Native UI Components

Edit

> **This documentation and the underlying platform code is a work in progress.**
> **Examples (C# and C++/WinRT):**
>
> - Native Module Sample in `microsoft/react-native-windows-samples`
> - Sample App in `microsoft/react-native-windows/packages/microsoft-reactnative-sampleapps`

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 UWP 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.

Overview

Similarly to authoring native modules, at a high level you must:

application.

3. Reference the new component within your React Native JSX code.

Note about UWP XAML controls

Some UWP XAML controls do not support being hosted in environments where 3D transforms are involved (i.e. the `Transform3D` property is set on the control or on any of the control's ancestors in the XAML tree).

Currently, React Native for Windows uses a global `PerspectiveTransform` to provide a 3D look to objects being rotated along the `x` or `y` axes, which means these controls that do not work in 3D environments, will not work out of the box (e.g. `InkCanvas`). However, a React Native for Windows app can opt out of the 3D perspective (and in so doing, enable using these controls) by setting the `IsPerspectiveEnabled` property on the `ReactRootView`.

Important: The `IsPerspectiveEnabled` property is experimental and support for it may be removed in the future.

Initial Setup

Prerequisite: Follow the Native Modules Setup Guide to create the Visual Studio infrastructure to author your own stand-alone native module for React Native Windows

Once you have set up your development environment and project structure, you are ready to write code.

If you are only planning on adding a native module to your existing React Native Windows app, i.e.:

1. You followed Getting Started, where
2. You ran `npx react-native-windows-init --overwrite` to add Windows to your project, and

If you are instead creating a standalone native module, or adding Windows support to an existing native module, check out the Native Modules Setup guide first.

Sample view manager

C# C++

Attributes

ATTRIBUTE	USE
`ViewManagerExportedViewConstant`	Specifies a field or property that represents a constant.
`ViewManagerProperty`	Specifies a method to be called to set a property on an instance of a native UI widget.
`ViewManagerCommand`	Specifies a method that can be called on an instance of a native UI widget.

For this sample, assume we have the following `CustomUserControl` that we want to use in React Native.

`CustomUserControl.cs` :

Copy

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace ViewManagerSample
{
    public sealed class CustomUserControl : Control
    {
        public static DependencyProperty LabelProperty { get; private set; }
```

```
            }
            set
            {
                SetValue(LabelProperty, value);
            }
        }

        static CustomUserControl()
        {
            LabelProperty = DependencyProperty.Register(
                nameof(Label),
                typeof(string),
                typeof(CustomUserControl),
                new PropertyMetadata(default(string))
                );
        }

        public CustomUserControl()
        {
            DefaultStyleKey = typeof(CustomUserControl);
        }
    }
}
```

Copy

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ViewManagerSample">

    <Style TargetType="local:CustomUserControl" >
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="local:CustomUserControl">
                    <Border
                        Background="{TemplateBinding Background}"
                        BorderBrush="{TemplateBinding BorderBrush}"
                        BorderThickness="{TemplateBinding BorderThickness}">
                        <TextBlock Foreground="{TemplateBinding Foreground}" Text="{Templat
                    </Border>
                </ControlTemplate>
```

1. Authoring your View Manager

Here is a sample view manager written in C# called `CustomUserControlViewManager`.

`CustomUserControlViewManager.cs`:

Copy

```
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Controls;

using Microsoft.ReactNative.Managed;
using System.Collections.Generic;

namespace ViewManagerSample
{
    internal class CustomUserControlViewManager : AttributedViewManager<CustomUserControl>
    {
        [ViewManagerProperty("label")]
        public void SetLabel(CustomUserControl view, string value)
        {
            if (null != value)
            {
                view.Label = value;
            }
            else
            {
                view.ClearValue(CustomUserControl.LabelProperty);
            }
        }

        [ViewManagerProperty("color")]
        public void SetColor(CustomUserControl view, Brush value)
        {
            if (null != value)
            {
                view.Foreground = value;
            }
            else
            {
```

```
        public void SetBackgroundColor(CustomUserControl view, Brush value)
        {
            if (null != value)
            {
                view.Background = value;
            }
            else
            {
                view.ClearValue(Control.BackgroundProperty);
            }
        }

        [ViewManagerCommand]
        public void CustomCommand(CustomUserControl view, IReadOnlyList<object> commandArgs
        {
            // Execute command
        }
    }
}
```

2. Registering your View Manager

As with native modules, we want to register our new `CustomUserControlViewManager` with React Native so we can actually use it. To do this, first we're going to create a `ReactPackageProvider` which implements `Microsoft.ReactNative.IReactPackageProvider`.

`ReactPackageProvider.cs`：

Copy

```
using Microsoft.ReactNative.Managed;

namespace ViewManagerSample
{
    public partial class ReactPackageProvider : IReactPackageProvider
    {
        public void CreatePackage(IReactPackageBuilder packageBuilder)
        {
            CreatePackageImplementation(packageBuilder);
```

```
        partial void CreatePackageImplementation(IReactPackageBuilder packageBuilder);
    }
}
```

Here we've implemented the `CreatePackage` method, which receives `packageBuilder` to build contents of the package.

Now that we have the `ReactPackageProvider`, it's time to register it within our `ReactApplication`. We do that by simply adding the provider to the `PackageProviders` property.

`App.xaml.cs`:

Copy

```
using Microsoft.ReactNative;

namespace SampleApp
{
    sealed partial class App : ReactApplication
    {
        public App()
        {
            /* Other Init Code */

            PackageProviders.Add(new Microsoft.ReactNative.Managed.ReactPackageProvider());
            PackageProviders.Add(new ViewManagerSample.ReactPackageProvider());

            /* Other Init Code */
        }
    }
}
```

This example assumes that the `ViewManagerSample.ReactPackageProvider` we created above is in a different project (assembly) than our application. However you'll notice that by default we also added a `Microsoft.ReactNative.Managed.ReactPackageProvider`.

actually want to define a separate `ReactPackageProvider`.

More extensibility points

- In some scenarios, a view manager might need to have more context at view creation time in order to decide what kind of control to instantiate. This can be achieved by having the view manager implement the `IViewManagerCreateWithProperties` interface. The `CreateViewWithProperties` method can then access the properties set in JSX by inspecting the `propertyMapReader`.

```
-internal class CustomUserControlViewManager : AttributedViewManager<CustomUserCon ol> {
+internal class CustomUserControlViewManager : AttributedViewManager<CustomUserControl>, IV
// rest of the view manager goes here...
+  // IViewManagerCreateWithProperties
+  public virtual object CreateViewWithProperties(Microsoft.ReactNative.IJSValueReader prop
+    propertyMapReader.ReaderValue(out IDictionary<string, JSValue> propertyMap);
+    // create a XAML FrameworkElement based on properties in propertyMap
+    if (propertyMap.ContainsKey("foo")) {
+      return new Button();
+    } else {
+      return new TextBox();
+    }
+  }
}
+}
```

- Your view manager is also able to declare that it wants to be responsible for its own sizing and layout. This is useful in scenarios where you are wrapping a native XAML control. To do so, implement the `Microsoft.ReactNative.IViewManagerRequiresNativeLayout` interface:

```
-internal class CustomUserControlViewManager : AttributedViewManager<CustomUserCon ol> {
+internal class CustomUserControlViewManager : AttributedViewManager<CustomUserControl>, IV
// rest of the view manager goes here...
```

3. Using your View Manager in JSX

ViewManagerSample.js :

Copy

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Button,
  requireNativeComponent,
  StyleSheet,
  UIManager,
  View,
} from 'react-native';

let CustomUserControl = requireNativeComponent('CustomUserControl');

class ViewManagerSample extends Component {
  onPress() {
    if (_customControlRef) {
      const tag = findNodeHandle(this._customControlRef);
      UIManager.dispatchViewManagerCommand(tag,
        UIManager.getViewManagerConfig('CustomUserControl').Commands.CustomCommand,
        ['arg1', 'arg2']);
    }
  }

  render() {
    return (
      <View style={styles.container}>
         <CustomUserControl style={styles.customcontrol}
           label="CustomUserControl!"
           ref={(ref) => { this._customControlRef = ref; }} />
         <Button onPress={() => { this.onPress(); }}
           title="Call CustomUserControl Commands!" />
      </View>);
  }
}

const styles = StyleSheet.create({
  container: {
```

```
  customcontrol: {
    color: '#333333',
    backgroundColor: '#006666',
    width: 200,
    height: 20,
    margin: 10,
  },
});

AppRegistry.registerComponent('ViewManagerSample', () => ViewManagerSample);
```

‹ Native Modules

Native Module Setup ›