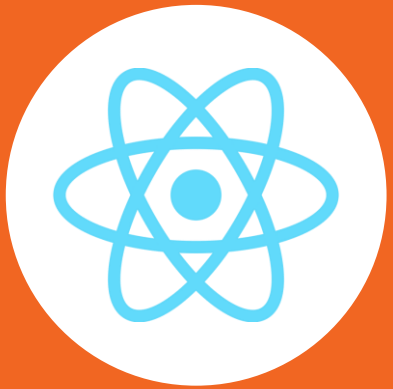
FPT Education

FPT POLYTECHNIC



LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

BÀI 3: ANIMATION TRONG REACT NATIVE

PHẦN 1: GIỚI THIỆU, CÀI ĐẶT THƯ VIỆN REANIMATED VÀ TÌM HIỂU WORKLETS

https://caodang.fpt.edu.vn/

Mục tiêu

- [ ] Hiểu và sử dụng hiệu quả thư viện Reanimated
- [ ] Tối ưu hóa hiệu suất cho animation
- [ ] Tương tác người dùng tốt hơn
- [ ] Xây dựng các hiệu ứng hoạt hình phức tạp
- [ ] Nắm vững kiến thức về JavaScript và React Native

Giới thiệu animation

- Animation trong React Native cho phép bạn tạo ra các hiệu ứng hoạt hình động trong ứng dụng di động của mình. Nó giúp cải thiện trải nghiệm người dùng và tạo ra giao diện tương tác và hấp dẫn hơn. React Native cung cấp các API animation mạnh mẽ để tạo ra các hiệu ứng như di chuyển, phóng to, thu nhỏ, xoay, đổi màu và nhiều hơn nữa.
- React native cũng đã cung cấp sẵn Animated để bạn có thể tạo animation cho mình mà không cần cài đặt bất kỳ thư viện nào. Nhưng tôi khuyến khích bạn không nên sử dụng thư viện này. Tạo animation với Animated sẽ tạo ra các hiệu ứng không mượt mà, gây chậm hiệu suất ứng dụng và làm giảm trải nghiệm người dùng. Vậy nên ở bài viết này tôi sẽ giới thiệu cho bạn thư viện Reanimated, nó sẽ tạo cho một một animation trơn tru, mạnh mẽ và mang đến cho người dùng trải nghiệm tốt nhất.

Reanimated

- Reanimated là một thư viện React Native cho phép tạo các ? và tương tác mượt mà chạy trên luồng UI.
- Lý do ra đời thư viện Reanimated
  - Trong các ứng dụng React Native, mã ứng dụng được thực thi bên ngoài luồng chính của ứng dụng. Đây là một trong những yếu tố chính trong kiến trúc của React Native và giúp ngăn ngừa giảm khung hình trong trường hợp JavaScript có một số công việc nặng phải làm. Thật không may, thiết kế này không hoạt động tốt khi nói đến các tương tác theo hướng sự kiện. Khi tương tác với màn hình cảm ứng, người dùng mong đợi các hiệu ứng trên màn hình sẽ ngay lập tức. Nếu các bản cập nhật đang diễn ra trong một luồng riêng biệt, thường xảy ra trường hợp các thay đổi được thực hiện trong chuỗi JavaScript không thể được thực thi trong cùng

?một frame. Trong React Native, theo mặc định, tất cả các bản cập nhật đều bị trì hoãn ít nhất một khung vì giao tiếp giữa các luồng giao diện người dùng và JavaScript không đồng bộ và luồng giao diện người dùng không bao giờ đợi luồng JavaScript kết thúc các sự kiện xử lý. Ngoài độ trễ với JavaScript đóng nhiều vai trò như chạy react diffing và cập nhật, thực thi logic nghiệp vụ của ứng dụng, xử lý yêu cầu mạng, v.v., Thường thì các sự kiện không thể được xử lý ngay lập tức, do đó gây ra sự chậm trễ đáng kể hơn. Reanimated nhằm mục đích cung cấp các cách giảm tải hoạt ảnh và logic xử lý sự kiện khỏi luồng JavaScript và vào luồng giao diện người dùng. Điều này đạt được bằng cách xác định các worklet Reanimated – các đoạn mã JavaScript nhỏ có thể được chuyển sang một máy ảo JavaScript riêng biệt và được thực thi đồng bộ trên luồng giao diện người dùng.

Điều này cho phép phản hồi các sự kiện cảm ứng ngay lập tức và cập nhật giao diện người dùng trong cùng một khung khi sự kiện xảy ra mà không phải lo lắng về tải được đặt trên luồng JavaScript chính.

- Ở lần soạn bài giảng này, phiên bản Reanimated 3.x đang là phiên bản mới nhất. Nên chúng ta sẽ cùng đi tìm hiểu và xây dựng animation ở phiên bản 3.x này.
- Vậy Reanimated 3.x này có gì mới?
  - Reanimated 3 tập trung vào việc cải thiện tính ổn định và hiệu suất. Phiên bản này sử dụng API Reanimated v2 mà bạn biết và yêu thích, vẫn giữ các worklet và share value

Reanimated

- Nó đi kèm với việc viết lại toàn bộ cơ chế Shared Value và Layout Animation và Shared element transition
- Bên cạnh nhiều cải tiến và tính năng, Reanimated 3.x cũng giới thiệu hỗ trợ cho Kiến trúc mới. Khi ứng dụng của bạn (hoặc thư viện bạn đang sử dụng) sử dụng API Reanimated v1, nó sẽ không hoạt động với Reanimated 3.x.

Bạn có thể truy cập trang document này để biết rõ hơn về chi tiết
https://docs.swmansion.com/react-native-reanimated/docs/

Cài đặt reanimated

- Cài đặt package

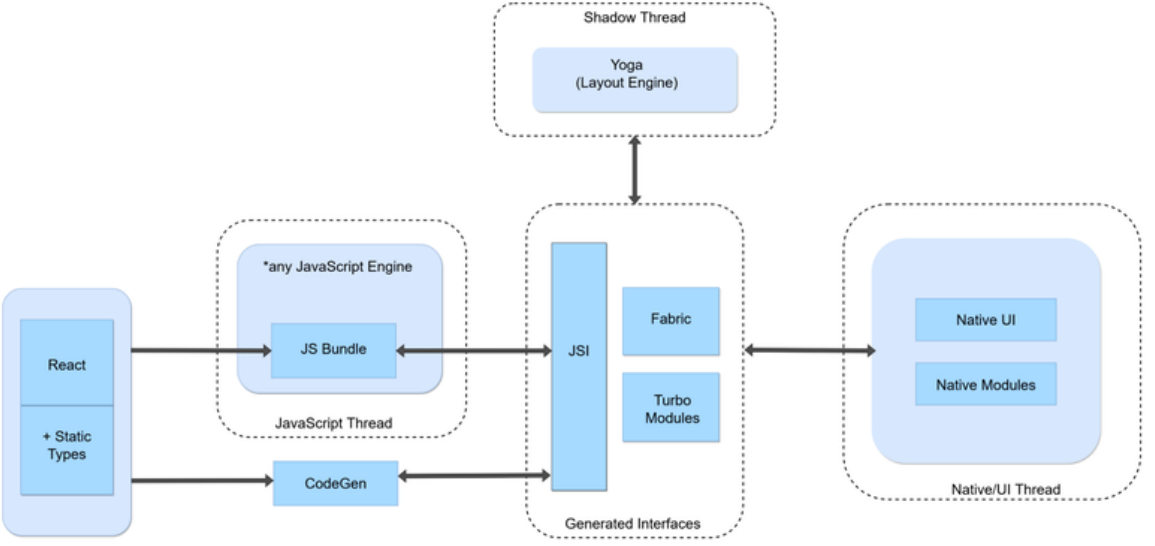
```
npm i react-native-reanimated
```

- Babel plugin
  - Thêm plugin babel của Reanimated vào file babel.config.js

```
module.exports = {
  presets: [
    ...
  ],
  plugins: [
    ...
    'react-native-reanimated/plugin',
  ],
};
```

Architect trong react native

- Trước khi đi vào phần tiếp theo bạn phải hiểu được Kiến trúc của React Native



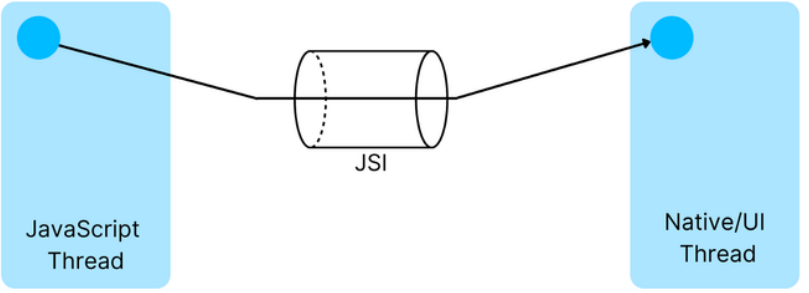
Kiến trúc React Native

3 Thread trong react native

- Khi bạn khởi chạy một React Native app, thì xin lưu ý là phần code JS của bạn sẽ được package (được gọi là JS Bundle) và được tách riêng ra với phần Native Code (Android/IOS)
- Một ứng dụng React Native sẽ chạy trên 3 thread
  - JS thread (Javascript thread/ Main thread): Được sử dụng bởi JS Engine, dùng để chạy JS bundle
  - Native/UI thread: Được sử dụng để khởi chạy các native module, các tiến trình render UI, animations, gesture handle, ...
  - Shadow thread: Được sử dụng để tính toán Layout của element trước khi render ra màn hình.

Vậy thì JSI hoạt động như nào?

- Nhờ JSI, các Native method sẽ được tiếp xúc với JS qua C++ Host Objects. Lúc này JS có thể giữ các tham chiếu (reference) đến các objects này và gọi trực tiếp đến các method qua các tham chiếu. Dễ hình dung hơn thì tương tự trên web, JS code có thể giữ một tham chiếu đến bất kỳ DOM element nào, và gọi phương thức qua nó



Worklets

- Mục tiêu của worklet là để chúng ta chạy các đoạn mã JavaScript mà khi chúng ta cập nhật thuộc tính view hoặc phản ứng với các sự kiện trên UI thread. Với Reanimated chúng ta tạo ra một ngữ cảnh JS phụ trên luồng giao diện người dùng mà sau đó có thể chạy các hàm JavaScript. Điều duy nhất cần thiết là chức năng đó phải có 'worklet' ở trên cùng đoạn code:

```
function someWorklet(greeting) {
  'worklet';
  console.log('Chào! Tôi đang chạy trên UI thread');
}
```

Worklets

- Các hàm là một cấu trúc tuyệt vời cho nhu cầu của chúng ta vì bạn có thể giao tiếp với mã mà chúng chạy bằng cách truyền các đối số. Mỗi hàm worklet có thể được chạy trên main React Native thread nếu bạn chỉ gọi hàm đó trong mã của mình hoặc bạn có thể thực thi nó trên luồng UI bằng runOnUI. Lưu ý rằng việc thực thi giao diện người dùng không đồng bộ từ quan điểm của người gọi. Khi bạn truyền các đối số, chúng sẽ được sao chép vào UI JS context.

```
function someWorklet(greeting: string) {
  'worklet';
  console.log('Chào! Tôi đang chạy trên UI thread ', greeting);
}

function onPress() {
  runOnUI(someWorklet)('Howdy');
}
```

Worklets

- Ngoài các đối số, các hàm cũng nắm bắt bối cảnh nơi chúng được xác định. Vì vậy, khi bạn có một biến được xác định bên ngoài phạm vi worklet nhưng được sử dụng bên trong một worklet, nó cũng sẽ được sao chép cùng với các đối số và bạn có thể sử dụng nó (chúng tôi gọi nó là capture params đã cho):

```
const width = 135.5;

function otherWorklet() {
  'worklet';
  console.log('Captured width is', width);
}
```

Worklets

- Worklet có thể capture (hoặc take as arguments) từ các worklet khác, trong trường hợp đó khi được gọi, chúng sẽ thực thi đồng bộ trên luồng UI:

```
function returningWorklet() {
  'worklet';
  return "I'm back";
}

function someWorklet() {
  'worklet';
  let what = returningWorklet();
  console.log('On the UI thread, other worklet says', what);
}
```

Worklets

- Trên thực tế, console.log không được định nghĩa trong UI JS context, mà là một tham chiếu đến một phương thức React Native JS context cung cấp. Vì vậy, khi chúng ta sử dụng console.log trong các ví dụ trước, nó thực sự được thực thi trên React Native thread. Để một hàm chạy trên UI thread gọi được hàm chạy trên JS thread bạn sử dụng runOnJS

```
function callback(text) {
  console.log('Running on the RN thread', text);
}

function someWorklet() {
  'worklet';
  console.log("I'm on UI but can call methods from the RN thread");
  runOnJS(callback)('can pass arguments too');
}
```

Worklets

- Trong thực tế, khi viết animation và tương tác với Reanimated, bạn sẽ hiếm khi cần tạo các worklet bằng cách sử dụng chỉ thị 'worklet'. Thay vào đó, thứ bạn sẽ sử dụng hầu hết là các worklet có thể được tạo bởi một trong các hook từ Reanimated API, ví dụ: useAnimatedStyle, useDerivedValue, useAnimatedGestureHandler, v.v. Khi sử dụng một trong các hook được liệt kê trong Tham chiếu API Reanimated, chúng tôi sẽ tự động phát hiện ra rằng phương thức được cung cấp là một worklet và không yêu cầu phải chỉ định lệnh. Phương thức được cung cấp cho hook sẽ được chuyển thành một worklet và được thực thi tự động trên chuỗi giao diện người dùng.

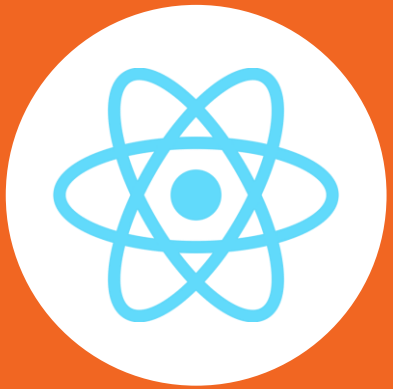
- Ví dụ sử dụng useAnimatedStyle

```
const style = useAnimatedStyle(() => {
  console.log('Running on the UI thread');
  return {
    opacity: 0.5,
  };
});
```

- Đọc đến đây thì cơ bản bạn đã biết Reanimated dùng để làm gì? Và biết cách sử dụng worklet, runOnUI để chạy code JS trên UI thread, sử dụng runOnJS để gọi hàm JS trong UI thread
- Để tìm hiểu sâu hơn về Reanimated thì các bạn chúng ta cùng tìm hiểu ở phần sau nhé

FPT Education

FPT POLYTECHNIC



LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

BÀI 3: ANIMATION TRONG REACT NATIVE

PHẦN 2: SHARED VALUE, ANIMATION VÀ EVENTS TRONG REANIMATED

Mục tiêu

- [ ] Lưu trữ giá trị cho animation bằng shared value
- [ ] Tạo các animation với reanimated
- [ ] Tương tác với các event con trỏ lên animation

Shared Values

- Shared Values là một trong những khái niệm cơ bản đằng sau Reanimated. Nếu bạn đã quen thuộc với API Animated của React Native, bạn có thể so sánh chúng với Animated.Values. Chúng phục vụ một mục đích tương tự là mang dữ liệu 'animateable', cung cấp khái niệm về tính phản ứng và animation động. Chúng ta sẽ thảo luận về từng vai trò chính của Shared Values trong các phần dưới đây. Cuối cùng, chúng tôi trình bày tổng quan ngắn gọn về sự khác biệt giữa Shared Values và Animated.Value cho độc giả quen thuộc với animated API.

Shared Values

- Một trong những mục tiêu chính của Shared Values là cung cấp một khái niệm về bộ nhớ được chia sẻ trong Reanimated. Như bạn có thể đã học trong bài viết về worklet, Reanimated chạy mã hoạt hình trong một luồng riêng biệt bằng cách sử dụng ngữ cảnh JS VM riêng biệt. Giá trị được chia sẻ cho phép duy trì tham chiếu đến dữ liệu có thể thay đổi có thể được đọc và sửa đổi một cách an toàn trên các luồng đó.
- Các đối tượng Shared Values đóng vai trò là tham chiếu đến các phần dữ liệu được chia sẻ có thể được truy cập và sửa đổi bằng thuộc tính .value của chúng. Điều quan trọng cần nhớ là cho dù bạn muốn truy cập hoặc cập nhật dữ liệu được chia sẻ, bạn nên sử dụng thuộc tính .value

Shared Values

- Để tạo tham chiếu Shared Value, bạn nên sử dụng hook useSharedValue:

```
const sharedVal = useSharedValue(3.1415);
```

- Hook Shared Values lấy một đối số duy nhất là giá trị ban đầu của Shared Values. Đây có thể là bất kỳ dữ liệu nguyên thủy hoặc lồng nhau nào như object, array, number, string or boolean.
- Để cập nhật Shared Value từ thread React Native hoặc từ worklet chạy trên UI thread, bạn nên đặt gọi .value của Shared Value đó.

```
const changeSharedValue = () => {
  sharedVal.value = Math.random();
};
```

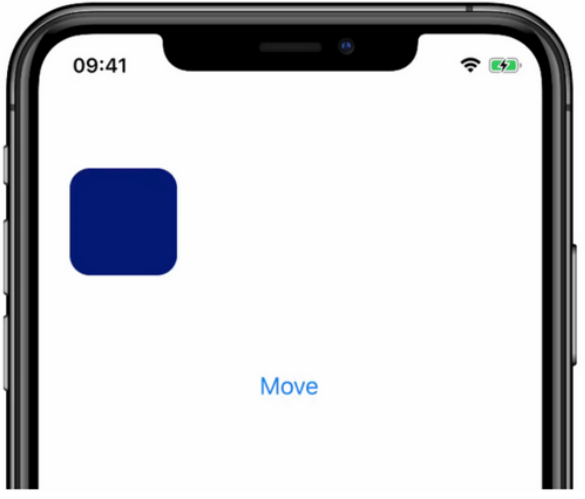
Shared Values

- Trong ví dụ trên, chúng ta cập nhật giá trị không đồng bộ từ thread React Native JS. Cập nhật có thể được thực hiện đồng bộ khi thực hiện chúng từ bên trong một worklet, như sau:

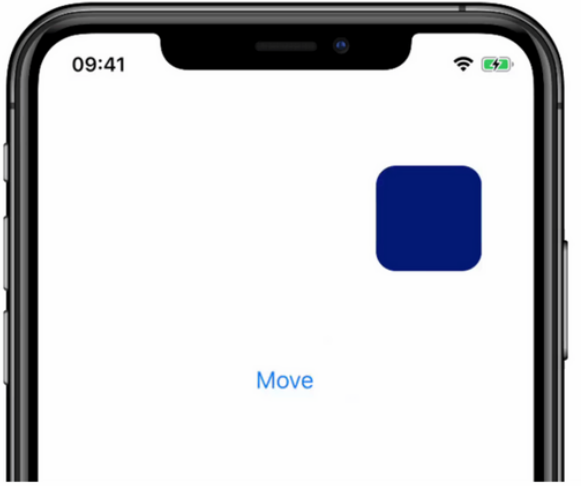
```
const scrollOffset = useSharedValue(0);

const scrollHandler = useAnimatedScrollHandler({
  onScroll: event => {
    scrollOffset.value = event.contentOffset.y;
  },
});

return (
  <Animated.ScrollView onScroll={scrollHandler}>
    <View />
  </Animated.ScrollView>
);
```

Shared Values

- Học lý thuyết chán rồi, bây giờ chúng ta sẽ đi vào một ví dụ. Chúng ta sẽ tạo một box và sẽ cho nó di chuyển random trong hàng, như hình bên dưới



Trước khi nhấn Move



Sau khi nhấn Move

Shared Values

- Đây là code mẫu của chúng ta:

```
const offset = useSharedValue(0);

const animatedStyles = useAnimatedStyle(() => {
  return {
    transform: [{translateX: offset.value}],
  };
});

return (
  <>
    <Animated.View style={[styles.box, animatedStyles]} />
    <Button
      onPress={() => (offset.value = Math.random() * 255)}
      title="Move"
    />
  </>
);
```

Shared Values

Giải thích

- useAnimatedStyle: là một hook được sử dụng để định nghĩa các kiểu ?animation các thành phần React Native. Hook này cho phép bạn tạo và điều khiển các hoạt hình dựa trên giá trị của các biến hoạt động trong thành phần. useAnimatedStyle thuộc tính hoạt hình như transform, opacity, width, height, và nhiều thuộc tính khác
- transform: định nghĩa sẽ tạo một animation di chuyển. translateX dùng để di chuyển theo phương ngang.

Shared Values

- Mọi thứ đến đây có vẻ ổn rồi phải không, tạo animation không có gì khó cả! Nhưng khoản, animation này chưa được mượt lắm. Để khắc phục điều này chúng ta sẽ sử dụng hook withTiming
- withTiming là một hàm trong thư viện Reanimated được sử dụng để tạo hiệu ứng chuyển động dựa trên thời gian. Hàm này cho phép bạn định nghĩa các giá trị tương ứng với một giá trị đầu và một giá trị đích, và quyết định cách giá trị thay đổi trong khoảng thời gian nhất định.
- Cú pháp sử dụng của withTiming trong Reanimated như sau:

```
import { withTiming } from 'react-native-reanimated';

const animatedValue = withTiming(toValue, config);
```

- Trong đó:
  - toValue: Giá trị cuối cùng mà thuộc tính muốn thay đổi đến.
  - config (tuỳ chọn): Đối tượng cấu hình để điều chỉnh hiệu ứng chuyển động. Các thuộc tính cấu hình bao gồm:
    - duration: Thời gian (tính bằng mili giây) để thực hiện chuyển động từ giá trị hiện tại đến giá trị đích. Mặc định là 300 (mili giây).
    - easing: Hàm để xác định cách giá trị thay đổi theo thời gian. Một số lựa chọn thông dụng là Easing.linear, Easing.ease, Easing.easeIn, Easing.easeOut, Easing.easeInOut và các hàm tùy chỉnh. Mặc định là Easing.linear.
    - delay: Thời gian chờ (tính bằng mili giây) trước khi bắt đầu chuyển động. Mặc định là 0.

Shared Values

- Thêm withSpring, sau đó khởi chạy lại ứng dụng bạn sẽ thấy animation mượt mà hơn nhiều rồi đấy.

```
return (
  <View>
    <Animated.View style={[styles.box, animatedStyles]} />
    <Button
      onPress={() => (offset.value = withSpring(Math.random() * 255))}
      title="Move"
    />
  </View>
);
```

Events

- Trong thế giới thực, không có gì thay đổi ngay lập tức - luôn có một cái gì đó giữa các quốc gia. Khi chúng ta chạm vào một cuốn sách, chúng ta không mong đợi nó mở ngay lập tức trên một trang nhất định. Để làm cho ứng dụng dành cho thiết bị di động cảm thấy tự nhiên hơn đối với người dùng, chúng tôi sử dụng hoạt ảnh để làm mượt mà các tương tác của người dùng với giao diện người dùng ứng dụng.
- Để cho thấy cách xử lý event được thực hiện trong Reanimated chúng tôi sẽ hướng dẫn bạn từng bước để đạt được kết quả sau:

- Để cho thấy cách xử lý event được thực hiện trong Reanimated chúng tôi sẽ hướng dẫn bạn từng bước để đạt được kết quả sau:

Element hình tròn sẽ di chuyển theo vị trí con trỏ

Handling gesture events

- Để thao tác các sự kiện liên quan đến con trỏ, các bạn cần cài đặt package react-native-gesture-handler vào dự án của mình.
- Đối với hệ điều hành Android, hãy đảm bảo gói điểm vào ứng dụng của bạn bằng <GestureHandlerRootView> thành phần từ thư viện react-native-gesture-handler để nắm bắt các sự kiện đúng cách.

```
import { GestureHandlerRootView } from 'react-native-gesture-handler';

export default function App() {
  return (
    <GestureHandlerRootView style={{ flex: 1 }}>
      {/* content */}
    </GestureHandlerRootView>
  );
}
```

- Quay trở lại ví dụ tương tác, chúng ta bắt đầu bằng cách chỉ tập trung vào các sự kiện chạm.
- Chúng ta sẽ xây dựng animation giống với hình demo dưới đây

Trước khi chạm

Sau khi chạm

Handling gesture events

- Tạo nút hình tròn, được bọc bởi TapGestureHandler để bắt sự kiện khi có tác động lên phần tử này

```
const EventsExample = () => {
  const pressed = useSharedValue(false);
  return (
    <TapGestureHandler onGestureEvent={eventHandler}>
      <Animated.View style={[styles.ball]} />
    </TapGestureHandler>
  );
};
```

- Animated.View cũng giống như View nhưng để biểu diễn các animation lên phần tử của Reanimated thì bạn phải sử dụng Animated.View

Handling gesture events

- Chúng ta sử dụng useAnimatedGestureHandler để bắt được sự kiện với con trỏ

```
const eventHandler = useAnimatedGestureHandler({
  onStart: (event, ctx) => {
    pressed.value = true;
  },
  onEnd: (event, ctx) => {
    pressed.value = false;
  },
});
```

- onStart: khi bắt đầu nhấn vào
- onEnd: khi thả tha

Handling gesture events

- ?Chúng ta sẽ style cho cục tròn bằng useAnimatedStyle, màu background và độ scale của hình tròn sẽ dựa vào pressed.value.

```
const uas = useAnimatedStyle(() => {
  return {
    backgroundColor: pressed.value ? '#FEEF86' : '#001972',
    transform: [{ scale: pressed.value ? 1.2 : 1 }],
  };
});
```

- Bây giờ bạn đã có thể đổi màu hình tròn của chúng ta bằng cách nhấn vào nó. Bây giờ chúng ta sẽ bắt sự kiện kéo hình tròn và cho nó di chuyển theo hướng di chuyển của con trỏ.

Handling gesture events

- Đây là cách chúng ta có thể lưu vị trí bắt đầu trong callback onStart bằng sử dụng context;

```
onStart: (event, ctx) => {
  pressed.value = true;
  ctx.startX = x.value;
  ctx.startY = y.value;
},
```

Handling gesture events

- Sau đó, chúng ta có thể sử dụng nó trong onActive để tính toán vị trí hiện tại

```
onActive: (event, ctx) => {
  x.value = ctx.startX + event.translationX;
  y.value = ctx.startY + event.translationY;
},
```

- Bạn sẽ tạo thêm một style bằng useAnimatedStyle để gán vị trí cho hình tròn bằng giá trị của x, y.

Tổng kết

- Khi đọc tới đây cơ bản bạn đã tạo được các animation cho riêng bình. Reanimated còn rất nhiều thứ để bạn khám phá và sử dụng nó. Vì thời lượng của bài học, chúng ta không thể nào giới thiệu hết được. Nhưng không sao, bạn có thể tự tìm hiểu thêm bằng cách truy cập document của Reanimated https://docs.swmansion.com/react-native-reanimated/docs/

Tổng kết

- [ ] Hiểu và sử dụng hiệu quả thư viện Reanimated
- [ ] Tối ưu hóa hiệu suất cho animation
- [ ] Tương tác người dùng tốt hơn
- [ ] Xây dựng các hiệu ứng hoạt hình phức tạp
- [ ] Nắm vững kiến thức về JavaScript và React Native

Kết thúc