Appendix

> **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.

I. Terminology

The whole New Architecture related guides will stick to the following **terminology**:

- **Legacy Native Components** - To refer to Components which are running on the old React Native architecture.
- **Legacy Native Modules** - To refer to Modules which are running on the old React Native architecture.
- **Fabric Native Components** - To refer to Components which have been adapted to work well with the New Architecture, namely the new renderer. For brevity you might find them referred as **Fabric Components**.
- **Turbo Native Modules** - To refer to Modules which have been adapted to work well with the New Architecture, namely the new Native Module System. For brevity you might find them referred as **Turbo Modules**

II. Flow Type to Native Type Mapping

You may use the following table as a reference for which types are supported and what they map to in each platform:

`string`

NULLABLE SUPPORT?	`?string`
ANDROID (JAVA)	`String`
IOS	`NSString`

`boolean`

NULLABLE SUPPORT?	`?boolean`
ANDROID (JAVA)	`Boolean`
IOS	`NSNumber`

Object literal

This is recommended over using plain `Object`, for type safety.

Example: `{| foo: string, ... |}`

NULLABLE SUPPORT?	`?{| foo: string, ...|}`
ANDROID (JAVA)	-
IOS	-

`Object`

> **NOTE**
>
> Recommended to use Object literal instead.

NULLABLE SUPPORT?	`?Object`
ANDROID (JAVA)	`ReadableMap`

IOS	`@` (untyped dictionary)

`Array<*>`

NULLABLE SUPPORT?	`?Array<*>`
ANDROID (JAVA)	`ReadableArray`
IOS	`NSArray` (or `RCTConvertVecToArray` when used inside objects)

`Function`

NULLABLE SUPPORT?	`?Function`
ANDROID (JAVA)	-
IOS	-

`Promise<*>`

NULLABLE SUPPORT?	`?Promise<*>`
ANDROID (JAVA)	`com.facebook.react.bridge.Promise`
IOS	`RCTPromiseResolve` and `RCTPromiseRejectBlock`

Type Unions

Type unions are only supported as callbacks.

Example: `'SUCCESS' | 'FAIL'`

NULLABLE SUPPORT?	Only as callbacks.
ANDROID (JAVA)	-
IOS	-

Callbacks

Callback functions are not type checked, and are generalized as `Object`s.

Example: `() =>`

NULLABLE SUPPORT?	Yes
ANDROID (JAVA)	`com.facebook.react.bridge.Callback`
IOS	`RCTResponseSenderBlock`

> **NOTE**
>
> You may also find it useful to refer to the JavaScript specifications for the core modules in React Native. These are located inside the `Libraries/` directory in the React Native repository.

III. TypeScript to Native Type Mapping

You may use the following table as a reference for which types are supported and what they map to in each platform:

`string`

NULLABLE SUPPORT?	`string | null`
ANDROID (JAVA)	`String`
IOS	`NSString`

`boolean`

NULLABLE SUPPORT?	`boolean | null`
ANDROID (JAVA)	`Boolean`

IOS	`NSNumber`

`number`

NULLABLE SUPPORT?	No
ANDROID (JAVA)	`double`
IOS	`NSNumber`

Object literal

This is recommended over using plain `Object`, for type safety.

Example: `{| foo: string, ... |}`

NULLABLE SUPPORT?	`{| foo: string, ...|}	null`
ANDROID (JAVA)	-
IOS	-

`Object`

> **NOTE**
>
> Recommended to use Object literal instead.

NULLABLE SUPPORT?	`Object	null`
ANDROID (JAVA)	`ReadableMap`
IOS	`@` (untyped dictionary)

`Array<*>`

NULLABLE SUPPORT?	`Array<*> | null`
ANDROID (JAVA)	`ReadableArray`
IOS	`NSArray` (or `RCTConvertVecToArray` when used inside objects)

`Function`

NULLABLE SUPPORT?	`?Function | null`
ANDROID (JAVA)	-
IOS	-

`Promise<*>`

NULLABLE SUPPORT?	`Promise<*> | null`
ANDROID (JAVA)	`com.facebook.react.bridge.Promise`
IOS	`RCTPromiseResolve` and `RCTPromiseRejectBlock`

Type Unions

Type unions are only supported as callbacks.

Example: `'SUCCESS' | 'FAIL'`

NULLABLE SUPPORT?	Only as callbacks.
ANDROID (JAVA)	-
IOS	-

Callbacks

Callback functions are not type checked, and are generalized as `Object`s.

Example: `() =>`

NULLABLE SUPPORT?	Yes
ANDROID (JAVA)	`com.facebook.react.bridge.Callback`
IOS	`RCTResponseSenderBlock`

You may also find it useful to refer to the JavaScript specifications for the core modules in React Native. These are located inside the `Libraries/` directory in the React Native repository.

IV. Invoking the code-gen during development

> This section contains information specific to v0.66 of React Native.

The Codegen is typically invoked at build time, but you may find it useful to generate your native interface code on demand for troubleshooting.

If you wish to invoke the codegen manually, you have two options:

1. Invoking a Gradle task directly (Android).
2. Invoking a script manually.

Android - Invoking a Gradle task directly

You can trigger the Codegen by invoking the following task:

```
./gradlew generateCodegenArtifactsFromSchema --rerun-tasks
```

The extra `--rerun-tasks` flag is added to make sure Gradle is ignoring the `UP-TO-DATE` checks for this task. You should not need it during normal development.

The `generateCodegenArtifactsFromSchema` task normally runs before the `preBuild` task, so you should not need to invoke it manually, but it will be triggered before your builds.

Invoking the script manually

Alternatively, you can invoke the Codegen directly, bypassing the Gradle Plugin or CocoaPods infrastructure. This can be done with the following commands.

The parameters to provide will look quite familiar to you now that you have already configured the Gradle plugin or CocoaPods library.

Generating the schema file

First, you'll need to generate a schema file from your JavaScript sources. You only need to do this whenever your JavaScript specs change. The script to generate this schema is provided as part of the `react-native-codegen` package. If running this from within your React Native application, you can use the package from `node_modules` directly:

```
node node_modules/react-native-codegen/lib/cli/combine/combine-js-to-schema-cli.js \
  <output_file_schema_json> <javascript_sources_dir>
```

> The source for the `react-native-codegen` is available in the React Native repository, under `packages/react-native-codegen`. Run `yarn install` and `yarn build` in that directory to build your own `react-native-codegen` package from source. In most cases, you will not want to do this as the guide assumes the use of the `react-native-codegen` package version that is associated with the relevant React Native nightly release.

Generating the native code artifacts

Once you have a schema file for your native modules or components, you can use a second script to generate the actual native code artifacts for your library. You can use the same schema file generated by the previous script.

```
node node_modules/react-native/scripts/generate-specs-cli.js \
  --platform <ios|android> \
  --schemaPath <generated_schema_json_file> \
  --outputDir <output_dir> \
  [--libraryName library_name] \
```

```
    [--javaPackageName java_package_name] \
    [--libraryType all(default)|modules|components]
```

> **NOTE:** The output artifacts of the Codegen are inside the build folder and should not be committed. They should be considered only for reference.

Example

The following is a basic example of invoking the Codegen script to generate native iOS interface code for a library that provides native modules. The JavaScript spec sources for this library are located in a `js/` subdirectory, and this library's native code expects the native interfaces to be available in the `ios` subdirectory.

```
# Generate schema - only needs to be done whenever JS specs change
node node_modules/react-native-codegen/lib/cli/combine/combine-js-to-schema-cli.js
/tmp/schema.json ./js

# Generate native code artifacts
node node_modules/react-native/scripts/generate-specs-cli.js \
  --platform ios \
  --schemaPath /tmp/schema.json \
  --outputDir ./ios \
  --libraryName MyLibSpecs \
  --libraryType modules
```

In the above example, the code-gen script will generate several files: `MyLibSpecs.h` and `MyLibSpecs-generated.mm`, as well as a handful of `.h` and `.cpp` files, all located in the `ios` directory.

V. Note on Existing Apps

This guide provides instructions for migrating an application that is based on the default app template that is provided by React Native. If your app has deviated from the template, or you are working with an application that was never based off the template, then the following sections might help.

Finding your bridge delegate

This guide assumes that the `AppDelegate` is configured as the bridge delegate. If you are not sure which is your bridge delegate, then place a breakpoint in `RCTBridge` and `RCTCxxBridge`, run your app, and inspect `self.delegate`.

Is this page useful? 👍 👎

Edit this page

*Last updated on **Aug 18, 2023***