Automate away iOS code signing renewal pain with fastlane match

APP DEVELOPMENT

fastlane iOS

Jared Sorge
JUNE 23, 2023

For the most part, building iOS apps is a lot of fun. We get to solve interesting problems for our users and work with the latest and greatest Apple has to offer. But there's a time of year that we all love the least: when we need to renew our distribution signing certificates and regenerate our provisioning profiles. Signing certificates and provisioning profiles help keep users and the wider Apple ecosystem secure, but managing them can definitely cause some headaches for you and your team.

Depending on how many apps you're working on, the different configurations of those apps, and if they include any extra extensions or binaries, the renew and regenerate process can range in pain level from "nagging nuisance" to "aging you another decade in a few days".

invalidate your old certificates and profiles, generate new ones, and store the new ones in git, Google Cloud, or Amazon S3 for the rest of your team and Continuous Integration (CI) systems to use. To get started with match, check out fastlane's code signing guide.

Renewing certs & profiles with fastlane

In an earlier post we talked about configuring fastlane with environments and we'll use that to our advantage here to set things up.

```
// In .env.default file
MATCH_TYPE = appstore
MATCH_GIT_URL = { URL of your stored credentials repo }
MATCH_STORAGE_MODE = git
MATCH_GIT_BRANCH = main
MATCH_USERNAME = { Apple ID }
MATCH_KEYCHAIN_NAME = { com.example.keychain }
MATCH_OUTPUT_PATH = $(PWD)/signing
MATCH_READONLY = true

ASC_KEY_ID = { The ID of your App Store Connect API Key }
ASC_KEY_ISSUER_ID = { The Issuer ID of your App Store Connect API Key }

// In .env.secret
MATCH_PASSWORD = { Password to decrypt stored match contents }
MATCH_KEYCHAIN_PASSWORD = { Password to unlock the custom keychain }
ASC_KEY_CONTENT = { The base64 encoded content of your App Store Connect API Key }
```

What we're doing here is setting some basic values that we want to make available to match. Note that the bottom two values are stored in a file called `.env.secret`, which we'll load in with our lane to make it available as we do the work here (`.env.default` is loaded in by fastlane automatically). As its name suggests, this is a file containing sensitive info and it should not be committed to your repository! In order to set up these secrets in a CI context, check your CI provider's secrets documentation.

Let's create our custom lane now and walk through the steps that we'll need to take.

```
lane :rebuild_signing do
  # 1
  Dotenv.overload ".env.secret"

  # 2
  app_store_connect_api_key(
    key_id: ENV["ASC_KEY_ID"],
    issuer_id: ENV["ASC_KEY_ISSUER_ID"],
    key_content: ENV["ASC_KEY_CONTENT"],
    is_key_content_base64: true
  )

  # 3
  create_keychain({
```

```
  # 4
  ios_bundle_ids = [
    'com.example.ourapp.ios',
    'com.example.ourapp.ios.intents',
    # any additional iOS bundle identifiers
  ]

  # 5
  match_nuke({
    type: "appstore",
    readonly: false,
    app_identifier: ios_bundle_ids
  })

  # 6
  match({
    readonly: false,
    app_identifier: ios_bundle_ids,
    platform: 'ios'
  })
end
```

1. First thing we do is load in the environment file containing our secrets.
2. This step loads in our App Store Connect API key. The values for the key are all loaded in via the environment, and I usually extract loading the key to its own lane to avoid calling `app_store_connect_api_key` in my Fastfile from multiple places.
3. Create a custom keychain to hold our certificates. More about this below.
4. We build up an array containing bundle identifiers: for our iOS apps iOS identifiers – this encompasses all the apps and app extensions.
5. Using a function called match_nuke, we invalidate our existing certificates and provisioning profiles.
6. Running the match action now, passing in the `ios` platform and its identifiers will have it create a new signing certificate and rebuild the provisioning profiles for our iOS identifiers. We explicitly call out that `readonly` mode is now disabled – more on `readonly` below.

Run this lane and you'll be all set. All the developers on the team will receive the new certificates and profiles when they run a lane that invokes match on their machines, and your CI systems will pull down the changes when they run match as well. You can kick back and relax 😀

iOS code signing best practices

- Xcode's automatic code signing functionality has come a long way over the past couple of years. It's great for local dev builds and can take care of things like getting development certificates and profiles generated. I like to leave the feature turned on for those, but I want extra control when it comes to release builds. That's where match comes in handy.

- If you work on a larger team then there may be some additional pitfalls to be

Xcode and you'll get an overview per configuration. This can help identify what settings may need to be changed if you're encountering problems.

fastlane match best practices

- Use a custom keychain for match (that `create_keychain` call above). This will keep your login keychain uncluttered and make it clear as day what fastlane is importing into your system when it runs match.
- Specify an output directory for match to use (we do this with the `MATCH_OUTPUT_PATH` environment variable above). This will tell match where certificates, certificate signing requests (CSRs) and the like can live. If you want, you can clean this directory up at the end of the lane but it's not a big deal. Just remember to add the path to that output directory to your `gitignore` so it doesn't accidentally get checked into version control.
- Have match always default to `readonly` mode so that match won't attempt to create new certificates or provisioning profiles. This is most easily accomplished with an environment setting. When you then need to write some values then set it explicitly as an argument to match. This works because arguments take precedence over the environment when the action loads.
- The `match_nuke` action requires user intervention unless the `skip_confirmation` argument is set to `true`. Leaving the default value of `false` means that our lane will need confirmation from a human to delete the certificates and profiles. Because of the destructive nature of `match_nuke`, your team may want to require that user intervention and leave this as a manual lane to run.

fastlane match and macOS

fastlane is a multi-platform tool. If you build macOS apps as well as iOS apps then this technique can be adapted to also work with your Mac apps. Apple unified distribution certificates a few years ago, so the same certificate that signs your iOS apps and extensions can also sign your macOS apps, extensions, and other executables.

Working off of our lane above we need to make the following tweaks for macOS:

1. Create an array of bundle IDs for your macOS projects as part of step 4.
2. Update the `app_identifier` being passed to `match_nuke` in step 5 to include the macOS IDs in addition to the iOS IDs.
3. Duplicate the `match` block in step 6, changing the `app_identifier` to be the macOS IDs and the `platform` to `macos`. This will tell match to generate the profiles for the macOS platform, using your macOS IDs. It will not generate a new distribution certificate if the iOS distribution certificate has already

fastlane match: the solution to your certs and profiles headache

Without fastlane match, the dreaded process of regenerating our distribution certificate and associated provisioning profiles was an annually recurring headache. Now, by harnessing the power of fastlane, and its `match` action specifically, we're able to handle all of that by running one simple command. Cheers to no more certs and profiles headache, and happy coding!

RELATED POSTS

How to use Mobile DevOps metrics to improve your team's DevOps practice

Isabel Barrera

JANUARY 19, 2023

What is Mobile DevOps, and why does it matter?

Gabriel Savit

SEPTEMBER 2, 2021

Introducing Quickstart CI/CD by Runway

FEBRUARY 22, 2023

Mobile insights
Build Distro
Automations
Security
Pricing
What's new
Explore sandbox

Use cases

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

- Pivotal Tracker

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

Slack
Monitoring
All integrations

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