Speeding up your Build phase

Building your React Native app could be **expensive** and take several minutes of developers time. This can be problematic as your project grows and generally in bigger organizations with multiple React Native developers.

To mitigate this performance hit, this page shares some suggestions on how to **improve your build time**.

INFO

If you're noticing slower build time with the **New Architecture on Android**, we recommend to upgrade to React Native 0.71

Build only one ABI during development (Android-only)

When building your android app locally, by default you build all the 4 Application Binary Interfaces (ABIs) : `armeabi-v7a`, `arm64-v8a`, `x86` & `x86_64`.

However, you probably don't need to build all of them if you're building locally and testing your emulator or on a physical device.

This should reduce your **native build time** by a ~75% factor.

If you're using the React Native CLI, you can add the `--active-arch-only` flag to the `run-android` command. This flag will make sure the correct ABI is picked up from either the running emulator or the plugged in phone. To confirm that this approach is working fine, you'll see a message like `info Detected architectures arm64-v8a` on console.

```
$ yarn react-native run-android --active-arch-only

[ ... ]
info Running jetifier to migrate libraries to AndroidX. You can disable it using "--no-
jetifier" flag.
Jetifier found 1037 file(s) to forward-jetify. Using 32 workers...
```

```
info JS server already running.
info Detected architectures arm64-v8a
info Installing the app...
```

This mechanism relies on the `reactNativeArchitectures` Gradle property.

Therefore, if you're building directly with Gradle from the command line and without the CLI, you can specify the ABI you want to build as follows:

```
$ ./gradlew :app:assembleDebug -PreactNativeArchitectures=x86,x86_64
```

This can be useful if you wish to build your Android App on a CI and use a matrix to parallelize the build of the different architectures.

If you wish, you can also override this value locally, using the `gradle.properties` file you have in the top-level folder of your project:

```
# Use this property to specify which architecture you want to build.
# You can also override it from the CLI using
# ./gradlew <task> -PreactNativeArchitectures=x86_64
reactNativeArchitectures=armeabi-v7a,arm64-v8a,x86,x86_64
```

Once you build a **release version** of your app, don't forget to remove those flags as you want to build an apk/app bundle that works for all the ABIs and not only for the one you're using in your daily development workflow.

Use a compiler cache

If you're running frequent native builds (either C++ or Objective-C), you might benefit from using a **compiler cache**.

Specifically you can use two type of caches: local compiler caches and distributed compiler caches.

Local caches

INFO

The following instructions will work for **both Android & iOS**. If you're building only Android apps, you should be good to go. If you're building also iOS apps, please follow the instructions in the XCode Specific Setup section below.

We suggest to use **ccache** to cache the compilation of your native builds. Ccache works by wrapping the C++ compilers, storing the compilation results, and skipping the compilation if an intermediate compilation result was originally stored.

To install it, you can follow the official installation instructions.

On macOS, we can install ccache with `brew install ccache`. Once installed you can configure it as follows to cache NDK compile results:

```
ln -s $(which ccache) /usr/local/bin/gcc
ln -s $(which ccache) /usr/local/bin/g++
ln -s $(which ccache) /usr/local/bin/cc
ln -s $(which ccache) /usr/local/bin/c++
ln -s $(which ccache) /usr/local/bin/clang
ln -s $(which ccache) /usr/local/bin/clang++
```

This will create symbolic links to `ccache` inside the `/usr/local/bin/` which are called `gcc`, `g++`, and so on.

This works as long as `/usr/local/bin/` comes first than `/usr/bin/` inside your `$PATH` variable, which is the default.

You can verify that it works using the `which` command:

```
$ which gcc
/usr/local/bin/gcc
```

If the results is `/usr/local/bin/gcc`, then you're effectively calling `ccache` which will wrap the `gcc` calls.

CAUTION

Please note that this setup of `ccache` will affect all the compilations that you're running on your machine, not only those related to React Native. Use it at your own risk. If you're failing to install/compile other software, this might be the reason. If that is the case, you can remove the symlink you created with:

```
unlink /usr/local/bin/gcc
unlink /usr/local/bin/g++
unlink /usr/local/bin/cc
unlink /usr/local/bin/c++
unlink /usr/local/bin/clang
unlink /usr/local/bin/clang++
```

to revert your machine to the original status and use the default compilers.

You can then do two clean builds (e.g. on Android you can first run `yarn react-native run-android`, delete the `android/app/build` folder and run the first command once more). You will notice that the second build was way faster than the first one (it should take seconds rather than minutes). While building, you can verify that `ccache` works correctly and check the cache hits/miss rate `ccache -s`

```
$ ccache -s
Summary:
  Hits:               196 /  3068 (6.39 %)
    Direct:             0 /  3068 (0.00 %)
    Preprocessed:     196 /  3068 (6.39 %)
  Misses:            2872
    Direct:          3068
    Preprocessed:    2872
  Uncacheable:          1
Primary storage:
  Hits:               196 /  6136 (3.19 %)
  Misses:            5940
  Cache size (GB): 0.60 / 20.00 (3.00 %)
```

Note that `ccache` aggregates the stats over all builds. You can use `ccache --zero-stats` to reset them before a build to verify the cache-hit ratio.

Should you need to wipe your cache, you can do so with `ccache --clear`

XCode Specific Setup

To make sure `ccache` works correctly with iOS and XCode, you need to follow a couple of extra steps:

1. You must alter the way Xcode and `xcodebuild` call for the compiler command. By default they use *fully specified paths* to the compiler binaries, so the symbolic links installed in `/usr/local/bin` will not be used. You may configure Xcode to use *relative* names for the compilers using either of these two options:

- environment variables prefixed on the command line if you use a direct command line: `CLANG=clang CLANGPLUSPLUS=clang++ LD=clang LDPLUSPLUS=clang++ xcodebuild <rest of xcodebuild command line>`
- A `post_install` section in your `ios/Podfile` that alters the compiler in your Xcode workspace during the `pod install` step:

```
post_install do |installer|
  react_native_post_install(installer)

  # ...possibly other post_install items here

  installer.pods_project.targets.each do |target|
    target.build_configurations.each do |config|
      # Using the un-qualified names means you can swap in different implementations,
for example ccache
      config.build_settings["CC"] = "clang"
      config.build_settings["LD"] = "clang"
      config.build_settings["CXX"] = "clang++"
      config.build_settings["LDPLUSPLUS"] = "clang++"
    end
  end

  __apply_Xcode_12_5_M1_post_install_workaround(installer)
end
```

2. You need a ccache configuration that allows for a certain level of sloppiness and cache behavior such that ccache registers cache hits during Xcode compiles. The

ccache configuration variables that are different from standard are as follows if configured by environment variable:

```
export
CCACHE_SLOPPINESS=clang_index_store,file_stat_matches,include_file_ctime,include_file_mtime
export CCACHE_FILECLONE=true
export CCACHE_DEPEND=true
export CCACHE_INODECACHE=true
```

The same may be configured in a `ccache.conf` file or any other mechanism ccache provides. More on this can be found in the official ccache manual.

Using this approach on a CI

Ccache uses the `/Users/$USER/Library/Caches/ccache` folder on macOS to store the cache. Therefore you could save & restore the corresponding folder also on CI to speedup your builds.

However, there are a couple of things to be aware:

1. On CI, we recommend to do a full clean build, to avoid poisoned cache problems. If you follow the approach mentioned in the previous paragraph, you should be able to parallelize the native build on 4 different ABIs and you will most likely not need `ccache` on CI.

2. `ccache` relies on timestamps to compute a cache hit. This doesn't work well on CI as files are re-downloaded at every CI run. To overcome this, you'll need to use the `compiler_check content` option which relies instead on hashing the content of the file.

Distributed caches

Similar to local caches, you might want to consider using a distributed cache for your native builds. This could be specifically useful in bigger organizations that are doing frequent native builds.

We recommend to use sccache to achieve this. We defer to the sccache distributed compilation quickstart for instructions on how to setup and use this tool.

Is this page useful?

Edit this page

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