

The Ultimate Guide to React Native Optimization

Tập 1



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

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.

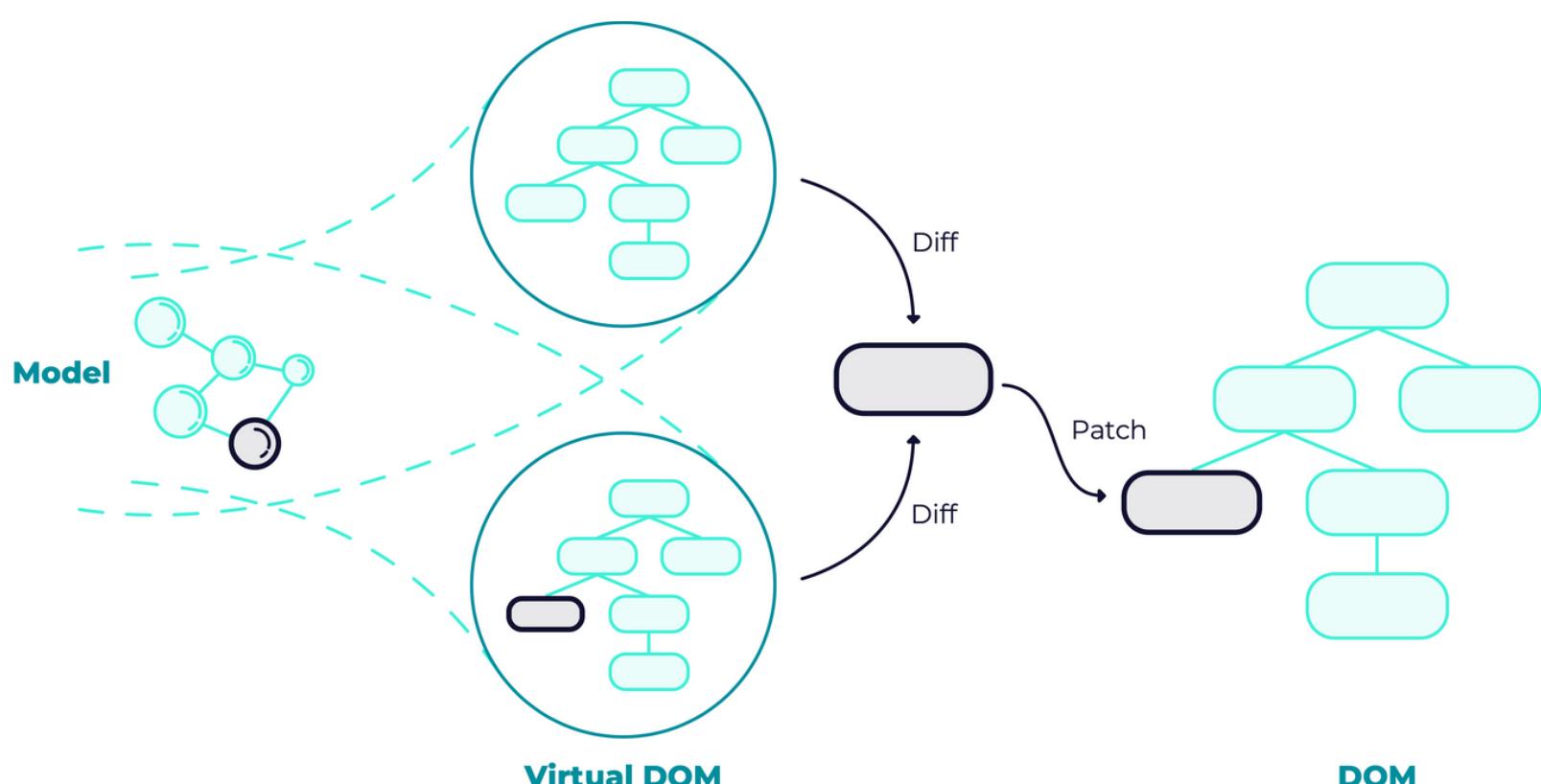
1. Chú ý đến việc UI re-renders

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.

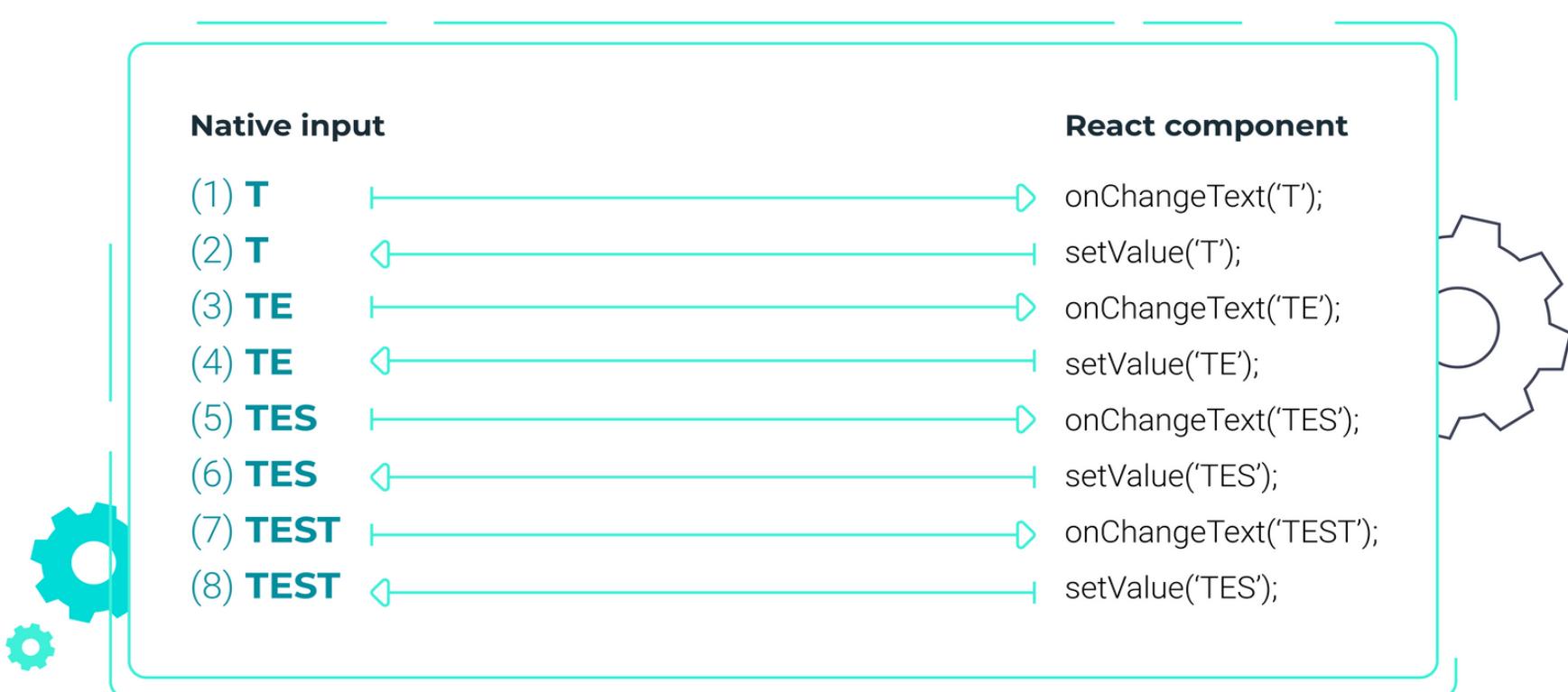


Diagram that shows what happens while typing TEST

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

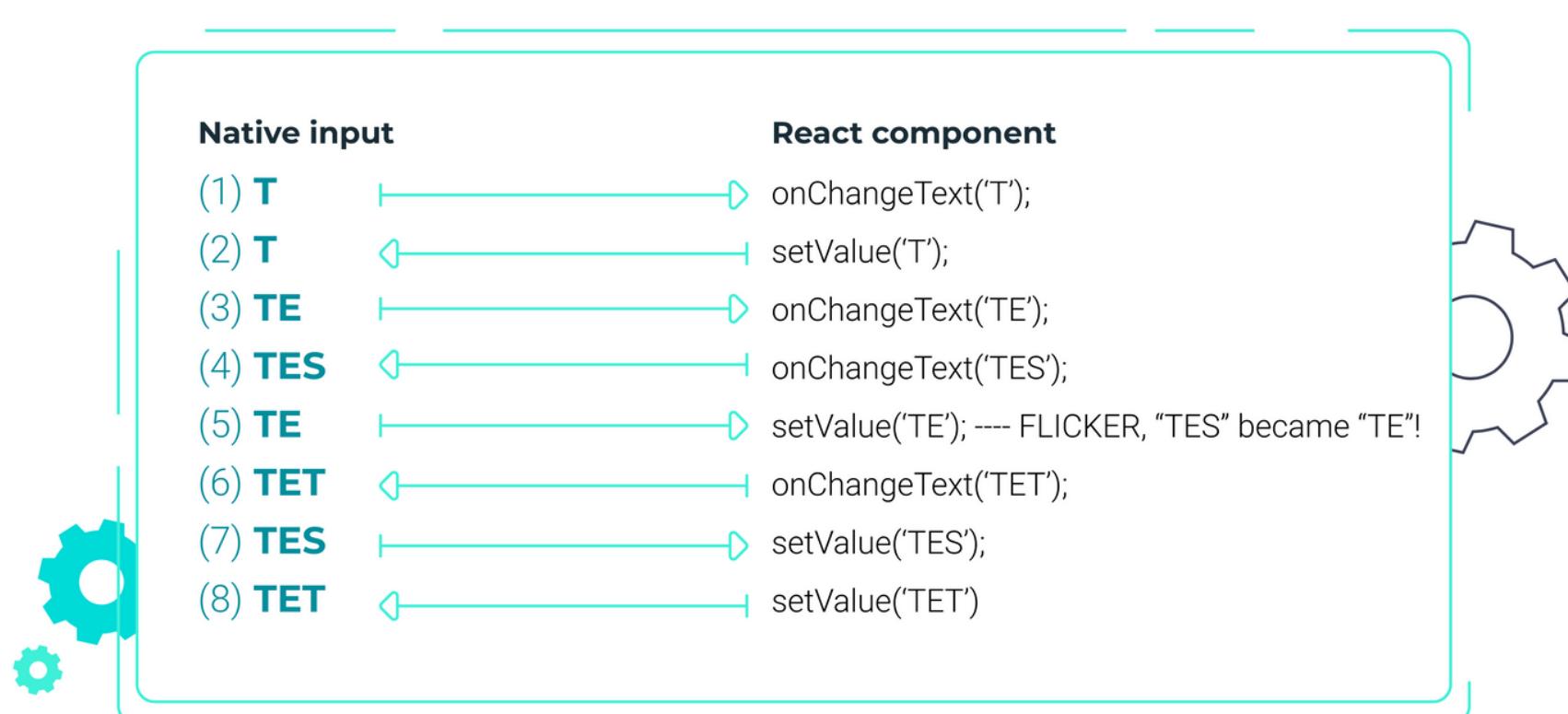


Diagram that shows what happens while typing TEST too fast

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 khiến 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.

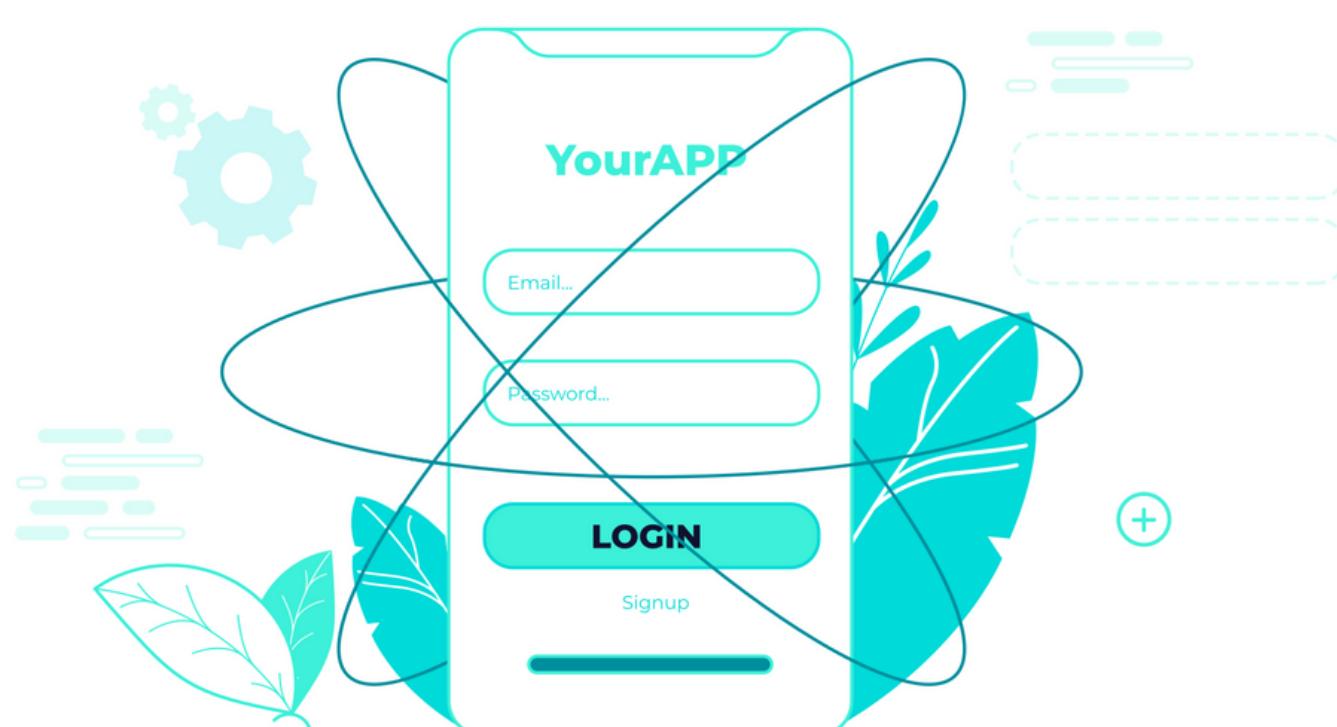
2. Sử dụng các Component chuyên dụng cho các bố cục (layout) nhất định

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.

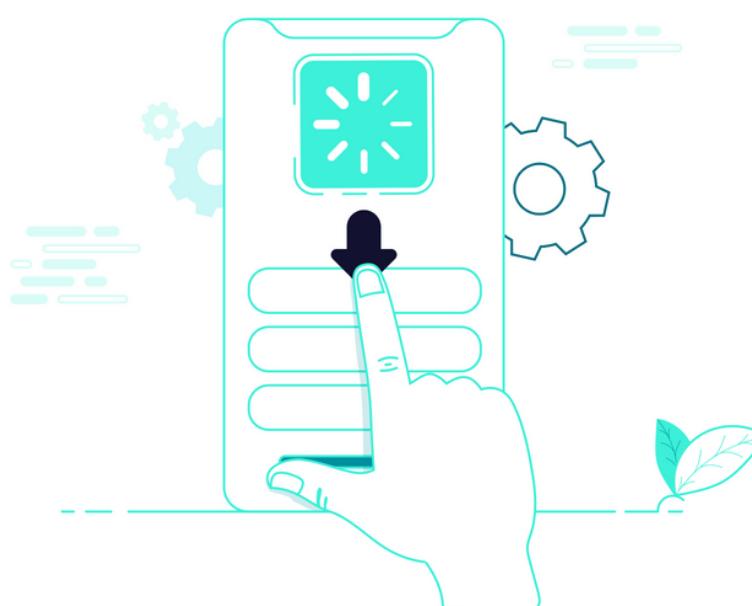


Ngoài các primitive component, React Native còn cung cấp thêm cho chúng ta các higher-order component để phục vụ cho việc tối ưu ứng dụng trong các mục đích nhất định.

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.

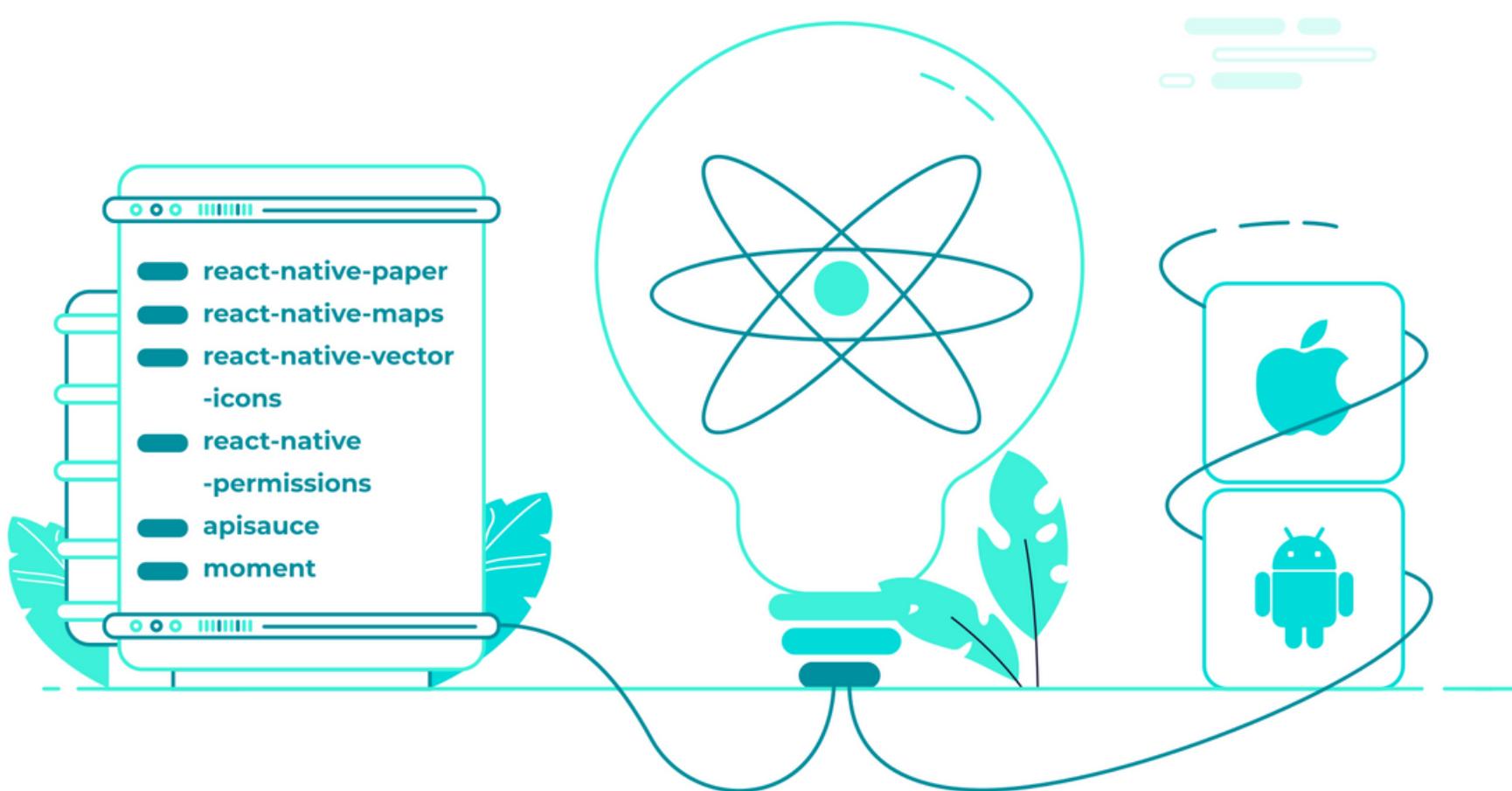
Đồng thời, bạn cũng đã tiết kiệm được rất nhiều thời gian cho việc xây dựng lại hầu hết các mẫu giao diện phổ biến từ đầu. Sticky Section Headers, Pull To Refresh - các component này đã được hỗ trợ mặc định nếu bạn chọn sử dụng cùng với FlatList.

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.

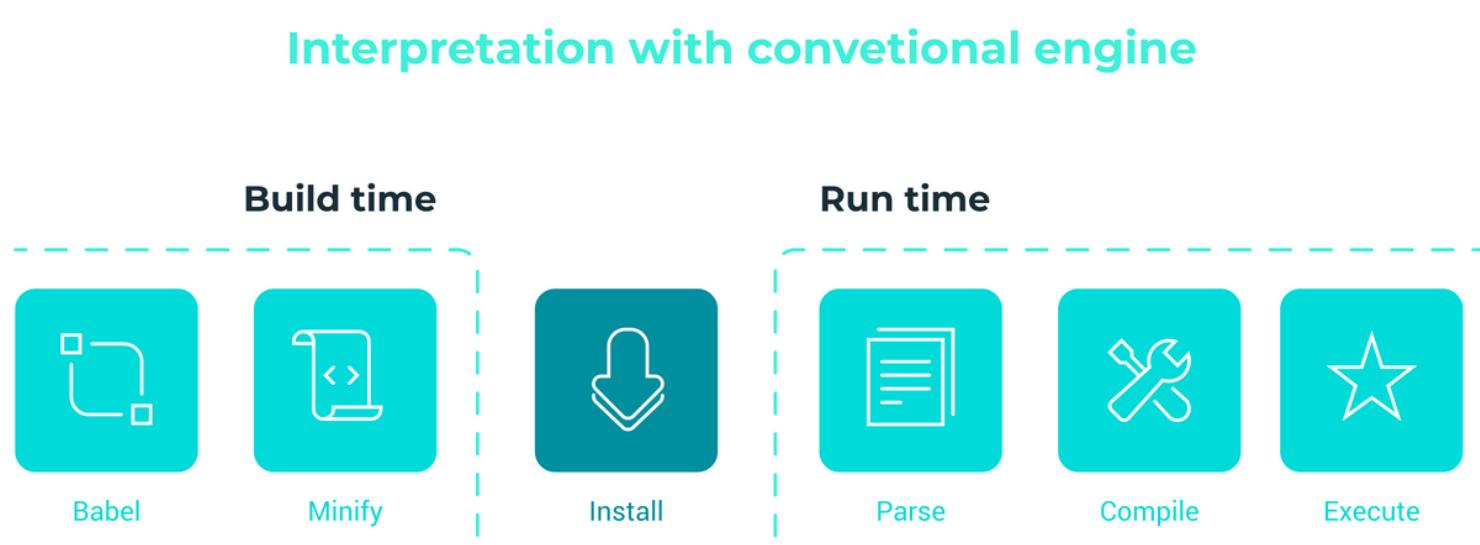
Sự thật là việc phát triển mobile app có những khác biệt từ nền tảng và chúng có những nguyên tắc riêng. Ví dụ: trong khi kích thước của assets là quan trọng trong phát triển web, thì nó không quan trọng trong React Native, vì chúng được đặt trong filesystem.

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.

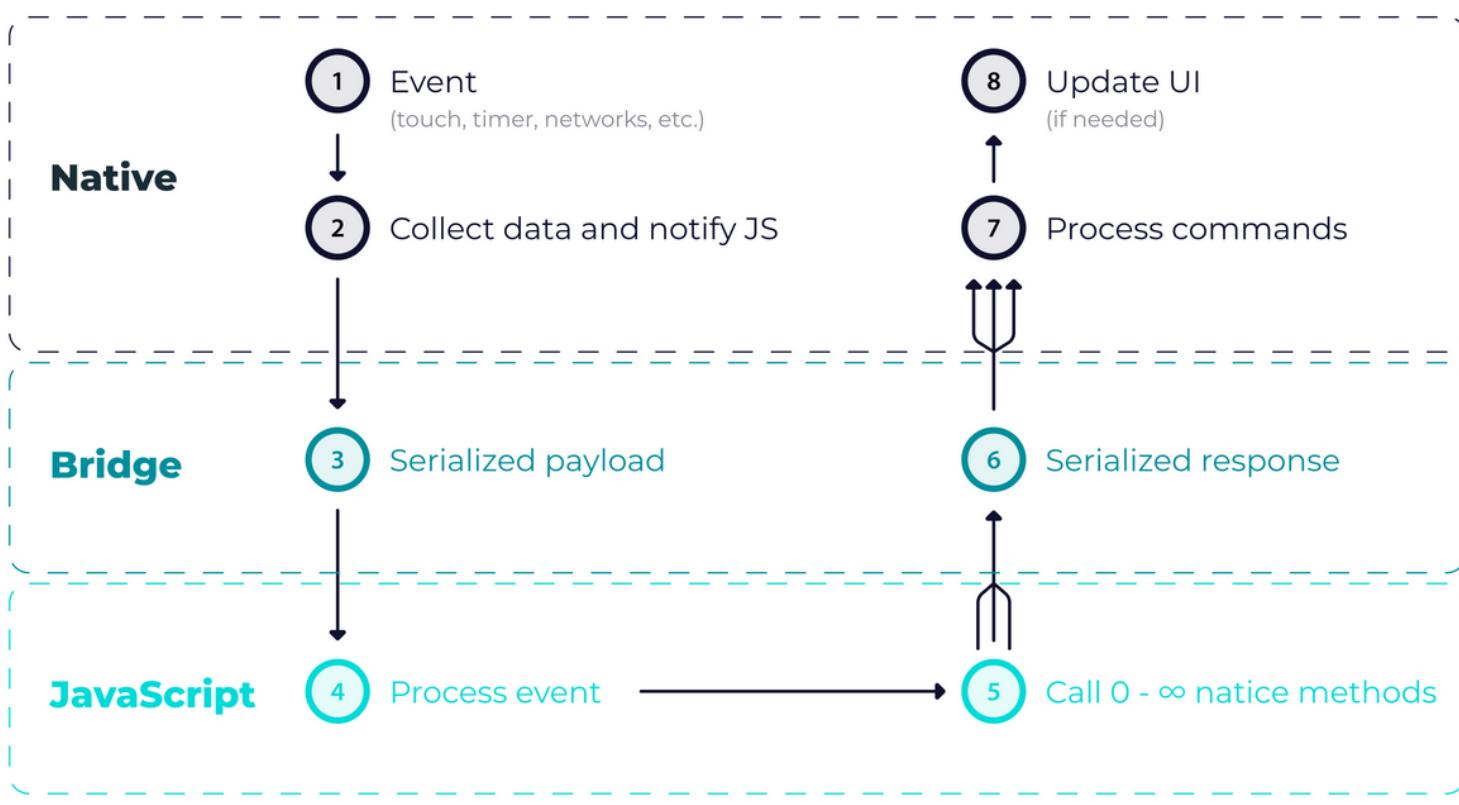
5. Tìm sự cân bằng giữa native và JavaScript

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.

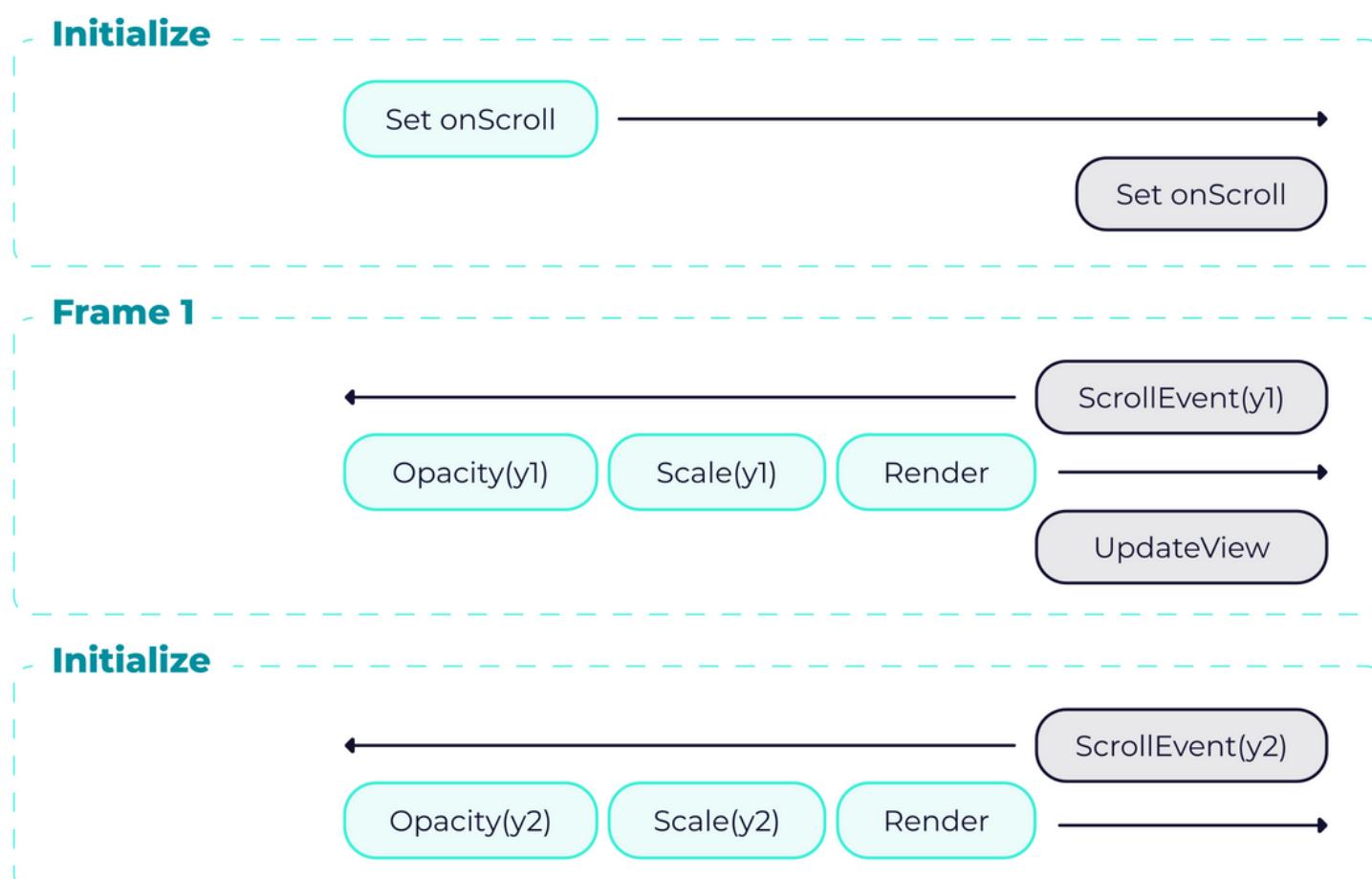
Khi làm việc với React Native, hầu hết thời gian bạn sẽ dùng JavaScript. Tuy nhiên, có những trường hợp bạn cần phải viết một ít mã native. Ví dụ: bạn đang làm việc với SDK của bên thứ 3 và nó chưa hỗ trợ React Native chính thức. Trong trường hợp đó, bạn cần tạo một native module chứa các native methods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

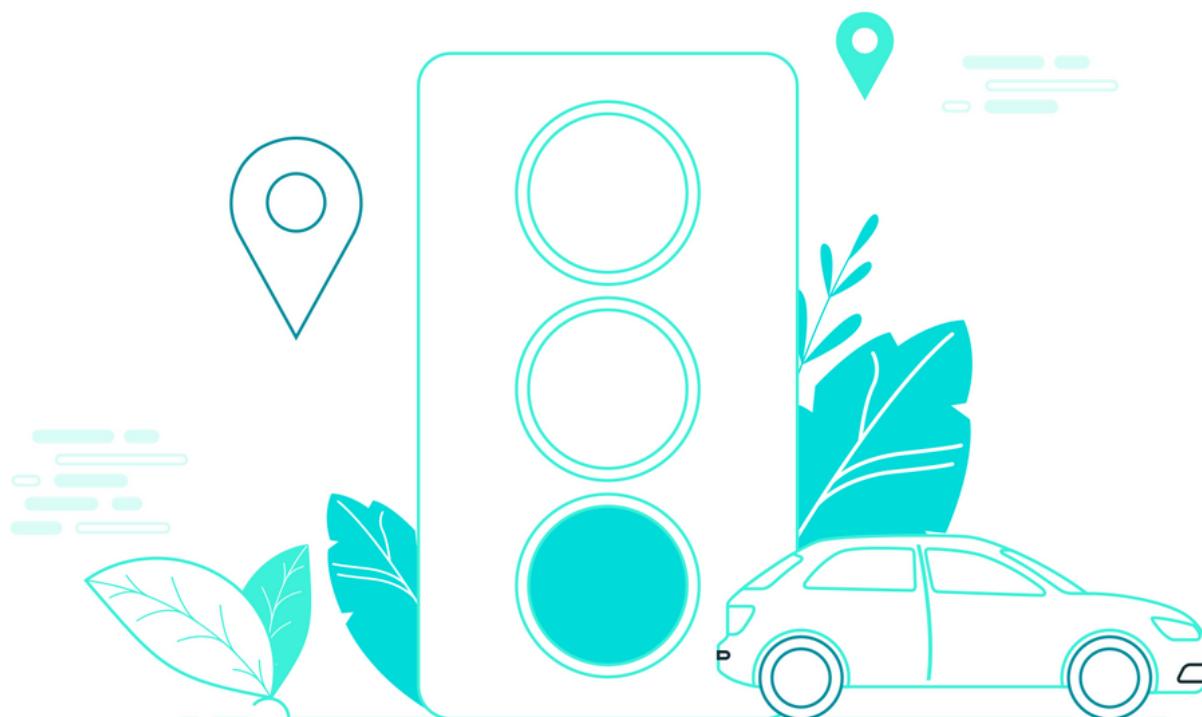
Xem thêm tại: <https://snack.expo.io/GUFPWl8BD>

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.

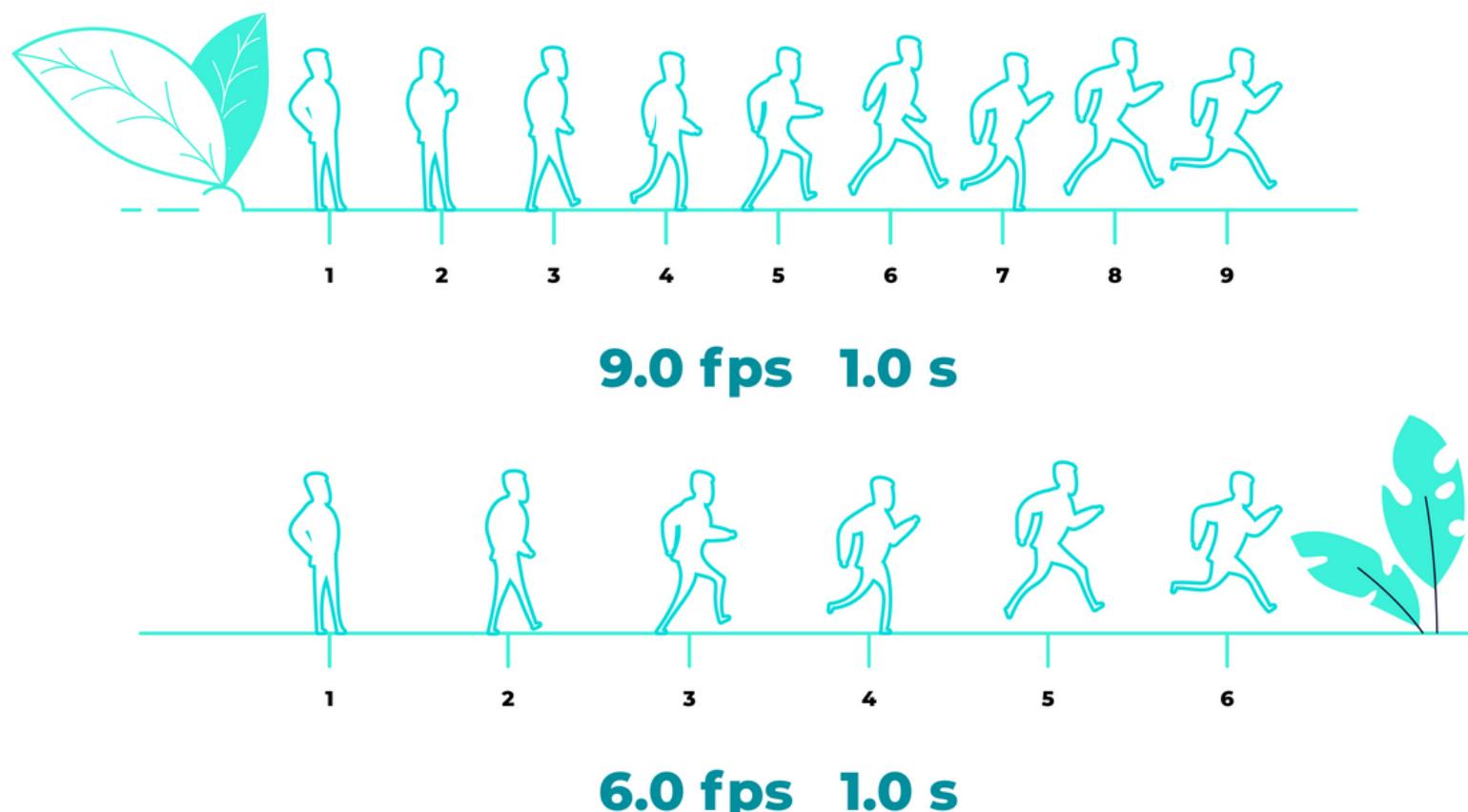
Nắm vững cách tiếp cận này để phát triển các native modules giúp cho nhiều nhà phát triển JavaScript có thể dễ dàng mở rộng ứng dụng của họ với các chức năng bổ sung mà không cần chuyên sâu về Objective-C hay Java.

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.

LỜI CẢM ƠN

Đây là phần 1 của series "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. Series vẫn còn 2 phần cuối với nội dung rất bổ ích.

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 cái tài liệu bổ ích trong tương lai nhé.

-*Phần 2: Cải thiện hiệu suất bằng cách sử dụng các tính năng React Native tân 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 bản dịch phần 2 và phần 3 của quyển ebook này liên hệ tại:

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

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