Codegen

> **⚠ 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 **Codegen** is not a proper pillar, but it is a tool that can be used to avoid writing a lot of repetitive code. Using **Codegen** is not mandatory: all the code that is generated by it can also be written manually. However, it generates scaffolding code that could save you a lot of time.

The **Codegen** is invoked automatically by React Native every time an iOS or Android app is built. Occasionally, you would like to run the scripts that generate the code manually to know which types and files are actually generated: this is a common scenario when developing **Turbo Native Modules** and **Fabric Native Components**, for example.

This guide teaches how to configure the **Codegen**, and how to invoke it manually for each platform, and describes the generated code.

Prerequisites

You always need a React Native app to generate the code properly, even when invoking the **Codegen** manually.

The **Codegen** process is tightly coupled with the build of the app, and the scripts are located in the `react-native` NPM package.

For the sake of this guide, create a project using the React Native CLI as follows:

```
npx react-native init SampleApp --version 0.70.0
```

> **NOTE**
>
> This guide assumes that the React Native version in use is 0.70.0 or greater. Previous versions of React Native uses a version of **Codegen** that requires a slightly different setup.

Then, add the module that requires the **Codegen** as an NPM dependency of the app:

```
yarn add <path/to/your/TurboNativeModule_or_FabricNativeComponent>
```

See how to create a Turbo Native Module or a Fabric Native Component to get more information on how to configure them.

The rest of this guide assumes that you have a `Turbo Native Module` and/or a `Fabric Native Component` properly set up.

iOS

Running the Codegen

The **Codegen** for iOS relies on some Node scripts that are invoked during the build process. The scripts are located in the `MyApp/node_modules/react-native/scripts/` folder.

The script that you have to run is the `generate-codegen-artifacts.js` script. This searches among all the dependencies of the app, looking for JS files that respects some specific conventions (look at TurboModules and Fabric Components sections for details), and it generates the required code.

To invoke the script, you can run this command from the root folder of your app:

```
node node_modules/react-native/scripts/generate-codegen-artifacts.js \
    --path SampleApp/ \
```

```
    --outputPath <an/output/path> \
```

Given that the app has `Turbo Native Modules` and/or `Fabric Native Components` configured as a dependency, **Codegen** looks for all of them and generates the code in the path you provided.

The Generated Code

If you run the **Codegen** in your app with an output path of `codegen`, for example, you obtain the following structure:

```
codegen
└── build
    └── generated
        └── ios
            ├── MyTurboModuleSpecs
            │   ├── MyTurboModuleSpecs-generated.mm
            │   └── MyTurboModuleSpecs.h
            ├── FBReactNativeSpec
            │   ├── FBReactNativeSpec-generated.mm
            │   └── FBReactNativeSpec.h
            ├── RCTThirdPartyFabricComponentsProvider.h
            ├── RCTThirdPartyFabricComponentsProvider.mm
            └── react
                └── renderer
                    └── components
                        ├── MyFabricComponent
                        │   ├── ComponentDescriptors.h
                        │   ├── EventEmitters.cpp
                        │   ├── EventEmitters.h
                        │   ├── Props.cpp
                        │   ├── Props.h
                        │   ├── RCTComponentViewHelpers.h
                        │   ├── ShadowNodes.cpp
                        │   └── ShadowNodes.h
                        └── rncore
                            ├── ComponentDescriptors.h
                            ├── EventEmitters.cpp
                            ├── EventEmitters.h
                            ├── Props.cpp
                            ├── Props.h
                            ├── RCTComponentViewHelpers.h
```

```
├── ShadowNodes.cpp
└── ShadowNodes.h
```

The `codegen` folder sits at the root of the hierarchy, as expected. Nested into it, there are two more folders: `build/generated`.

Then, there is an `ios` folder that contains:

- A custom folder for each TurboModule.
- The header (`.h`) and implementation (`.mm`) files for the `RCTThirdPartyFabricComponentsProvider`.
- A base `react/renderer/components` folder which contains a custom folder for each `Fabric Native Component`.

In the example above, there are both a TurboModule and a set of Fabric Native Components. These are generated by React Native itself: `FBReactNativeSpec` and `rncore`. These modules will always appear even if you don't have any extra TurboModule or Fabric Native Component: React Native requires them in order to work properly.

Turbo Native Modules

Each folder contains two files: an interface file and an implementation file.

The interface files have the same name as that of the Turbo Native Module and contain methods to initialize the JSI interface.

The implementation files, instead, have the `-generated` suffix and contain the logic to invoke the native methods from JS and vice-versa.

Fabric Native Components

The content of each Fabric Native Component folder contains several files. The basic element for a Fabric Native Component is the `ShadowNode`: it represents a node in the React abstract tree. The `ShadowNode` represents a React entity; therefore, it could need some props, which are defined in the `Props` files and, sometimes, an `EventEmitter`, defined in the corresponding file.

Additionally, the **Codegen** also creates a `ComponentDescriptor.h` and an `RCTComponentViewHelpers.h` files: the first one is used by React Native and Fabric to properly get a reference to the Fabric Native Component, while the latter contains some helper methods and protocols that can be implemented by the Native View to properly respond to JSI invocations.

For further details about how Fabric works, have a look at the Renderer section.

RCTThirdPartyFabricComponentsProvider

These are interface and implementation files for a registry. React Native uses this registry at runtime to retrieve the right class for a required Fabric Native Component. Once React Native has a handle to that class, it can instantiate it.

Android

Running the Codegen

Android `Codegen` relies on a Gradle task to generate the required code. First, you need to configure the Android app to work with the New Architecture; otherwise, the Gradle task fails.

1. Open the `MyApp/android/gradle.properties` file.
2. Flip the `newArchEnabled` flag from `false` to `true`.

After that, you can navigate into the `SampleApp/android` folder and run:

```
./gradlew generateCodegenArtifactsFromSchema
```

These tasks invoke the `generateCodegenArtifactsFromSchema` on all the the imported projects of the app (the app and all the node modules which are linked to it). It generates the code in the corresponding `node_modules/<dependency>` folder. So, for example, if you have a Fabric Native Component whose node module is called `my-fabric-component`, the

generated code is located in the `SampleApp/node_modules/my-fabric-component/android/build/generated/source/codegen` path.

The Generated Code

Once the Gradle task completes, you can see different structures for a Turbo Native Module or for a Fabric Native Component. The following tab shows how they appear:

turbomodules fabric-components

```
codegen
├── java
│   └── com
│       └── MyTurbomodule
│           └── MyTurbomodule.java
├── jni
│   ├── Android.mk
│   ├── CMakeLists.txt
│   ├── MyTurbomodule-generated.cpp
│   ├── MyTurbomodule.h
│   └── react
│       └── renderer
│           └── components
│               └── MyTurbomodule
│                   ├── ComponentDescriptors.h
│                   ├── EventEmitters.cpp
│                   ├── EventEmitters.h
│                   ├── Props.cpp
│                   ├── Props.h
│                   ├── ShadowNodes.cpp
│                   └── ShadowNodes.h
└── schema.json
```

Java can't interoperate seamlessly with C++ as Objective-C++ does. To work properly, **Codegen** creates some bridging between the Java and the C++ world in the `jni` folder, where the Java Native Interfaces are defined.

Notice that both Turbo Native Modules and Fabric Native Components come with two build file descriptors: the `Android.mk` and the `CMakeLists.txt`. These are used by the

Android app to actually build the external modules.

Turbo Native Module

The **Codegen** generates a Java abstract class in the `java` package with the same name as that of the TurboModule. This abstract class has to be implemented by the JNI C++ implementation.

Then, it generates the C++ files in the `jni` folder. They follow the same iOS convention: there is an interface called `MyTurbomodule.h` and an implementation file called `MyTurbomodule-generated.cpp`. The former is an interface that allows React Native to initialize the JSI interface for the TurboModule. The latter is the implementation file which contains the logic to invoke the native method from JS and vice-versa.

Fabric Native Component

The **Codegen** for a Fabric Native Component contains a `MyFabricComponentManagerInterface.java` and a `MyFabricComponentManagerDelegate.java` in the `java` package. They are implemented and used by the native `MyFabricComponentManager` required to properly load the component at runtime (See the guide on how to create a Fabric Native Component for details).

Then, there is a layer of JNI C++ files that are used by Fabric to render the components. The basic element for a Fabric Component is the `ShadowNode`: it represents a node in the React abstract tree. The `ShadowNode` represents a React entity; therefore it could need some props, which are defined in the `Props` files and, sometimes, an `EventEmitter`, defined in the corresponding file.

The **Codegen** also creates a `ComponentDescriptor.h`, which is required to get a proper handle on the Fabric Native Component.

Is this page useful?

Edit this page

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