iOS Native UI Components

INFO

Native Module and Native Components are our stable technologies used by the legacy architecture. They will be deprecated in the future when the New Architecture will be stable. The New Architecture uses Turbo Native Module and Fabric Native Components to achieve similar results.

There are tons of native UI widgets out there ready to be used in the latest apps - some of them are part of the platform, others are available as third-party libraries, and still more might be in use in your very own portfolio. React Native has several of the most critical platform components already wrapped, like `ScrollView` and `TextInput`, but not all of them, and certainly not ones you might have written yourself for a previous app. Fortunately, we can wrap up these existing components for seamless integration with your React Native application.

Like the native module guide, this too is a more advanced guide that assumes you are somewhat familiar with iOS programming. This guide will show you how to build a native UI component, walking you through the implementation of a subset of the existing `MapView` component available in the core React Native library.

iOS MapView example

Let's say we want to add an interactive Map to our app - might as well use `MKMapView`, we only need to make it usable from JavaScript.

Native views are created and manipulated by subclasses of `RCTViewManager`. These subclasses are similar in function to view controllers, but are essentially singletons - only one instance of each is created by the bridge. They expose native views to the `RCTUIManager`, which delegates back to them to set and update the properties of the views as necessary. The `RCTViewManager`s are also typically the delegates for the views, sending events back to JavaScript via the bridge.

To expose a view you can:

- Subclass `RCTViewManager` to create a manager for your component.
- Add the `RCT_EXPORT_MODULE()` marker macro.
- Implement the `-(UIView *)view` method.

RNTMapManager.m

```
#import <MapKit/MapKit.h>

#import <React/RCTViewManager.h>

@interface RNTMapManager : RCTViewManager
@end

@implementation RNTMapManager

RCT_EXPORT_MODULE(RNTMap)

- (UIView *)view
{
  return [[MKMapView alloc] init];
}

@end
```

NOTE

Do not attempt to set the `frame` or `backgroundColor` properties on the `UIView` instance that you expose through the `-view` method. React Native will overwrite the values set by your custom class in order to match your JavaScript component's layout props. If you need this granularity of control it might be better to wrap the `UIView` instance you want to style in another `UIView` and return the wrapper `UIView` instead. See Issue 2948 for more context.

INFO

In the example above, we prefixed our class name with `RNT`. Prefixes are used to avoid name collisions with other frameworks. Apple frameworks use two-letter prefixes, and

React Native uses `RCT` as a prefix. In order to avoid name collisions, we recommend using a three-letter prefix other than `RCT` in your own classes.

Then you need a little bit of JavaScript to make this a usable React component:

MapView.tsx

```
import {requireNativeComponent} from 'react-native';

// requireNativeComponent automatically resolves 'RNTMap' to 'RNTMapManager'
module.exports = requireNativeComponent('RNTMap');
```

MyApp.tsx

```
import MapView from './MapView.js';

...

render() {
  return <MapView style={{flex: 1}} />;
}
```

Make sure to use `RNTMap` here. We want to require the manager here, which will expose the view of our manager for use in JavaScript.

NOTE

When rendering, don't forget to stretch the view, otherwise you'll be staring at a blank screen.

```
  render() {
    return <MapView style={{flex: 1}} />;
  }
```

This is now a fully-functioning native map view component in JavaScript, complete with pinch-zoom and other native gesture support. We can't really control it from JavaScript yet, though :(

Properties

The first thing we can do to make this component more usable is to bridge over some native properties. Let's say we want to be able to disable zooming and specify the visible region. Disabling zoom is a boolean, so we add this one line:

RNTMapManager.m

```
RCT_EXPORT_VIEW_PROPERTY(zoomEnabled, BOOL)
```

Note that we explicitly specify the type as `BOOL` - React Native uses `RCTConvert` under the hood to convert all sorts of different data types when talking over the bridge, and bad values will show convenient "RedBox" errors to let you know there is an issue ASAP. When things are straightforward like this, the whole implementation is taken care of for you by this macro.

Now to actually disable zooming, we set the property in JS:

MyApp.tsx

```
<MapView zoomEnabled={false} style={{flex: 1}} />
```

To document the properties (and which values they accept) of our MapView component we'll add a wrapper component and document the interface with React `PropTypes`:

MapView.tsx

```
import PropTypes from 'prop-types';
import React from 'react';
import {requireNativeComponent} from 'react-native';
```

```
class MapView extends React.Component {
  render() {
    return <RNTMap {...this.props} />;
  }
}

MapView.propTypes = {
  /**
   * A Boolean value that determines whether the user may use pinch
   * gestures to zoom in and out of the map.
   */
  zoomEnabled: PropTypes.bool,
};

const RNTMap = requireNativeComponent('RNTMap');

module.exports = MapView;
```

Now we have a nicely documented wrapper component to work with.

Next, let's add the more complex `region` prop. We start by adding the native code:

RNTMapManager.m

```
RCT_CUSTOM_VIEW_PROPERTY(region, MKCoordinateRegion, MKMapView)
{
  [view setRegion:json ? [RCTConvert MKCoordinateRegion:json] : defaultView.region
animated:YES];
}
```

Ok, this is more complicated than the `BOOL` case we had before. Now we have a `MKCoordinateRegion` type that needs a conversion function, and we have custom code so that the view will animate when we set the region from JS. Within the function body that we provide, `json` refers to the raw value that has been passed from JS. There is also a `view` variable which gives us access to the manager's view instance, and a `defaultView` that we use to reset the property back to the default value if JS sends us a null sentinel.

You could write any conversion function you want for your view - here is the implementation for `MKCoordinateRegion` via a category on `RCTConvert`. It uses an already existing category of ReactNative `RCTConvert+CoreLocation`:

RNTMapManager.m

```
#import "RCTConvert+Mapkit.h"
```

RCTConvert+Mapkit.h

```
#import <MapKit/MapKit.h>
#import <React/RCTConvert.h>
#import <CoreLocation/CoreLocation.h>
#import <React/RCTConvert+CoreLocation.h>

@interface RCTConvert (Mapkit)

+ (MKCoordinateSpan)MKCoordinateSpan:(id)json;
+ (MKCoordinateRegion)MKCoordinateRegion:(id)json;

@end

@implementation RCTConvert(MapKit)

+ (MKCoordinateSpan)MKCoordinateSpan:(id)json
{
  json = [self NSDictionary:json];
  return (MKCoordinateSpan){
    [self CLLocationDegrees:json[@"latitudeDelta"]],
    [self CLLocationDegrees:json[@"longitudeDelta"]]
  };
}

+ (MKCoordinateRegion)MKCoordinateRegion:(id)json
{
  return (MKCoordinateRegion){
    [self CLLocationCoordinate2D:json],
    [self MKCoordinateSpan:json]
  };
}

@end
```

These conversion functions are designed to safely process any JSON that the JS might throw at them by displaying "RedBox" errors and returning standard initialization values when missing keys or other developer errors are encountered.

To finish up support for the `region` prop, we need to document it in `propTypes`:

MapView.tsx

```
MapView.propTypes = {
  /**
   * A Boolean value that determines whether the user may use pinch
   * gestures to zoom in and out of the map.
   */
  zoomEnabled: PropTypes.bool,

  /**
   * The region to be displayed by the map.
   *
   * The region is defined by the center coordinates and the span of
   * coordinates to display.
   */
  region: PropTypes.shape({
    /**
     * Coordinates for the center of the map.
     */
    latitude: PropTypes.number.isRequired,
    longitude: PropTypes.number.isRequired,

    /**
     * Distance between the minimum and the maximum latitude/longitude
     * to be displayed.
     */
    latitudeDelta: PropTypes.number.isRequired,
    longitudeDelta: PropTypes.number.isRequired,
  }),
};
```

MyApp.tsx

```
render() {
  const region = {
    latitude: 37.48,
    longitude: -122.16,
    latitudeDelta: 0.1,
    longitudeDelta: 0.1,
  };
  return (
    <MapView
      region={region}
      zoomEnabled={false}
      style={{flex: 1}}
    />
  );
}
```

Here you can see that the shape of the region is explicit in the JS documentation.

Events

So now we have a native map component that we can control freely from JS, but how do we deal with events from the user, like pinch-zooms or panning to change the visible region?

Until now we've only returned a `MKMapView` instance from our manager's `-(UIView *)view` method. We can't add new properties to `MKMapView` so we have to create a new subclass from `MKMapView` which we use for our View. We can then add a `onRegionChange` callback on this subclass:

RNTMapView.h

```
#import <MapKit/MapKit.h>

#import <React/RCTComponent.h>

@interface RNTMapView: MKMapView

@property (nonatomic, copy) RCTBubblingEventBlock onRegionChange;

@end
```

RNTMapView.m

```
#import "RNTMapView.h"

@implementation RNTMapView

@end
```

Note that all `RCTBubblingEventBlock` must be prefixed with `on`. Next, declare an event handler property on `RNTMapManager`, make it a delegate for all the views it exposes, and forward events to JS by calling the event handler block from the native view.

RNTMapManager.m

```
#import <MapKit/MapKit.h>
#import <React/RCTViewManager.h>

#import "RNTMapView.h"
#import "RCTConvert+Mapkit.h"

@interface RNTMapManager : RCTViewManager <MKMapViewDelegate>
@end

@implementation RNTMapManager

RCT_EXPORT_MODULE()

RCT_EXPORT_VIEW_PROPERTY(zoomEnabled, BOOL)
RCT_EXPORT_VIEW_PROPERTY(onRegionChange, RCTBubblingEventBlock)

RCT_CUSTOM_VIEW_PROPERTY(region, MKCoordinateRegion, MKMapView)
{
  [view setRegion:json ? [RCTConvert MKCoordinateRegion:json] : defaultView.region
animated:YES];
}

- (UIView *)view
{
  RNTMapView *map = [RNTMapView new];
  map.delegate = self;
  return map;
}
```

```
#pragma mark MKMapViewDelegate

- (void)mapView:(RNTMapView *)mapView regionDidChangeAnimated:(BOOL)animated
{
  if (!mapView.onRegionChange) {
    return;
  }

  MKCoordinateRegion region = mapView.region;
  mapView.onRegionChange(@{
    @"region": @{
      @"latitude": @(region.center.latitude),
      @"longitude": @(region.center.longitude),
      @"latitudeDelta": @(region.span.latitudeDelta),
      @"longitudeDelta": @(region.span.longitudeDelta),
    }
  });
}
@end
```

In the delegate method `-mapView:regionDidChangeAnimated:` the event handler block is called on the corresponding view with the region data. Calling the `onRegionChange` event handler block results in calling the same callback prop in JavaScript. This callback is invoked with the raw event, which we typically process in the wrapper component to simplify the API:

MapView.tsx

```
class MapView extends React.Component {
  _onRegionChange = event => {
    if (!this.props.onRegionChange) {
      return;
    }

    // process raw event...
    this.props.onRegionChange(event.nativeEvent);
  };
  render() {
    return (
      <RNTMap
        {...this.props}
        onRegionChange={this._onRegionChange}
      />
    );
  }
}
MapView.propTypes = {
  /**
   * Callback that is called continuously when the user is dragging the map.
   */
  onRegionChange: PropTypes.func,
  ...
};
```

MyApp.tsx

```
class MyApp extends React.Component {
  onRegionChange(event) {
    // Do stuff with event.region.latitude, etc.
  }

  render() {
    const region = {
      latitude: 37.48,
      longitude: -122.16,
      latitudeDelta: 0.1,
      longitudeDelta: 0.1,
    };
    return (
      <MapView
        region={region}
        zoomEnabled={false}
        onRegionChange={this.onRegionChange}
      />
    );
  }
}
```

Handling multiple native views

A React Native view can have more than one child view in the view tree eg.

```
<View>
  <MyNativeView />
  <MyNativeView />
  <Button />
</View>
```

In this example, the class `MyNativeView` is a wrapper for a `NativeComponent` and exposes methods, which will be called on the iOS platform. `MyNativeView` is defined in `MyNativeView.ios.js` and contains proxy methods of `NativeComponent`.

When the user interacts with the component, like clicking the button, the `backgroundColor` of `MyNativeView` changes. In this case `UIManager` would not know which `MyNativeView` should be handled and which one should change `backgroundColor`. Below you will find a solution to this problem:

```
<View>
  <MyNativeView ref={this.myNativeReference} />
  <MyNativeView ref={this.myNativeReference2} />
  <Button
    onPress={() => {
      this.myNativeReference.callNativeMethod();
    }}
  />
</View>
```

Now the above component has a reference to a particular `MyNativeView` which allows us to use a specific instance of `MyNativeView`. Now the button can control which `MyNativeView` should change its `backgroundColor`. In this example let's assume that `callNativeMethod` changes `backgroundColor`.

MyNativeView.ios.tsx

```
class MyNativeView extends React.Component {
  callNativeMethod = () => {
    UIManager.dispatchViewManagerCommand(
      ReactNative.findNodeHandle(this),
      UIManager.getViewManagerConfig('RNCMyNativeView').Commands
        .callNativeMethod,
      [],
    );
  };

  render() {
    return <NativeComponent ref={NATIVE_COMPONENT_REF} />;
  }
}
```

`callNativeMethod` is our custom iOS method which for example changes the `backgroundColor` which is exposed through `MyNativeView`. This method uses `UIManager.dispatchViewManagerCommand` which needs 3 parameters:

- `(nonnull NSNumber \*)reactTag` - id of react view.
- `commandID:(NSInteger)commandID` - Id of the native method that should be called
- `commandArgs:(NSArray<id> \*)commandArgs` - Args of the native method that we can pass from JS to native.

RNCMyNativeViewManager.m

```
#import <React/RCTViewManager.h>
#import <React/RCTUIManager.h>
#import <React/RCTLog.h>

RCT_EXPORT_METHOD(callNativeMethod:(nonnull NSNumber*) reactTag) {
    [self.bridge.uiManager addUIBlock:^(RCTUIManager *uiManager, NSDictionary<NSNumber
*,UIView *> *viewRegistry) {
        NativeView *view = viewRegistry[reactTag];
        if (!view || ![view isKindOfClass:[NativeView class]]) {
            RCTLogError(@"Cannot find NativeView with tag #%@", reactTag);
            return;
        }
        [view callNativeMethod];
    }];

}
```

Here the `callNativeMethod` is defined in the `RNCMyNativeViewManager.m` file and contains only one parameter which is `(nonnull NSNumber*) reactTag`. This exported function will find a particular view using `addUIBlock` which contains the `viewRegistry` parameter and returns the component based on `reactTag` allowing it to call the method on the correct component.

Styles

Since all our native react views are subclasses of `UIView`, most style attributes will work like you would expect out of the box. Some components will want a default style, however, for example `UIDatePicker` which is a fixed size. This default style is important for the layout algorithm to work as expected, but we also want to be able to override the default style when using the component. `DatePickerIOS` does this by wrapping the native component in an extra view, which has flexible styling, and using a fixed style (which is generated with constants passed in from native) on the inner native component:

DatePickerIOS.ios.tsx

```
import {UIManager} from 'react-native';
const RCTDatePickerIOSConsts = UIManager.RCTDatePicker.Constants;
...
  render: function() {
    return (
      <View style={this.props.style}>
        <RCTDatePickerIOS
          ref={DATEPICKER}
          style={styles.rkDatePickerIOS}
          ...
        />
      </View>
    );
  }
});

const styles = StyleSheet.create({
  rkDatePickerIOS: {
    height: RCTDatePickerIOSConsts.ComponentHeight,
    width: RCTDatePickerIOSConsts.ComponentWidth,
  },
});
```

The `RCTDatePickerIOSConsts` constants are exported from native by grabbing the actual frame of the native component like so:

RCTDatePickerManager.m

```
- (NSDictionary *)constantsToExport
{
  UIDatePicker *dp = [[UIDatePicker alloc] init];
  [dp layoutIfNeeded];

  return @{
    @"ComponentHeight": @(CGRectGetHeight(dp.frame)),
    @"ComponentWidth": @(CGRectGetWidth(dp.frame)),
    @"DatePickerModes": @{
      @"time": @(UIDatePickerModeTime),
      @"date": @(UIDatePickerModeDate),
      @"datetime": @(UIDatePickerModeDateAndTime),
    }
  };
}
```

This guide covered many of the aspects of bridging over custom native components, but there is even more you might need to consider, such as custom hooks for inserting and laying out subviews. If you want to go even deeper, check out the source code of some of the implemented components.

Is this page useful?

Edit this page

*Last updated on **Sep 1, 2023***