fastlane Tips and Tricks: Top fastlane Actions and How to Use Them

APP DEVELOPMENT

fastlane Android iOS

Derrick Wadek

AUGUST 20, 2023

So you've been working on your project for months, weathering countless test failures and bug fixes in the process, and now your app is ready for release. Submitting to the

straightforward as you'd think. You have to build the correct binaries for each store, manage provisioning profiles and distribution certs and sign the binaries properly, sort out dozens of screenshots (maybe hundreds if your app is multilingual), and update your app's metadata. It's the kind of menial, repetitive work that quickly wastes time and is prone to errors.

This is where tools like fastlane really shine. fastlane is a mobile release automation tool that helps you streamline tasks like testing, building, signing, and deploying your application to the Apple and Google app stores. fastlane *actions* are the scripts you can run to automate processes, and *lanes* allow you to create defined workflows tailored to your specific needs.

In this overview, we'll share some of the most widely used fastlane actions that can help you build, sign, release, and push assets automatically. We'll mostly focus on the highlights here, but if you'd like to see a step-by-step tutorial for using fastlane to automate your iOS app release, check out this detailed tutorial.

Building your app with fastlane gym or gradle

gym - also known as `build_app` (there are lots of aliases in fastlane!) - is a fastlane action that helps you build and package your iOS applications by automating beta and production releases. It's primarily used to build and sign your app and generates a signed .ipa file as output.

To start a build with gym, run:

```
bundle exec fastlane gym
```

complex.

For example, the following runs gym with bitcode and symbols included in the .ipa file:

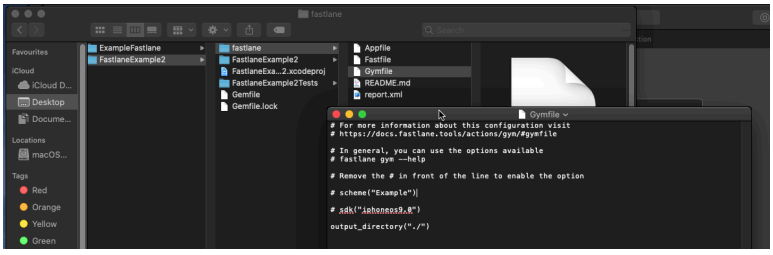
```
bundle exec fastlane gym --
include_bitcode true --include_symbols
true
```

Manually specifying these additional parameters every time you use the gym action can be tedious. By using a Gymfile to define defaults, you can avoid manually inputting each parameter every time you run gym.

The code below creates a new Gymfile:

```
bundle exec fastlane gym init
```

If you want a little more control over your build process, edit the parameters in your default Gymfile accordingly. A new, empty Gymfile should look like this:



you can specify the build parameters used in each environment:

```
lane :beta do
 scan
 gym(scheme: "MyApp")
 crashlytics
end
```

On the Android side, the gradle action leverages your existing Gradle build to handle building of your app.

Apple code signing with fastlane match

Apple recommends that every development machine has a unique identity for signing code. This proliferation of signing identities becomes a challenge for large teams when everyone needs to download the newest provisioning profiles whenever a certificate expires or a new developer joins.

fastlane has an action called *match* that elegantly solves this pain point by creating a single identity that can be shared across your entire team. It encrypts and stores your keys and certificates in a private git repository. By setting permissions (e.g., `read-and-write` or `read-only`) and enabling two-factor

Let's look at how to get started with match and some of the key features it offers.

The command below will initialize match:

```
bundle exec fastlane match init
```

match gives you the option of storing your keys and profiles with a cloud hosting provider (like Google Cloud or AWS) or a private git repository (like GitHub or Bitbucket). After you've selected your preferred storage option, fastlane will create a new Matchfile in the fastlane folder. This file ensures that your configuration options are committed to version control and shared with the rest of your team.

After you've initialized match, you must generate or download your team's certificate and provisioning profiles by running the following commands:

```
# Generate and download a new
provisioning file for development use
bundle exec fastlane match development

# Generate a distribution certificate
and provisioning file for app store use
bundle exec fastlane match appstore
```

The nice thing is that, once configured with your storage provider, match will

the `--readonly` flag:

```
bundle exec fastlane match appstore --
readonly
```

If you stop working on an app for a while and the certificate lapses, you won't be able to auto-renew it, but you can remove and regenerate it manually. If using git, this process should look something like this:

```
# Clone the old cert repo and remove the
profiles and certificates
git clone git@github.com:example/app-
certs.git
cd app-certs
git rm -r certs/development
profiles/development
git commit -m 'Removes expired dev
certificates and profiles'
git push origin
cd ..
rm -Rf app-certs

# Generate the profile and certificate
again
bundle exec fastlane match development
```

TestFlight deployment with fastlane pilot

Uploading your app to TestFlight is another task that's great to automate with fastlane. Each time you make an update, Testflight requires you to manually log in to the Apple

screenshots, and upload the relevant builds to initiate TestFlight.

With fastlane, the upload part of this process can be automated with the *pilot* action (an alias for `upload_to_testflight`). pilot commands require either an API key or your Apple ID, but once authenticated, fastlane will do the hard work of uploading your app and assets to TestFlight.

Assuming you've already built your .ipa file manually or with gym as described above, you can run pilot with the following command:

```
bundle exec fastlane pilot upload
```

This action will upload the .ipa file in your project directory and use the login credentials from your fastlane configuration.

There are several flags you can include in the pilot action. For example, you can specify the filepath to the .ipa file with `--ipa` or include "what to test" text with the `--changelog` flag. To debug errors encountered during the upload process with pilot, you can use the `--verbose` flag:

```
bundle exec fastlane pilot upload --
changelog "some new items" --ipa
```

Building your app with fastlane gym or gradle

Apple code signing with fastlane match

TestFlight deployment with fastlane pilot

Pushing assets to the App Store with fastlane deliver

This allows you to dynamically swap out test users and groups as a function of the type of beta you're releasing. For example, you might only want internal testers to get early beta releases, saving your external testers for more polished versions of the app.

Pushing assets to the App Store with fastlane deliver

To bring it all together, let's talk about pushing assets to the Apple App Store. fastlane *deliver* is an action that allows you to automatically upload localized screenshots, binary .ipa files, and metadata to the App Store. By integrating this action with your continuous integration and delivery pipeline, you can make releases quite hands-off for your team.

deliver stores your release configuration files in git, making them available to any developer, so you no longer need to have a single person in charge of every release. Additionally, it provides an HTML preview of the metadata you're about to send to the App Store, allowing your team to check the final product before you ship it.

This will prompt you to log in to App Store Connect if you haven't already done so in your fastlane project. It will also ask you to enter your bundle identifier to ensure you have the necessary permissions. Finally, deliver will download the current version of your app's metadata and screenshots.

To upload your binary file and submit your updated app for review, run the following:

```
bundle exec fastlane deliver --ipa
"Final.ipa" --submit_for_review
```

There are many other options you can pass into the deliver action, including language codes, custom paths for your metadata and screenshots, and your app's categories. Like all the actions outlined here, you can also add deliver's flags to your Fastfile so they're checked into version control and automatically applied:

```
lane :submit_prod do
 deliver(
   submit_for_review: true,
   automatic_release: true
 )
end
```

fastlane *supply* is the Google Play Console equivalent of fastlane *deliver*. Like deliver, supply allows you to automate the app store submission and update process and run everything via your command line or CI/CD tool of choice. supply ships new .apk files or .aab bundles, updates metadata, uploads your application icon, graphics, and screenshots, and incrementally updates the app version number.

Most of supply's features are similar to deliver's, so if you read the section above, you should be in familiar territory. To initialize supply and download the existing metadata, run the following:

```
bundle exec fastlane supply init
```

You may have a description, title, and/or video clips as part of your app's `metadata` folder:

Assuming you've already built your Android App Bundle, you can run the following command to push your updates to the Play Console:

fastlane supply supports them as well:

```
bundle exec fastlane supply --apk
path/to/app.apk
```

You can release your update to a specific track (production, beta, alpha, internal) by adding the `--track` flag. Similarly, you can promote a test version of your app to production after users have tried it out using the `--track_promote_to` flag.

There are many other parameters covered in the docs, and like the options available in deliver, you can specify these in your Fastfile to ensure developers don't forget them.

Conclusion

fastlane is an extremely capable tool, but it can quickly become a complex and sometimes fragile piece of your toolchain — so it's a good idea to get comfortable with a few basic actions and build your way towards a more fully automated ideal.

In this piece, we've highlighted some of the most helpful and commonly used fastlane actions. Together, they help you handle building, signing, and delivery to the app stores, and they form a solid foundation on

RELATED POSTS

WWDC 2022 highlights

Alessandro Martin

JUNE 7, 2022

How to scale your iOS org: sharing tooling across multiple iOS projects

Jared Sorge

AUGUST 18, 2022

Merge queues: An intro for mobile engineers

Bruno Rocha

FEBRUARY 9, 2022

Product

Mobile release management
End-to-end automation
Rollouts
Mobile insights
Build Distro
Automations
Security
Pricing
What's new
Explore sandbox

Use cases

fastlane
Release trains
Mobile DevOps
Cross-platform
- React Native
- Flutter

Integrations

Project management
- Jira
- Linear
- Pivotal Tracker

Version control
CI/CD
App stores
- App Store Connect
- Google Play Console

Slack
Monitoring
All integrations

Resources

Blog
FAQ
Documentation
Quickstart CI/CD
App review times
App Store Connect status page
App hotfix leaderboard

Company

About us
Contact
Terms of service
Privacy policy
Careers
Status