

# How to build a custom fastlane action — and share it with the world as a fastlane plugin

APP DEVELOPMENT



Jared Sorge  
APRIL 7, 2023

[Anatomy of a fastlane action](#)[Filling out the fastlane action](#)[Running the fastlane action](#)[Getting the word out: packaging your action up into a fastlane plugin](#)[Lights, camera, action!](#)

fastlane's virtues have been [praised here on the blog](#) over and over again. It's such a fantastic tool for automating everything from running tests, to capturing screenshots, to building and distributing your apps to the App Store and Play Store. If you've dabbled in fastlane before, you'll know that fastlane scripts are built up through the use of *actions* which are combined into methods called *lanes*. Lanes are basically functions that can include pure Ruby code, calls to existing fastlane actions, or a mix of both. You build up lanes in your *Fastfile* to accomplish specific tasks, and they can be invoked directly from the command line.

fastlane comes with a bevy of [pre-built actions](#) out-the-box, but you can also make your own! Plus, in addition to building custom fastlane actions for your own internal use, you can share them with the world by publishing them as fastlane [plugins](#). Plugins can be used by anyone in the fastlane ecosystem. Building custom fastlane actions can come in handy for a few different reasons: fastlane actions provide an easy way to share code between multiple repositories, which can be a way to standardize the process of building, testing, and distributing your apps. Or maybe you work on a product where there's a piece of automation that could help your customers — Sentry (a popular stability monitoring tool) for example built an action to help automate the upload of dSYM files to their servers. There are potentially many different use cases for building custom fastlane actions.

In this post we are going to look at how to build an action using a lane that we wrote about in a recent post on [automating away the pain of code signing](#). We'll adapt this lane to be its own action, and then look at how we might go about publishing it as a fastlane plugin. By the end, you'll be well-equipped to evaluate areas of overlap in your own Fastfiles, and refactor them into actions of their own.

Let's dive in!

[Product](#)[Integrations](#)[Automations](#)[Customers](#)[Pricing](#)[LOG IN](#)[CONTACT SALES](#)[GET STARTED](#)

fastlane gives you a command to create the shell of your new action: `fastlane new_action`. For our action I'm going to call it `rebuild_signing` (which mirrors what the prior lane was called). We'll get a new file at `fastlane/actions/rebuild_signing.rb` with the action code.

The file we're given is written in Ruby (like the rest of fastlane and your fastfile) so hopefully the syntax isn't too foreign. Let's take a look at the boilerplate that's given to us:

```
#0
module Fastlane
  module Actions
    # 1
    module SharedValues
      CUSTOM_VALUE = :CUSTOM_VALUE
    end

    class RebuildSigningAction < Action
      def self.run(params)
        # 2
      end

      #####
      # @!group Documentation
      #####

      def self.description
        # 3
      end

      def self.details
        # 4
      end

      def self.available_options
        # 5
        []
      end

      def self.output
        # 6
        []
      end

      def self.return_value
        # 7
      end

      def self.authors
        # 8
        []
      end

      def self.is_supported?(platform)
        # 9
        platform == :ios
      end
    end
  end
end
```

0. This class that we get becomes part of the `Fastlane.Actions` namespace. This stays as it is.

1. `SharedValues` works together with #6 to define the output — if any — of your action. This is how your action could interact with other actions, or how a lane using your action would work with your action's output. It is not necessary to have any output (and our action will not). The

[Product](#)[Integrations](#)[Automations](#)[Customers](#)[Pricing](#)[LOG IN](#)[CONTACT SALES](#)[GET STARTED](#)

2. The `run` method is part of the `Action` superclass that we are subclassing. It's the meat of what the action will do. Any output we define in items 1 & 6 will be filled in by this method.
3. Now we get into documentation. This little block is a short (80 characters or less) blurb about what the action does.
4. This is where we can expand documentation with additional details about what the action does.
5. `available_options` lets you define inputs for your action. Inputs can be passed in to your action via arguments or environment variables — and thankfully fastlane takes care of mapping these values to the action so you're not left with code paths that check both arguments and environment variables for inputs. There are a lot of properties that can be sent into an option (which are all [very helpfully documented here](#)). There's some really good stuff in here, like value validation and default values.
6. This output array is what is prettily formatted by fastlane at the beginning of your action running (if you've used `match`, `gym`, `scan`, or any of the standard tools, you've seen these). Each value in the array is a 2-member tuple of strings with the first member being a key containing an exported shared value (like we defined in #1) and the value is a description for what that value is.
7. If your action is going to return a value then this is where that is described.
8. The `authors` block is where you take credit for your creation 😊
9. Last but not least, the `platform` block lets you run logic that tells fastlane what platforms your action will support (if it is platform-bound).

## Filling out the fastlane action

Full code of our action is [available in this gist](#).

Now that we've seen what the shell of an action looks like, let's add some functionality to it. Our custom action is pretty simple and won't need to have any return values. Let's first take a look at what an input to the action will look like.

```
FastlaneCore::ConfigItem.new(  
  key: :ios_bundle_ids,  
  env_name: 'FL_IOS_BUNDLE_IDS',  
  description: 'An array of strings which are the iOS bundle IDs to re-provision. These should be provided as an array',  
  type: Array  
)
```

Here, our input is an array of bundle identifiers which will get the renewed provisioning profiles. fastlane will look for the input from the method call directly and then look to the `FL_IOS_BUNDLE_IDS` environment variable. If neither of those are present, an error will be thrown and the action will exit.

There's also an optional `verify_block` parameter to a new config item which lets you validate

## Running the fastlane action

Every action subclass has a `run` method attached that fastlane uses to do its work. There is a single argument to the method of `params`, which is a key-value pair hash of all the config items defined in the `self.available_options` array. Accessing a config item is done by using the key for a given item as a subscript. So, to get at our array of bundle IDs we'll use `params[:ios_bundle_ids]` (the leading `:` is Ruby syntax).

Inside the action's `run` method we can do lots of different things. We can execute shell commands using `sh`. We can use the `UI` class methods to communicate information, warnings, and errors to the user. And we can call other fastlane actions from our action's `run` method. To call a fastlane action, we need to prefix the action name with `other_action` — literally. For example, `other_action.match` calls the `match` action.

We can even import other gems into our actions to do what we need. It really is super flexible.

## Getting the word out: packaging your action up into a fastlane plugin

The action we created lives inside of our local fastlane directory and is picked up automatically. But what if we wanted to distribute this action to others? There's a way to do that too! Actions can be packaged up as [fastlane Plugins](#). They are then distributed as [Ruby gems](#) and imported as plugins in to fastlane via the `Pluginfile` using the same `gem` syntax as the `Gemfile`.

There's one tweak to make in the `Gemfile` so that it looks for the `Pluginfile`:

```
plugins_path = File.join(File.dirname(__FILE__), 'fastlane', 'Pluginfile')
eval_gemfile(plugins_path) if File.exist?(plugins_path)
```

This supplies the path to your `Pluginfile` and tells `bundler` to evaluate that file if it exists. Once this is done, your `bundle install` command will pick up the plugins as gems and you're in business.

## Lights, camera, action!

Having the ability to build out your own custom actions is undoubtedly a very powerful feature of fastlane — but it can come with some difficulties. Debugging custom actions can be tricky, documentation hard to come by, and the syntax to call other actions from your action is easy to mis-remember (like forgetting to call `resume` on a `URLSessionDataTask` 😅).

But getting custom actions in place can also prove incredibly powerful. They are easy to share, so if you work with multiple repositories, sharing functionality across projects becomes simpler. The validation on action input can be pretty useful for making sure that you're working with correct values. And being able to make your actions available to the world as plugins (and take advantage of plugins others have written!) certainly makes the fastlane ecosystem more dynamic and accessible. If there's some piece of functionality you need from fastlane that doesn't exist, there's a good chance that someone out there has already created a plugin for it.

Hopefully this post has helped demystify fastlane actions a bit and has even gotten your wheels


Release better with Runway.

Runway integrates with all the tools you’re already using to level-up your release coordination and automation, from kickoff to release to rollout. No more cat-herding, spreadsheets, or steady drip of manual busywork.

CONTACT SALESGET STARTED

RELATED POSTS


WWDC 2021 highlights



Alessandro Martin

JUNE 10, 2021

Leaving iOS or: How I Learned to Stop Worrying and Love the Web



Jan Klaus

NOVEMBER 15, 2022

Introducing Build Distro by Runway

MAY 23, 2023

SIGN UP FOR THE FLIGHT DECK, OUR MONTHLY NEWSLETTER

Email

SIGN UP

TRUSTED BY THE BEST MOBILE TEAMS



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

