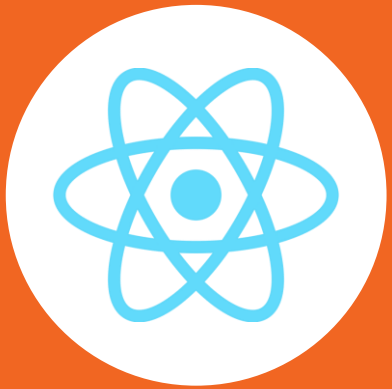
FPT Education

FPT POLYTECHNIC



LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

BÀI 1: GIỚI THIỆU VỀ CUSTOM COMPONENT TRONG REACT NATIVE

PHẦN 1: TẦM QUAN TRỌNG, VÀ CÁCH KHỞI TẠO CUSTOM COMPONENT

Mục tiêu

- Hiểu rõ tầm quan trọng và lợi ích khi tạo custom component
- Các bước để tạo một custom component
- Tạo <Header /> component
- Tạo <Wrapper /> component

React native custom component

React native là một framework JavaScript phổ biến để phát triển các ứng dụng di động native mạnh mẽ và dựa trên react, sử dụng các thành phần khai báo để tạo giao diện người dùng. Nó là một tập hợp các thư viện, cung cấp quyền truy cập vào các API native tương ứng. Như vậy, các ứng dụng được phát triển với react native sẽ có quyền truy cập vào các tính năng native như âm thanh, video, camera, vị trí, v.v.

React native custom component

React native sử dụng một tập hợp JavaScript và JSX (cú pháp giống XML) để tạo các thành phần cho giao diện người dùng như một hàm của trạng thái ứng dụng hiện tại. Trước đây, việc sử dụng các công nghệ khác như Cordova, Titanium, v.v., cung cấp nền tảng để xây dựng các ứng dụng di động bằng các công nghệ web như HTML, CSS và JavaScript.

Không giống như các ứng dụng được viết trên Cordova hoặc Titanium, tạo ra các phần tử giao diện người dùng bằng cách gói nội dung trong WebView, React Native sử dụng các thành phần JavaScript hiển thị dưới dạng các widget giao diện native. Điều này cung cấp trải nghiệm native thực sự, trong khi hầu hết công việc phát triển của bạn chỉ được thực hiện một lần cho nhiều nền tảng như iOS, Android và Windows.

Điều quan trọng cần lưu ý là ngay cả với Cordova, chúng ta có thể truy cập các API native, tuy nhiên cốt lõi vẫn sẽ là HTML và JavaScript được hiển thị trong chế độ xem web. Mặt khác, với React Native, các thành phần JavaScript được hiển thị dưới dạng các **widget native**, gọi API actual trên host platform bằng cách sử dụng một lớp trừu tượng được gọi là **Bridge**. Điều này cung cấp ứng dụng có hiệu năng cao.

Ở kiến trúc mới của React Native, **Bridge** không còn được sử dụng nữa. Thay vào đó, nó sử dụng kiến trúc của **Hermes.** Giúp giảm thời gian khởi động ứng dụng.

- Tại sao phải viết custom component cho ứng dụng của bạn!

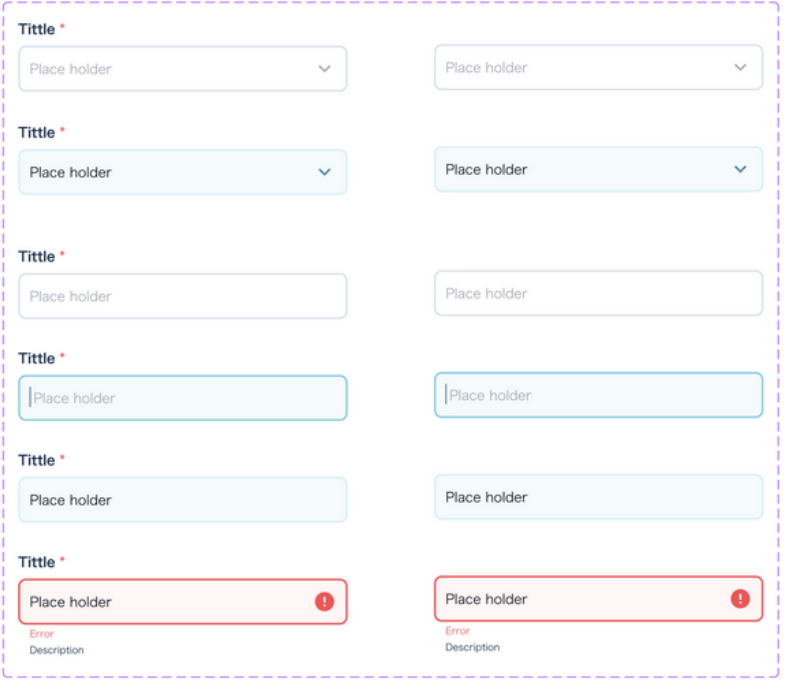
Trong quá trình phát triển thực tế ứng dụng React Native, sẽ có rất nhiều thành phần được lặp đi lặp lại nhiều lần. Việc phải viết lại các thành phần này rất tốn thời gian xây dựng và kể cả trong quá trình bảo trì, sửa lỗi cũng rất mất thời gian. Việc tạo một custom component cho một thành phần cụ thể nào đó, giúp bạn dễ dàng quản lý các thành phần của mình hơn.

Có thể nói viết custom component là một kỹ thuật không thể thiếu trong tất cả các ứng dụng được phát triển bằng React Native, tận dụng được sức mạnh của framework này mang lại

Custom component là gì?

Trong React Native, "**custom component**" là một thành phần được tạo ra bởi người dùng hoặc nhà phát triển để tái sử dụng trong ứng dụng của họ. Các thành phần này được xây dựng bằng cách kết hợp các thành phần có sẵn từ React Native để tạo ra một thành phần mới có thể thực hiện một số chức năng cụ thể hoặc có giao diện người dùng mà thành các core component không thể làm được.

- Đây làm một số input component thường gặp:



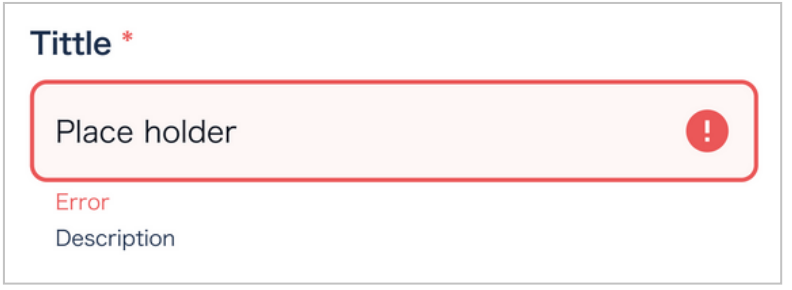
Ý tưởng

- Sau khi nhìn qua hình ảnh các input component bên trên bạn đã có ý tưởng gì để xây dựng custom component cho mình các input này chưa?
- Dưới đây là input cơ bản mà bạn có thể tạo từ thành phần <TextInput /> của React Native

- Nhưng để các bạn có thể thêm tiêu đề, mô tả dưới input hoặc thêm icon bên trong input thì các bạn cần viết một custom component cho mình.

Ý tưởng

- Để xây dựng custom component cho input sau các bạn cần tạo ra 2 custom component:



- WrapInput

  Component này gồm các thành phần bên ngoài input, dùng để bọc input của chúng ta. Gồm Tiêu đề, text hiển thị lỗi và mô tả của input.

Ý tưởng

TextInput

Component này không phải là **<TextInput />** của React Native, mà là tên một custom component mới chúng ta sẽ tạo ra có chứa **<TextInput />** của React Native.

Trong custom **<TextInput />** này các bạn thêm hàm kiểm tra giá trị để trả về màu background, màu border, ... Bạn thêm 2 thành phần icon ở đầu input và cuối input.

Ý tưởng

- Như vậy từ custom component WrapInput các bạn có thể tái sử dụng lại cho các input có cùng chức năng khác như select, dropdown, pickerdatetime, textarea...
- Thật tuyệt phải không nào, bây giờ chúng ta sẽ bắt tay vào xây dựng custom component đơn giản và vô cùng hữu ích, tên gọi là Wrapper, thành phần này dùng để bọc lại cả màn hình, chứa Statusbar, căn lề cho nội dung ứng dụng, tránh cho nội dung bị tràn.

Wrapper

- Wrapper sẽ được sử dụng để bọc ngoài cùng các thành phần của screen như sau:

```
<Wrapper backgroundColor="white" barStyle="dark-content">
  <View style={styles.container}>
    <Text style={styles.text}>Màn hình</Text>
  </View>
</Wrapper>
```

Wrapper

- Đầu tiên chúng ta bắt đầu đi vào xây dựng custom component Wrapper
  - Khai báo các props của Wrapper

```
interface WrapperProps {
  barStyle?: StatusBarStyle;
  children: ReactNode;
  disableAvoidKeyboard?: boolean;
  backgroundColor?: string;
  bottomSafeArea?: boolean;
  disableAvoidStatusBar?: boolean;
  style?: ViewStyle;
}
```

- Các bạn sẽ cần install thêm thư viện react-native-safe-area-context để sử dụng hook useSafeAreaInsets, nó có thể lấy được các khoảng cách lề mà nội dung ứng dụng có thể bị che.

- paddingBottom là khoảng cách an toàn với lề dưới để nội dung không bị che đi bởi thiết bị

```
const {top, bottom} = useSafeAreaInsets();
const paddingBottom = bottomSafeArea ? bottom : 0;
```

Wrapper

- Style cho View component, sau đó truyền thêm các giá trị từ prop vào styles của Wrapper

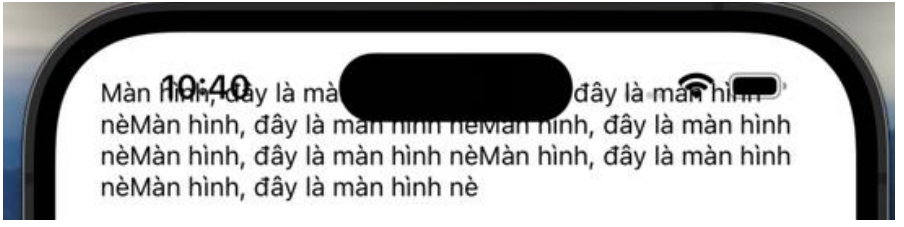
```
<View
  style={[
    styles.container,
    {
      backgroundColor,
      paddingBottom,
    },
    style,
  ]}>
  <KeyboardAvoidingView…
  </KeyboardAvoidingView>
</View>
```

Wrapper

- Bọc KeyboardAvoidingView để đẩy, tránh keyboard che input

```
<KeyboardAvoidingView
  style={styles.flexFill}
  behavior={'padding'}
  enabled={!disableAvoidKeyboard}>
  <StatusBar
    barStyle={barStyle}
    translucent
    backgroundColor={'transparent'}
  />
  {!disableAvoidStatusBar && (
    <View style={[{height: top}, styles.statusBar]} />
  )}
  {children}
</KeyboardAvoidingView>
```

- Để các thuộc tính **translucent** và background **transparent** để màu nền **Statusbar** trong suốt và ẩn **StausBar** đi
- **disableAvoidStatusBar** là thành phần che phủ phần StatusBar, bạn có thể ẩn hoặc hiện thành phần này đi. Nếu **disableAvoidStatusBar** = true thì nội dung bên trong Wrapper sẽ như sau:



Wrapper

- **children** chính là tất cả các thành phần con nằm bên trong **Wrapper**

```
<Wrapper backgroundColor="white" barStyle="dark-content">
  <View style={styles.container}>
    <Text style={styles.text}>Màn hình</Text>
  </View>
</Wrapper>
```

- Chạy ứng dụng và chúng ta có kết quả như sau:

Header

- Header là một thành phần không thể thiếu của các ứng dụng hiện tại, việc tái sử dụng chúng là thường xuyên. Thế nên chúng ta sẽ bắt tay vào xây dựng custom Header component.

Header

- Khai báo các props của Header cần có, các props này sẽ tuỳ thuộc vào từng yêu cầu ứng dụng của các bạn, sẽ cần thêm hoặc bớt.

```
interface HeaderProps {
  title?: string;
  iconLeft?: ImageSourcePropType;
  iconRight?: ImageSourcePropType;
  onPressRight?: () => void;
  onPressLeft?: () => void;
```

- title: tiêu đề của Header component
- iconLeft: là đường dẫn đến icon phía bên trái
- iconRight: là đường dẫn đến icon phía bên phải

- onPressRight: là một prop function, function sẽ được kích hoạt khi người dùng nhấn vào vùng icon bên phải
- onPressLeft: là một prop function, function sẽ được kích hoạt khi người dùng nhấn vào vùng icon bên trái

```
leftComponent?: ReactNode;
centerComponent?: ReactNode;
rightComponent?: ReactNode;
bgColor?: string;
iconLeftColor?: string;
iconRightColor?: string;
leftIconSize?: number;
rightIconSize?: number;
numberOfLines?: number;
```

- leftComponent: là một component tuỳ chỉnh thay thế cho icon mặc định bên trái của Header, prop này giúp component của chúng ta mang tính tuỳ chỉnh cho nhiều trường hợp hơn.

- centerComponent: là một component tuỳ chỉnh thay thế tiêu đề mặc định ở giữa của Header, prop này giúp component của chúng ta mang tính tuỳ chỉnh cho nhiều trường hợp hơn.

- rightComponent: là một component tuỳ chỉnh thay thế cho icon mặc định bên phải của Header, prop này giúp component của chúng ta mang tính tuỳ chỉnh cho nhiều trường hợp hơn.
- Các prop còn lại là các giá trị tuỳ chỉnh style cho Header component của chúng ta.

Header

- Một Header component của chúng ta, sẽ được chia ra làm 3 phành phần nhỏ renderLeft, renderCenter, renderRight.

```
<View style={[styles.container]}>
  {renderLeft()}
  {renderCenter()}
  {renderRight()}
</View>
```

Header

- renderLeft function chịu trách nhiệm render thành phần bên trái của Header, nếu bạn có truyền thành phần leftComponent thì nó sẽ render nó, điều này giúp bạn tuỳ chỉnh, thêm bất kỳ thành phần nào mà bạn muốn.

```
const renderLeft = () => {
  return (
    leftComponent || (
      <View>
        {iconLeft ? (…
        ) : (
          <View style={{width: leftIconSize, height: leftIconSize}} />
        )}
      </View>
    )
  );
};
```

Header

- Hiển thị icon bên trái của Header, là thành phần được sử dụng nhiều, nên chúng ta viết sẵn một image component để hiển thị hình ảnh từ prop iconLeft được truyền vào.

```
{iconLeft ? (
  <Pressable hitSlop={15} onPress={onPressLeft || Navigator.goback}>
    <Image
      source={iconLeft}
      tintColor={iconLeftColor}
    />
  </Pressable>
) : (
  <View style={{width: leftIconSize, height: leftIconSize}} />
)}
```

Header

- renderCenter hàm render component ở giữa, nếu bạn muốn tuỳ chỉnh giao diện thì truyền prop centerComponent, không thì bạn có thể tuyền prop title để đặt tên cho Header.

```
const renderCenter = () => {
  return (
    centerComponent || (
      <View style={styles.containerCenter}>
        <Text style={styles.title} numberOfLines={numberOfLines}>
          {title}
        </Text>
      </View>
    )
  );
};
```

Header

- renderRight có các thành phần tương tự như renderLeft, chỉ thay bằng các prop khác như iconRight, rightIconSize,...

```
const renderRight = () => {
  return (
    rightComponent || (
      <View style={styles.containerRight}>
        {iconRight ? (…
        ) : (
          <View style={{width: rightIconSize, height: rightIconSize}} />
        )}
      </View>
    )
  );
};
```

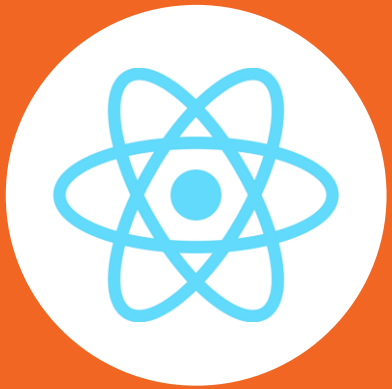
Header

- Bây giờ, bạn có thể sử dụng custom component Header đã được viết. Ngoài ra bạn có thể sử dụng nhiều hơn các prop có trong Header.

```
<Header
  title="Đây là tiêu đề"
  iconLeft={{
    uri: 'https://cdn-icons-png.flaticon.com/512/271/271220.png',
  }}
  rightComponent={customRightHeader()}
/>
```

FPT Education

FPT POLYTECHNIC



LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

BÀI 1: GIỚI THIỆU VỀ CUSTOM COMPONENT TRONG REACT NATIVE

PHẦN 2: CUSTOM COMPONENT VIEW VÀ SECTION VIEW

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

Mục tiêu

- Tuỳ chỉnh core component <View /> mang các thuộc tính style ra ngoài prop.
- Tạo <SectionView /> custom component, hiện thị dữ liệu dạng nhiều thành phần

View component

- Như những gì bạn đã được học ở môn học "Đa nền tảng 1", <View /> là một vùng chứa và là thành phần được sử dụng nhiều nhất trong quá trình phát triển ứng dụng di động.
- Để bạn có thể style cho <View /> component bạn phải style inline hoặc bạn truyền style qua StyleSheet:

```
<View style={styles.container} />
```

```
<View
   style={{
      backgroundColor: 'red',
       height: 200,
       width: 100,
       borderColor: 'black',
    }}
 />
```

- Việc style kiểu như này rất tốn thời gian, bạn phải suy nghĩ cho tên của style nếu như xài StyleSheet, nếu style inline thì sẽ khiến hiệu năng, bộ nhớ của ứng dụng không được tốt, điện thoại phải liên tục cấp bộ nhớ mới cho style qua các lần re-render. Ngoài ra style như thế cũng khiến code ứng dụng của bạn rất khó đọc, bảo trì và nâng cấp sau này.

View component

- Sau khi xây dựng xong <View /> custom component thì sẽ style component trông giống như thế này:

```
<View
  height={200}
  width={200}
  margin={10}
  bg="gray"
  radius={20}
  justifyCenter
  alignCenter>
  <Text>Sử dụng View custom component bọc Text</Text>
</View>
```

View component

- Điều đầu tiên trước khi bắt đầu xây dựng bất kỳ một custom component, bạn phải xác định được các chức năng, component này sẽ dùng cho các chức năng gì và dự đoán được khả năng này có thể được sử dụng cho các thành phần khác trong tương lai. Điều này giúp tăng khả năng tái sử dụng custom component.
- Đối với <View /> custom component này thì khá đơn giản, chúng ta chỉ cần khai báo các prop có sẵn trong prop style của View, các bạn có thể đặt tên cho các prop ngắn gọn hơn, hoặc viết một số thuộc tính style vào trong component <View /> và gọi sử dụng thông qua prop của component.

View component

- Chúng ta sẽ khai báo một số props cho <View /> custom component của chúng ta

```
export interface RViewProps extends ViewProps {
  bg?: string;
  row?: boolean;
  radius?: number;
  margin?: number;
  padding?: number;
  height?: DimensionValue;
  width?: DimensionValue;
```

View component

- Props:
  - bg: xác định màu nền cho thành thành.
  - row: hướng sắp của các thành phần bên trong (mặc định thành phần được sắp xếp theo dọc).
  - radius: độ cong của thành phần.
  - margin: căn lề ngoài thành phần.
  - padding: căn lề bên trong thành phần.
  - height: chiều cao của ứng dụng.
  - width: chiều rộng của ứng dụng.

View component

- Tạo một danh sách viewStyle định nghĩa các giá trị style từ prop nhận được

```
const viewStyle: {} = [
  abs && styles.absolute,
  row && styles.row,
  bg && {backgroundColor: bg},
  radius && {borderRadius: radius},
  height && {height},
  width && {width},
  margin && {margin},
  padding && {padding},
  {...StyleSheet.flatten(style)},
];
```

View component

- Style chi tiết cho từng thuộc props nhận được:
  - styles.absolute

```
absolute: {
    position: 'absolute',
}
```

  - styles.absolute

```
row: {
    flexDirection: 'row',
}
```

- StyleSheet.flatten()

```
static flatten(style: Object[]): Object;
```

StyleSheet.flatten có chức năng gộp một mảng các đối tượng style, thành một đối tượng style tổng hợp. Ngoài ra, phương pháp này có thể được sử dụng để tra cứu ID, được trả về bởi StyleSheet.register.

View component

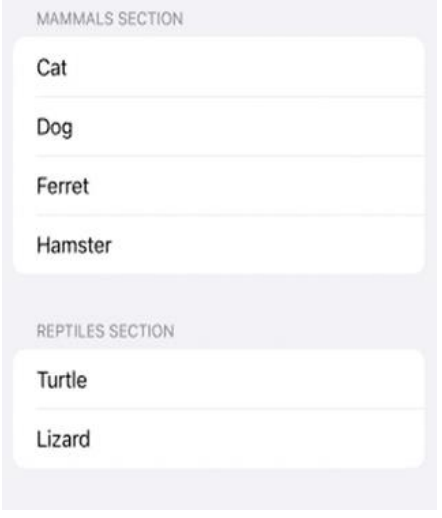
- Cuối cùng bạn return ra component của custom component. <View /> dưới đây là component của React Native.

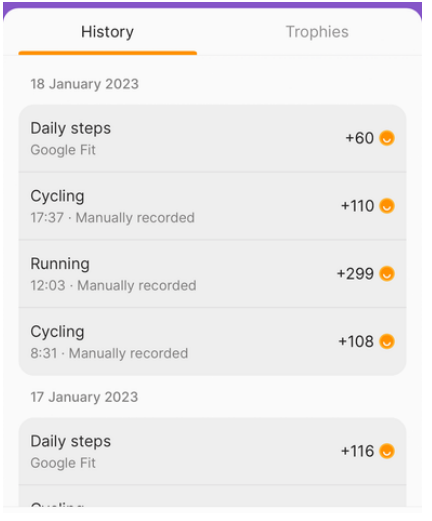
```
return <View {...rest} ref={ref} style={viewStyle}
```

- ...rest là các props của ViewProps mà chúng ta đã extends trong RViewProps ở phần trên.
  - Ngoài những props các bạn đã được hướng dẫn, các bạn có thể khai báo thêm những props khác để cho **<View />** component của các bạn dễ dàng sử dụng hơn nữa.

Section component

- Đối với những giao diện mang tính lặp đi lặp lại, trên những data gần như tương đồng nhau mà được gop thành từng mục, ví dụ như các mục cài đặt trong ứng dụng, danh sách danh bạ...



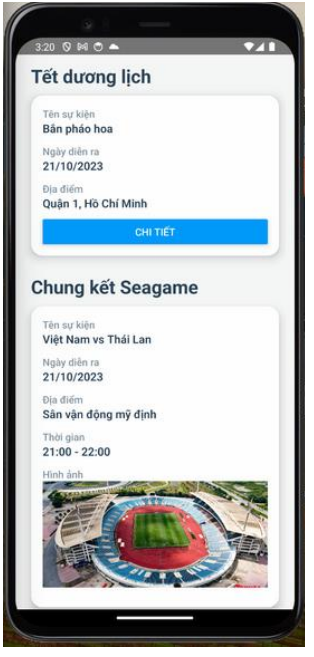


Section component

- Đối với yêu cầu xây dựng ứng dụng như vậy, chúng ta nên viết custom component cho chúng, để quá trình lập trình được nhanh hơn, giảm số lượng bug gặp phải và dễ dàng bảo trì code sau này.

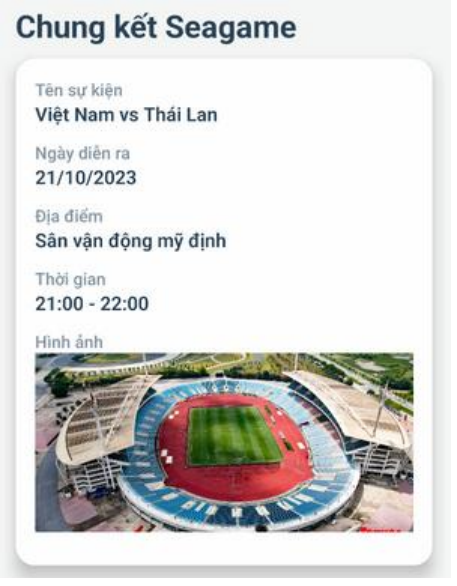
Section component

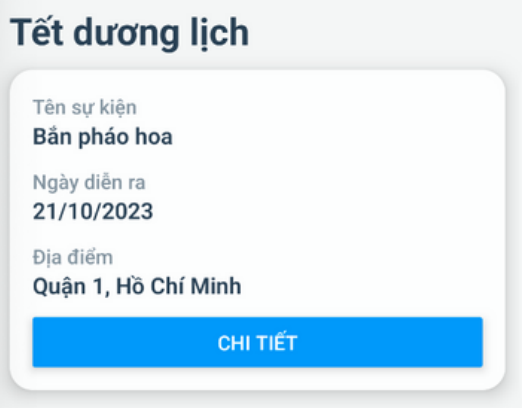
- Mục tiêu của bài học, các bạn sẽ xây dựng một component hiển thị như sau:



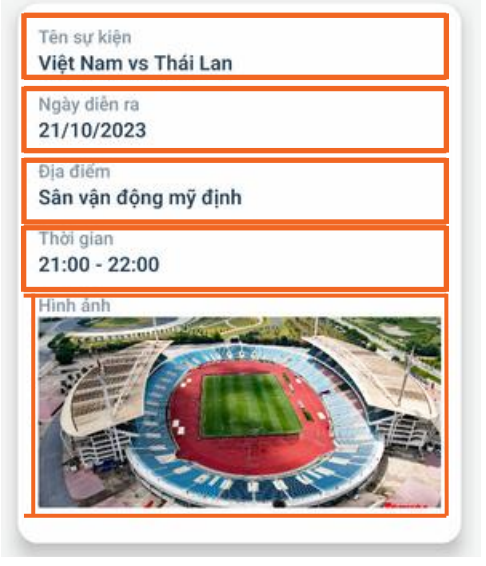
Section component

- Section component của chúng ta sẽ bao gồm 2 thành phần chính:
  - Phần 1: Các item render ra thành phần chính gồm **tiêu đề** và **card**.





- Phần 2: Các item render ra nội dung trong card



Section component

- Tương ứng với mỗi phần, chúng ta sẽ khai báo các prop data type cho Section component.
  - Phần 1:

```
interface SectionProps {
    title: string;
    titleStyle?: TextStyle;
    events?: EventProps[];
}
```

- Phần 2:

```
interface EventProps {
  title?: string;
  content?: string;
  titleStyle?: TextStyle;
  contentStyle?: TextStyle;
  contentComponent?: ReactNode;
  eventComponent?: ReactNode;
}
```

Section component

- Phần 1: Xây dựng function render Section item, bao gồm title và phần dưới render các item trong card

```
const renderSection = (data: SectionProps, index: number) => {
  const {title, events, titleStyle} = data;
  return (
    <View key={index.toString()} style={[styles.section]}>
      <Text style={[styles.titleSection, titleStyle]}>{title}</Text>
      <View style={[styles.sectionBody, styles.shadow]}>
        {events?.map(renderChild)}
      </View>
    </View>
  );
};
```

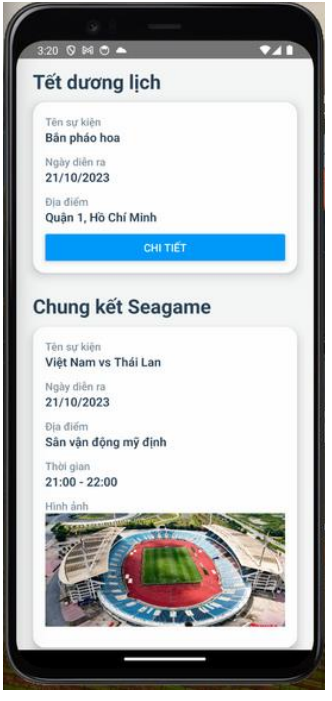
Section component

- Phần 2: function render ra các item bên trong card, bạn có thể tuỳ chỉnh cả nội dung item bằng **eventComponent** hoặc nếu muốn tuỳ chỉnh nội dung dưới title bạn có thể truyền vào **contentComponent**

```
const renderChild = (data: EventProps, index: number) => {
  const {…
  } = data;

  return (
    eventComponent || (
      <View key={index.toString()} style={styles.containerChild}>
        <Text style={[styles.titleChild, titleStyle]}>{title}</Text>
        {contentComponent || (
          <Text style={[styles.contentChild, contentStyle]}>{content}</
          Text>
        )}
      </View>
    )
  );
};
```

- Reload lại ứng dụng và bạn sẽ có giao diện như sau:



Tổng kết

- Hiểu rõ tầm quan trọng và lợi ích khi tạo custom component
- Các bước để tạo một custom component
- Tạo <Header /> component
- Tạo <Wrapper /> component
- Tuỳ chỉnh core component <View /> mang các thuộc tính style ra ngoài prop.
- Tạo <SectionView /> component, hiện thị dữ liệu dạng nhiều thành phần

Kết thúc