Publishing to Google Play Store

Android requires that all apps be digitally signed with a certificate before they can be installed. In order to distribute your Android application via Google Play store it needs to be signed with a release key that then needs to be used for all future updates. Since 2017 it is possible for Google Play to manage signing releases automatically thanks to App Signing by Google Play functionality. However, before your application binary is uploaded to Google Play it needs to be signed with an upload key. The Signing Your Applications page on Android Developers documentation describes the topic in detail. This guide covers the process in brief, as well as lists the steps required to package the JavaScript bundle.

> **INFO**
>
> If you are using Expo, read the Expo guide for Deploying to App Stores to build and submit your app for the Google Play Store. This guide works with any React Native app to automate the deployment process.

Generating an upload key

You can generate a private signing key using `keytool`.

Windows

On Windows `keytool` must be run from `C:\Program Files\Java\jdkx.x.x_x\bin`, as administrator.

```
keytool -genkeypair -v -storetype PKCS12 -keystore my-upload-key.keystore -alias my-key-alias -keyalg RSA -keysize 2048 -validity 10000
```

This command prompts you for passwords for the keystore and key and for the Distinguished Name fields for your key. It then generates the keystore as a file called `my-`

`upload-key.keystore`.

The keystore contains a single key, valid for 10000 days. The alias is a name that you will use later when signing your app, so remember to take note of the alias.

macOS

On macOS, if you're not sure where your JDK bin folder is, then perform the following command to find it:

```
/usr/libexec/java_home
```

It will output the directory of the JDK, which will look something like this:

```
/Library/Java/JavaVirtualMachines/jdkX.X.X_XXX.jdk/Contents/Home
```

Navigate to that directory by using the command `cd /your/jdk/path` and use the keytool command with sudo permission as shown below.

```
sudo keytool -genkey -v -keystore my-upload-key.keystore -alias my-key-alias -keyalg RSA
-keysize 2048 -validity 10000
```

> **CAUTION**
>
> Remember to keep the keystore file private. In case you've lost upload key or it's been compromised you should follow these instructions.

Setting up Gradle variables

1. Place the `my-upload-key.keystore` file under the `android/app` directory in your project folder.
2. Edit the file `~/.gradle/gradle.properties` or `android/gradle.properties`, and add the following (replace `*****` with the correct keystore password, alias and key password),

```
MYAPP_UPLOAD_STORE_FILE=my-upload-key.keystore
MYAPP_UPLOAD_KEY_ALIAS=my-key-alias
MYAPP_UPLOAD_STORE_PASSWORD=*****
MYAPP_UPLOAD_KEY_PASSWORD=*****
```

These are going to be global Gradle variables, which we can later use in our Gradle config to sign our app.

> **NOTE ABOUT USING GIT**
>
> Saving the above Gradle variables in `~/.gradle/gradle.properties` instead of `android/gradle.properties` prevents them from being checked in to git. You may have to create the `~/.gradle/gradle.properties` file in your user's home directory before you can add the variables.

> **NOTE ABOUT SECURITY**
>
> If you are not keen on storing your passwords in plaintext, and you are running macOS, you can also store your credentials in the Keychain Access app. Then you can skip the two last rows in `~/.gradle/gradle.properties`.

Adding signing config to your app's Gradle config

The last configuration step that needs to be done is to setup release builds to be signed using upload key. Edit the file `android/app/build.gradle` in your project folder, and add the signing config,

```
...
android {
    ...
    defaultConfig { ... }
    signingConfigs {
        release {
            if (project.hasProperty('MYAPP_UPLOAD_STORE_FILE')) {
                storeFile file(MYAPP_UPLOAD_STORE_FILE)
                storePassword MYAPP_UPLOAD_STORE_PASSWORD
```

```
                keyAlias MYAPP_UPLOAD_KEY_ALIAS
                keyPassword MYAPP_UPLOAD_KEY_PASSWORD
            }
        }
    }
    buildTypes {
        release {
            ...
            signingConfig signingConfigs.release
        }
    }
}
...
```

Generating the release AAB

Run the following command in a terminal:

```
npx react-native build-android --mode=release
```

This command uses Gradle's `bundleRelease` under the hood that bundles all the JavaScript needed to run your app into the AAB (Android App Bundle). If you need to change the way the JavaScript bundle and/or drawable resources are bundled (e.g. if you changed the default file/folder names or the general structure of the project), have a look at `android/app/build.gradle` to see how you can update it to reflect these changes.

> **NOTE**
>
> Make sure `gradle.properties` does not include `org.gradle.configureondemand=true` as that will make the release build skip bundling JS and assets into the app binary.

The generated AAB can be found under `android/app/build/outputs/bundle/release/app-release.aab`, and is ready to be uploaded to Google Play.

In order for Google Play to accept AAB format the App Signing by Google Play needs to be configured for your application on the Google Play Console. If you are updating an

existing app that doesn't use App Signing by Google Play, please check our migration section to learn how to perform that configuration change.

Testing the release build of your app

Before uploading the release build to the Play Store, make sure you test it thoroughly. First uninstall any previous version of the app you already have installed. Install it on the device using the following command in the project root:

npm **Yarn**

```
yarn android --mode release
```

Note that `--mode release` is only available if you've set up signing as described above.

You can terminate any running bundler instances, since all your framework and JavaScript code is bundled in the APK's assets.

Publishing to other stores

By default, the generated APK has the native code for both `x86`, `x86_64`, `ARMv7a` and `ARM64-v8a` CPU architectures. This makes it easier to share APKs that run on almost all Android devices. However, this has the downside that there will be some unused native code on any device, leading to unnecessarily bigger APKs.

You can create an APK for each CPU by adding the following line in your `android/app/build.gradle` file:

```
android {

    splits {
        abi {
            reset()
            enable true
            universalApk false
```

```
            include "armeabi-v7a", "arm64-v8a", "x86", "x86_64"
        }
    }

}
```

Upload these files to markets which support device targeting, such as Amazon AppStore or F-Droid, and the users will automatically get the appropriate APK. If you want to upload to other markets, such as APKFiles, which do not support multiple APKs for a single app, change the `universalApk false` line to `true` to create the default universal APK with binaries for both CPUs.

Please note that you will also have to configure distinct version codes, as suggested in this page from the official Android documentation.

Enabling Proguard to reduce the size of the APK (optional)

Proguard is a tool that can slightly reduce the size of the APK. It does this by stripping parts of the React Native Java bytecode (and its dependencies) that your app is not using.

> **IMPORTANT**
>
> Make sure to thoroughly test your app if you've enabled Proguard. Proguard often requires configuration specific to each native library you're using. See `app/proguard-rules.pro`.

To enable Proguard, edit `android/app/build.gradle`:

```
/**
 * Run Proguard to shrink the Java bytecode in release builds.
 */
def enableProguardInReleaseBuilds = true
```

Migrating old Android React Native apps to use App Signing by Google Play

If you are migrating from previous version of React Native chances are your app does not use App Signing by Google Play feature. We recommend you enable that in order to take advantage from things like automatic app splitting. In order to migrate from the old way of signing you need to start by generating new upload key and then replacing release signing config in `android/app/build.gradle` to use the upload key instead of the release one (see section about adding signing config to gradle). Once that's done you should follow the instructions from Google Play Help website in order to send your original release key to Google Play.

Default Permissions

By default, `INTERNET` permission is added to your Android app as pretty much all apps use it. `SYSTEM_ALERT_WINDOW` permission is added to your Android APK in debug mode but it will be removed in production.

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