Prerequisites for Libraries

> ⚠ **CAUTION**
>
> This documentation is still **experimental** and details are subject to changes as we iterate. Feel free to share your feedback on the discussion inside the working group for this page.
>
> Moreover, it contains several **manual steps**. Please note that this won't be representative of the final developer experience once the New Architecture is stable. We're working on tools, templates and libraries to help you get started fast on the New Architecture, without having to go through the whole setup.

The following steps will help ensure your modules and components are ready for the New Architecture.

Define Specs in JavaScript

The JavaScript specs serve as the source of truth for the methods provided by each native module. They define all APIs provided by the native module, along with the types of those constants and functions. Using a **typed** spec file allows you to be intentional and declare all the input arguments and outputs of your native module's methods.

> ⓘ **INFO**
>
> **TypeScript** support is in beta right now.

To adopt the New Architecture, you start by creating these specs for your native modules and native components. You can do this before migrating to the New Architecture: the specs will be used later on to generate native interface code for all the supported platforms as a way to enforce uniform APIs across platforms.

Turbo Native Modules

JavaScript spec files **must** be named `Native<MODULE_NAME>.js`, and they export a `TurboModuleRegistry Spec` object. The name convention is important because the Codegen process looks for modules whose `js` (`jsx`, `ts`, or `tsx`) spec file starts with the keyword `Native`.

The following is a basic JavaScript spec template, written using the Flow and TypeScript syntax.

Flow TypeScript

```
// @flow strict

import type {TurboModule} from 'react-native/Libraries/TurboModule/RCTExport';
import {TurboModuleRegistry} from 'react-native';

export interface Spec extends TurboModule {
  +getConstants: () => {||};

  // your module methods go here, for example:
  getString(id: string): Promise<string>;
}

export default (TurboModuleRegistry.get<Spec>(
  '<MODULE_NAME>',
): ?Spec);
```

Fabric Native Components

JavaScript spec files **must** be named `<FABRIC COMPONENT>NativeComponent.js` (for TypeScript use extension `.ts` or `.tsx`) and they export a `HostComponent` object. The name convention is important: the Codegen process looks for components whose spec file (either JavaScript or TypeScript) ends with the suffix `NativeComponent`.

The following snippet shows a basic JavaScript spec template, written in Flow as well as TypeScript.

Flow TypeScript

```
// @flow strict-local

import type {ViewProps} from 'react-native/Libraries/Components/View/ViewPropTypes';
import type {HostComponent} from 'react-native';
import codegenNativeComponent from 'react-
native/Libraries/Utilities/codegenNativeComponent';

type NativeProps = $ReadOnly<{|
  ...ViewProps,
  // add other props here
|}>;

export default (codegenNativeComponent<NativeProps>(
  '<FABRIC COMPONENT>',
): HostComponent<NativeProps>);
```

Supported Types

When using Flow or TypeScript, you will be using type annotations to define your spec. Keeping in mind that the goal of defining a JavaScript spec is to ensure the generated native interface code is type-safe, the set of supported types will be those that can be mapped one-to-one to a corresponding type on the native platform.

In general, this means you can use primitive types (strings, numbers, booleans), function types, object types, and array types. Union types, on the other hand, are not supported. All types must be read-only. For Flow: either `+` or `$ReadOnly<>` or `{||}` objects. For TypeScript: `readonly` for properties, `Readonly<>` for objects, and `ReadonlyArray<>` for arrays.

> See Appendix II. Flow Type to Native Type Mapping. See Appendix III. TypeScript to Native Type Mapping.

Codegen Helper Types

You can use predefined types for your JavaScript spec, here is a list of them:

- `Double`

- `Float`
- `Int32`
- `UnsafeObject`
- `WithDefault<Type, Value>` - Sets default value for type
- `BubblingEventHandler<T>` - For events that are propagated (bubbled) up the component tree from child to parent up to the root (eg: `onStartShouldSetResponder`).
- `DirectEventHandler<T>` - For events that are called only on element recieving the event (eg: `onClick`) and don't bubble.

Later on those types are compiled to coresponding equivalents on target platforms.

Be Consistent Across Platforms and Eliminate Type Ambiguity

Before adopting the New Architecture in your native module, you should ensure your methods are consistent across platforms. You will realize this as you set out to write the JavaScript spec for your native module - remember that JavaScript spec defines what the methods will look like on all supported platforms.

If your existing native module has methods with the same name on multiple platforms, but with different numbers or types of arguments across platforms, you will need to find a way to make these consistent. If you have methods that can take two or more different types for the same argument, then you need to find a way to resolve this type of ambiguity as type unions are intentionally not supported.

Configure Codegen

Codegen is a tool that runs when you build an Android app or install the dependencies of an iOS app. It creates some scaffolding code that you won't have to create manually.

Codegen can be configured in the `package.json` file of your Library. Add the following JSON object at the end of it.

```
   },
+  "codegenConfig": {
+    "name": "<library name>",
+    "type": "all",
+    "jsSrcsDir": ".",
+    "android": {
+      "javaPackageName": "com.facebook.fbreact.specs"
+    }
+  }
}
```

- The `codegenConfig` is the key used by the Codegen to verify that there is some code to generate.
- The `name` field is the name of the library.
- The `type` field is used to identify the type of module we want to create. We suggest keeping `all` to support libraries that contain both Turbo Native Module and Fabric Native Components.
- The `jsSrcsDir` is the directory where the codegen will start looking for JavaScript specs.
- The `android.javaPackageName` is the name of the package where the generated code ends up.

Android also requires to have the React Gradle Plugin properly configured in your app.

Migrating from `UIManager` JavaScript APIs

In the New Architecture, most `UIManager` methods will become available as instance methods on native component instances obtained via `ref`:

```
function MyComponent(props: Props) {
  const viewRef = useRef(null);

  useEffect(() => {
    viewRef.current.measure(((left, top, width, height, pageX, pageY) => {
      // ...
    });
  }, []);

  return <View ref={viewRef} />;
}
```

This new API design provides several benefits:

- Better developer ergonomics by removing the need for separately importing `UIManager` or calling `findNodeHandle`.
- Better performance by avoiding the node handle lookup step.
- Directionally aligned with the analogous deprecation of `findDOMNode`.

We will eventually deprecate `UIManager`. However, we recognize that migrations demand a high cost for many application and library authors. In order to minimize this cost, we plan to continue supporting as many of the methods on `UIManager` as possible in the New Architecture.

Support for `UIManager` methods in the New Architecture is actively being developed. While we make progress here, early adopters can still experiment with the New Architecture by following these steps to migrate off common `UIManager` APIs:

1. Move the call to `requireNativeComponent` to a separate file
2. Migrating off `dispatchViewManagerCommand`
3. Creating NativeCommands with `codegenNativeCommands`

Move the call to `requireNativeComponent` to a separate file

This will prepare for the JS to be ready for the new codegen system for the New Architecture. The new file should be named `<ComponentName>NativeComponent.js`.

Old way

```
const RNTMyNativeView = requireNativeComponent('RNTMyNativeView');

[...]

return <RNTMyNativeView />;
```

New way

RNTMyNativeNativeComponent.js

```
import RNTMyNativeViewNativeComponent from './RNTMyNativeViewNativeComponent';

[...]

return <RNTMyNativeViewNativeComponent />;
```

RNTMyNativeViewNativeComponent.js

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

const RNTMyNativeViewNativeComponent = requireNativeComponent(
  'RNTMyNativeView',
);

export default RNTMyNativeViewNativeComponent;
```

Flow support

If `requireNativeComponent` is not typed, you can temporarily use the `mixed` type to fix the Flow warning, for example:

```
// @flow strict-local

import type {HostComponent} from 'react-native/Libraries/Renderer/shims/ReactNativeTypes';
// ...
const RCTWebViewNativeComponent: HostComponent<mixed> =
  requireNativeComponent<mixed>('RNTMyNativeView');
```

Later on you can replace `requireNativeComponent`

When you are ready to migrate to Fabric you can replace `requireNativeComponent` with `codegenNativeComponent`:

RNTMyNativeViewNativeComponent.js

```
// @flow strict-local

export default (codegenNativeComponent<NativeProps>(
  'RNTMyNativeView',
): HostComponent<NativeProps>);
```

And update the main file:

RNTMyNativeNativeComponent.js

```
// @flow strict-local

export default require('./RNTMyNativeViewNativeComponent')
  .default;
```

Migrating off `dispatchViewManagerCommand`

Similar to the one above, in an effort to avoid calling methods on the UIManager, all view manager methods are now called through an instance of `NativeCommands`. `codegenNativeCommands` is a new API to code-generate `NativeCommands` given an interface of your view manager's commands.

Before

```
class MyComponent extends React.Component<Props> {
  _moveToRegion: (region: Region, duration: number) => {
    UIManager.dispatchViewManagerCommand(
      ReactNative.findNodeHandle(this),
      'moveToRegion',
      [region, duration]
    );
  }

  render() {
    return <MyCustomMapNativeComponent onPress={this._moveToRegion} />
  }
}
```

Creating NativeCommands with `codegenNativeCommands`

MyCustomMapNativeComponent.js

```
// @flow strict-local

import codegenNativeCommands from 'react-native/Libraries/Utilities/codegenNativeCommands';
import type {HostComponent} from 'react-native/Libraries/Renderer/shims/ReactNativeTypes';

type MyCustomMapNativeComponentType = HostComponent<NativeProps>;

interface NativeCommands {
  +moveToRegion: (
    viewRef: React.ElementRef<MyCustomMapNativeComponentType>,
    region: MapRegion,
    duration: number,
  ) => void;
}

export const Commands: NativeCommands =
  codegenNativeCommands<NativeCommands>({
    supportedCommands: ['moveToRegion'],
  });
```

Note:

- The first argument in the `moveToRegion` command is a HostComponent ref of the native component
- The arguments to the `moveToRegion` command are enumerated in the signature
- The command definition is co-located with the native component. This is an encouraged pattern
- Ensure you have included your command name in the `supportedCommands` array

Using Your Command

```
// @flow strict-local

import {Commands, ...} from './MyCustomMapNativeComponent';
```

```
class MyComponent extends React.Component<Props> {
  _ref: ?React.ElementRef<typeof MyCustomMapNativeComponent>;

  _captureRef: (ref: React.ElementRef<typeof MyCustomMapNativeComponent>) => {
    this._ref = ref;
  }

  _moveToRegion: (region: Region, duration: number) => {
    if (this._ref != null) {
      Commands.moveToRegion(this._ref, region, duration);
    }
  }

  render() {
    return <MyCustomMapNativeComponent
       ref={this._captureRef}
       onPress={this._moveToRegion} />
  }
}
```

Updating Native Implementation

In the example, the code-generated `Commands` will dispatch `moveToRegion` call to the native component’s view manager. In addition to writing the JS interface, you’ll need to update your native implementation signatures to match the dispatched method call. See the mapping for Android argument types andiOS argument types for reference.

iOS

```
RCT_EXPORT_METHOD(moveToRegion:(nonnull NSNumber *)reactTag
                        region:(NSDictionary *)region
                      duration:(double)duration
{
   ...
}
```

Android

Java Kotlin

```
// receiveCommand signature has changed to receive String commandId
@Override
  public void receiveCommand(
      ReactMapDrawerView view, String commandId, @Nullable ReadableArray args) {
    switch (commandId) {
      case "moveToRegion":
        if (args == null) {
          break;
        }

        ReadableMap region = args.getMap(0);
        int durationMs = args.getInt(1);
        // ... act on the view...
        break;
    }
  }
```

Is this page useful? 👍 👎

✎ Edit this page

Last updated on ***Jun 21, 2023***