Turbo Modules as Legacy 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.

> **INFO**
>
> Creating a backward compatible Turbo Native Module requires the knowledge of how to create a Legacy Native Module. To recall these concepts, have a look at this guide.
>
> TurboModules only works when the New Architecture is properly set up. If you already have a library that you want to migrate to the New Architecture, have a look at the migration guide as well.

Creating a backward compatible TurboModule lets your users continue to leverage your library, independently from the architecture they use. The creation of such a module requires a few steps:

1. Configure the library so that dependencies are prepared set up properly for both the Old and the New Architecture.
2. Update the codebase so that the New Architecture types are not compiled when not available.
3. Uniform the JavaScript API so that your user code won't need changes.

> **INFO**
>
> For the sake of this guide we're going to use the following **terminology**:

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

> **CAUTION**
>
> The TypeScript support for the New Architecture is still in beta.

While the last step is the same for all the platforms, the first two steps are different for iOS and Android.

Configure the Turbo Native Module Dependencies

iOS

The Apple platform installs Turbo Native Modules using CocoaPods as a dependency manager.

If you are already using the `install_module_dependencies` function, then **there is nothing to do**. The function already takes care of installing the proper dependencies when the New Architecture is enabled and avoids them when it is not enabled.

Otherwise, your Turbo Native Module's `podspec` should look like this:

```
require "json"

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

folly_compiler_flags = '-DFOLLY_NO_CONFIG -DFOLLY_MOBILE=1 -DFOLLY_USE_LIBCPP=1 -Wno-comma -Wno-shorten-64-to-32'

Pod::Spec.new do |s|
  # Default fields for a valid podspec
  s.name            = "<TM Name>"
```

```
  s.version          = package["version"]
  s.summary          = package["description"]
  s.description      = package["description"]
  s.homepage         = package["homepage"]
  s.license          = package["license"]
  s.platforms        = { :ios => "11.0" }
  s.author           = package["author"]
  s.source           = { :git => package["repository"], :tag => "#{s.version}" }

  s.source_files     = "ios/**/*.{h,m,mm,swift}"
  # React Native Core dependency
  s.dependency "React-Core"

  # The following lines are required by the New Architecture.
  s.compiler_flags = folly_compiler_flags + " -DRCT_NEW_ARCH_ENABLED=1"
  s.pod_target_xcconfig    = {
      "HEADER_SEARCH_PATHS" => "\"$(PODS_ROOT)/boost\"",
      "CLANG_CXX_LANGUAGE_STANDARD" => "c++17"
  }

  s.dependency "React-Codegen"
  s.dependency "RCT-Folly"
  s.dependency "RCTRequired"
  s.dependency "RCTTypeSafety"
  s.dependency "ReactCommon/turbomodule/core"

end
```

You should install the extra dependencies when the New Architecture is enabled, and avoid installing them when it's not. To achieve this, you can use the `install_modules_dependencies`. Update the `.podspec` file as it follows:

```
require "json"

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

-folly_compiler_flags = '-DFOLLY_NO_CONFIG -DFOLLY_MOBILE=1 -DFOLLY_USE_LIBCPP=1 -Wno-
comma -Wno-shorten-64-to-32'

Pod::Spec.new do |s|
  # Default fields for a valid podspec
  s.name             = "<TM Name>"
  s.version          = package["version"]
  s.summary          = package["description"]
```

```
  s.description     = package["description"]
  s.homepage        = package["homepage"]
  s.license         = package["license"]
  s.platforms       = { :ios => "11.0" }
  s.author          = package["author"]
  s.source          = { :git => package["repository"], :tag => "#{s.version}" }

  s.source_files    = "ios/**/*.{h,m,mm,swift}"
  # React Native Core dependency
+ install_modules_dependencies(s)
- s.dependency "React-Core"
-
- # The following lines are required by the New Architecture.
- s.compiler_flags = folly_compiler_flags + " -DRCT_NEW_ARCH_ENABLED=1"
- s.pod_target_xcconfig    = {
-     "HEADER_SEARCH_PATHS" => "\"$(PODS_ROOT)/boost\"",
-     "CLANG_CXX_LANGUAGE_STANDARD" => "c++17"
- }
-
- s.dependency "React-Codegen"
- s.dependency "RCT-Folly"
- s.dependency "RCTRequired"
- s.dependency "RCTTypeSafety"
- s.dependency "ReactCommon/turbomodule/core"
end
```

Android

To create a module that can work with both architectures, you need to configure Gradle to choose which files need to be compiled depending on the chosen architecture. This can be achieved by using **different source sets** in the Gradle configuration.

> **NOTE**
>
> Please note that this is currently the suggested approach. While it might lead to some code duplication, it will ensure the maximum compatibility with both architectures. You will see how to reduce the duplication in the next section.

To configure the Turbo Native Module so that it picks the proper sourceset, you have to update the `build.gradle` file in the following way:

build.gradle

```
+// Add this function in case you don't have it already
+ def isNewArchitectureEnabled() {
+    return project.hasProperty("newArchEnabled") && project.newArchEnabled == "true"
+}


// ... other parts of the build file

defaultConfig {
        minSdkVersion safeExtGet('minSdkVersion', 21)
        targetSdkVersion safeExtGet('targetSdkVersion', 31)
+        buildConfigField("boolean", "IS_NEW_ARCHITECTURE_ENABLED",
isNewArchitectureEnabled().toString())
+    }
+
+    sourceSets {
+        main {
+            if (isNewArchitectureEnabled()) {
+                java.srcDirs += ['src/newarch']
+            } else {
+                java.srcDirs += ['src/oldarch']
+            }
+        }
    }
}
```

This changes do three main things:

1. The first lines define a function that returns whether the New Architecture is enabled or not.
2. The `buildConfigField` line defines a build configuration boolean field called `IS_NEW_ARCHITECTURE_ENABLED`, and initialize it using the function declared in the first step. This allows you to check at runtime if a user has specified the `newArchEnabled` property or not.
3. The last lines leverage the function declared in step one to decide which source sets we need to build, depending on the chosen architecture.

Update the codebase

iOS

The second step is to instruct Xcode to avoid compiling all the lines using the New Architecture types and files when we are building an app with the Old Architecture.

There are two files to change. The module implementation file, which is usually a `<your-module>.mm` file, and the module header, which is usually a `<your-module>.h` file.

That implementation file is structured as follows:

- Some `#import` statements, among which there is a `<GeneratedSpec>.h` file.
- The module implementation, using the various `RCT_EXPORT_xxx` and `RCT_REMAP_xxx` macros.
- The `getTurboModule:` function, which uses the `<MyModuleSpecJSI>` type, generated by The New Architecture.

The **goal** is to make sure that the `Turbo Native Module` still builds with the Old Architecture. To achieve that, we can wrap the `#import "<GeneratedSpec>.h"` and the `getTurboModule:` function into an `#ifdef RCT_NEW_ARCH_ENABLED` compilation directive, as shown in the following example:

```
#import "<MyModuleHeader>.h"
+ #ifdef RCT_NEW_ARCH_ENABLED
#import "<GeneratedSpec>.h"
+ #endif

// ... rest of your module

+ #ifdef RCT_NEW_ARCH_ENABLED
 - (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:
    (const facebook::react::ObjCTurboModule::InitParams &)params
 {
    return std::make_shared<facebook::react::<MyModuleSpecJSI>>(params);
 }
+ #endif
```

```
@end
```

A similar thing needs to be done for the header file. Add the following lines at the bottom of your module header. You need to first import the header and then, if the New Architecture is enabled, make it conform to the Spec protocol.

```
#import <React/RCTBridgeModule.h>
+ #ifdef RCT_NEW_ARCH_ENABLED
+ #import <YourModuleSpec/YourModuleSpec.h>
+ #endif

@interface YourModule: NSObject <RCTBridgeModule>

@end

+ #ifdef RCT_NEW_ARCH_ENABLED
+ @interface YourModule () <YourModuleSpec>

+ @end
+ #endif
```

This snippets uses the same `RCT_NEW_ARCH_ENABLED` flag used in the previous section. When this flag is not set, Xcode skips the lines within the `#ifdef` during compilation and it does not include them into the compiled binary.

Android

As we can't use conditional compilation blocks on Android, we will define two different source sets. This will allow to create a backward compatible Turbo Native Module with the proper source that is loaded and compiled depending on the used architecture.

Therefore, you have to:

1. Create a Legacy Native Module in the `src/oldarch` path. See this guide to learn how to create a Legacy Native Module.

2. Create a Turbo Native Module in the `src/newarch` path. See this guide to learn how to create a Turbo Native Module

and then instruct Gradle to decide which implementation to pick.

Some files can be shared between a Legacy Native Module and a Turbo Native Module: these should be created or moved into a folder that is loaded by both the architectures. These files are:

- the `<MyModule>Package.java` file used to load the module.
- a `<MyTurboModule>Impl.java` file where we can put the code that both the Legacy Native Module and the Turbo Native Module has to execute.

The final folder structure looks like this:

```
my-module
├── android
│   ├── build.gradle
│   └── src
│       ├── main
│       │   ├── AndroidManifest.xml
│       │   └── java
│       │       └── com
│       │           └── mymodule
│       │               ├── MyModuleImpl.java
│       │               └── MyModulePackage.java
│       ├── newarch
│       │   └── java
│       │       └── com
│       │           └── MyModule.java
│       └── oldarch
│           └── java
│               └── com
│                   └── MyModule.java
├── ios
├── js
└── package.json
```

The code that should go in the `MyModuleImpl.java`, and that can be shared by the Legacy Native Module and the Turbo Native Module is, for example:

Java Kotlin

example of MyModuleImpl.java

```
package com.mymodule;

import androidx.annotation.NonNull;
import com.facebook.react.bridge.Promise;
import java.util.Map;
import java.util.HashMap;

public class MyModuleImpl {

    public static final String NAME = "MyModule";

    public void foo(double a, double b, Promise promise) {
        // implement the logic for foo and then invoke promise.resolve or
        // promise.reject.
    }
}
```

Then, the Legacy Native Module and the Turbo Native Module can be updated with the following steps:

1. Create a private instance of the `MyModuleImpl` class.
2. Initialize the instance in the module constructor.
3. Use the private instance in the modules methods.

For example, for a Legacy Native Module:

Java Kotlin

Native Module using the Impl module

```
public class MyModule extends ReactContextBaseJavaModule {

    // declare an instance of the implementation
    private MyModuleImpl implementation;
```

```
    MyModule(ReactApplicationContext context) {
        super(context);
        // initialize the implementation of the module
        implementation = MyModuleImpl();
    }

    @Override
    public String getName() {
        // NAME is a static variable, so we can access it using the class name.
        return MyModuleImpl.NAME;
    }

    @ReactMethod
    public void foo(int a, int b, Promise promise) {
        // Use the implementation instance to execute the function.
        implementation.foo(a, b, promise);
    }
}
```

And, for a Turbo Native Module:

Java Kotlin

TurboModule using the Impl module

```
public class MyModule extends MyModuleSpec {
    // declare an instance of the implementation
    private MyModuleImpl implementation;

    MyModule(ReactApplicationContext context) {
        super(context);
        // initialize the implementation of the module
        implementation = MyModuleImpl();
    }

    @Override
    @NonNull
    public String getName() {
        // NAME is a static variable, so we can access it using the class name.
        return MyModuleImpl.NAME;
    }

    @Override
    public void foo(double a, double b, Promise promise) {
```

```
        // Use the implementation instance to execute the function.
        implementation.foo(a, b, promise);
    }
}
```

For a step-by-step example on how to achieve this, have a look at this repo.

Unify the JavaScript specs

> **CAUTION**
>
> The TypeScript support for the New Architecture is still in beta.

The last step makes sure that the JavaScript behaves transparently to chosen architecture.

For a Turbo Native Module, the source of truth is the `Native<MyModule>.js` (or `.ts`) spec file. The app accesses the spec file like this:

```
import MyModule from 'your-module/src/index';
```

Since `TurboModuleRegistry.get` taps into the old Native Modules API under the hood, we need to re-export our module, to avoid registering it multiple times.

Flow TypeScript

```
// @flow
export default require('./Native<MyModule>').default;
```

Is this page useful? 👍 👎

Edit this page

Last updated on ***Sep 3, 2023***