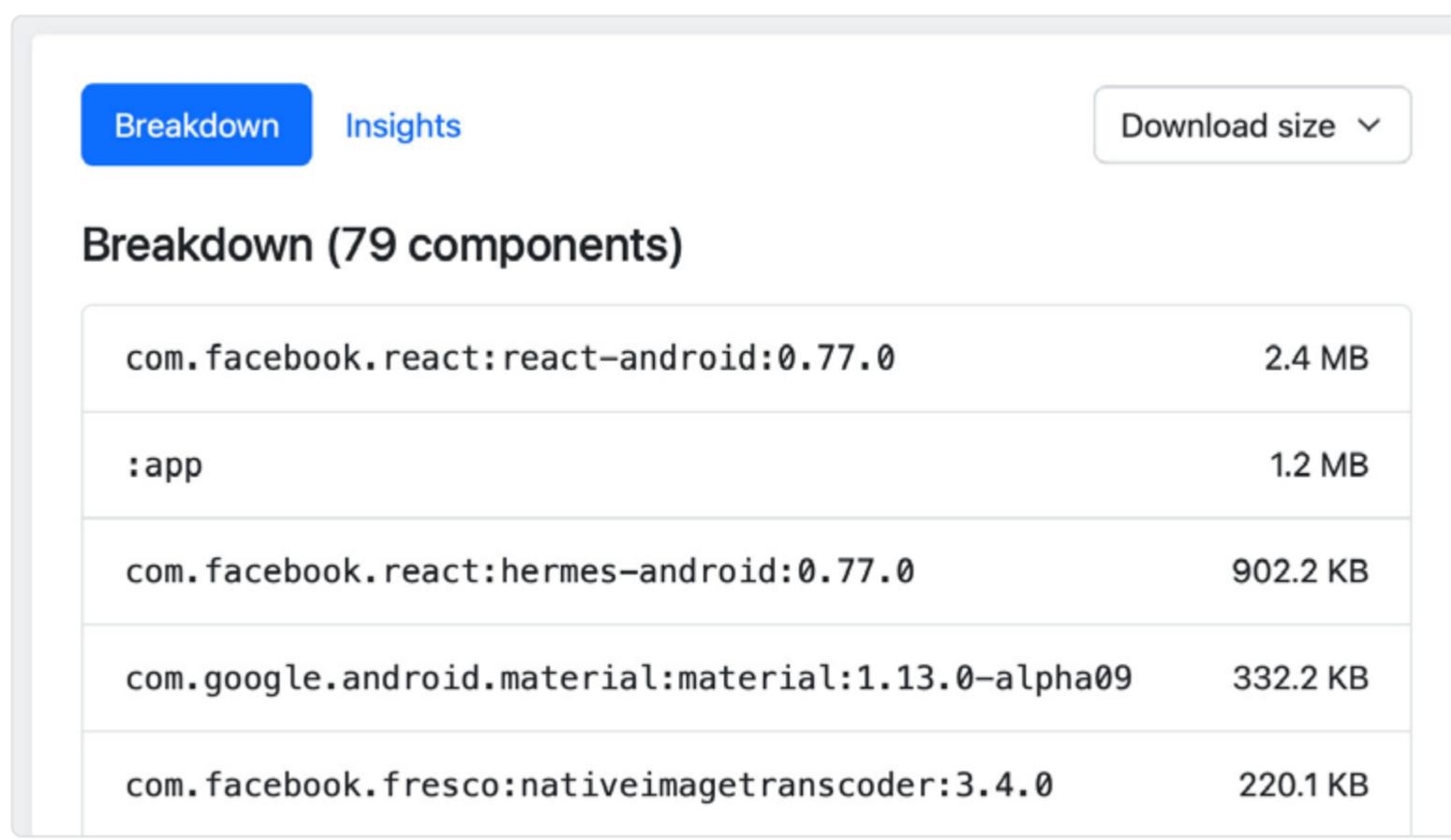


샘플 앱 크기 보고서 (Ruler 플러그인)



이 샘플 앱은 R8에 의해 최적화되어 다운로드 크기가 6.3MB입니다. 이러한 빌드 타임 최적화가 없는 동일한 앱의 경우 9.5MB입니다. 더 자세히 알고 싶다면 “Shrink Code With R8 Android” 챕터를 참조하십시오.

또한 Ruler는 가장 큰 것부터 가장 작은 것 순으로 내부 및 외부 구성 요소의 상향식 분해를 보여줍니다.



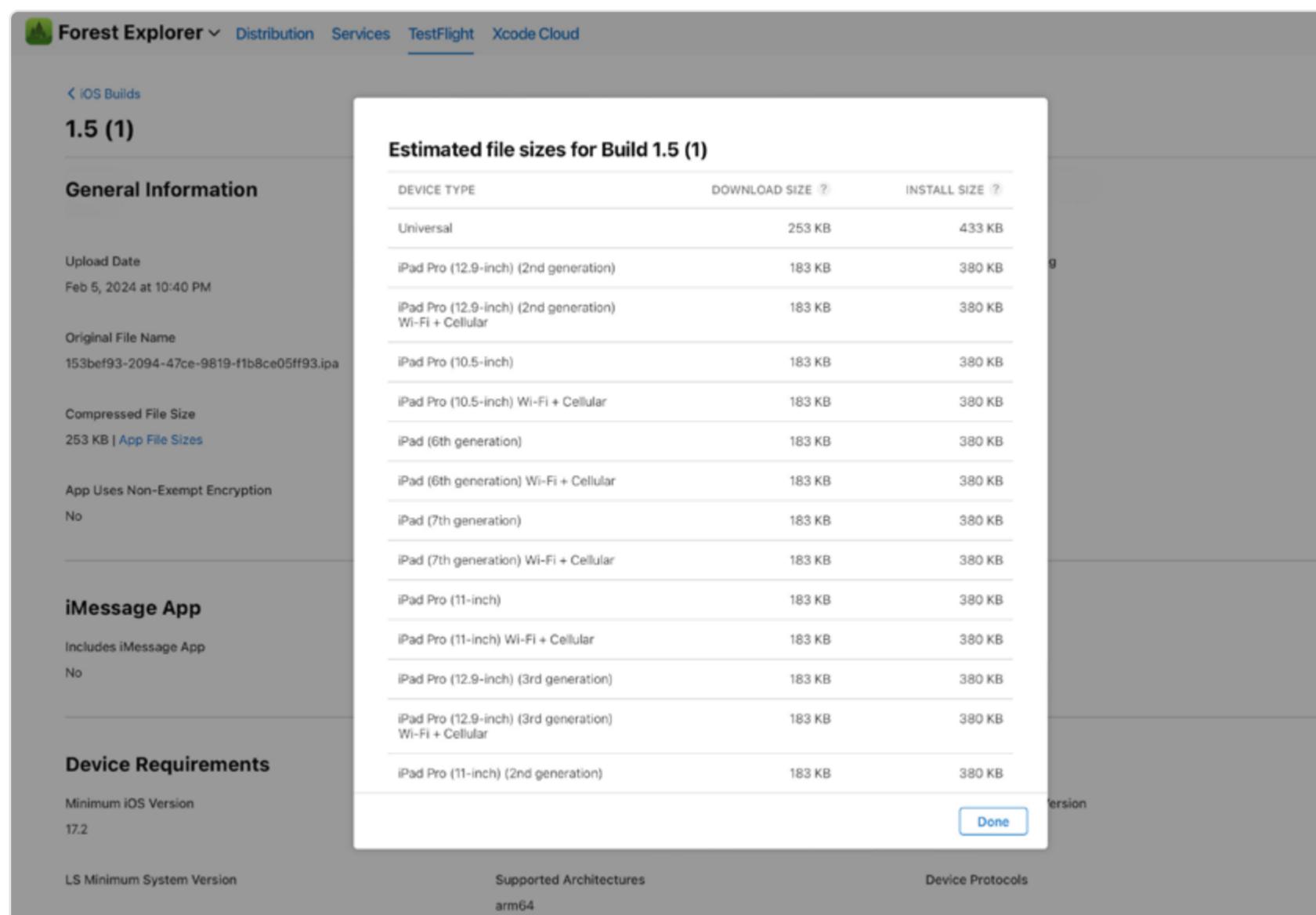
Android 앱 구성 요소 크기 분해

이상적으로는 CI 파이프라인에 크기 검사를 추가하여 크기를 자동으로 모니터링하고, 예를 들어 PR 아래에 GitHub 댓글을 보내는 것을 원합니다. Ruler를 사용하면 별도의 도구가 없어도 크기에 대한 유효성을 검사하고 앱이 너무 크면 빌드를 실패하도록 설정할 수 있습니다.

```
ruler {  
    // ...  
    verification {  
        downloadSizeThreshold = 20 * 1024 * 1024 // 20 MB in  
        bytes  
        installSizeThreshold = 50 * 1024 * 1024 // 2 MB in bytes  
    }  
}
```

iOS 앱 크기 측정

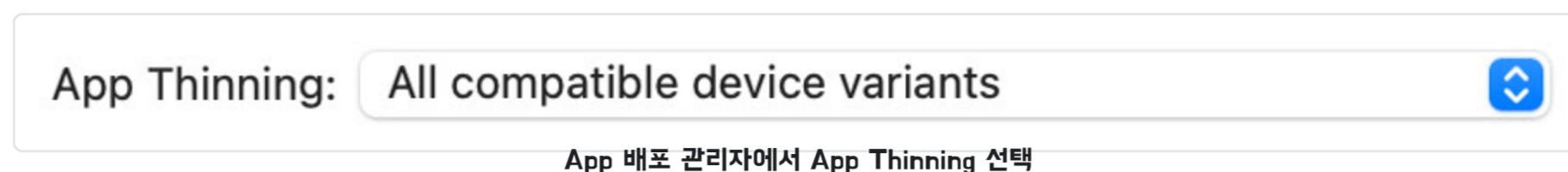
앱이 App Store 또는 TestFlight를 통해 배포되어 있다면 App Store Connect가 가장 정확한 크기 정보를 제공합니다. 각 앱 버전별 크기를 표시하고 모바일 인터넷 연결을 통한 다운로드 제한을 초과하는 경우 경고합니다.



App Store Connect에서 제공하는 다양한 변종 크기를 확인할 수 있는 표

TestFlight 빌드는 테스트를 위한 추가 데이터를 포함하여 App Store 버전보다 크기가 클 수 있습니다. 하지만 App Store 처리 과정(DRM 추가 및 재압축 포함) 이후, 최종 앱 크기는 업로드된 원본 이진 파일보다 약간 증가할 수 있습니다. 이는 앱 배포를 안전하게 보호하고 최적화하기 위해 추가적으로 수행되는 작업 때문입니다.

App Store Connect는 정확한 측정을 제공하지만, Xcode 개발 과정 중에도 다운로드 및 설치 크기에 대한 근사치를 제공하는 앱 크기 보고서를 생성할 수 있습니다. 앱 크기 보고서를 생성하려면 먼저 Xcode에서 서명된 IPA 파일을 생성해야 합니다. 그런 다음, Organizer에서 “배포 앱” 버튼을 클릭하고 “사용자 지정”을 선택한 후, 원하는 배포 옵션을 선택하고 “모든 호환 기기 변종” 옵션을 App Thinning 설정에서 활성화해야 합니다.



Xcodebuild를 사용하여 이 모든 것을 처리하려면 ExportOptions.plist 파일에서 해당 설정을 확인하십시오.

```
<key>thinning</key>
<string>&lt;thin-for-all-variants&gt;</string>
```

이 프로세스는 앱의 아티팩트가 담긴 폴더를 생성합니다:

- 더 이상 지원되지 않는 이전 기기용의 유니버설 IPA 파일. 이 단일 IPA 파일에는 앱의 모든 변종의 자산과 실행 파일이 포함되어 있습니다.
- 각 앱 변종에 대한 얇은 IPA 파일. 이 파일에는 하나의 변종에만 자산과 실행 파일이 포함되어 있습니다.

내보낸 앱 폴더에는 앱 크기 보고서도 포함되어 있습니다. “App Thinning Size Report.txt”라는 이름의 파일입니다. 이 보고서는 앱의 IPA 파일 각각에 대한 압축 및 압축 해제 크기를 나열합니다. 압축 해제 크기는 장치에 설치된 앱의 크기와 동일하며, 압축 크기는 앱의 다운로드 크기입니다.

다음은 샘플 앱의 앱 크기 보고서의 시작 부분입니다:

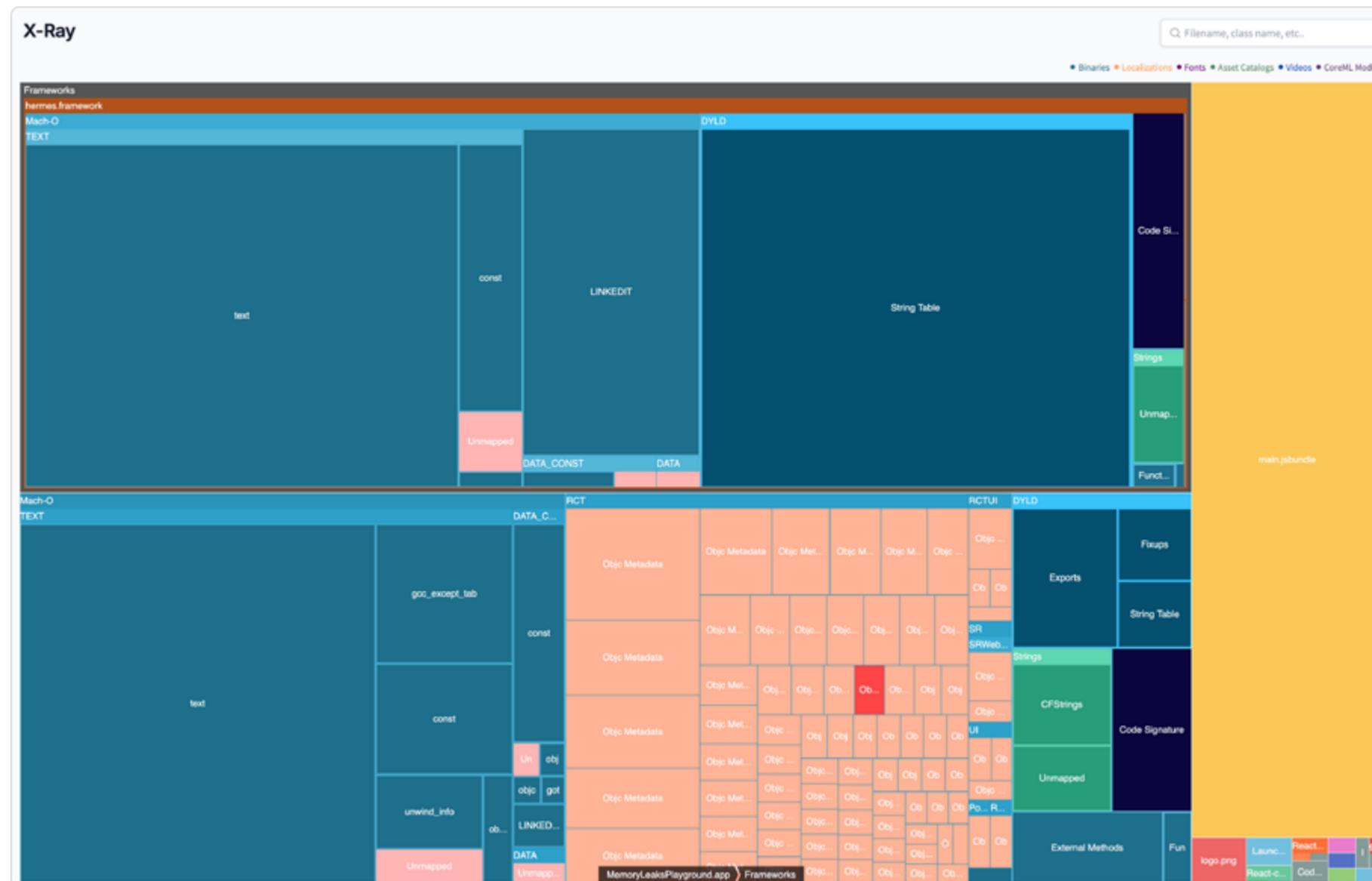
```
> App Thinning Size Report for All Variants of SampleApp

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 도구

이 가이드의 저자들은 Emerge Tools와 어떠한 금전적 관계도 없습니다. Emerge Tools는 유료 서비스이며, 14일 무료 평가판을 제공합니다. 그러나 앱 크기 검사 시 유용하게 판명되는 이 도구를 사용하고 있으므로, iOS 및 Android 앱 모두에 작동하는 실행 가능한 대안으로 여기에 목록을 추가하고 싶습니다.

IPA, APK 또는 AAB 파일을 업로드하면 X-Ray 또는 Breakdown 도구를 통해 다음과 같은 UI와 유사한 화면에 앱 정보가 표시됩니다.



Emerge 도구의 X-Ray 도구를 사용하여 IPA 파일의 샘플 보고서입니다.

소스 맵 탐색기와 마찬가지로, JS 모듈 및 라이브러리를 폴더 또는 모듈 경계별로 그룹화된 사각형 형태로 제시하여 살펴볼 수 있는 것처럼, X-Ray 도구는 Android 및 iOS 번들(bundle)에 대한 이진 정보를 유사한 형태로 제시합니다.

이 도구는 몇 가지 인사이트를 제공하지만, 가장 신뢰할 만한 도구라고 보기에는 어렵습니다. 예를 들어, Hermes JS 엔진 전체를 제거할 것을 제안하지만, 이는 앱에 상당히 부정적인 영향을 미칠 것입니다.

Insights

2 insights

Strip binary symbols
Potential savings 2.4 MB (21.84%)

These binaries have unnecessary symbols in the final build. Strip them out to save space. [Learn more](#)

-2.4 MB /Frameworks/hermes.framework/hermes

조심하세요. 모든 인사이트가 실제로 구현할 가치가 있는 것은 아니니까요.

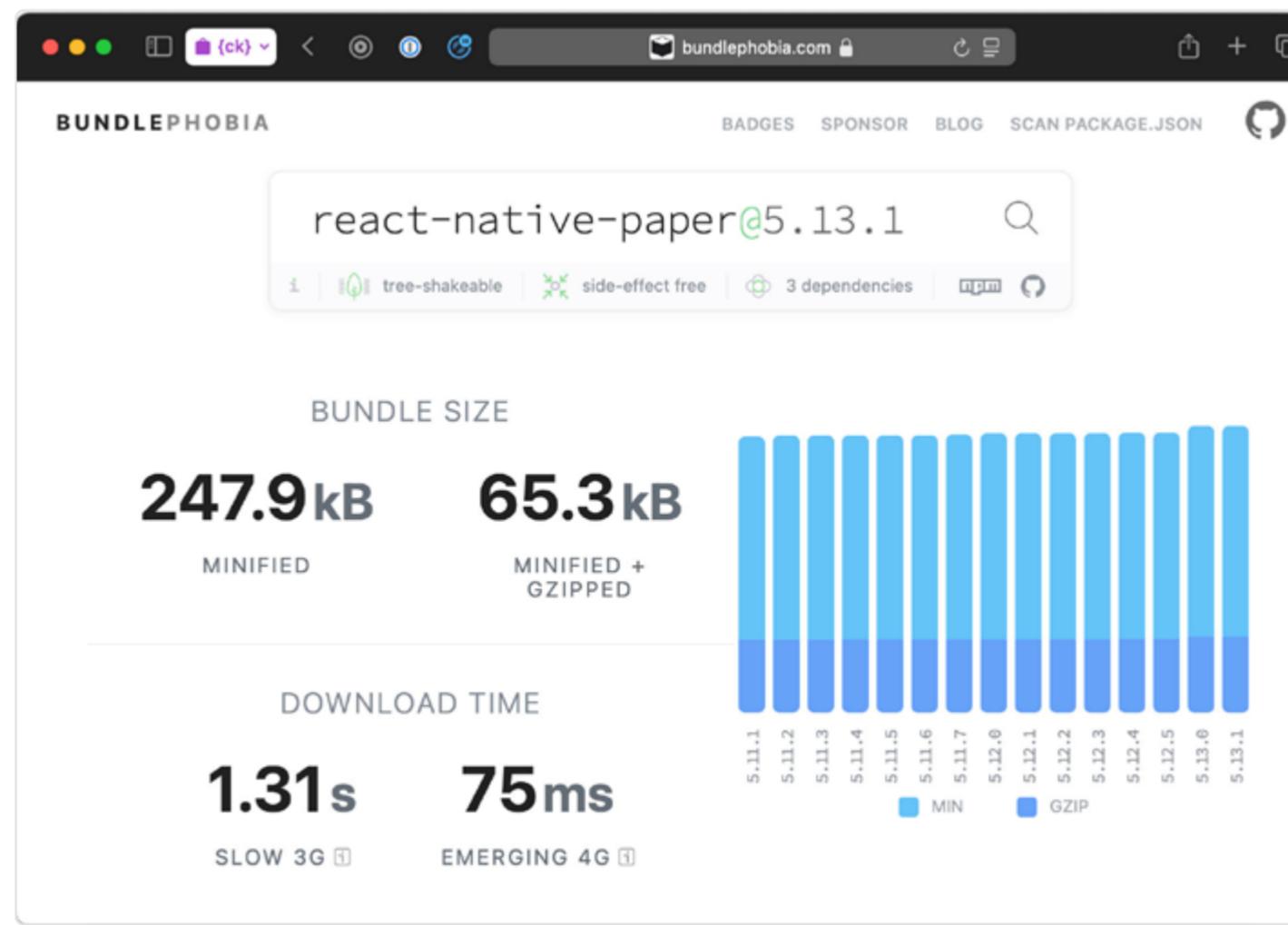
BEST PRACTICE

서드파티 라이브러리의 실제 사이즈 파악하기

JavaScript 생태계에서는 똑같은 일을 두 번 하는 것을 싫어합니다. 오픈 웹 플랫폼의 오픈 소스 문화와 결합하면 npm(Node Package Manager 및 온라인 패키지 레지스트리)이 수백만 개의 재사용 가능한 JavaScript 라이브러리를 보유하고 있어 앱을 더 빠르게 만들 수 있으며, 종종 node_modules 폴더에 기가바이트를 차지하게 되는 것도 놀라운 일이 아닙니다. JS 번들러 덕분에 그중 일부만 React Native 앱에 들어갑니다. 이 장에서는 프로젝트의 타당한 외부 의존성 크기를 파악하는 데 저희가 매일 사용하는 도구를 공유하고자 합니다.

bundlephobia.com

이 웹사이트는 저희가 가장 좋아하는 웹사이트 중 하나입니다. npm에 있는 패키지를 입력하면 축소된 크기, 압축된 크기 및 다운로드 시간과 같은 정보를 보여줍니다 – 특히 웹 앱의 경우 유용합니다.



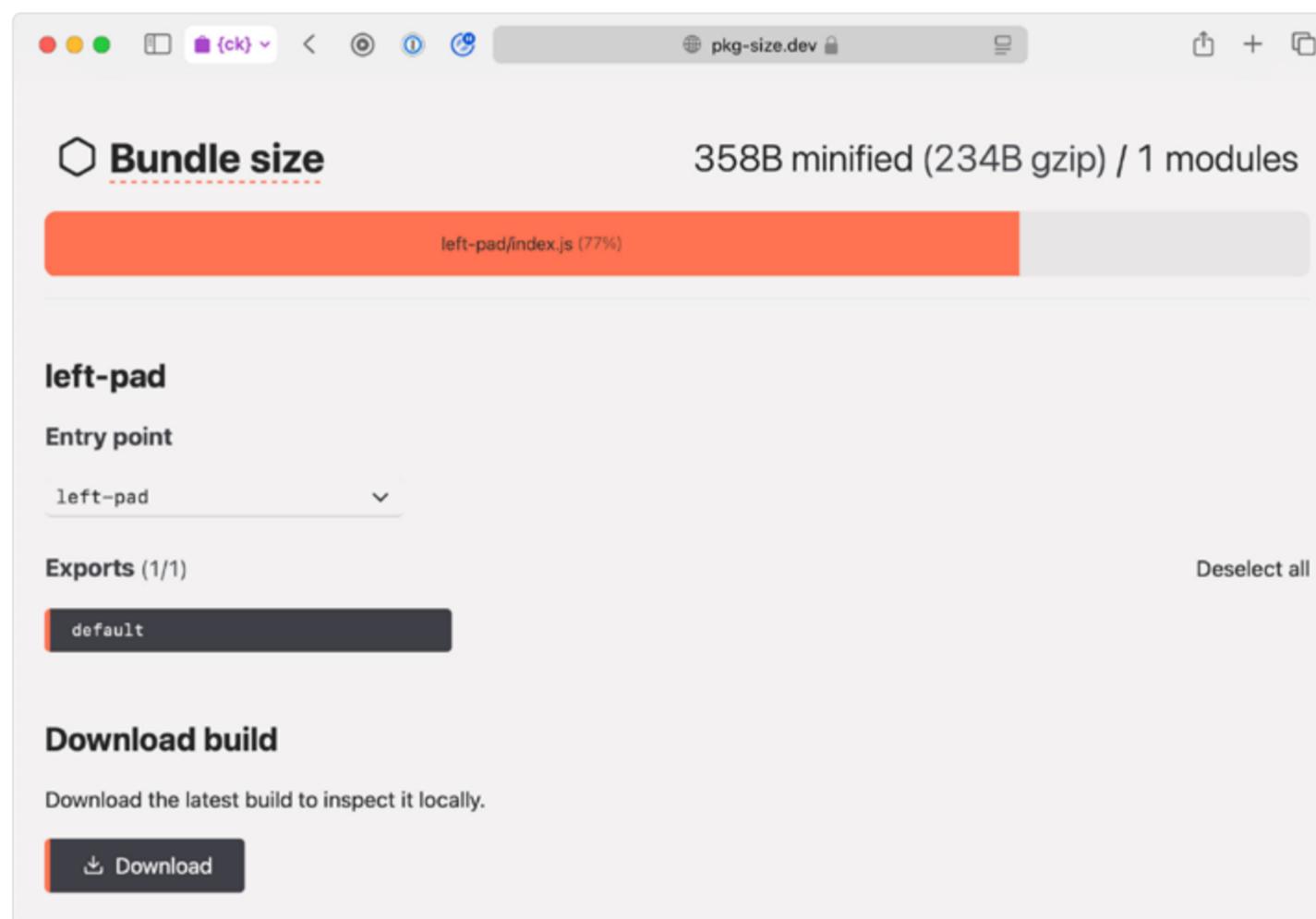
[bundlephobia.com 웹사이트 스크린샷 – react-native-paper 의존성 표시](#)

Bundlephobia의 또 다른 편리한 기능은 패키지 및 의존성 분해 기능으로, 뒤에서 이 장에서 설명하는 `react-native-bundle-visualizer`과 유사합니다.

pkg-size.dev

때때로 Bundlephobia가 일부 패키지에서는 작동하지 않거나 장애가 발생하는 경우도 있습니다.

이러한 경우, 저희는 백업 계획을 가지고 있습니다: pkg-size.dev입니다. 이는 저희 패키지를 웹 컨테이너에 설치하고 축소되고 압축된 패키지 크기를 보여주는 유용한 서비스입니다.



A screenshot of pkg-size.dev website showing left-pad dependency

Import Cost VS Code 확장

위에 설명된 모든 도구의 UX는 간헐적인 단회성 검사에 초점을 맞추고 있습니다. 하지만 실시간 피드백이 필요할 수도 있습니다. 그러한 경우에는 최적의 방법은 선택한 IDE와 직접 통합하는 것입니다.

VS Code 또는 Cursor와 같은 파생 버전을 사용하는 경우, [Import Cost](#) 확장을 사용하여 이를 달성할 수 있습니다.

```
import React, {useMemo, memo} from 'react'; 6.9k (gzipped: 2.7k)
import {View, Text, StyleSheet, Pressable} from 'react-native';

'react' 패키지에서 가져온 임포트의 크기를 보여주는 주석
```

Import Cost는 Webpack을 사용하여 임포트를 처리하는 방식으로 작동합니다. 하지만 이 번들러는 React Native에 대한 기본 지원이 부족하여, 네이티브 코드를 사용하는 패키지에서는 확장 프로그램이 종종 오류가 발생합니다. 그럼에도 불구하고, 나머지 패키지에는 유용하게 활용할 수 있습니다.



[Re.Pack](#)을 통해 Webpack 또는 Rspack을 React Native 프로젝트의 번들러로 활용할 수 있습니다.

BEST PRACTICE

Barrel Export 피하기

JavaScript에서 배럴 파일은 여러 모듈을 단일 파일에 그룹화하고 내보내는 방법입니다. 이를 통해 그룹화된 모듈을 보다 쉽게 가져올 수 있도록 중앙 집중식 액세스 지점을 제공합니다. 이러한 유형의 임포트는 일반적으로 “인덱스 파일”에서 나오는데, 이는 여러 디렉터리에서 파일들을 모아놓은 것입니다.

```
// components/index.ts -- an index file
export { Button } from './Button';
export { Card } from './Card';
export { Modal } from './Modal';
```

이 모듈들은 이후 단일 진입점을 통해 쉽게 가져올 수 있습니다.

```
import { Button, Card, Modal, Sidebar } from './components';
```

배럴 임포트와 내보내기의 문제점

배럴 내보내기와 임포트는 편리해 보일 수 있지만, 여러 가지 심각한 문제점을 야기합니다.

번들 크기 증가 (Bundle Size Overhead)

Metro 번들러는 배럴 파일에서 모든 모듈을 최종 번들에 포함시키기 때문에, 실제로 사용하지 않는 모듈까지 포함될 수 있습니다. 예를 들어, 배럴 파일을 통해 Button 컴포넌트만 임포트했다고 해도 Metro는 Card, Modal 컴포넌트와 배럴 파일에서 내보낸 모든 컴포넌트의 코드까지 번들 안에 포함시킵니다.



React Native Paper과 같은 일부 제3자 라이브러리는 사용하지 않는 임포트를 제거하기 위한 전용 Babel 플러그인을 내보냅니다. 따라서 사용하지 않는 컴포넌트에 대해 걱정할 필요가 없습니다.

런타임 오버헤드

배럴 임포트를 사용할 경우, 배럴 파일 내의 모든 모듈이 요청되기 전에 평가되어야 합니다. 즉, 필 요한 컴포넌트 하나만 사용하더라도 JavaScript는 배럴 파일에 있는 다른 모든 모듈을 처리해야 합니다. 이는 불필요한 런타임 오버헤드를 발생시켜 TTI (Time to Interactive) 지표에 영향을 미칩니다.

순환 의존성

배럴 파일이 제공하는 추상화는 모듈 간의 의도치 않은 순환 의존성을 쉽게 만들 수 있습니다. 이러한 순환 의존성은 개발 워크플로우에 특히 문제가 될 수 있으며, 종종 Hot Module Replacement (HMR)을 방해하고, 의존성 사이클에 있는 모듈이 변경될 때마다 전체 로드를 강제 합니다.

Warning

```
Require cycle: ../../node_modules/react-native-maps/lib/components/MapView.js -> ../../node_modules/react-native-maps/lib/components/Geojson.js -> ../../node_modules/react-native-maps/lib/components/MapView.js
```

```
Require cycles are allowed, but can result in uninitialized values. Consider refactoring to remove the need for a cycle.
```

Metro가 순환 의존성을 감지하면 발생하는 예시 경고

배럴 임포트 사용을 완전히 피하기

위에 언급된 문제들을 해결하기 위한 가장 좋은 해결책은 배럴 임포트 사용을 완전히 피하는 것입니다. 대신 다음과 같이 하십시오:

```
// 배럴 파일에서 임포트하는 것은 권장되지 않습니다.  
import { Button, Card, Modal, Sidebar } from './components';
```

이렇게 수정하세요

```
// 개별 파일에서 임포트하는 것이 훨씬 낫습니다!  
import Button from './components/Button';  
import Card from './components/Card';  
import Modal from './components/Modal';
```

프로젝트에서 이와 같은 규칙을 유지하고 싶다면, eslint 플러그인을 추가하여 임포트 스타일을 유지하는 것이 좋습니다. [eslint-plugin-no-barrel-imports](#)를 사용하면 됩니다.

하지만 현실적으로 이 방법은 가장 아름답지 않습니다. 배럴 임포트를 자동화하는 방법도 있습니다. **React Native**에서는 다음 도구를 사용하여 위에서 언급한 효과를 얻을 수 있습니다:

- **Expo SDK 52**는 트리 쉐이킹을 위한 실험적인 지원을 도입하여 배럴 임포트를 자동으로 최적화합니다. 애플리케이션에서 실제로 사용되는 모듈만 포함하도록 합니다. 자세한 내용은 다음 [링크](#)에서 확인할 수 있습니다.
- **rnx-kit**에서 제공하는 `metro-serializer-esbuild`를 사용하면 **Metro**에서 트리 쉐이킹을 활성화할 수 있습니다. **ESBuild**는 자체적으로 트리 쉐이킹을 지원하므로 번들링을 담당합니다. 자세한 내용은 다음 [링크](#)에서 확인할 수 있습니다.
- **Webpack** 및 **Rspack**은 배럴 파일에서 사용하지 않는 내보내기를 제거할 수 있는 고급 트리 쉐이킹 기능을 제공합니다. 자세한 내용은 다음 [링크](#)에서 확인할 수 있습니다.

실제 사용 라이브러리 예시: `date-fns`

`date-fns`는 날짜 조작에 대한 포괄적인 도구를 제공하는 인기 있는 JavaScript 날짜 유틸리티 라이브러리입니다. 메인 엔트리 포인트를 사용하여 모든 것을 임포트하는 것이 가능하지만, 이 라이브러리는 더 나은 제어를 위해 서브모듈 임포트를 지원하도록 특별히 설계되었습니다. 따라서 효과적으로 라이브러리를 전체적으로 임포트하는 대신:

```
// 이렇게 하면 date-fns 라이브러리를 모두 임포트합니다.  
import { format, addDays, isToday } from 'date-fns';
```

we can split the imports to only include the code we need:

```
// 이러한 임포트들은 여러분이 필요로 하는 코드만을 가져옵니다.  
import format from 'date-fns/format';  
import addDays from 'date-fns/addDays';  
import isToday from 'date-fns/isToday';
```

서브모듈 임포트를 사용하면 실제 사용되는 함수만 최종 번들에 포함되도록 보장할 수 있습니다. [How to Analyze JS Bundle Size](#) 장에서 설명된 도구를 사용하여 최종 JS 번들의 영향을 측정하시기 바랍니다.

BEST PRACTICE

트리 쉐이킹 실험하기

트리 쉐이킹은 최신 JavaScript 번들러에서 사용되지 않는 코드를 최종 번들에서 제거하는 데 사용되는 죽은 코드 제거 기술입니다. "트리 쉐이킹"이라는 용어는 죽은 잎을 제거하기 위해 나무를 흔드는 것에서 유래했으며, 이는 JavaScript 모듈에서 사용되지 않는 코드를 제거하는 것과 유사합니다.

트리 쉐이킹 작동 방식

트리 쉐이킹은 애플리케이션에서 사용되지 않는 코드를 식별하고 제거합니다. 작동 방식은 두 단계로 나뉩니다.:

1 죽은 코드 찾기

- 내보낸 코드가 사용되지 않았지만 내보내지는 코드를 찾습니다.
- 코드를 제거해도 안전한지 확인합니다.
- 어떤 코드가 어떤 코드에 의존하는지 맵핑을 만듭니다.

2 죽은 코드 제거

- 사용되지 않는 코드를 삭제하도록 표시합니다.
- 빌드 최적화 중에 제거합니다.
- 앱에서 실제로 사용하는 코드만 유지합니다. 다음 예제 코드를 살펴보겠습니다:

```
// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// app.js
import { add } from './math';
```

이 코드가 트리 쉐이킹되면 `subtract` 함수는 제거됩니다. 왜냐하면 애플리케이션에서 사용되지 않기 때문입니다. 이는 비교적 간단한 예제이지만 Webpack, Rspack, ESBUILD 등과 같은 번들러는 함수와 모듈 간의 더 복잡한 종속성을 잘 처리할 수 있습니다.

- 💡 라이브러리는 EcmaScript 모듈(ESM)일 때 더 효과적으로 트리 쉐이킹될 수 있습니다. 역사적으로 React Native는 항상 CommonJS 모듈 시스템을 사용했으며 기본적으로 Babel 프리셋을 통해 ESM 라이브러리를 CommonJS 형식으로 변환했습니다. React Native 생태계의 많은 라이브러리는 이 변환을 기대하며 변환 없이 제대로 작동하지 않을 수 있습니다.

운영 환경에서 빌드 과정에서 번들러는 일반적으로 다음과 같은 네 가지 유형의 최적화를 수행합니다:

- 1 사용된 내보내기 확인 - 실제로 사용되는 내보내기를 찾아 사용하지 않는 것을 제거합니다.
- 2 부작용 확인 - 외부 상태를 수정하는 코드를 식별하고 사용하지 않는 코드의 안전한 제거를 보장합니다.
- 3 내보내기 추적 - 모듈 간의 연결을 매핑하고 재내보내기를 추적합니다.
- 4 변수 사용 추적 - 변수가 어떻게 사용되는지 추적하고 필요한 코드의 제거를 방지합니다.

리액트 네이티브 생태계에서의 트리 쉐이킹

트리 쉐이킹에 대한 지원은 다양한 리액트 네이티브 개발 환경에서 다릅니다:

- Metro - 현재 트리 쉐이킹을 지원하지 않지만, `rnx-kit`의 `metro-serializer-esbuild`를 사용하여 가짜로 사용할 수 있습니다.
- Expo - Expo SDK 52부터 실험적 기능으로 트리 쉐이킹을 사용할 수 있습니다.
- Re.Pack - Webpack 및 Rspack 번들러와의 통합을 통해 트리 쉐이킹을 완전히 지원합니다.

- 💡 리액트 네이티브는 본질적으로 멀티 플랫폼 프레임워크입니다. Platform 모듈을 통해 플랫폼별 동작을 정의할 수 있습니다. Expo와 Metro는 이 사실을 이용하여 플랫폼 쉐이킹이라는 죽은 코드 제거 기술을 활성화합니다. 즉, 대상 플랫폼(예: iOS, 안드로이드 또는 웹)에 종속된 코드를 제거하는 것입니다. Platform 모듈을 `react-native` 패키지에서 직접 가져올 경우에만 이 기능이 작동한다는 점에 유의하는 것이 중요합니다.

Expo

Expo SDK 52에서 트리 쉐이킹을 활성화하려면 Metro 구성에서 `experimentalImportSupport`를 활성화하고 앱이 예상대로 빌드되고 실행되는지 확인해야 합니다. 필요한 순환 또는 CommonJS와 ESM 가져오기의 혼합과 관련된 문제가 발생할 수 있으므로 주의해야 합니다.

```
const { getDefaultConfig } = require('expo/metro-config');

const config = getDefaultConfig(__dirname);

config.transformer.getTransformOptions = async () => ({
  transform: {
    experimentalImportSupport: true,
  },
});

module.exports = config;
```

metro.config.js file

실험적 가져오기 지원

실험적 가져오기 지원은 Metro의 `@babel/plugin-transform-modules-commonjs` 플러그인 버전을 사용하여, 의존성 해결 횟수를 크게 줄이고 출력 번들을 단순화합니다. 이 기능은 Expo 프로젝트에서 기본적으로 꺼져 있는 `inlineRequires`와 함께 사용하여 번들을 추가로 실험적으로 최적화할 수 있습니다.

그런 다음 `.env` 파일을 편집하여 `EXPO_UNSTABLE_METRO_OPTIMIZE_GRAPH` 및 `EXPO_UNSTABLE_TREE_SHAKING` 변수를 활성화합니다:

```
EXPO_UNSTABLE_METRO_OPTIMIZE_GRAPH=1
EXPO_UNSTABLE_TREE_SHAKING=1
```

.env file

이 설정은 운영 환경에서만 사용됩니다.

Re.Pack

Re.Pack은 리액트 네이티브 생태계에서 가장 검증된 트리 쉐이킹 구현을 제공합니다. Webpack 번들러가 자바스크립트 생태계에 이 개념을 도입하고 수년간 지원해 왔기 때문입니다. Re.Pack은 기본적으로 트리 쉐이킹이 활성화되어 있으며, Rspack과도 호환되므로 걱정할 필요가 없습니다.

어떤 개선을 기대할 수 있나요?

우리는 Expensify라는 대규모 오픈 소스 리액트 네이티브 애플리케이션을 사용하여 Metro와 Re.Pack 번들러 간의 번들 크기 비교를 위한 종합적인 벤치마크를 만들었습니다. 비교를 완료하기 위해, 우리는 Metro와 Re.Pack이 생성하는 번들의 다양한 변형을 포함했습니다:

- 개발 환경 - 개발자 도구가 포함된 디버그 버전.
- 운영 환경 - 최적화된 릴리스 버전.
- 운영 환경 (압축) - 릴리스 버전, 이후 Terser(자바스크립트 압축 산업 표준)를 사용하여 더 작은 크기로 압축됨.

- 운영 환경 (HBC) - 운영 환경과 동일하지만 Hermes 바이트코드로 변환됨.
- 운영 환경 (압축, HBC) - 운영 환경 (압축)과 동일하지만 Hermes 바이트코드로 변환됨.

번들 크기 비교



벤치마크가 포함된 저장소는 [여기에서 찾을 수 있습니다.](#)

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%



우리는 동일한 코드베이스에서 Expo의 트리 쇼이킹을 실험해 보았지만 결과가 명확하지 않아 벤치마크에 포함하지 못했습니다. 우리는 Metro와 비교했을 때 번들 크기에서 회귀를 발견했으며, 이는 잘못된 구성으로 인한 것이라고 믿습니다. 또한 이 기능이 실험용으로 표시된 사실과 결합하여 벤치마크 결과에 포함하는 것을 막았습니다.

Re.Pack의 프로덕션 번들 크기가 더 큰 것(+8.00%)은 처음에는 놀랍지만, 예상했던 결과입니다. Re.Pack(Rspack을 사용)은 사용하지 않는 코드만 표시하고, 이후 압축 과정에서 제거되기 때문입니다. 번들을 Hermes 바이트코드로 변환하는 과정에는 압축이 포함되어 있기 때문에, Terser를 사용하기 전에도 압축된 번들에서 상당한 개선(-11%에서 -14%)을 볼 수 있습니다.

전반적으로, Re.Pack을 사용하여 모든 최적화를 활성화하면 최종 번들 크기에서 약 10-15% 감소를 기대할 수 있습니다. 우리는 TTI(Time To Interactive) 지표에 대한 영향을 측정하지 않았지만, Hermes 바이트코드 크기의 감소를 보면 앱 부팅 시간이 약간 더 빠르고 메모리 풋프린트가 낮아질 것으로 예상됩니다.

BEST PRACTICE

필요할 때 원격으로 코드를 로드하기

웹 개발에서는 코드를 원격 서버에서 로드하는 것이 일반적입니다. 결국, 사용자가 앱에 접근하는 방식은 클라이언트(예: 브라우저)가 서버에서 HTML, CSS, JS 및 기타 에셋을 로드하는 방식입니다. 그러나 모바일에서는 상황이 조금 다르며 이 패턴을 실행하기가 더 어렵습니다. 사용자는 앱 스토어에서 앱을 다운로드하고, 우리의 JS 코드가 이미 거기에 있으며 오프라인으로 제공됩니다. 그러나 React Native를 사용하여 원격 코드 로딩을 활용하는 것은 가능합니다. 그러나 웹과 비교하여 사용할 때는 다른 사고 방식을 채택해야 합니다.

원격 코드 로딩을 고려할 때

동적 코드 로딩 또는 코드 분할의 효과는 JavaScript 엔진 선택에 따라 크게 달라집니다. Hermes(RN의 기본 엔진)를 사용하는 경우, 메모리 매핑 기술을 통해 번들의 바이트코드에서 필요한 부분만 효율적으로 읽어오기 때문에 온디맨드 코드 로딩의 이점이 미미합니다. 그러나 JavaScriptCore 또는 V8을 사용하는 경우, 청크를 온디맨드 방식으로 로딩하여 초기 로딩 시간을 의미 있게 개선할 수 있습니다.

따라서 현대적인 RN 애플리케이션에서 코드 분할의 이점을 누릴 수 있는 경우는 거의 없지만 여전히 몇 가지 경우가 있습니다. 다음과 같은 경우 코드 분할을 시도해 볼 수 있습니다:

- Hermes JS 엔진을 사용하지 않는 경우.
- 원격 코드 로딩에 대한 특정 요구 사항이 있는 경우.
- 앱 크기가 사용자 경험에 크게 영향을 미치는 경우.
- 앱 크기가 Google Play 스토어(200MB)의 크기 제한을 초과하는 경우.
- 모듈 연방을 사용하여 마이크로 프론트엔드 아키텍처를 구축하는 경우.
- 다른 최적화 기술을 모두 사용한 경우.

Re.Pack으로 이동

기본 Metro 번들러는 프로덕션 빌드에서 코드 분할을 지원하지 않습니다. 이 기능을 구현하려면 Webpack을 기반으로 구축된 React Native 번들링을 지원하는 도구인 Re.Pack으로 전환해야 합니다. 이러한 전환에는 기존 구성을 마이그레이션하는 추가 작업이 필요하지만, 다른 최적화 옵션을 모두 소진한 경우 시도할 가치가 있을 수 있습니다.

Re.Pack으로 시작하려면 초기화 도구를 사용하는 것이 좋습니다:



이 명령어는 프로젝트 파일에 필요한 변경 사항을 적용하고 프로세스를 안내합니다. 완료되면 다음에 'npm start'를 실행할 때 JS는 Metro 대신 Webpack 또는 Rspack을 번들러로 사용하여 Re.Pack 개발 서버에 의해 제공됩니다. Re.Pack이 Metro의 거의 드롭인 교체품이 되려고 노력하지만 일부 호환되지 않는 문제가 발생할 수 있으므로 [공식 마이그레이션 가이드](#)를 따르십시오.



Re.Pack은 Callstack에서 개발한 도구입니다. 최근에 Rspack에 대한 지원을 추가했는데, 이는 빌드 시간을 크게 개선하여 대부분의 경우 Metro보다 더 빠르게 만들어 줍니다.

번들 분할

코드를 분할하는 가장 일반적인 방법은 동적 가져오기(dynamic imports)와 React의 'lazy' 함수 및 'Suspense' 컴포넌트를 사용하는 것입니다. 분할 지점을 도입하는 것은 일반적으로 다음과 같습니다:

```
// Before - 일반적인 임포트
import FeatureComponent from './FeatureComponent';

// After - 분할 포인트 만들기
const FeatureComponent = React.lazy(() =>
  import(/* webpackChunkName: "feature" */ './FeatureComponent')
);
```

동적 가져오기를 통한 코드 분할 도입

이러한 변경으로 인해 FeatureComponent가 앱을 번들링할 때 feature.chunk.bundle라는 별도의 파일에 위치하게 됩니다.

앱의 화면에 대해 코드 분할을 구현하고자 할 때 실제로 적용하는 방법은 다음과 같습니다:

```
-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>
  );
};

동기적으로 가져온 컴포넌트를 동적으로 가져온 컴포넌트로 변환
```

원격 위치에서 코드 지연 적재

webpackChunkName로 표시된 주석을 사용하면 청크에 이름을 지정할 수 있습니다. 이는 마지막 단계에서 유용합니다. 즉, 더 이상 메인 번들의 일부가 아니며 원격 서버에서 다운로드할 수 있는 청크를 Re.Pack에 로드하는 방법을 알려주는 것입니다.

```
// index.js
import { ScriptManager, Script } from '@callstack/repack/client';

// Re.Pack에 청크를 로드할 위치를 알리기
ScriptManager.shared.addResolver((scriptId) =>
({ url: __DEV__
  // 개발 환경에서는 dev 서버에서 로드
  ? Script.getDevServerURL(scriptId)
  // 운영 환경에서는 CDN으로 로드
  : `https://my-cdn.com/assets/${scriptId}`,
}));


// 그런 다음 이전과 마찬가지로 앱을 실행합니다
AppRegistry.registerComponent(appName, () => App);
```

'ScriptManager'를 사용하여 Re.Pack에 앱의 비동기 청크를 얻는 방법을 알려주기

ScriptManager는 Re.Pack의 런타임 클라이언트이며, 메인 번들 이외의 청크를 찾고 로드하는 동작을 구성할 수 있습니다. 이러한 단계는 코드 분할의 가장 기본적인 경우에 요구되는 모든 것입니다.

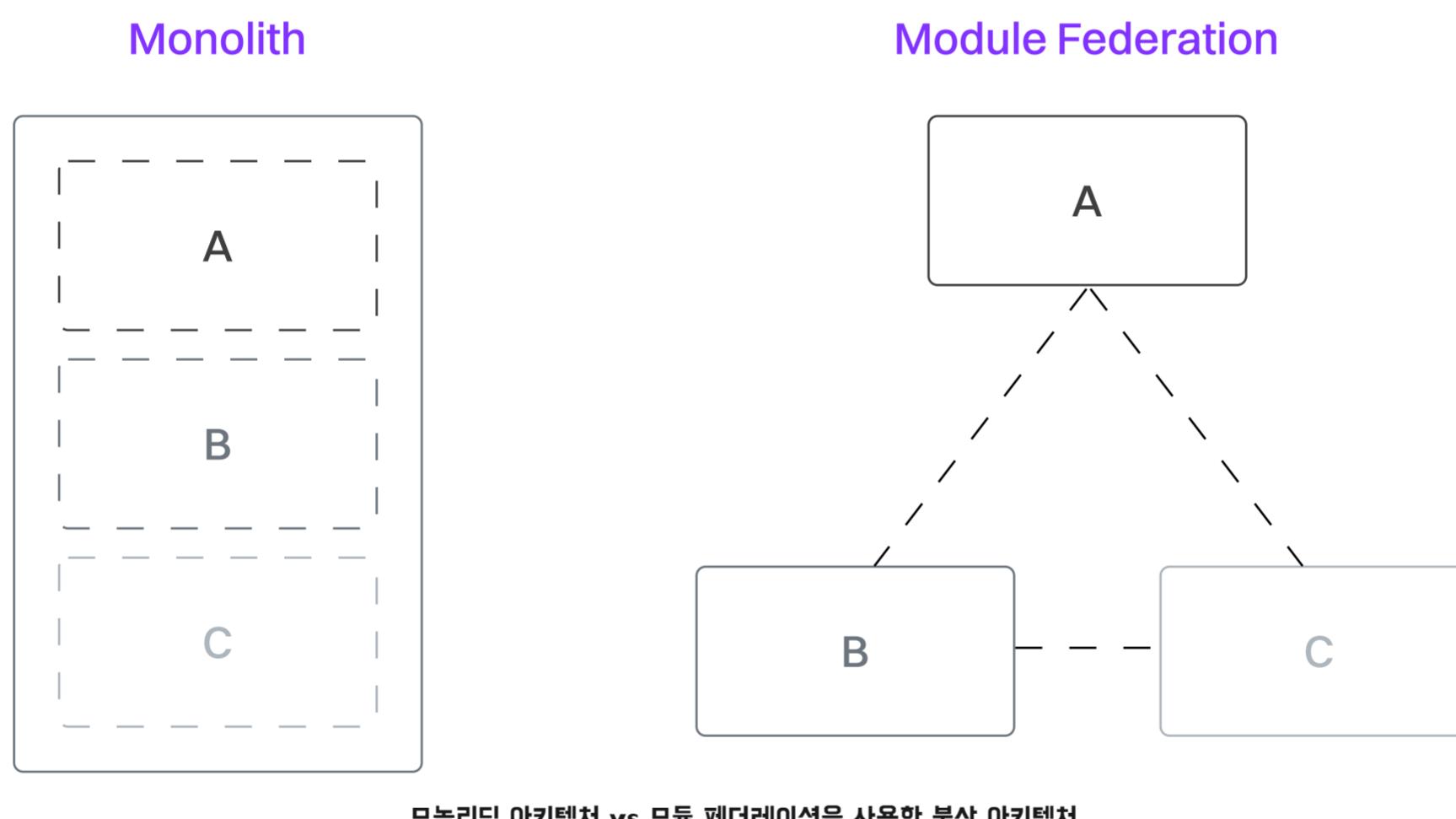


코드 분할을 사용할 때는 캐싱 또는 프로덕션 CDN에 배포하는 것과 같은 더 많은 요소가 고려되어야 합니다. 원하는 효과를 달성하는 방법에 대한 자세한 정보는 공식 [Re.Pack](#) 문서를 참조하십시오.

모듈 페더레이션

코드 분할의 장점 중 하나는 특정 사용자가 필요로 하는 기능만 로드하고 필요하지 않은 사용자에게는 로드하지 않는다는 것입니다. 예를 들어, 대규모 엔터프라이즈 애플리케이션에서 서로 다른 사용자 역할에 따라 완전히 다른 기능 세트에 대한 접근이 필요할 수 있습니다. 이 경우 모든 것을 한꺼번에 번들링하는 대신 사용자 권한이나 선호도에 따라 기능을 주문형으로 로드할 수 있습니다. 이 접근 방식은 초기 로드를 최적화하고 보다 맞춤형 경험을 제공합니다.

보다 복잡한 시나리오에서는 애플리케이션의 서로 다른 부분에 대해 여러 개의 격리된 팀을 관리하거나 팀에 개발 및 배포에 더 많은 자율성을 부여하려는 경우, Re.Pack의 모듈 페더레이션을 사용한 마이크로 프론트엔드 아키텍처를 고려할 수 있습니다. 모듈 페더레이션은 웹 및 모바일에서 마이크로 프론트엔드에 대한 인기 있고 검증된 솔루션입니다.



그러나 마이크로프론트엔드는 번들 크기 최적화만을 위해 선택되어서는 안 됩니다. 이 아키텍처는 자체적인 복잡성을 가지고 있으며, 조직 및 개발 워크플로우의 이점이 추가적인 오버헤드를 상쇄할 때만 고려되어야 합니다.



마이크로프론트엔드의 복잡성 중 하나는 버전 관리입니다. 직접 처리하기는 어렵고, 경험에 따르면 상당히 번거로울 수 있습니다. 만약 제3자 서비스에 개방적이라면, [Zephyr Cloud](#)를 추천합니다. 이는 이 문제를 간소화하고, 초 단위 배포를 허용하며, 공식적으로 Re.Pack과 통합됩니다.

BEST PRACTICE

R8을 사용한 안드로이드 코드 축소

앱 크기 분석 방법 장에서 언급했듯이 앱 크기는 중요한 지표 중 하나이므로 주의 깊게 모니터링 해야 합니다. 이전 장에서는 JavaScript 번들 크기를 최적화하는 방법에 대해 논의했습니다. 그러나 일부 경우에는 네이티브 코드가 앱 크기에 영향을 줄 수도 있습니다. 이 절에서 설명하는 도구를 사용하면 앱 크기를 추가로 메가바이트 단위로 줄이고, 난독화 기술을 사용하여 더 안전하게 만들 수 있습니다.

R8 활성화

ProGuard에 대해 들어보았을 것입니다. ProGuard는 Android Gradle의 도구로, APK 크기를 줄이기 위해 앱 코드의 크기를 줄이고, 최적화하고, 난독화합니다. 그러나 프로젝트에서 React Native v0.60 이상을 사용하는 경우, 대부분의 Android 프로젝트의 핵심 종속성인 Android Gradle Plugin은 더 이상 ProGuard로 코드를 최적화하지 않습니다. 대신 R8을 사용합니다.

R8은 ProGuard를 대체하여 APK를 축소, 최적화 및 난독화하는 안드로이드 도구입니다. 개선된 성능과 현대적인 기능, 예를 들어 코틀린과 같은 기능과의 호환성을 제공합니다. 또한 ProGuard 도구와 호환되며 구성 파일과 같이 여전히 일부 Android 파일에서 "proguard" 참조를 볼 수 있습니다.

React Native 프로젝트에서 R8을 활성화하려면 app/build.gradle 파일을 편집하십시오.

```
def enableProguardInReleaseBuilds = true  
android/app/build.gradle 파일을 편집하여 Proguard를 활성화합니다.
```

기본 React Native 템플릿에서 이 변수는 릴리즈 빌드 변형에서 minifyEnabled 설정을 true로 설정합니다. 이 설정은 Gradle에게 앱을 배포하기 위해 번들링할 때 R8 최적화를 적용하도록 지시합니다.

앱 크기 분석 방법 장에서 우리는 기본 ProGuard 규칙으로 R8을 활성화했을 때 앱 크기가 얼마나 줄어드는지 테스트했습니다. 해당 장에서 언급한 샘플 앱은 R8을 사용하지 않았을 때 9.5MB의 크기를 가졌습니다.

R8을 활성화한 후에는 6.3MB로 줄어들어 33%의 차이가 발생했습니다. 더 큰 프로젝트는 개선 폭이 조금 더 작을 것으로 예상되지만, 이 정도면 충분히 가치 있는 작업입니다.

R8을 위한 ProGuard 규칙 및 구성

이미 언급했듯이 R8은 "ProGuard 규칙"으로 구성됩니다. 전용 파일에서 빌드 최적화 프로세스 중에 유지하거나 삭제할 클래스, 메서드 및 필드를 제어할 수 있습니다. 또한 규칙을 통해 코드 난독화 패턴(예: 클래스/메서드 이름 바꾸기)을 구성하여 역공학을 더 어렵게 만들 수 있습니다. React Native 앱에서는 이러한 규칙이 app/android/proguard-rules.pro에 정의됩니다. 이 파일은 기본적으로 비어 있습니다.

대부분의 경우 수동으로 규칙을 정의할 필요는 없습니다. 그러나 코드 제거 후 앱이 올바르게 작동하지 않는 문제가 발생하면 제외할 클래스를 추가해야 할 수 있습니다. 이는 특정 종속성이 클래스 이름에 대해 가정하기 때문에 필요할 수 있습니다. 예를 들어 Firebase를 사용할 때 문제가 발생하면 R8에게 해당 클래스를 유지하도록 지시할 수 있습니다(제거 방지).

```
# Firebase
-keep class io.invertase.firebaseio.** { *; }
-dontwarn io.invertase.firebaseio.**
```

proguard-rules.pro file

난독화

minifyEnabled를 지정하면 난독화가 기본적으로 활성화됩니다.

그러나 어떤 이유로 난독화를 비활성화하고 싶다면 proguard-rules.pro 파일에 -dontobfuscate 설정을 사용할 수 있습니다.

```
-dontobfuscate
```

proguard-rules.pro file

ProGuard 규칙 정의에 대한 자세한 정보는 [공식 문서를 참조하십시오.](#)

리소스 축소

릴리즈 빌드 변형에서 R8을 활성화하면 Gradle의 리소스 최소화 덕분에 추가 최적화를 잠금 해제할 수 있습니다. 이를 통해 앱 번들을 더욱 줄일 수 있습니다. 활성화되면 다음과 같은 여러 최적화가 수행됩니다.

1 중복 리소스를 병합합니다.

2 PNG 파일의 색상 깊이를 줄이고 무손실 압축을 적용하여 최적화합니다.

3 벡터 드로어블을 처리합니다.

리소스 축소는 app/build.gradle 파일의 shrinkResources 설정을 사용하여 활성화할 수 있습니다.

```
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

이제 죽은 코드를 제거하여 앱의 크기를 줄이는 방법을 알게 되었습니다. 앱의 다운로드된 크기와 설치된 크기 간의 차이를 확인하려면 Ruler 도구를 사용할 수 있습니다. 이 기술을 활성화한 상태에서 앱을 배포하기 전에 충분한 테스트를 수행하여 문제가 없는지 확인하십시오.

BEST PRACTICE

Native Assets 폴더 사용

앱 크기의 상당 부분은 이미지, 사운드 또는 비디오와 같은 에셋으로 구성됩니다. iOS와 Android는 모두 장치의 화면 크기 또는 픽셀 밀도에 따라 이러한 에셋의 전달을 최적화하는 방법을 제공합니다. 예를 들어, 모바일 폰에서는 더 큰 이미지를 로드하여 선명하고 깨끗하게 표시하는 것이 좋습니다. 모바일 장치는 일반적으로 작은 화면에 큰 해상도를 가지고 있어 높은 픽셀 밀도를 나타냅니다. 화면에 더 많은 픽셀이 있으므로 더 큰 이미지가 선호됩니다. 올바르게 사용되면 이러한 플랫폼 최적화 메커니즘은 사용자가 앱 스토어에서 앱을 다운로드할 때 필요한 이미지 해상도만 로드하는 데 도움이 될 수 있습니다.



에셋의 크기를 줄이기 위해 전용 도구를 사용하여 에셋을 최적화하는 것을 잊지 마십시오.

size suffixes 사용

React Native 애플리케이션에서 에셋을 처리할 때 일반적으로 동일한 이미지의 여러 크기 변형을 만드는 것이 좋습니다. 즉, 기본 크기(1x), 두 배 크기(2x), 세 배 크기(3x)입니다. 일반적으로 파일 이름에 크기 접미사(@2x 또는 @3x)를 사용하는 규칙을 따릅니다.

```
assets/
  └── image.jpg      // 1x resolution
  └── image@2x.jpg   // 2x resolution
  └── image@3x.jpg   // 3x resolution
```

logo.png의 크기 밀도 확장 디렉토리 구조

표준 require로 이러한 이미지를 가져올 때 React Native는 현재 장치의 화면 밀도에 따라 가장 적합한 이미지를 선택합니다.

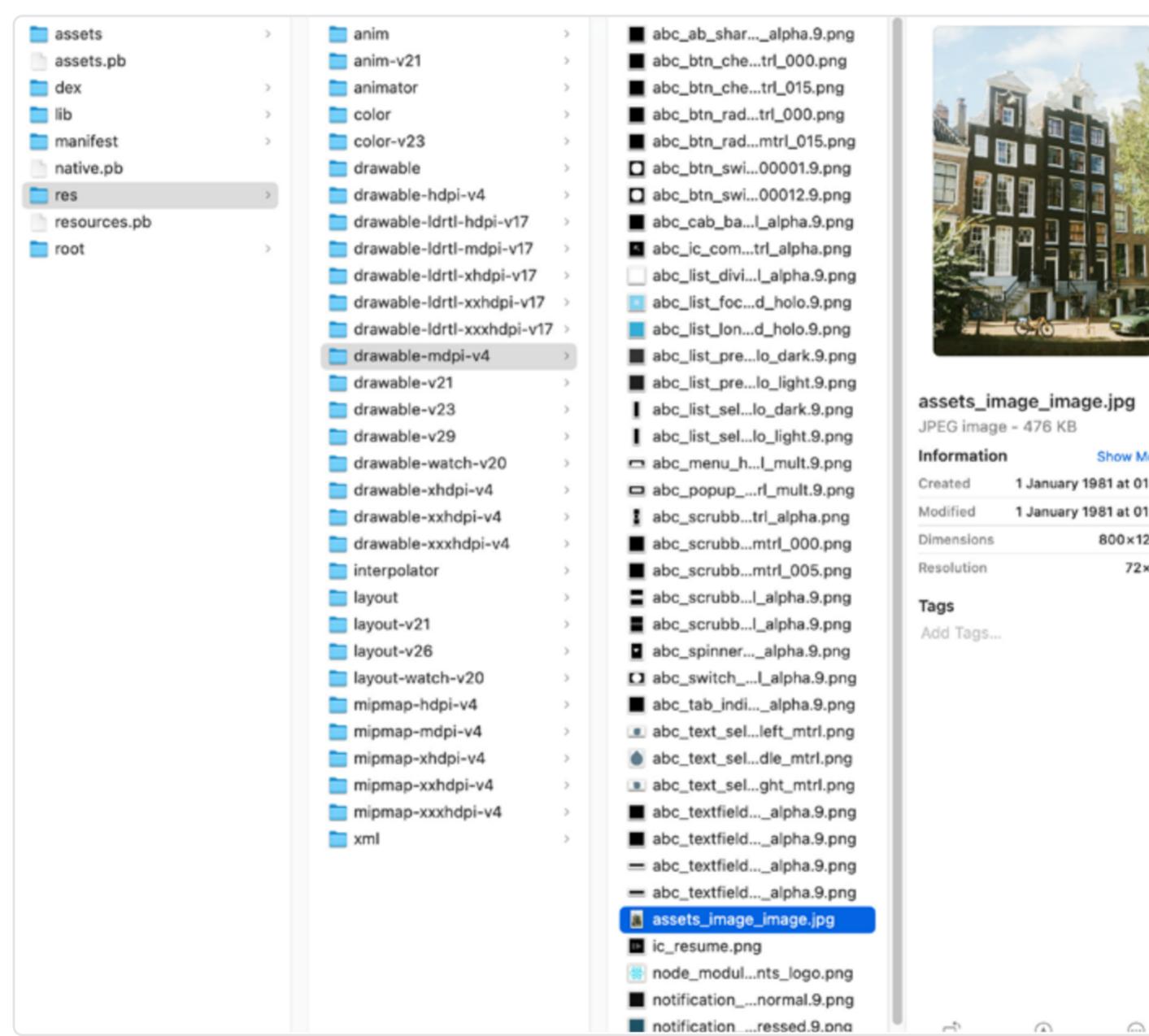
기본 에셋 폴더

이미지에 크기 접미사가 있는 한 Android는 이 최적화를 자동으로 적용합니다.
 이를 통해 내부 작동을 살펴보고 무엇이 일어나고 있는지 더 잘 이해할 수 있는 기회를 갖도록 하겠습니다.
 이를 위해 다음 명령어를 실행하여 AAB 파일을 생성하겠습니다.



이 명령어를 실행하면 Play 스토어가 사용자의 장치 아키텍처에 최적화된 애플리케이션 바이너리를 주문형으로 생성하는 데 사용할 수 있는 Android App Bundle이 생성됩니다. 이에 대해 자세히 알아보거나 다른 형식에 대해 알아보려면 소개로 돌아가십시오.

완료되면 android/app/build/outputs/bundle/release/app-release 폴더로 이동하십시오.
 번들의 내용을 검사하려면 먼저 확장명을 .zip으로 변경하여 압축을 풀겠습니다(결국 AAB는 ZIP 아카이브일 뿐입니다).



Android App Bundle의 내용을 압축을 풀어서 내부를 살펴보겠습니다.

리소스 카탈로그로 이동하면 drawable-xhdpi-v4, drawable-xxhdpi-v4와 같은 Android 화면 밀도에 해당하는 이름의 여러 폴더가 표시됩니다. 폴더 내부의 이미지를 보면 @2x와 같은 접미사가 사라진 것을 확인할 수 있습니다. 대신 각 폴더에는 사용 가능한 최고의 밀도에 따라 이미지가 위치하고 있습니다.

사용자가 Google Play에서 애플리케이션을 다운로드하면 사용자의 장치에 맞는 주문형 애플리케이션에는 해당 밀도에 맞는 에셋만 포함되며 나머지는 포함되지 않습니다.



맞춤형 컴파일 과정에 대한 논의는 이 [가이드의 범위를 벗어납니다](#). 그러나 이 주제에 대해 더 자세히 알고 싶다면 다음 리소스를 참조할 수 있습니다.

Xcode Asset Catalog

iOS에서도 유사한 결과를 얻을 수 있습니다. 그러나 Android와 달리 React Native와 Metro는 번들을 생성하는 동안 이를 자동으로 수행하지 않습니다. 유사한 최적화를 이용하기 위해서는 Xcode Asset Catalog (.xcassets)를 명시적으로 생성하고 컴파일 파이프라인의 일부로 구성해야 합니다.

시작하기 위해 RNAssets.xcassets 파일을 생성하고 ios 폴더 안에 배치해 보겠습니다.



이 폴더는 React Native 번들 스크립트 내에서 하드 코딩된 경로 때문에 RNAssets.xcassets로 명명되어야 합니다. 다행히도, 이 제약은 미래에 해제될 것으로 기대됩니다.

폴더를 생성한 후, 프로젝트 설정에서 Build Phases로 이동합니다. 그런 다음, EXTRA_PACKAGER_ARGS 변수를 사용하여 8번 라인 위에서 Bundle React Native 코드 및 이미지 빌드 단계를 수정합니다.

```
export EXTRA_PACKAGER_ARGS="--asset-catalog-dest ./"
```

The screenshot shows the Xcode project settings with the 'Build Phases' tab selected. A new script phase has been added under the 'Bundle React Native code and images' section. The script content is:

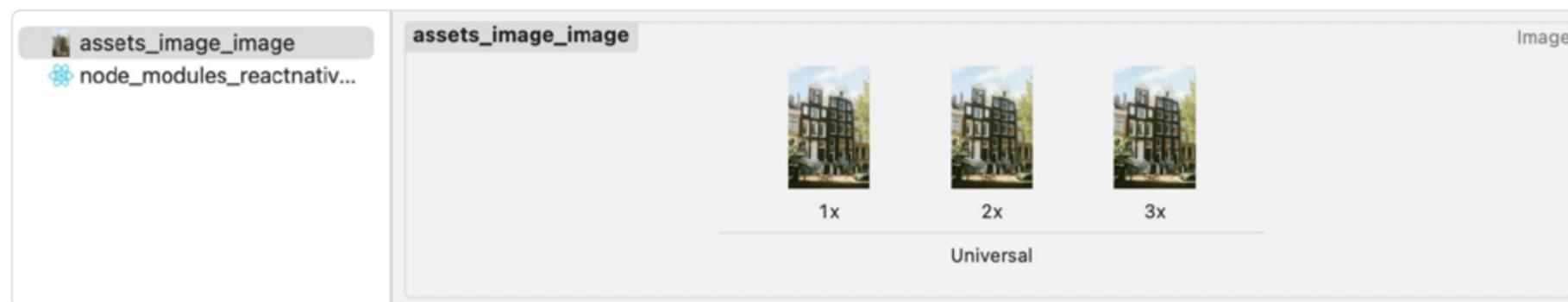
```
2
3 WITH_ENVIRONMENT="$REACT_NATIVE_PATH/scripts/xcode/w
ith-environment.sh"
4 REACT_NATIVE_XCODE="$REACT_NATIVE_PATH/scripts/react
-native-xcode.sh"
5
6 export EXTRA_PACKAGER_ARGS="--asset-catalog-dest ./"
7
8 /bin/sh -c "$WITH_ENVIRONMENT $REACT_NATIVE_XCODE"
```

Below the script, the 'Run script' options are set to 'Based on dependency analysis'. The log message indicates that the script will be skipped if inputs, context, and outputs haven't changed. There are also checkboxes for 'Show environment variables in build log' and 'Use discovered dependency file'.

코드 및 이미지 번들링은 빌드 프로세스의 한 단계로 수행됩니다.

그 결과, 빌드 단계가 실행되면 Metro는 자산들을 **RNAAssets.xcassets** 카탈로그에 넣습니다.

첫 번째 빌드를 실행한 후, 자산 폴더는 다음과 같아야 합니다:



Xcode Asset Catalog를 살펴보겠습니다. 여기에는 샘플 이미지들이 포함되어 있습니다.

문제가 발생하거나 모든 것이 제대로 작동하는지 테스트하고 싶을 때, 수동으로 번들 명령어를 실행할 수 있습니다. (`--asset-catalog-dest`는 이제 ios를 가리킵니다. 프로젝트 루트에서 스クリプ트를 실행하기 때문에)

```
> 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>
```

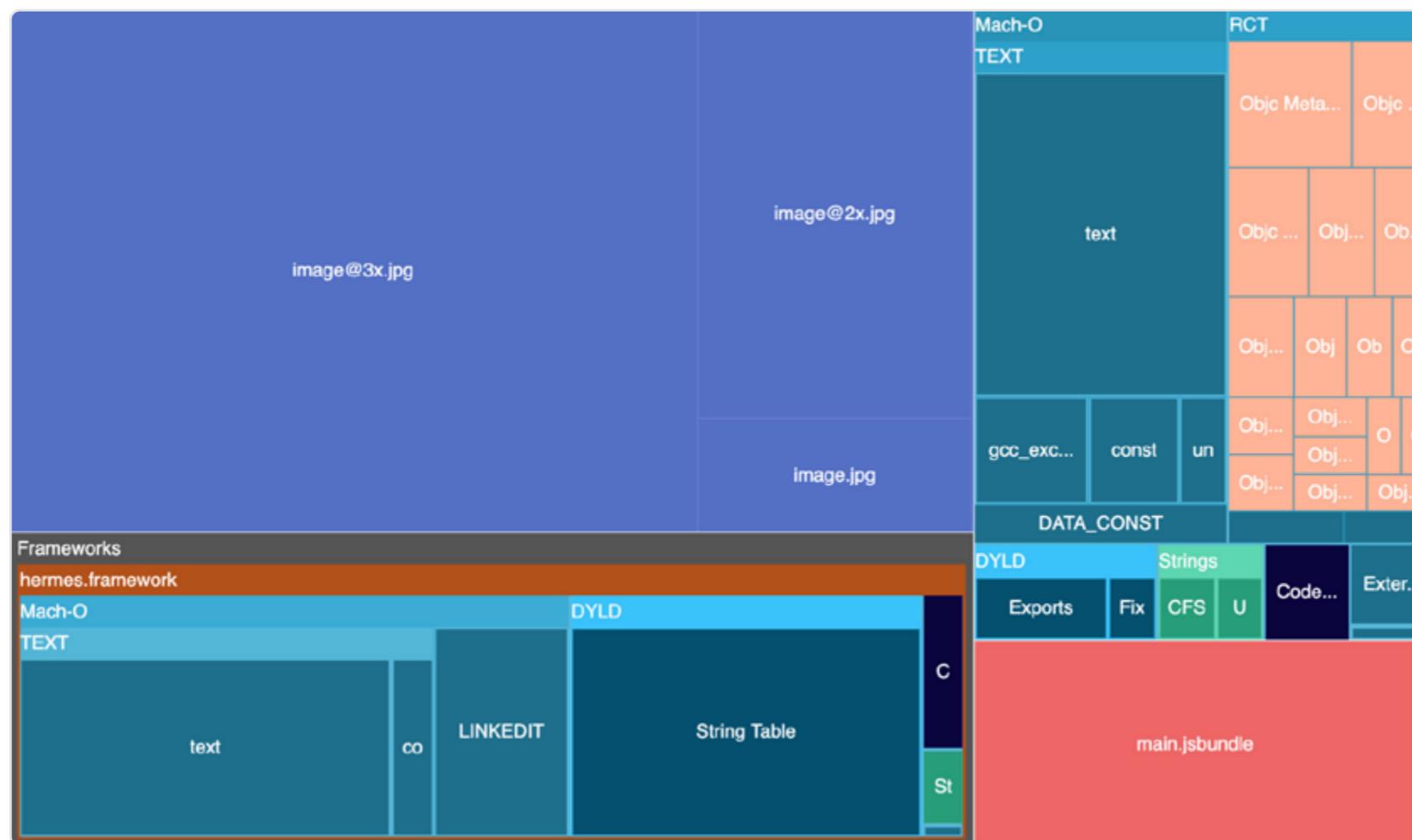
이제 앱을 앱 스토어에 업로드하면 앱 용량 줄이기 메커니즘이 작동하여 사용자의 장치에 따라 적절한 이미지 크기를 선택합니다!



iOS에서는 "앱 용량 줄이기(App Thinning)"을 통해 장치 특정 버전을 생성하여 필요한 자산만 포함함으로써 앱 전달을 최적화합니다. Android는 "Android 앱 번들(AAB)"을 통해 비슷한 결과를 달성하며, 각 장치 구성에 대해 최적화된 APK를 생성하고 앱을 수요에 따라 다운로드할 수 있는 모듈식 구성 요소로 분할합니다.

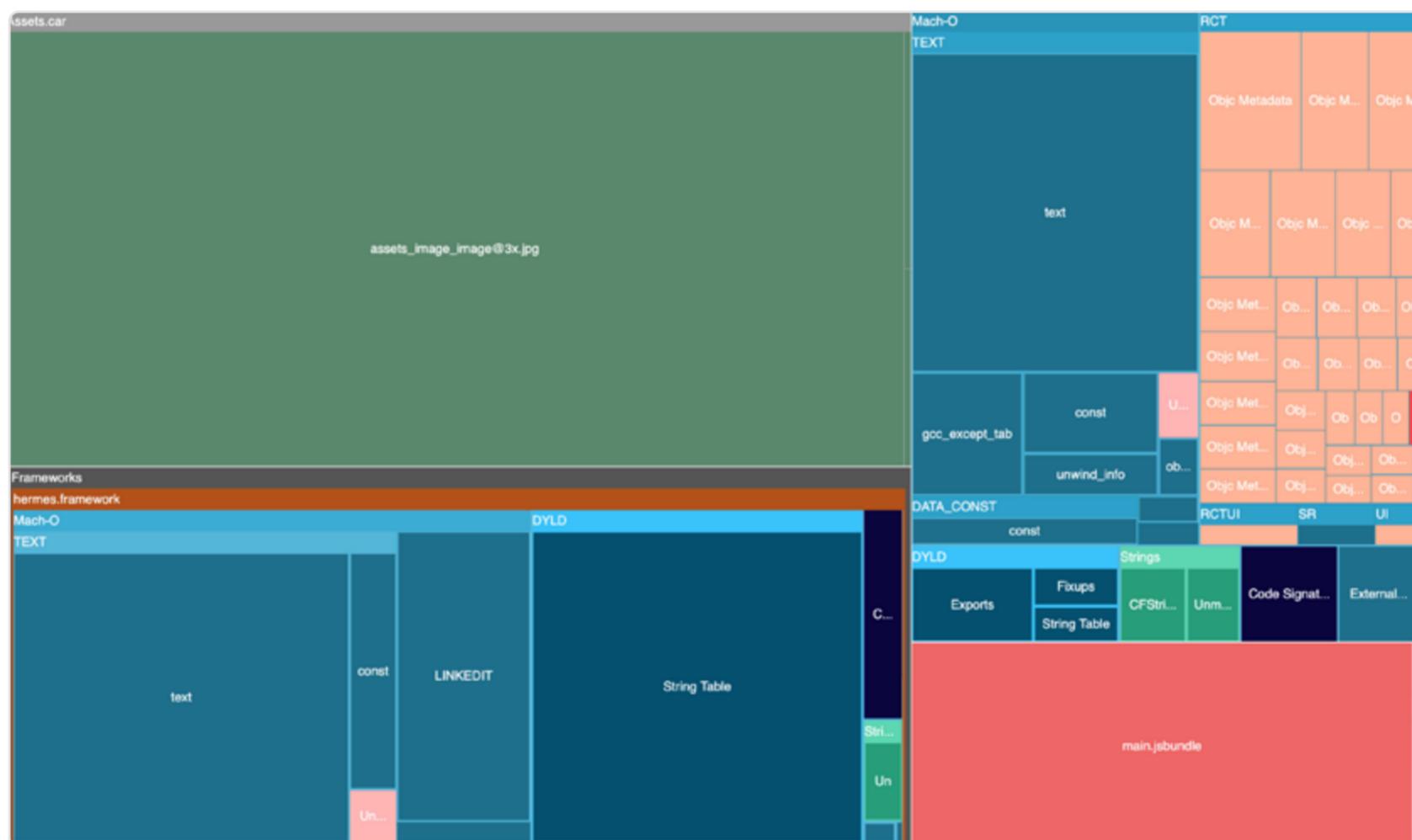
테스트해보기

iOS에서 이 최적화를 적용하기 전과 후의 실제 차이점을 살펴보겠습니다. 최적화 없이, 번들에는 세 가지 이미지가 모두 포함되어 있는데, 이는 기기가 하나의 크기만 사용할 것이기 때문에 낭비입니다.



iOS에서 자산 카탈로그 최적화 전의 Emerge Tools 스크린샷

위에서 언급한 기술을 적용하고 자산 카탈로그를 사용하여 번들을 확인하면, 관련 이미지 (`asset_image_image@3x.jpg`)만 존재하는 것을 볼 수 있습니다.



앱 용량 줄이기 활성화 후의 Emerge Tools 스크린샷

이 최적화 기술은 앱이 특정 플랫폼에 필요한 이미지만을 포함하도록 보장하여 앱 크기 지표에 긍정적인 영향을 미치는 훌륭한 방법입니다.



이 장을 작성하는 시점(2025년 1월)에는 자산 카탈로그가 기본적으로 활성화되지 않았지만, 기본 선택 사항으로 만들기 위한 지속적인 노력이 진행 중입니다.

BEST PRACTICE

JS 번들 압축 비활성화

자바스크립트 번들 압축 해제

우리는 이미 APK 및 Android 앱 번들이 압축된 아카이브라는 것을 배웠습니다. Android 빌드 시스템은 APK 또는 AAB의 전반적인 크기를 줄이기 위해 패키징 중에 파일에 압축을 적용하며, 이는 다운로드 크기와 장치의 스토리지 사용량을 줄이는 데 도움이 될 수 있습니다. 압축은 일반적으로 이미지 파일(.png, .jpg)과 같이 이미 효율적으로 압축된 형식은 건너뛰는 것을 제외하고 파일 형식 간의 구분 없이 적용됩니다.

Hermes를 사용하는 Android React Native 앱에서는 최종 JS 번들이 index.android.bundle 파일에 패키징됩니다. 바이너리이지만 빌드 시스템이 명시적으로 압축하지 말라고 지시하지 않는 한 대부분의 리소스 파일에 압축을 적용하기 때문에 앱에 패키징된 다른 리소스처럼 처리될 수 있습니다.

흥미로운 점은 이 압축 파일이 메모리 매핑(mmap) 절차 중에 건너뛰어진다는 것입니다. 이는 [React Native 코어에 대한 이 Pull Request에서 조사되었듯이](#), 가장 중요한 Hermes 최적화가 제대로 작동하지 않아 TTI(Time To Interactive) 메트릭에 부정적인 영향을 미칩니다.

 이 글을 작성하는 시점(2025년 2월 17일) 기준으로 React Native Core 팀은 이 동작을 기본적으로 포함하고 있습니다. 이 기능은 React Native 0.79에서 제공될 가능성이 높습니다.

이 동작을 선택 해제하고 React Native 버전과 관계없이 Hermes의 잠재력을 최대한 활용하려면 앱의 build.gradle 파일에서 androidResources를 설정하여 번들 확장자를 가진 파일에 대한 압축을 비활성화하도록 Android 빌드 시스템에 수동으로 지시할 수 있습니다.

```
android {  
    +     androidResources {
```

```
+     noCompress += [ "bundle" ]  
+ }  
}  
.bundle 파일에 대한 압축이 비활성화된 app/build.gradle 파일
```

결과적으로 `index.android.bundle` 파일은 압축되지 않아 전체 앱 번들 크기가 효과적으로 증가합니다. 이 크기 증가는 앱의 설치 크기에 영향을 주지만 다운로드 크기에는 영향을 주지 않습니다. 가장 중요한 점은 Hermes가 메모리 매핑을 활용하고 JS 번들을 더 빠르게 로드할 수 있으므로 앱이 훨씬 더 빠르게 로드되어야 한다는 것입니다. 초기 테스트에서 75.9MB 설치 크기의 APK의 경우 6.1MB 크기 증가(+8%)였으며 TTI는 450ms(-16%) 감소했습니다.

다양한 이유로 인해 앱의 크기가 증가하지 않을 가능성성이 있지만 가능성은 낮습니다. 결과를 측정하고 사용자의 행동을 관찰하여 이 변경 사항이 순이익이 아닌 손실을 초래하지 않도록 해야 합니다. 그러나 대부분의 경우 이점은 비용보다 클 것입니다.

저자 소개

**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.

[𝕏 @grabbou](#) [ଓ @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.

[𝕏 @o_kwasniewski](#) [ଓ @okwasniewski](#)

Robert Pasiński, Senior Software Engineer

Passionate about all things development, low-level coding enthusiast.

[𝕏 @rob_pasinski](#) [ଓ @robik](#)

Michał Pierzchała, Principal Engineer

Passionate about building mobile and web tooling in the Open Source. Core React Native Community contributor.

[𝕏 @thymikee](#) [ଓ @thymikee](#)

Jakub Romańczyk, Senior Software Engineer

JS Infra nerd. Passionate about React Native, OSS & JS tooling. Souls-like games enthusiast.

[𝕏 @_jbroma](#) [ଓ @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](#)

이 가이드에서 언급된 라이브러리 및 도구

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

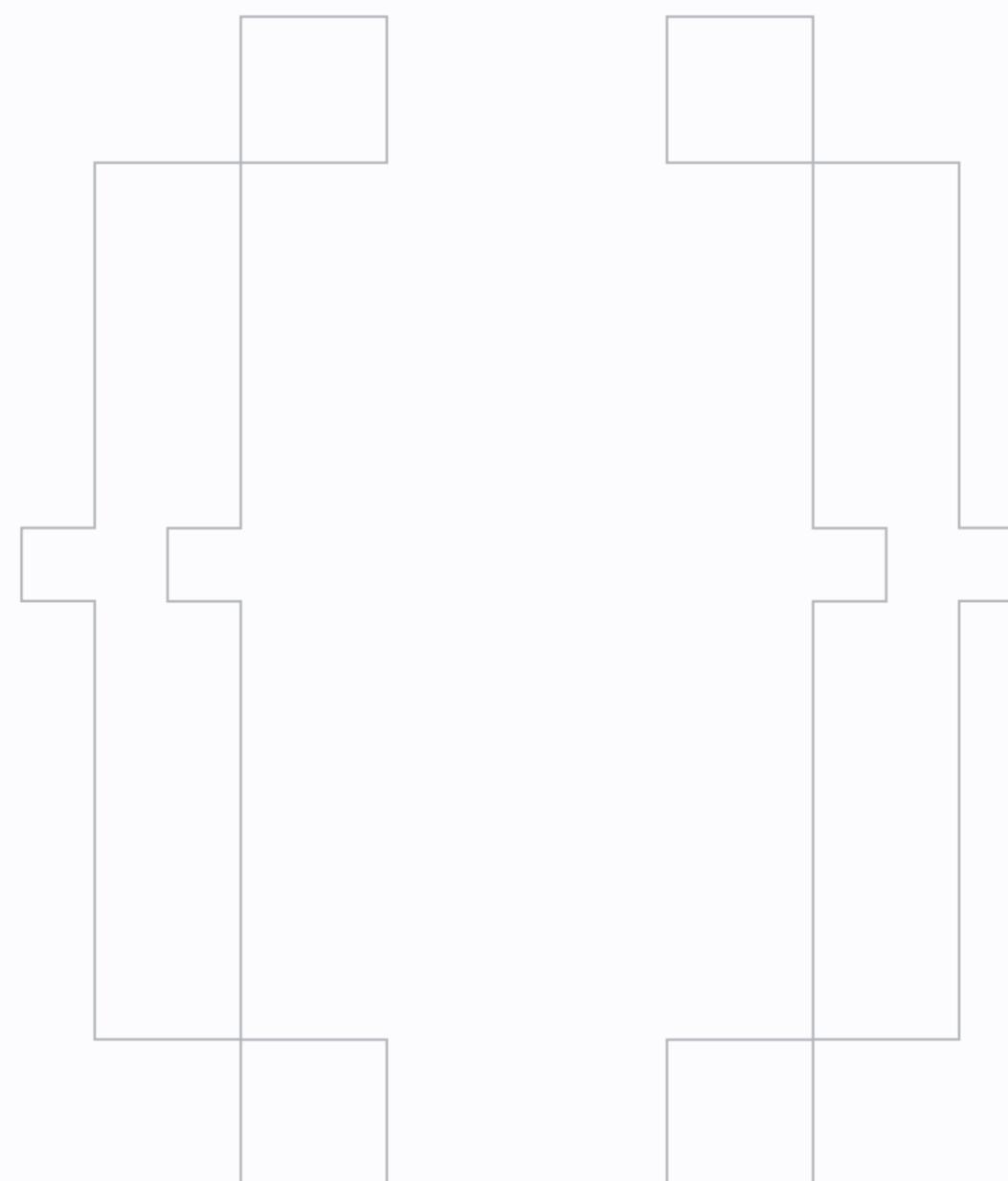
- [React Native Builder Bob](#)—a tool for scaffolding new React Native libraries with support for the new architecture.
- [Ninja](#)—a fast build system commonly used for compiling native code in React Native projects.

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.