Fabric Components as Legacy Native Components

> **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 Fabric Native Component requires the knowledge of how to create a Legacy Native Component. To recall these concepts, have a look at this guide.
>
> Fabric Native Components only work 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 Fabric Native Component lets your users continue to leverage your library independently from the architecture they use. The creation of such a component requires a few steps:

1. Configure the library so that dependencies are prepared to 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 Components** - To refer to Components 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 Native Renderer, Fabric. For brevity you might find them referred as **Fabric Components**.

> **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 Fabric Native Component Dependencies

iOS

The Apple platform installs Fabric Native Components 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 avoiding them when it is not enabled.

Otherwise, your Fabric Native Component'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            = "<FC 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\"",
      "OTHER_CPLUSPLUSFLAGS" => "-DFOLLY_NO_CONFIG -DFOLLY_MOBILE=1 -
DFOLLY_USE_LIBCPP=1",
      "CLANG_CXX_LANGUAGE_STANDARD" => "c++17"
  }

  s.dependency "React-RCTFabric"
  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            = "<FC 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\"",
-      "OTHER_CPLUSPLUSFLAGS" => "-DFOLLY_NO_CONFIG -DFOLLY_MOBILE=1 -
DFOLLY_USE_LIBCPP=1",
-      "CLANG_CXX_LANGUAGE_STANDARD" => "c++17"
-  }
-
-  s.dependency "React-RCTFabric"
-  s.dependency "React-Codegen"
-  s.dependency "RCT-Folly"
-  s.dependency "RCTRequired"
-  s.dependency "RCTTypeSafety"
-  s.dependency "ReactCommon/turbomodule/core"
end
```

Android

To create a Native Component 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 maximum compatibility with both architectures. You will

> see how to reduce the duplication in the next section.

To configure the Fabric Native Component 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']
+            }
+        }
    }
}
```

These 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.

A Fabric Native Component requires a header file and an implementation file to add the actual `View` to the module.

For example, the `RNMyComponentView.h` header file could look like this:

RNMyComponentView.h

```
#import <React/RCTViewComponentView.h>
#import <UIKit/UIKit.h>

#ifndef NativeComponentExampleComponentView_h
#define NativeComponentExampleComponentView_h

NS_ASSUME_NONNULL_BEGIN

@interface RNMyComponentView : RCTViewComponentView
@end

NS_ASSUME_NONNULL_END

#endif /* NativeComponentExampleComponentView_h */
```

The implementation `RNMyComponentView.mm` file, instead, could look like this:

RNMyComponentView.mm

```
#import "RNMyComponentView.h"
```

```
// <react/renderer imports>

#import "RCTFabricComponentsPlugins.h"

using namespace facebook::react;

@interface RNMyComponentView () <RCTMyComponentViewViewProtocol>

@end

@implementation RNMyComponentView {
    UIView * _view;
}

+ (ComponentDescriptorProvider)componentDescriptorProvider
{
    // ... return the descriptor ...
}

- (instancetype)initWithFrame:(CGRect)frame
{
  // ... initialize the object ...
}

- (void)updateProps:(Props::Shared const &)props oldProps:(Props::Shared const &)oldProps
{
  // ... set up the props ...

  [super updateProps:props oldProps:oldProps];
}

Class<RCTComponentViewProtocol> MyComponentViewCls(void)
{
  return RNMyComponentView.class;
}

@end
```

To make sure that Xcode skips these files, we can wrap **both** of them in some `#ifdef RCT_NEW_ARCH_ENABLED` compilation pragma. For example, the header file could change as follows:

```
+ #ifdef RCT_NEW_ARCH_ENABLED
#import <React/RCTViewComponentView.h>
```

```
#import <UIKit/UIKit.h>

// ... rest of the header file ...

#endif /* NativeComponentExampleComponentView_h */
+ #endif
```

The same two lines should be added in the implementation file, as first and last lines.

The above snippet 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. The compiled binary will have a the `RNMyComponentView.o` object but it will be an empty object.

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 TurboModule with the proper source that is loaded and compiled depending on the used architecture.

Therefore, you have to:

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

and then instruct Gradle to decide which implementation to pick.

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

- the `<MyComponentView>.java` that instantiate and configure the Android View for both the components.
- the `<MyComponentView>ManagerImpl.java` file where which contains the logic of the ViewManager that can be shared between the Legacy and the Fabric Component.
- the `<MyComponentView>Package.java` file used to load the component.

The final folder structure looks like this:

```
my-component
├── android
│   ├── build.gradle
│   └── src
│       ├── main
│       │   ├── AndroidManifest.xml
│       │   └── java
│       │       └── com
│       │           └── mycomponent
│       │               ├── MyComponentView.java
│       │               ├── MyComponentViewManagerImpl.java
│       │               └── MyComponentViewPackage.java
│       ├── newarch
│       │   └── java
│       │       └── com
│       │           └── MyComponentViewManager.java
│       └── oldarch
│           └── java
│               └── com
│                   └── MyComponentViewManager.java
├── ios
├── js
└── package.json
```

The code that should go in the `MyComponentViewManagerImpl.java` and that can be shared between the Native Component and the Fabric Native Component is, for example:

Java Kotlin

example of MyComponentViewManager.java

```java
package com.mycomponent;

import androidx.annotation.Nullable;
import com.facebook.react.uimanager.ThemedReactContext;

public class MyComponentViewManagerImpl {

    public static final String NAME = "MyComponent";
```

```
    public static MyComponentView createViewInstance(ThemedReactContext context) {
        return new MyComponentView(context);
    }

    public static void setFoo(MyComponentView view, String param) {
        // implement the logic of the foo function using the view and the param passed.
    }
}
```

Then, the Native Component and the Fabric Native Component can be updated using the function declared in the shared manager.

For example, for a Native Component:

Java Kotlin

Native Component using the ViewManagerImpl

```
public class MyComponentViewManager extends SimpleViewManager<MyComponentView> {

    ReactApplicationContext mCallerContext;

    public MyComponentViewManager(ReactApplicationContext reactContext) {
        mCallerContext = reactContext;
    }

    @Override
    public String getName() {
        // static NAME property from the shared implementation
        return MyComponentViewManagerImpl.NAME;
    }

    @Override
    public MyComponentView createViewInstance(ThemedReactContext context) {
        // static createViewInstance function from the shared implementation
        return MyComponentViewManagerImpl.createViewInstance(context);
    }

    @ReactProp(name = "foo")
    public void setFoo(MyComponentView view, String param) {
        // static custom function from the shared implementation
        MyComponentViewManagerImpl.setFoo(view, param);
    }
```

```
}
```

And, for a Fabric Native Component:

Java Kotlin

Fabric Component using the ViewManagerImpl

```
// Use the static NAME property from the shared implementation
@ReactModule(name = MyComponentViewManagerImpl.NAME)
public class MyComponentViewManager extends SimpleViewManager<MyComponentView>
        implements MyComponentViewManagerInterface<MyComponentView> {

    private final ViewManagerDelegate<MyComponentView> mDelegate;

    public MyComponentViewManager(ReactApplicationContext context) {
        mDelegate = new MyComponentViewManagerDelegate<>(this);
    }

    @Nullable
    @Override
    protected ViewManagerDelegate<MyComponentView> getDelegate() {
        return mDelegate;
    }

    @NonNull
    @Override
    public String getName() {
        // static NAME property from the shared implementation
        return MyComponentViewManagerImpl.NAME;
    }

    @NonNull
    @Override
    protected MyComponentView createViewInstance(@NonNull ThemedReactContext context) {
        // static createViewInstance function from the shared implementation
        return MyComponentViewManagerImpl.createViewInstance(context);
    }

    @Override
    @ReactProp(name = "foo")
    public void setFoo(MyComponentView view, @Nullable String param) {
        // static custom function from the shared implementation
```

```
            MyComponentViewManagerImpl.setFoo(view, param);
        }
    }
```

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 Fabric Native Component, the source of truth is the `<YourModule>NativeComponent.js` (or `.ts`) spec file. The app accesses the spec file like this:

```
import MyComponent from 'your-component/src/index';
```

Since `codegenNativeComponent` is calling the `requireNativeComponent` under the hood, we need to re-export our component, to avoid registering it multiple times.

Flow TypeScript

```
// @flow
export default require('./MyComponentNativeComponent').default;
```

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