

Android 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 Android SDK programming. This guide will show you how to build a native UI component, walking you through the implementation of a subset of the existing `ImageView` component available in the core React Native library.

ImageView example

For this example we are going to walk through the implementation requirements to allow the use of `ImageViews` in JavaScript.

Native views are created and manipulated by extending `ViewManager` or more commonly `SimpleViewManager`. A `SimpleViewManager` is convenient in this case because it applies common properties such as background color, opacity, and Flexbox layout.

These subclasses are essentially singletons - only one instance of each is created by the bridge. They send native views to the `NativeViewHierarchyManager`, which delegates back

to them to set and update the properties of the views as necessary. The `ViewManagers` are also typically the delegates for the views, sending events back to JavaScript via the bridge.

To send a view:

1. Create the `ViewManager` subclass.
2. Implement the `createViewInstance` method
3. Expose view property setters using `@ReactProp` (or `@ReactPropGroup`) annotation
4. Register the manager in `createViewManagers` of the applications package.
5. Implement the JavaScript module

1. Create the `ViewManager` subclass

In this example we create view manager class `ReactImageManager` that extends `SimpleViewManager` of type `ReactImageView`. `ReactImageView` is the type of object managed by the manager, this will be the custom native view. Name returned by `getName` is used to reference the native view type from JavaScript.

Java Kotlin

```
public class ReactImageManager extends SimpleViewManager<ReactImageView> {

    public static final String REACT_CLASS = "RCTImageView";
    ReactApplicationContext mCallerContext;

    public ReactImageManager(ReactApplicationContext reactContext) {
        mCallerContext = reactContext;
    }

    @Override
    public String getName() {
        return REACT_CLASS;
    }
}
```

2. Implement method `createViewInstance`

Views are created in the `createViewInstance` method, the view should initialize itself in its default state, any properties will be set via a follow up call to `updateView`.

Java Kotlin

```
@Override
public ReactImageView createViewInstance(ThemedReactContext context) {
    return new ReactImageView(context, Fresco.newDraweeControllerBuilder(), null,
mCallerContext);
}
```

3. Expose view property setters using `@ReactProp` (or `@ReactPropGroup`) annotation

Properties that are to be reflected in JavaScript needs to be exposed as setter method annotated with `@ReactProp` (Or `@ReactPropGroup`). Setter method should take view to be updated (of the current view type) as a first argument and property value as a second argument. Setter should be public and not return a value (i.e. return type should be `void` in Java or `Unit` in Kotlin). Property type sent to JS is determined automatically based on the type of value argument of the setter. The following type of values are currently supported (in Java): `boolean`, `int`, `float`, `double`, `String`, `Boolean`, `Integer`, `ReadableArray`, `ReadableMap`. The corresponding types in Kotlin are `Boolean`, `Int`, `Float`, `Double`, `String`, `ReadableArray`, `ReadableMap`.

Annotation `@ReactProp` has one obligatory argument `name` of type `String`. Name assigned to the `@ReactProp` annotation linked to the setter method is used to reference the property on JS side.

Except from `name`, `@ReactProp` annotation may take following optional arguments: `defaultBoolean`, `defaultInt`, `defaultFloat`. Those arguments should be of the corresponding type (accordingly `boolean`, `int`, `float` in Java and `Boolean`, `Int`, `Float` in Kotlin) and the value provided will be passed to the setter method in case when the property that the setter is referencing has been removed from the component. Note that "default" values are only provided for primitive types, in case when setter is of some

complex type, `null` will be provided as a default value in case when corresponding property gets removed.

Setter declaration requirements for methods annotated with `@ReactPropGroup` are different than for `@ReactProp`, please refer to the `@ReactPropGroup` annotation class docs for more information about it. **IMPORTANT!** in ReactJS updating the property value will result in setter method call. Note that one of the ways we can update component is by removing properties that have been set before. In that case setter method will be called as well to notify view manager that property has changed. In that case "default" value will be provided (for primitive types "default" value can be specified using `defaultBoolean`, `defaultFloat`, etc. arguments of `@ReactProp` annotation, for complex types setter will be called with value set to `null`).

Java Kotlin

```
@ReactProp(name = "src")
public void setSrc(ReactImageView view, @Nullable ReadableArray sources) {
    view.setSource(sources);
}

@ReactProp(name = "borderRadius", defaultFloat = 0f)
public void setBorderRadius(ReactImageView view, float borderRadius) {
    view.setBorderRadius(borderRadius);
}

@ReactProp(name = ViewProps.RESIZE_MODE)
public void setResizeMode(ReactImageView view, @Nullable String resizeMode) {
    view.setScaleType(ImageResizeMode.toScaleType(resizeMode));
}
```

4. Register the ViewManager

The final step is to register the ViewManager to the application, this happens in a similar way to Native Modules, via the applications package member function `createViewManagers`.

Java Kotlin

```

@Override
public List<ViewManager> createViewManagers(
    ReactApplicationContext reactContext) {
    return Arrays.<ViewManager>asList(
        new ReactImageManager(reactContext)
    );
}

```

5. Implement the JavaScript module

The very final step is to create the JavaScript module that defines the interface layer between Java/Kotlin and JavaScript for the users of your new view. It is recommended for you to document the component interface in this module (e.g. using TypeScript, Flow, or plain old comments).

ImageView.tsx

```

import {requireNativeComponent} from 'react-native';

/**
 * Composes `View`.
 *
 * - src: string
 * - borderRadius: number
 * - resizeMode: 'cover' | 'contain' | 'stretch'
 */
module.exports = requireNativeComponent('RCTImageView');

```

The `requireNativeComponent` function takes the name of the native view. Note that if your component needs to do anything more sophisticated (e.g. custom event handling), you should wrap the native component in another React component. This is illustrated in the `MyCustomView` example below.

Events

So now we know how to expose native view components that we can control freely from JS, but how do we deal with events from the user, like pinch-zooms or panning? When a

native event occurs the native code should issue an event to the JavaScript representation of the View, and the two views are linked with the value returned from the `getId()` method.

Java **Kotlin**

```
class MyCustomView extends View {
    ...
    public void onReceiveNativeEvent() {
        WritableMap event = Arguments.createMap();
        event.putString("message", "MyMessage");
        ReactContext reactContext = (ReactContext)getContext();
        reactContext
            .getJSModule(RCTEventEmitter.class)
            .receiveEvent(getId(), "topChange", event);
    }
}
```

To map the `topChange` event name to the `onChange` callback prop in JavaScript, register it by overriding the `getExportedCustomBubblingEventTypeConstants` method in your `ViewManager`:

Java **Kotlin**

```
public class ReactImageManager extends SimpleViewManager<MyCustomView> {
    ...
    public Map getExportedCustomBubblingEventTypeConstants() {
        return MapBuilder.builder().put(
            "topChange",
            MapBuilder.of(
                "phasedRegistrationNames",
                MapBuilder.of("bubbled", "onChange")
            )
        ).build();
    }
}
```

This callback is invoked with the raw event, which we typically process in the wrapper component to make a simpler API:

MyCustomView.tsx

```
class MyCustomView extends React.Component {
  constructor(props) {
    super(props);
    this._onChange = this._onChange.bind(this);
  }
  _onChange(event) {
    if (!this.props.onChangeMessage) {
      return;
    }
    this.props.onChangeMessage(event.nativeEvent.message);
  }
  render() {
    return <RCTMyCustomView {...this.props} onChange={this._onChange} />;
  }
}
MyCustomView.propTypes = {
  /**
   * Callback that is called continuously when the user is dragging the map.
   */
  onChangeMessage: PropTypes.func,
  ...
};

const RCTMyCustomView = requireNativeComponent(`RCTMyCustomView`);
```

Integration with an Android Fragment example

In order to integrate existing Native UI elements to your React Native app, you might need to use Android Fragments to give you a more granular control over your native component than returning a `View` from your `ViewManager`. You will need this if you want to add custom logic that is tied to your view with the help of lifecycle methods, such as `onViewCreated`, `onPause`, `onResume`. The following steps will show you how to do it:

1. Create an example custom view

First, let's create a `CustomView` class which extends `FrameLayout` (the content of this view can be any view that you'd like to render)

Java Kotlin

CustomView.java

```
// replace with your package
package com.mypackage;

import android.content.Context;
import android.graphics.Color;
import android.widget.FrameLayout;
import android.widget.ImageView;
import android.widget.TextView;

import androidx.annotation.NonNull;

public class CustomView extends FrameLayout {
    public CustomView(@NonNull Context context) {
        super(context);
        // set padding and background color
        this.setPadding(16,16,16,16);
        this.setBackgroundColor(Color.parseColor("#5FD3F3"));

        // add default text view
        TextView text = new TextView(context);
        text.setText("Welcome to Android Fragments with React Native.");
        this.addView(text);
    }
}
```

2. Create a Fragment

Java Kotlin

MyFragment.java

```
// replace with your package
package com.mypackage;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
```



```
import android.view.ViewGroup;
import androidx.fragment.app.Fragment;

// replace with your view's import
import com.mypackage.CustomView;

public class MyFragment extends Fragment {
    CustomView customView;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup parent, Bundle
savedInstanceState) {
        super.onCreateView(inflater, parent, savedInstanceState);
        customView = new CustomView(this.getContext());
        return customView; // this CustomView could be any view that you want to render
    }

    @Override
    public void onViewCreated(View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        // do any logic that should happen in an `onCreate` method, e.g:
        // customView.onCreate(savedInstanceState);
    }

    @Override
    public void onPause() {
        super.onPause();
        // do any logic that should happen in an `onPause` method
        // e.g.: customView.onPause();
    }

    @Override
    public void onResume() {
        super.onResume();
        // do any logic that should happen in an `onResume` method
        // e.g.: customView.onResume();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        // do any logic that should happen in an `onDestroy` method
        // e.g.: customView.onDestroy();
    }
}
```

3. Create the ViewManager subclass

Java

Kotlin

MyViewManager.java

```
// replace with your package
package com.mypackage;

import android.view.Choreographer;
import android.view.View;
import android.widget.FrameLayout;

import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.fragment.app.FragmentActivity;

import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReadableArray;
import com.facebook.react.common.MapBuilder;
import com.facebook.react.uimanager.annotations.ReactProp;
import com.facebook.react.uimanager.annotations.ReactPropGroup;
import com.facebook.react.uimanager.ViewGroupManager;
import com.facebook.react.uimanager.ThemedReactContext;

import java.util.Map;

public class MyViewManager extends ViewGroupManager<FrameLayout> {

    public static final String REACT_CLASS = "MyViewManager";
    public final int COMMAND_CREATE = 1;
    private int propWidth;
    private int propHeight;

    ReactApplicationContext reactContext;

    public MyViewManager(ReactApplicationContext reactContext) {
        this.reactContext = reactContext;
    }

    @Override
    public String getName() {
        return REACT_CLASS;
    }
}
```

```
/**
 * Return a FrameLayout which will later hold the Fragment
 */
@Override
public FrameLayout createViewInstance(ThemedReactContext reactContext) {
    return new FrameLayout(reactContext);
}

/**
 * Map the "create" command to an integer
 */
@Nullable
@Override
public Map<String, Integer> getCommandsMap() {
    return MapBuilder.of("create", COMMAND_CREATE);
}

/**
 * Handle "create" command (called from JS) and call createFragment method
 */
@Override
public void receiveCommand(
    @NonNull FrameLayout root,
    String commandId,
    @Nullable ReadableArray args
) {
    super.receiveCommand(root, commandId, args);
    int reactNativeViewId = args.getInt(0);
    int commandIdInt = Integer.parseInt(commandId);

    switch (commandIdInt) {
        case COMMAND_CREATE:
            createFragment(root, reactNativeViewId);
            break;
        default: {}
    }
}

@ReactPropGroup(names = {"width", "height"}, customType = "Style")
public void setStyle(FrameLayout view, int index, Integer value) {
    if (index == 0) {
        propWidth = value;
    }

    if (index == 1) {
        propHeight = value;
    }
}
```

```

    }

    /**
     * Replace your React Native view with a custom fragment
     */
    public void createFragment(FrameLayout root, int reactNativeViewId) {
        ViewGroup parentView = (ViewGroup) root.findViewById(reactNativeViewId);
        setupLayout(parentView);

        final MyFragment myFragment = new MyFragment();
        FragmentActivity activity = (FragmentActivity) reactContext.getCurrentActivity();
        activity.getSupportFragmentManager()
            .beginTransaction()
            .replace(reactNativeViewId, myFragment, String.valueOf(reactNativeViewId))
            .commit();
    }

    public void setupLayout(View view) {
        Choreographer.getInstance().postFrameCallback(new Choreographer.FrameCallback() {
            @Override
            public void doFrame(long frameTimeNanos) {
                manuallyLayoutChildren(view);
                view.getViewTreeObserver().dispatchOnGlobalLayout();
                Choreographer.getInstance().postFrameCallback(this);
            }
        });
    }

    /**
     * Layout all children properly
     */
    public void manuallyLayoutChildren(View view) {
        // propWidth and propHeight coming from react-native props
        int width = propWidth;
        int height = propHeight;

        view.measure(
            View.MeasureSpec.makeMeasureSpec(width, View.MeasureSpec.EXACTLY),
            View.MeasureSpec.makeMeasureSpec(height, View.MeasureSpec.EXACTLY));

        view.layout(0, 0, width, height);
    }
}

```

4. Register the viewManager

Java Kotlin

MyPackage.java

```
// replace with your package
package com.mypackage;

import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.Arrays;
import java.util.List;

public class MyPackage implements ReactPackage {

    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext reactContext) {
        return Arrays.<ViewManager>asList(
            new MyViewManager(reactContext)
        );
    }
}
```

5. Register the Package

Java Kotlin

MainApplication.java

```
@Override
protected List<ReactPackage> getPackages() {
    List<ReactPackage> packages = new PackageList(this).getPackages();
    ...
    packages.add(new MyPackage());
    return packages;
}
```

6. Implement the JavaScript module

I. Start with custom View manager:

MyViewManager.tsx

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

export const MyViewManager =
  requireNativeComponent('MyViewManager');
```

II. Then implement custom View calling the create method:

MyView.tsx

```
import React, {useEffect, useRef} from 'react';
import {
  PixelRatio,
  UIManager,
  findNodeHandle,
} from 'react-native';

import {MyViewManager} from './my-view-manager';

const createFragment = viewId =>
  UIManager.dispatchViewManagerCommand(
    viewId,
    // we are calling the 'create' command
    UIManager.MyViewManager.Commands.create.toString(),
    [viewId],
  );

export const MyView = () => {
  const ref = useRef(null);


  useEffect(() => {
    const viewId = findNodeHandle(ref.current);
    createFragment(viewId);
  }, []);

  return (
    <MyViewManager
```

```
    style={{
      // converts dpi to px, provide desired height
      height: PixelRatio.getPixelSizeForLayoutSize(200),
      // converts dpi to px, provide desired width
      width: PixelRatio.getPixelSizeForLayoutSize(200),
    }}
    ref={ref}
  />
);
};
```

If you want to expose property setters using `@ReactProp` (or `@ReactPropGroup`) annotation see the [ImageView example](#) above.

Is this page useful?  

 Edit this page

Last updated on **Jun 21, 2023**