

C++ Turbo Native Modules

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.

This guide shows you how to implement a Turbo Native Module using C++ only, a way to share the same implementation with any supported platform (Android, iOS, macOS or Windows).

Before continuing with this guide, please read the [Turbo Native Modules](#) section. As a further reference, we prepared an example for the RNTester app ([NativeCxxModuleExample](#)) and a sample run in our community repository ([run/pure-cxx-module](#)).

CAUTION

C++ Turbo Native Modules work with the **New Architecture** enabled. To migrate to the **New Architecture**, follow the [Migration guide](#)

How to Create a C++ Turbo Native Module

To create a C++ Turbo Native Module, you need to:

1. Define the JavaScript specification.
2. Configure Codegen to generate the scaffolding.
3. Register the native module.

4. Write the native code to finish implementing the module.

Setup a Test App for the New Architecture

As first step, create a new application:

```
npx react-native init CxxTurboModulesGuide  
cd CxxTurboModulesGuide  
yarn install
```

On Android enable the New Architecture by modifying the `android/gradle.properties` file:

```
- newArchEnabled=false  
+ newArchEnabled=true
```

On iOS enable the New Architecture when running `pod install` in the `ios` folder:

```
RCT_NEW_ARCH_ENABLED=1 bundle exec pod install
```

Turbo Module Folder Setup

Create a `tm` folder inside the project. It will contain all C++ TurboModules of your application. The final result should look like this:

```
CxxTurboModulesGuide  
├─ android  
├─ ios  
├─ js  
└─ tm
```

1. JavaScript Specification

Create the following spec inside the `tm` folder:

TypeScript Flow

NativeSampleModule.ts

```
import type {TurboModule} from 'react-native/Libraries/TurboModule/RCTExport';
// import type {TurboModule} from 'react-native'; in future versions
import {TurboModuleRegistry} from 'react-native';

export interface Spec extends TurboModule {
  readonly reverseString: (input: string) => string;
}

export default TurboModuleRegistry.getEnforcing<Spec>(
  'NativeSampleModule',
);
```

2. Codegen Configuration

Next, you need to add some configuration for Codegen.

Application

Update your app's `package.json` file with the following entries:

package.json

```
{
  // ...
  "description": "React Native with Cxx Turbo Native Modules",
  "author": "<Your Name> <your_email@your_provider.com> (https://github.com/<your_github_handle>)",
  "license": "MIT",
  "homepage": "https://github.com/<your_github_handle>/#readme",
  // ...
  "codegenConfig": {
    "name": "AppSpecs",
    "type": "all",
    "jsSrcsDir": "tm",
    "android": {
```

```

    "javaPackageName": "com.facebook.fbreact.specs"
  }
}
}

```

It adds necessary properties which we will later re-use in the iOS podspec file and configures **Codegen** to search for specs inside the `tm` folder.

⚠ CAUTION

C++ Turbo Native Modules don't autolink and need to be manually included into the app with the described steps below.

iOS: Create the podspec file

For iOS, you'll need to create a `AppTurboModules.podspec` file in the `tm` folder - which will look like:

AppTurboModules.podspec

```

require "json"

package = JSON.parse(File.read(File.join(__dir__, "../package.json")))

Pod::Spec.new do |s|
  s.name           = "AppTurboModules"
  s.version        = package["version"]
  s.summary        = package["description"]
  s.description    = package["description"]
  s.homepage       = package["homepage"]
  s.license        = package["license"]
  s.platforms      = { :ios => "12.4" }
  s.author         = package["author"]
  s.source         = { :git => package["repository"], :tag => "#{s.version}" }
  s.source_files   = "**/*.h", "**/*.cpp"
  s.pod_target_xcconfig = {
    "CLANG_CXX_LANGUAGE_STANDARD" => "c++17"
  }
  install_modules_dependencies(s)
end

```

You need to add it as a dependency to your application in `ios/Podfile`, e.g., after the `use_react_native!(...)` section:

```
if ENV['RCT_NEW_ARCH_ENABLED'] == '1'  
  pod 'AppTurboModules', :path => "../tm"  
end
```

Android: `build.gradle`, `CMakeLists.txt`, `Onload.cpp`

For Android, you'll need to create a `CMakeLists.txt` file in the `tm` folder - which will look like:

```
cmake_minimum_required(VERSION 3.13)  
set(CMAKE_VERBOSE_MAKEFILE on)  
  
add_compile_options(  
    -fexceptions  
    -frtti  
    -std=c++17)  
  
file(GLOB tm_SRC CONFIGURE_DEPENDS *.cpp)  
add_library(tm STATIC ${tm_SRC})  
  
target_include_directories(tm PUBLIC .)  
target_include_directories(react_codegen_AppSpecs PUBLIC .)  
  
target_link_libraries(tm  
    jsi  
    react_nativemodule_core  
    react_codegen_AppSpecs)
```

It defines the `tm` folder as a source for native code and sets up necessary dependencies.

You need to add it as a dependency to your application in `android/app/build.gradle`, e.g., at the very end of that file:

build.gradle

```

android {
  externalNativeBuild {
    cmake {
      path "src/main/jni/CMakeLists.txt"
    }
  }
}

```

NOTE

Ensure to pick the correct **android/app/build.gradle** file and not android/build.gradle.

3. Module Registration

iOS

To register a C++ Turbo Native Module in your app you will need to update `ios/CxxTurboModulesGuide/AppDelegate.mm` with the following entries:

```

#import "AppDelegate.h"

#import <React/RCTBundleURLProvider.h>
+ #import <React/CoreModulesPlugins.h>
+ #import <ReactCommon/RCTTurboModuleManager.h>
+ #import <NativeSampleModule.h>

+ @interface AppDelegate () <RCTTurboModuleManagerDelegate> {}
+ @end

// ...

- (Class)getModuleClassFromName:(const char *)name
{
  return RCTCoreModulesClassProvider(name);
}

+ #pragma mark RCTTurboModuleManagerDelegate

+ - (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:(const std::string
&)name

```

```

+                                     jsInvoker:
(std::shared_ptr<facebook::react::CallInvoker>)jsInvoker
+ {
+   if (name == "NativeSampleModule") {
+     return std::make_shared<facebook::react::NativeSampleModule>(jsInvoker);
+   }
+   return nullptr;
+ }

```

This will instantiate a `NativeSampleModule` associated with the name `NativeSampleModule` as defined in our JavaScript spec file earlier.

Android

Android apps aren't setup for native code compilation by default.

1.) Create the folder `android/app/src/main/jni`

2.) Copy `CMakeLists.txt` and `Onload.cpp` from `node_modules/react-native/ReactAndroid/cmake-utils/default-app-setup` into the `android/app/src/main/jni` folder.

Update `Onload.cpp` with the following entries:

```

// ...

#include <react/renderer/componentregistry/ComponentDescriptorProviderRegistry.h>
#include <rncli.h>
+ #include <NativeSampleModule.h>

// ...

std::shared_ptr<TurboModule> cxxModuleProvider(
    const std::string &name,
    const std::shared_ptr<CallInvoker> &jsInvoker) {
+ if (name == "NativeSampleModule") {
+   return std::make_shared<facebook::react::NativeSampleModule>(jsInvoker);
+ }
    return nullptr;
}

```

```
// ...
```

Update `CMakeLists.txt` with the following entries, e.g., at the very end of that file:

```
// ...

# This file includes all the necessary to let you build your application with the New
Architecture.
include(${REACT_ANDROID_DIR}/cmake-utils/ReactNative-application.cmake)

+ # App needs to add and link against tm (TurboModules) folder
+ add_subdirectory(${REACT_ANDROID_DIR}/../../tm/tm_build)
+ target_link_libraries(${CMAKE_PROJECT_NAME} tm)
```

This will instantiate a `NativeSampleModule` associated with the name `NativeSampleModule` as defined in our JavaScript spec file earlier.

4. C++ Native Code

For the final step, you'll need to write some native code to connect the JavaScript side to the native platforms. This process requires two main steps:

- Run **Codegen** to see what it generates.
- Write your native code, implementing the generated interfaces.

Run Codegen

! INFO

Follow the [Codegen](#) guide for general information.

On iOS Codegen is run each time you execute in the `ios` folder:

```
RCT_NEW_ARCH_ENABLED=1 bundle exec pod install
```


You can inspect the generated `AppSpecsJSI.h` and `AppSpecsJSI-generated.cpp` files inside the `CxxTurboModulesGuide/ios/build/generated/ios` folder.

Those files are prefixed with `AppSpecs` as this matches the `codegenConfig.name` parameter added earlier to `package.json`.

On Android Codegen is run each time you execute:

```
yarn android
```

You can inspect the generated `AppSpecsJSI.h` and `AppSpecsJSI-generated.cpp` files inside the `CxxTurboModulesGuide/android/app/build/generated/source/codegen/jni` folder.

You only need to re-run codegen if you have changed your JavaScript spec.

The C++ function generated for our JavaScript spec file looks like:

```
virtual jsj::String reverseString(jsj::Runtime &rt, jsj::String input) = 0;
```

You can directly work with the lower level `jsj::` types - but for convenience C++ Turbo Native Modules automatically bridge into `std::` types for you.

Implementation

Now create a `NativeSampleModule.h` file with the following content:

NOTE

Due to current differences in the CMake and CocoaPod setup we need some creativity to include the correct Codegen header on each platform.

```
#pragma once

#if __has_include(<React-Codegen/AppSpecsJSI.h>) // CocoaPod headers on Apple
#include <React-Codegen/AppSpecsJSI.h>
```

```

#elif __has_include("AppSpecsJSI.h") // CMake headers on Android
#include "AppSpecsJSI.h"
#endif
#include <memory>
#include <string>

namespace facebook::react {

class NativeSampleModule : public NativeSampleModuleCxxSpec<NativeSampleModule> {
public:
    NativeSampleModule(std::shared_ptr<CallInvoker> jsInvoker);

    std::string reverseString(jsi::Runtime& rt, std::string input);
};

} // namespace facebook::react

```

In this case you can use any C++ type which bridges to a `jsi::String` - default or custom one. You can't specify an incompatible type such as `bool`, `float` or `std::vector<>` as it does not bridge to `jsi::String` and hence results in a compilation error.

Now add a `NativeSampleModule.cpp` file with an implementation for it:

```

#include "NativeSampleModule.h"

namespace facebook::react {

NativeSampleModule::NativeSampleModule(std::shared_ptr<CallInvoker> jsInvoker)
    : NativeSampleModuleCxxSpec(std::move(jsInvoker)) {}

std::string NativeSampleModule::reverseString(jsi::Runtime& rt, std::string input) {
    return std::string(input.rbegin(), input.rend());
}

} // namespace facebook::react

```

As we have added new C++ files run in the `ios` folder:

```
RCT_NEW_ARCH_ENABLED=1 bundle exec pod install
```

for iOS. In Xcode they appear under the Pods target in the Development Pods \ TurboModules subfolder.

You should now be able to compile your app both on Android and iOS.

```
CxxTurboModulesGuide
├── android
│   └── app
│       ├── src
│       │   ├── main
│       │   │   └── jni
│       │   │       ├── CMakeLists.txt
│       │   │       └── OnLoad.cpp
│       └── build.gradle (updated)
├── ios
│   ├── CxxTurboModulesGuide
│   └── AppDelegate.mm (updated)
├── js
│   └── App.tsx|jsx (updated)
└── tm
    ├── CMakeLists.txt
    ├── NativeSampleModule.h
    ├── NativeSampleModule.cpp
    ├── NativeSampleModule.ts|js
    └── TurboModules.podspec
```

5. Adding the C++ Turbo Native Module to your App

For demo purposes we can update our app's App.tsx|jsx with the following entries:

```
//...
import {
  Colors,
  DebugInstructions,
  Header,
  LearnMoreLinks,
  ReloadInstructions,
} from 'react-native/Libraries/NewAppScreen';
+ import NativeSampleModule from './tm/NativeSampleModule';
//...

<View
```

```

        style={{
          backgroundColor: isDarkMode ? Colors.black : Colors.white,
        }}>
+      <Section title="Cxx TurboModule">
+        NativeSampleModule.reverseString(...) ={' '}
+        {NativeSampleModule.reverseString(
+          'the quick brown fox jumps over the lazy dog'
+        )}
+      </Section>;
      <Section title="Step One">
        Edit <Text style={styles.highlight}>App.tsx</Text> to change this
        screen and then come back to see your edits.
      </Section>
    //...

```

Run the app to see your C++ Turbo Native Module in action!

App TurboModuleProvider [Optional]

You can avoid some code duplication once you added multiple C++ Turbo Native Modules by declaring an AppTurboModuleProvider:

AppTurboModuleProvider.h

```

#pragma once

#include <ReactCommon/TurboModuleBinding.h>
#include <memory>
#include <string>

namespace facebook::react {

class AppTurboModuleProvider {
public:
  std::shared_ptr<TurboModule> getTurboModule(
    const std::string& name,
    std::shared_ptr<CallInvoker> jsInvoker) const;
};

} // namespace facebook::react

```

And implementing it:

AppTurboModuleProvider.cpp

```
#include "AppTurboModuleProvider.h"
#include "NativeSampleModule.h"

namespace facebook::react {

std::shared_ptr<TurboModule> AppTurboModuleProvider::getTurboModule(
    const std::string& name,
    std::shared_ptr<CallInvoker> jsInvoker) const {
    if (name == "NativeSampleModule") {
        return std::make_shared<facebook::react::NativeSampleModule>(jsInvoker);
    }
    // Other C++ Turbo Native Modules for you app
    return nullptr;
}

} // namespace facebook::react
```

And then re-using it in OnLoad.cpp for Android and AppDelegate.mm for iOS, e.g., via:

```
static facebook::react::AppTurboModuleProvider appTurboModuleProvider;
return appTurboModuleProvider.getTurboModule(name, jsInvoker);
```

in the corresponding functions.

Calling OS specific APIs


You can still call OS specific functions in the compilation unit (e.g., NS/CF APIs on Apple or win32/WinRT APIs on Windows) as long as the method signatures only use `std::` or `jsi::` types.

For Apple specific APIs you need to change the extension of your implementation file from `.cpp` to `.mm` to be able to consume NS/CF APIs.

Extending C++ Turbo Native Modules

If you need to support some types that are not supported yet, have a look at [this other guide](#).

Is this page useful?  

 Edit this page

Last updated on **Jun 21, 2023**