

The Ultimate Guide to React Native Optimization



Cải thiện trải nghiệm người dùng, hiệu năng
và sự ổn định cho app của bạn

Created by {callstack}

Translated by 200Lab Education

Mục lục

Tập 1

1. Hãy lưu ý UI re-renders
2. Hãy sử dụng các components chuyên dụng cho các layout nhất định
3. Hãy cân nhắc khi sử dụng các thư viện ngoài
4. Hãy luôn nhớ rằng luôn có các thư viện riêng cho mobile app
5. Tìm kiếm sự cân bằng giữa native và JavaScript
6. Diễn hoạt ở 60FPS bất kể vì điều gì

Mục lục

Tập 2

1. Luôn sử dụng phiên bản React Native mới nhất để có các tính năng mới
2. Cách gỡ lỗi nhanh hơn và tốt hơn với Flipper
3. Tự động hóa việc quản lý dependency của bạn với “autolinking”
4. Tối ưu hóa thời gian khởi động ứng dụng Android của bạn với Hermes
5. Tối ưu hóa kích thước ứng dụng Android của bạn với các cài đặt Gradle

Mục lục

Tập 3

1. Chạy thử nghiệm cho các phần chính của ứng dụng
2. Hoạt động Continuous Integration (CI) đúng chỗ
3. Đừng ngại ship nhanh với Continuous Deployment
4. Gửi OTA (Over-The-Air) trong trường hợp khẩn cấp

Giới thiệu

React Native optimization

React Native đảm nhận việc render UI.
Nhưng hiệu năng là còn tuỳ vào trường hợp.

Trong React Native, bạn tạo ra những components mà chúng sẽ cho biết thông tin giao diện sẽ trông như thế nào. Trong lúc thực thi, React Native chuyển hoá chúng thành các component cụ thể với từng platform hơn là giao tiếp trực tiếp xuống tầng native. Việc của bạn là tập trung vào xây dựng trải nghiệm người dùng trên ứng dụng mà thôi.

Tuy nhiên, đó không có nghĩa là tất cả ứng dụng sử dụng React Native sẽ hoạt động nhanh như nhau và cung cấp cùng một mức độ trải nghiệm người dùng.

Mọi phương pháp tiếp cận "hướng khai báo" (declarative) (bao gồm React Native) được xây dựng bởi "các điều lệnh" (imperative) APIs. Thế nên, bạn cần phải cẩn thận khi làm việc với những thứ liên quan tới imperative.

Khi bạn xây dựng ứng dụng theo hướng imperative, bạn nên phân tích cẩn thận các lệnh gọi ra bên ngoài (external APIs). Ví dụ, trong môi trường multi-threaded, bạn sẽ viết code để có thể "thread safe", và phải luôn lưu ý tới ngữ cảnh (context) và tài nguyên (resources) và đoạn code bạn đang sử dụng đến.



Mặc dù có nhiều sự khác biệt ở cả hai hướng tiếp cận imperative và declarative, chúng vẫn có nhiều điểm chung. Declarative có thể phân tách thành nhiều câu lệnh imperative. Ví dụ, React Native sử dụng chung các APIs để xây dựng giao diện trên iOS cũng như các native developer đang sử dụng hàng ngày.

React Native có hiệu năng đồng nhất nhưng không làm nó vượt trội!

Mặc dù bạn không phải lo lắng về hiệu suất của các lệnh gọi API iOS và Android cơ bản, nhưng cách bạn kết hợp các thành phần lại với nhau có thể tạo ra tất cả sự khác biệt. Tất cả các thành phần của bạn sẽ cung cấp cùng một mức hiệu suất và khả năng đáp ứng.

Nhưng đồng nhất có đồng nghĩa là tốt nhất không? Không.

Đây là nơi mà cuốn sách này của chúng tôi phát huy tác dụng. Sử dụng React Native đúng với tiềm năng của nó.

Như đã thảo luận trước đây, React Native là một declarative framework và đảm nhận việc render UI cho ứng dụng của bạn. Nói cách khác, bạn không cần tốn nhiều công sức cho công đoạn này.

Công việc của bạn là xác định các thành phần giao diện người dùng (thông số UI) và hãy để React Native lo phần còn lại. Tuy nhiên, điều đó không có nghĩa là hiệu năng ứng dụng của bạn được tối ưu sẵn. Để tạo ra các ứng dụng nhanh và mượt. Bạn phải hiểu cách nó tương tác với các API nền tảng bên dưới.



Tập 1

Cải thiện hiệu năng bằng cách hiểu rõ chi tiết bên trong React Native.

Giới thiệu

Trong tập này, chúng ta sẽ đào sâu hơn vào các nút thắt cổ chai hiệu năng phổ biến nhất của React Native. Điều này không chỉ đơn giản là giới thiệu kỹ thuật nâng cao trong React Native, mà còn giúp bạn cải thiện đáng kể tính ổn định và hiệu năng cho ứng dụng của bạn bằng cách thực hiện các tinh chỉnh và thay đổi nhỏ.

Phần tiếp theo sẽ tập trung vào điểm đầu tiên nhất liên quan tới chiến lược tối ưu hiệu năng React Native: UI re-renders. Nó là phần rất quan trọng trong quá trình tối ưu React Native vì nó cho phép giảm lượng pin được tiêu thụ, mà điều này lại ảnh hưởng trực tiếp đến trải nghiệm người dùng trong app của bạn.

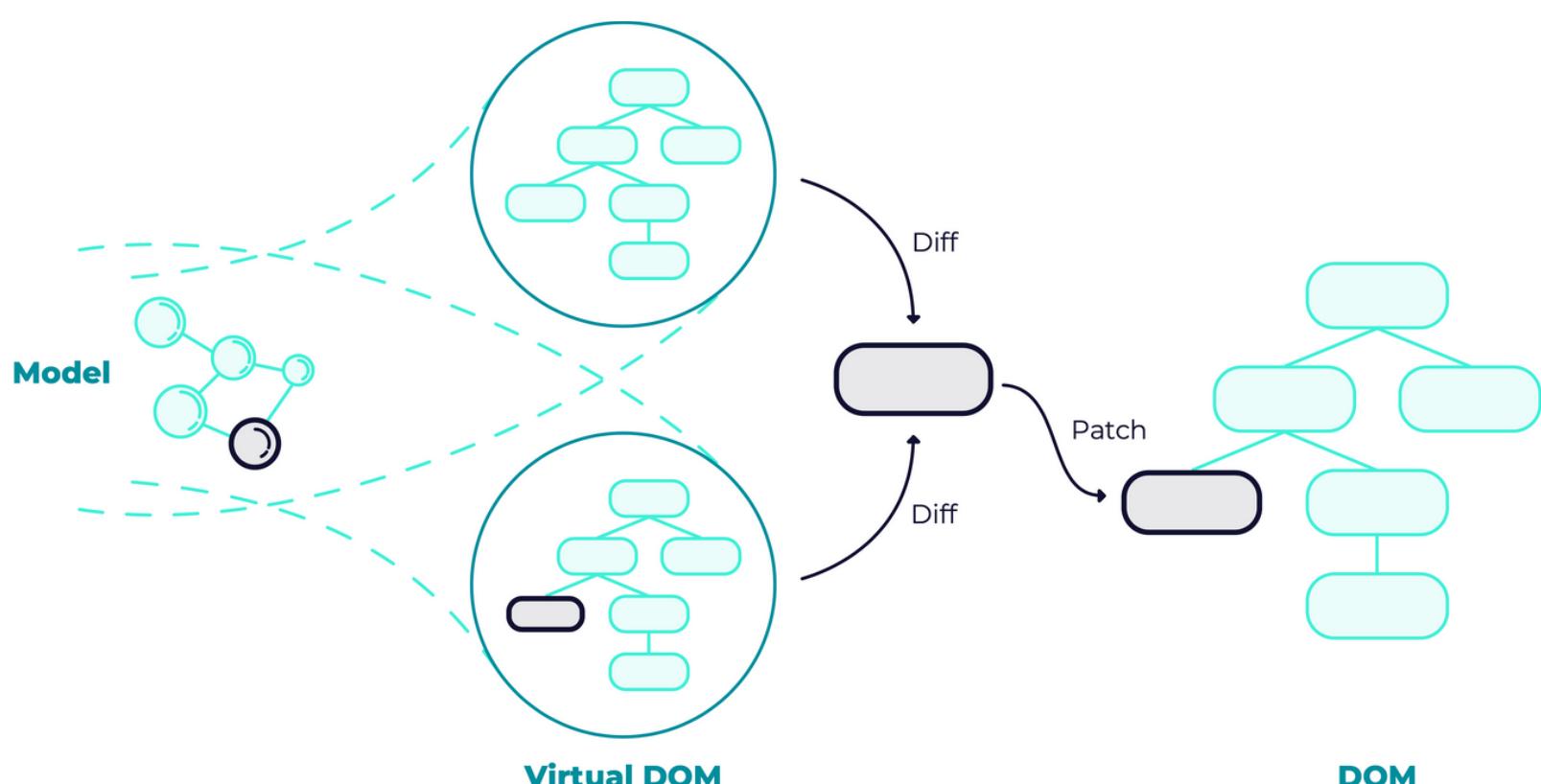
1. Chú ý đến việc UI re-renders

Tối ưu số lần các state tính toán, hãy nhớ rằng chúng ta có 2 loại component: pure (thuần) & memorized(ghi nhớ) để làm cho ứng dụng của chúng ta chạy nhanh hơn nhưng lại cần tài nguyên ít hơn.

Vấn đề: Việc cập nhật state không chính xác gây ra các vòng render không liên quan / hoặc thiết bị quá chậm

Như đã nói ở trên, React Native đảm nhận việc render UI ứng dụng cho bạn. Công việc của bạn là xác định toàn bộ các component nào cần sử dụng và lắp ghép chúng lại với nhau để tạo thành một bản giao diện hoàn chỉnh. Với cách tiếp cận này, bạn không thể kiểm soát được các render lifecycle của ứng dụng.

Nói cách khác, khi nào và làm thế nào để vẽ lại toàn bộ những thứ trên màn hình hoàn toàn là trách nhiệm của React Native. React tìm kiếm những sự thay đổi mà bạn đã thực hiện với các Component của bạn, so sánh chúng và thực hiện số lượng cập nhật thực tế được yêu cầu nhỏ nhất.



Quy tắc ở đây rất đơn giản - theo mặc định, Component có thể re-render nếu Component cha của nó đang re-render hoặc props khác nhau. Điều này có nghĩa là phương thức render các Component của bạn đôi khi có thể chạy ngay cả khi các props của nó không hề thay đổi. Đây là một sự đánh đổi có thể tạm chấp nhận được trong một số tình huống, vì so sánh 2 đối tượng (props trước đó và hiện tại) sẽ mất nhiều thời gian hơn.

Tác động tiêu cực đến hiệu năng ứng dụng, giao diện giật lag hoặc làm giảm FPS

Mặc dù các phỏng đoán trên có thể đúng trong hầu hết các trường hợp, nhưng việc thực hiện quá nhiều các hành động tính toán có thể gây ra các vấn đề đến hiệu năng, đặc biệt đối với các dòng điện thoại cấp thấp.

Do đó, bạn có thể thấy giao diện trên ứng dụng của bạn nhấp nháy (khi việc cập nhật đang thực hiện) hoặc khung hình giảm(frames dropping) trong khi có các Animation đang diễn ra và sắp có bản cập nhật mới

Lưu ý: Bạn không nên thực hiện bất kỳ tối ưu ứng dụng của bạn vào thời điểm ban đầu bởi vì nó có thể gây ra các phản tác dụng. Bạn chỉ nên xem xét các vấn đề ngay khi bạn phát hiện khung hình giảm (frames dropping) hoặc hiệu năng của ứng dụng không được như bạn mong muốn.

Ngay khi bạn phát hiện một trong các vấn đề trên, đây mới là thời điểm thích hợp để bạn xem xét sâu hơn vào lifecycle ứng dụng của bạn và tìm kiếm các tính toán không cần thiết mà bạn không muốn nó xảy ra.

Giải pháp: Tối ưu số lần các state tính toán và hãy nhớ sử dụng các pure và memorized component khi cần thiết

Có rất nhiều cách làm cho ứng dụng của bạn thực hiện các cycle render không cần thiết và sẽ có giá trị hơn nếu nó được đề cập riêng trong một bài viết khác. Trong chương này, chúng ta sẽ tập trung vào 2 trường hợp phổ biến - sử dụng một Component được điều khiển(controlled), chẳng hạn như TextInput và global state.

Component được kiểm soát và không cần kiểm soát

Chúng ta hãy bắt đầu với ý đầu tiên. Hầu hết các ứng dụng React Native chứa ít nhất một TextInput được kiểm soát bởi state của component như đoạn code sau.

```

import React, { Component } from 'react';
import { TextInput } from 'react-native';

export default function UselessTextInput() {
  const [value, onChangeText] = React.useState('Text');

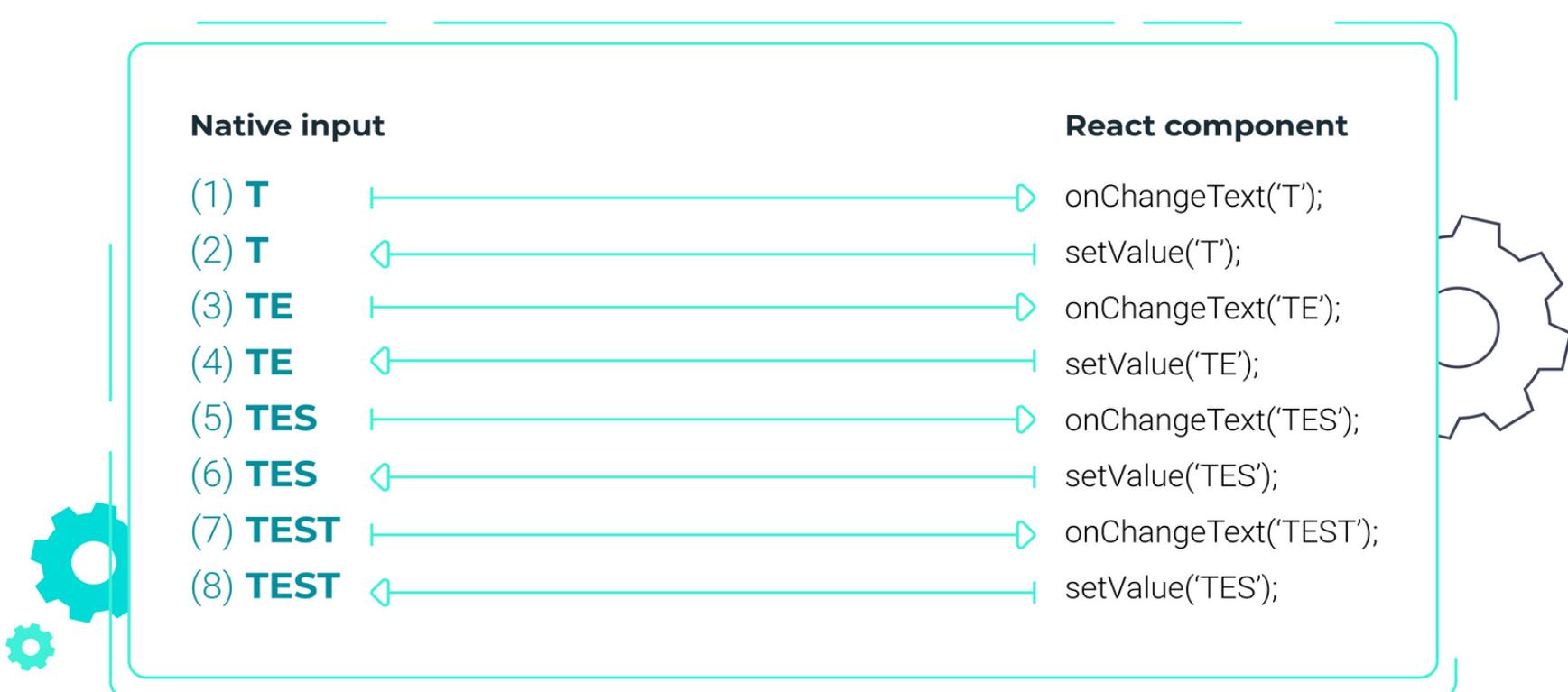
  return (
    <TextInput
      style={{ height: 40, borderColor: 'gray', borderWidth: 1 }}
      onChangeText={text => onChangeText(text)}
      value={value}
    />
  );
}

```

Xem thêm tại: <https://snack.expo.io/q75wcVYnE>

Code phía trên sẽ chạy trong hầu hết các trường hợp. Tuy nhiên đối với các thiết bị yếu, và trong một số trường hợp người dùng gõ các ký tự vào TextInput quá nhanh, nó có thể gây ra vấn đề với việc cập nhật hiển thị trên màn hình.

Lý do dẫn đến vấn đề trên hết sức đơn giản - Bản chất React Native chạy bất đồng bộ (asynchronous). Để hiểu rõ hơn những gì đang diễn ra, trước tiên hãy cùng xem thứ tự của các hoạt động xảy ra trong quá trình User typing và <TextInput /> nhận vào một ký tự mới.



Ngay khi User bắt đầu nhập một ký tự mới vào bàn phím native, các cập nhật sẽ được gửi đến React Native thông qua prop onChangeText (hoạt động như sơ đồ số 1). React xử lý thông tin đó và cập nhật state tương ứng của nó thông qua việc gọi function setState. Tiếp theo, các Component được điều khiển đồng bộ hóa giá trị JavaScript của nó với giá trị của native Component (hoạt động như sơ đồ số 2)

Lợi ích của hướng tiếp cận này rất đơn giản. React là nguồn gốc để quyết định các giá trị đầu vào của bạn. Kỹ thuật này cho phép bạn thay đổi giá trị đầu vào của người dùng khi nó xảy ra, chẳng hạn như thực hiện chức năng validation, che hoặc sửa đổi hoàn toàn dữ liệu.

Mặc dù cách làm trên trông có vẻ rõ ràng và tuân thủ đúng các quy tắc hoạt động của React nhưng thật không may với hướng tiếp cận trên, nó lại lộ rõ nhược điểm khi các tài nguyên có sẵn bị hạn chế hoặc user gõ các ký tự rất

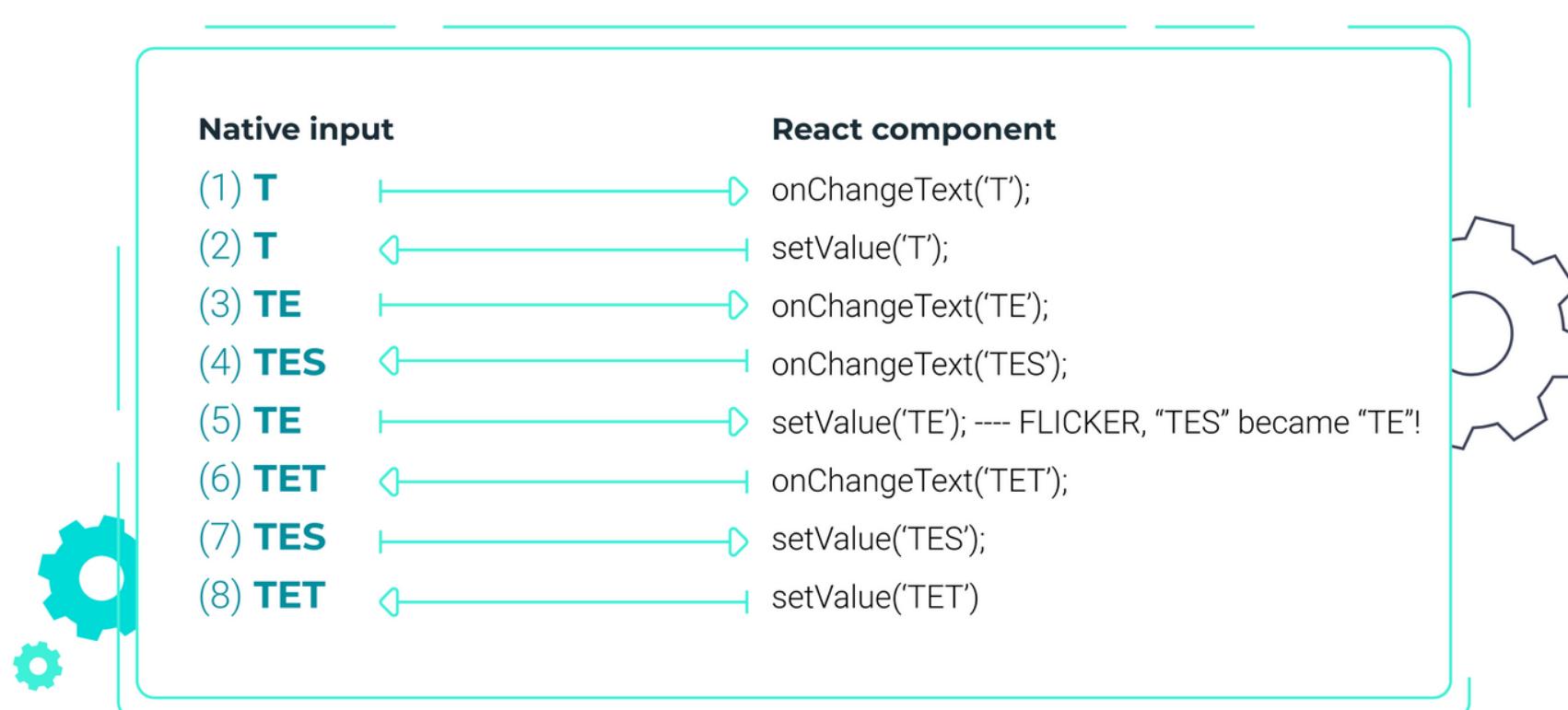


Diagram that shows what happens while typing TEST too fast

Khi các cập nhật thông qua onChangeText đến trước khi React Native đồng hóa các dữ liệu, giao diện sẽ bắt đầu nhấp nháy, giật, lag. Bản cập nhật đầu tiên (hoạt động 1 và 2) thực hiện mà không gặp bất kỳ các lỗi (issues) nào khi User bắt đầu nhập chữ T.

Tiếp đến, khi hoạt động 3 diễn ra, tiếp theo đó là một cập nhật khác (hoạt động 4). User đã nhập vào E & S trong khi đó React Native đang bận làm việc khác, khiến cho quá trình đồng bộ chữ E bị chậm lại. Do đó, bàn phím native sẽ tạm thời thay đổi giá trị của nó trở lại từ TES thành TE

Bây giờ, user đã nhập các ký tự khác quá nhanh để nó thay đổi giá trị mới, trong khi giá trị hiện tại của TextInput chỉ vừa được thiết lập là TE trong khoảng một giây. Kết quả là những thay đổi khác đến (hoạt động 6), với giá trị TET. Đây không phải là sự cố ý cũng như user không mong đợi giá trị của TextInput bị chuyển từ TES thành TE

Cuối cùng, hoạt động 7 đồng bộ hoá lại dữ liệu đầu vào chính xác với những gì user đã gõ 1 vài ký tự trước đó (hoạt động 4 báo cho chúng ta biết sự thay đổi thành TES). Thật không may, nó đã nhanh chóng bị ghi đè lên bởi một sự thay đổi khác (hoạt động 8), bản cập nhật này giá trị của nó đã chuyển thành TET - giá trị cuối cùng của dữ liệu đầu vào.

Nguyên nhân chính của tình huống này đó là thứ tự hoạt động. Nếu hoạt động số 5 được thực thi trước hoạt động số 4, mọi thứ sẽ chạy trơn tru hơn. Ngoài ra, nếu user không nhập vào ký tự T khi giá trị là TE thay vì TES, giao diện sẽ bị giựt lag nhưng giá trị đầu vào của dữ liệu sẽ chuẩn xác.

Một trong các giải pháp cho vấn đề đồng bộ hoá dữ liệu là loại bỏ hoàn toàn giá trị bên trong prop khỏi TextInput. Từ đó, dữ liệu sẽ được di chuyển theo một chiều từ tầng native đến tầng JavaScript, loại bỏ được các vấn đề hiện tại của chúng ta.

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

export default function PizzaTranslator() {
  const [text, setText] = useState('');
  return (
    <View style={{padding: 10}}>
      <TextInput
        style={{height: 40}}
        placeholder="Type here to translate!"
        onChangeText={text => setText(text)}
        defaultValue={text}
      />
      <Text style={{padding: 10, fontSize: 42}}>
```

```
{text.split(' ').map((word) => word && ).join(' ')}

</Text>
</View>
);
```

Xem thêm tại: <https://snack.expo.io/q75wcVYnE>

Tuy nhiên, theo như [@nparashuram](#) chia sẻ trên kênh Youtube riêng của anh ấy, chỉ giải pháp đó thôi chưa đủ để cover hết các trường hợp. Ví dụ, khi thực hiện việc kiểm tra giá trị đầu vào (validation) hoặc che dấu dữ liệu, bạn vẫn cần kiểm soát dữ liệu mà người dùng nhập vào và thay đổi những gì mà TextInput hiển thị. Team React Native nhận thức được các hạn chế đó và hiện họ đang cải tiến một kiến trúc mới để giải quyết được vấn đề trên.

State toàn cục

Những vấn đề về hiệu năng phổ biến khác là cách mà component phụ thuộc vào các global state (state toàn cục) của ứng dụng. Trường hợp tệ nhất là khi state của một control thay đổi ví dụ như TextInput hoặc CheckBox thì nó sẽ truyền sự thay đổi state đó đến toàn bộ những nơi trong ứng dụng đang sử dụng những control đó. Lý do xảy ra vấn đề này bắt nguồn từ việc thiết kế quản lý state toàn cục không tốt.

Chúng tôi khuyến khích sử dụng các thư viện đặc biệt như Redux hoặc Overmind.js để xử lý việc quản lý state tốt và tối ưu hơn.

Trước hết, thư viện quản lý state của bạn chỉ nên dành sự quan tâm đến việc cập nhật dữ liệu của các component khi các dữ liệu con của nó được xác định đã thay đổi - đây là chức năng mặc định khi sử dụng Redux.

Tiếp theo, nếu component sử dụng dữ liệu ở dạng khác với những gì được lưu trữ trong state của bạn, component đó có thể sẽ bị render lại, mặc dù dữ liệu thực chất không đổi. Để tránh tình huống này, bạn nên tạo 1 “bộ chọn lựa” với chức năng chính là ghi nhớ lại các kết quả phát sinh cho đến khi nó thực sự thay đổi.

```

import { createSelector } from 'reselect'

const getVisibilityFilter = (state) => state.visibilityFilter
const getTodos = (state) => state.todos

const getVisibleTodos = createSelector(
  [ getVisibilityFilter, getTodos ],
  (visibilityFilter, todos) => {
    switch (visibilityFilter) {
      case 'SHOW_ALL':
        return todos
      case 'SHOW_COMPLETED':
        return todos.filter(t => t.completed)
      case 'SHOW_ACTIVE':
        return todos.filter(t => !t.completed)
    }
  }
)

const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state)
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
)(TodoList)

export default VisibleTodoList

```

A typical example of selectors with redux state management library

Một vấn đề phổ biến khi hiệu năng ứng dụng bị giảm đi đó là niềm tin vào việc tự viết trên React Context để thay thế cho thư viện quản lý state. Nó có vẻ rất hữu ích trong thời điểm ban đầu bởi vì nó làm code của chúng ta giảm đi rất nhiều so với việc sử dụng theo cấu trúc quản lý state. Nhưng việc sử dụng cơ chế này mà không có các phương thức ghi nhớ chính xác sẽ dẫn đến những hạn chế về hiệu năng rất lớn cho ứng dụng của bạn. Và cuối cùng thì bạn sẽ phải refactor việc quản lý state này thành cơ chế quản lý state của redux, bởi vì nó sẽ trở nên dễ dàng hơn.

Bạn cũng có thể tối ưu ứng dụng của bạn trên cấp độ từng Component riêng lẻ. Chỉ cần sử dụng Pure component thay vì các component thông thường và sử dụng trình bao bọc (memo wrapper) cho các chức năng chính của nó để giúp bạn giảm thiểu rất nhiều re-render trên giao diện. Nó có thể không có tác động nào ở lần xuất hiện đầu tiên, nhưng bạn sẽ nhìn thấy sự khác biệt khi mà các component không được ghi nhớ được sử dụng trong danh sách hiển thị dữ liệu lớn xuất hiện.

Đừng cố gắng sử dụng các kỹ thuật này, bởi vì cách tối ưu ứng dụng như vậy thường rất hiếm sử dụng trong thực tế và chỉ phù hợp với vài trường hợp cụ thể

Lợi ích: Tài nguyên cần sử dụng càng ít, ứng dụng của chúng ta chạy càng nhanh

Bạn phải luôn nhớ một điều rằng phải tối ưu ứng dụng của chúng ta chạy nhanh nhất có thể nhưng cũng đừng cố gắng tối ưu hóa mọi thứ từ sớm bởi vì điều đó thực sự không cần thiết trong giai đoạn đầu. Bạn sẽ lãng phí rất nhiều thời gian vào việc giải quyết các vấn đề nan giải và tự đưa mình vào thế khó

Hầu hết các vấn đề về hiệu năng mà chúng ta không thể giải quyết được thường là do kiến trúc ứng dụng xây dựng ban đầu khá tệ cùng với việc quản lý state không tốt, cho nên hãy đảm bảo bạn đã thiết kế kiến trúc và quản lý tốt từ ban đầu. Các component thông thường chúng ta sử dụng thường gây ra các vấn đề về hiệu năng miễn là bạn sử dụng các Pure component hoặc memo wrapper

Sau tất cả, hãy nhớ các bước cần thực hiện trong đầu rằng, ứng dụng của bạn nên thực hiện ít việc tính toán hơn và tiêu tốn tài nguyên ít hơn để hoàn thành một công việc xử lý nào đó. Kết quả là nó sẽ dẫn đến việc ứng dụng sẽ ít tốn pin hơn và người dùng sẽ có được trải nghiệm tốt hơn khi sử dụng ứng dụng, không còn vấn đề bị giật, lag giao diện nữa.

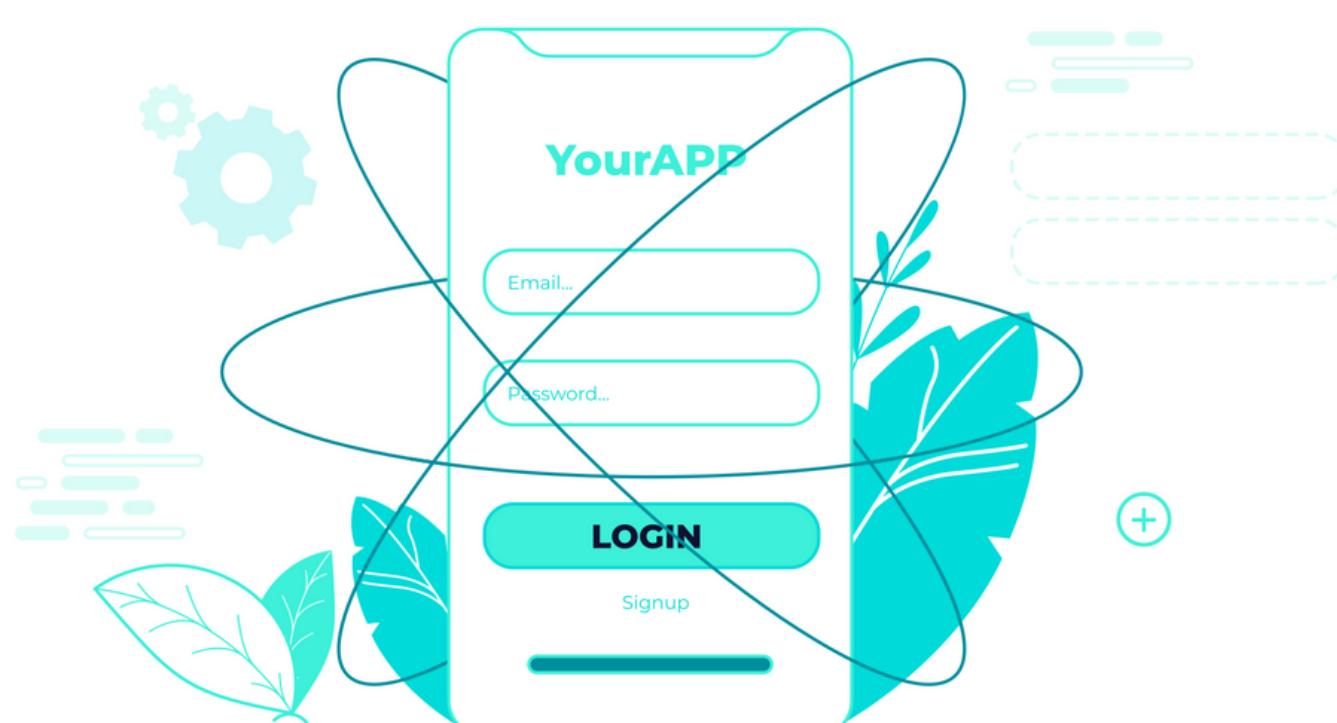
2. Sử dụng các Component chuyên dụng cho các bố cục (layout) nhất định

Tìm hiểu cách sử dụng các component chuyên dụng có thứ tự cao hơn(higher-ordered) được xây dựng sẵn của React Native giúp nâng cao trải nghiệm sử dụng cho người dùng và hiệu năng ứng dụng.

Vấn đề: Bạn không biết hoặc nhận ra React Native đã cung cấp cho chúng ta các component chuyên dụng.

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

Trong React Native mọi thứ đều là component. Ở cuối các cấu trúc component, có một số loại gọi là các component nguyên thuỷ (primitive), chẳng hạn như Text, View, TextInput. Những component này được thực hiện bởi React Native và tương ứng với nền tảng mà bạn đang nhắm đến để hỗ trợ hầu hết các thao tác đơn giản cho người dùng trong lúc sử dụng ứng dụng.

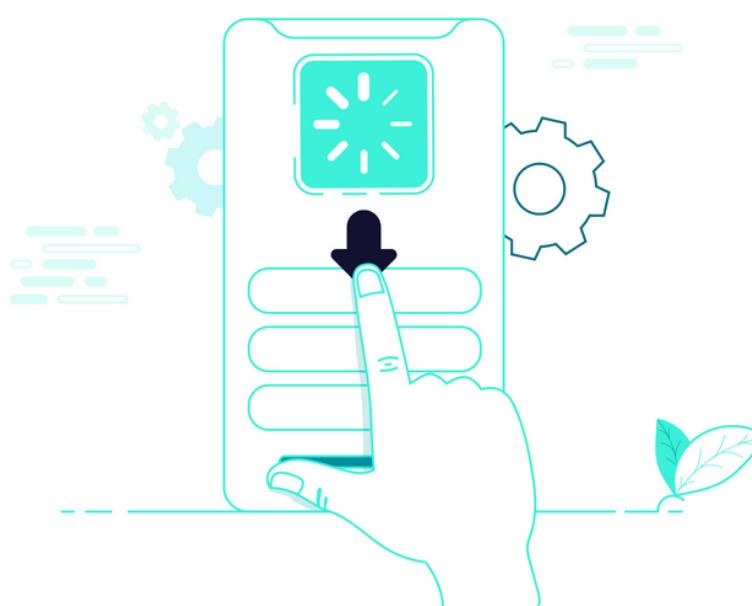


Ngoài các primitive component, React Native còn cung cấp thêm cho chúng ta các higher-order component để phục vụ cho việc tối ưu ứng dụng trong các mục đích nhất định.

Việc không biết đến các higher-order component hoặc không sử dụng chúng có thể gây ra các ảnh hưởng cực lớn cho hiệu năng ứng dụng của bạn, đặc biệt khi bạn đã chuyển từ môi trường phát triển (develop) sang môi trường chạy dữ liệu thật. Hiệu năng kém có thể gây tổn hại nghiêm trọng đến trải nghiệm người dùng. Hậu quả có thể dẫn đến là khách hàng hoặc người dùng không hài lòng với sản phẩm của bạn và chuyển sang sử dụng sản phẩm của đối thủ

Việc không sử dụng các thành phần chuyên biệt sẽ ảnh hưởng đến hiệu năng và trải nghiệm người dùng khi dữ liệu trong ứng dụng lớn dần.

Nếu bạn không sử dụng các thành phần chuyên biệt, bạn đã từ chối việc tối ưu hiệu năng ứng dụng và có nguy cơ làm giảm trải nghiệm người dùng khi ứng dụng đã được đưa vào môi trường thực tế. Bạn nên lưu ý rằng khi phát triển ứng dụng trong môi trường develop, các dữ liệu thường là giả (mocked data) cho nên nó thường rất bé và không phản ánh được quy mô của dữ liệu trong môi trường thực tế (production) .



Các component chuyên biệt thường toàn diện hơn và cung cấp nhiều API để hỗ trợ cho chúng ta gần như tất cả trường hợp có thể xảy ra.

Giải pháp: Luôn sử dụng các component chuyên biệt có sẵn, ví dụ: FlatList cho các danh sách

Chúng ta sẽ lấy ví dụ là một danh sách dài. Mỗi ứng dụng thường chứa ít nhất một danh sách tại một thời điểm nào đó.

Cách nhanh và sai lầm nhất để khởi tạo danh sách chứa nhiều phần tử đó là kết hợp ScrollView & View.

Tuy nhiên, ví dụ trên sẽ nhanh chóng có vấn đề khi dữ liệu chúng ta ngày càng nhiều. Để đối phó với các dữ liệu lớn, scroll không giới hạn (infinite scrolling), và quản lý bộ nhớ, chúng ta có một component tuyệt vời để đáp ứng các điều trên - `FlatList`. `FlatList` là một component chuyên dụng của React Native để hiển thị và làm việc với cấu trúc dữ liệu như vậy. Hãy so sánh hiệu năng của việc thêm phần tử mới vào danh sách với `ScrollView`.

```
import React, { Component, useCallback, useState } from 'react';
import { ScrollView, View, Text, Button } from 'react-native';

const objects = [
  ['avocado', '🥑'],
  ['apple', '🍏'],
  ['orange', '🍊'],
  ['cactus', '🌵'],
  ['eggplant', '🍆'],
  ['strawberry', '🍓'],
  ['coconut', '🥥'],
];

const getRanomItem = () => {
  const item = objects[~~(Math.random() * objects.length)];
  return {
    name: item[0],
    icon: item[1],
    id: Date.now() + Math.random(),
  };
};

const _items = Array.from(new Array(5000)).map(() => getRanomItem());

export default function List() {
  const [items, setItems] = useState(_items);

  const addItem = useCallback(() => {
    setItems([getRanomItem()].concat(items));
  }, []);
}
```

```

return (
  <View style={{marginTop: 20}}>
    <Button title="add item" onPress={addItem} />
    <ScrollView>
      {items.map(({name, icon}) => (
        <View
          style={{
            borderWidth: 1,
            margin: 3,
            padding: 5,
            flexDirection: 'row',
          }}>
          <Text style={{fontSize: 20, width: 150}}>{name}</Text>
          <Text style={{fontSize: 20}}>{icon}</Text>
        </View>
      ))}
    </ScrollView>
  </View>
);

```

Xem thêm tại <https://snack.expo.io/qjtEVHrdV>

đối với danh sách dựa trên FlatList

```

import React, { Component, useCallback, useState } from 'react';
import { View, Text, Button, FlatList, SafeAreaView } from 'react-native';

const objects = [
  ['avocado', '🥑'],
  ['apple', '🍏'],
  ['orange', '🍊'],
  ['cactus', '🌵'],
]

```

```

['eggplant', '🍆'],
['strawberry', '🍓'],
['coconut', '🥥'],
];

const getRanomItem = () => {
  const item = objects[~~(Math.random() * objects.length)].split(' ');
  return {
    name: item[0],
    icon: item[1],
    id: Date.now() + Math.random(),
  };
};

const _items = Array.from(new Array(5000)).map(() => getRanomItem());

export default function List() {
  const [items, setItems] = useState(_items);

  const addItem = useCallback(() => {
    setItems([getRanomItem()].concat(items));
  }, [items]);

  return (
    <View style={{ marginTop: 20 }}>
      <Button title="add item" onPress={addItem} />
      <FlatList
        data={items}
        keyExtractor={({ id }) => id}
        renderItem={({ item: { name, icon } }) => (
          <View
            style={{
              borderWidth: 1,
              margin: 3,
              padding: 5,
            }
          >
            <Image alt={icon} />
            <Text>{name}</Text>
          </View>
        )}
      </FlatList>
    </View>
  );
}

```

```

        flexDirection: 'row',
    }}>
    <Text style={{ fontSize: 20, width: 150 }}>{item[0]}</Text>
    <Text style={{ fontSize: 20 }}>{item[1]}</Text>
</View>
)
/
</View>
);
}

```

Xem thêm tại: <https://snack.expo.io/1muB1wKya>

Trong ví dụ trên, đối với danh sách 5000 phần tử, phiên bản ScrollView thậm chí còn không thể scroll mượt mà khi thao tác với danh sách. Bạn có thấy được sự khác biệt rất lớn không?

Rốt cuộc thì FlatList cũng sử dụng các component ScrollView & View tương tự - vậy khác gì đây nhỉ?

Chìa khoá chính là các logic đã được trừu tượng hoá nằm trong FlatList. Nó chứa rất nhiều kinh nghiệm và sự tính toán nâng cao của JavaScript để giảm thiểu các render không liên quan xảy ra cũng như chỉ render có đủ những thứ bạn đang nhìn thấy trên màn hình => điều đó giúp chúng ta trải nghiệm scrolling khi thao tác với danh sách lúc nào cũng đạt được 60 FPS (frame per second).

Nếu bạn chỉ sử dụng FlatList thôi thì cũng chưa đáp ứng được hết một số trường hợp. Điều giúp cho FlatList tối ưu hoá hiệu năng của ứng dụng đó là không render những phần tử không hiển thị trên màn hình. Nhưng đánh đổi cho việc đó là layout phải tính toán để hiển thị. FlatList phải tính toán layout của bạn để xác định xem phải tốn bao nhiêu không gian trong vùng scroll nên được dành cho những phần tử tiếp theo.

Đối với danh sách chứa các phần tử hiển thị layout phức tạp, nó có thể làm chậm sự tương tác của người dùng với FlatList. Mỗi khi FlatList chạm đến đáy để hiển thị dữ liệu tiếp theo, nó sẽ đợi tất cả các phần tử mới hiển thị để tính toán chiều cao của chúng.



Tuy nhiên, bạn có thể triển khai getItemHeight () để xác định chiều cao phần tử từ trước mà không cần đo lường. Nó không giúp cho các items xác định được vị trí nếu không có ràng buộc về chiều cao. Bạn có thể tính toán giá trị dựa trên số dòng văn bản và các ràng buộc khác về bố cục. Chúng tôi khuyên bạn nên sử dụng thư viện react-native-text-size để tính toán chiều cao của văn bản được hiển thị cho tất cả các mục danh sách cùng một lúc. Trong trường hợp của chúng tôi, nó đã cải thiện đáng kể khả năng phản hồi cho các sự kiện scroll của FlatList trên Android.

Lợi ích: Ứng dụng của bạn sẽ hoạt động nhanh hơn, hiển thị cấu trúc dữ liệu phức tạp và bạn chọn nó để cải thiện thêm hiệu năng.

=> Nhờ vào việc sử dụng các component chuyên biệt, ứng dụng của bạn sẽ chạy nhanh nhất có thể. Bạn hiển nhiên sẽ được hỗ trợ tối ưu hiệu năng tự động từ React Native, điều đó vẫn đang được cập nhật từ xưa đến nay.

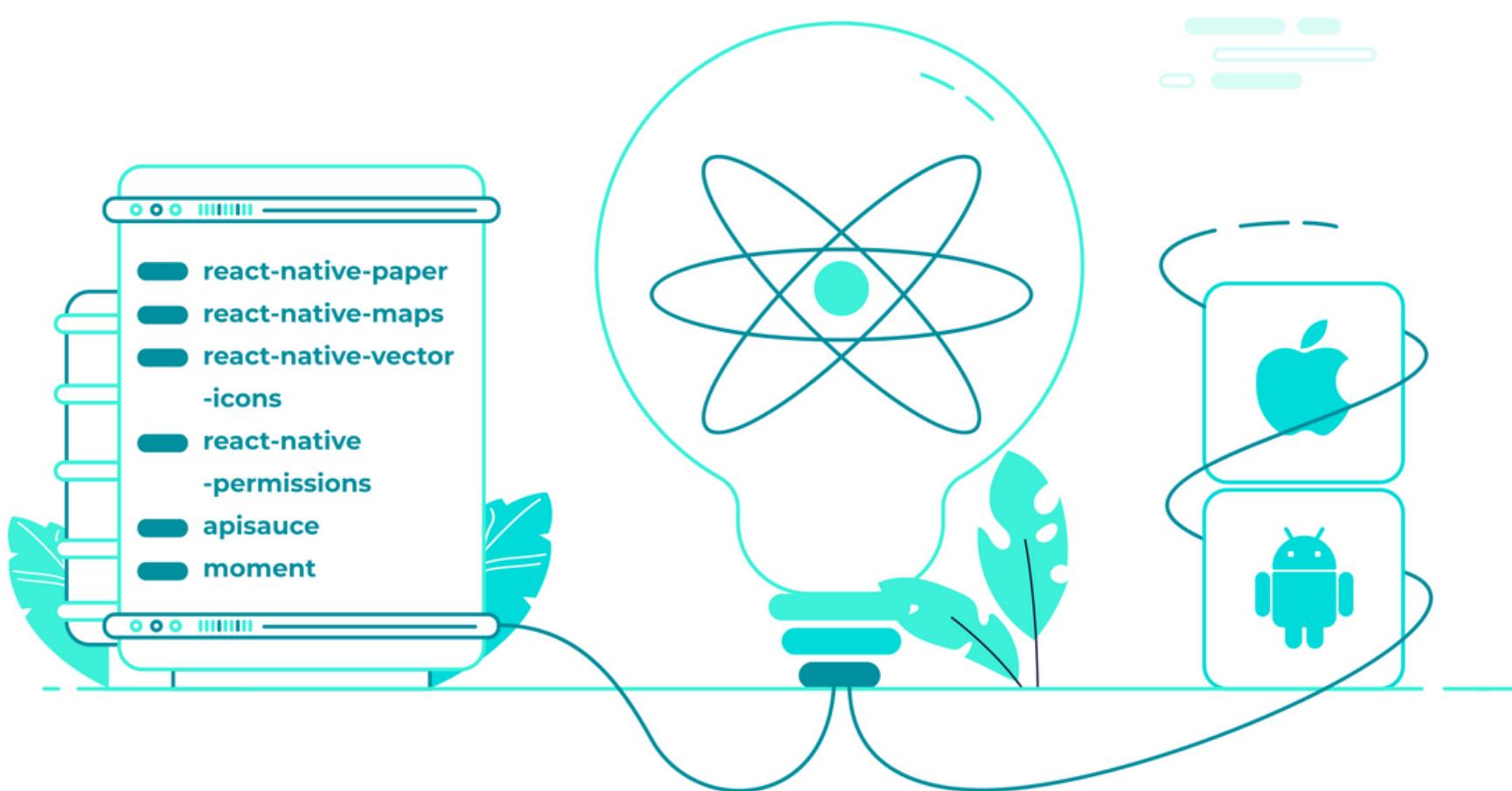
Đồng thời, bạn cũng đã tiết kiệm được rất nhiều thời gian cho việc xây dựng lại hầu hết các mẫu giao diện phổ biến từ đầu. Sticky Section Headers, Pull To Refresh - các component này đã được hỗ trợ mặc định nếu bạn chọn sử dụng cùng với FlatList.

3. Hãy suy nghĩ kỹ trước khi chọn một thư viện bên ngoài.

Chọn đúng thư viện JavaScript phù hợp có thể giúp bạn tăng tốc độ và hiệu năng của ứng dụng.

Vấn đề: Bạn chọn thư viện nhưng lại không tìm hiểu xem thử bên trong có gì.

Việc phát triển JavaScript giống như là việc lắp ráp các ứng dụng từ những khối nhỏ hơn. Ở một mức độ nhất định, nó rất giống với việc xây dựng các ứng dụng react native. Thay vì tạo ra React components từ đầu, bạn sẽ săn lùng những thư viện JavaScript để hiện thực hóa những gì mà mình nghĩ. Hệ sinh thái JavaScript thúc đẩy cách tiếp cận như vậy để phát triển và khuyến khích cấu trúc các ứng dụng xoay quanh các modules nhỏ và có thể tái sử dụng.



Kiểu hệ sinh thái này có nhiều ưu điểm, nhưng cũng có một số nhược điểm nghiêm trọng. Một trong số đó là các developers có thể khó chọn từ nhiều thư viện để hỗ trợ cùng một trường hợp sử dụng.

Khi chọn một thư viện để sử dụng trong dự án tiếp theo, họ thường nghiên cứu các chỉ số cho biết thư viện có hoạt động tốt và được duy trì tốt hay không. Chẳng hạn như số sao trên Github, số lượng phát hành, cộng tác viên và PRs.

Họ thường có xu hướng bỏ qua kích thước của thư viện, số lượng các tính năng được hỗ trợ và số lượng các thư viện hỗ trợ nó bên ngoài. Họ giả định rằng vì React Native được viết bằng JavaScript và bao gồm chuỗi công cụ hiện có nên họ sẽ dùng những tiêu chuẩn có sẵn và các best practices mà họ biết từ bên web.

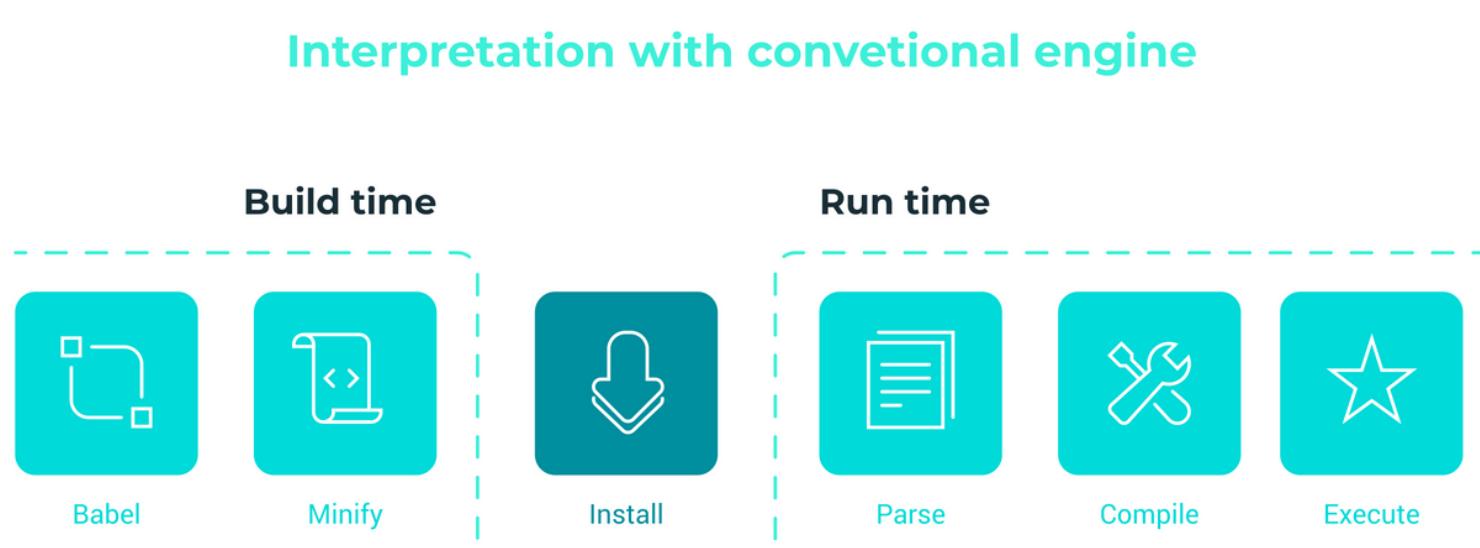
Sự thật là việc phát triển mobile app có những khác biệt từ nền tảng và chúng có những nguyên tắc riêng. Ví dụ: trong khi kích thước của assets là quan trọng trong phát triển web, thì nó không quan trọng trong React Native, vì chúng được đặt trong filesystem.

Sự khác biệt chính nằm ở hiệu suất của các thiết bị di động và công cụ được sử dụng để đóng gói và biên dịch ứng dụng.

Mặc dù bạn sẽ không thể làm gì nhiều về các giới hạn của thiết bị, nhưng bạn có thể kiểm soát mã JavaScript của mình. Nói chung, ít mã hơn có nghĩa là thời gian mở app nhanh hơn. Và một trong những yếu tố quan trọng nhất ảnh hưởng đến kích thước tổng thể code của bạn là những thư viện.

Các thư viện phức tạp cản trở tốc độ của những ứng dụng của bạn.

Không giống như một ứng dụng native hoàn toàn, một ứng dụng React Native chứa một JavaScript bundle cần được tải vào bộ nhớ. Sau đó, nó được phân tích cú pháp và thực thi bởi JavaScript VM. kích thước tổng thể của mã JavaScript là một yếu tố quan trọng.



Xem thêm tại: <https://snack.expo.io/7H5S504j3>

Trong khi điều đó xảy ra, ứng dụng vẫn ở trạng thái tải. Chúng ta thường mô tả quá trình này là **TTI - Time to Interactive**. Đó là khoảng thời gian được biểu thị bằng milisecond (tốt, hy vọng) từ lúc app của bạn khởi động cho đến khi nó sẵn sàng (nhận tương tác).

Thật không may, Metro - Bộ đóng gói sử dụng cho React Native - không hỗ trợ tree shaking

Nó có nghĩa là tất cả mã mà bạn kéo từ `npm` và được đưa vào dự án của bạn sẽ có trong production bundle của bạn, được tải vào bộ nhớ và được phân tích cú pháp.

Điều đó có thể có tác động tiêu cực đến tổng thời gian khởi động ứng dụng của bạn.

Giải pháp: Hãy chọn lọc hơn và sử dụng các thư viện nhỏ hơn, chuyên biệt hơn

Cách dễ nhất để khắc phục vấn đề này là sử dụng chiến lược phù hợp để thiết kế dự án từ trước.

Nếu bạn chuẩn bị kéo một thư viện phức tạp, hãy kiểm tra xem có các lựa chọn thay thế nhỏ hơn có chức năng mà bạn đang tìm kiếm không

Dưới đây là một ví dụ: Một trong những tính năng thường thấy nhất là thao tác ngày tháng. Hãy tưởng tượng bạn sắp tính thời gian đã trôi qua. Thay vì kéo toàn bộ thư viện moment.js xuống (67,9 Kb) để làm việc này:

```
import moment from 'moment'  
const date = moment("12-25-1995", "MM-DD-YYYY");
```

Parsing date with moment.js

Chúng ta có thể sử dụng day.js (chỉ 2Kb) nhỏ hơn đáng kể và chỉ cung cấp chức năng mà chúng ta cần.

```
import dayjs from 'dayjs'  
const date = dayjs("12-25-1995", "MM-DD-YYYY");
```

Parsing date with day.js

Nếu không có lựa chọn thay thế, nguyên tắc chung là kiểm tra xem bạn có thể dùng một phần nhỏ hơn của thư viện hay không.

Ví dụ, nhiều thư viện như lodash đã tự phân chia thành các bộ tiện ích nhỏ hơn và hỗ trợ các môi trường nơi có thể loại bỏ mã chết không khả dụng.

Giả sử bạn muốn sử dụng bản đồ lodash. Thay vì nhập toàn bộ thư viện, như được trình bày ở đây:

```
import { map } from 'lodash';  
  
const square = x => x * x;  
  
map([4, 8], square);
```

Using lodash map by importing the whole library

Bạn có thể chỉ nhập một package duy nhất:

```
import map from 'lodash/map';  
  
const square = x => x * x;  
  
map([4, 8], square);
```

Using lodash map by importing only single function

Do đó, bạn có thể hưởng lợi từ các tiện ích là một phần của gói lodash mà không cần kéo tất cả chúng vào gói ứng dụng.

Lợi ích: Ứng dụng của bạn tải nhanh hơn, điều này có thể tạo ra sự khác biệt

Lập trình di động là một mảng cực kỳ cạnh tranh, với rất nhiều ứng dụng được thiết kế để phục vụ các mục đích tương tự và lôi kéo khách hàng giống nhau. Thời gian khởi động nhanh hơn, tương tác mượt mà hơn và UX là cách duy nhất để bạn nổi bật.

Theo báo cáo của Akamai về kết quả bán lẻ trực tuyến, chỉ cần chậm trễ một giây trong thời gian tải trên thiết bị di động có thể giảm tới 20% tỷ lệ chuyển đổi.

Đó là lý do tại sao bạn không nên hạ thấp tầm quan trọng của việc chọn đúng bộ thư viện.

Việc lựa chọn nhiều hơn với các phụ thuộc của bên thứ ba thoát đầu có vẻ không liên quan. Nhưng tất cả số mili giây tiết kiệm được cộng lại sẽ thành lợi nhuận đáng kể theo thời gian.

4. Luôn nhớ sử dụng những thư viện dành riêng cho nền tảng mobile

Sử dụng những thư viện dành riêng cho mobile giúp xây dựng những tính năng nhanh hơn trên nhiều nền tảng cùng một lúc. Dù vậy vẫn không ảnh hưởng đến hiệu năng và trải nghiệm người dùng.

Vấn đề: Bạn sử dụng những thư viện dành cho web mà những thư viện đó không tối ưu cho mobile.

Như đã nói ở trên, một trong những ưu điểm của React Native là được viết bằng JavaScript. Do đó, chúng ta có thể sử dụng lại những React components và làm business logic với thư viện quản lý state yêu thích của mình.

Mặc dù React Native cung cấp những tính năng giống như web nhưng các bạn phải hiểu rằng đó là hai nền tảng riêng biệt. React Native có những best practices, cách tối ưu và hạn chế riêng của nó.

Ví dụ, chúng ta không cần quá quan tâm đến vấn đề hao pin khi xây dựng ứng dụng web. Bởi vì hầu hết các website đều chạy trên desktop hoặc các thiết bị có pin lớn hoặc cắm sạc trực tiếp.

Còn trên mobile thì khác, các thiết bị khác nhau có thiết kế, cấu tạo khác nhau và hầu hết thời gian chúng sử dụng pin dung lượng nhỏ. Vì vậy, vấn đề pin là vấn đề chúng ta cần phải xem xét khi lập trình ứng dụng trên mobile.



Nói cách khác, bạn tối ưu vấn đề pin ở cả foreground (đang chạy) và background (chạy nền) có thể tạo ra điều khác biệt.

Những thư viện không tối ưu có thể gây hao pin và làm chậm app. Hệ điều hành có thể giới hạn khả năng của ứng dụng

Mặc dù chạy cùng một ngôn ngữ JavaScript như trên trình duyệt web nhưng không có nghĩa là bạn sẽ áp dụng cùng một cách cho React Native trên mobile. Mọi thứ đều có ngoại lệ.

Nếu thư viện phụ thuộc nhiều vào networking, chẳng hạn như nhắn tin thời gian thực hoặc cung cấp khả năng hiển thị đồ họa nâng cao (cấu trúc 3D, diagrams), thì rất có thể bạn nên sử dụng thư viện di động chuyên biệt hơn.

Lý do rất đơn giản, ngay từ đầu những thư viện này đã được phát triển trong môi trường web, mang tính năng cũng như các ràng buộc riêng cho Web. Kết quả là thư viện dùng cho Web sẽ tiêu thụ CPU và RAM nhiều hơn.

Một số hệ điều hành nhất định, chẳng hạn như iOS, được biết là liên tục phân tích tài nguyên mà ứng dụng tiêu thụ để tối ưu hóa tuổi thọ pin. Nếu ứng dụng của bạn được đăng ký để thực hiện các hoạt động nền và các hoạt động này chiếm quá nhiều tài nguyên, thì khoảng thời gian dành cho ứng dụng của bạn có thể được điều chỉnh, làm giảm tần suất cập nhật nền mà bạn đã đăng ký ban đầu.

Giải pháp: sử dụng phiên bản thư viện dành riêng cho từng nền tảng cụ thể

Hãy lấy Firebase làm ví dụ. Firebase là một nền tảng di động của Google cho phép bạn xây dựng ứng dụng của mình nhanh hơn. Nó là một tổ hợp các công cụ và thư viện cung cấp tính năng tức thời (real-time) trong ứng dụng của bạn.

Firebase chứa SDKs cho web và mobile - iOS và Android tương ứng. Mỗi SDK contains hỗ trợ cho Cơ sở dữ liệu thời gian thực.



Nhờ React Native, bạn có thể chạy phiên bản web của nó mà không gặp sự cố gì nghiêm trọng:

```
import database from 'firebase/database';

database()
.ref('/users/123')
.on('value', snapshot => {
  console.log('User data: ', snapshot.val());
});
```

An example reading from Firebase Realtime Database in RN

Tuy nhiên, đây không phải là điều bạn nên làm. Mặc dù ở ví dụ trên không xảy ra vấn đề gì tuy nhiên nó không cung cấp hiệu năng tương tự như thiết bị di động. Bản thân SDK cũng chứa ít tính năng hơn. Tất nhiên, web thì khác và không có lý do gì Firebase.js phải cung cấp hỗ trợ các tính năng dành cho thiết bị di động.

Trong ví dụ cụ thể này, tốt hơn là sử dụng một thư viện Firebase riêng cung cấp một layer mỏng bên trên các native SDKs chuyên biệt. Và nó cũng cung cấp hiệu năng và độ ổn định tương tự như bất kỳ ứng dụng native khác.



Một minh họa cho ví dụ trên:

```
import database from '@react-native-firebase/database';

database()
.ref('/users/123')
.on('value', snapshot => {
  console.log('User data: ', snapshot.val());
});
```

An example reading from Firebase Realtime Database in RN

Như bạn thấy, chẳng có sự khác biệt và chung quy lại vẫn là tạo thành một câu lệnh import khác. Trong trường hợp này, các tác giả thư viện đã làm rất tốt việc sao chép API để giảm sự nhầm lẫn tiềm ẩn khi chuyển đổi qua lại giữa web và thiết bị di động.

Lợi ích: Hỗ trợ nhanh và hiệu năng cao nhất mà không gây hại đến tuổi thọ pin

React Native cung cấp cho bạn quyền kiểm soát và tự do lựa chọn cách bạn muốn xây dựng ứng dụng của mình.

Đối với những thứ đơn giản và có khả năng tái sử dụng cao, bạn có thể chọn lựa sử dụng phiên bản web của thư viện. Điều đó sẽ cho bạn quyền truy cập vào các tính năng tương tự như trong trình duyệt dễ dàng hơn.

Đối với các trường hợp nâng cao, bạn có thể dễ dàng mở rộng React Native với chức năng native và gọi trực tiếp tới các mobile SDKs. Cách giải quyết như vậy làm cho React Native trở nên cực kỳ linh hoạt và phù hợp cho các nghiệp vụ trọng yếu.

Nó cho phép bạn xây dựng các tính năng nhanh hơn trên nhiều nền tảng cùng một lúc mà không ảnh hưởng đến hiệu năng và trải nghiệm người dùng. Một vài frameworks hybrid khác đã thực hiện cân bằng và đánh đổi hợp lý cho việc này.

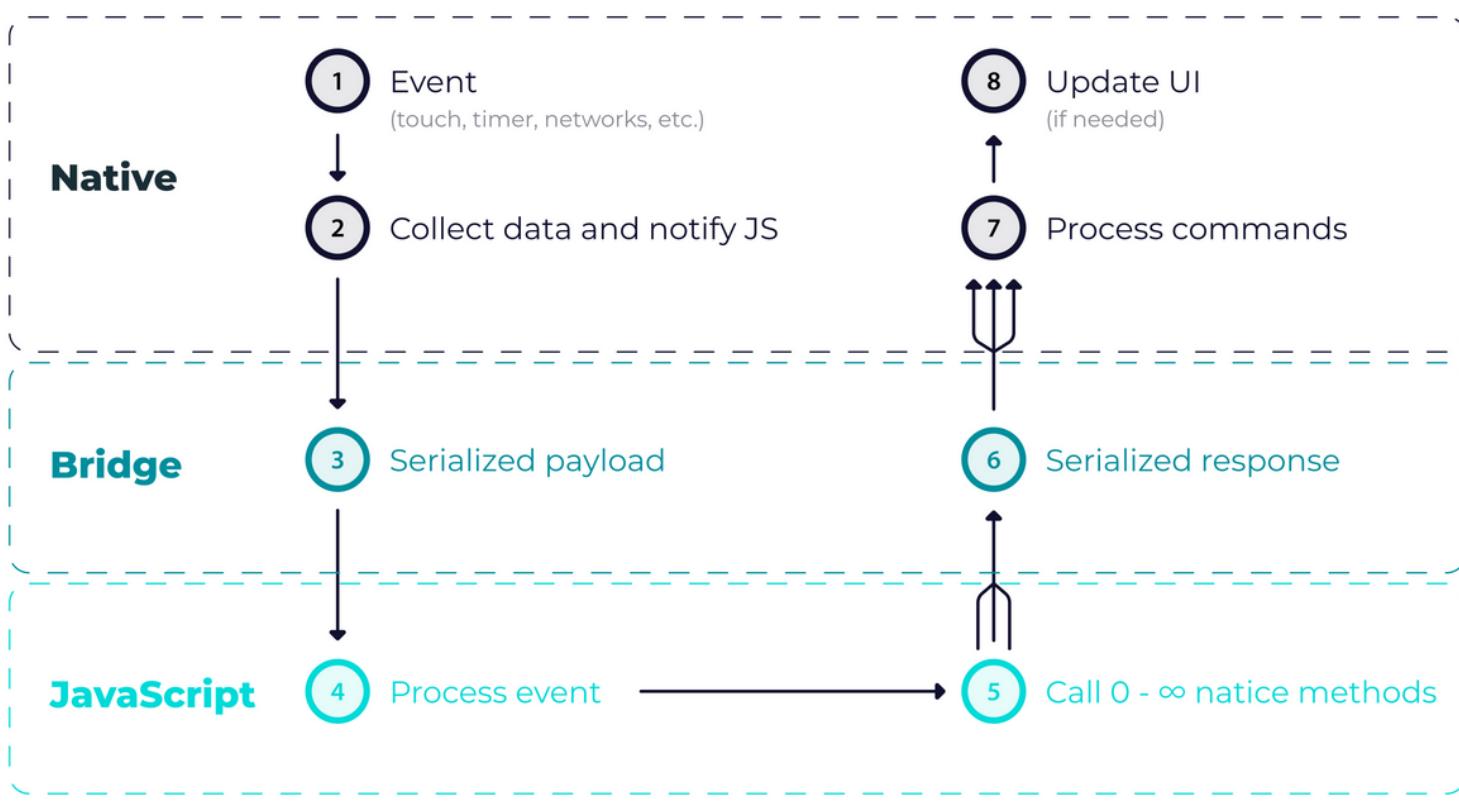
5. Tìm sự cân bằng giữa native và JavaScript

Tìm sự hòa hợp giữa Native và JavaScript để làm việc nhanh và dễ dàng bảo trì app

Vấn đề: Trong lúc làm việc trên native modules, bạn rất dễ nhầm lẫn đâu là native và đâu là code JS.

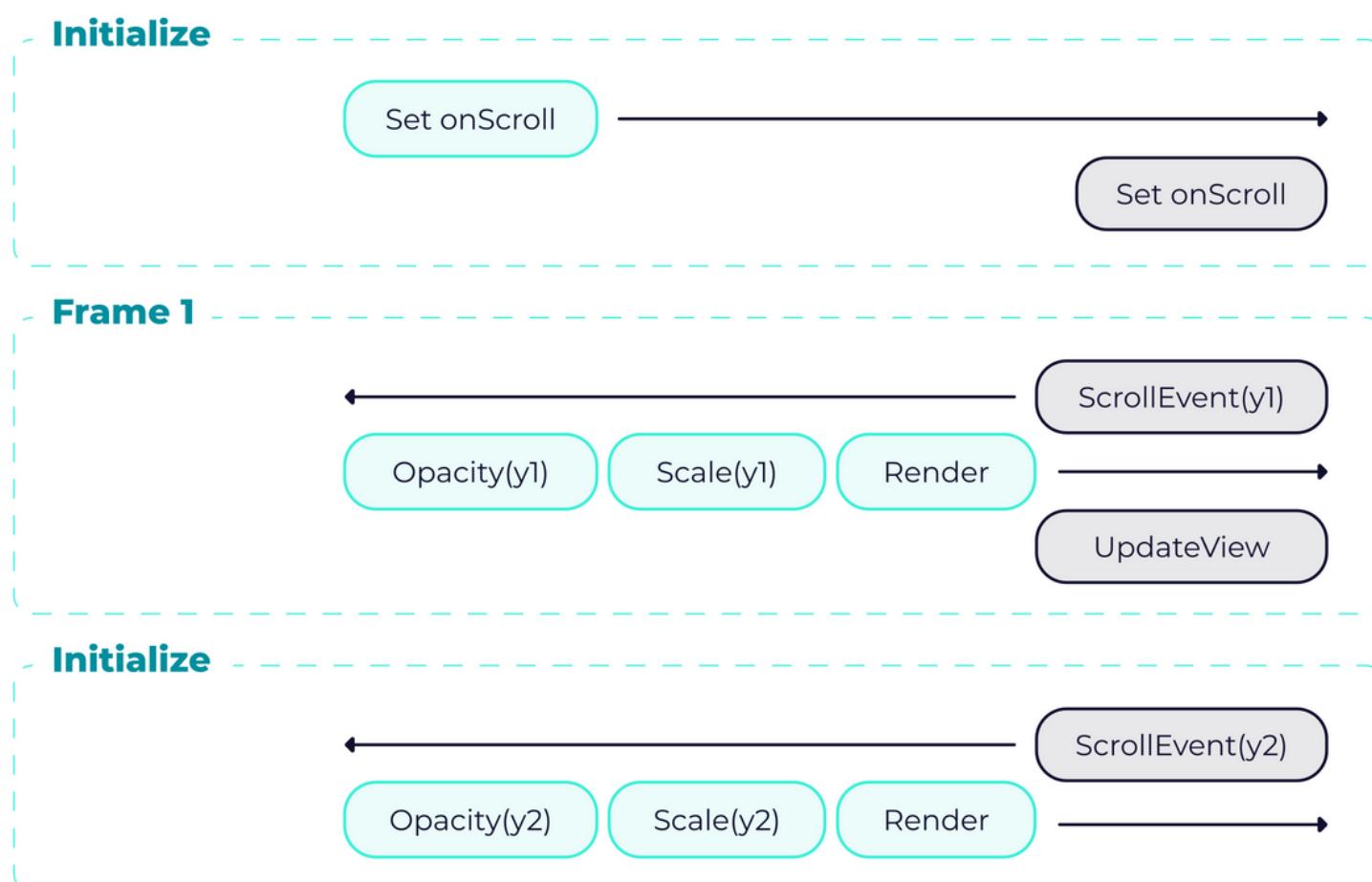
Khi làm việc với React Native, hầu hết thời gian bạn sẽ dùng JavaScript. Tuy nhiên, có những trường hợp bạn cần phải viết một ít mã native. Ví dụ: bạn đang làm việc với SDK của bên thứ 3 và nó chưa hỗ trợ React Native chính thức. Trong trường hợp đó, bạn cần tạo một native module chứa các native methods cơ bản để dùng sang React Native

Tất cả các native methods đều cần các tham số thực để hoạt động. React Native được xây dựng trên nền tảng trừu tượng được gọi là cầu giao tiếp (bridge), nó cung cấp giao tiếp hai chiều giữa JavaScript và thế giới Native. Kết quả là JavaScript có thể thực thi các native APIs và chuyển ngữ cảnh cần thiết để nhận giá trị trả về mong muốn. Bản thân việc giao tiếp đó là bất đồng bộ - có nghĩa là trong khi nơi gọi hàm đang đợi kết quả đến từ phía native, JavaScript vẫn có thể thực hiện một tác vụ khác.



Số lượng lệnh gọi JavaScript đến cầu giao tiếp không thể xác định và có thể thay đổi theo thời gian, tùy thuộc vào số lượng tương tác mà bạn thực hiện trong ứng dụng của mình. Ngoài ra, mỗi lệnh gọi cần có thời gian, vì các đối số JavaScript cần được chuyển hóa thành JSON, đây là định dạng đã thiết lập để có thể hiểu được bởi hai môi trường này.

Ví dụ, khi cầu giao tiếp (bridge) đang bận xử lý dữ liệu, một lệnh khác sẽ bị chặn và chờ. Nếu lệnh đó có liên quan đến gestures và animation thì nó sẽ bị giảm frame (UI của bạn sẽ bị lag và giật).



Một số thư viện nhất định, chẳng hạn như Animated cung cấp các cách giải quyết đặc biệt (trong trường hợp này, hãy sử dụng NativeDriver).

Nó tuân tự hóa animation, chuyển qua native thread hết một lần duy nhất và không vượt qua cầu giao tiếp trong khi animation đang chạy (ngăn không cho nó đột nhiên giảm frame xuống trong khi một loại công việc khác đang diễn ra). Đó là lý do tại sao điều quan trọng là cần phải giữ cho cầu giao tiếp hiệu quả và nhanh chóng (giảm số lần sử dụng cầu giao tiếp).

Càng nhiều lưu lượng đi qua cầu giao tiếp thì càng ít không gian hơn cho những thứ khác

Việc có nhiều lưu lượng truy cập hơn đồng nghĩa với việc có ít không gian hơn cho những thứ quan trọng khác. Những thứ mà React Native có thể muốn giao tiếp native vào thời điểm đó. Do đó, ứng dụng của bạn có thể không phản hồi với các gestures hoặc các tương tác khác trong khi bạn đang thực hiện các lệnh gọi native.

Nếu bạn thấy hiệu năng UI bị suy giảm khi thực hiện một số lệnh gọi native qua cầu giao tiếp hoặc thấy CPU tiêu thụ một cách đáng kể, bạn nên xem lại cách mà bạn đang làm với các thư viện bên ngoài. Rất có thể chúng đang sử dụng cầu giao tiếp quá mức cần thiết.

Hãy dùng code phía JS hợp lý - hãy kiểm tra kiểu dữ liệu cần gọi qua native trước khi dùng nó

Khi xây dựng một native module, bạn nên ủy quyền lệnh gọi ngay lập tức cho phía native và để nó thực hiện phần còn lại. Tuy nhiên, có những trường hợp như các tham số không hợp lệ. Điều đó dẫn đến việc đi vòng không cần thiết qua cầu giao tiếp chỉ để biết rằng chúng ta đã không cung cấp tập hợp các đối số chính xác.

Lấy một module JavaScript đơn giản làm ví dụ, nó sẽ không làm gì khác ngoài việc ủy quyền lệnh gọi đến thẳng module native bên dưới.

```
import { NativeModules } from 'react-native';
const { ToastExample } = NativeModules;

export const show = (message, duration) => {
  ToastExample.show(message, duration)
};
```

Trong trường hợp tham số không chính xác hoặc bị thiếu, native module có khả năng đưa ra một lỗi ngoài dự tính (exception). Phiên bản hiện tại của React Native không cung cấp thông tin tóm tắt để đảm bảo các thông số JavaScript và các thông số mà mã native của bạn cần được đồng bộ hóa. Lệnh gọi của bạn sẽ được tuần tự hóa thành JSON, được chuyển sang phía native và được thực thi.

Thao tác đó sẽ thực hiện mà không gặp bất kỳ sự cố nào, thậm chí chúng ta chưa thông qua danh sách đầy đủ của các đối số cần thiết để nó hoạt động. Lỗi sẽ xuất hiện trong lần xử lý tiếp theo khi phía native xử lý lệnh gọi và nhận được một lỗi từ native module.

Trong trường hợp này, bạn đã lãng phí thời gian để chờ đợi lỗi xảy ra trong khi lẽ ra đã có thể kiểm tra trước đó.

```
import { NativeModules } from 'react-native';
const { ToastExample } = NativeModules;

export const show = (message, duration) => {
  if (typeof message !== 'string' || message.length > 100) {
    throw new Error('Invalid Toast content!')
  }
  if (!Number.isInteger(duration) || duration > 20000) {
    throw new Error('Invalid Toast duration!')
  }
  ToastExample.show(message, duration)
}
```

Using native module with arguments validation

Những điều trên không chỉ gắn liền với chính các native modules. Cần lưu ý rằng mọi component gốc của React Native đều có native code tương đương của nó và các component props được chuyển qua cầu giao tiếp mỗi khi rendering - giống như bạn thực thi native method của mình với các đối số JavaScript.

Để có góc nhìn tốt hơn, hãy xem xét kỹ hơn về cách tạo styling (thông số UI) React Native

```

import * as React from 'react';
import { View } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={{flex: 1, justifyContent: 'center', alignItems: 'center'}}>
        <View style={{
          backgroundColor: 'coral',
          width: 200,
          height: 200
        }}/>
      </View>
    );
  }
}

```

Xem thêm tại: <https://snack.expo.io/7H5S504j3>

Cách dễ nhất để xác định kiểu dáng cho component (styling component) là truyền vào cho nó một đối tượng style. Thật ra bạn sẽ không thấy điều này xuất hiện nhiều. Nó thường được coi là anti-pattern (hay gọi là bad practice), trừ khi bạn đang xử lý các giá trị động, chẳng hạn như thay đổi style của component dựa trên state.

```

import * as React from 'react';
import { View, StyleSheet } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.caontainer}>
        <View style={styles.box} />
      </View>
    );
  }
}

```

```
const styles = StyleSheet.create({
  caontainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  box: {
    backgroundColor: 'coral',
    width: 200,
    height: 200,
  },
});
```

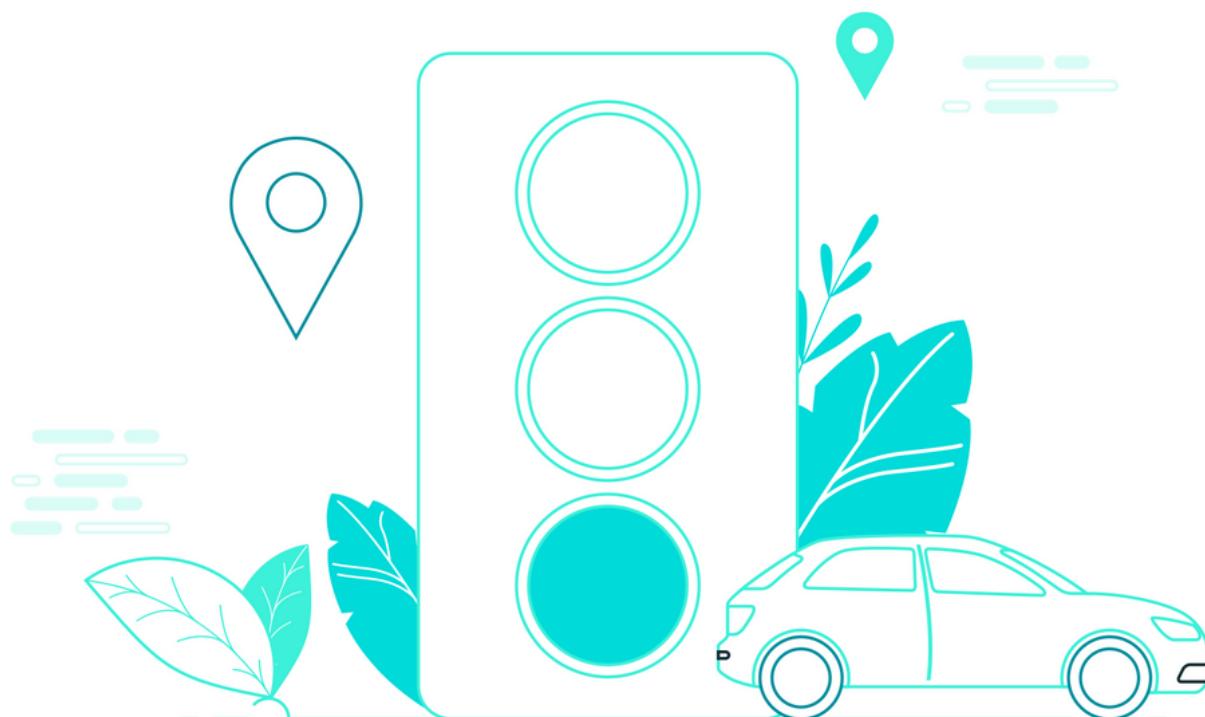
Xem thêm tại: <https://snack.expo.io/GUFPWl8BD>

Hầu hết thời gian, React Native sử dụng StyleSheet API để truyền các kiểu qua cầu giao tiếp. API đó xử lý các style data của bạn và đảm bảo rằng chúng chỉ được chuyển qua cầu giao tiếp một lần thôi. Trong khi thực thi, nó thay thế giá trị của style prop bằng một con số id duy nhất tương ứng với cached style ở phía native.

Như vậy, thay vì gửi một mảng lớn các đối tượng mỗi khi React Native re render UI của nó, cầu giao tiếp giờ đây chỉ nhận một mảng số, điều này dễ dàng hơn nhiều để xử lý và chuyển giao

Lợi ích: Code nhanh hơn và dễ bảo trì hơn

Cho dù bạn đang đối mặt với bất kỳ thách thức hiệu năng nào lúc này, thì đó cũng là cơ hội tốt để bạn triển khai một tập hợp các best practices xoay quanh các native modules. Lợi ích bạn đạt được không chỉ về tốc độ phát triển mà còn là trải nghiệm người dùng.



Chắc chắn, việc duy trì lưu lượng phù hợp qua cầu giao tiếp cuối cùng sẽ góp phần giúp ứng dụng của bạn hoạt động tốt và trơn tru hơn. Như bạn có thể thấy, một số kỹ thuật nhất định được đề cập trong phần này đã và đang được sử dụng tích cực bên trong React Native để mang lại cho bạn một hiệu năng như ý. Biết được những điều đó sẽ giúp bạn tạo ra các ứng dụng hoạt động tốt hơn khi có tải cao.

Tuy nhiên, một lợi ích bổ sung dễ nhận ra được đó chính là việc bảo trì (maintenance)

Việc giữ lại các yếu tố trừu tượng nặng nề và nâng cao, chẳng hạn như xác thực dữ liệu, về phía JavaScript, sẽ dẫn đến một lớp native rất mỏng, không có gì khác ngoài một lớp bao bọc xung quanh một native SDK bên dưới. Nói cách khác, phần native của module sẽ trông giống như một bản sao chép trực tiếp từ tài liệu - rất đơn giản và dễ hiểu.

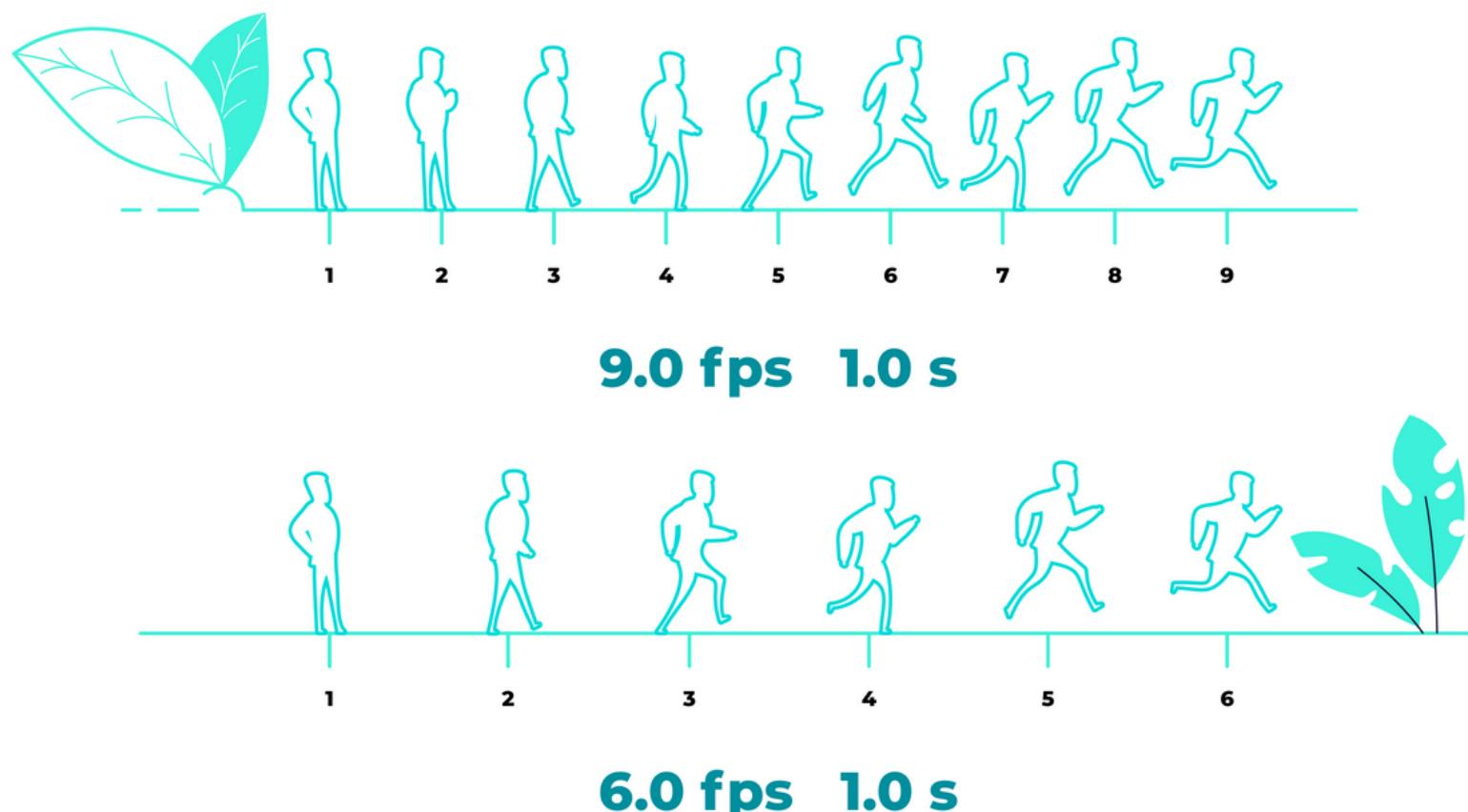
Nắm vững cách tiếp cận này để phát triển các native modules giúp cho nhiều nhà phát triển JavaScript có thể dễ dàng mở rộng ứng dụng của họ với các chức năng bổ sung mà không cần chuyên sâu về Objective-C hay Java.

6. Diễn hoạt ở 60FPS bất kể vì điều gì

Sử dụng các giải pháp từ native để đạt được sự chuyển động mượt mà và các tương tác gesture (vuốt- chạm- giữ) tại 60 FPS.

Vấn đề: Animation theo JavaScript đang chiếm sự di chuyển qua lại của cây cầu trung gian giữa code JS và native và làm chậm ứng dụng.

Người dùng di động hầu hết đã quen với các ứng dụng chạy mượt và được thiết kế đẹp cũng như phản hồi nhanh chóng, trực quan khi thao tác trên màn hình. Đó là một trong những điều không thực sự đúng khi phát triển ứng dụng trên web nhưng lại là điểm khác biệt trên di động. Do đó, những ứng dụng phải sử dụng rất nhiều Animation tại rất nhiều nơi để chạy trong khi những công việc khác đang diễn ra.



Như chúng ta đã biết ở phần trước, số lượng thông tin cần được xử lý và thực hiện thông qua cây cầu này có giới hạn. Hiện tại chưa có hàng đợi ưu tiên nào được tích hợp vào. Nói cách khác, bạn phải thiết kế và cấu trúc ứng dụng của mình bằng cách xử lý cả business logic (tính toán nghiệp vụ) và các animation mà không bị bất cứ gián đoạn nào xảy ra. Đây là sự khác biệt giữa cách mà chúng ta thường hay thực hiện animation. Ví dụ, trên IOS, các API có sẵn cung cấp hiệu năng tuyệt vời và chúng luôn được sắp xếp với độ ưu tiên phù hợp. Tóm lại, chúng ta không cần phải lo lắng quá mức về việc đảm bảo chúng ta luôn đạt được 60 FPS khi chạy ứng dụng.

Với React Native câu chuyện này có chút khác biệt. Nếu bạn không nghĩ trước về các hoạt ảnh từ trên xuống và lựa chọn đúng công cụ để giải quyết thử thách này, sớm muộn gì bạn cũng khiến cho ứng dụng của bạn bị giảm khung hình.

Giật, lag hoặc các hoạt ảnh diễn ra chậm khiến người dùng nghĩ rằng ứng dụng của bạn trông chậm chạp và chưa hoàn thiện.

Trong vô vàn các ứng dụng hiện nay, việc cung cấp một ứng dụng chạy mượt mà và tương tác UI tốt là một trong những cách duy nhất khiến cho khách hàng chọn lựa ứng dụng của bạn.

Nếu ứng dụng của bạn thất bại trong việc cung cấp giao diện hoạt động tốt với các tương tác của người dùng (chẳng hạn như cử chỉ (gestures)), nó không chỉ ảnh hưởng đến khách hàng mới mà còn làm giảm lợi nhuận cho ứng dụng và thiện cảm của người dùng.

Người dùng di động thích các giao diện “follow” theo cách sử dụng của họ, trông đẹp mắt nhất, đảm bảo các hoạt ảnh hoạt động mượt mà là một phần cơ bản nhất trong việc xây dựng trải nghiệm người dùng

Giải pháp: Nếu có thể, hãy sử dụng các animation từ native.

Sử dụng native driver là cách dễ dàng và nhanh chóng nhất trong việc cải thiện hiệu năng các animation của bạn. Tuy nhiên, các style props được sử dụng cùng với native driver có thể bị hạn chế. Bạn có thể sử dụng nó cùng với các thuộc tính không phải là style layout ví dụ như transform hoặc opacity. Nó sẽ không hoạt động với những màu sắc (color), chiều cao (height) và những thứ khác nữa. Nhưng nó cũng đủ để thực hiện hầu hết

các animation trong ứng dụng của bạn, bởi vì bạn thường xuyên muốn hiện/ẩn hoặc thay đổi vị trí của component nào đó.

```
const fadeAnim = useRef(new Animated.Value(0)).current;

const fadeIn = () => {
  Animated.timing(fadeAnim, {
    toValue: 1,
    duration: 1000,
    useNativeDriver: true, // enables native driver
  }).start();
};

// [...]

<Animated.View style={{ opacity: fadeAnim }}/>
```

Enabling native driver for opacity animation

Trong các trường hợp phức tạp hơn, bạn có thể sử dụng thư viện Reanimated. Các API của nó tương thích với các thư viện Animated cơ bản và giới thiệu thêm các chức năng sâu hơn (low-level) để kiểm soát các animation của bạn. Một điều quan trọng hơn nữa là, nó cung cấp khả năng có thể dùng animation cho style props cùng với native driver. Vì vậy, chiều cao hoặc màu sắc không là còn vấn đề nữa. Tuy nhiên, transform hoặc opacity các animation sẽ nhanh hơn một chút vì chúng được tăng tốc bởi GPU. Người dùng thường sẽ không thể phát hiện ra điểm khác biệt.

Các animation tương tác

Điều tuyệt vời nhất khi sử dụng với các animation đó là có thể tương tác được với các animation đó. Đối với khách hàng của bạn, đây là phần thú vị nhất trong giao diện. Nó tạo thiện cảm và làm app của chúng ta trông có vẻ rất mượt và phản hồi nhanh. React Native vô cùng hạn chế khi kết hợp giữa tương tác với native driven animation. Bạn có thể sử dụng sự kiện scroll của ScrollView để xây dựng những đoạn tiêu đề có thể đóng mở được (collapsible header) rất mượt mà.

Đối với các trường hợp phức tạp hơn, chúng ta có một thư viện tuyệt vời để giải quyết vấn đề trên - React Native Gesture Handler - nó cho phép chúng ta xử lý các tương tác khác nhau một cách tự nhiên hơn và interpolate (chia nhiều giai đoạn) các animation đó. Bạn có thể xây dựng một cử chỉ vượt đơn giản bằng cách kết hợp các Animated lại với nhau. Tuy nhiên, nó vẫn yêu cầu các callback về JS, nhưng có một giải pháp cho vấn đề này.

Bộ đôi công cụ mạnh mẽ nhất cho các animation tương tác đó là kết hợp Gesture Handler cùng với Reanimated. Chúng được thiết kế để kết hợp cùng với nhau và cung cấp khả năng xây dựng các tương tác animation phức tạp vì nó được tính toán kỹ càng ở tầng native. Giới hạn duy nhất tồn tại là khả năng/trí tưởng tượng của bạn.

```
import React, { Component } from 'react';
import { StyleSheet, View } from 'react-native';
import { PanGestureHandler, State } from 'react-native-gesture-handler';
import Animated from 'react-native-reanimated';
import runSpring from './runSpring';

const {
  set,
  cond,
  eq,
  add,
  multiply,
  lessThan,
  spring,
  startClock,
  stopClock,
  clockRunning,
  sub,
  defined,
  Value,
  Clock,
  event,
  SpringUtils,
} = Animated;
```

```

class Snappable extends Component {
  constructor(props) {
    super(props);

    const TOSS_SEC = 0.2;

    const dragX = new Value(0);
    const state = new Value(-1);
    const dragVX = new Value(0);

    this._onGestureEvent = event([
      { nativeEvent: { translationX: dragX, velocityX: dragVX, state: state } },
    ]);

    const transX = new Value();
    const prevDragX = new Value(0);

    const clock = new Clock();

    const snapPoint = cond(
      lessThan(add(transX, multiply(TOSS_SEC, dragVX)), 0),
      -100,
      100
    );
  }

  this._transX = cond(
    eq(state, State.ACTIVE),
    [
      stopClock(clock),
      set(transX, add(transX, sub(dragX, prevDragX))),
      set(prevDragX, dragX),
      transX,
    ],
    [
      set(prevDragX, 0),
    ]
  );
}

```

```

    set(
      transX,
      cond(defined(transX), runSpring(clock, transX, dragVX, snapPoint), 0)
    ),
  ]
);

render() {
  const { children, ...rest } = this.props;
  return (
    <PanGestureHandler
      {...rest}
      maxPointers={1}
      minDist={10}
      onGestureEvent={this._onGestureEvent}
      onHandlerStateChange={this._onGestureEvent}>
      <Animated.View style={{ transform: [{ translateX: this._transX }] }}>
        {children}
      </Animated.View>
    </PanGestureHandler>
  );
}

export default class Example extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Snappable>
          <View style={styles.box} />
        </Snappable>
      </View>
    );
  }
}

```

```

const BOX_SIZE = 100;

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  box: {
    width: BOX_SIZE,
    height: BOX_SIZE,
    borderColor: '#F5FCFF',
    alignSelf: 'center',
    backgroundColor: 'plum',
    margin: BOX_SIZE / 2,
  },
});

```

Read more: <https://snack.expo.io/EMOKZfwJd>

Việc xử lý các cử chỉ (gesture) ở cấp thấp có thể không dễ, nhưng may mắn thay, chúng ta có rất nhiều thư viện bên thứ ba đã được giới thiệu trước đó và hỗ trợ CallbackNodes. CallbackNodes không khác gì AnimatedValue cả, nhưng nó bắt nguồn từ một hành vi tương tác cụ thể nào đó. Phạm vi giá trị của nó thường nằm trong khoảng từ 0 đến 1. Bạn có thể interpolate các giá trị diễn hoạt trên màn hình. Một số thư viện tuyệt vời sau đây giúp bạn lấy được CallbackNode đó là reanimated-bottom-sheet & react-native-tab-view.

```

import * as React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import Animated from 'react-native-reanimated';
import BottomSheet from 'reanimated-bottom-sheet';
import Lorem from './Lorem';

const { block, set, greaterThan, lessThan, Value, cond, sub, interpolate } =

```

```

Animated;

export default class Example extends React.Component {
gestureCallbackNode = new Value(0);

contentPos = this.gestureCallbackNode;

renderHeader = name => (
<View
style={{
width: '100',
backgroundColor: 'lightgrey',
height: 40,
borderWidth: 2,
}}>
<Text style={{textAlign: 'center', fontSize: 20, padding: 5}}>Drag me</Text>
</View>
);

renderInner = () => (
<View style={{backgroundColor: 'lightblue'}}>
<Animated.View
style={{
opacity: interpolate(this.contentPos, {inputRange:[0,1],
outputRange:[1,0]}),
transform: [
translateY : interpolate(this.contentPos, {inputRange:[0,1],
outputRange:[0,100]}),
]
}}>
<Lorem />
<Lorem />
</Animated.View>
</View>
);

```

```

render() {
  return (
    <View style={styles.container}>
      <BottomSheet
        callbackNode={this.gestureCallbackNode}
        snapPoints={[50, 400]}
        initialSnap={1}
        renderHeader={this.renderHeader}
        renderContent={this.renderInner}
      />
    </View>
  );
}

const IMAGE_SIZE = 200;

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
});

```

Read more: <https://snack.expo.io/KFpkVKYB9>

Đặt độ ưu tiên thấp cho các tính toán bên JS

Không nhất thiết lúc nào cũng cần kiểm soát hoàn toàn cách các animation được thực hiện. Ví dụ, React Navigation sử dụng kết hợp giữa React Native Gesture Handler và Animated nhưng vẫn cần sử dụng JavaScript để kiểm soát runtime của animation. Kết quả là animation của bạn có thể bắt đầu giật, lag nếu màn hình bạn sắp chuyển qua đang tải 1 bộ UI lớn. Thật may mắn, bạn có thể hoãn việc thực thi các hành động này bằng cách sử dụng InteractionManager.

```

import React, { useState, useRef } from 'react';
import {
  Text,
  View,
  StyleSheet,
  Button,
  Animated,
  InteractionManager,
  Platform,
} from 'react-native';
import Constants from 'expo-constants';

const ExpensiveTaskStates = {
  notStarted: 'not started',
  scheduled: 'scheduled',
  done: 'done',
};

export default function App() {
  const animationValue = useRef(new Animated.Value(100));
  const [animationState, setAnimationState] = useState(false);
  const [expensiveTaskState, setExpensiveTaskState] = useState(
    ExpensiveTaskStates.notStarted
);

  const startAnimationAndScheduleExpensiveTask = () => {
    Animated.timing(animationValue.current, {
      duration: 2000,
      toValue: animationState ? 100 : 300,
      useNativeDriver: false,
    }).start(() => {
      setAnimationState(!animationState);
    });
    setExpensiveTaskState(ExpensiveTaskStates.scheduled);
    InteractionManager.runAfterInteractions(() => {

```

```

        setExpensiveTaskState(ExpensiveTaskStates.done);
    });

};

return (
    <View style={styles.container}>
        {Platform.OS === 'web' ? (
            <Text style={{ textAlign: 'center' }}>
                !InteractionManager works only on native platforms. Open example on iOS
                or Android!
            </Text>
        ) : (
            <>
                <Button
                    title="Start animation and schedule expensive task"
                    onPress={startAnimationAndScheduleExpensiveTask}
                />
                <Animated.View
                    style={[styles.box, { width: animationValue.current }]}>
                    <Text>Animated box</Text>
                </Animated.View>
                <Text style={styles.paragraph}>
                    Expensive task status:{' '}
                    <Text style={{ fontWeight: 'bold' }}>{expensiveTaskState}</Text>
                </Text>
            </>
        )}
    </View>
);

}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',

```

```
paddingTop: Constants.statusBarHeight,  
padding: 8,  
,  
paragraph: {  
margin: 24,  
fontSize: 18,  
textAlign: 'center',  
,  
box: {  
backgroundColor: 'coral',  
marginVertical: 20,  
height: 50,  
,  
});
```

Read more: <https://snack.expo.io/Wv8u!mKw>

Module tiện dụng này của React Native cho phép chúng ta thực thi bất kỳ đoạn code nào sau khi tất cả các animation đã kết thúc. Trong thực tế, bạn có thể hiển thị đặt chỗ trước (placeholder), đợi đến khi các animation kết thúc và sau đó mới thực sự render giao diện mà mình mong muốn. Điều đó sẽ giúp cho JavaScript animations chạy mượt hơn và bỏ đi sự can thiệp không cần thiết từ các tính toán khác. Việc này mang lại trải nghiệm tốt nhất cho user.

Lợi ích: Animation sẽ mượt mà và các thao tác tương tác mượt mà ở 60 FPS

Không có một cách đúng đắn nào cho việc thực hiện các animation trong React Native. Hệ sinh thái của nó đầy đủ các thư viện và hướng tiếp cận khác nhau để xử lý tương tác. Trong chương này chỉ là những lời khuyên, bạn không nên coi quá trọng sự mượt mà trong giao diện người dùng.

Điều quan trọng hơn là việc thực hiện hoá những thứ trong đầu bạn cùng với các tương tác trong ứng dụng và lựa chọn những cách đúng đắn nhất để giải quyết nó. Có rất nhiều trường hợp các animation trên JavaScript hoạt động khá ổn.Thêm vào đó, đồng thời ta dùng thêm animation dưới native (hoặc toàn bộ view native) sẽ giúp ứng dụng bạn mượt mà.

Với cách tiếp cận này, ứng dụng bạn sẽ mượt mà và linh động hơn. Nó không chỉ làm hài lòng người dùng của bạn mà còn giúp bạn dễ gỡ lỗi (debug) hơn và vui vẻ trong quá trình phát triển.



Tập 2

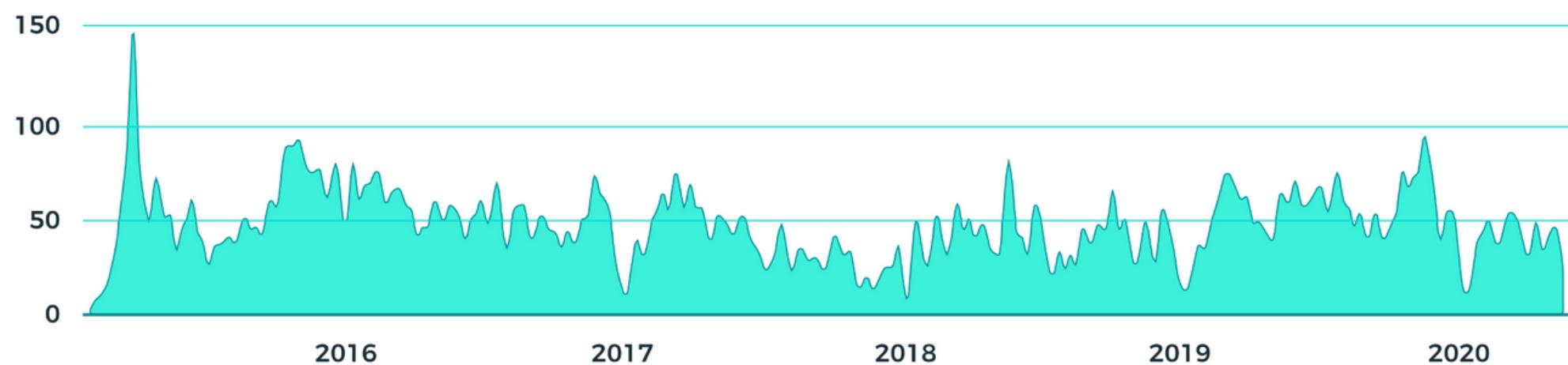
Cải thiện hiệu suất bằng cách sử dụng
các tính năng React Native tân tiến nhất.

1. Luôn sử dụng phiên bản React Native mới nhất để có các tính năng mới

Nâng cấp ứng dụng của bạn lên phiên bản mới nhất để có nhiều tính năng hơn và hỗ trợ tốt hơn

Vấn đề: Bạn đang sử dụng phiên bản React Native cũ và không được hỗ trợ, đồng thời không có các tính năng và cải tiến mới.

Duy trì ứng dụng của bạn luôn cập nhật với các framework mà bạn sử dụng là rất quan trọng. Bằng cách đó, bạn đăng ký nhận các tính năng mới nhất, cải tiến hiệu suất và các bản sửa lỗi bảo mật mới nhất. Hệ sinh thái JavaScript đặc biệt thú vị ở khía cạnh này, vì nó phát triển rất nhanh. Nếu bạn không cập nhật ứng dụng của mình thường xuyên, rất có thể code của bạn sẽ bị lạc hậu khiến cho việc nâng cấp nó sẽ trở nên khó khăn và rủi ro.



React Native là một trong những thư viện đang phát triển rất nhanh chóng.

Mỗi ngày, các nhà phát triển từ khắp nơi trên thế giới giới thiệu các tính năng mới, các bản sửa lỗi quan trọng và các bản vá bảo mật. Trung bình, mỗi bản phát hành bao gồm khoảng 500 commits. Hầu hết các thay đổi phổ biến trong cộng đồng bao gồm các tính năng như **Fast Refresh** hoặc

Autolinking - cả hai đều được chúng tôi mô tả trong các phần sau.

Trong hệ sinh thái React Native, thông thường các thư viện không tương thích ngược. Các tính năng mới thường sử dụng các tính năng bổ sung không có sẵn trong các phiên bản trước của framework. Có nghĩa là nếu ứng dụng của bạn vận hành trên phiên bản React Native cũ hơn, cuối cùng bạn sẽ bắt đầu bỏ lỡ những cải tiến mới nhất.

@react-native-community/cli	react-native
^5.0.0 (next)	master
^4.0.0 (master)	^0.62.0
^3.0.0	^0.61.0
^2.0.0	^0.60.0
^1.0.0	^0.59.0

Đó là lý do tại sao, duy trì các bản nâng cấp React Native mới nhất có thể xem như là cách duy nhất.

Thật không may, có một số việc nghiêm trọng liên quan đến việc nâng cấp code React Native của bạn với mỗi bản phát hành mới. Số lượng của nó sẽ phụ thuộc vào số lượng các thay đổi cơ bản đối với các chức năng gốc và các phần cốt lõi. Hầu hết thời gian, bạn phải phân tích và so sánh cẩn thận dự án của mình với phiên bản mới nhất và tự mình thực hiện các điều chỉnh. Nhiệm vụ này sẽ dễ dàng hơn nếu bạn đã cảm thấy thoải mái với việc di chuyển xung quanh môi trường gốc. Nhưng đối với hầu hết chúng ta, nó có thể nhiều hơn một chút thách thức.

Ví dụ, có thể các module và thành phần bạn đã sử dụng trong code của mình không còn là một phần của lõi react-native nữa.

Đó là do những thay đổi được Facebook đưa ra trong suốt quá trình được gọi là the lean core. Mục tiêu của sự nỗ lực là để:

- Làm cho gói react-native nhỏ hơn, linh hoạt hơn và dễ bảo trì hơn bằng cách trích xuất một số phần của lõi và chuyển chúng vào kho lưu trữ cộng đồng react-native,
- Chuyển giao việc bảo trì các module đã trích xuất cho cộng đồng.

Quá trình này đã thúc đẩy sự phát triển của các module cụ thể và làm cho toàn bộ hệ sinh thái được tổ chức tốt hơn. Nhưng nó cũng có một số ảnh hưởng tiêu cực đến việc nâng cấp react-native. Bây giờ, bạn phải cài đặt các packages được giải nén như một phần dependency bổ sung và cho đến khi bạn làm điều đó, ứng dụng của bạn sẽ không biên dịch hoặc gặp sự cố trong thời gian chạy. Tuy nhiên, từ quan điểm của nhà phát triển, việc di chuyển sang các package cộng đồng thường không có gì khác hơn là giới thiệu dependency mới và viết lại các lần nhập.

Một vấn đề quan trọng khác là sự hỗ trợ của các bên thứ ba. Code của bạn thường dựa vào các thư viện bên ngoài và có nguy cơ là chúng cũng có thể không tương thích với phiên bản React Native mới nhất.

Có ít nhất hai cách để giải quyết vấn đề này:

- Chờ người bảo trì dự án thực hiện các điều chỉnh cần thiết trước khi bạn nâng cấp,
- Tìm kiếm các giải pháp thay thế hoặc tự vá các module - bằng cách sử dụng một tiện ích tiện dụng được gọi là patch-package hoặc tạo một nhánh tạm thời với các bản sửa lỗi cần thiết.

Vận hành trên phiên bản cũ có nghĩa là đang shipping với các vấn đề có thể làm nản lòng người dùng của bạn

Nếu bạn đang vận hành trên một phiên bản cũ hơn, có khả năng là bạn đang bị tụt hậu so với đối thủ của bạn - những người sử dụng các phiên bản mới nhất của framework.

Số lượng các bản sửa lỗi, cải thiện và cải tiến trong framework React Native thực sự ấn tượng. Nếu bạn đang chơi trò chơi bắt kịp, bạn chọn không nhận nhiều bản cập nhật sẽ giúp cuộc sống của bạn dễ dàng hơn rất nhiều. Khối lượng công việc và chi phí liên quan đến việc nâng cấp thường xuyên luôn được bù đắp bằng các cải tiến DX (trải nghiệm nhà phát triển) ngay lập tức.

Trong phần này, chúng tôi trình bày một số phương pháp đã được thiết lập tốt để dễ dàng nâng cấp React Native lên phiên bản mới hơn.

Giải pháp: Nâng cấp lên phiên bản React Native mới nhất (chúng tôi sẽ hướng dẫn bạn cách thực hiện)

Nâng cấp React Native có thể không phải là điều dễ dàng nhất trên thế giới. Nhưng có những công cụ làm cho quá trình này trở nên đơn giản hơn nhiều và loại bỏ hầu hết các vấn đề. Khối lượng công việc thực tế sẽ phụ thuộc vào số lượng thay đổi và phiên bản cơ sở của bạn. Tuy nhiên, các bước được trình bày trong phần này có thể được áp dụng cho mọi bản nâng cấp, bất kể trạng thái ứng dụng của bạn là gì.

Chuẩn bị cho việc nâng cấp

[React Native Upgrade Helper](#) là một nơi tốt để bắt đầu. Ở cấp độ cao, nó cung cấp cho bạn cái nhìn tổng quan về những thay đổi đã xảy ra với React Native kể từ lần cuối cùng bạn nâng cấp phiên bản cục bộ của mình.

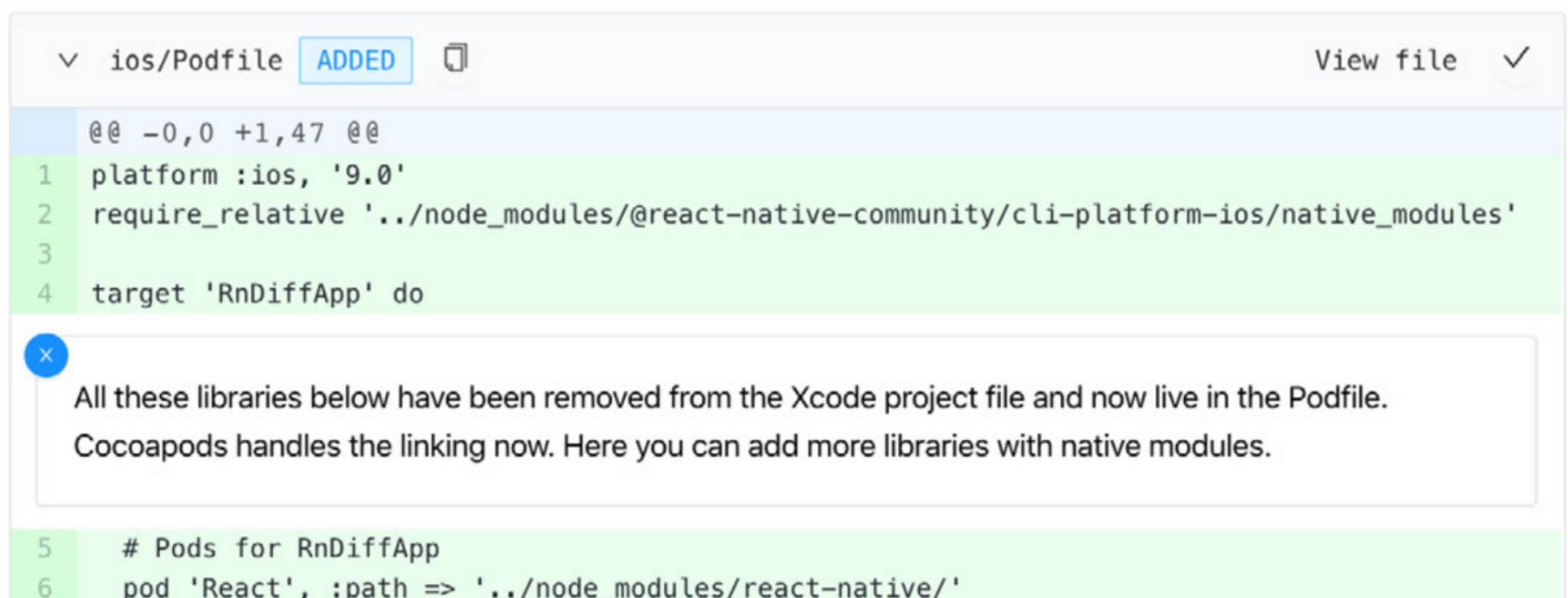
The screenshot shows the React Native Upgrade Helper interface. At the top, it asks for the current version (0.59.10) and the target version (0.60.0). A large blue button says "Show me how to upgrade!". Below this is a diff viewer comparing "package.json MODIFIED" and ".eslintrc.js ADDED". The package.json diff highlights changes from line 6 to 22, showing updated dependencies like react-native@0.60.0 and babel dependencies. The .eslintrc.js file is a new addition with one line of configuration.

```
diff --git a/package.json b/package.json
--- a/package.json
+++ b/package.json
@@ -3,20 +3,23 @@
 3   "version": "0.0.1",
 4   "private": true,
 5   "scripts": {
 6     "start": "node node_modules/react-native/local-cli/cli.js start",
 7     "test": "jest"
 8   },
 9   "dependencies": {
10     "react": "16.8.3",
11     "react-native": "0.59.10"
12   },
13   "devDependencies": {
14     "@babel/core": "^7.4.5",
15     "@babel/runtime": "^7.4.5",
16     "babel-jest": "^24.8.0",
17     "jest": "^24.8.0",
18     "metro-react-native-babel-preset": "^0.54.1",
19     "react-test-renderer": "16.8.3"
20   },
21   "jest": {
22     "preset": "react-native"
23   }
24 }
25 }

diff --git a/.eslintrc.js b/.eslintrc.js
--- a/.eslintrc.js
+++ b/.eslintrc.js
@@ -0,0 +1,1 @@
+module.exports = {
+  extends: ['react-native-community/eslint-config'],
+  rules: {}
+}
```

Sự khác nhau giữa 'package.json' giữa React Native 0.59 và React Native 0.60

Để làm như vậy, trình trợ giúp so sánh các dự án React Native đơn thuần được tạo bằng cách chạy `npx react-native init` với phiên bản của bạn và phiên bản bạn đang nâng cấp lên. Tiếp theo, nó cho thấy sự khác biệt giữa các dự án, giúp bạn biết mọi sửa đổi nhỏ diễn ra trong thời gian chờ đợi. Một số thay đổi có thể được chú thích thêm bằng một thông tin đặc biệt sẽ cung cấp thêm ngữ cảnh về lý do tại sao điều gì đó đã xảy ra.



The screenshot shows the React Native Upgrade Helper interface. The top navigation bar has tabs for 'ios/Podfile' (selected), 'AndroidManifest.xml', and 'Info.plist'. The status bar indicates 'ADDED' for the Podfile. On the right, there are 'View file' and a checkmark icon. The main area displays the following code:

```
@@ -0,0 +1,47 @@
1 platform :ios, '9.0'
2 require_relative '../node_modules/@react-native-community/cli-platform-ios/native_modules'
3
4 target 'RnDiffApp' do
5   # Pods for RnDiffApp
6   pod 'React', :path => '../node_modules/react-native/'
```

A note below the code states: "All these libraries below have been removed from the Xcode project file and now live in the Podfile. Cocoapods handles the linking now. Here you can add more libraries with native modules." There is also a small 'X' icon next to the note.

Giải thích bổ sung về những thay đổi thú vị đối với tệp người dùng

Có cái nhìn tổng quan hơn về những thay đổi sẽ giúp bạn tiến nhanh hơn và hành động tự tin hơn.

Lưu ý: Việc có thêm ngữ cảnh thực sự quan trọng vì không có sự tự động hóa nào khi nâng cấp - bạn sẽ phải tự áp dụng các thay đổi.

React Native Upgrade Helper cũng đề xuất nội dung hữu ích để đọc trong khi nâng cấp. Điều đó trong hầu hết các trường hợp bao gồm một bài đăng blog chuyên dụng được xuất bản trên blog React Native cũng như bản thay đổi thô.

▶ Useful content for upgrading

React Native 0.60 includes Cocoapods integration by default, AndroidX support, auto-linking libraries, a brand new Start screen and more.

1. [Official blog post about the major changes on React Native 0.60](#)
2. [\[External\] Tutorial on upgrading to React Native 0.60](#)
3. [React Native 0.60 changelog](#)

Check out [Upgrade Support](#) if you are experiencing issues related to React Native during the upgrading process.

Keep in mind that `RnDiffApp` and `rndiffapp` are placeholders. When upgrading, you should replace them with your actual project's name. You can also provide your app name by clicking the settings icon on the top right.

Nội dung hữu ích để đọc khi nâng cấp lên React Native 0.60

Chúng tôi khuyên bạn nên đọc các tài nguyên được đề xuất để nắm rõ hơn về bản phát hành sắp tới và tìm hiểu về những điểm nổi bật của nó.

Nhờ đó, bạn không chỉ nhận thức được những thay đổi mà còn hiểu được lý do đằng sau chúng. Và bạn sẽ sẵn sàng bắt đầu dự án của mình và bắt tay vào thực hiện.

Áp dụng các thay đổi JavaScript

Quá trình nâng cấp phần JavaScript của React Native cũng tương tự như nâng cấp các khung JavaScript khác. Khuyến nghị của chúng tôi ở đây là thực hiện nâng cấp từng bước - cập nhật từng thư viện một.

Theo ý kiến của chúng tôi, cách tiếp cận này tốt hơn là nâng cấp mọi thứ cùng một lúc vì nó mang lại cho bạn nhiều quyền kiểm soát hơn và giúp bắt các lỗi quy dễ dàng hơn nhiều.

Bước đầu tiên là chuyển các phụ thuộc React và React Native vào các phiên bản mong muốn và thực hiện các thay đổi cần thiết (bao gồm cả các thay đổi phá vỡ). Để làm như vậy, bạn có thể tra cứu các đề xuất do React Native Upgrade Helper cung cấp và áp dụng chúng theo cách thủ công. Sau khi hoàn tất, hãy đảm bảo cài đặt lại `node_modules` của bạn.

Lưu ý: Khi thực hiện nâng cấp, bạn có thể thấy nhiều thay đổi đến từ các tệp dự án iOS (mọi thứ bên trong .xcodeproj, bao gồm cả .pbxproj). Đây là các tệp được tạo bởi Xcode khi bạn làm việc với phần iOS của ứng dụng React Native.

Thay vì sửa đổi tệp nguồn, tốt hơn nên thực hiện các thay đổi thông qua giao diện người dùng Xcode. Đây là trường hợp nâng cấp lên React Native 0.60 và các hoạt động thích hợp đã được mô tả trong số này.

Cuối cùng, bạn nên chạy thử ứng dụng. Nếu mọi thứ đang hoạt động hoàn hảo, việc nâng cấp diễn ra suôn sẻ thì bạn có thể gọi đó là một ngày tuyệt vời! Tuy nhiên, có một lưu ý quan trọng hơn - bây giờ bạn nên kiểm tra xem có phiên bản mới hơn của các dependency khác mà bạn sử dụng hay không! Họ có thể shipping các cải tiến hiệu suất quan trọng.

Thật không may, cũng có một kịch bản khác bi quan hơn một chút. Ứng dụng của bạn có thể hoàn toàn không được xây dựng hoặc có thể gặp sự cố ngay lập tức với màn hình màu đỏ. Trong trường hợp đó, rất có thể một số dependency bên thứ ba của bạn không hoạt động bình thường và bạn cần làm cho chúng tương thích với phiên bản React Native của mình.

Lưu ý: Nếu bạn gặp sự cố với các bản nâng cấp của mình, bạn có thể kiểm tra dự án [Upgrade Support](#). Đây là một kho lưu trữ nơi các nhà phát triển chia sẻ kinh nghiệm của họ và giúp nhau giải quyết một số hoạt động khó khăn nhất liên quan đến nâng cấp.

Nâng cấp thư viện bên thứ ba

Trong hầu hết các trường hợp, đó là phần dependency React Native của bạn mà bạn nên xem xét đầu tiên. Không giống như các gói JavaScript / React thông thường, chúng thường phụ thuộc vào các hệ thống xây dựng native và các API React Native nâng cao hơn. Điều này khiến họ gặp phải các lỗi tiềm ẩn khi framework hoàn thiện thành API ổn định hơn.

Nếu lỗi xảy ra trong thời gian xây dựng, chỉ cần chuyển phần dependency lên phiên bản mới nhất của nó sẽ làm cho nó hoạt động.

Sau khi ứng dụng của bạn được tạo, bạn đã sẵn sàng kiểm tra bảng thay đổi và làm quen với các thay đổi JavaScript đã xảy ra với API công khai.

Bước này rất dễ bị bỏ qua và thường là kết quả của runtime exceptions. Sử dụng Flow hoặc TypeScript sẽ đảm bảo rằng các thay đổi đã được áp dụng đúng cách.

Như bạn có thể thấy, không có trò ảo thuật nào có thể sửa tất cả các lỗi và nâng cấp các dependency một cách tự động. Đây hầu hết là một công việc thủ công phải được thực hiện với sự kiên nhẫn và chuyên tâm. Nó cũng yêu cầu nhiều thử nghiệm để đảm bảo rằng bạn không làm hỏng bất kỳ tính năng nào trong suốt quá trình.

Lợi ích: Bạn đang sử dụng phiên bản mới nhất có nhiều tính năng hơn và hỗ trợ tốt hơn

Nâng cấp phiên bản React Native mới nhất sẽ không khác với việc cập nhật các framework và thư viện khác của bạn. Ngoài những cải tiến quan trọng về hiệu suất và bảo mật, các bản phát hành React Native mới cũng giải quyết những thay đổi cơ bản mới nhất đối với iOS và Android.

Đây là một ví dụ: Năm ngoái, [Google đã thông báo rằng tất cả các ứng dụng Android được gửi lên Google Play sau ngày 1 tháng 8 năm 2019 phải là 64-bit](#). Để tiếp tục phát triển ứng dụng của mình và cung cấp các tính năng mới, bạn phải nâng cấp lên React Native 0.59 và thực hiện các điều chỉnh cần thiết.

Những nâng cấp như thế này thực sự rất quan trọng để giữ cho người dùng của bạn hài lòng. Sau tất cả, họ sẽ thất vọng nếu ứng dụng bắt đầu gặp sự cố với phiên bản hệ điều hành mới hơn hoặc biến mất khỏi App Store. Có thể có một số khối lượng công việc bổ sung liên quan đến mỗi bản phát hành, nhưng việc cập nhật sẽ mang lại lợi ích cho người dùng hạnh phúc hơn, ứng dụng ổn định hơn và trải nghiệm phát triển tốt hơn.

2. Cách gỡ lỗi nhanh hơn và tốt hơn với Flipper

Thiết lập vòng phản hồi tốt hơn bằng cách triển khai Flipper và vui hơn khi làm việc trên ứng dụng của bạn.

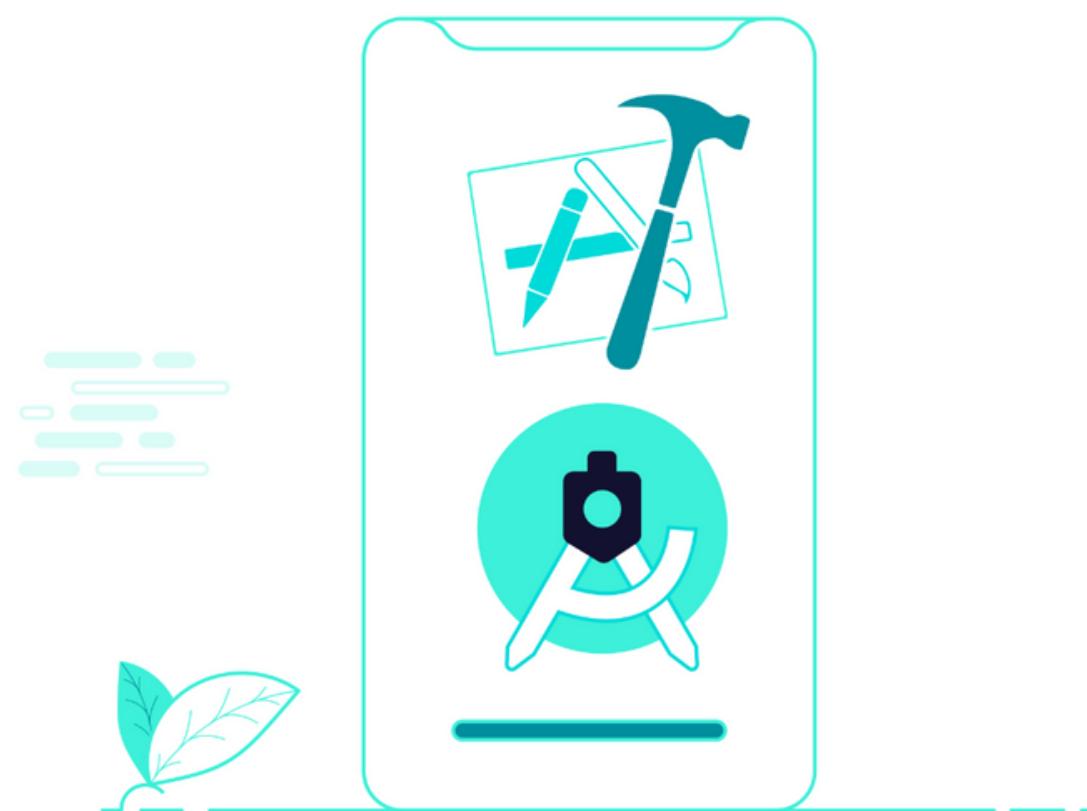
Sự cố: Bạn đang sử dụng Chrome Debugger hoặc một số cách hack khác để gỡ lỗi và lập hồ sơ ứng dụng React Native của bạn

Gỡ lỗi là một trong những phần khó khăn hơn trong công việc hàng ngày của mọi nhà phát triển. Việc giới thiệu một tính năng mới tương đối dễ dàng khi mọi thứ dường như hoạt động, nhưng việc phát hiện ra điều gì sai có thể khiến bạn rất chịu. Chúng tôi thường cố gắng sửa lỗi càng sớm càng tốt, đặc biệt là khi chúng rất nghiêm trọng và làm cho một ứng dụng không hoạt động. Thời gian là một yếu tố quan trọng trong quá trình đó và chúng ta thường phải nhanh nhẹn để nhanh chóng giải quyết các vấn đề.



Tuy nhiên, gỡ lỗi React Native không đơn giản lắm, vì vấn đề bạn đang cố gắng giải quyết có thể xảy ra ở các cấp độ khác nhau. Cụ thể, nó có thể do:

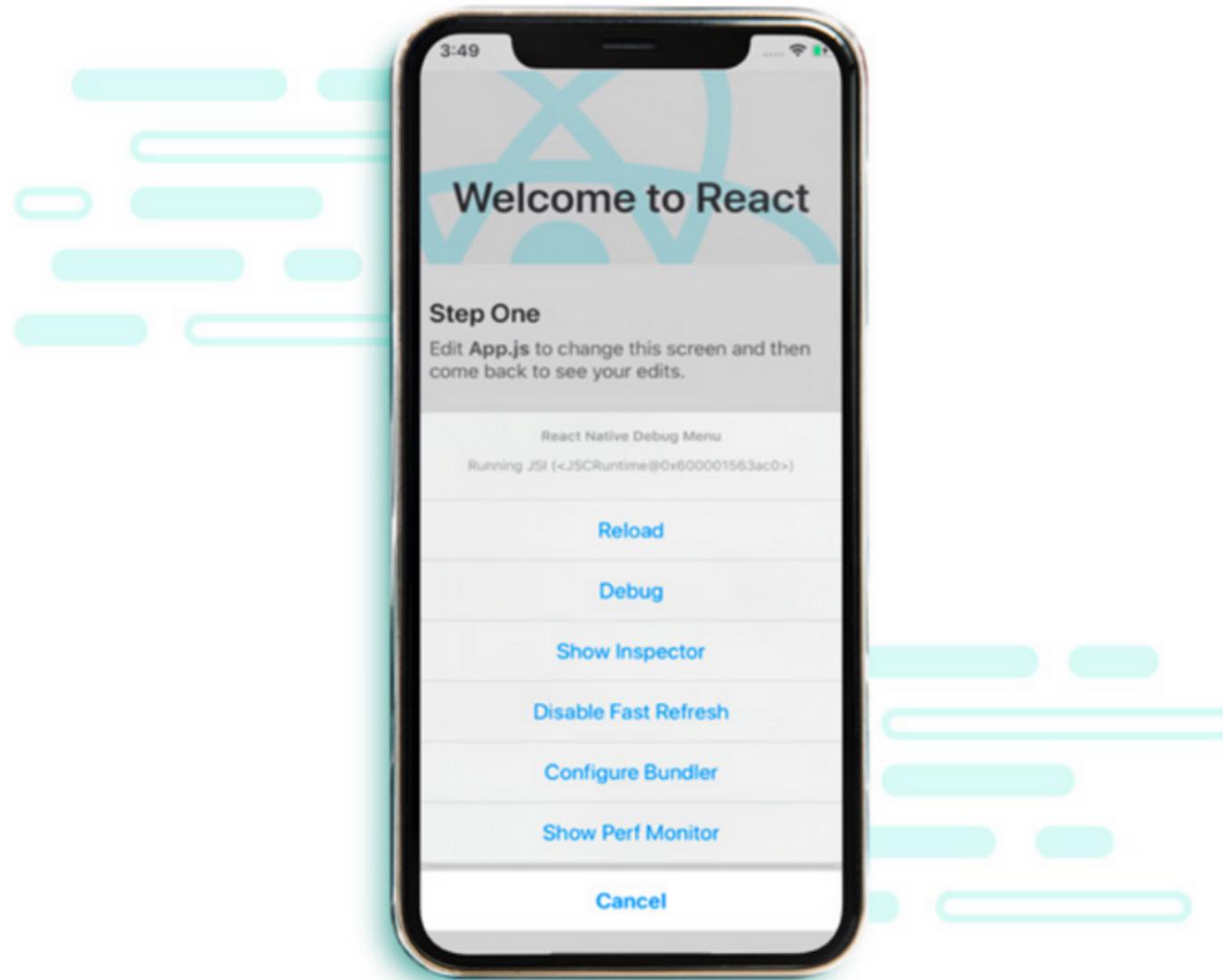
- JavaScript: code ứng dụng của bạn hoặc React Native, hay
- Native Code: thư viện của bên thứ ba hoặc chính React Native.



Khi nói đến gỡ lỗi native code, bạn phải sử dụng các công cụ được tích hợp trong Android Studio và Xcode.

Khi nói đến gỡ lỗi code JavaScript, bạn có thể gặp một số khó khăn. Cách đầu tiên và đơn giản nhất để gỡ lỗi là viết console.logs vào code của bạn và kiểm tra nhật ký trong thiết bị đầu cuối. Phương pháp này chỉ hoạt động để giải quyết các lỗi nhỏ hoặc khi tuân theo thuật chia để trị (divide and conquer technique). Trong tất cả các trường hợp khác, bạn có thể cần sử dụng trình gỡ lỗi bên ngoài.

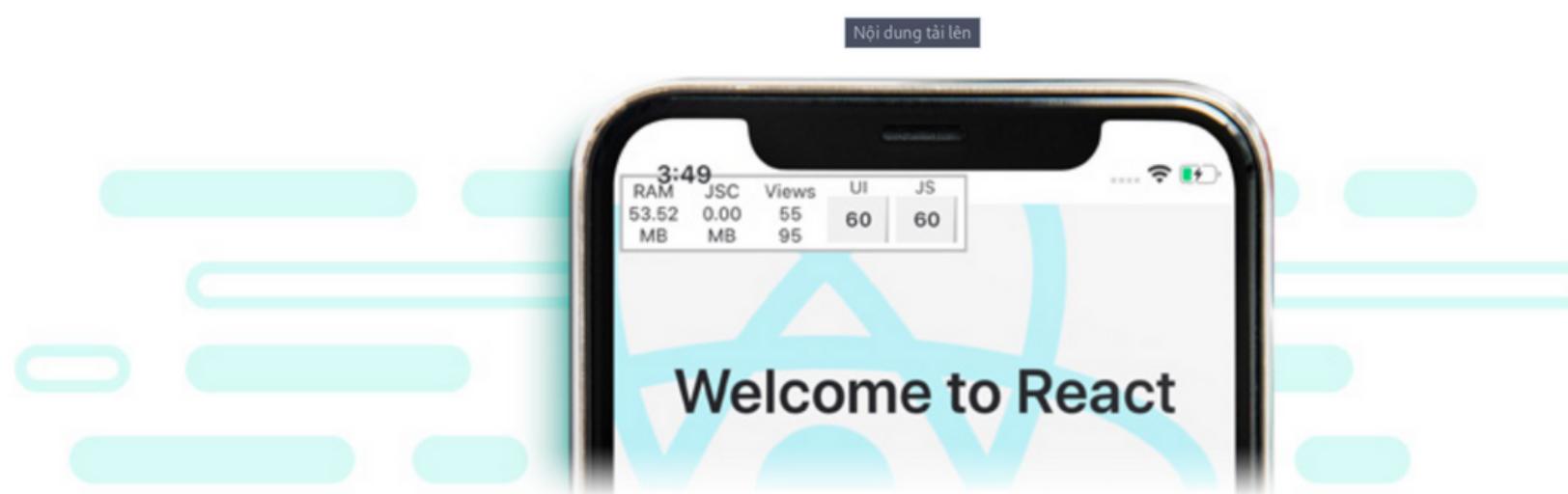
Theo mặc định, React Native có các tiện ích gỡ lỗi tích hợp sẵn.



Một trong những trình gõ lỗi phổ biến nhất là [Google Chrome Debugger](#). Nó cho phép bạn thiết lập các điểm ngắt trong code của mình hoặc xem trước nhật ký theo cách tiện lợi hơn so với trong một thiết bị đầu cuối. Rất tiếc, việc sử dụng Chrome Debugger có thể dẫn đến các sự cố khó phát hiện. Đó là vì code của bạn được thực thi trong công cụ V8 của Chrome thay vì công cụ dành riêng cho nền tảng như JSC hoặc Hermes.

Hướng dẫn được tạo trong Chrome được gửi qua Websocket đến trình mô phỏng hoặc thiết bị. Điều đó có nghĩa là bạn thực sự không thể sử dụng trình gõ lỗi để lập hồ sơ ứng dụng của mình để nó phát hiện các vấn đề về hiệu suất. Nó có thể cung cấp cho bạn một ý tưởng sơ bộ về những gì có thể gây ra sự cố, nhưng bạn sẽ không thể gõ lỗi trường hợp thực do thông điệp của WebSocket được truyền đi.

Một điều bất tiện khác là bạn không thể dễ dàng gõ lỗi các yêu cầu mạng bằng Chrome Debugger (nó cần thiết lập bổ sung và vẫn có những hạn chế của nó). Để gõ lỗi tất cả các yêu cầu có thể có, bạn phải mở trình gõ lỗi mạng chuyên dụng bằng menu nhà phát triển của trình mô phỏng. Tuy nhiên, giao diện của nó rất nhỏ và không hợp thời do kích thước của màn hình trình giả lập.



Từ menu nhà phát triển, bạn có thể truy cập các tiện ích gõ lỗi khác, chẳng hạn như trình kiểm tra bố cục hoặc trình giám sát hiệu suất. Cái sau tương đối thuận tiện để sử dụng, vì nó chỉ hiển thị một phần thông tin nhỏ. Tuy nhiên, việc sử dụng cái trước là một cuộc đấu tranh vì không gian làm việc hạn chế mà nó cung cấp.

Dành nhiều thời gian hơn để gõ lỗi và tìm các vấn đề về hiệu suất đồng nghĩa với việc trải nghiệm của nhà phát triển kém hơn và ít hài lòng hơn

Không giống như các nhà phát triển native, những người làm việc với React Native có quyền truy cập vào một loạt các công cụ và kỹ thuật gõ lỗi. Mỗi thứ bắt nguồn từ một hệ sinh thái khác nhau, chẳng hạn như iOS, Android hoặc JS.

Mặc dù thoát nghe có vẻ tuyệt vời, nhưng bạn cần nhớ rằng mọi công cụ đều yêu cầu một mức độ chuyên môn khác nhau trong việc phát triển native. Điều đó làm cho sự lựa chọn trở nên thách thức đối với đại đa số các nhà phát triển JavaScript.

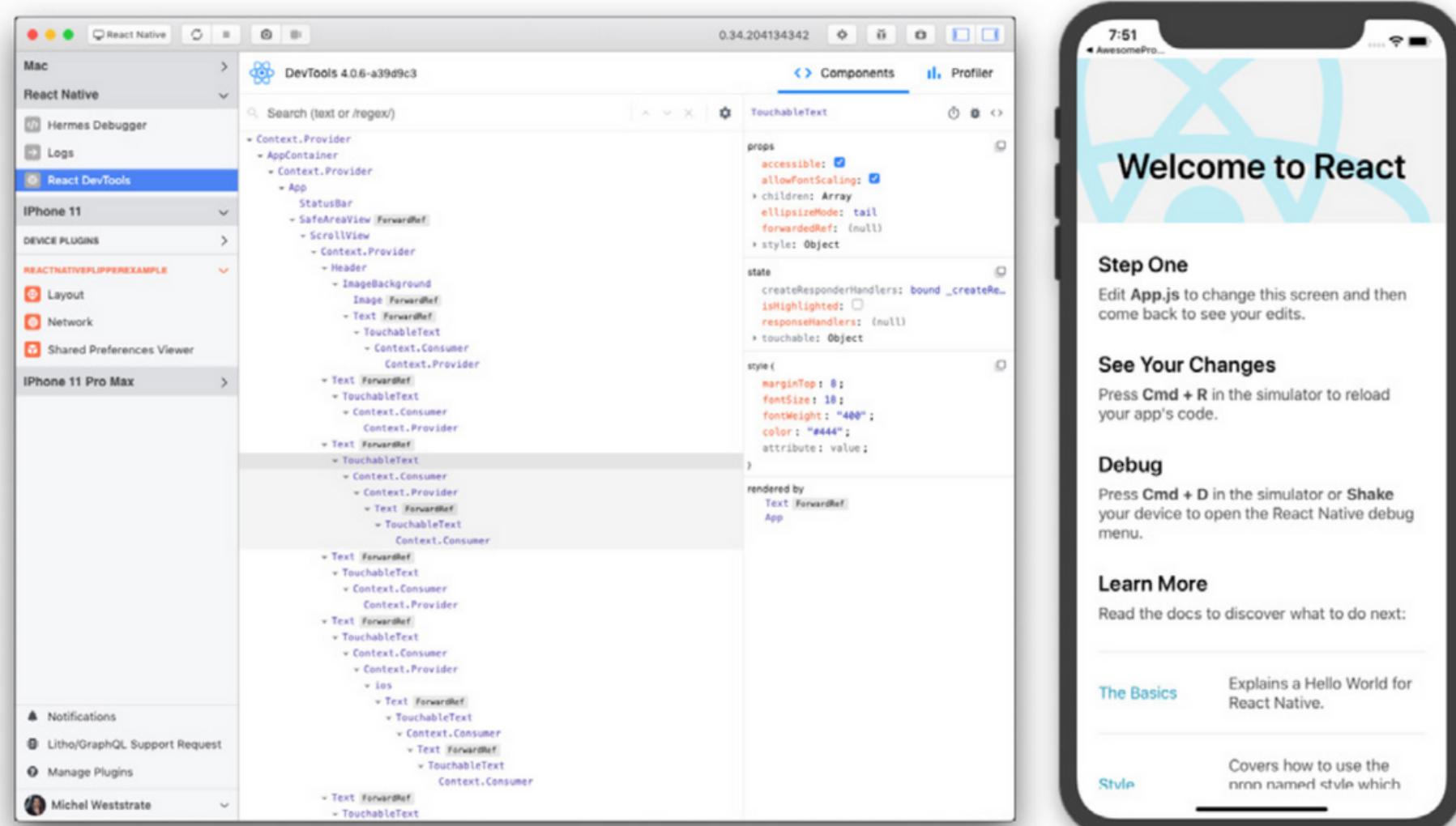
Một công cụ bất tiện thường làm giảm tốc độ của nhóm và khiến các thành viên của họ thất vọng. Kết quả là chúng không hiệu quả như mong muốn. Điều đó ảnh hưởng đến chất lượng của ứng dụng và làm cho các bản phát hành ít thường xuyên hơn.

Giải pháp: Bật Flipper và bắt đầu gõ lỗi với nó

Không có gì tốt hơn nếu có một công cụ toàn diện để xử lý tất cả các trường hợp sử dụng ở trên? Tất nhiên! Và đó là lúc **Flipper** phát huy tác dụng!



Flipper là một nền tảng gõ lỗi cho các ứng dụng di động. Nó cũng hỗ trợ React Native với tư cách là first-class citizen. Nó được ship theo mặc định với React Native kể từ phiên bản 0.62 và được khởi chạy vào tháng 9 năm 2019.



Xem thêm tại: <https://fbflipper.com/docs/features/react-native>

It là một ứng dụng dành cho máy tính để bàn có giao diện thuận tiện, tích hợp trực tiếp với JS của ứng dụng của bạn và native code. Điều này có nghĩa là bạn không còn phải lo lắng về sự khác biệt trong thời gian chạy JS và các cảnh báo về hiệu suất khi sử dụng Chrome Debugger. Nó đi kèm với trình kiểm tra mạng, React DevTools và thậm chí cả công cụ phân cấp native view.

Hơn nữa, Flipper cho phép xem trước nhật ký từ native code và theo dõi các native crash, vì vậy bạn không cần phải chạy Android Studio hoặc Xcode để kiểm tra những gì đang xảy ra ở phía native!

Flipper có thể dễ dàng mở rộng, vì vậy có khả năng cao nó sẽ được bổ sung thêm nhiều plugin hữu ích do cộng đồng phát triển. Tại thời điểm này, bạn có thể sử dụng Flipper cho các tác vụ như phát hiện rò rỉ bộ nhớ, xem trước nội dung của Shared Preferences hoặc kiểm tra hình ảnh đã tải. Ngoài ra, Flipper cho React Native được ship với React DevTools, Hermes Debugger và Metro Bundler integration.

Điều thú vị nhất là tất cả các tiện ích cần thiết đều được đặt trong một desktop app. Điều đó giảm thiểu việc chuyển đổi ngữ cảnh. Nếu không có Flipper, nhà phát triển gỡ lỗi sự cố liên quan đến việc hiển thị dữ liệu được tìm nạp từ chương trình phụ trợ phải sử dụng trình gỡ lỗi Chrome Debugger (để xem trước nhật ký), trình gỡ lỗi yêu cầu mạng trong giả lập và có thể là trình kiểm tra bối cảnh trong giả lập hoặc ứng dụng React Devtools độc lập. Với Flipper, tất cả các công cụ đó đều có sẵn dưới dạng các plugin tích hợp sẵn. Chúng có thể dễ dàng truy cập từ bảng điều khiển bên cạnh và có giao diện người dùng (UI) và trải nghiệm người dùng (UX) tương tự.

Lợi ích: Bạn cảm thấy thú vị hơn khi làm việc với React Native và thiết lập vòng phản hồi tốt hơn

Quy trình gỡ lỗi tốt hơn giúp chu kỳ phát triển ứng dụng của bạn nhanh hơn và dễ dự đoán hơn. Kết quả là, nhóm của bạn có thể tạo ra code đáng tin cậy hơn và phát hiện bất kỳ loại vấn đề nào dễ dàng hơn nhiều.

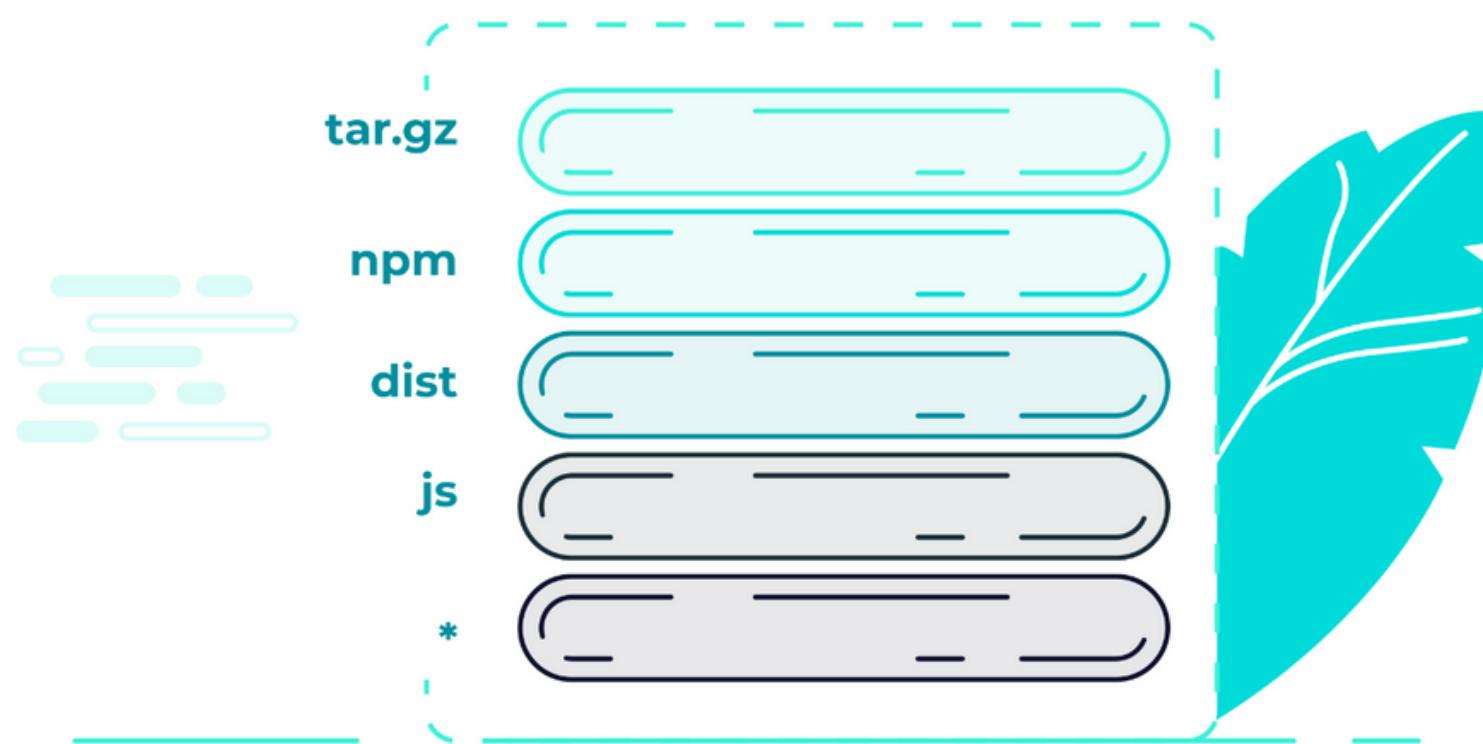
Có tất cả các tiện ích gỡ lỗi trong một giao diện chắc chắn là tiện lợi và không làm gián đoạn bất kỳ tương tác nào với trình giả lập hoặc thiết bị. Quá trình này sẽ ít gánh nặng hơn cho nhóm của bạn và điều đó sẽ tác động tích cực đến tốc độ phát triển sản phẩm và sửa lỗi.

3. Tự động hóa việc quản lý dependency của bạn với “autolinking”

Chuyển sang “autolinking” để thêm các package mới một cách nhanh chóng mà không phải lo lắng về native code.

Vấn đề: Bạn đang thêm thư viện theo cách thủ công hoặc sử dụng “react-native link” không được dùng nữa

Không giống như hầu hết các package có sẵn trên `npm`, thư viện React Native thường bao gồm nhiều hơn một code JavaScript. Tùy thuộc vào loại chức năng mà chúng cung cấp, chúng có thể chứa native code bổ sung cho nền tảng Android và iOS.



Ví dụ: [react-native-fbads](#) là một module React Native được sử dụng để tương tác với Facebook SDK bên dưới và - như tên cho thấy - để hiển thị quảng cáo trong ứng dụng. Để làm như vậy, mô-đun gửi kèm theo code JavaScript cho phép gọi SDK từ vùng React Native. Trên hết, nó cung cấp các native module Objective-C (cho iOS) và Java (cho Android) để ủy quyền các lệnh gọi JavaScript đến các phần SDK Facebook thích hợp. Nó cũng yêu cầu Facebook SDK phải có trong ứng dụng của bạn. Nói cách khác, việc cài đặt nó tùy thuộc vào bạn.

Trước đây, React Native không cung cấp một giải pháp hoàn hảo cho những trường hợp như vậy. Các nhà phát triển được khuyến khích làm theo các phương pháp hay nhất cho một nền tảng nhất định.

Trên Android, đề xuất là sử dụng Gradle, vốn đã là một nền tảng được cộng đồng Android lựa chọn. React Native đã sử dụng Gradle để xây dựng source code và kéo các dependency của riêng nó, điều này tự nhiên buộc tất cả các package cộng đồng phải tuân theo cùng một chiến lược.

Mặt khác, trên iOS, tình hình phức tạp hơn một chút. Theo mặc định, các dự án React Native không sử dụng bất kỳ công cụ phức tạp nào để quản lý các dependency - việc kéo chúng xuống là do bạn. Một số module cộng đồng đã bắt đầu sử dụng CocoaPods, tương tự như Gradle, vì cách nó cấu trúc dự án và cung cấp quản lý dependency thích hợp. Rất tiếc, [CocoaPods](#) không tương thích với cách tất cả các dự án React Native sử dụng để quản lý các dependency theo mặc định.

React Native đã cố gắng giải quyết một phần vấn đề này bằng cách giới thiệu “react-native link” - một lệnh CLI khi chạy sẽ cố gắng thực hiện tất cả các bước cần thiết cho bạn. Nó thực hiện tìm kiếm và thay thế cơ bản trong các tệp cấu hình của bạn và cố gắng thêm các gói được yêu cầu.



\$ rnpm install react-native-vector-icons

```
-> MyAwesomeProject rnpm install react-native-vector-icons  
loadDep:lodash -> 304
```

Thật không may, luôn có rủi ro gặp phải dependency không tương thích với cách bạn quản lý dependency của mình. Trong trường hợp đó, giải pháp duy nhất là chuyển sang một hệ thống hoạt động. Chỉ riêng nhiệm vụ đó đã không dễ dàng - nó đòi hỏi rất nhiều kiến thức liên quan đến native và hiểu biết về các hệ thống xây dựng. Nếu bạn đã từng nâng cấp lên phiên bản React Native đã giới thiệu một số thay đổi native nhất định, bạn sẽ hoàn toàn biết chúng ta đang nói về điều gì.

Theo thời gian, CocoaPods đã bắt đầu ngày càng trở nên phổ biến hơn trong cộng đồng. Cuối cùng, React Native đã quyết định chuyển sang CocoaPods và đặt nó làm cách mặc định để quản lý các dependency bên ngoài trên iOS.

Do đó, cả iOS và Android hiện đã có một giải pháp đầy đủ tính năng để quản lý dependency. Nhờ đó, các nhà phát triển có thể sử dụng một công cụ giống npm để kéo các dependency xuống, thay vì tải xuống các tệp theo cách thủ công và đặt chúng ở đâu đó trên đĩa

Mặc dù điều này đã giải quyết được sự nhầm lẫn xung quanh việc thêm các native dependency bên ngoài, nhưng tình huống vẫn đòi hỏi các bước bổ sung để chạy sau khi `yarn add` đơn giản.



3. Install The Javascript Package

Add the package to your project using your favorite package manager

```
$ yarn add react-native-fbads  
Link the native dependencies  
$ react-native link react-native-fbads
```

For RN < 0.60

If you have more than one Targets on your Xcode project, you might link some of them manually by dragging

[Libraries/ReactNativeAdsFacebook.xcodeproj/Products/libReactNativeAdsFacebook.a](#) to 'Build Phases' -> 'Link Binary With Libraries'.

For RN >= 0.60

If you are working with RN > 0.60 kindly add the following line in your Podfile

```
pod 'ReactNativeAdsFacebook', :path => '../node_modules/react-native-fbads'
```

Nhờ thực tế là cả Gradle và CocoaPods đều có API công khai có thể được sử dụng để thao tác dự án, nhóm React Native đã nhanh chóng đưa ra một tính năng gọi là autolinking. Nó tự động hóa tất cả các bước đã đề cập và loại bỏ sự khác biệt giữa gói React Native và JavaScript.

```
$ yarn add react-native-fbads
```

Installing React Native package should be no different from a regular JavaScript library

Câu chuyện ngắn - nếu bạn đang thực hiện các bước bổ sung sau khi cài đặt modules React Native, bạn nên tiếp tục đọc!

Codebase khó nâng cấp và bảo trì hơn và bạn mất nhiều thời gian hơn để thêm các gói bổ sung

Nếu bạn vẫn đang quản lý các dependency của mình theo “cách cũ” như được mô tả ở trên, bạn đang bỏ lỡ các cải tiến và tự động hóa bản dựng. Do đó, việc thử nghiệm các dependency mới trở nên khó khăn hơn và mất nhiều thời gian hơn để thiết lập chúng. Một số thư viện thậm chí có thể ngừng hoạt động khi các nhà phát triển chuyển chúng sang hệ thống xây dựng mới.

Ngoài ra, bạn cần dành nhiều thời gian hơn để nâng cấp lên các phiên bản React Native mới hơn vì có một loạt các native dependency và native code phải được sửa đổi và nâng cấp

Hệ thống mới dựa trên các công cụ xây dựng native chuyên dụng, chẳng hạn như CocoaPods và Gradle. Do đó, nó có thể xử lý rất nhiều công đoạn tỉ mỉ đó cho bạn.

Giải pháp: Chuyển sang “autolinking” (dựa trên CocoaPods / Gradle)

Autolinking là một cách mới để quản lý các native dependency của bạn, theo thiết kế, hoàn toàn minh bạch và không yêu cầu thêm bất kỳ nỗ lực nào từ phía bạn. Nó rất dễ dàng để tích hợp và nó kết nối ở những nơi mà bạn phải tự xử lý.

Nó hoạt động giống nhau cho cả iOS và Android. Với mục đích của phần này, hãy tập trung vào Android.

Trước

```
app/settings.gradle
```
rootProject.name = 'HelloWorld'

include ':react-native-fs'
project(':react-native-fs').
projectDir = new File(rootProject.
projectDir, '../node_modules/react-
native-fs/android')

include ':app'
```

```

Sau

```
app/settings.gradle
```
rootProject.name = 'HelloWorld'

apply from: file("../node_modules/@
react-native-community/cli-platform-
android/native_modules.gradle");
applyNativeModulesSettingsGradle
(settings);

include ':app'
```

```

```
app/build.gradle
```
dependencies {
 compile project(':react-native-
fs')
 compile fileTree(dir: "libs",
include: ["*.jar"])
 compile "com.android.
support:appcompat-v7:23.0.1"
 compile "com.facebook.
react:react-native:+" // From node_
modules
}
```

```

```
app/build.gradle
```
dependencies {
 compile fileTree(dir: "libs",
include: ["*.jar"])
 compile "com.android.
support:appcompat-v7:23.0.1"
 compile "com.facebook.
react:react-native:+" // From node_
modules
}

apply from: file("../node_modules/@
react-native-community/cli-platform-
android/native_modules.gradle");
applyNativeModulesAppBuildGradle
(project)
```

```

```
MainApplication.java
```
import com.rnfs.RNFSPackage;
```
```
@Override
protected List<ReactPackage>
getPackages() {
 return Arrays.<ReactPackage>asList(
 new MainReactPackage(),
 new RNFSPackage()
);
}
```
```

```

```
MainApplication.java
```
import com.facebook.react.
PackageList;
```
```
@Override
protected List<ReactPackage>
getPackages() {
    return new PackageList(this).
        getPackages();
}
```
```

```

Thay vì cho Gradle biết chi tiết của mọi package bạn đang sử dụng, bạn thay thế danh sách các package bằng một dòng duy nhất gọi vào React Native CLI. Trình trợ giúp nhỏ này sẽ kiểm tra “package.json” của bạn để tìm các package React Native có thể có và tự động thực hiện các hành động cần thiết.

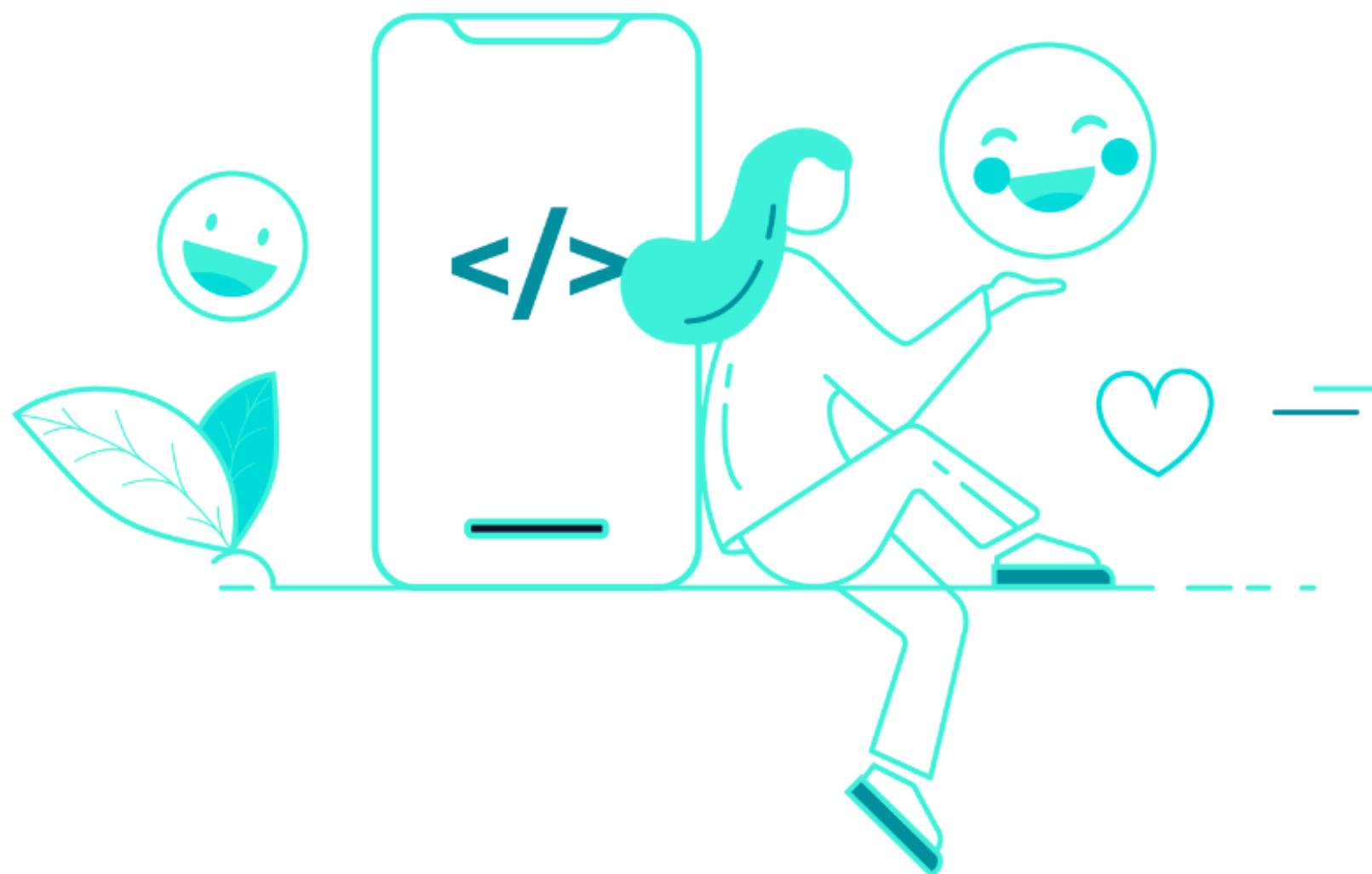
Đây là một ví dụ: Trong build.gradle, lệnh gọi vào React Native CLI dẫn đến một mảng các package sau đó được đăng ký trong đường ống. Cần lưu ý rằng các đường dẫn được tính toán động, dựa trên vị trí của các tệp nguồn của bạn. Do đó, tất cả các kiến trúc phi tiêu chuẩn khác nhau, bao gồm cả các ‘monorepo’s phổ biến, hiện được hỗ trợ theo mặc định.

Một đặc điểm tuyệt vời khác của autolinking là nó tạo danh sách các package cho bạn trên Android. Nhờ đó, tất cả các package được xác định bởi các dependency bên ngoài của bạn sẽ được đăng ký tự động mà không cần phải mở Android Studio và tìm hiểu cách nhập các package trong Java.

Nguyên tắc ở đây rất đơn giản - bạn không cần phải biết thư viện bạn đang tải xuống bao gồm những gì. Khả năng khám phá những chi tiết đó nên được để như một tùy chọn cho những nhà phát triển tò mò nhất.

Lợi ích: Bạn có thể nhanh chóng thêm các package mới và không còn lo lắng về native code

Nhờ sử dụng autolinking, bạn có thể quên đi tất cả sự khác biệt giữa các package JavaScript và React Native thông thường và chỉ việc tập trung vào việc xây dựng ứng dụng của mình.



Bạn không còn phải lo lắng về các dependency bên ngoài hoặc các bước xây dựng bổ sung, bao gồm cả việc kéo SDK hoặc liên kết nội dung.

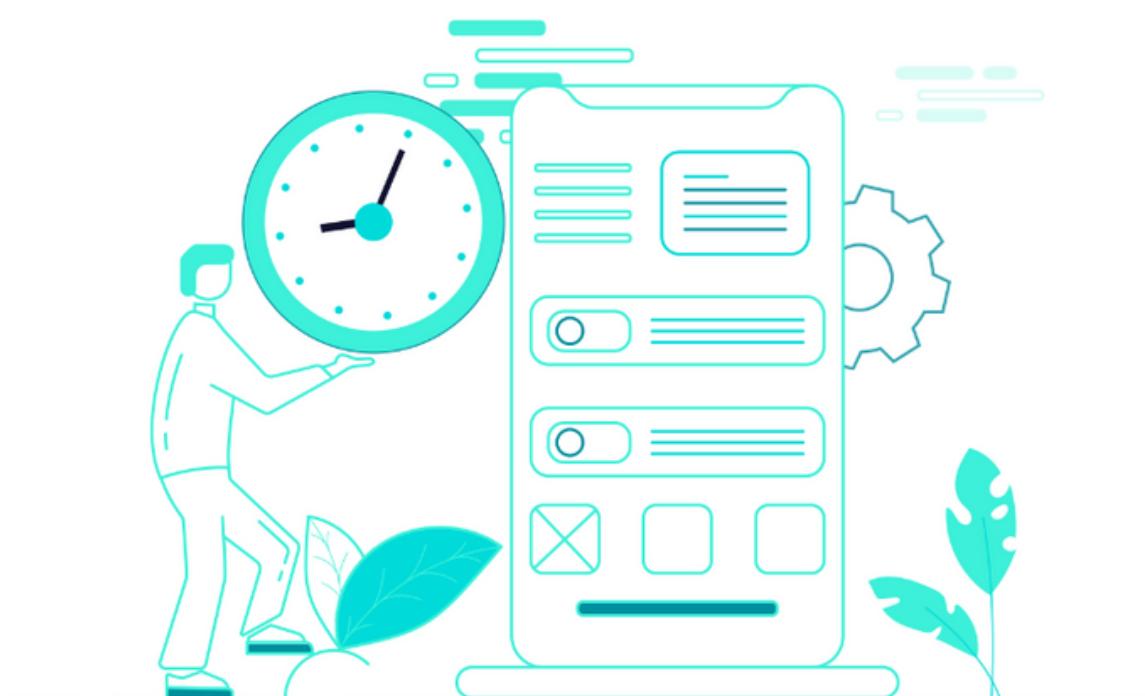
Về lâu dài, bạn sẽ đánh giá cao cách tiếp cận này vì tính dễ bảo trì và tốc độ nâng cấp. Các trình trợ giúp CocoaPods, Gradle và React Native CLI đảm bảo rằng kiến thức cần thiết để thiết lập autolinking và sử dụng nó trong ứng dụng càng cơ bản càng tốt và dễ nắm bắt đối với các nhà phát triển JavaScript.

4. Tối ưu hóa thời gian khởi động ứng dụng Android của bạn với Hermes

Đạt được hiệu năng tốt hơn cho các ứng dụng của bạn với Hermes.

Vấn đề: Bạn đang tải nhiều packages Android trong thời gian startup, điều này là không cần thiết. Ngoài ra, bạn đang sử dụng một công cụ không được tối ưu hóa cho Android.

Người dùng mong đợi các ứng dụng phải phản hồi và tải nhanh. Ứng dụng không đáp ứng được các yêu cầu này có thể nhận được xếp hạng xấu trong App Store hoặc Play Store. Trong những tình huống khắc nghiệt nhất, nó thậm chí có thể bị bỏ rơi vì sự cạnh tranh



Không dễ để mô tả thời gian khởi động bằng một số liệu duy nhất. Đó là do có nhiều giai đoạn khác nhau của giai đoạn tải có thể ảnh hưởng đến cảm giác "nhanh" hoặc "chậm" của ứng dụng. Ví dụ: trong báo cáo Lighthouse, **có sáu chỉ số hiệu suất được sử dụng để lập hồ sơ ứng dụng web của bạn**. Một trong số đó là Time to Interactive (viết tắt là TTI), đo thời gian cho đến khi ứng dụng sẵn sàng cho lần tương tác đầu tiên.

Có khá nhiều điều xảy ra ngay từ khi bạn nhấn vào biểu tượng ứng dụng từ drawer lần đầu tiên.



Quá trình tải bắt đầu với native initialization (1), nó sẽ tải JavaScript VM và khởi tạo tất cả các native module. Sau đó, nó tiếp tục đọc JavaScript từ đĩa (2), tải nó vào bộ nhớ, phân tích cú pháp và bắt đầu thực thi. Chi tiết của hoạt động này đã được thảo luận trước đó trong phần về việc chọn các thư viện phù hợp cho ứng dụng của bạn.

Trong bước tiếp theo (3), React Native bắt đầu tải các thành phần React và gửi bộ hướng dẫn cuối cùng đến UIManager. Cuối cùng, UIManager xử lý thông tin nhận được từ JavaScript và bắt đầu thực hiện các lệnh native (4) sẽ dẫn đến native interface cuối cùng.

Như bạn có thể thấy trên sơ đồ trên, có hai nhóm hoạt động ảnh hưởng đến thời gian khởi động chung của ứng dụng của bạn.



Cách thứ nhất liên quan đến các hoạt động 1 và 2 từ sơ đồ và mô tả thời gian cần thiết để React Native khởi động (để xoay vòng VM và để VM thực thi mã JavaScript). Thao tác còn lại bao gồm các thao tác 3 và 4 còn lại và được liên kết với business logic mà bạn đã tạo cho ứng dụng của mình. Độ dài của nhóm này phụ thuộc nhiều vào số lượng thành phần và độ phức tạp tổng thể của ứng dụng của bạn.

Phần này tập trung vào nhóm đầu tiên - những cải tiến liên quan đến cấu hình của bạn chứ không phải bản thân business logic.

Nếu bạn chưa đo được thời gian khởi động tổng thể của ứng dụng hoặc chưa chơi với những thứ như Hermes - hãy tiếp tục đọc.

Thời gian khởi động lâu và trải nghiệm người dùng chậm trên Android có thể là một trong những lý do khiến ứng dụng của bạn bị xếp hạng kém và cuối cùng bị bỏ rơi

Tạo ra các ứng dụng thú vị để chơi là cực kỳ quan trọng, đặc biệt là trong bối cảnh thị trường di động đã rất bão hòa. Giờ đây, tất cả các ứng dụng dành cho thiết bị di động không chỉ phải dễ hiểu và trực quan mà chúng cũng nên dễ dàng khi tương tác.

Có một quan niệm sai lầm phổ biến rằng các ứng dụng React Native đi kèm với sự đánh đổi hiệu năng so với các native counterpart của chúng. Sự thật là với đủ sự chú ý và chỉnh sửa cấu hình, chúng có thể tải nhanh như nhau và không có bất kỳ sự khác biệt đáng kể nào

Giải pháp: Bật Hermes để hưởng lợi từ hiệu suất tốt hơn

Trong khi ứng dụng React Native chăm sóc native interface, nó vẫn yêu cầu logic JavaScript để chạy trong thời gian thực hiện tác vụ. Để làm như vậy, nó sẽ tách ra khỏi máy ảo JavaScript của chính nó. Theo mặc định, nó sử dụng [JavaScriptCore](#). Công cụ này là một phần của WebKit và theo mặc định chỉ khả dụng trên iOS. Giờ đây, nó cũng là một lựa chọn ưu tiên cho các mục đích tương thích trên Android. Đó là bởi vì việc sử dụng công cụ V8 (đi kèm với Chrome) có thể làm tăng sự khác biệt giữa Android và iOS và khiến việc chia sẻ code giữa các nền tảng trở nên khó khăn hơn.

Các công cụ JavaScript không có một vòng đời dễ dàng. Chúng liên tục đưa ra các phương pháp heuristics mới để cải thiện hiệu năng tổng thể, bao gồm cả thời gian cần thiết để tải code và sau đó thực thi nó. Để làm như vậy, chúng đánh giá các hoạt động JavaScript phổ biến và thách thức CPU và bộ nhớ cần thiết để hoàn thành quá trình này.

Nhóm V8 gần đây đã xuất bản một bài đăng trên blog về việc [cải thiện hiệu suất của biểu thức chính quy](#). Hãy chắc chắn kiểm tra nó.

Hầu hết công việc của các nhà phát triển xử lý các công cụ JavaScript đang được thử nghiệm trên các trang web lớn và phổ biến nhất, chẳng hạn như Facebook hoặc Twitter. Không có gì ngạc nhiên khi React Native sử dụng JavaScript theo một cách khác. Ví dụ: công cụ JavaScript được tạo cho web không phải lo lắng nhiều về thời gian khởi động. Trình duyệt rất có thể đã chạy tại thời điểm tải trang. Do đó, công cụ có thể chuyển sự chú ý sang mức tiêu thụ bộ nhớ và CPU tổng thể, vì các ứng dụng web có thể thực hiện rất nhiều hoạt động và tính toán phức tạp, bao gồm cả đồ họa 3D

Như bạn có thể thấy trên biểu đồ hiệu suất được trình bày trong phần trước, máy ảo JavaScript tiêu thụ một phần lớn thời gian tải của ứng dụng. Thật không may, bạn có thể làm được rất ít điều đó trừ khi bạn xây dựng công cụ của riêng mình. Đó là những gì mà nhóm Facebook đã làm



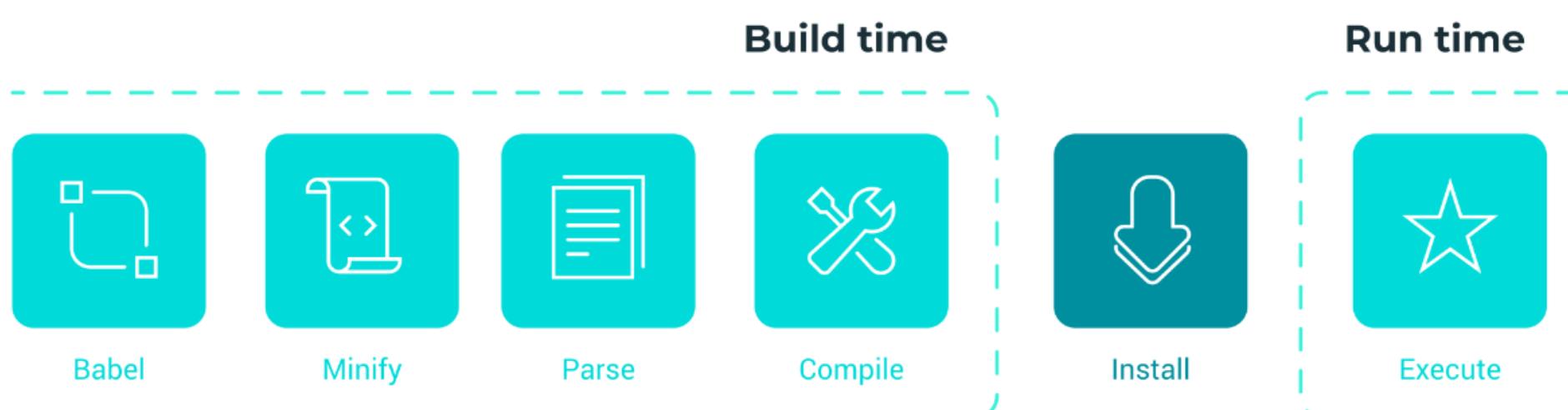
Hermes - một công cụ JavaScript được tạo riêng cho React Native. Nó được tối ưu hóa cho thiết bị di động và tập trung vào các chỉ số tương đối không nhạy cảm với CPU, chẳng hạn như kích thước ứng dụng hoặc Thời gian tương tác. Hiện tại, nó chỉ có sẵn trên Android, với khả năng hỗ trợ trong tương lai cho iOS.

Trước khi chúng ta đi vào chi tiết về cách kích hoạt Hermes trong các ứng dụng React Native hiện có, hãy cùng xem xét một số quyết định kiến trúc chính của nó.

Biên dịch trước Bytecode

Thông thường, máy ảo JavaScript truyền thống hoạt động bằng cách phân tích cú pháp source code JavaScript trong thời gian chạy và sau đó tạo ra mã bytecode. Do đó, việc thực thi mã bị trì hoãn cho đến khi quá trình phân tích cú pháp hoàn tất. Nó không giống với Hermes. Để giảm thời gian cần thiết để công cụ thực thi logic nghiệp vụ, nó tạo ra mã bytecode trong thời gian xây dựng.

Bytecode precompilation with Hermes



Nó có thể dành nhiều thời gian hơn để tối ưu hóa bundle bằng nhiều kỹ thuật khác nhau để làm cho nó nhỏ hơn và hiệu quả hơn. Ví dụ: mã bytecode được thiết kế theo cách để nó có thể sắp xếp trong bộ nhớ mà không cần tải toàn bộ tệp. Tối ưu hóa quy trình đó mang lại những cải tiến đáng kể về TTI vì các hoạt động I / O trên thiết bị di động có xu hướng làm tăng độ trễ tổng thể.

Không JIT

Phần lớn các công cụ trình duyệt hiện đại sử dụng trình biên dịch Just-in-time (JIT). Nó có nghĩa là code được dịch và thực thi từng dòng một.

Tuy nhiên, trình biên dịch JIT theo dõi các đoạn warm code (những đoạn xuất hiện một vài lần) và các đoạn hot code (những đoạn chạy nhiều lần). Các đoạn mã thường xuyên xuất hiện này sau đó được gửi đến một trình biên dịch, tùy thuộc vào số lần chúng xuất hiện trong chương trình, biên dịch chúng thành machine code và thực hiện một số tối ưu hóa theo tùy chọn.

Hermes, không giống như các động cơ khác, là động cơ AOT (đi trước thời đại). Nó có nghĩa là toàn bộ bundle được biên dịch thành bytecode trước thời hạn. Do đó, các tối ưu hóa nhất định mà trình biên dịch JIT sẽ thực hiện trên các phân đoạn hot code sẽ không xuất hiện.

Một mặt, nó làm cho các bundle Hermes hoạt động kém hiệu quả trong các tiêu chuẩn định hướng CPU. Tuy nhiên, những điểm chuẩn này không thực sự so sánh được với trải nghiệm ứng dụng di động ngoài đời thực, nơi TTI và kích thước ứng dụng được ưu tiên.

Mặt khác, các công cụ JIT giảm TTI vì chúng cần thời gian để phân tích bundle và thực thi nó kịp thời. Họ cũng cần thời gian để “warm up”. Cụ thể, họ phải chạy code vài lần để phát hiện các mẫu phổ biến và bắt đầu tối ưu hóa chúng.

Nếu bạn muốn bắt đầu sử dụng Hermes, hãy đảm bảo rằng bạn đang chạy ít nhất React Native 0.60.4 và chuyển các mục sau trong `android / app / build.gradle` của bạn:

```
project.ext.react = [
    entryFile: "index.js",
    enableHermes: true
]
```

`enableHermes` is set to `false` at the time of writing this content. Be sure to swap to `true`.

Nhờ đó, dự án của bạn sẽ được dọn dẹp và xây dựng lại thành công. Nếu điều đó xảy ra, xin chúc mừng - ứng dụng của bạn hiện đang sử dụng Hermes

Lợi ích: Thời gian khởi động tốt hơn dẫn đến hiệu suất tốt hơn. Đó là một câu chuyện chưa có hồi kết

Làm cho ứng dụng của bạn tải nhanh không phải là một nhiệm vụ dễ dàng. Đó là một nỗ lực không ngừng và kết quả cuối cùng của nó sẽ phụ thuộc vào nhiều yếu tố.

Bạn có thể kiểm soát một số ứng dụng bằng cách điều chỉnh cả cấu hình ứng dụng của mình và các công cụ ứng dụng sử dụng để biên dịch source code. Hermes là một trong những điều bạn có thể làm ngay hôm nay để cải thiện đáng kể hiệu năng ứng dụng của mình.

Ngoài ra, bạn cũng có thể xem xét những cải tiến quan trọng khác do nhóm Facebook thực hiện. Để làm như vậy, hãy làm quen với ghi chép của họ về hiệu suất React Native. Nó thường là một cuộc chơi của những cải tiến nhỏ và đơn giản để tạo ra tất cả sự khác biệt khi được áp dụng cùng một lúc.

Như chúng tôi đã đề cập trong phần chạy React Native mới nhất, Hermes là một trong những tài sản mà bạn có thể tận dụng miễn là bạn luôn cập nhật phiên bản React Native của mình.

Làm như vậy sẽ giúp ứng dụng của bạn luôn dẫn đầu trong cuộc chơi hiệu năng và cho phép nó chạy ở tốc độ tối đa.

5.Tối ưu hóa kích thước ứng dụng Android của bạn với các cài đặt Gradle

Cải thiện TTI, giảm mức sử dụng bộ nhớ và kích thước của ứng dụng bằng cách điều chỉnh các quy tắc Proguard cho các dự án của bạn.

Sự cố: Bạn không bật Proguard cho các bản dựng phát hành và tạo APK bằng code cho tất cả các kiến trúc CPU. Bạn ship APK lớn hơn

Khi bắt đầu mỗi dự án React Native, bạn thường không quan tâm đến kích thước ứng dụng. Rốt cuộc, thật khó để đưa ra những dự đoán như vậy sớm trong quá trình này. Nhưng chỉ cần một số dependency bổ sung để ứng dụng phát triển từ 5 MB tiêu chuẩn lên 10, 20 hoặc thậm chí 50, tùy thuộc vào codebase

Bạn có nên thực sự quan tâm đến kích thước ứng dụng trong thời đại Internet di động siêu nhanh và truy cập WiFi ở khắp mọi nơi? Tại sao kích thước bundle tăng nhanh như vậy? Chúng tôi sẽ trả lời những câu hỏi đó trong vài đoạn tiếp theo. Nhưng trước tiên, chúng ta hãy xem một React Native bundle điển hình được tạo thành như thế nào.

Theo mặc định, ứng dụng React Native trên Android bao gồm:

- Bốn bộ mã nhị phân được biên dịch cho các kiến trúc CPU khác nhau,
- Thư mục với các tài nguyên như hình ảnh, phông chữ, v.v.,
- Gói JavaScript với business logic và các React component của bạn,
- Những tập tin khác.

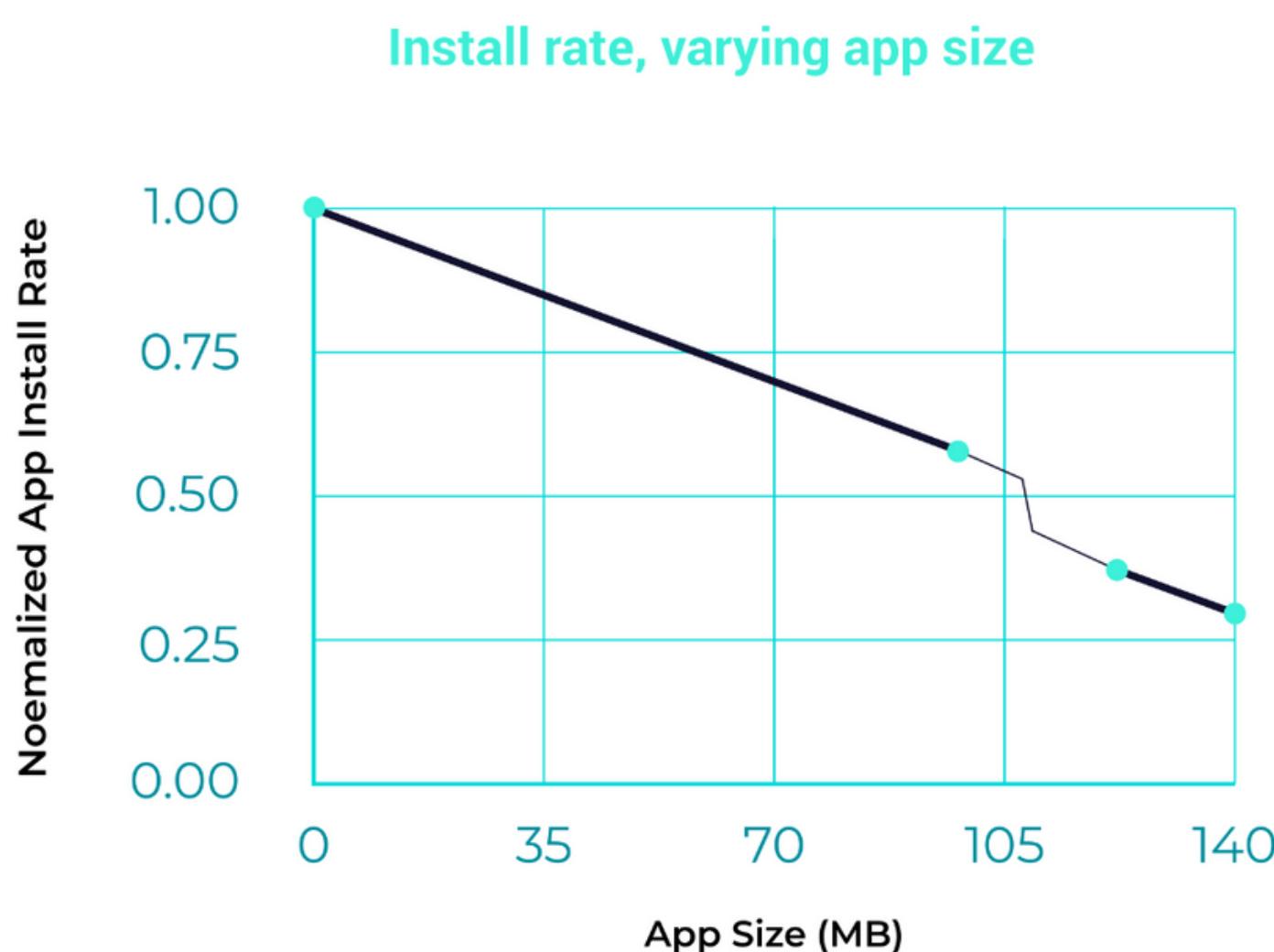
React Native cung cấp một số tối ưu hóa cho phép bạn cải thiện cấu trúc của bundle và kích thước tổng thể của nó. Nhưng chúng bị vô hiệu hóa theo mặc định.

Nếu bạn không sử dụng chúng một cách hiệu quả, đặc biệt là khi ứng dụng của bạn phát triển, bạn đang tăng kích thước tổng thể của ứng dụng theo byte một cách không cần thiết. Điều đó có thể có tác động tiêu cực đến trải nghiệm của người dùng của bạn. Chúng ta sẽ thảo luận về nó trong phần tiếp theo.

Kích thước APK lớn hơn có nghĩa là cần nhiều thời gian hơn để tải xuống từ app store và nhiều bytecode hơn để tải vào bộ nhớ

Thật tuyệt khi bạn và nhóm của mình hoạt động trên các thiết bị mới nhất và có quyền truy cập Internet nhanh chóng và ổn định. Nhưng bạn cần nhớ rằng không phải ai cũng có điều kiện như nhau. Vẫn còn những nơi trên thế giới mà khả năng truy cập và độ tin cậy của mạng còn lâu mới hoàn hảo. Các dự án như Loon hứa hẹn sẽ cải thiện tình hình đó, nhưng điều đó sẽ mất thời gian.

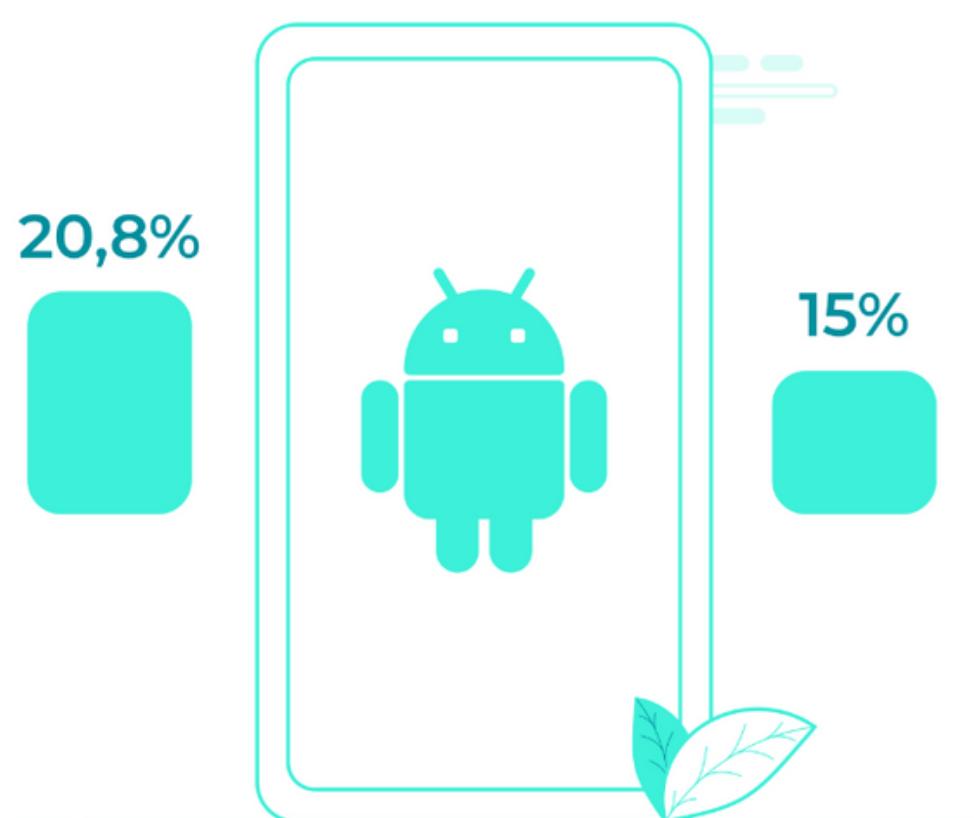
Hiện tại, vẫn có những thị trường mà mỗi megabyte lưu lượng truy cập đều có giá của nó. Ở những khu vực đó, kích thước của ứng dụng ảnh hưởng trực tiếp đến chuyển đổi và cài đặt / tỉ lệ hủy bỏ tăng lên cùng với kích thước ứng dụng.



Nguồn: <https://segment.com/blog/mobile-app-size-effect-on-downloads/>

Đó cũng là niềm tin chung rằng mọi ứng dụng được thiết kế cẩn thận và hoàn thiện không chỉ cung cấp giao diện đẹp mà còn được tối ưu hóa cho thiết bị mục tiêu. Mặc dù không phải lúc nào cũng vậy. Và bởi vì thị trường Android rất cạnh tranh, có khả năng lớn là một giải pháp thay thế nhỏ hơn cho những ứng dụng lớn nhưng đẹp đó đã thu hút được nhiều sự chú ý hơn từ cộng đồng.

Một yếu tố quan trọng khác là sự phân mảnh của thiết bị. Thị trường Android rất đa dạng về mặt đó. Có một tỷ lệ tương đối lớn các thiết bị tầm trung đến cấp thấp có thể gặp vấn đề khi xử lý các APK lớn hơn



In March 2019 ONLY 20.8% Android smartphones were high-end, up from 15% in March 2018

Nguồn: <https://newzoo.com/insights/infographics/10-key-facts-about-the-android-smartphone-market/>

Như chúng tôi đã nhấn mạnh, thời gian khởi động ứng dụng của bạn là điều cần thiết. Thiết bị càng phải thực thi nhiều mã trong khi mở mã của bạn, thì càng mất nhiều thời gian để khởi chạy ứng dụng và sẵn sàng cho lần tương tác đầu tiên

Bây giờ, hãy chuyển đến yếu tố cuối cùng đáng được đề cập đến trong bối cảnh này - bộ nhớ thiết bị.

Các ứng dụng thường chiếm nhiều dung lượng hơn sau khi cài đặt. Đôi khi chúng thậm chí có thể không vừa với bộ nhớ thiết bị. Trong trường hợp đó, người dùng có thể quyết định bỏ qua cài đặt sản phẩm của bạn nếu điều đó có nghĩa là xóa các tài nguyên khác như ứng dụng hoặc hình ảnh.

Giải pháp: Lật cờ boolean `enableProguardInReleaseBuilds` thành true, điều chỉnh các quy tắc Proguard theo nhu cầu của bạn và kiểm tra các bản phát hành phát hành xem có lỗi không. Ngoài ra, hãy chuyển `enableSeparateBuildPerCPUArchitecture` thành true

Android là một hệ điều hành chạy trên nhiều thiết bị với các kiến trúc khác nhau, vì vậy bản dựng của bạn phải hỗ trợ hầu hết chúng. React Native hỗ trợ bốn codes: armeabi-v7a, arm64-v8a, x86 và x86_64.

Trong khi phát triển ứng dụng của bạn, Gradle tạo tệp `apk` có thể được cài đặt trên bất kỳ thiết bị kiến trúc CPU nào được đề cập. Nói cách khác, `apk` của bạn (tệp được xuất ra từ quá trình xây dựng) thực sự là bốn ứng dụng riêng biệt được đóng gói thành một tệp duy nhất. Điều này làm cho việc thử nghiệm dễ dàng hơn vì ứng dụng có thể được phân phối trên nhiều thiết bị thử nghiệm khác nhau cùng một lúc.

Thật không may, cách tiếp cận này có những hạn chế của nó. Kích thước tổng thể của ứng dụng hiện lớn hơn nhiều so với kích thước của nó vì nó chứa các tệp được yêu cầu bởi tất cả các kiến trúc. Kết quả là người dùng sẽ phải tải xuống code không liên quan thậm chí không tương thích với điện thoại của họ.

Rất may, bạn có thể tối ưu hóa quy trình phân phối bằng cách tận dụng [App Bundles](#) khi phát hành phiên bản sản xuất của ứng dụng.

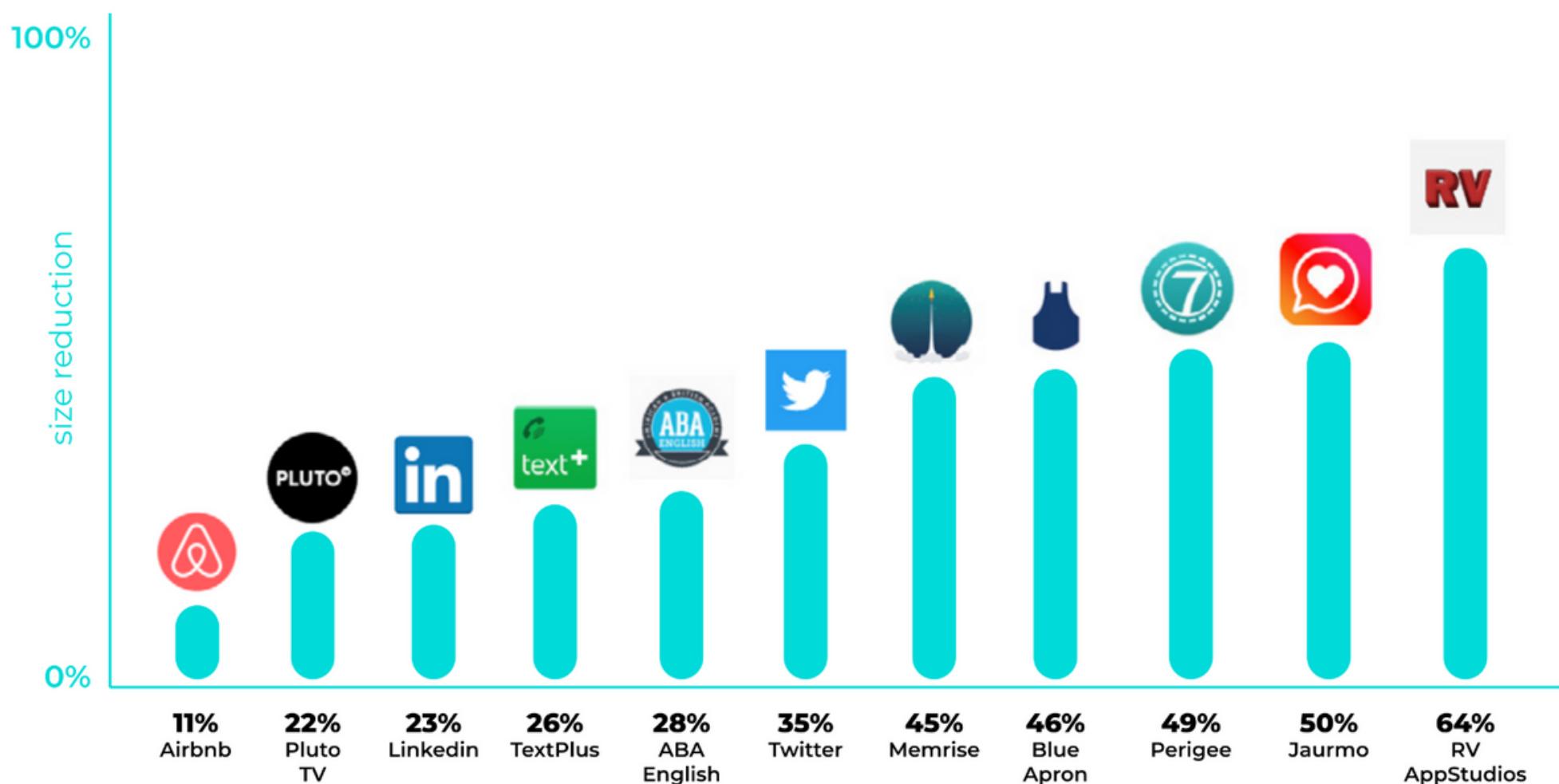
[App Bundle](#) là một định dạng xuất bản cho phép bạn chứa tất cả các code và tài nguyên đã biên dịch. Tất cả là do Google Play Store Dynamic Delivery sau này sẽ tạo các APK phù hợp tùy thuộc vào thiết bị của người dùng mục tiêu.

Để xây dựng App Bundle, bạn chỉ cần gọi một tập lệnh khác với thông thường. Thay vì sử dụng `./gradlew assembleRelease`, hãy sử dụng `./gradlew packRelease`, như được trình bày ở đây:

```
$ cd android  
$ ./gradlew bundleRelease
```

Building a React Native app as App Bundle

Ưu điểm chính của Android App Bundle so với các bản dựng cho nhiều kiến trúc trên mỗi CPU là dễ phân phối. Sau cùng, bạn chỉ phải ship một hiện vật và Dynamic Delivery sẽ làm toàn bộ điều kỳ diệu cho bạn. Nó cũng mang lại cho bạn sự linh hoạt hơn trên các nền tảng được hỗ trợ. Bạn không phải lo lắng về kiến trúc CPU mà thiết bị của người dùng mục tiêu của bạn có. Theo nhóm Android, mức giảm kích thước trung bình cho một ứng dụng là khoảng 35%, nhưng trong một số trường hợp, nó thậm chí có thể giảm một nửa.



Nguồn: <https://medium.com/google-developer-experts/exploring-the-android-app-bundle-ca16846fa3d7>

Một cách khác để giảm kích thước bản dựng là bật Proguard. Proguard hoạt động theo cách tương tự như loại bỏ dead code khỏi JavaScript - nó loại bỏ code không sử dụng khỏi SDK của bên thứ ba và thu nhỏ codebase

Tuy nhiên, Proguard có thể không hoạt động hiệu quả với một số dự án và thường yêu cầu thiết lập bổ sung để đạt được kết quả tối ưu. Trong ví dụ này, chúng tôi có thể giảm kích thước của bản dựng 28 MB được đề cập xuống 700Kb. Nó không phải là nhiều, nhưng nó vẫn là một cải tiến

```
def enableProguardInReleaseBuilds = true
```

Enabling `proguard` in `android/app/build.gradle`

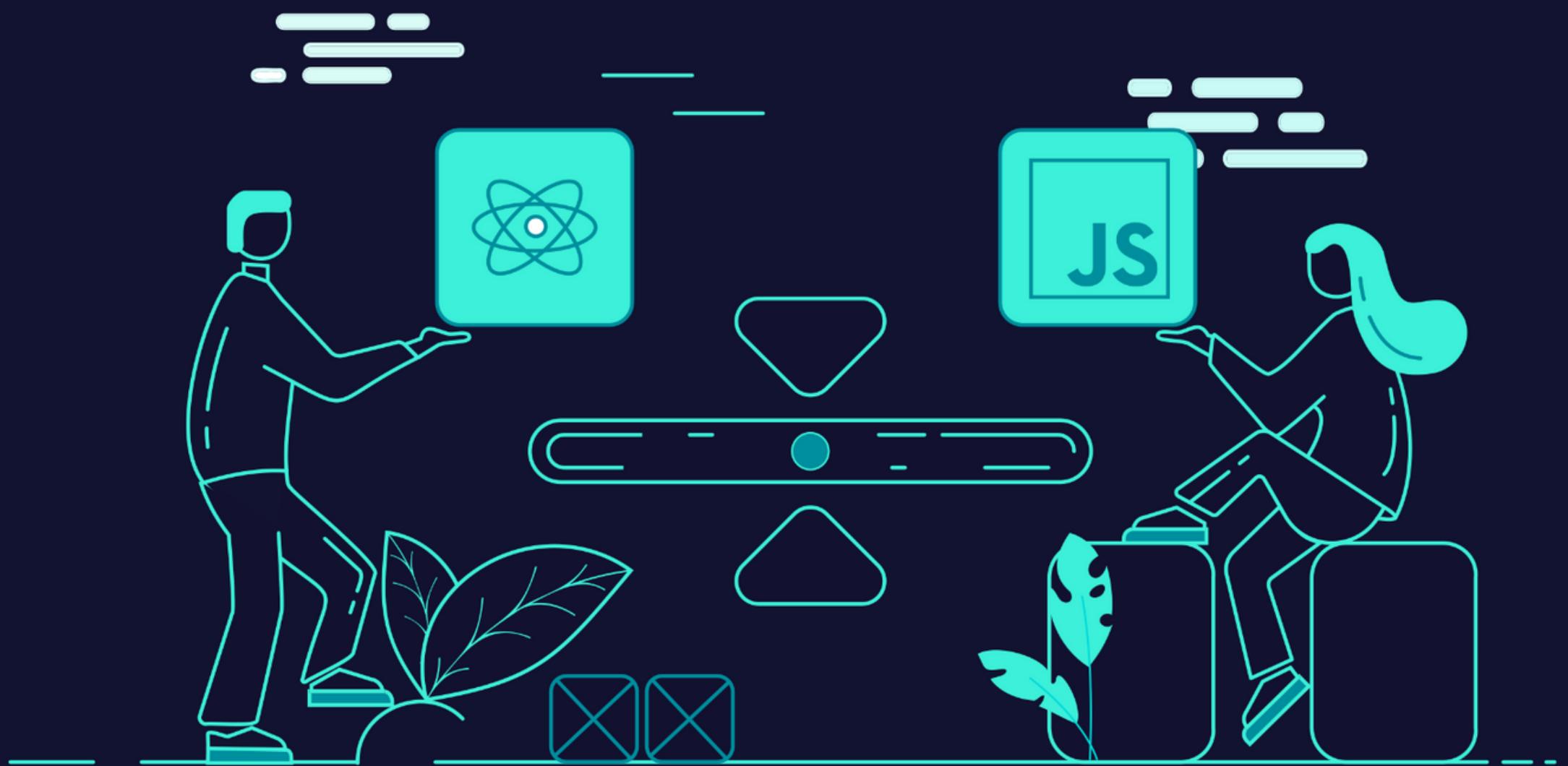
Một phương pháp hay khác là theo dõi tối ưu hóa tài nguyên. Mỗi ứng dụng chứa một số đồ họa svg hoặc png có thể được tối ưu hóa bằng các công cụ web miễn phí. Giảm văn bản thừa từ svg và nén hình ảnh png có thể tiết kiệm một số byte khi dự án của bạn đã có nhiều byte.

Lợi ích: APK nhỏ hơn, TTI nhanh hơn một chút, sử dụng ít bộ nhớ hơn một chút

Tất cả các bước được đề cập đều tương đối dễ giới thiệu và đáng thực hiện khi bạn đang gặp khó khăn với kích thước ứng dụng ngày càng tăng. Bạn sẽ đạt được mức giảm kích thước đáng kể nhất bằng cách xây dựng ứng dụng cho các kiến trúc khác nhau. Nhưng danh sách các cách tối ưu hóa có thể không chỉ dừng lại ở đó.

Bằng cách cố gắng đạt được kích thước APK nhỏ hơn, bạn sẽ cố gắng hết sức để giảm tỷ lệ hủy tải xuống. Ngoài ra, khách hàng của bạn sẽ được hưởng lợi từ thời gian tương tác ngắn hơn và có xu hướng sử dụng ứng dụng thường xuyên hơn.

Cuối cùng, bạn sẽ chứng minh rằng bạn quan tâm đến mọi người dùng, không chỉ những người có thiết bị topnotch và kết nối internet nhanh. Nền tảng của bạn càng lớn, thì việc hỗ trợ các nhóm nhỏ đó càng quan trọng hơn, vì mỗi phần trăm người dùng chuyển thành hàng trăm nghìn người dùng thực tế.



Tập 3

Cách để phát hành ứng dụng nhanh hơn
trong môi trường phát triển ổn định.

Giới thiệu

React Native là lựa chọn tuyệt vời để shipping nhanh chóng và tin cậy, nhưng bạn đã sẵn sàng cho điều đó chưa?

Ngày nay, việc có một setup phát triển ổn định và thoải mái, khuyến khích shipping các tính năng mới và không làm bạn chậm lại là điều bắt buộc. Bạn phải ship nhanh và đi trước đối thủ cạnh tranh của mình.

React Native hoạt động thực sự tốt trong môi trường như vậy. Ví dụ: một trong những điểm nổi trội nhất của nó cho phép bạn gửi các bản cập nhật cho các ứng dụng của mình mà không cần gửi qua App Store. Nó được gọi là cập nhật qua mạng hoặc OTA trong ngắn hạn.

Câu hỏi đặt ra là: Ứng dụng của bạn đã sẵn sàng cho việc đó chưa? Quy trình phát triển của bạn có đầy nhanh quá trình phát triển và shipping các tính năng với React Native không?

Hầu hết thời gian, câu trả lời bạn muốn chỉ đơn giản là có. Nhưng trong thực tế, nó trở nên phức tạp.

Trong phần này, chúng tôi trình bày một số phương pháp hay nhất và đề xuất cho phép bạn ship ứng dụng của mình nhanh hơn và tự tin hơn. Và không chỉ là bật cập nhật Over The Air như hầu hết các bài báo đề xuất. Đó là về việc xây dựng một môi trường phát triển ổn định và lành mạnh, nơi React Native tỏa sáng và đầy nhanh sự đổi mới.

Và đó chính là nội dung của phần này

1. Chạy thử nghiệm cho các phần chính của ứng dụng

Tập trung thử nghiệm vào các phần chính của ứng dụng để có cái nhìn tổng quan hơn về các tính năng và chỉnh sửa mới.

Vấn đề: Bạn hoàn toàn không viết các bài test hoặc viết các bài test với chất lượng thấp mà không có mức độ phù hợp thực sự và chỉ dựa vào kiểm tra thủ công

Tự tin xây dựng và triển khai các ứng dụng là một nhiệm vụ đầy thách thức. Tuy nhiên, việc xác minh xem mọi thứ có thực sự hoạt động hay không đòi hỏi nhiều thời gian và nỗ lực - bất kể nó có được tự động hóa hay không. Có ai đó xác minh thủ công rằng phần mềm hoạt động như mong đợi là điều quan trọng đối với sản phẩm của bạn.

Rất tiếc, quá trình này không mở rộng quy mô cũng như số lượng các chức năng ứng dụng của bạn phát triển. Nó cũng không cung cấp phản hồi trực tiếp cho các nhà phát triển viết code. Do đó, nó làm tăng thời gian cần thiết để phát hiện và sửa lỗi.

Vậy các nhà phát triển phải làm gì để đảm bảo phần mềm của họ luôn sẵn sàng sản xuất và không phụ thuộc vào người kiểm tra là con người? Họ viết các bài test tự động. Và React Native cũng không ngoại lệ. Bạn có thể viết nhiều bài test khác nhau cho cả code JS của mình - chứa business logic và UI - và native code được sử dụng bên dưới. Bạn có thể làm điều đó bằng cách sử dụng các framework thử nghiệm end-to-end, quay vòng các trình mô phỏng, trình giả lập hoặc thậm chí là các thiết bị thực tế. Một trong những tính năng tuyệt vời của React Native là nó bundles vào một native app bundle, vì vậy nó cho phép bạn sử dụng tất cả các khuôn khổ thử nghiệm end-to-end mà bạn yêu thích và sử dụng trong các native project của mình.

Nhưng hãy cẩn thận, viết một bài test có thể là một nhiệm vụ khó khăn, đặc biệt là nếu bạn thiếu kinh nghiệm. Rất dễ kết thúc với một bài test không có mức độ phù hợp tốt về các tính năng của bạn. Hoặc chỉ để kiểm tra một hành vi tích cực, mà không cần xử lý các trường hợp ngoại lệ. Việc gặp phải các thử nghiệm chất lượng thấp không mang lại quá nhiều giá trị và do đó sẽ không giúp bạn tự tin shipping code là điều rất phổ biến.

Cho dù bạn sẽ viết loại bài test nào, có thể là đơn vị, tích hợp hoặc E2E (viết tắt của end-to-end), có một quy tắc vàng sẽ giúp bạn tránh viết sai. Và quy tắc đó là "tránh kiểm tra chi tiết triển khai". Hãy tuân thủ nó và thử nghiệm của bạn sẽ bắt đầu cung cấp giá trị theo thời gian.

Bạn không thể tiến nhanh như đối thủ, khả năng bị thut lùi rất cao, ứng dụng có thể bị xóa khỏi app store khi nhận được đánh giá không tốt.

Mục tiêu chính của việc kiểm tra code của bạn là triển khai nó một cách tự tin bằng cách giảm thiểu số lượng bugs bạn đưa vào codebase của mình. Và việc không shipping bugs cho người dùng đặc biệt quan trọng đối với các ứng dụng dành cho thiết bị di động, những ứng dụng này thường được xuất bản lên app stores. Do đó, chúng là đối tượng của một quá trình xem xét kéo dài, có thể mất từ vài giờ đến vài ngày. Và điều cuối cùng bạn muốn là làm người dùng thất vọng với một bản cập nhật khiến ứng dụng của bạn bị lỗi. Điều đó có thể dẫn đến việc giảm xếp hạng của bạn và trong trường hợp nghiêm trọng, thậm chí có thể gỡ ứng dụng khỏi store.

Những trường hợp như vậy có vẻ khá hiếm, nhưng chúng vẫn xảy ra, sau đó, nhóm của bạn có thể trở nên lo sợ về một sự thut lùi và sụp đổ khác đến mức mất toàn bộ tốc độ và sự tự tin

Giải pháp: Đừng nhắm đến mức độ phù hợp 100%, hãy tập trung vào các phần chính của ứng dụng. Sử dụng các bài kiểm tra đơn vị (Ảnh chụp nhanh), kiểm tra tích hợp (Detox)

Chạy thử nghiệm không phải là câu hỏi về “nếu” mà là “như thế nào”. Bạn cần đưa ra kế hoạch làm thế nào để đạt được giá trị tốt nhất cho thời gian đã bỏ ra. Rất khó để có 100% dòng code và phần phụ thuộc của bạn được cover. Ngoài ra, nó thường khá phi thực tế.

Hầu hết các ứng dụng dành cho thiết bị di động hiện có không cần toàn bộ phạm vi kiểm tra của mã mà chúng viết.

Một số trường hợp ngoại lệ là các tình huống trong đó khách hàng yêu cầu bảo hiểm đầy đủ vì quy định của chính phủ mà họ phải tuân theo. Nhưng trong trường hợp như vậy, bạn có thể đã nhận thức được vấn đề.

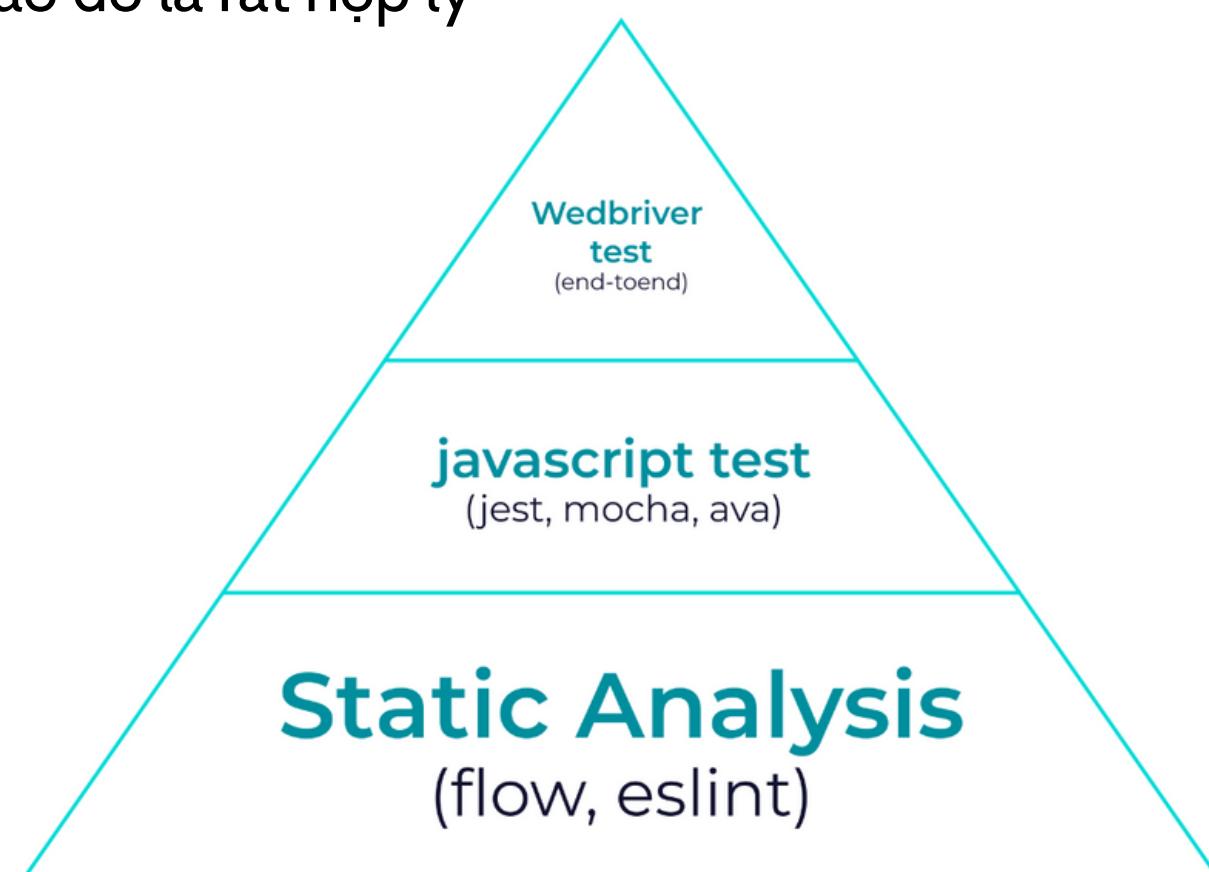
Điều quan trọng là bạn phải tập trung thời gian của mình vào việc thử nghiệm điều phù hợp. Học cách xác định các tính năng và khả năng quan trọng của doanh nghiệp thường quan trọng hơn là viết một bài test. Rốt cuộc, bạn muốn tăng thêm sự tin cậy vào code của mình chứ không phải là một bài test viết vì lợi ích của nó. Một khi bạn làm điều đó, tất cả những gì bạn cần làm là quyết định chạy nó như thế nào. Bạn có khá nhiều tùy chọn để lựa chọn.

Trong React Native, ứng dụng của bạn bao gồm nhiều lớp code, một số được viết bằng JS, một số bằng Java/Kotlin, một số bằng Objective-C / Swift và một số thậm chí bằng C ++, đang được áp dụng trong React Native

Do đó, vì những lý do thực tế, chúng ta có thể phân biệt giữa:

- Kiểm tra JavaScript - với sự trợ giúp của Jest framework. Trong ngữ cảnh React Native nếu bạn nghĩ về các bài kiểm tra “đơn vị” hoặc “tích hợp”, thì đây là danh mục cuối cùng chúng rơi vào. Từ quan điểm thực tế, không có lý do gì để phân biệt giữa hai nhóm đó.
- Thử nghiệm ứng dụng end-to-end - với sự trợ giúp của Detox, Appium hoặc mobile testing framework khác mà chúng ta quen thuộc.

Bởi vì hầu hết code doanh nghiệp của bạn đều nằm trong JS, nên việc tập trung nỗ lực của bạn vào đó là rất hợp lý



Nguồn: https://twitter.com/aaronabramov_/status/805913874704674816

JavaScript testing

Viết các bài test cho các chức năng tiện ích sẽ khá đơn giản. Để làm như vậy, bạn có thể sử dụng test runner yêu thích của mình. Phổ biến nhất và được đề xuất trong cộng đồng React Native là Jest. Chúng tôi cũng sẽ đề cập đến nó trong các phần sau.

Tuy nhiên, để kiểm tra các thành phần React, bạn cần các công cụ nâng cao hơn. Hãy lấy thành phần sau làm ví dụ:

```
function QuestionsBoard({ questions, onSubmit }) {
  const [data, setData] = React.useState({});

  return (
    <ScrollView>
      {questions.map((q, index) => {
        return (
          <View key={q}>
            <Text>{q}</Text>
            <TextInput
              accessibilityLabel="answer input"
              onChangeText={text => {
                setData(state => ({
                  ...state,
                  [index + 1]: { q, a: text },
                }));
              }}
            />
            </View>
        );
      })}
      <TouchableOpacity onPress={() => onSubmit(data)}>
        <Text>Submit</Text>
      </TouchableOpacity>
    </ScrollView>
  );
}
```

Nó là một thành phần React hiển thị danh sách các câu hỏi và cho phép trả lời chúng. Bạn cần đảm bảo rằng logic của nó hoạt động bằng cách kiểm tra xem hàm gọi lại có được gọi với tập hợp các câu trả lời do người dùng cung cấp hay không.

Để làm như vậy, bạn có thể sử dụng thư viện react-test-renderer chính thức từ nhóm chính của React. Nó là một trình render thử nghiệm - và nói cách khác - nó cho phép bạn render component của mình và tương tác với vòng đời của nó mà không thực sự xử lý các API native. Một số người có thể thấy nó khá đáng sợ và khó làm việc vì API cấp thấp.

Đó là lý do tại sao cộng đồng xung quanh React Native đã ra mắt các thư viện trợ giúp, chẳng hạn như React Native Testing Library, cung cấp cho chúng ta một bộ trợ giúp tốt để viết các bài test chất lượng cao một cách hiệu quả.

Một điều tuyệt vời về thư viện này là API của nó buộc bạn phải tránh kiểm tra các chi tiết triển khai của các thành phần của bạn, làm cho nó linh hoạt hơn đối với các trình tái cấu trúc nội bộ.

Một bài kiểm tra cho thành phần QuestionBoard sẽ trông như sau:

```
import { render, fireEvent } from 'react-native-testing-library';
import QuestionsBoard from '../QuestionsBoard';

test('form submits two answers', () => {
  const allQuestions = ['q1', 'q2'];
  const mockFn = jest.fn();
  const { getAllByA11yLabel, getByText } = render(
    <QuestionsBoard questions={allQuestions} onSubmit={mockFn} />
  );
  const answerInputs = getAllByA11yLabel('answer input');

  fireEvent.changeText(answerInputs[0], 'a1');
  fireEvent.changeText(answerInputs[1], 'a2');
  fireEvent.press(getByText('Submit'));
});
```

```
expect(mockFn).toBeCalledWith({  
  '1': { q: 'q1', a: 'a1' },  
  '2': { q: 'q2', a: 'a2' },  
});  
});
```

Test suite taken from the official RNTL documentation

Trước tiên, bạn sẽ 'render' thành phần QuestionsBoard với bộ câu hỏi của bạn. Tiếp theo, bạn sẽ query the tree theo vai trò trợ năng để truy cập vào một mảng câu hỏi, như được hiển thị bởi thành phần. Cuối cùng, bạn sẽ đặt các câu trả lời đúng và nhấn nút 'submit'

Nếu mọi thứ diễn ra tốt đẹp, xác nhận của bạn sẽ được chấp nhận, đảm bảo rằng hàm `verifyQuestions` đã được gọi với tập hợp các đối số phù hợp.

Lưu ý: Bạn cũng có thể đã nghe nói về một kỹ thuật được gọi là "snapshot testing" cho JS. Nó có thể giúp bạn trong một số tình huống thử nghiệm, khi dữ liệu lặp lại được xác nhận từ thử nghiệm. Kỹ thuật này được áp dụng rộng rãi trong hệ sinh thái React, do có sự hỗ trợ tích hợp từ Jest. Nhưng đó là một API cấp thấp và nên tránh, trừ khi bạn có kinh nghiệm testing vững chắc.

Nếu bạn muốn tìm hiểu thêm về snapshots, hãy xem một trong những bài nói chuyện của Rogelio (một trong những cộng tác viên của Jest) về snapshot testing và kho lưu trữ dự án có thể giúp bạn kiểm tra sự khác biệt giữa các trạng thái dữ liệu khác nhau, bao gồm cả các thành phần React

E2E tests

Điểm chính của kim tự tháp thử nghiệm của chúng tôi là một bộ các thử nghiệm end-to-end. Tốt nhất là nên bắt đầu với cái gọi là "smoke test" - một thử nghiệm đảm bảo rằng ứng dụng của bạn không gặp sự cố trong lần chạy đầu tiên. Điều quan trọng là phải có một bài test như thế này, vì nó sẽ giúp bạn tránh gửi một ứng dụng bị lỗi cho người dùng của mình. Sau khi hoàn thành các kiến thức cơ bản, bạn nên sử E2E testing framework mà mình lựa chọn để cover các chức năng quan trọng nhất của ứng dụng. Đây có thể là ví dụ: đăng nhập (thành công hoặc không), đăng xuất, chấp nhận thanh toán, hiển thị danh sách dữ liệu bạn lấy từ máy chủ của bạn hoặc bên thứ ba

Lưu ý: Hãy lưu ý rằng các bài test này thường khó thiết lập hơn một chút so với các bài test JS. Ngoài ra, họ có nhiều khả năng thất bại vì các vấn đề liên quan ví dụ như: mạng, hoạt động của hệ thống tệp hoặc bộ nhớ hoặc memory shortage. Hơn nữa, họ cung cấp cho bạn một ít thông tin về lý do tại sao họ làm điều đó. Chất lượng của thử nghiệm này (không chỉ đối với các thử nghiệm E2E) được gọi là "không ổn định" và cần phải tránh bằng mọi giá, vì nó làm giảm niềm tin của bạn vào bộ thử nghiệm. Đó là lý do tại sao việc phân chia các testing assertions thành các nhóm nhỏ hơn là rất quan trọng, vậy nên việc gỡ lỗi đã xảy ra sẽ dễ dàng hơn.

Với mục đích của phần này, chúng ta sẽ xem xét Detox - test runner E2E phổ biến nhất trong cộng đồng React Native và là một phần của quy trình thử nghiệm React Native. Sử dụng nó, bạn sẽ có thể đảm bảo rằng framework lựa chọn của bạn được hỗ trợ bởi các phiên bản React Native mới nhất. Điều đó đặc biệt quan trọng trong bối cảnh những thay đổi trong tương lai có thể xảy ra ở framework level.

Trước khi tiếp tục, bạn phải cài đặt Detox. Quá trình này yêu cầu bạn thực hiện một số “native steps” bổ sung trước khi bạn sẵn sàng chạy phần mềm đầu tiên của mình. Làm theo tài liệu chính thức vì các bước có thể thay đổi trong tương lai.

Khi bạn đã cài đặt và định cấu hình Detox thành công, bạn đã sẵn sàng để bắt đầu với việc kiểm tra đầu tiên của mình.

```
it('should display the questions', async () => {
  await device.reloadReactNative();

  await element(by.text(allQuestions[0])).toBeVisible();
});
```

Đoạn trích nhanh được hiển thị ở trên sẽ đảm bảo rằng câu hỏi đầu tiên được hiển thị. Trước khi xác nhận đó được thực thi, bạn nên tải lại phiên bản React Native để đảm bảo rằng không có trạng thái nào trước đó can thiệp vào kết quả

Lưu ý: Khi bạn đang xử lý nhiều phần tử (ví dụ: trong trường hợp của chúng ta - một component hiển thị nhiều câu hỏi), bạn nên chỉ định testID hậu tố với chỉ mục của phần tử để có thể truy vấn một phần tử cụ thể. Điều này cũng như một số kỹ thuật thú vị khác là khuyến nghị Detox chính thức.

Có nhiều đối tượng và kỳ vọng khác nhau có thể giúp bạn xây dựng bộ test của mình theo cách bạn muốn.

Lợi Ích: Bạn có cái nhìn tổng quan hơn về các tính năng và chỉnh sửa mới, có thể tự tin ship và khi các bài kiểm tra có màu xanh lá cây - bạn tiết kiệm thời gian của người khác (nhóm QA)

Một bộ test chất lượng cao sẽ cung cấp đủ bao quát các tính năng cốt lõi cho bạn và đây là một khoản đầu tư vào vận tốc cho sự phát triển của nhóm. Sau tất cả, đủ nhanh thôi chứ đừng nhanh quá. Và tất cả các bài test đều nhằm đảm bảo rằng bạn đang đi đúng hướng.

Cộng đồng React Native đang nỗ lực để làm cho việc testing trở nên dễ thở nhất có thể - cho cả nhóm của bạn và nhóm QA. Nhờ đó, bạn có thể dành nhiều thời gian hơn vào việc đổi mới và làm hài lòng người dùng với các chức năng mới hào nhoáng, đồng thời không phát sinh lỗi và hồi quy lặp đi lặp lại.

2. Hoạt động Continuous Integration (CI) đúng chỗ

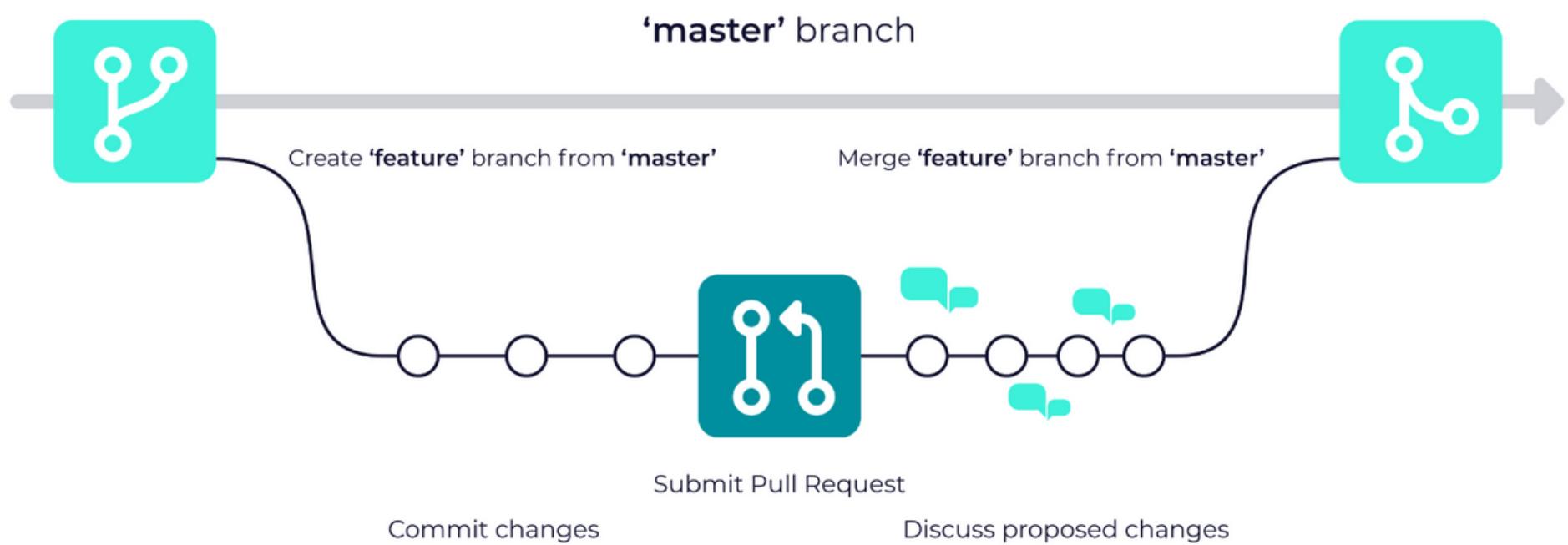
Sử dụng nhà cung cấp CI để cải thiện việc building, testing và phân phối ứng dụng của bạn

Vấn đề: Thiếu CI hoặc không ổn định có nghĩa là vòng phản hồi dài hơn - bạn không biết liệu code của mình có hoạt động hay không và bạn chậm tiến độ với các developer khác

Như đã tìm hiểu từ phần trước, bao gồm code của bạn với các bài test có thể rất hữu ích để tăng độ tin cậy tổng thể cho ứng dụng của bạn. Tuy nhiên, mặc dù việc test sản phẩm là rất quan trọng, nhưng nó không phải là điều kiện tiên quyết duy nhất để shipping của bạn nhanh hơn và tự tin hơn.

Điều quan trọng không kém là bạn phát hiện ra các lỗi quy tiềm nǎng nhanh như thế nào và việc tìm kiếm chúng có phải là một phần trong development lifecycle hàng ngày của bạn hay không. Nói cách khác - tất cả đều đi vào vòng phản hồi

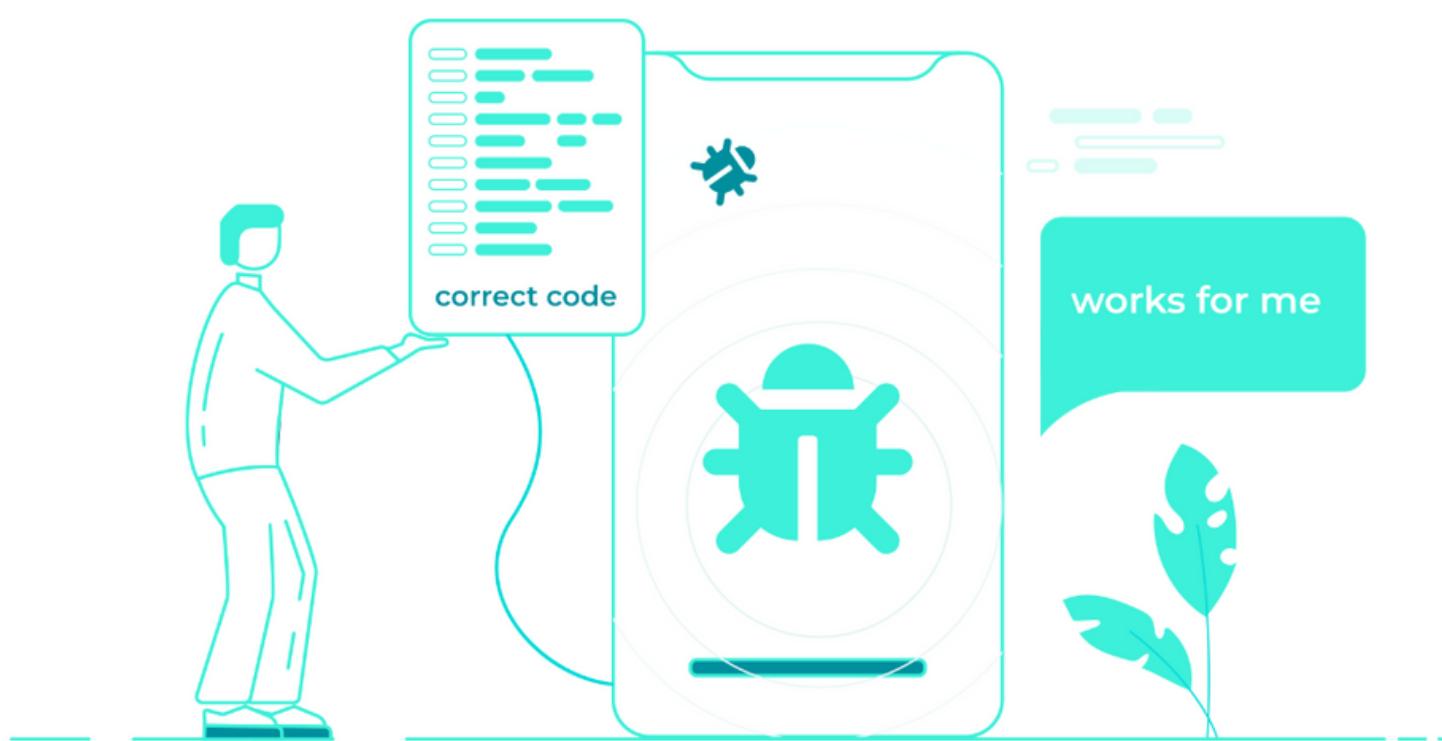
Để có ngũ cảm tốt hơn, hãy cùng xem xét những ngày đầu của quá trình phát triển. Khi bạn bắt đầu, trọng tâm của bạn là shipping lần lặp đầu tiên (MVP) càng nhanh càng tốt. Do đó bạn có thể bỏ qua tầm quan trọng của kiến trúc. Khi thực hiện xong các thay đổi, bạn gửi chúng đến kho lưu trữ, cho các thành viên khác trong nhóm của bạn biết rằng tính năng đã sẵn sàng để được xem xét.



An example of workflow on Github, where changes are proposed in a form of a PR

Mặc dù kỹ thuật này có thể rất hữu ích, nhưng tự nó tiềm ẩn nguy hiểm, đặc biệt là khi nhóm của bạn phát triển về quy mô. Trước khi sẵn sàng chấp nhận PR, bạn không chỉ nên kiểm tra code mà còn nên sao chép nó vào môi trường của bạn và kiểm tra kỹ lưỡng. Vào cuối quá trình đó, có thể những thay đổi được đề xuất đưa ra một hồi quy mà tác giả ban đầu đã không phát hiện ra.

Lý do cho điều đó rất đơn giản - tất cả chúng ta đều có thiết lập, môi trường và cách làm việc khác nhau.



Trong khi cùng dựa trên một cấu hình, một nguyên tắc phát triển tương tự, việc chú ý đến chi tiết sẽ giúp bạn phát triển nhanh hơn, đặc biệt là ở những giai đoạn đầu. Hệ quả là chúng ta sẽ đưa ra những phiên bản mà phá vỡ những bài test.

Việc giới thiệu thành viên mới vào tổ chức của bạn sẽ khó hơn. Bạn không thể gửi và kiểm tra các bài PR và các khoản đóng góp khác nhau khi chúng xảy ra

Nếu bạn đang thử nghiệm các thay đổi của mình theo cách thủ công. Bạn đang làm chậm tốc độ phát triển chung. Rất may, với bộ phương pháp phù hợp và một chút tự động hóa, bạn có thể vượt qua thử thách này một lần và mãi mãi.

Đây là lúc Continuous Integration (CI) phát huy tác dụng. CI là một thực tiễn phát triển trong đó các thay đổi được đề xuất nhóm để các nhóm phát triển đăng ký vào kho lưu trữ ngược dòng nhiều lần trong ngày. Tiếp theo, chúng được xác minh bằng một bản dựng tự động, cho phép nhóm phát hiện sớm các thay đổi

Các bản dựng tự động được thực hiện bởi nhà cung cấp CI dựa trên đám mây chuyên dụng thường tích hợp với nơi bạn lưu trữ code của mình. Hầu hết các nhà cung cấp đám mây hiện có ngày nay đều hỗ trợ Github, một nền tảng do Microsoft sở hữu để cộng tác trong các dự án sử dụng git làm hệ thống kiểm soát phiên bản của họ.

Hệ thống CI kéo các thay đổi theo thời gian thực và thực hiện một loạt các bài kiểm tra đã chọn, để cung cấp cho bạn phản hồi sớm về kết quả của bạn. Cách tiếp cận này sẽ giới thiệu một điểm duy nhất đáng tin cậy để thử nghiệm và cho phép các nhà phát triển có môi trường khác nhau nhận được thông tin thuận tiện và đáng tin cậy

Sử dụng dịch vụ CI, bạn không chỉ có thể kiểm tra code của mình mà còn có thể build phiên bản tài liệu mới cho dự án của bạn, build ứng dụng của bạn và phân phối nó giữa testers hoặc bản phát hành. Kỹ thuật này được gọi là Continuous Deployment và tập trung vào việc tự động hóa các bản phát hành. Nó đã được đề cập sâu hơn trong phần này.

Giải pháp: Sử dụng nhà cung cấp CI (chẳng hạn như CircleCI) để xây dựng ứng dụng của bạn. Chạy tất cả các thử nghiệm bắt buộc và thực hiện các bản phát hành xem trước nếu có thể.

Có rất nhiều nhà cung cấp CI để bạn lựa chọn, trong đó phổ biến nhất là CircleCI, Travis và Github Actions được phát hành gần đây. Với mục đích của phần này, chúng tôi đã chọn CircleCI.

Đây là nhà cung cấp CI mặc định cho React Native và tất cả các dự án được tạo bởi Cộng đồng của nó. Trên thực tế, có một dự án ví dụ chứng minh việc sử dụng CI với React Native. Bạn có thể tìm hiểu thêm về nó ở đây. Chúng tôi sử dụng nó ở phần sau của phần này để trình bày các khái niệm CI khác nhau.

Lưu ý: Nguyên tắc chung là tận dụng những gì mà các dự án React Native / React Native Community đã sử dụng. Đi theo lộ trình đó, bạn có thể đảm bảo có thể khiến nhà cung cấp đã chọn của bạn hoạt động với React Native và những thách thức phổ biến nhất đã được Core Team giải quyết.

Với hầu hết các nhà cung cấp CI, điều cực kỳ quan trọng là phải nghiên cứu các tệp thiết lập của họ trước khi bạn làm bất cứ điều gì khác.

Hãy xem tệp thiết lập mẫu cho CircleCI, được lấy từ ví dụ React Native đã đề cập

```
version: 2
jobs:
  android:
    working_directory: ~/repo
    docker:
      - image: reactnativecommunity/react-native-android
    steps:
      - checkout
      - run: npm i -g envinfo && envinfo
      - run: yarn install
      - run: cd android && chmod +x gradlew && ./gradlew assembleRelease
  workflows:
    version: 2
    build_and_test:
      jobs:
        - android
```

Example of .circleci/config.yml

Cấu trúc là một cú pháp Yaml tiêu chuẩn cho các tệp thiết lập dựa trên văn bản. Bạn có thể tìm hiểu về những điều cơ bản của nó trước khi tiếp tục bất kỳ bước nào.

Lưu ý: Nhiều dịch vụ CI, chẳng hạn như CircleCI hoặc Github Actions, dựa trên vùng chứa Docker và ý tưởng soạn các công việc khác nhau thành quy trình công việc. Github và Github Actions của nó là một ví dụ về nhà cung cấp như vậy. Bạn có thể tìm thấy nhiều điểm tương đồng giữa các dịch vụ đó.

Có ba khối xây dựng quan trọng nhất của các lệnh thiết lập CircleCI: jobs và workflows.

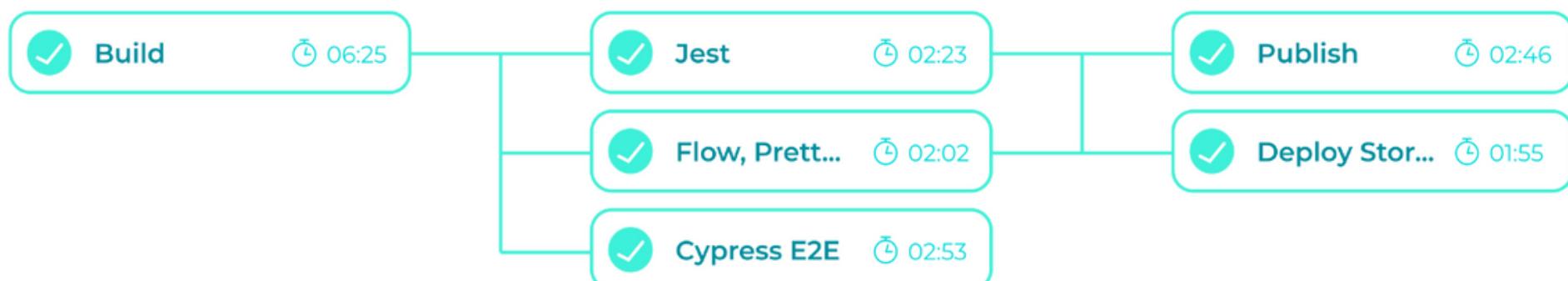
Các lệnh này cũng chỉ là một shell script. Nó được thực thi trong môi trường được chỉ định. Ngoài ra, nó là thứ thực hiện actual job trên đám mây. Nó có thể là bất cứ thứ gì, từ một lệnh đơn giản để cài đặt các dependence của bạn, chẳng hạn như 'yarn install' (nếu bạn đang sử dụng Yarn) đến một lệnh phức tạp hơn một chút `./gradlew assembleDebug` tạo các tệp Android.

Công việc là một chuỗi các lệnh - được mô tả dưới dạng các bước - tập trung vào việc đạt được một mục tiêu xác định. Công việc có thể được chạy trong các môi trường khác nhau, bằng cách chọn một vùng chứa Docker thích hợp.

Ví dụ: bạn có thể sử dụng Node container nếu bạn chỉ cần chạy các bài test đơn vị React của mình. Kết quả container sẽ nhỏ hơn, ít dependence hơn và cài đặt nhanh hơn. Nếu bạn muốn xây dựng một ứng dụng React Native trên đám mây, bạn có thể chọn một container khác, ví dụ: với Android NDK / SDK hoặc hệ điều hành sử dụng OS X để xây dựng nền tảng của Apple.

Lưu ý: Để giúp bạn chọn container sử dụng khi chạy các thử nghiệm React Native, nhóm đã chuẩn bị react-native-android Docker container bao gồm cả Node và Android dependencies cần thiết để thực hiện các thử nghiệm và build Android.

Để thực hiện một Job, nó phải được gán cho một Workflow. Theo mặc định, Jobs sẽ được thực hiện song song trong workflow, nhưng điều này có thể được thay đổi bằng cách chỉ định các yêu cầu cho Job.

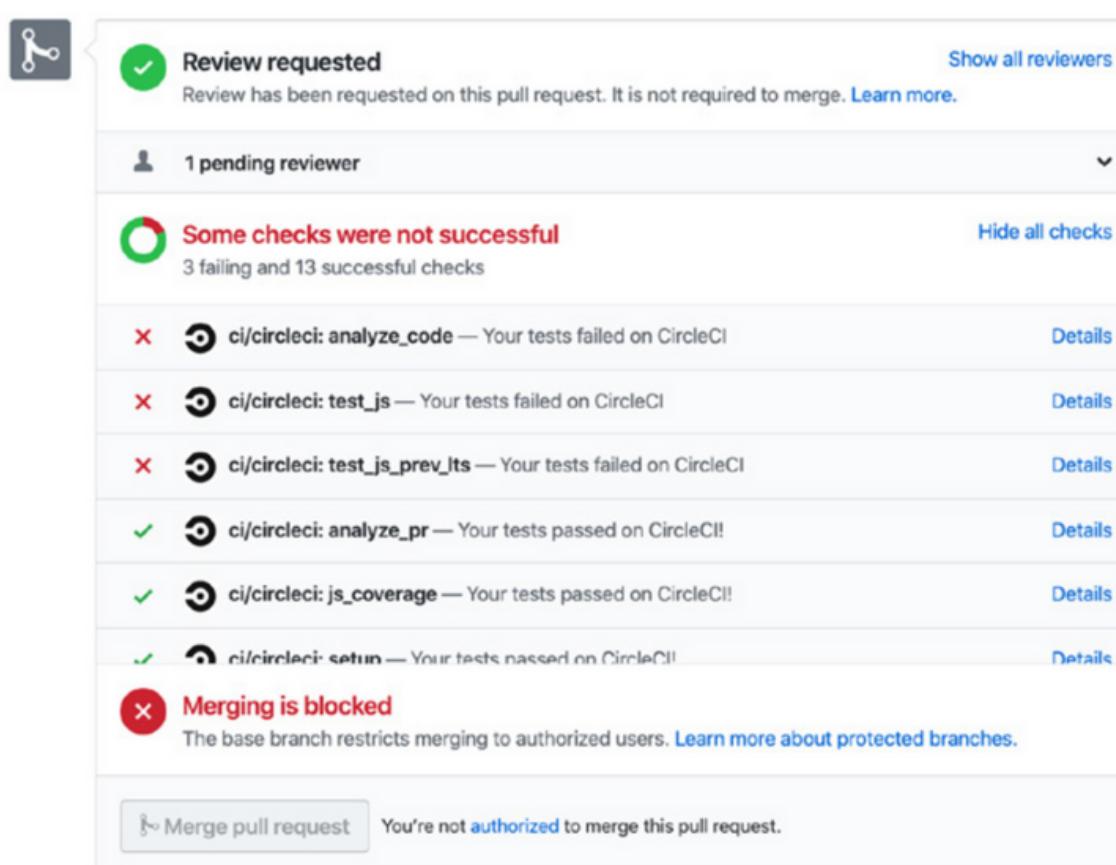


Bạn cũng có thể sửa đổi lịch trình thực thi công việc bằng cách thêm bộ lọc, vì vậy, ví dụ: một công việc triển khai sẽ chỉ chạy nếu các thay đổi trong code đề cập đến một nhánh chính.

Bạn có thể xác định workflows cho các mục đích khác nhau, ví dụ: một mặt các thử nghiệm sẽ chạy sau khi mở PR và mặt khác để triển khai phiên bản mới của ứng dụng. Đây là những gì React Native làm để thỉnh thoảng tự động phát hành các phiên bản mới của nó.

Lợi ích: Bạn nhận được phản hồi sớm về các tính năng được bổ sung, nhanh chóng phát hiện các regression. Ngoài ra, bạn không lãng phí thời gian của các developer khác vào việc thử nghiệm các thay đổi không hoạt động.

Nhà cung cấp CI hoạt động và được thiết lập phù hợp có thể giúp bạn tiết kiệm rất nhiều thời gian khi gửi phiên bản mới của ứng dụng.



Github UI reporting the status of CircleCI jobs, an example taken from React Native repository

Bằng cách phát hiện trước các lỗi, bạn có thể giảm bớt nỗ lực cần thiết để xem xét các PR và bảo vệ sản phẩm của bạn khỏi regressions và bugs có thể trực tiếp làm giảm thu nhập của bạn.

3. Đừng ngại ship nhanh với Continuous Deployment

Thiết lập một continuous deployment setup để cung cấp các tính năng mới và xác minh các lỗi nghiêm trọng nhanh hơn.

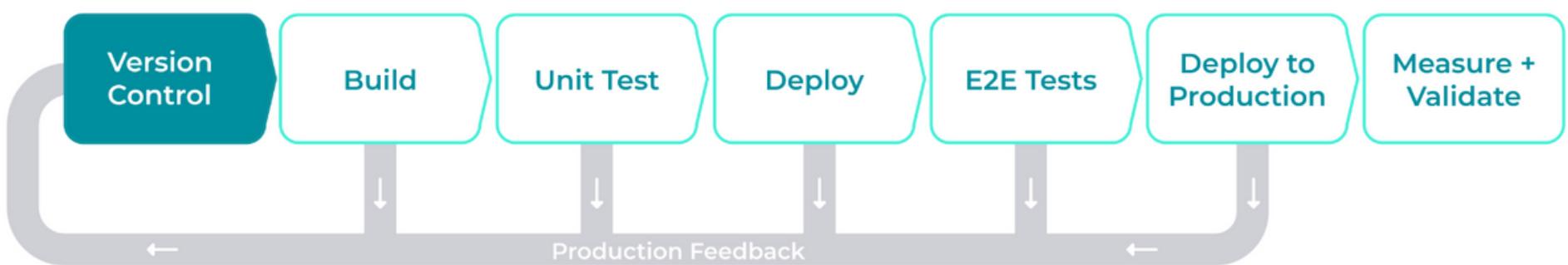
Vấn đề: Việc xây dựng và phân phối ứng dụng của bạn theo cách thủ công là một quá trình phức tạp và tốn thời gian

Như bạn đã biết trong phần trước, tự động hóa các phần quan trọng của vòng đời phát triển đó thể giúp bạn cải thiện tốc độ phát triển tổng thể và bảo mật. Vòng phản hồi ngắn hơn, nhóm của bạn có thể tái thiết trên chính sản phẩm càng nhanh.

Tuy nhiên, testing và development chỉ là một phần của các hoạt động mà bạn phải thực hiện khi làm việc trên một sản phẩm. Một bước quan trọng khác là deployment - building và distributing ứng dụng để sản xuất. Hầu hết thời gian, quá trình này là thủ công.

Lý do cho điều đó rất đơn giản - việc triển khai cần thời gian để set up và phức tạp hơn nhiều so với việc chỉ chạy thử nghiệm trên đám mây. Ví dụ: trên iOS, Xcode tự động thiết lập nhiều cài đặt và certificates. Điều này đảm bảo trải nghiệm tốt hơn cho những người đang làm việc trên một ứng dụng native. Các developer đã quen với cách tiếp cận như vậy thường cảm thấy khó khăn khi chuyển việc triển khai lên đám mây và set up những thứ như certificates theo cách thủ công

Nhược điểm lớn nhất của phương pháp thủ công là tốn thời gian và không mở rộng được quy mô. Do đó, các nhóm không đầu tư vào các cải tiến cho quy trình này sẽ kết thúc việc phát hành phần mềm của họ với tốc độ chậm hơn.



Continuous Deployment là một chiến lược trong đó phần mềm được phát hành thường xuyên thông qua một tập hợp các script tự động. Nó nhằm mục đích building, testing và releasing phần mềm với tốc độ và tần suất lớn hơn. Cách tiếp cận này giúp giảm chi phí, thời gian và rủi ro thực hiện các thay đổi bằng cách cho phép cập nhật nhiều hơn cho các ứng dụng trong sản xuất.

Bạn không shipping các tính năng mới và các bản sửa lỗi nhanh như bạn muốn

Building và distributing ứng dụng của bạn theo cách thủ công sẽ làm chậm quá trình phát triển của bạn bất kể nhóm của bạn lớn như thế nào. Ngay cả trong các nhóm nhỏ khoảng 5 người, automated build pipelines giúp công việc của mọi người dễ dàng hơn và giảm bớt các giao tiếp không cần thiết. Điều này đặc biệt quan trọng đối với các công ty ở xa.

Continuous Deployment cũng cho phép bạn giới thiệu các tiêu chuẩn và thực tiễn tốt nhất tập trung vào việc cải thiện hiệu suất tổng thể của ứng dụng. Một số chúng đã được thảo luận trước đây trong hướng dẫn này. Với tất cả các bước cần thiết để triển khai, rất dễ dàng để đảm bảo rằng tất cả các bản phát hành đều được thực hiện theo cùng một cách và enrolled các tiêu chuẩn của toàn công ty.

Giải pháp: Thiết lập một continuous deployment setup (dựa trên `fastlane`) để build và generates the changelog. Ship cho người dùng của bạn ngay lập tức

Khi nói đến tự động hóa deployment các ứng dụng di động, có hai cách để thực hiện

Cách đầu tiên là viết một bộ tập lệnh từ đầu bằng cách tương tác trực tiếp với `xcode` và `gradle`. Thật không may, có sự khác biệt đáng kể giữa công cụ của Android và iOS và không nhiều developers có đủ kinh nghiệm để xử lý tự động hóa điều này. Trên hết, iOS phức tạp hơn nhiều so với Android do các chính sách phân phối và code signing nâng cao. Và như đã nói trước đây, nếu bạn đang làm điều đó theo cách thủ công, ngay cả Xcode cũng không thể giúp bạn bằng cách thực hiện điều kỳ diệu của nó.



Cách thứ hai là sử dụng một công cụ có sẵn trong đó các developer đã xử lý phần lớn các trường hợp. Công cụ yêu thích của chúng tôi là fastlane - một tập hợp các tiện ích modular được viết bằng Ruby cho phép bạn xây dựng các ứng dụng iOS và Android của mình bằng cách viết một bộ hướng dẫn vào tệp thiết lập.

Sau khi bạn đã tạo thành công các tệp nhị phân của mình, đã đến lúc triển khai nó đến bước cuối cùng. Một lần nữa, bạn có thể tải các tệp lên một dịch vụ mong muốn (ví dụ: App Store) theo cách thủ công hoặc sử dụng một công cụ sẽ giải quyết việc đó cho bạn. Vì những lý do tương tự như trước đây, chúng tôi muốn sử dụng một giải pháp hiện có - trong trường hợp này là AppCenter của Microsoft.



AppCenter là một dịch vụ đám mây với công cụ để automation và deployment ứng dụng của bạn. Ưu điểm lớn nhất của nó là nhiều cài đặt có thể được thiết lập từ giao diện đồ họa. Việc set up App Store và Play Store deployment theo cách này sẽ dễ dàng hơn nhiều so với việc tải lên từ dòng lệnh.

Mục đích của phần này là chúng tôi sẽ sử dụng fastlane và AppCenter trong đường ống CircleCI để tự động hóa hoàn toàn quá trình phân phối ứng dụng đến người dùng cuối cùng

Lưu ý: Việc mô tả chi tiết và những thứ ngoài lề của setup sẽ khiến bài viết này quá dài. Đó là lý do tại sao chúng tôi chọn chỉ tham khảo một vài tài liệu cụ thể. Mục tiêu của chúng tôi là cung cấp cho bạn cái nhìn tổng quan chứ không phải hướng dẫn từng bước, vì thiết lập cuối cùng sẽ khác nhau đối với mỗi dự án.

Thiết lập Fastlane

Trước khi đi vào chi tiết cho Android và iOS, bạn phải đảm bảo rằng Fastlane đã được cài đặt và được thiết lập trên thiết bị.

Tiếp theo, bạn phải chạy lệnh init trong dự án React Native. Chúng tôi sẽ chạy lệnh `fastlane` hai lần, từ mỗi thư mục native. Điều này là do React Native thực sự là hai ứng dụng riêng biệt ở low-level.

```
> cd ./ios && fastlane init  
> cd ./android && fastlane init
```

Kết quả là, lệnh này sẽ tạo các tệp setup trong cả hai thư mục ‘ios’ và ‘android’. Tệp chính trong mỗi thư mục sẽ được gọi là Fastfile và đó là nơi tất cả các lane sẽ được thiết lập. Trong danh pháp fastlane, một lane giống như một workflow - một phần nhóm các hoạt động low-level triển khai ứng dụng của bạn

Các hoạt động low-level có thể được thực hiện bằng cách calling actions - các hoạt động fastlane được xác định trước giúp đơn giản hóa workflow của bạn. Chúng tôi sẽ cho bạn thấy chúng hoạt động như thế nào trong phần tiếp theo.

Thiết lập Fastlane trên Android

Bây giờ bạn đã set up thành công fastlane trong các dự án, bạn đã sẵn sàng để tự động hóa deployment ứng dụng Android. Để làm được như vậy, bạn có thể chọn một action cụ thể của Android - trong trường hợp này là gradle. Gradle là một action cho phép bạn đạt được kết quả tương tự như với Android Gradle được sử dụng độc lập.

Lane của chúng tôi sử dụng gradle action trước tiên để làm sạch thư mục xây dựng (build folder), sau đó lắp ráp APK với signature dựa trên các thông số đã được truyền vào.

```
default_platform(:android)  
  
project_dir = File.expand_path("../", Dir.pwd)  
  
platform :android do  
  lane :build do |options|  
    if (ENV['ANDROID_KEYSTORE_PASSWORD'] && ENV['ANDROID_KEY_PASSWORD'])
```

```

properties = {
    "RELEASE_STORE_PASSWORD" => ENV['ANDROID_KEYSTORE_PASSWORD'],
    "RELEASE_KEY_PASSWORD" => ENV['ANDROID_KEY_PASSWORD']
}

end

gradle(
    task: "clean",
    project_dir: project_dir,
    properties: properties,
    print_command: false
)

gradle(
    task: "assemble",
    build_type: "Release",
    project_dir: project_dir,
    properties: properties,
    print_command: false
)
end

end

```

Part of the android/fastlane/Fastfile that defines Android lane, named build

Bạn sẽ có thể chạy lane build bằng cách triển khai:

```
$ cd ./android && fastlane build
```

Điều này sẽ tạo ra signed Android apk.

Lưu ý: Đừng quên đặt các biến môi trường để truy cập cửa hàng từ khóa. Chúng là `RELEASE_STORE_PASSWORD` và `RELEASE_KEY_PASSWORD` đã được thiết lập trong ví dụ được trình bày ở trên.

Thiết lập Fastlane trên iOS

Với việc Android build được tự động hóa, bạn đã sẵn sàng chuyển sang iOS ngay bây giờ. Như chúng ta đã thảo luận trước đó, iOS phức tạp hơn một chút do hồ sơ chứng nhận và cấp phép. Chúng được Apple thiết kế để tăng tính bảo mật. May mắn thay, fastlane ships với một số actions chuyên dụng giúp chúng tôi vượt qua những phức tạp này.

Bạn có thể bắt đầu với *match* action. Nó giúp quản lý và phân phối chứng chỉ iOS và hồ sơ cấp phép giữa các thành viên trong nhóm của bạn. Bạn có thể đọc ý tưởng đằng sau *match* trong [codesigning_guide concept](#).

Nói một cách đơn giản, *match* sẽ quan tâm đến việc thiết lập thiết bị của bạn theo cách mà nó có thể tạo thành công một ứng dụng sẽ được xác thực và chấp nhận bởi các máy chủ của Apple.

Lưu ý: Trước khi bạn tiếp tục, hãy đảm bảo rằng init của bạn phù hợp với dự án của bạn. Nó sẽ tạo ra certificates cần thiết và lưu trữ chúng trong một kho lưu trữ trung tâm, nơi nhóm của bạn và các công cụ tự động hóa khác có thể tìm nạp chúng

Một action khác mà bạn có thể sử dụng ngoài *match* là *gym*. *Gym* tương tự như Gradle action theo cách mà nó thực hiện việc build ứng dụng của bạn. Để làm như vậy, nó sử dụng certificates đã tìm nạp trước đó và signs các cài đặt từ *match*.

```
default_platform(:ios)

  ios_directory = File.expand_path("../", Dir.pwd)
  base_path = File.expand_path("../", ios_directory)
  ios_workspace_path = "#{ios_directory}/YOUR_WORKSPACE.xcworkspace"
  ios_output_dir = File.expand_path('./output', base_path)
  ios_app_id = 'com.example'
  ios_app_scheme = 'MyScheme'

  before_all do
    if is_ci? && FastlaneCore::Helper.mac?
      setup_circle_ci
    end
  end
```

```

platform :ios do
  lane :build do |options|
    match(
      type: options[:type],
      readonly: true,
      app_identifier: ios_app_id,
    )

    cocoapods(podfile: ios_directory)

    gym(
      configuration: "Release",
      scheme: ios_app_scheme,
      export_method: "ad-hoc",
      workspace: ios_workspace_path,
      output_directory: ios_output_dir,
      clean: true,
      xcargs: "-UseModernBuildSystem=NO"
    )
  end
end

```

Part of ios/fastlane/Fastfile where iOS lane is defined

Bạn có thể chạy lane build bằng cách chạy lệnh tương tự như đối với Android:

```
$ cd ./ios && fastlane build
```

Điều này cũng sẽ tạo ra một ứng dụng iOS ngay bây giờ.

Triển khai mã nhị phân

Bây giờ bạn đã tự động hóa quá trình build, bạn có thể tự động hóa phần cuối cùng của quá trình - deployment. Để làm như vậy, bạn có thể sử dụng App Center, như đã thảo luận trước đó trong hướng dẫn này.

Lưu ý: Bạn phải tạo tài khoản trong App Center, các ứng dụng dành cho Android và iOS trong trang tổng quan và tạo code thông báo truy cập cho từng ứng dụng.

Bạn cũng sẽ cần một Fastlane đặc biệt đặt vào hành động thích hợp cho toolbelt. Để làm như vậy, có thể chạy lệnh `fastlane add_plugin appcenter`.

Khi bạn đã hoàn tất việc thiết lập các dự án của mình, bạn đã sẵn sàng để tiếp tục viết lane sẽ package các tệp nhị phân được produce và tải chúng lên App Center.

```
lane :deploy do
  build

  appcenter_upload(
    api_token: ENV['APPCENTER_TOKEN'],
    owner_name: "ORGANIZATION_OR_USER_NAME",
    owner_type: "organization", # 'user' | 'organization'
    app_name: "YOUR_APP_NAME",
    file: "#{ios_output_dir}/YOUR_WORKSPACE.ipa",
    notify_testers: true
  )
end
```

Part of ios/fastlane/Fastfile with upload lane

Bây giờ đã đến lúc deploy ứng dụng bằng cách thực hiện deploy lanei từ máy cục bộ của bạn.

Tích hợp với CircleCI

Sử dụng tất cả các lệnh này, bạn có thể build và distribute ứng dụng cục bộ. Bây giờ, bạn có thể thiết lập máy chủ CI của mình để nó hoạt động tương tự trên mọi commit to master. Để làm như vậy, bạn sẽ sử dụng CircleCI - nhà cung cấp mà chúng tôi đã sử dụng trong suốt hướng dẫn này.

Lưu ý: Chạy Fastlane trên máy chủ CI thường yêu cầu một số setup bổ sung. Tham khảo tài liệu chính thức để hiểu rõ hơn sự khác biệt giữa cài đặt trong môi trường cục bộ và môi trường CI.

Để deploy ứng dụng từ CircleCI, bạn có thể thiết lập workflow chuyên dụng vì nó sẽ tập trung vào việc building và deploying ứng dụng. Nó sẽ chứa một công việc duy nhất, được gọi là `deploy_ios`, sẽ thực thi lệnh `fastlane`.

```
deploy_ios:  
  macos:  
    xcode: '11.3.1'  
    working_directory: ~/CI-CD  
    steps:  
      - checkout  
      - attach_workspace:  
          at: ~/CI-CD  
      - run: HOMEBREW_NO_AUTO_UPDATE=1 brew install watchman  
      - run: bundle install  
      - run: cd ios && bundle exec fastlane deploy  
  
workflows:  
  version: 2  
  deploy:  
    jobs:  
      - deploy_ios
```

Part of CircleCI configuration that executes Fastlane build lane

Pipeline cho Android sẽ trông khá giống nhau. Sự khác biệt chính sẽ là người thực thi. Thay vì macOS, nên sử dụng hình ảnh docker `reactnativecommunity/react-native-android`

Lưu ý: Đây chỉ là cách sử dụng mẫu trong CircleCI. Trong trường hợp của bạn, có thể có ý nghĩa hơn khi xác định filters và dependencies vào các công việc khác, để đảm bảo `deploy_ios` được chạy đúng thời điểm.

Bạn có thể sửa đổi hoặc tham số hóa các lane được trình bày để sử dụng chúng cho các loại deploys khác, chẳng hạn như dành riêng cho nền tảng App Store. Để tìm hiểu chi tiết về các trường hợp nâng cao như vậy, hãy làm quen với tài liệu Fastlane chính thức.

Lợi ích: Vòng phản hồi ngắn cùng với các nightly hay weekly build cho phép bạn xác minh các tính năng nhanh hơn và ship các bugs nghiêm trọng thường xuyên hơn.

Với việc deployment tự động, bạn không còn lãng phí thời gian cho các manual build và gửi các phần mềm tạo tác đến các thiết bị test hoặc cửa hàng ứng dụng. Các stakeholder của bạn có thể xác minh các tính năng nhanh hơn và rút ngắn vòng phản hồi hơn nữa. Với các regular build, bạn sẽ có thể dễ dàng catch hoặc ship sửa lỗi bất kì bugs nghiêm trọng nào.

4. Gửi OTA (Over-The-Air) trong trường hợp khẩn cấp

Gửi các bản sửa lỗi và bản cập nhật quan trọng ngay lập tức thông qua OTA.

Gửi các bản sửa lỗi và bản cập nhật quan trọng ngay lập tức thông qua OTA.

Mô hình truyền thống để gửi các bản cập nhật trên thiết bị di động về cơ bản khác với mô hình chúng ta biết khi viết các ứng dụng JavaScript cho các nền tảng khác. Không giống như web, triển khai trên thiết bị di động phức tạp hơn nhiều và đi kèm với khả năng bảo mật tốt hơn. Chúng ta đã nói về điều đó một cách chi tiết trong phần trước tập trung vào CI / CD

Vậy nó có ý nghĩa gì đối với doanh nghiệp của bạn? Mọi bản cập nhật, bất kể developers của bạn ship nhanh đến mức nào, thường sẽ phải đợi một thời gian trong khi nhóm App Store và Play Store xem xét sản phẩm của bạn theo chính sách và phương pháp hay nhất của họ.

Quá trình này đặc biệt khó khăn ở tất cả các nền tảng của Apple, nơi các ứng dụng thường bị gỡ xuống hoặc bị từ chối do không tuân theo các chính sách nhất định hoặc đáp ứng tiêu chuẩn bắt buộc cho giao diện người dùng. Rất may, nguy cơ ứng dụng của bạn bị từ chối với React Native được giảm xuống mức tối thiểu, vì bạn đang làm việc trên phần JavaScript của ứng dụng. Nhóm React Native Core đảm bảo rằng tất cả các thay đổi được thực hiện đối với framework không ảnh hưởng đến sự thành công của việc gửi đơn đăng ký của bạn.

Kết quả là, quá trình gửi sẽ mất một thời gian. Và nếu bạn sắp gửi một bản cập nhật quan trọng, thì mỗi phút đều có giá trị.

May mắn thay, với React Native, có thể tự động gửi các thay đổi JavaScript của bạn trực tiếp đến người dùng của bạn, bỏ qua quá trình xem xét trên App Store. Kỹ thuật này thường được gọi là over the air update. Nó cho phép bạn thay đổi giao diện ứng dụng của mình ngay lập tức, dành cho tất cả người dùng, theo kỹ thuật mà bạn đã chọn.

**Khi các bug nghiêm trọng xảy ra - thời gian trở nên rất quan trọng.
Đừng đợi Apple và Google xem xét ứng dụng của bạn.**

Nếu ứng dụng của bạn chưa sẵn sàng OTA, bạn có nguy cơ bị để lại một bug nghiêm trọng trên nhiều thiết bị, miễn là Apple / Google xem xét sản phẩm của bạn và cho phép nó được phân phối.

Mặc dù thời gian xem xét đã ngắn hơn nhiều, nhưng nó vẫn là một giải pháp tốt để có thể khôi phục ngay lập tức sau khi phát hiện lỗi trong quá trình thử nghiệm và đưa vào sản xuất.

Giải pháp: Triển khai cập nhật OTA với App Center / CodePush

Như đã đề cập trước đó, React Native đã sẵn sàng với OTA. Có nghĩa là các lựa chọn kiến trúc và thiết kế của React Native cho phép những cập nhật đó có thể thực hiện được. Tuy nhiên, nó không ship kèm với cơ sở hạ tầng để thực hiện được các hoạt động như vậy. Để làm như vậy, bạn sẽ cần phải tích hợp một dịch vụ của bên thứ 3 có cơ sở hạ tầng riêng để làm như vậy.

Công cụ phổ biến và được sử dụng rộng rãi nhất để cập nhật OTA là CodePush, một dịch vụ hiện là một phần của App Center của Microsoft.

Lưu ý: Bạn phải tạo một tài khoản trong App Center để tiếp tục.. Tùy chọn OTA sẽ hiển thị trong ứng dụng bạn đã tạo. Thông thường, bạn nên sử dụng cả OTA và release capabilities từ App Center để thiết lập dễ dàng hơn.

Thiết lập native side

Để tích hợp CodePush vào ứng dụng của bạn, có các bước bắt buộc bạn phải làm theo đối với iOS và Android tương ứng. Chúng tôi quyết định liên kết đến các hướng dẫn chính thức thay vì chỉ có các bước ở đây vì chúng bao gồm native code bổ sung để áp dụng và điều đó rất có thể thay đổi trong những tháng tới.

Thiết lập JavaScript side

Khi bạn set up dịch vụ ở native side, bạn có thể sử dụng API JavaScript để kích hoạt các bản cập nhật và xác định khi nào chúng sẽ xảy ra. Cách đơn giản nhất cho phép tìm nạp các bản cập nhật khi khởi động ứng dụng là sử dụng trình `codePush` wrapper và wrap component chính của bạn

```
import codePush from "react-native-code-push";  
  
class MyApp extends Component {}  
  
MyApp = codePush(MyApp);
```

Basic CodePush integration

Nếu bạn đã thực hiện tất cả các thay đổi ở native side, ứng dụng của bạn hiện đã sẵn sàng OTA.

Đối với các trường hợp nâng cao hơn, bạn cũng có thể thay đổi cài đặt mặc định về thời điểm kiểm tra các bản cập nhật và thời điểm tải xuống và áp dụng chúng. Ví dụ: bạn có thể buộc CodePush kiểm tra các bản cập nhật mỗi khi ứng dụng được đưa trở lại foreground và cài đặt các bản cập nhật trong bản lý lịch tiếp theo.

Đoạn code sơ đồ sau minh họa giải pháp như vậy

```
class MyApp extends Component { }  
MyApp = codePush({  
  updateDialog: true,  
  checkFrequency: codePush.CheckFrequency.ON_APP_RESUME,
```

```
installMode: codePush.InstallMode.ON_NEXT_RESUME  
}) (MyApp);
```

Custom CodePush setup

Shipping các bản cập nhật cho ứng dụng

Sau khi thiết lập CodePush trên cả JavaScript và native side của React Native, đã đến lúc khởi chạy bản cập nhật và cho phép khách hàng mới của bạn enjoy nó. Để làm như vậy, chúng ta có thể thực hiện việc này từ dòng lệnh, bằng cách sử dụng App Center CLI

```
npm install -g appcenter-cli  
appcenter login
```

Và sau đó, phát hành một lệnh để bundle các tệp và nội dung React Native và gửi chúng lên đám mây:

```
appcenter codepush release-react -a <ownerName>/<appName>
```

Khi các bước này hoàn tất, tất cả người dùng đang chạy ứng dụng của bạn sẽ nhận được bản cập nhật bằng cách sử dụng trải nghiệm bạn đã thiết lập trong phần trước.

Lưu ý: Trước khi xuất bản bản phát hành CodePush mới, bạn sẽ phải tạo một ứng dụng trong App Center dashboard. Điều đó sẽ cung cấp cho bạn `ownerName` và `appName` mà bạn đang tìm kiếm. Như đã nói trước đây, bạn có thể thực hiện việc này qua UI bằng cách truy cập App Center hoặc bằng cách sử dụng App Center CLI

Lợi ích: Ship các bản sửa lỗi quan trọng và một số nội dung ngay lập tức cho người dùng

Với các bản cập nhật OTA được tích hợp vào ứng dụng của bạn, bạn có thể gửi các bản cập nhật JavaScript cho tất cả người dùng của mình chỉ trong vài phút. Khả năng này có thể rất quan trọng để sửa các significant bug hoặc gửi các bản vá lỗi tức thì.

Ví dụ: có thể xảy ra trường hợp backend của bạn ngừng hoạt động và gây ra sự cố khi khởi động. Đó có thể là một lỗi được xử lý sai - bạn chưa bao giờ gặp lỗi backend trong quá trình phát triển và quên xử lý các trường hợp như vậy.

Cách khắc phục tiềm năng cho vấn đề này rất đơn giản - chỉ cần hiển thị thông báo dự phòng và thông báo cho người dùng về sự cố là đủ. Trong khi quá trình development sẽ mất khoảng một giờ, quá trình cập nhật và xem xét thực tế có thể mất hàng giờ nếu không phải vài ngày.

Với các bản cập nhật OTA được set up, bạn có thể react với điều này trong vài phút mà không phải chịu rủi ro rằng trải nghiệm người dùng xấu sẽ ảnh hưởng đến phần lớn người dùng.

LỜI CẢM ƠN

Đây là toàn bộ series của "The Ultimate Guide to React Native Optimization". Cảm ơn bạn đã dành thời gian để đọc tài liệu này.

Hy vọng những chia sẻ trên sẽ giúp bạn cải thiện được hiệu quả cũng như nâng cao kỹ năng trong công việc lập trình của mình.

Hãy lưu lại cuốn sách này cho bản thân và chia sẻ cho cộng đồng nếu bạn thấy thật sự hữu ích. Việc này sẽ tạo thêm động lực để team **200Lab Edu** tiếp tục cung cấp thêm các tài liệu bổ ích trong tương lai nhé.

-*Phần 1: Cải thiện hiệu năng bằng cách hiểu rõ chi tiết bên trong React Native.*

-*Phần 2: Cải thiện hiệu suất bằng cách sử dụng các tính năng React Native tiên tiến nhất.*

-*Phần 3: Cách để phát hành ứng dụng nhanh hơn trong môi trường phát triển ổn định*

**Câu hỏi thắc mắc và đăng ký nhận các bản dịch khác liên hệ tại:

Fanpage: <https://www.facebook.com/edu.200lab>

Website: <https://edu.200lab.io>