Enabling in Android Library

> **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.

Once you have defined the JavaScript specs for your native modules as part of the prerequisites, set up the configuration of the Codegen, and follow the Android/Gradle setup, you are now ready to migrate your library to the new architecture. Here are the steps you can follow to accomplish this.

1. Extend or Implement the Code-generated Native Interfaces

The JavaScript spec for your native module or component will be used to generate native interface code for each supported platform (i.e., Android and iOS). These native interface files will be generated **when a React Native application that depends on your library is built**.

While this generated native interface code **will not ship as part of your library**, you do need to make sure your Java/Kotlin code conforms to the protocols provided by these native interface files.

You can invoke the `generateCodegenArtifactsFromSchema` Gradle task to generate your library's native interface code in order to use them **as a reference:**

```
./gradlew generateCodegenArtifactsFromSchema
```

The files that are output can be found inside `build/generated/source/codegen` and **should not be committed**, but you'll need to refer to them to determine what changes you need to make to your native modules in order for them to provide an implementation for each generated interface.

The output of the Codegen for a module called `NativeAwesomeManager` will look like this:

```
app/build/generated/source/codegen
├── java
│   └── com
│       └── example
│           └── samplelibrary
│               └── NativeAwesomeManagerSpec.java
├── jni
│   ├── Android.mk
│   ├── CMakeLists.txt
│   ├── react
│   │   └── renderer
│   │       └── components
│   │           └── samplelibrary
│   │               ├── ComponentDescriptors.h
│   │               ├── EventEmitters.cpp
│   │               ├── EventEmitters.h
│   │               ├── Props.cpp
│   │               ├── Props.h
│   │               ├── ShadowNodes.cpp
│   │               └── ShadowNodes.h
│   ├── samplelibrary-generated.cpp
│   └── samplelibrary.h
└── schema.json
```

Extends the Abstract Class Provided by the Codegen

Update your native module or component to ensure it **extends the abstract class** that has been code-generated from your JavaScript specs (i.e., the `NativeAwesomeManagerSpec.java` file from the previous example).

Following the example set forth in the previous section, your library might import `NativeAwesomeManagerSpec`, implement the relevant native interface and the necessary methods for it:

Java Kotlin

```java
import androidx.annotation.NonNull;

import com.example.samplelibrary.NativeAwesomeManagerSpec;
import com.facebook.react.bridge.Promise;
import com.facebook.react.bridge.ReactApplicationContext;

public class NativeAwesomeManager extends NativeAwesomeManagerSpec {

    public static final String NAME = "NativeAwesomeManager";

    public NativeAwesomeManager(ReactApplicationContext reactContext) {
        super(reactContext);
    }

    @Override
    public void getString(String id, Promise promise) {
        // Implement this method
    }

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

Please note that the **generated abstract class** that you're now extending (`MyAwesomeSpec` in this example) is itself extending `ReactContextBaseJavaModule`. Therefore you should not use access to any of the method/fields you were previously using (e.g., the `ReactApplicationContext` and so on). Moreover, the generated class will now also implement the `TurboModule` interface for you.

Is this page useful? 

Edit this page

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