

Fabric 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.

A Fabric Native Component is a Native Component rendered on the screen using the [Fabric Renderer](#). Using Fabric Native Components instead of Legacy Native Components allows us to reap all the [benefits](#) of the **New Architecture**:

- Strongly typed interfaces that are consistent across platforms.
- The ability to write your code in C++, either exclusively or integrated with another native platform language, hence reducing the need to duplicate implementations across platforms.
- The use of JSI, a JavaScript interface for native code, which allows for more efficient communication between native and JavaScript code than the bridge.

A Fabric Native Component is created starting from a **JavaScript specification**. Then [Codegen](#) creates some C++ scaffolding code to connect the component-specific logic (for example, accessing some native platform capability) to the rest of the React Native infrastructure. The C++ code is the same for all the platforms. Once the component is properly connected with the scaffolding code, it is ready to be imported and used by an app.

The following section guides you through the creation of a Fabric Native Component, step-by-step, targeting the latest version of React Native.

CAUTION

Fabric Native Components only works with the **New Architecture** enabled. To migrate to the **New Architecture**, follow the [Migration guide](#)

How to Create a Fabric Native Components

To create a Fabric Native Component, you have to follow these steps:

1. Define a set of JavaScript specifications.
2. Configure the component so that **Codegen** can create the shared code and it can be added as a dependency for an app.
3. Write the required native code.

Once these steps are done, the component is ready to be consumed by an app. The guide shows how to add it to an app by leveraging *autolinking*, and how to reference it from the JavaScript code.

1. Folder Setup

In order to keep the component decoupled from the app, it's a good idea to define the module separately from the app and then add it as a dependency to your app later. This is also what you'll do for writing Fabric Native Component that can be released as open-source libraries later.

For this guide, you are going to create a Fabric Native Component that centers some text on the screen.

Create a new folder at the same level of the app and call it `RTNCenteredText`.

In this folder, create three subfolders: `js`, `ios`, and `android`.

The final result should look like this:

```
├── MyApp
└── RTNCenteredText
```



2. JavaScript Specification

The **New Architecture** requires interfaces specified in a typed dialect of JavaScript (either Flow or TypeScript). **Codegen** uses these specifications to generate code in strongly-typed languages, including C++, Objective-C++, and Java.

There are two requirements the file containing this specification must meet:

1. The file **must** be named `<MODULE_NAME>NativeComponent`, with a `.js` or `.jsx` extension when using Flow, or a `.ts`, or `.tsx` extension when using TypeScript. **Codegen** only looks for files matching this pattern.
2. The file must export a `HostComponent` object.

Below are specifications of the `RTNCenteredText` component in both Flow and TypeScript. Create a `RTNCenteredTextNativeComponent` file with the proper extension in the `js` folder.

TypeScript Flow

```

import type {ViewProps} from 'ViewPropTypes';
import type {HostComponent} from 'react-native';
import codegenNativeComponent from 'react-native/Libraries/Utilities/codegenNativeComponent';

export interface NativeProps extends ViewProps {
  text?: string;
  // add other props here
}

export default codegenNativeComponent<NativeProps>(
  'RTNCenteredText',
) as HostComponent<NativeProps>;

```

At the beginning of the spec files, there are the imports. The most important imports, required by every Fabric Native Component are:

- The `HostComponent`: type the exported component needs to conform to.
- The `codegenNativeComponent` function: responsible to actually register the component in the JavaScript runtime.

The second section of the files contains the **props** of the component. Props (short for "properties") are component-specific information that let you customize React components. In this case, you want to control the `text` property of the component.

Finally, the spec file exports the returned value of the `codegenNativeComponent` generic function, invoked passing the name of the component.

CAUTION

The JavaScript files imports types from libraries, without setting up a proper node module and installing its dependencies. The outcome of this is that the IDE may have troubles resolving the import statements and it can output errors and warnings. These will disappear as soon as the Fabric Native Component is added as a dependency of a React Native app.

3. Component Configuration

Next, you need to add some configuration for Codegen and auto-linking.

Some of these configuration files are shared between iOS and Android, while the others are platform-specific.

Shared

The shared configuration is a `package.json` file that will be used by yarn when installing your module. Create the `package.json` file in the root of the `RTNCenteredText` directory.

`package.json`

```
{  
  "name": "rtn-centered-text",
```

```

"version": "0.0.1",
"description": "Showcase a Fabric Native Component with a centered text",
"react-native": "js/index",
"source": "js/index",
"files": [
  "js",
  "android",
  "ios",
  "rtn-centered-text.podspec",
  "!android/build",
  "!ios/build",
  "!**/__tests__",
  "!**/__fixtures__",
  "!**/__mocks__"
],
"keywords": ["react-native", "ios", "android"],
"repository": "https://github.com/<your_github_handle>/rtn-centered-text",
"author": "<Your Name> <your_email@your_provider.com>
(https://github.com/<your_github_handle>)",
"license": "MIT",
"bugs": {
  "url": "https://github.com/<your_github_handle>/rtn-centered-text/issues"
},
"homepage": "https://github.com/<your_github_handle>/rtn-centered-text#readme",
"devDependencies": {},
"peerDependencies": {
  "react": "*",
  "react-native": "*"
},
"codegenConfig": {
  "name": "RTNCenteredTextSpecs",
  "type": "components",
  "jsSrcsDir": "js"
}
}

```

The upper part of the file contains some descriptive information like the name of the component, its version and its source files. Make sure to update the various placeholders which are wrapped in `<>`: replace all the occurrences of the `<your_github_handle>`, `<Your Name>`, and `<your_email@your_provider.com>` tokens.

Then there are the dependencies for this package. For this guide, you need `react` and `react-native`.

Finally, the **Codegen** configuration is specified by the `codegenConfig` field. It contains an array of libraries, each of which is defined by three other fields:

- `name`: The name of the library. By convention, you should add the `Spec` suffix.
- `type`: The type of module contained by this package. In this case, it is a Fabric Native Component, thus the value to use is `components`.
- `jsSrcsDir`: the relative path to access the `js` specification that is parsed by **Codegen**.

iOS: Create the `.podspec` file

For iOS, you'll need to create a `rtn-centered-text.podspec` file which will define the module as a dependency for your app. It will stay in the root of `RTNCenteredText`, alongside the `ios` folder.

The file will look like this:

`rtn-centered-text.podspec`

```
require "json"

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

Pod::Spec.new do |s|
  s.name           = "rtn-centered-text"
  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}"

  install_modules_dependencies(s)
end
```

The `.podspec` file has to be a sibling of the `package.json` file, and its name is the one we set in the `package.json`'s `name` property: `rtn-centered-text`.

The first part of the file prepares some variables that we use throughout the file. Then, there is a section that contains some information used to configure the pod, like its name, version, and description.

All the requirements for the New Architecture have been encapsulated in the `install_modules_dependencies`. It takes care of installing the proper dependencies based on which architecture is currently enabled. It also automatically installs the `React-Core` dependency in the old architecture.

Android: `build.gradle` and the `ReactPackage` class

To prepare Android to run **Codegen** you have to:

1. Update the `build.gradle` file.
2. Create a Java/Kotlin class that implements the `ReactPackage` interface.

At the end of these steps, the `android` folder should look like this:

```
android
├── build.gradle
├── src
│   └── main
│       └── java
│           └── com
│               └── rtncenteredtext
│                   └── CenteredTextPackage.java
```

The `build.gradle` file

First, create a `build.gradle` file in the `android` folder, with the following contents:

Java Kotlin

`build.gradle`

```

buildscript {
  ext.safeExtGet = {prop, fallback ->
    rootProject.ext.has(prop) ? rootProject.ext.get(prop) : fallback
  }
  repositories {
    google()
    gradlePluginPortal()
  }
  dependencies {
    classpath("com.android.tools.build:gradle:7.3.1")
  }
}

apply plugin: 'com.android.library'
apply plugin: 'com.facebook.react'

android {
  compileSdkVersion safeExtGet('compileSdkVersion', 33)
  namespace "com.rtncenteredtext"

  defaultConfig {
    minSdkVersion safeExtGet('minSdkVersion', 21)
    targetSdkVersion safeExtGet('targetSdkVersion', 33)
    buildConfigField("boolean", "IS_NEW_ARCHITECTURE_ENABLED", "true")
  }
}

repositories {
  mavenCentral()
  google()
}

dependencies {
  implementation 'com.facebook.react:react-native'
}

```

The ReactPackage class

Then, you need a class that implements the `ReactPackage` interface. To run the **Codegen** process, you don't have to completely implement the `Package` class: an empty implementation is enough for the app to pick up the module as a proper React Native dependency and to try and generate the scaffolding code.

Create an `android/src/main/java/com/rtncenteredtext` folder and, inside that folder, create a `CenteredTextPackage.java` file.

Java Kotlin

CenteredTextPackage.java

```
package com.rtncenteredtext;

import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.Collections;
import java.util.List;

public class CenteredTextPackage implements ReactPackage {

    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext reactContext) {
        return Collections.emptyList();
    }

    @Override
    public List<NativeModule> createNativeModules(ReactApplicationContext reactContext) {
        return Collections.emptyList();
    }

}
```

The `ReactPackage` interface is used by React Native to understand what native classes the app has to use for the `ViewManager` and `Native Modules` exported by the library.

4. Native Code

The last step requires you to write some native code to connect the JavaScript side of the Component to what is offered by the platforms. This process requires two main steps:

1. Run **Codegen** to see what would be generated.

2. Write the native code that will make it work.

When developing a React Native app that uses a Fabric Native Component, it is the responsibility of the app to actually generate the code using **Codegen**. However, when developing a Fabric Component as a library, it needs to reference the generated code, and it is useful to see what the app will generate.

As the first step for both iOS and Android, this guide shows how to execute manually the scripts used by **Codegen** to generate the required code. Further information on **Codegen** can be found [here](#).

CAUTION

The code generated by **Codegen** in this step should not be committed to the versioning system. React Native apps are able to generate the code when the app is built. This allows an app to ensure that all libraries have code generated for the correct version of React Native.

iOS

Generate the code - iOS

To run Codegen for the iOS platform, open a terminal and run the following command:

```
cd MyApp
yarn add ../RTNCenteredText
cd ..
node MyApp/node_modules/react-native/scripts/generate-codegen-artifacts.js \
  --path MyApp/ \
  --outputPath RTNCenteredText/generated/
```

This script first adds the `RTNCenteredText` module to the app with `yarn add`. Then, it invokes **Codegen** via the `generate-codegen-artifacts.js` script.

The `--path` option specifies the path to the app, while the `--outputPath` option tells the script where to output the generated code.

The output of this process is the following folder structure:

```

generated
├── build
│   └── generated
│       └── ios
│           ├── FBReactNativeSpec
│           │   ├── FBReactNativeSpec-generated.mm
│           │   └── FBReactNativeSpec.h
│           ├── RCTThirdPartyFabricComponentsProvider.h
│           ├── RCTThirdPartyFabricComponentsProvider.mm
│           └── react
│               └── renderer
│                   └── components
│                       ├── RTNCenteredTextSpecs
│                       │   ├── 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 relevant path for the component is

`generated/build/generated/ios/react/renderer/components/RTNCenteredTextSpecs`. This folder contains all the generated code required by your Component.

See the [Codegen](#) section for further details on the generated files.

NOTE

When generating the scaffolding code using **Codegen**, iOS does not clean the `build` folder automatically. If you changed the Spec name, for example, and then run **Codegen** again, the old files will be retained. If that happens, remember to remove the `build` folder before running the **Codegen** again.

```
cd MyApp/ios
rm -rf build
```

Write the Native iOS Code

Now that the scaffolding code has been generated, it's time to write the Native code for your Fabric Component. You need to create three files in the `RTNCenteredText/ios` folder:

1. The `RTNCenteredTextManager.mm`, an Objective-C++ file that declares what the Component exports.
2. The `RTNCenteredText.h`, a header file for the actual view.
3. The `RTNCenteredText.mm`, the implementation of the view.

RTNCenteredTextManager.mm

RTNCenteredTextManager.mm

```
#import <React/RCTLog.h>
#import <React/RCTUIManager.h>
#import <React/RCTViewManager.h>

@interface RTNCenteredTextManager : RCTViewManager
@end

@implementation RTNCenteredTextManager

RCT_EXPORT_MODULE(RTNCenteredText)

RCT_EXPORT_VIEW_PROPERTY(text, NSString)

@end
```

This file is the manager for the Fabric Native Component. React Native runtime uses manager objects to register the modules, properties and methods to make them available to the JavaScript side.

The most important call is to the `RCT_EXPORT_MODULE`, which is required to export the module so that Fabric can retrieve and instantiate it.

Then, you have to expose the `text` property for the Fabric Native Component. This is done with the `RCT_EXPORT_VIEW_PROPERTY` macro, specifying a name and a type.

! INFO

There are other macros that can be used to export custom properties, emitters, and other constructs. You can view the code that specifies them [here](#).

RTNCenteredText.h

RTNCenteredText.h

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

NS_ASSUME_NONNULL_BEGIN

@interface RTNCenteredText : RCTViewComponentView

@end

NS_ASSUME_NONNULL_END
```

This file defines the interface for the `RTNCenteredText` view. Here, you can add any native method you may want to invoke on the view. For this guide, you don't need anything, therefore the interface is empty.

RTNCenteredText.mm

RTNCenteredText.mm

```

#import "RTNCenteredText.h"

#import <react/renderer/components/RTNCenteredTextSpecs/ComponentDescriptors.h>
#import <react/renderer/components/RTNCenteredTextSpecs/EventEmitters.h>
#import <react/renderer/components/RTNCenteredTextSpecs/Props.h>
#import <react/renderer/components/RTNCenteredTextSpecs/RCTComponentViewHelpers.h>

#import "RCTFabricComponentsPlugins.h"

using namespace facebook::react;

@interface RTNCenteredText () <RCTRTNCenteredTextViewProtocol>
@end

@implementation RTNCenteredText {
    UIView *_view;
    UILabel *_label;
}

+ (ComponentDescriptorProvider)componentDescriptorProvider
{
    return concreteComponentDescriptorProvider<RTNCenteredTextComponentDescriptor>();
}

- (instancetype)initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame]) {
        static const auto defaultProps = std::make_shared<const RTNCenteredTextProps>();
        _props = defaultProps;

        _view = [[UIView alloc] init];
        _view.backgroundColor = [UIColor redColor];

        _label = [[UILabel alloc] init];
        _label.text = @"Initial value";
        [_view addSubview:_label];

        _label.translatesAutoresizingMaskIntoConstraints = false;
        [NSLayoutConstraint activateConstraints:@[
            [_label.leadingAnchor constraintEqualToAnchor:_view.leadingAnchor],
            [_label.topAnchor constraintEqualToAnchor:_view.topAnchor],
            [_label.trailingAnchor constraintEqualToAnchor:_view.trailingAnchor],
            [_label.bottomAnchor constraintEqualToAnchor:_view.bottomAnchor],
        ]];

        _label.textAlignment = NSTextAlignmentCenter;
    }
}

```

```

        self.contentView = _view;
    }

    return self;
}

- (void)updateProps:(Props::Shared const &)props oldProps:(Props::Shared const &)oldProps
{
    const auto &oldViewProps = *std::static_pointer_cast<RTNCenteredTextProps const>
(_props);
    const auto &newViewProps = *std::static_pointer_cast<RTNCenteredTextProps const>
(props);

    if (oldViewProps.text != newViewProps.text) {
        _label.text = [[NSString alloc] initWithCString:newViewProps.text.c_str()
encoding:NSUTF8StringEncoding];
    }

    [super updateProps:props oldProps:oldProps];
}

@end

Class<RCTComponentViewProtocol> RTNCenteredTextCls(void)
{
    return RTNCenteredText.class;
}

```

This file contains the actual implementation of the view.

It starts with some imports, which require you to read the files generated by **Codegen**.

The component has to conform to a specific protocol generated by **Codegen**, in this case, `RCTRTNCenteredTextViewProtocol`.

Then, the file defines a static (ComponentDescriptorProvider) `componentDescriptorProvider` method which Fabric uses to retrieve the descriptor provider to instantiate the object.

Then, there is the constructor of the view: the `init` method. In this method, it is important to create a `defaultProps` struct using the `RTNCenteredTextProps` type from **Codegen**. You need to assign it to the private `_props` to initialize the Fabric Native

Component correctly. The remaining part of the initializer is standard Objective-C code to create views and layout them with `AutoLayout`.

The last two pieces are the `updateProps` method and the `RTNCenteredTextCls` method.

The `updateProps` method is invoked by Fabric every time a prop changes in JavaScript. The props passed as parameters are downcasted to the proper `RTNCenteredTextProps` type, and then they are used to update the native code if needed. Notice that the superclass method `[super updateProps]` must be invoked as the last statement of this method; otherwise the `props` and `oldProps` struct will have the same values, and you'll not be able to use them to make decisions and to update the component.

Finally, the `RTNCenteredTextCls` is another static method used to retrieve the correct instance of the class at runtime.

CAUTION

Differently from Legacy Native Components, Fabric requires to manually implement the `updateProps` method. It's not enough to export properties with the `RCT_EXPORT_XXX` and `RCT_REMAP_XXX` macros.

Android

Android follows some similar steps to iOS. You have to generate the code, and then you have to write some native code to make it works.

Generate the Code - Android

To generate the code, you need to manually invoke **Codegen**. This is done similarly to what you need to do for iOS: first, you need to add the package to the app and then you need to invoke a script.

Running Codegen for Android

```
cd MyApp
yarn add ../RTNCenteredText
```



```
cd android
./gradlew generateCodegenArtifactsFromSchema
```

This script first adds the package to the app, in the same way iOS does. Then, after moving to the `android` folder, it invokes a Gradle task to generate the scaffolding code.

NOTE

To run **Codegen**, you need to enable the **New Architecture** in the Android app. This can be done by opening the `gradle.properties` files and by switching the `newArchEnabled` property from `false` to `true`.

The generated code is stored in the `MyApp/node_modules/rtn-centered-text/android/build/generated/source/codegen` folder and it has this structure:

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

You can see that the content of the `codegen/jni/react/renderer/components/RTNCenteredTextSpecs` looks similar to the files created by the iOS counterpart. The `Android.mk` and `CMakeList.txt` files configure the Fabric Native Component in the app, while the `RTNCenteredTextManagerDelegate.java` and `RTNCenteredTextManagerInterface.java` files are meant use in your manager.

See the [Codegen](#) section for further details on the generated files.

Write the Native Android Code

The native code for the Android side of a Fabric Native Components requires three pieces:

1. A `CenteredText.java` that represents the actual view.
2. A `CenteredTextManager.java` to instantiate the view.
3. Finally, you have to fill the implementation of the `CenteredTextPackage.java` created in the previous step.

The final structure within the Android library should be like this.

```
android
├── build.gradle
├── src
│   └── main
│       └── java
│           └── com
│               └── rtncenteredtext
│                   ├── CenteredText.java
│                   ├── CenteredTextManager.java
│                   └── CenteredTextPackage.java
```

CenteredText.java

Java Kotlin

CenteredText.java

```
package com.rtncenteredtext;

import androidx.annotation.Nullable;
import android.content.Context;
import android.util.AttributeSet;
import android.graphics.Color;

import android.widget.TextView;
import android.view.Gravity;

public class CenteredText extends TextView {

    public CenteredText(Context context) {
        super(context);
        this.configureComponent();
    }

    public CenteredText(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
        this.configureComponent();
    }

    public CenteredText(Context context, @Nullable AttributeSet attrs, int defStyleAttr)
    {
        super(context, attrs, defStyleAttr);
        this.configureComponent();
    }

    private void configureComponent() {
        this.setBackgroundColor(Color.RED);
        this.setGravity(Gravity.CENTER_HORIZONTAL);
    }
}
```

This class represents the actual view Android is going to represent on screen. It inherits from `TextView` and it configures the basic aspects of itself using a private `configureComponent()` function.

CenteredTextManager.java

Java Kotlin

CenteredTextManager.java

```
package com.rtncenteredtext;

import androidx.annotation.NonNull;
import androidx.annotation.Nullable;

import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.module.annotations.ReactModule;
import com.facebook.react.uimanager.SimpleViewManager;
import com.facebook.react.uimanager.ThemedReactContext;
import com.facebook.react.uimanager.ViewManagerDelegate;
import com.facebook.react.uimanager.annotations.ReactProp;
import com.facebook.react.viewmanagers.RTNCenteredTextManagerInterface;
import com.facebook.react.viewmanagers.RTNCenteredTextManagerDelegate;

@ReactModule(name = CenteredTextManager.NAME)
public class CenteredTextManager extends SimpleViewManager<CenteredText>
    implements RTNCenteredTextManagerInterface<CenteredText> {

    private final ViewManagerDelegate<CenteredText> mDelegate;

    static final String NAME = "RTNCenteredText";

    public CenteredTextManager(ReactApplicationContext context) {
        mDelegate = new RTNCenteredTextManagerDelegate<>(this);
    }

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

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

    @NonNull
    @Override
    protected CenteredText createViewInstance(@NonNull ThemedReactContext context) {
        return new CenteredText(context);
    }
}
```

```

@Override
@ReactProp(name = "text")
public void setText(CenteredText view, @Nullable String text) {
    view.setText(text);
}
}

```

The `CenteredTextManager` is a class used by React Native to instantiate the native component. It is the class that implements the interfaces generated by **Codegen** (see the `RTNCenteredTextManagerInterface` interface in the `implements` clause) and it uses the `RTNCenteredTextManagerDelegate` class.

It is also responsible for exporting all the constructs required by React Native: the class itself is annotated with `@ReactModule` and the `setText` method is annotated with `@ReactProp`.

CenteredTextPackage.java

Finally, open the `CenteredTextPackage.java` file in the `android/src/main/java/com/rtncenteredtext` folder and update it with the following lines

Java Kotlin

CenteredTextPackage.java update

```

package com.rtncenteredtext;

import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.Collections;
import java.util.List;

public class CenteredTextPackage implements ReactPackage {

    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext reactContext) {
        + return Collections.singletonList(new CenteredTextManager(reactContext));
    }
}

```

```
}

@Override
public List<NativeModule> createNativeModules(ReactApplicationContext reactContext) {
    return Collections.emptyList();
}

}
```

The added lines instantiate a new `RTNCenteredTextManager` object so that the React Native runtime can use it to render our Fabric Native Component.

5. Adding the Fabric Native Component To Your App

This is the last step to finally see your Fabric Native Component running on your app.

Shared

First of all, you need to add the NPM package which contains the Component to the app. This can be done with the following command:

```
cd MyApp
yarn add ../RTNCenteredText
```

This command adds the `RTNCenteredText` Component to the `node_modules` of your app.

If the package was previously added to your app, you will need to update it:

```
cd MyApp
yarn upgrade rtn-centered-text
```

iOS

Then, you need to install the new dependencies in your iOS project. To do so, you need to run these commands:

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

This command installs the iOS dependencies for the project. The `RCT_NEW_ARCH_ENABLED=1` flag instructs **CocoaPods** that it has to execute some additional operations to run **Codegen**.

NOTE

You may have to run `bundle install` once before you can use `RCT_NEW_ARCH_ENABLED=1 bundle exec pod install`. You won't need to run `bundle install` anymore, unless you need to change the ruby dependencies.

Android

Android configuration requires to enable the **New Architecture**.

1. Open the `android/gradle.properties` file
2. Scroll down to the end of the file and switch the `newArchEnabled` property from `false` to `true`.

JavaScript

Finally, you can use the component in your JavaScript application.

TypeScript Flow

App.tsx


```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 *
 * @format
 */
import React from 'react';
```

```
import RTNCenteredText from 'rtn-centered-text/js/RTNCenteredTextNativeComponent';

const App: () => JSX.Element = () => {
  return (
    <RTNCenteredText
      text="Hello World!"
      style={{width: '100%', height: 30}}
    />
  );
};
export default App;
```

Now, you can run the React Native app and see your Component on the screen.

Is this page useful?  

 Edit this page

Last updated on **Aug 24, 2023**