SPEEDING UP YOUR IOS DEVELOPMENT

WITH FASTLANE

JACOB VAN ORDER

BEFORE WE GET STARTED . . .

I’d like to prevent anyone wasting their time at this excellent conference

2 THINGS:

http://files.starwarsuncut.com/assets/film_2/hd/361.jpg

I am sick while giving this presentation.
I have a cold and feel terrible.

from: http://to-hollywood-and-beyond.wikia.com/wiki/Fastlane_(2002)

This talk is not about the edgy television show that aired on Fox from 2002 to 2003 and starred Bill Bellamy, Tiffany Amber-Theissen, and that other guy.

Anyone who assumed so should probably go to the Daniel Steinberg talk. No shame in leaving right now.

I'll start by saying this talk and the associated code is available on GitHub. That way you can worry about judging me and my speaking style and less on taking notes.

AGENDA

- **The Problem(s)** (5 mins)
- **Fastlane Overview** (15 min)
- **How You Might Use It** (20 min)
- **Tips, Gotchas** (5 min)
- **Custom Action** *time permitting* (10 min)
- **Q/A** (10 min)

Let’s jump right in.

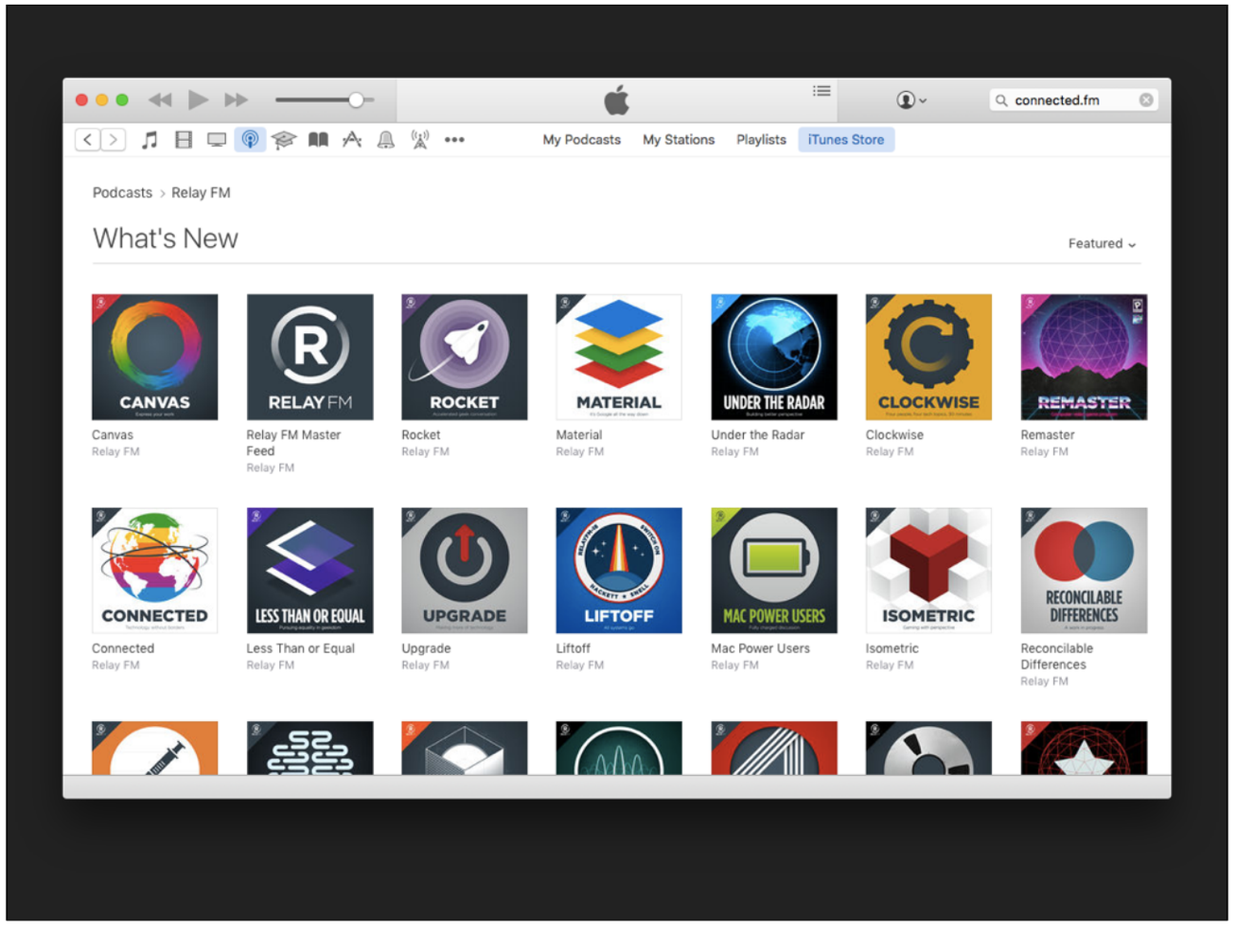
SPEEDING UP YOUR IOS DEVELOPMENT WITH FAST LANE

DEVELOPING FOR IOS CAN BE HARD

Let’s start with the honest truth…

from:http://imgur.com/gallery/Y7Ig3

Counting all your money from the app store sales



Constantly recording your podcasts

from: wikipedia.org

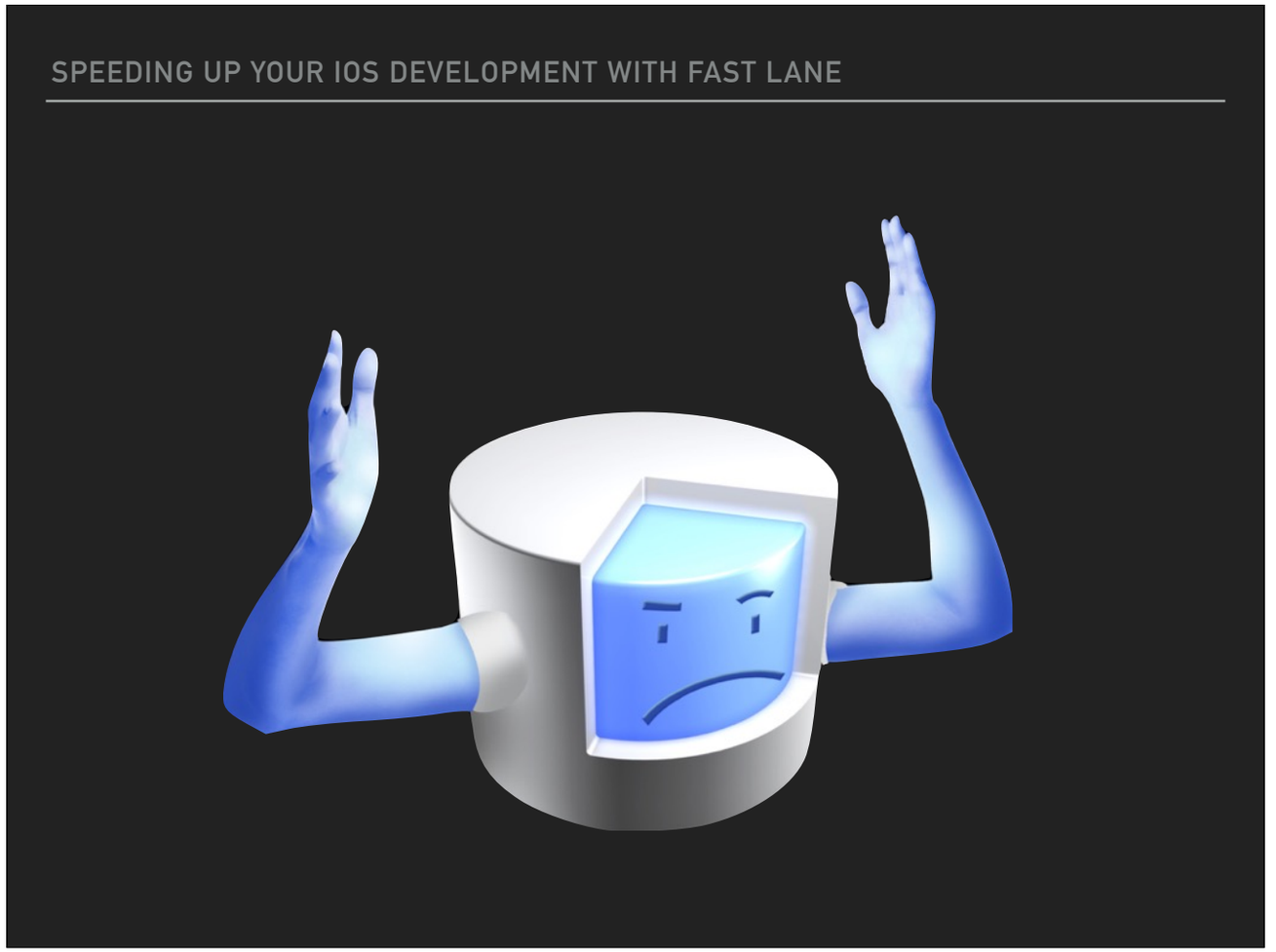
Driving around in your BWM

WHO HAS TIME FOR:

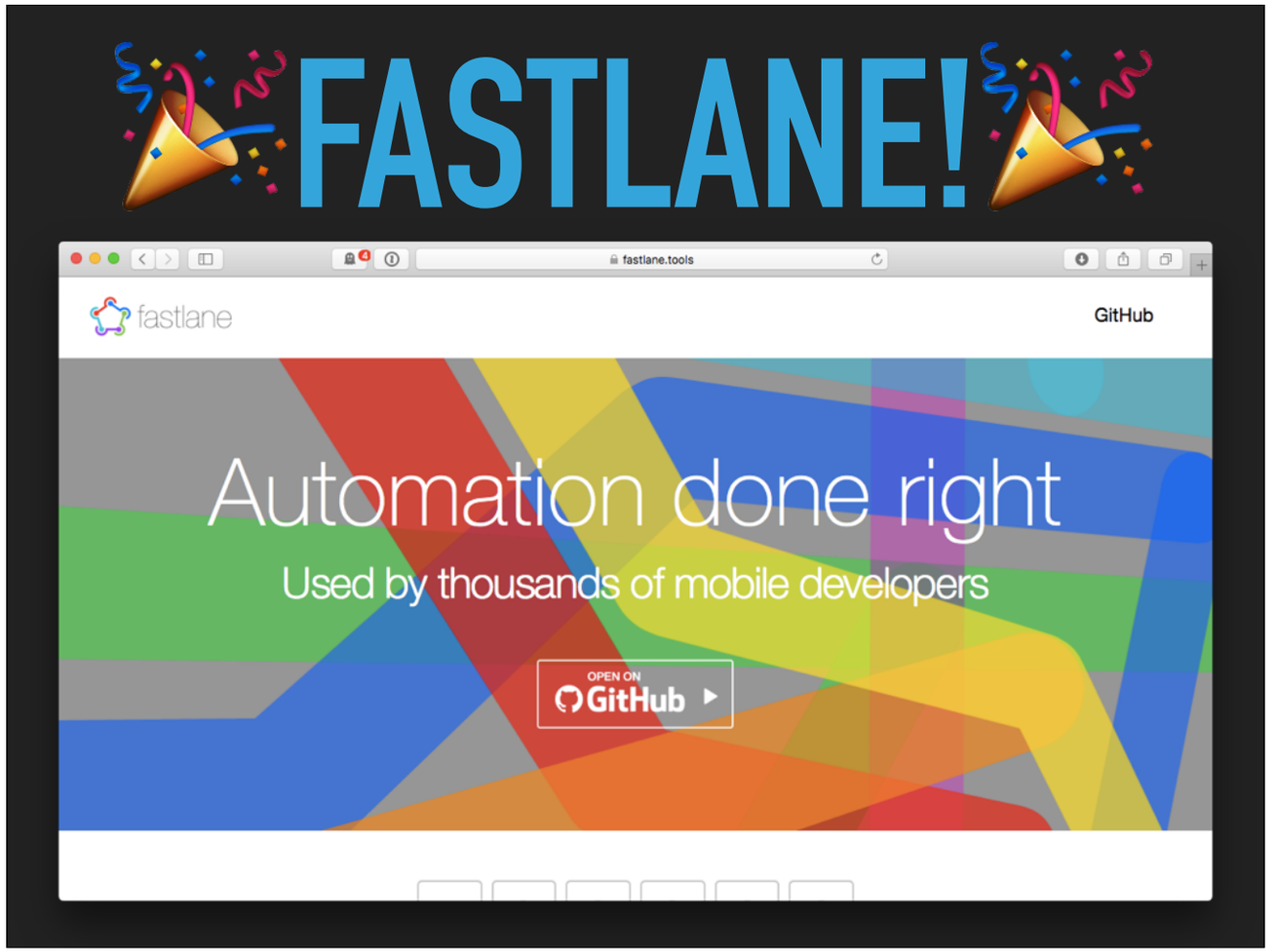
- Remembering to do Unit Testing
- Building .ipa, .dsym, .xcarchive files for yourself or QA
- Provisioning Profiles / Certificates
- Managing App Store Metadata

Not me!

So what is going to come and do my job for me?



So, what’s an iOS developer supposed to do?



Luckily Fastlane is here to solve all of your problems ever.

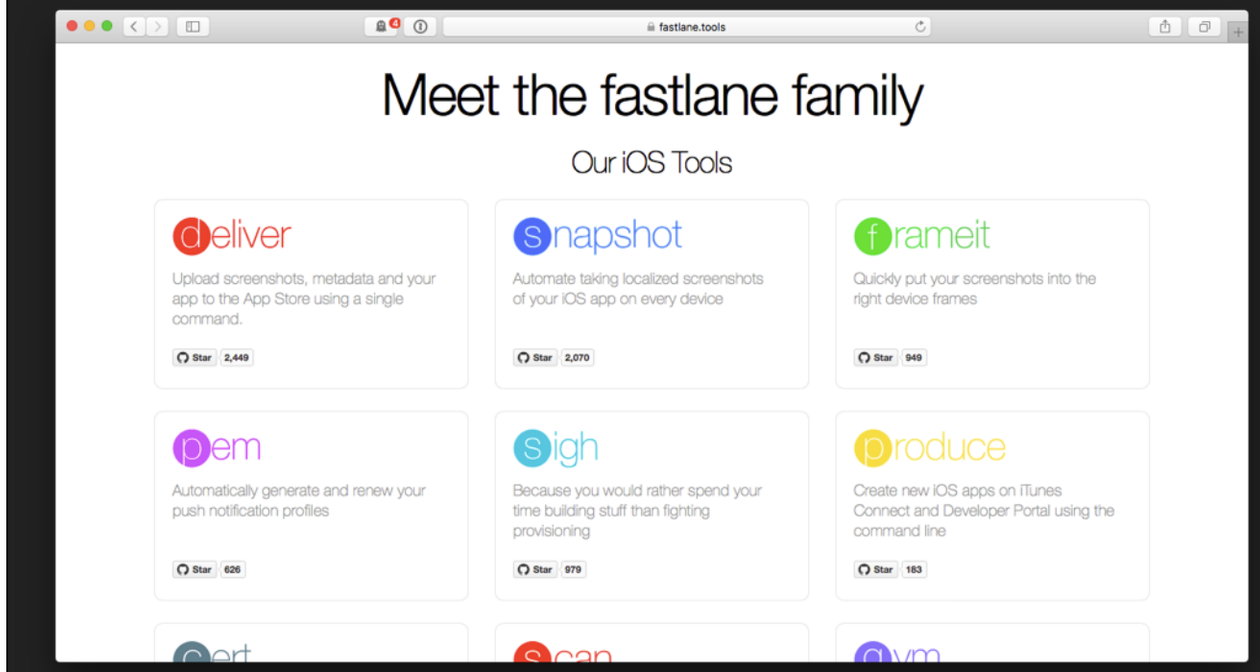
WHAT IS FASTLANE?

IT'S A RUBY GEM

from: http://blog.mailgun.com/the-official-mailgun-ruby-sdk-is-here/

Don’t let that dissuade you!

COLLECTION OF TOOLS



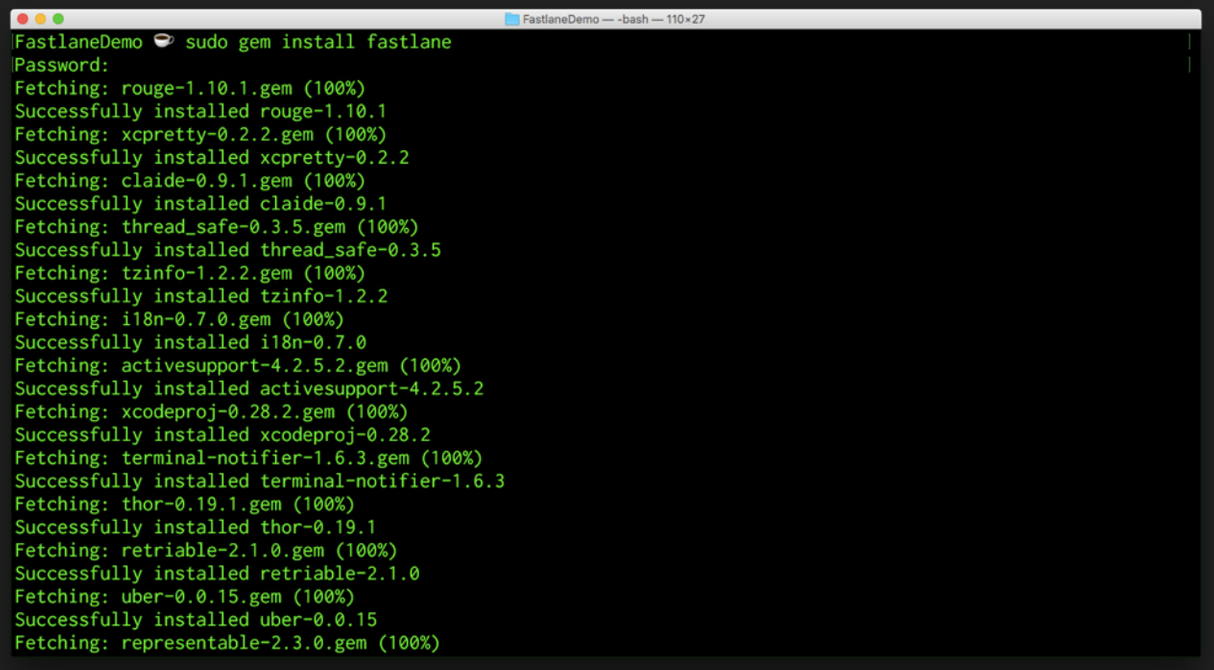
Collection of tools to smooth out the rough patches on getting to the App Store.
It’s open source on Github and free.

IN THEIR WORDS...

"FASTLANE LETS YOU DEFINE AND RUN YOUR DEPLOYMENT PIPELINES FOR DIFFERENT ENVIRONMENTS. IT HELPS YOU UNIFY YOUR APP'S RELEASE PROCESS AND AUTOMATE THE WHOLE PROCESS. FASTLANE CONNECTS ALL FASTLANE TOOLS AND THIRD PARTY TOOLS, LIKE COCOAPODS AND GRADLE."

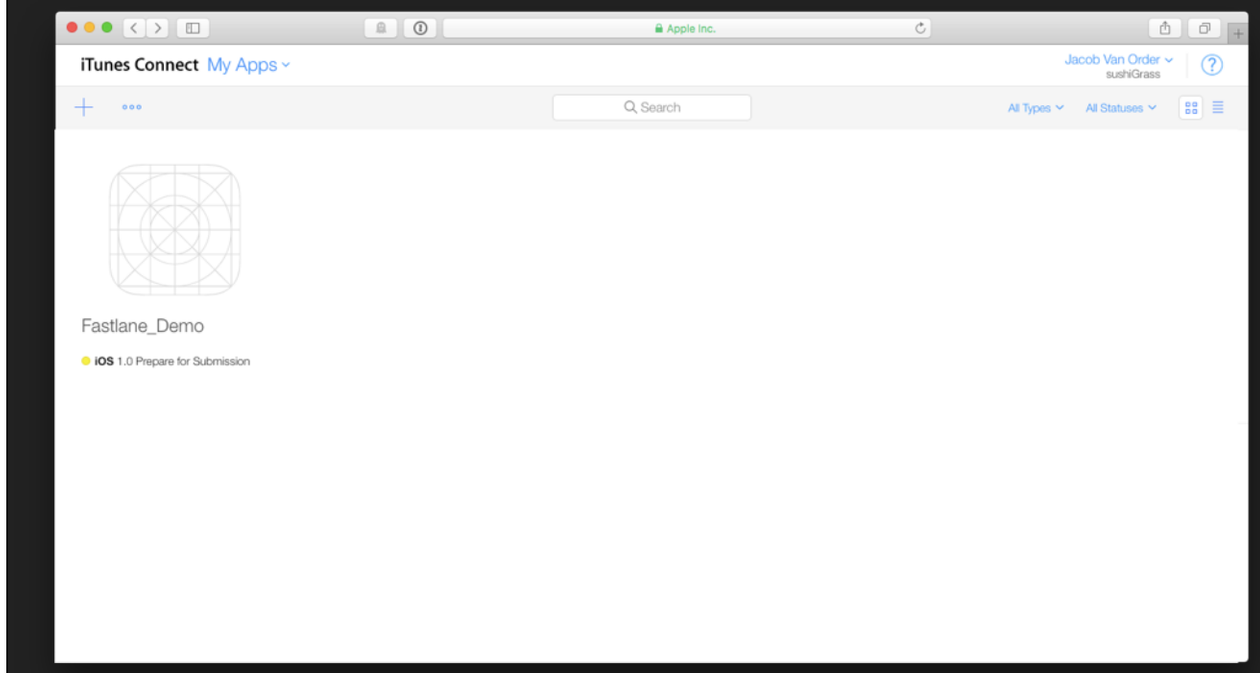
I emphasized “you write ruby” and want to make note that it’s not too late to move to Daniel Steinberg’s talk.

SETUP? EASY!



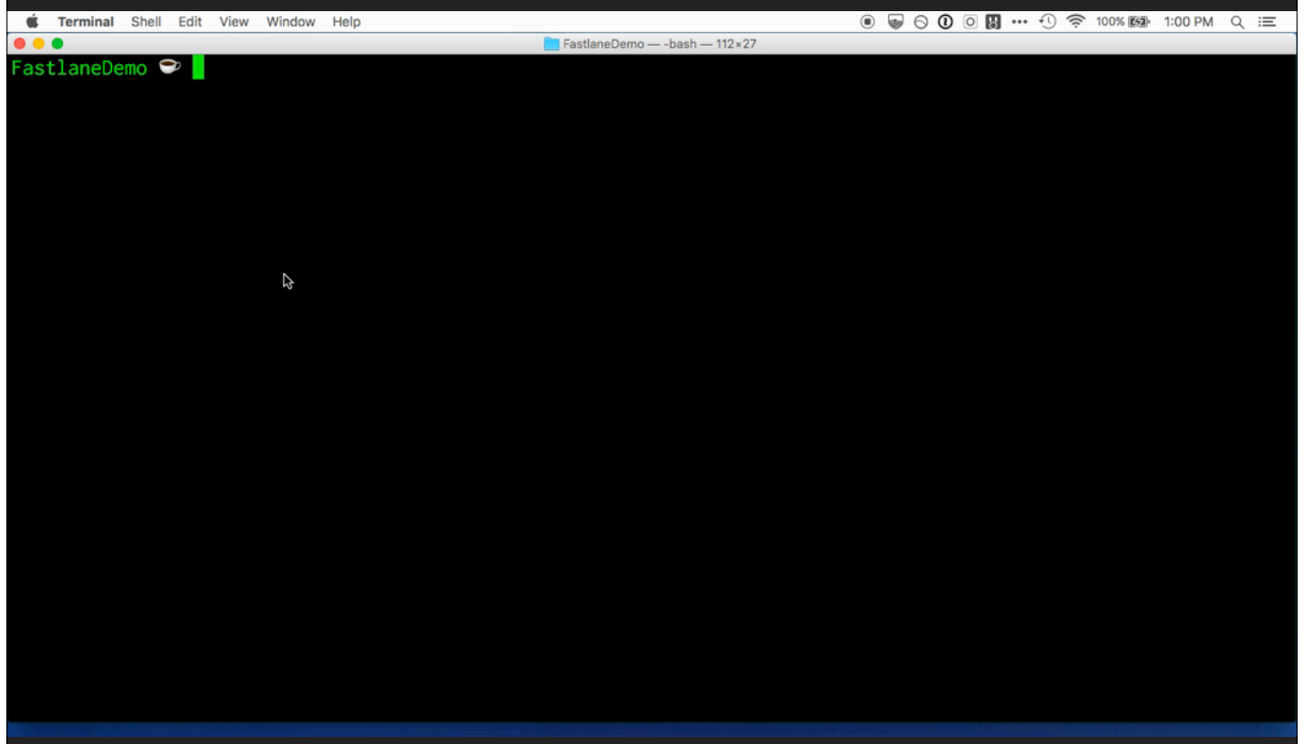
`sudo gem install fastlane`
Yes, I know I shouldn’t use `sudo` but instead a ruby manager like RVM
In my experience, adding another layer of mystery introduces another realm of complications.

ONE ASSUMPTION:



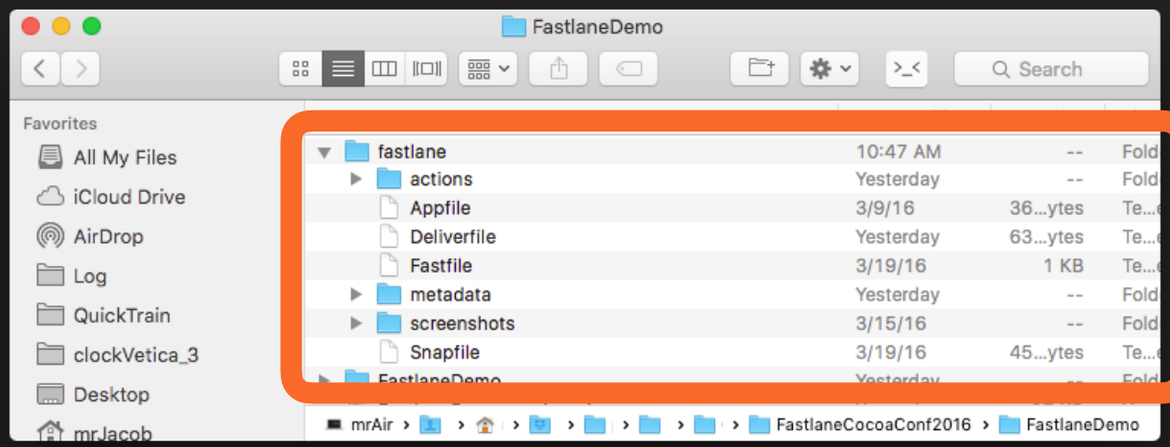
You have set up the app ID in the developer provisioning portal of death and the app itself in iTunes Connect.
Fastlane can even be used to do that but for brevity, I'm skipping that.

$ fastlane init



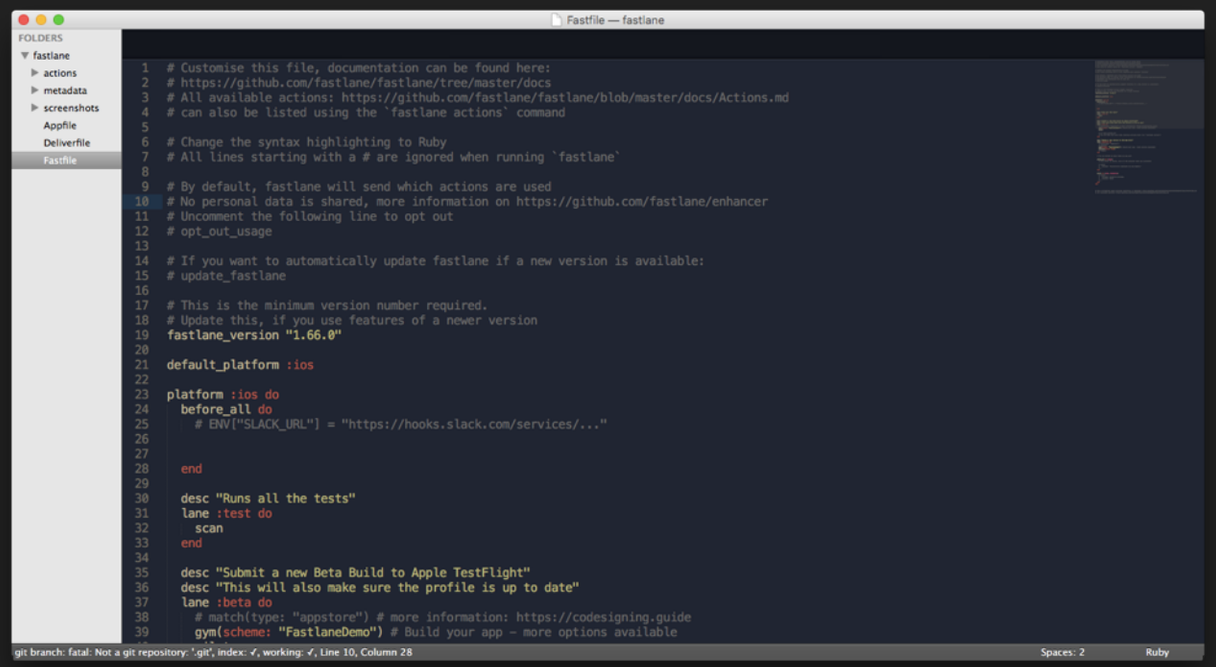
Type in `fastlane init` in your project.
It will ask for your Apple ID (and password)
It will confirm the app you already set up in iTunes Connect.
It will set up Deliver, Fastfile, and let you know it’s going to send bug information unless you opt out.

WHAT JUST HAPPENED?



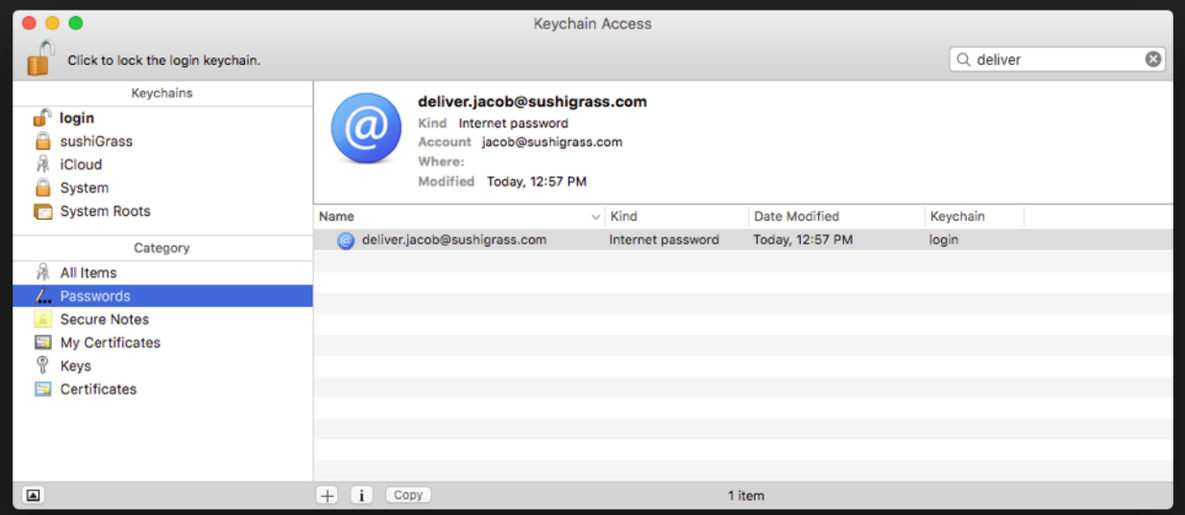
It just created a folder in your project that looks like this.
It should be under version control.
You have some folders
The Appfile which has general information about your app
The Deliverfile will be used by the Deliver tool which we'll talk about later.
But that last one is the Fastfile which is the center of all of this.

FASTFILE



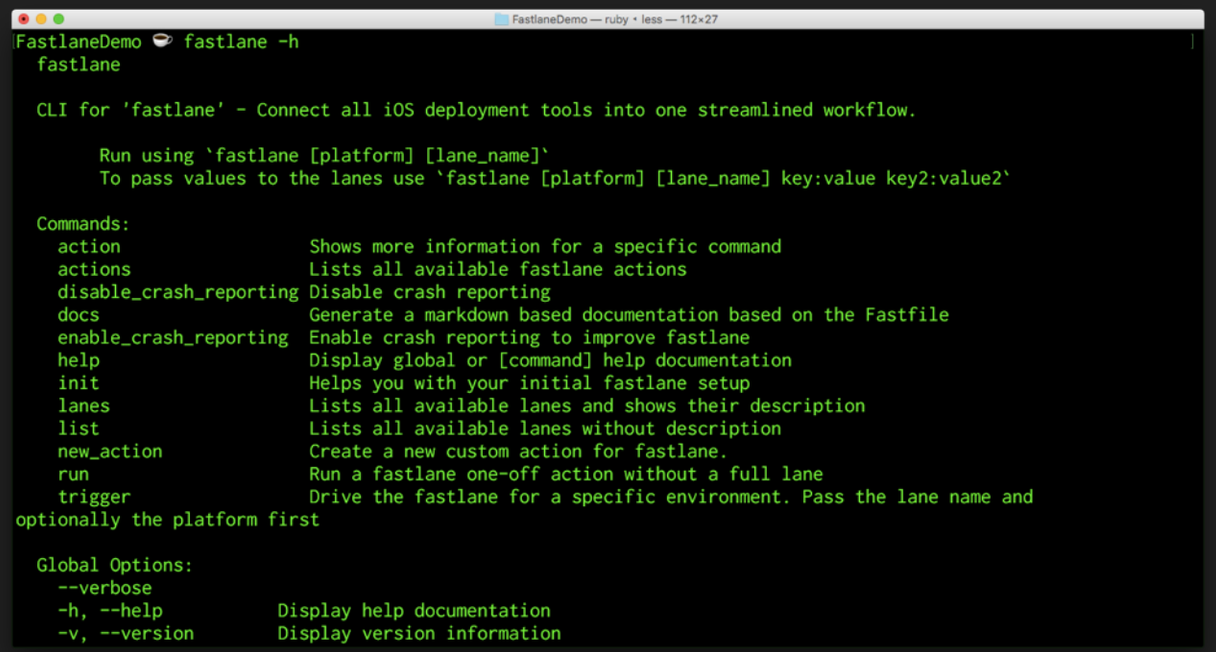
We’ll get to this soon enough but this is generated and is where you start.

USERNAME/PASSWORD?



That username and password you just entered are saved in the Keychain prefixed with “deliver”.

HACKERTIME



You can now use the tool from the command line.

NOPE

If you want to back out:
Delete that folder
Delete that keychain entry

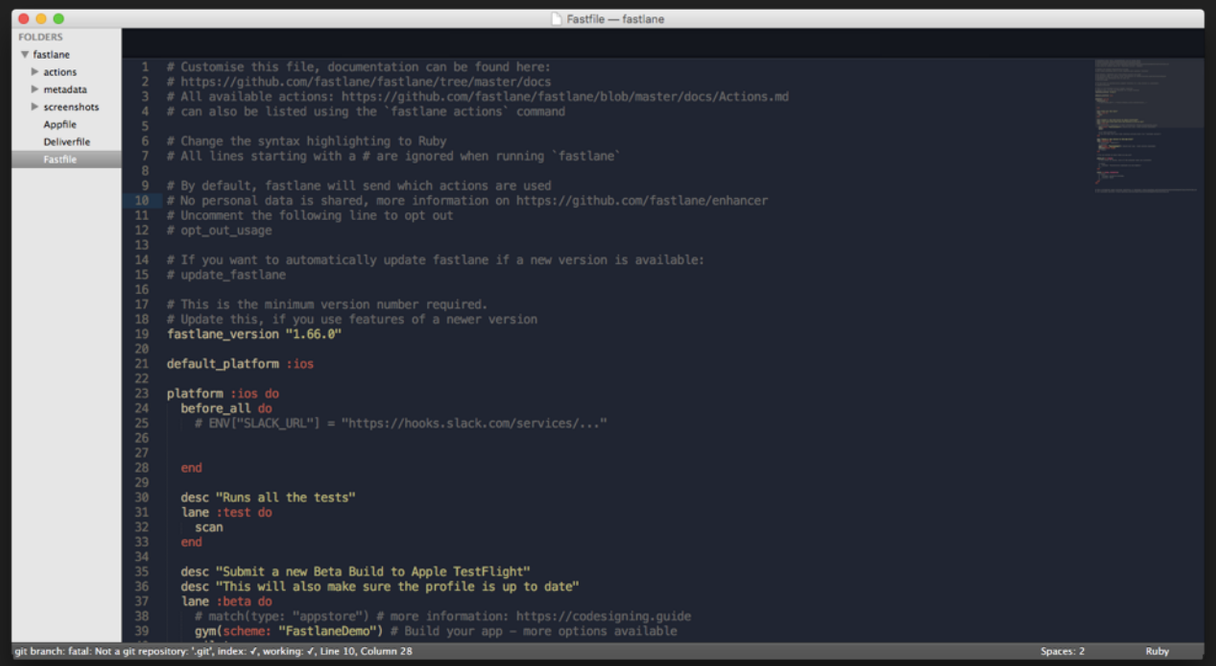
GETTING STARTED RECAP

- `$ sudo gem install fastlane`
- `$ fastlane init`
- Take a look at ./fastlane
- No step 4!

This is how it all ties together.

We’ll start by talking about the first part.

FASTFILE



Let’s go back to the Fastfile.
It’s in Ruby and this is how it looks in it’s stock form.
Your Fastfile is a blank slate where you can define what you want to do.

```
fastlane_version "1.66.0"

default_platform :ios

platform :ios do
  before_all do
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

This is the structure of what the file looks like.
One note: I’m no Ruby pro for those of you confused the colon is a token string (or a symbol) which is like a constant string.

```
fastlane_version "1.66.0"

default_platform :ios

platform :ios do
  before_all do
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

Start with the Version

```
fastlane_version "1.66.0"

default_platform :ios

platform :ios do
  before_all do
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

You define what default platform you’re on, iOS or Mac

```
fastlane_version "1.66.0"

default_platform :ios

platform :ios do
  before_all do
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

You have a path for each platform.
In here, you define your lanes.

```
fastlane_version "1.66.0"

default_platform :ios

platform :ios do
  before_all do
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

Much like Unit Testing, you have a method that gets called before each lane, after each lane.
Method for when an error is reached and it’s passed in as an argument.

```
fastlane_version "1.66.0"

default_platform :ios

platform :ios do
  before_all do
   ENV[“SLACK_URL”] = “http://…”
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

And this is Ruby so there are Environmental variables that are passed around.

```
fastlane_version "1.66.0"

default_platform :ios

platform :ios do
  before_all do
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

And here is your “lane” which could be thought of as “jobs” or “tasks”.

Don’t get scared here, we’re going to write some example lanes in a minute but think of these as your commands for Fastlane.

You define, write, and fill these out.

A lane is a method on the object Fastlane. These can be public (seen here) or private, which we’ll get to.

```
$ fastlane your_custom_lane
```

After you establish that, you can start by running Fastlane from the CL using
`fastlane`
and then the name of your lane

```
fastlane_version "1.66.0"

default_platform :ios

platform :ios do
  before_all do
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
    puts “What goes here?”
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

What actually goes in those methods?

MY WORKFLOW

- Come up with a task I want to automate
  - “I need to get a beta build to Testflight.”
- Research the tools to see what can accomplish that task.
  - “build” = Gym
  - “Testflight” = Pilot
- That task becomes a lane where I use Gym and Pilot to accomplish the goal.

```
default_platform :ios

platform :ios do
  before_all do
  end

  desc "This is your custom lane that you write."
  lane :your_custom_lane do
    gym
    pilot
  end

  after_all do |lane|
  end

  error do |lane, exception|
  end
end
```

What actually goes in those methods?

```
$ fastlane your_custom_lane
```

After you establish that, you can start by running Fastlane from the CL using
`fastlane`
and then the name of your lane

TOOLS AND ACTIONS

You can run straight Ruby in your lane if you wanted to but generally, you'll make use of Tools and Actions.

TOOLS

Let’s start with the bigger piece…

TOOLS

- Scan (Testing)
- Deliver (iTunes Connect submission)
- Gym (Building)
- Sigh (Provisioning Profiles)
- Cert (Certificates)
- etc…

These are just some of the tools you can use from the command line or within your fastfile.
The full list on at fastlane.tools

TOOLS

- First-class citizens that are their own Ruby Gems
- Run from Command Line or within Fastfile (as an action)
- Structured similarly to Fastlane Core
  - Start with `init`
  - Generates its own *file (e.g., Scanfile, Deliverfile)

That last point I’d like to talk about more

*FILE

- Ruby
- Follows pattern of “<parameter> <value>”
- These are the “default” values that can be overridden with arguments within the Fastfile

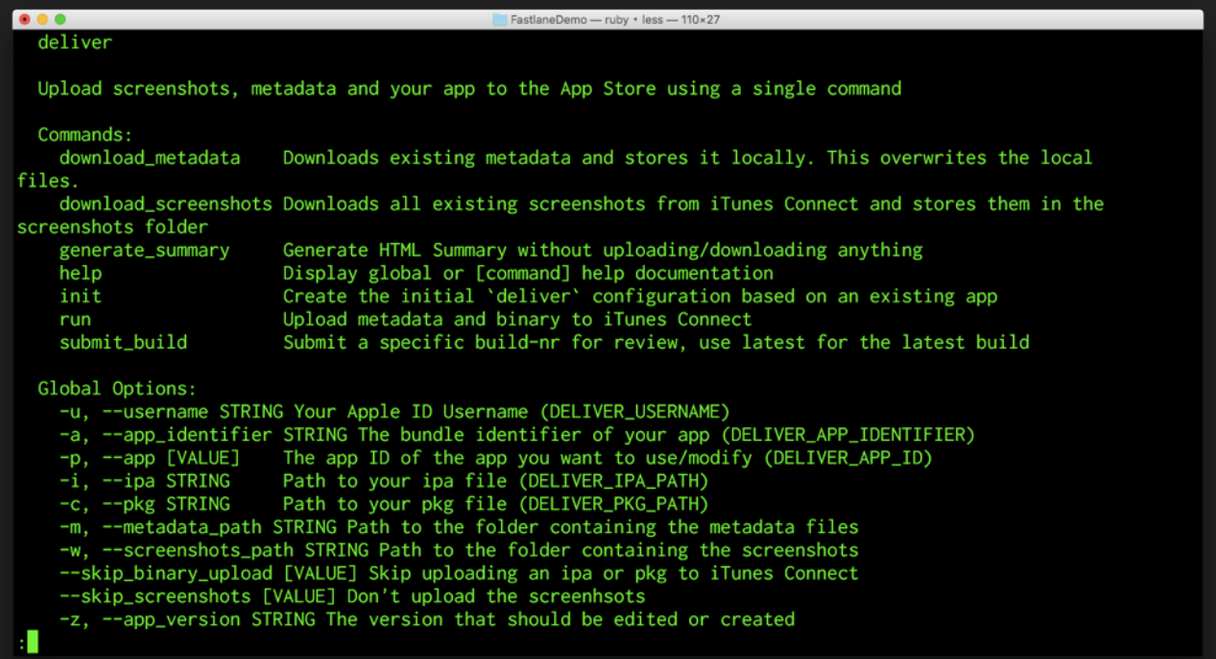
Think of these as values that will never change.
You always want to test on an iPad Air or the bundle identifier will never change.

DELIVERFILE

```
#Deliverfile

app_identifier "com.sushiGrass.FastlaneDemo"
username “jacob@sushigrass.com”

submission_information({
  add_id_info_uses_idfa: false,
  export_compliance_uses_encryption: false,
  export_compliance_encryption_updated: false
})
```

For instance, this is the automatically generated Deliverfile
In this case, it has the key and value for app_identifier and username
These are the default values are always used
Can be overridden in command line or Fastfile as arguments

$ deliver -h



See the options for tools:
use the -h flag on the command line and use the long form

EXAMPLES OF STANDALONE USES

- `$ deliver download_metadata`
- `$ sigh download_all`
- `$ sigh repair`

Find these commands using the -h flag from the command line.

That last point I’d like to talk about more

ACTIONS

The other part are Actions

ACTIONS

- sh (run shell commands)
- clear_derived_data
- cocoapods
- slack
- hockey
- git_pull
- full list at: https://github.com/fastlane/fastlane/blob/master/fastlane/docs/Actions.md

ACTIONS

- Less fully-featured than Tools
- No *file generated so it's all arguments
- Use within Fastfile
- Command Line
  - `$ fastlane run <action_name>`
  - Doesn't allow for all types of arguments
- Run `$ fastlane action` to get a list of actions.
- Run `$ fastlane action` & the name of the action to see options.

$ fastlane action push_to_git_remote

FastlaneDemo — -bash — 112×27

```
+-----------------+-------------------------------------------------------------------+--------------------------
+
|                                        push_to_git_remote
|
+-----------------+-------------------------------------------------------------------+--------------------------
+
| Key             | Description                                                       | Env Var
|
+-----------------+-------------------------------------------------------------------+--------------------------
+
|   local_branch  | The local branch to push from. Defaults to the current branch     | FL_GIT_PUSH_LOCAL_BRANCH
|
|   remote_branch | The remote branch to push to. Defaults to the local branch        | FL_GIT_PUSH_REMOTE_BRANCH
|
| * force         | Force push to remote. Defaults to false                           | FL_PUSH_GIT_FORCE
|
| * remote        | The remote to push to. Defaults to `origin` (default: 'origin')   | FL_GIT_PUSH_REMOTE
|
+-----------------+-------------------------------------------------------------------+--------------------------
+
2 of the available parameters are required
They are marked with an asterisk *

More information can be found on https://github.com/fastlane/fastlane/blob/master/docs/Actions.md

FastlaneDemo ☕
```

See the options for actions
Arguments, what’s required, return values, etc…

ACTIONS (DIY)

Time permitting we'll cover this but it's as simple as:

- `fastlane new_action <your_action_name_here>`

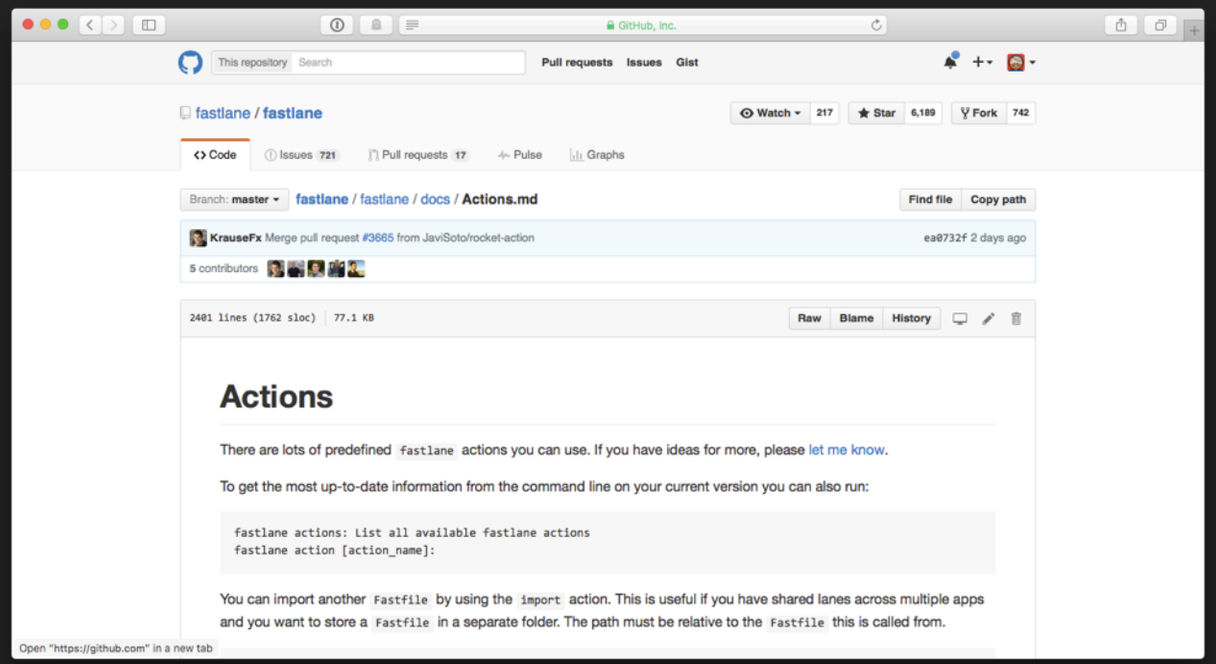
Think of these as values that will never change.
You always want to test on an iPad Air or the bundle identifier will never change.

ACTIONS AND TOOLS

- Tools are also Actions
- Think of this as a subset of functionality
- Find out parameters to pass to these ToolActions using
  - `fastlane action sigh`

Think of these as values that will never change.
You always want to test on an iPad Air or the bundle identifier will never change.

DOCUMENTATION!

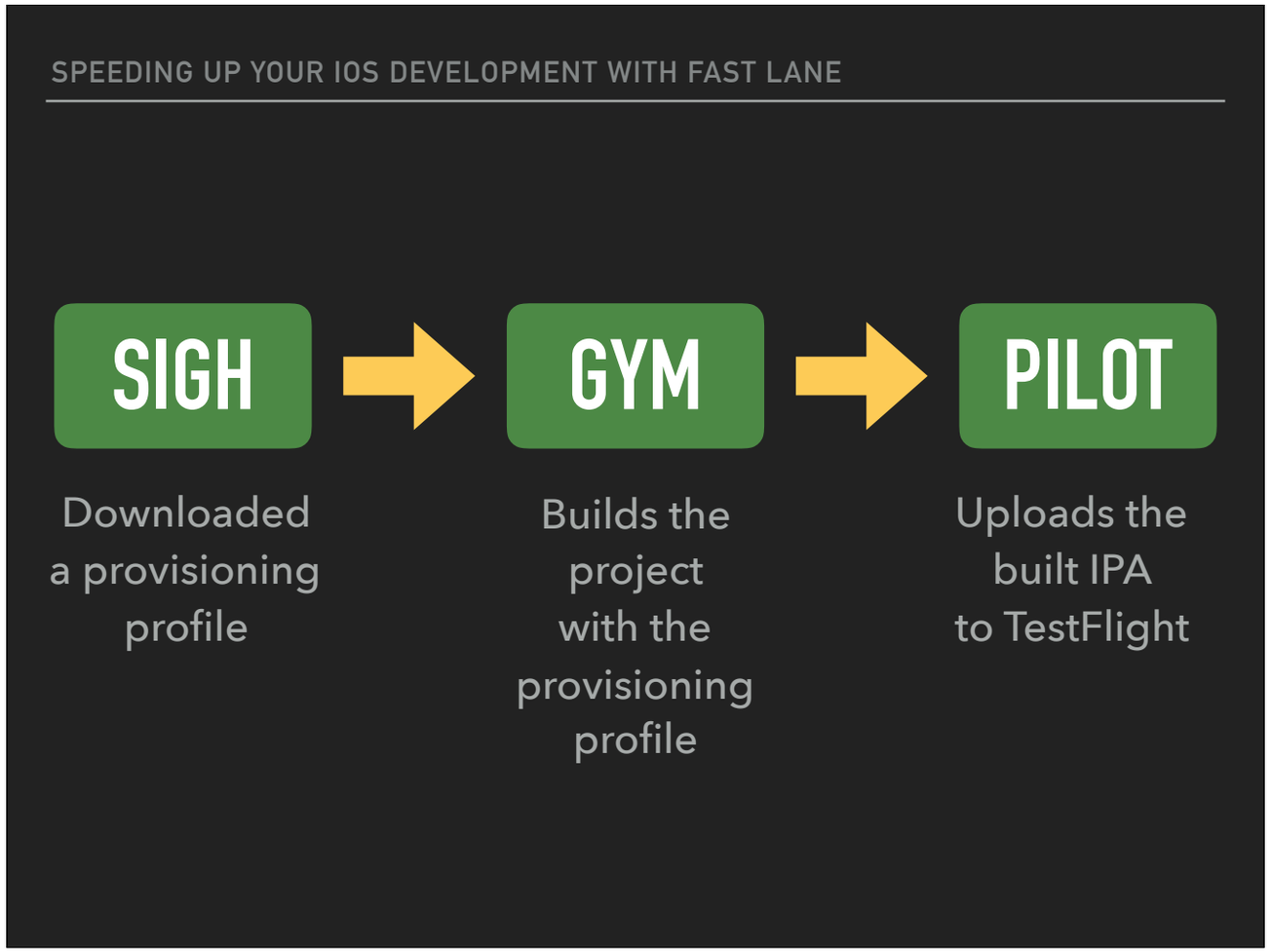


https://github.com/fastlane/fastlane/blob/master/fastlane/docs/Actions.md

Let’s go back to the Fastfile.
It’s in Ruby and this is how it looks in it’s stock form.

BUT HERE'S THE BIG DEAL

These abilities might be really wonderful on their own.
They are tedious tasks that we all face and wish we could have a robot do.
But that's not the real beauty of Fastlane.



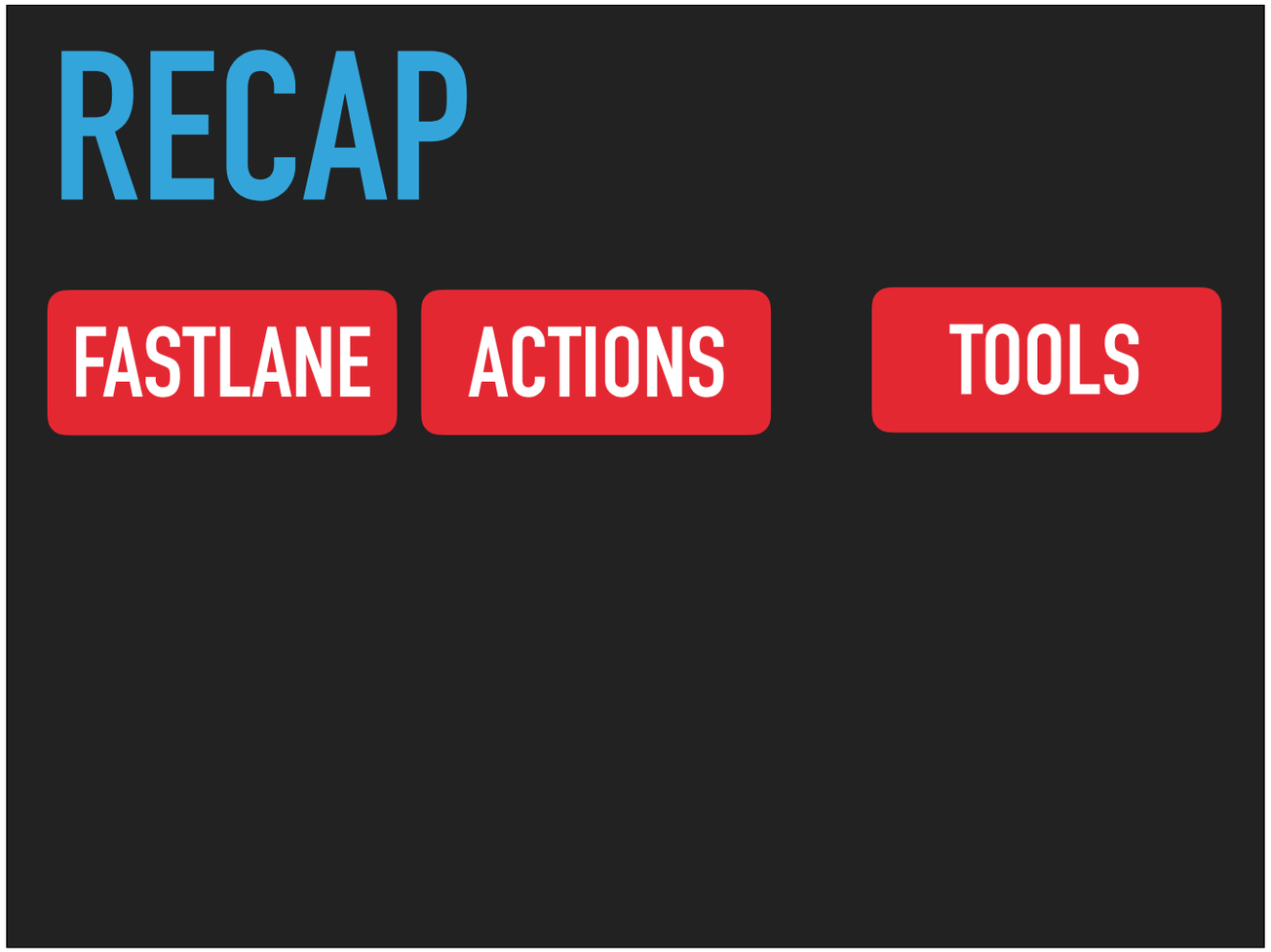
Fastlane allows for all of these tools to be seamlessly combined with each other.

Use Sigh to download a provisioning profile.

Use Gym to build the project with that provisioning profile.

Upload the artifacts from that build to TestFlight.

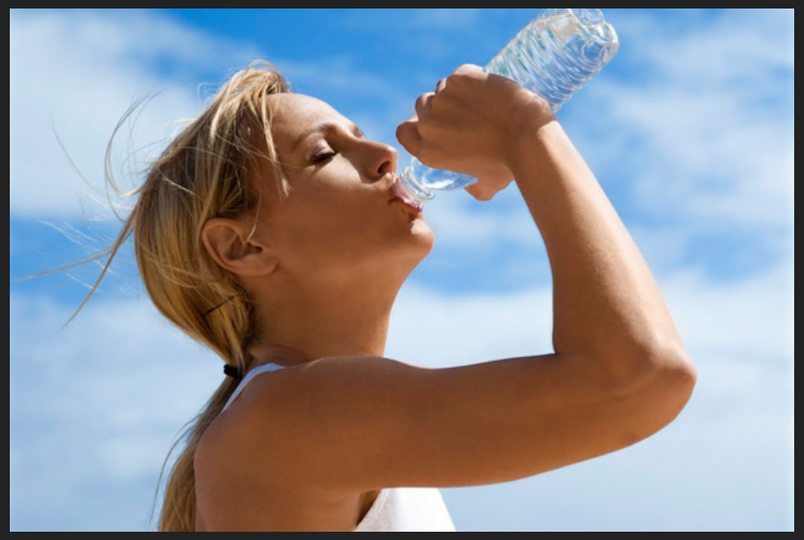
You don't need to pass parameters from one tool to the next because it's all integrated together.



In recap…

Here’s the gist of those three elements.

WHEW.



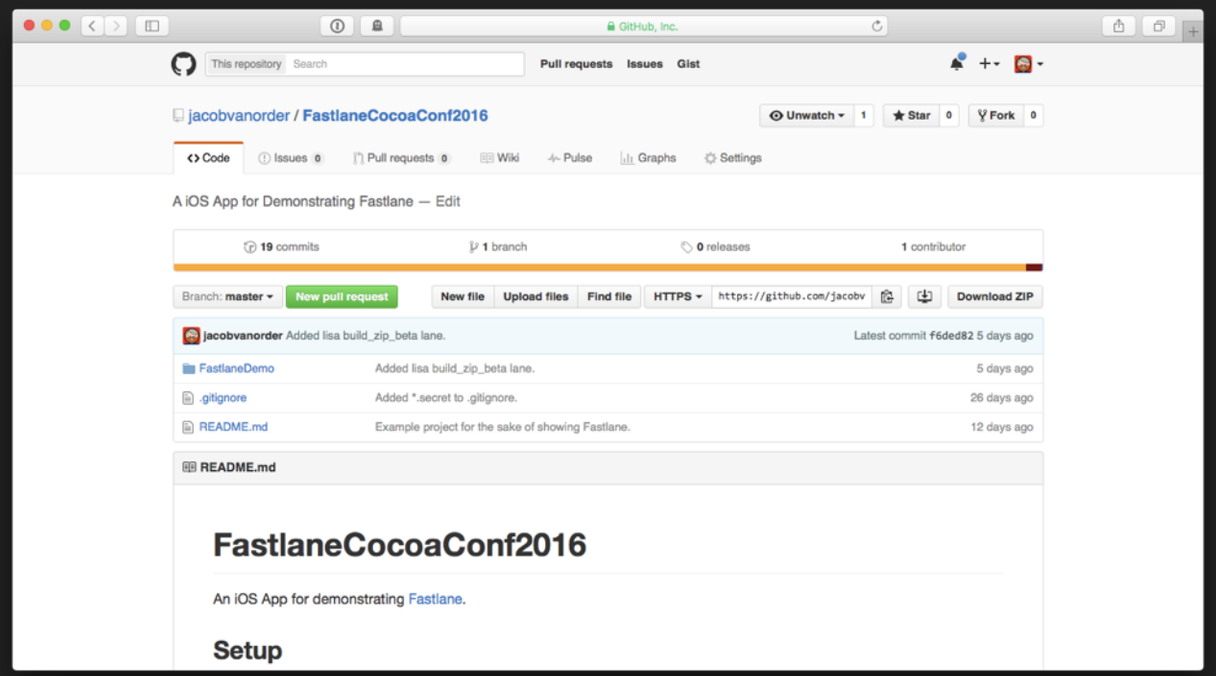
http://claimyourjourney.com/2012/06/run-and-hydrate-louisiana/

This is a reminder for me to drink water.

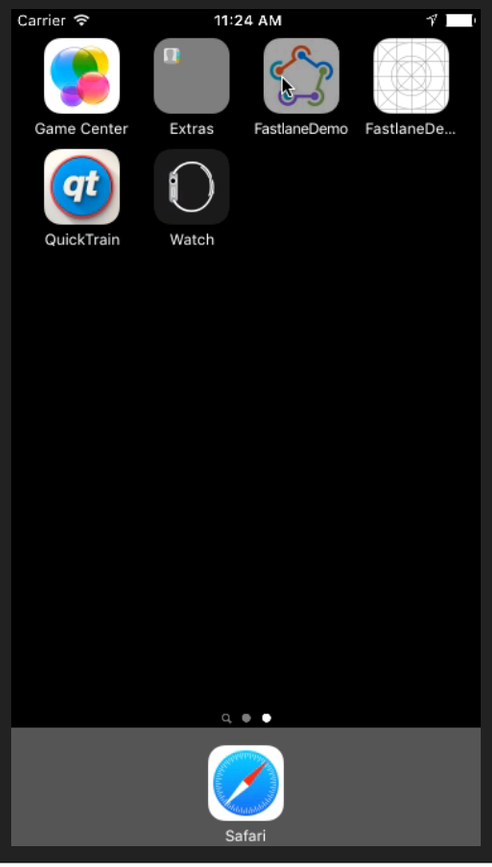
DEVELOPING FOR IOS CAN BE HARD

Let’s get back to the original issue…

https://github.com/jacobvanorder/FastlaneCocoaConf2016



I made an app for this talk. You can get it on Github.



Utilizes the Flickr Api to get a list of camera brands.
Drill down deeper to get to the cameras
Select the camera to get brand, name, and photo
It has Unit and UI Tests written around it

In 2008, that app would have made you $400,000.

TALE OF TWO IOS DEVELOPERS

LISA

DARIUS

Let’s talk about two fictional iOS Developers:

- Lisa
- Darius

Both have their own needs that can be assisted by Fastlane

TALE OF TWO IOS DEVELOPERS

- Independent iOS Developer
- Works with clients
- Does QA by herself
- Needs to archive everything

LISA

Actually @brynn

TALE OF TWO IOS DEVELOPERS

- Works at a large corporation
- Has 7 team members on his team
- Includes QA members on team
- Needs to have repository of symbolication files

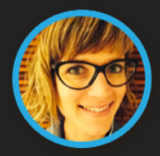
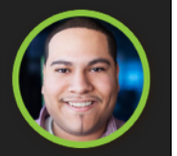
DARIUS

Actually @angelceballos

ISSUE #1: PROVISIONING PROFILES

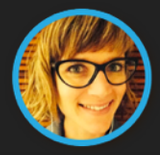
Yes, this is Issue #1 for all of us.

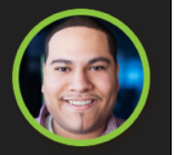
LISA AND DARIUS NEED TO MANAGE THEIR PROVISIONING PROFILES

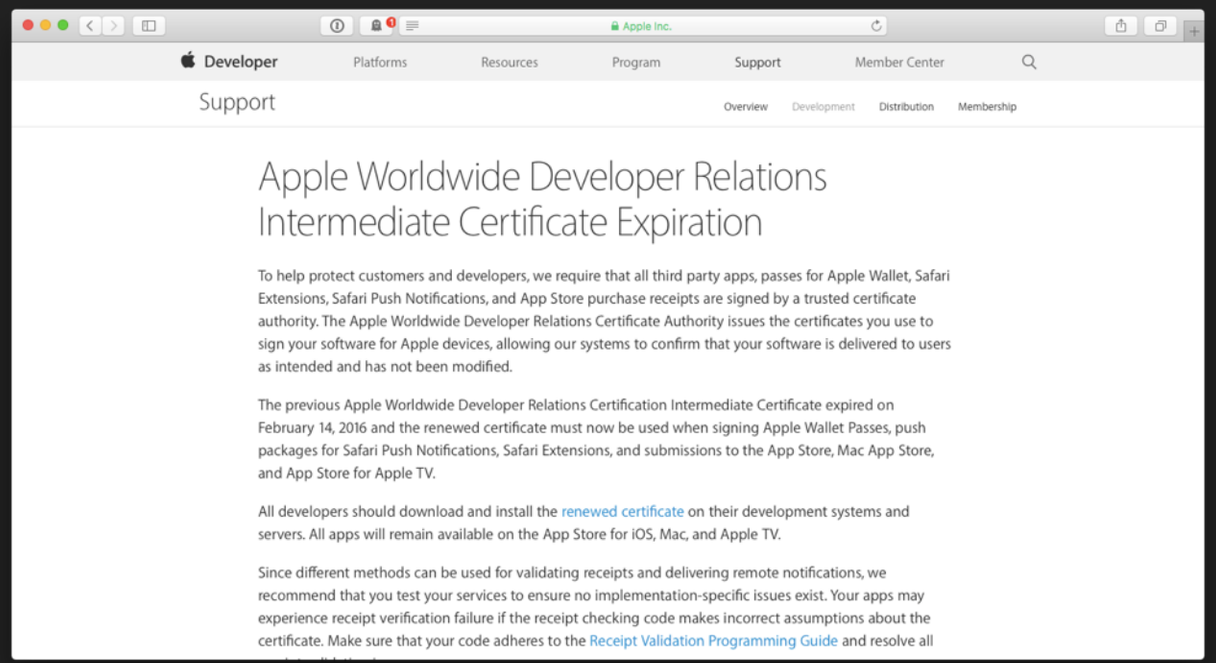



- Provisioning Profiles are the worst.
- Especially Ad-Hoc
- Use Sigh to automatically update them.

UGH



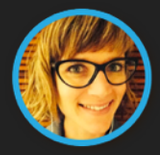


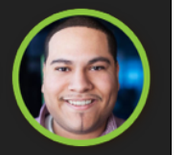


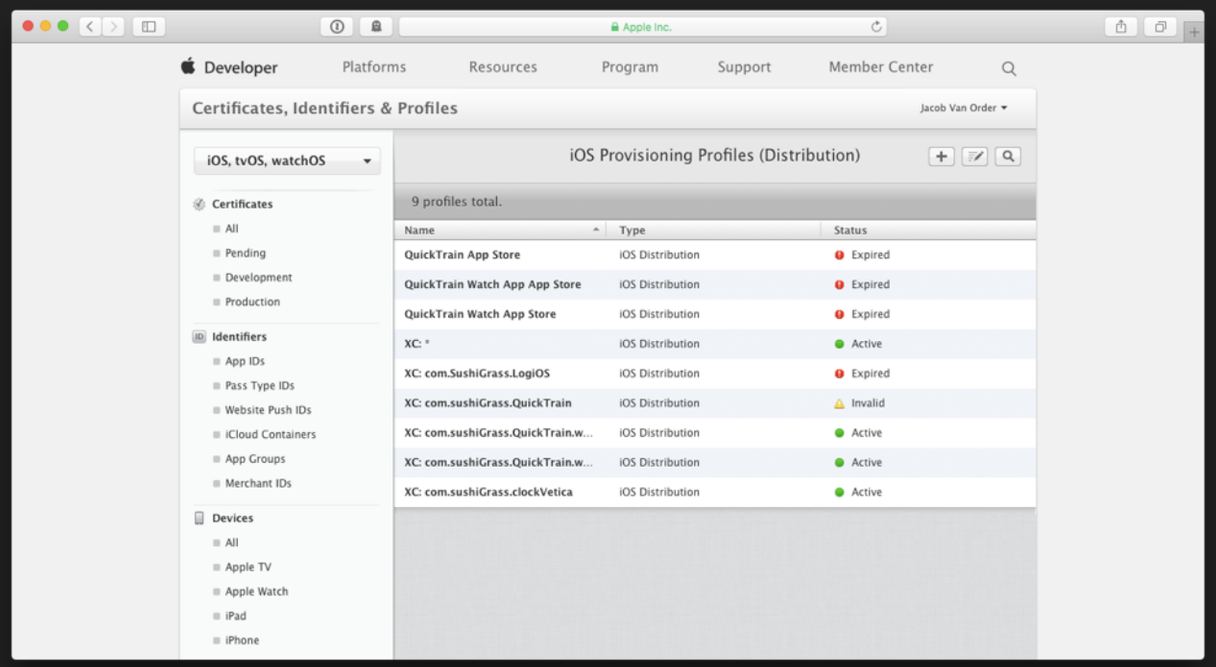
We all had to deal with this.

The WWDR cert expired and builds weren’t building.

UGH

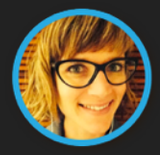


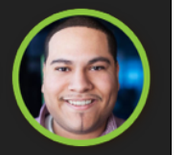




The result is that some of your distribution profiles are no longer valid.

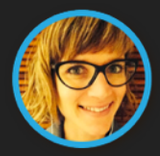
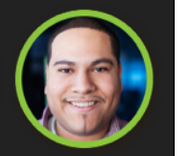
UGH



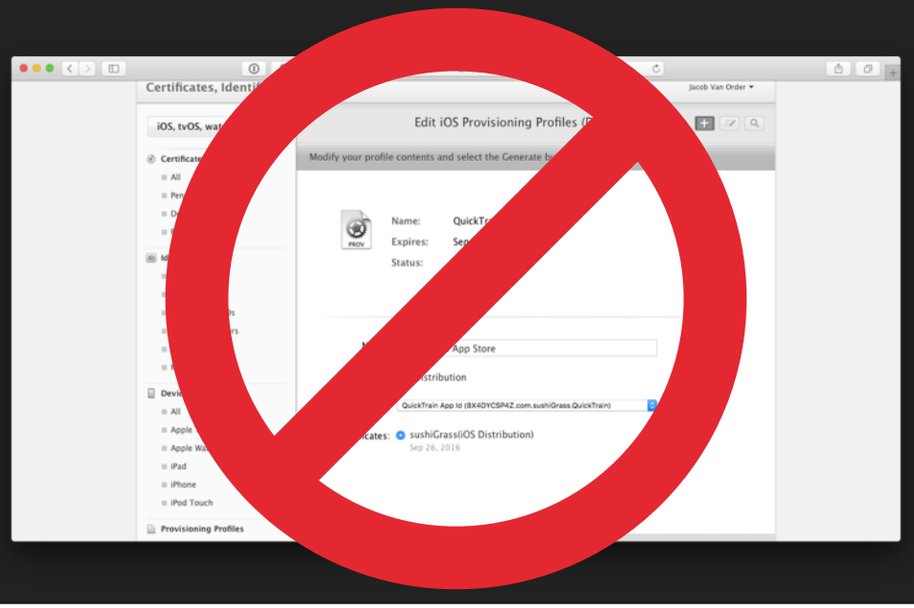


They updated Fastlane to automatically download the new WWDR cert which will be great in 2024.

SIGH TO THE RESCUE

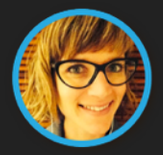
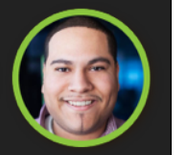



```
$ sigh repair
```

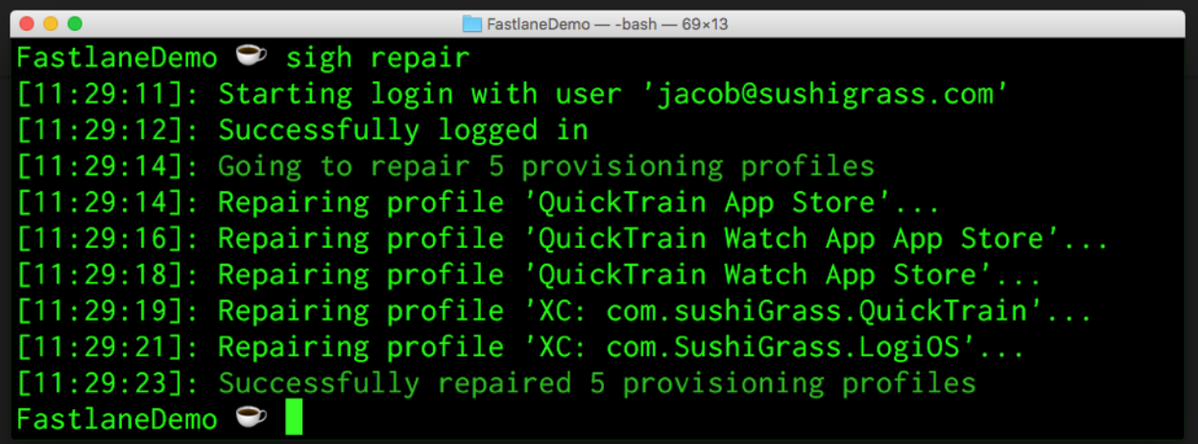


Running this command saves you having to go to each profile’s page and update it.

SIGH TO THE RESCUE

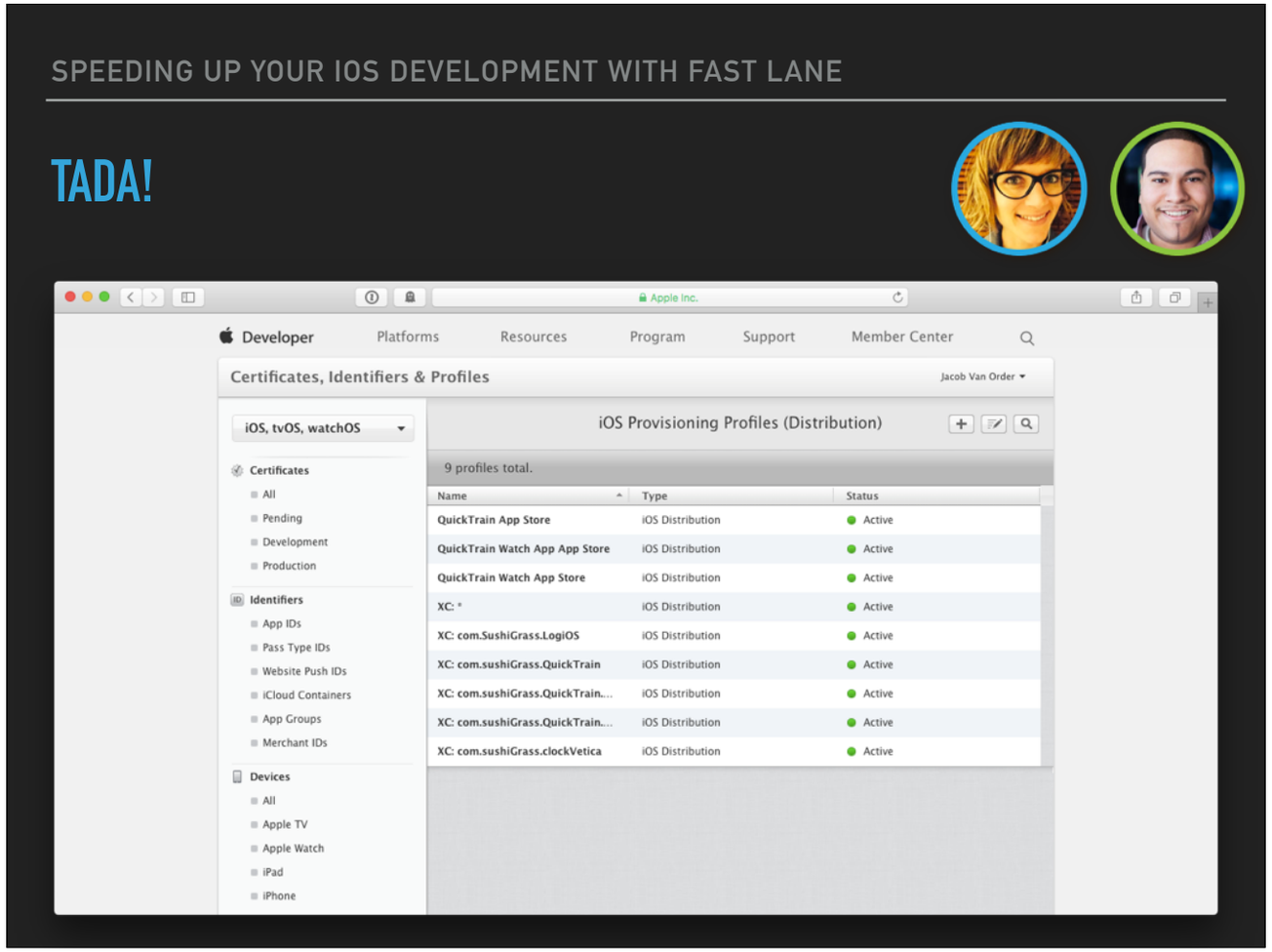



```
$ sigh repair
```



Instead it does it automatically and easily.

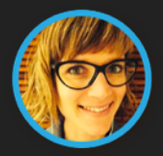
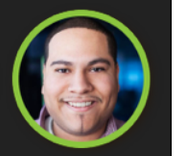
This is an example of using sigh as a tool with a command.



And now it’s fixed!

But what about your build servers? Your machine?

AUTOMATICALLY UPDATED

```
before_all do |lane, options|
  sigh
end
```

Each time a lane runs, sigh is being called.
This is an example of it being used as an action within the Fastfile.

Now, before any build, Fastlane will detect the appropriate provisioning profile, download, and install it.

Any subsequent uses of Gym will automatically use it to build.

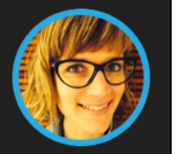
No more updating all your CI machines.


MAGIC

ISSUE #2: REMEMBERING TO DO YOUR UNIT TESTING

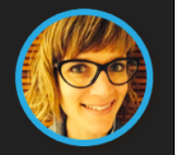
Unit Tests are great but what if you’re not going to run them?

LISA NEEDS TO REMEMBER TO TEST



- Would like it if Unit Tests were ran and succeeded before being able to push
- Creates a lane called test_and_push
- Runs Scan
- If successful, uses push_to_git_remote

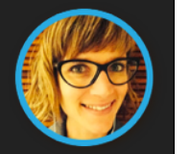
HOW THIS MIGHT LOOK



```
desc "Test and then push to git remote"
lane :test_and_push do
  scan
end

after_all do |lane|
  if lane == :test_and_push
    push_to_git_remote
  end
end
```

This is the lane you might want to run.

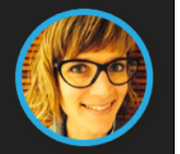
HOW THIS MIGHT LOOK



```
desc "Test and then push to git remote"
lane :test_and_push do
  scan
end

after_all do |lane|
  if lane == :test_and_push
    push_to_git_remote
  end
end
```

First is runs scan…

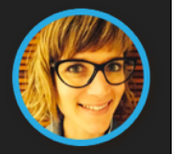
HOW THIS MIGHT LOOK

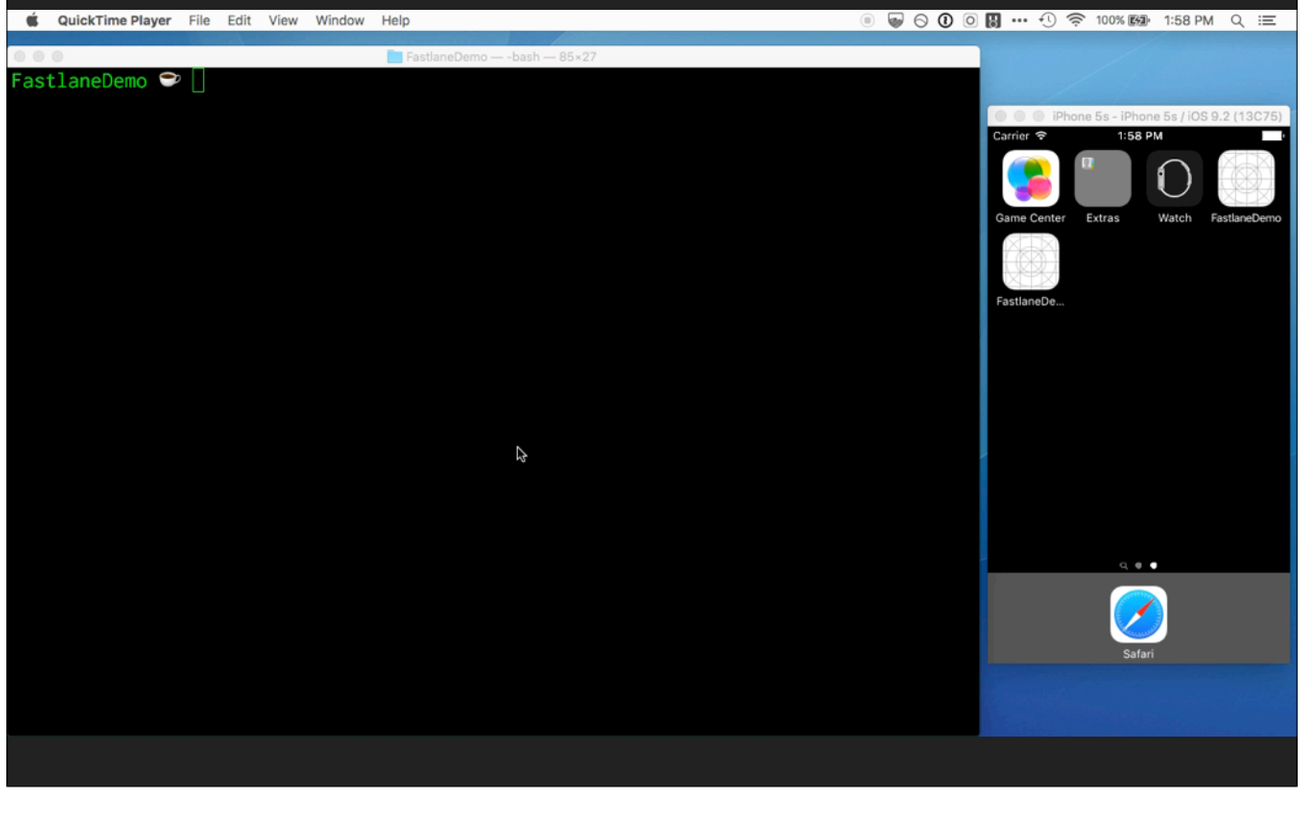


```
desc "Test and then push to git remote"
lane :test_and_push do
  scan
end

after_all do |lane|
  if lane == :test_and_push
    push_to_git_remote
  end
end
```

And, if successful, it will push to git remote.

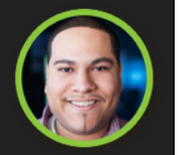
LISA TEST_AND_PUSH





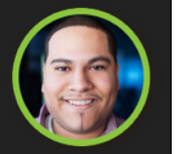
This is how it would look

DARIUS (AND HIS TEAM) NEEDS TO REMEMBER TO TEST



- Would like it if Unit Tests were continuously ran and notified team if something failed
- Creates a lane called test
- Runs Scan
- If it fails, it notifies Slack

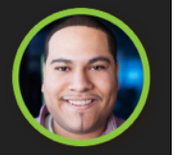
HOW THIS MIGHT LOOK



```
before_all do
  ENV["SLACK_URL"] = “https://hooks.slack.com/…"
 clear_derived_data
 ensure_git_status_clean
end

desc "Continuously tests"
lane :test do
   scan(slack_only_on_failure: true,
        slack_channel: '#ios')
end
```

This is the lane you might want to run.

HOW THIS MIGHT LOOK

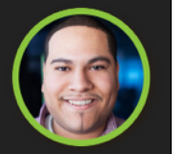


```
before_all do
  ENV["SLACK_URL"] = “https://hooks.slack.com/…"
 clear_derived_data
 ensure_git_status_clean
end

desc "Continuously tests"
lane :test do
   scan(slack_only_on_failure: true,
        slack_channel: '#ios')
end
```

In the before_all method, you set the slack url using the environmental variable.

- clear derived data
- ensure clean git status

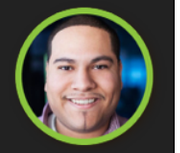
HOW THIS MIGHT LOOK



```
before_all do
  ENV["SLACK_URL"] = “https://hooks.slack.com/…"
 clear_derived_data
 ensure_git_status_clean
end

desc "Continuously tests"
lane :test do
   scan(slack_only_on_failure: true,
        slack_channel: '#ios')
end
```

I used this as an example of arguments but these would be put in the Scanfile if they were to remain constant.

DARIUS TEST



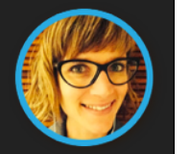
from: https://gigaom.com/2010/06/15/the-new-mac-mini-the-next-apple-tv/

What he sees is… nothing because he’s running on a CI server somewhere

ISSUE #3: BUILDING .IPA, .DSYM , .XCARCHIVE FILES FOR YOURSELF OR QA

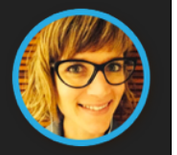
Keeping track of your builds can be a little confusing and weird.
.dsym files are needed for crash log desymbolication.
.xcarchives are needed for the recent addition of crash reporting in Xcode.

LISA NEEDS TO BUILD IPA FILES



- Would like it if builds for beta testing could automatically be built, archived, and submitted to TestFlight.
- Creates a lane called build_zip_beta
- Gym for building
- Shell command for zipping archives
- Pilot action for submitting to TestFlight

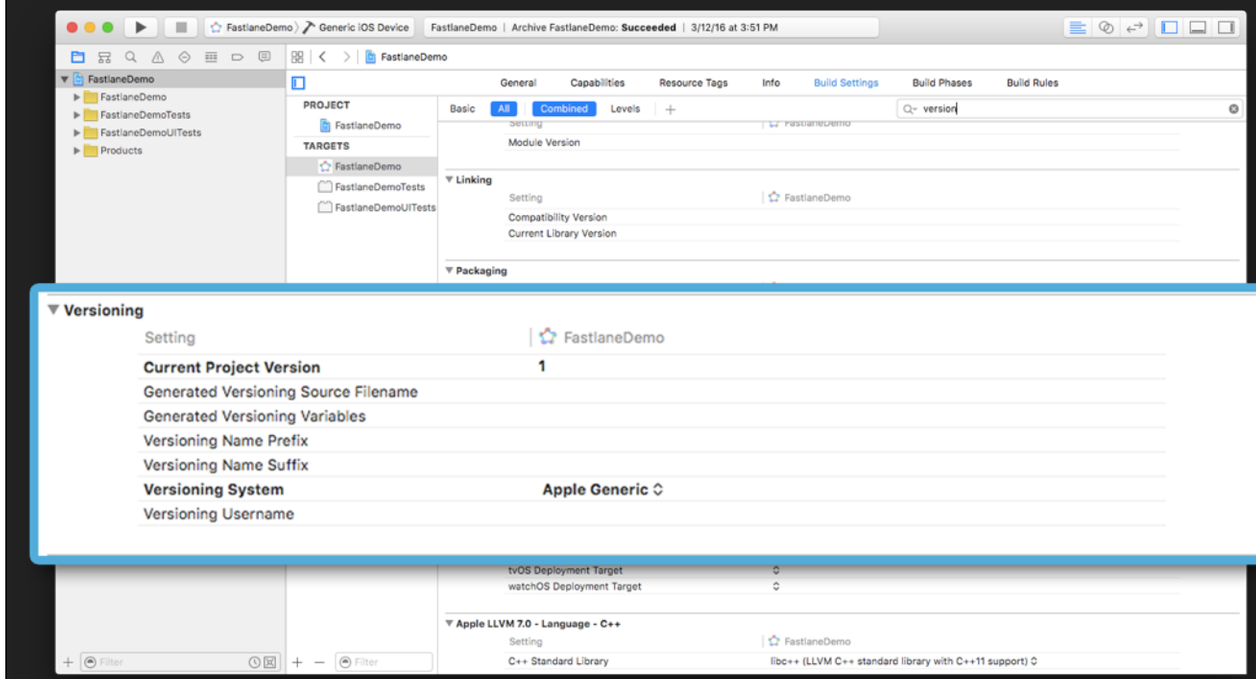
HOW THIS MIGHT LOOK (1/4)



```
lane :build_zip_beta do
  # Get the build number
  begin
    build_number =
latest_testflight_build_number(version:
get_version_number)
  rescue Exception => e
    build_number = 0 #Doesn't exist, set to zero
  end
  build_number = build_number += 1
  increment_build_number(build_number: build_number)
    …
end
```

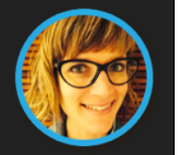
We have to split the lane into 4 parts in order to walk through how we do this.

First part is getting a build number from TestFlight.
Wrap this in a ruby exception handler as there might not be a previous build already.
We increment the number
Use the increment build number action to set the project's build number



One thing of note:
For you to use that feature, you have to enable Apple Generic Versioning System
Set the project version to 0.
The action will keep this number in line with every CFBundleVersion in your project

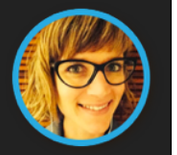
HOW THIS MIGHT LOOK (2/4)



```
lane :build_zip_beta do
  …
  build_name = “FastlaneDemo_#{get_version_number
_build_#{get_build_number}”

  # Make the folder for the artifacts
  artifacts_path = '../artifacts/' #Note!
  if !Dir.exist?(artifacts_path)
    Dir.mkdir(artifacts_path)
  end
  …
end
```

For the second part, we generate a build name using the get version number and get build number actions
The next step is that we make a folder for the artifacts with Ruby.

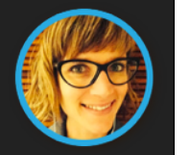
HOW THIS MIGHT LOOK (3/4)



```
lane :build_zip_beta do
  …
    # Build
   gym(clean: true,
      output_directory: './artifacts/',
      archive_path: './artifacts/' + build_name,
      output_name: build_name)
    # Zip up the artifacts
   sh("zip -r ~/Desktop/#{build_name}.zip
#{artifacts_path}")
  …
end
```

For the third section, we utilize Gym.
Everything there is pretty self explanatory but notice only one period in the artifacts directory.
By setting an archive path, it will export it there instead of the default location.
After that, we zip up the artifacts.

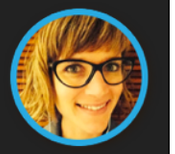
HOW THIS MIGHT LOOK (4/4)

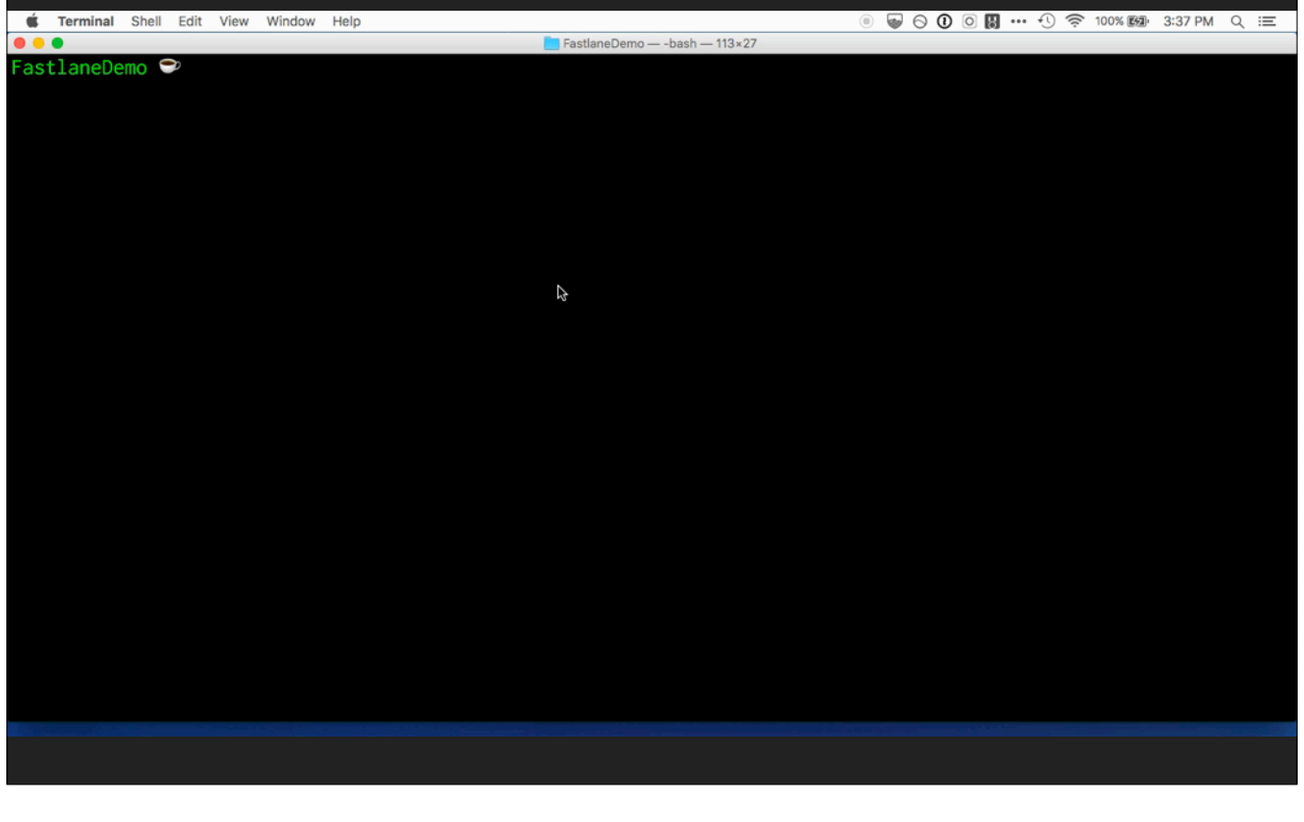


```
lane :build_zip_beta do
  …
    pilot(skip_submission: true)
end
```

Lastly, we use Pilot to submit to Testflight.
That skip submission flag has it so that Fastlane doesn’t wait for the IPA to finish processing on iTunes Connect which takes about forever.

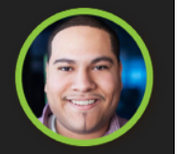
LISA BUILD_ZIP_BETA





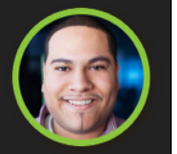
This is how it would look

DARIUS NEEDS TO AUTOMATICALLY GENERATE ARCHIVES



- Would like it if artifacts could be generated for QA to use for testing and be notified by Slack
- Creates a lane called continuously_build
- Gym for building
- The CI bot should automatically pick up the artifacts
- Slack notification

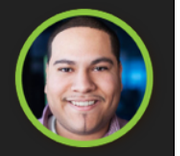
HOW THIS MIGHT LOOK (1/3)



```
desc “Continuously Build”
lane :continuously_build do
  ci_build_number = ENV[’your_ci_build_number’]
     build_name =
  “Fastlane_Demo_v.#{get_version_number}
  _b._#{ci_build_number}”
  increment_build_number(build_number: ci_build_number)
  …
end
```

The first part we’ll get the build number from the CI server. In this case, the CI server passes in as an environmental variable what the build number is. We use that to generate the build name.

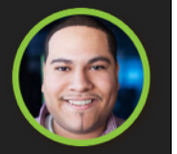
HOW THIS MIGHT LOOK (2/3)



```
desc “Continuously Build”
lane :continuously_build do
  …
     # Make the folder for the artifacts
     artifacts_path = '../artifacts/' #Note!
     if !Dir.exist?(artifacts_path)
        Dir.mkdir(artifacts_path)
     end
  …
end
```

Just like before, we make the artifacts path.
Our CI server will point to that folder and pick up anything put inside as artifacts that it should store.

HOW THIS MIGHT LOOK (3/3)



```
desc “Continuously Build”
lane :continuously_build do
  …
     # Build
     gym(clean: true,
        output_directory: './artifacts/',
        output_name: build_name)
end

after_all do |lane|
  if lane == :continuously_build
     slack
  end
end
```

And just like Lisa, we use Gym to build and store those artifacts.
The difference is that we don’t need the archive each time a change is made.

Finally, if we are successful and the lane is continuously build, we send a slack message.

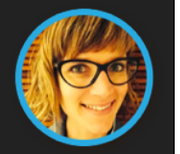
Again, we don’t need to see what it looks like. It’s on a CI.

ISSUE #4: SHIPPING THE APP IS TEDIOUS

Shipping is hard!

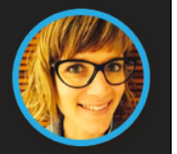
Screenshots, metadata, iTunes Connect! It’s all a huge amount of overhead.

LISA NEEDS TO SHIP THE APP



- Would like it if screenshots were automatically generated. She will use the Gym command to build and get artifacts but just submit that .xcarchive from Xcode.
- Use Snapshot to automatically generate her screenshots
- Use Deliver to automatically upload them to iTunes Connect
- Gym command from before but moved into a private lane.

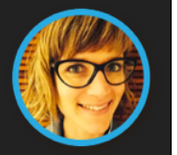
HOW THIS MIGHT LOOK (1/4)



```
desc "Take screenshots of the app and ship it!"
lane :snapshot_and_ship do
  snapshot(skip_open_summary: true)
  deliver
  #private_lane
  get_last_testflight_build_number_and_increment
  #private_lane
  build_and_archive
end
```

We utilize snapshot to generate our screenshots.
In order to utilize Snapshot, we have to do some set up.

HOW THIS MIGHT LOOK (1/4)

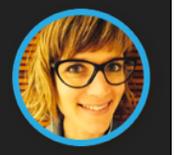


```
$ snapshot init

Open your Xcode project and make sure to do the
following:
1) Add the ./fastlane/SnapshotHelper.swift to your UI
Test target
   You can move the file anywhere you want
2) Call `setupSnapshot(app)` when launching your app

  let app = XCUIApplication()
  setupSnapshot(app)
  app.launch()

3) Add `snapshot("0Launch")` to wherever you want to
create the screenshots
```

We first run snapshot init.

It will install a Snapfile but also include these instructions.
Hopefully, we already have UI Test written so we can in just insert the setup method as well as the method to take snapshots in the desired places.

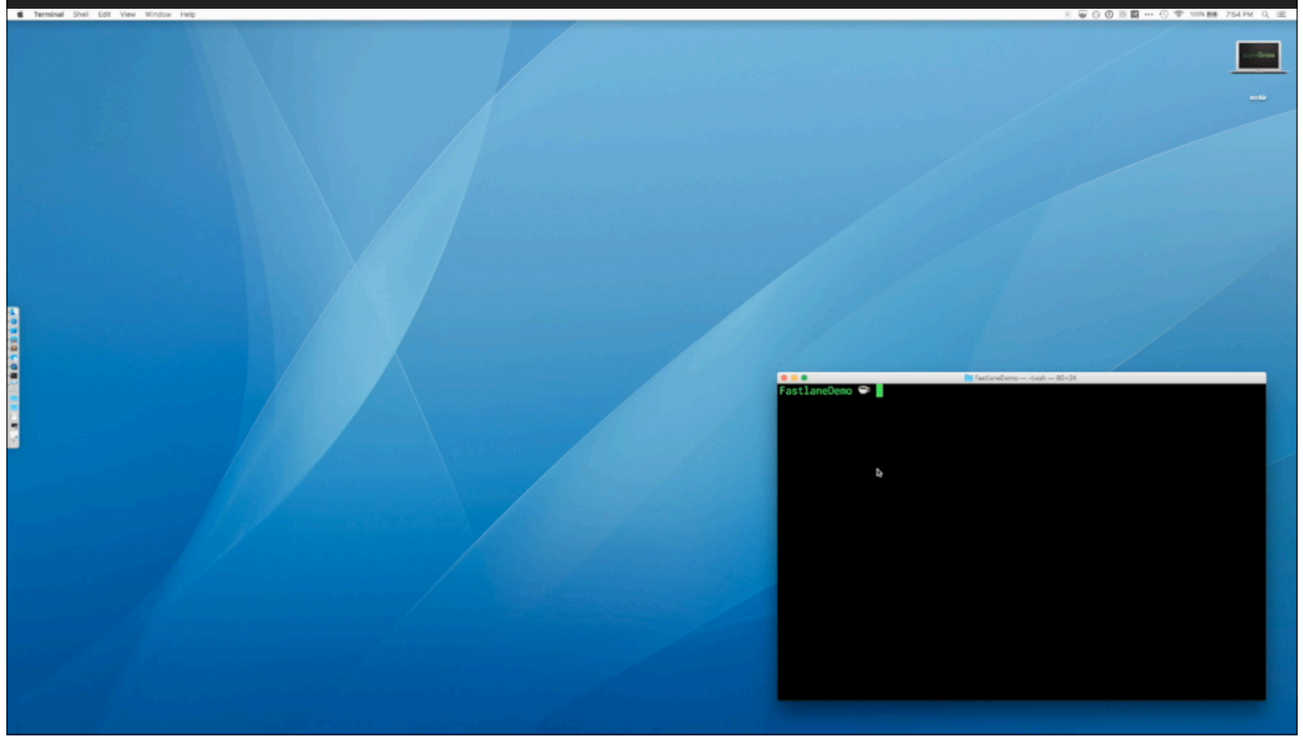
HOW THIS MIGHT LOOK (1/4)



```
# A list of devices you want to take the screenshots from
devices([
  "iPhone 4s",
  "iPhone 5",
  "iPhone 6",
  "iPhone 6 Plus",
  "iPad Pro",
  "iPad Air"
])

languages([
  "en-US",
  "es-ES"
])
```

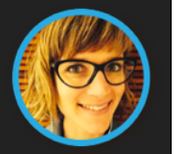
The Snapfile will include which languages and devices you want to generate screenshots for.
In this case, we want the many phone sizes and both iPad sizes.
We also want English and Spanish.

SNAPSHOT IN ACTION



For such a simple command, the complexity is quite impressive.
I run the Snapshot command from the Command Line.
As you can see, it runs through my UI Tests.
Hope you’re comfy because for those 6 devices, this take 20 minutes.
Those calls to take screenshots are being called and they are being saved for each simulator size and language.
Think of all the time you are saving here given even changing the language on the sim takes FOREVER.
At the end of this all, we get a nice overview of what was generated for review.

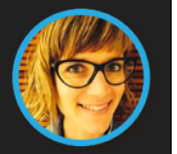
HOW THIS MIGHT LOOK (1/2)



```
desc "Take screenshots of the app and ship it!"
lane :snapshot_and_ship do
  snapshot(skip_open_summary: true)
  deliver
  #private_lane
  get_last_testflight_build_number_and_increment
  #private_lane
  build_and_archive
end
```

Back to our lane…
In the video we showed the preview website but for automation’s sake,
we pass true to skipping the opening of the summary because we just want to keep going.

HOW THIS MIGHT LOOK (2/4)

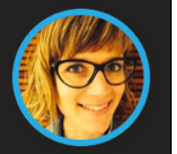


```
desc "Take screenshots of the app and ship it!"
lane :snapshot_and_ship do
  snapshot(skip_open_summary: true
  deliver
  #private_lane
  get_last_testflight_build_number_and_increment
  #private_lane
  build_and_archive
end
```

Next, we use Deliver to upload those screenshots directly to iTunes Connect.

Out of the box, it automatically looks for metadata and screenshots. We will talk about a more advanced usage of Deliver for Darius.

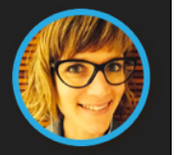
HOW THIS MIGHT LOOK (3/4)



```
desc "Take screenshots of the app and ship it!"
lane :snapshot_and_ship do
  snapshot(skip_open_summary: true
  deliver
  #private_lane
  get_last_testflight_build_number_and_increment
  #private_lane
  build_and_archive
end
```

For the next step, we are going to use the same functionality we had in build, zip, beta.
Rather than have the same code in the same place, we can use a private method in the Fastfile called a private lane.
In this case, we are going to move the code for getting the last TestFlight build and incrementing into a method.

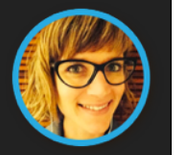
HOW THIS MIGHT LOOK (3/4)



```
private_lane :get_last_testflight_build_number_and_increment do
    # Get the build number
    begin
      build_number = latest_testflight_build_number(version:
get_version_number)
    rescue Exception => e
      build_number = 0 #Doesn't exist, set to zero
    end
    build_number = build_number += 1
    increment_build_number(build_number: build_number)
  end
```

This is the same exact code used before but now it’s moved into a private lane.

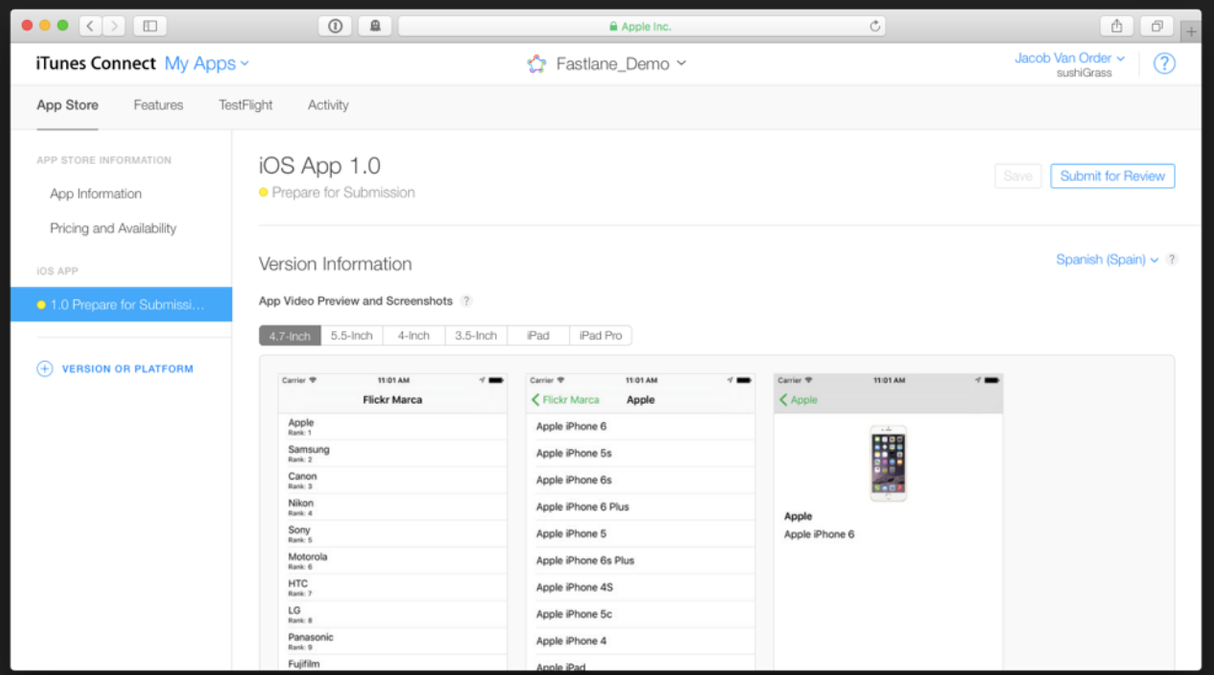
HOW THIS MIGHT LOOK (4/4)



```
desc "Take screenshots of the app and ship it!"
lane :snapshot_and_ship do
  snapshot(skip_open_summary: true
  deliver
  #private_lane
  get_last_testflight_build_number_and_increment
  #private_lane
  build_and_archive
end
```

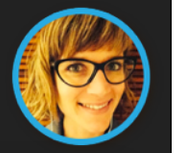
And again, we move the code that created the artifacts directory, built, and zipped into a private lane for reuse.
She'll use that .xcarchive to submit to the store using Xcode.

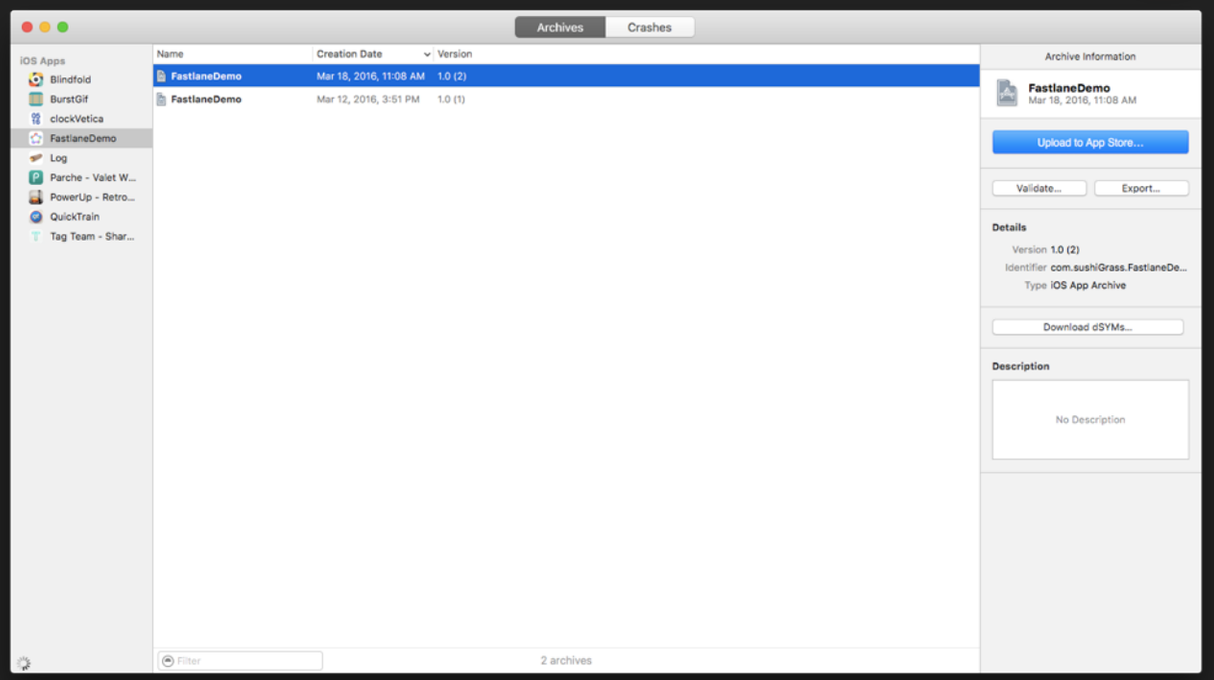
THE RESULT (1/2)



Two things happened:
It uploaded the screenshots that it generated automatically to iTunes Connect.
You just have to make sure you're set up on iTC for those languages.

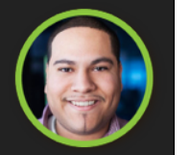
THE RESULT (2/2)





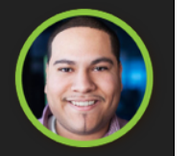
Secondly, she got the artifacts saved to her Desktop.
By going into that, she can open up the .xcarchive and upload that directly using Xcode.

DARIUS NEEDS TO SHIP AUTOMATICALLY



- Would like it if, after merging to Master branch, screenshots, builds, and metadata written by the Product Managers were uploaded. Finally, the commit is tagged and pushed to the repo.
- The same process as Lisa
- Metadata filled out
- Slack notification

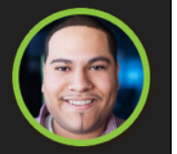
HOW THIS MIGHT LOOK (1/4)



```
desc "Ship the app!"
lane :ship_it do
 snapshot(skip_open_summary: true)
 #private_lane #parent file
 get_last_testflight_build_number_and_increment
 #private lane
 build_and_archive
 deliver(force: true, submit_for_review: true)

 add_git_tag(tag:"v.#{get_version_number}")
 push_git_tags
end
```

This first part is similar to Lisa’s shipping lane.
Snapshot
Get the latest test flight build number and increment
build and archive (but not zip)

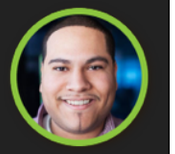
HOW THIS MIGHT LOOK (2/4)

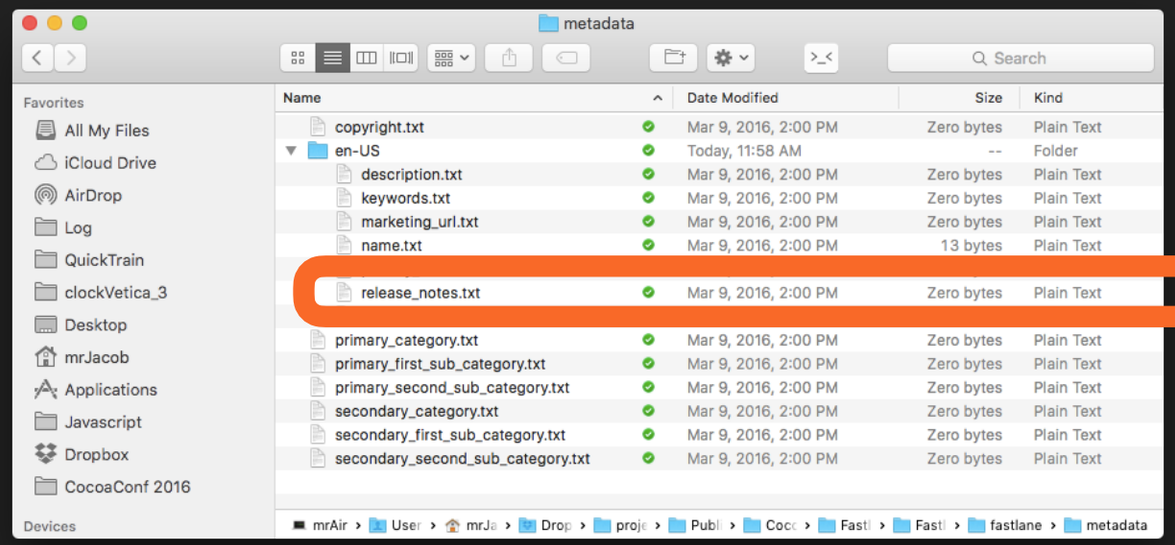


```
desc "Ship the app!"
lane :ship_it do
 snapshot(skip_open_summary: true)
 #private_lane #parent file
 get_last_testflight_build_number_and_increment
 #private lane
 build_and_archive
 deliver(force: true, submit_for_review: true)

 add_git_tag(tag:"v.#{get_version_number}")
 push_git_tags
end
```

What is not the same is that we move the Deliver action AFTER we build.
That way Deliver will pick up the built IPA and push that up to iTunes Connect along with the screenshots and metadata.
About that metadata…

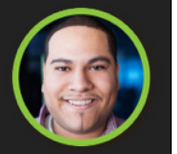
HOW THIS MIGHT LOOK (2/4)





If you initialized Deliver, you'll have a metadata folder generated.
If it's an existing app in the store, it will gather all of the info.
When they are ready to ship, they get all of the information from the Product Manager ready.
They can write what they want in a .txt file, attach it to the ticket for the update, and be good.
Particularly Release Notes if they are doing an update.

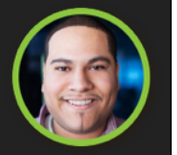
HOW THIS MIGHT LOOK (2/4)



```
desc "Ship the app!"
lane :ship_it do
 snapshot(skip_open_summary: true)
 #private_lane #parent file
 get_last_testflight_build_number_and_increment
 #private lane
 build_and_archive
 deliver(force: true, submit_for_review: true)

 add_git_tag(tag:"v.#{get_version_number}")
 push_git_tags
end
```

Back to the deliver action, adding the submit_for_review flag does just that.

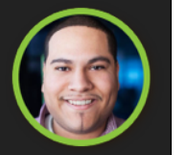
HOW THIS MIGHT LOOK (3/4)



```
desc "Ship the app!"
lane :ship_it do
 snapshot(skip_open_summary: true)
 #private_lane #parent file
 get_last_testflight_build_number_and_increment
 #private lane
 build_and_archive
 deliver(force: true, submit_for_review: true)

 add_git_tag(tag:"v.#{get_version_number}")
 push_git_tags
end
```

Next, we tag the release and push to git.

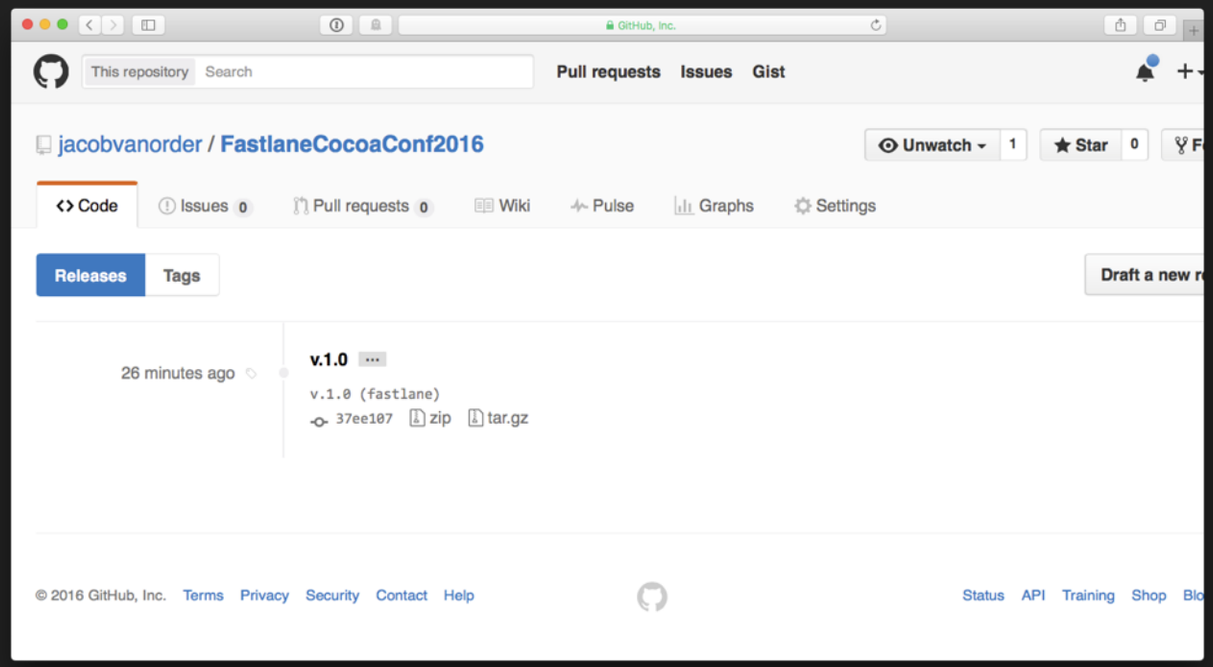
HOW THIS MIGHT LOOK (4/4)



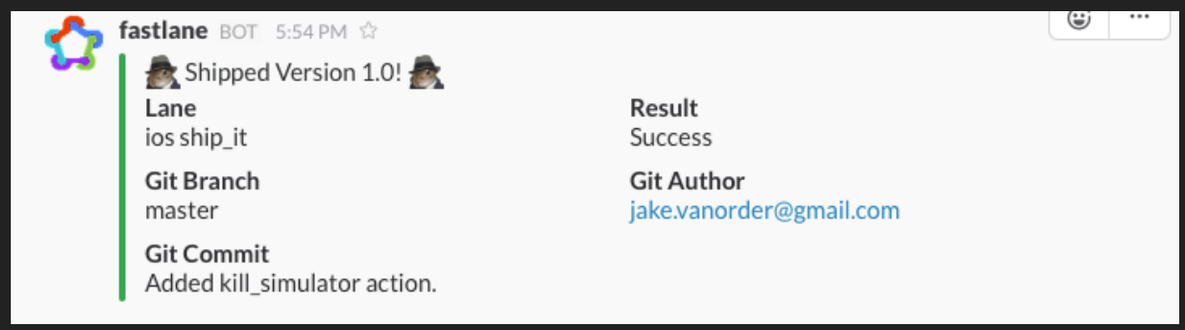
```
after_all do |lane|
  if lane == :continuously_build
    slack(success: true)
  elsif lane == :ship_it
    message = ":shipit: Shipped Version
#{get_version_number}! :shipit:"
    slack(message:message)
  end
end
```

Finally, we celebrate on Slack, complete with emoji.

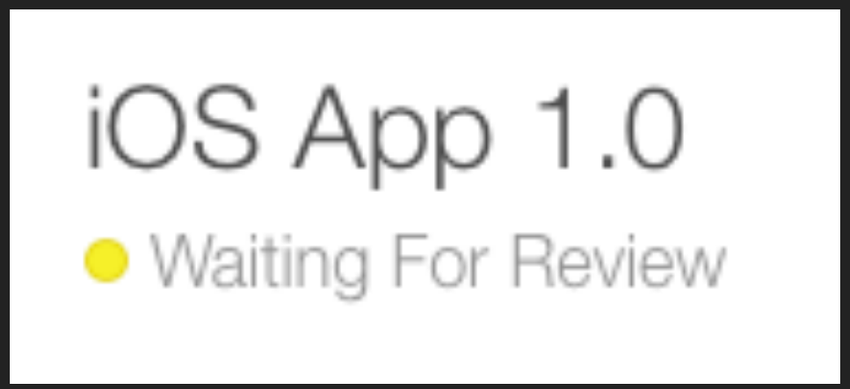
RELEASE IS TAGGED



SLACK IS NOTIFIED



MOST IMPORTANTLY...

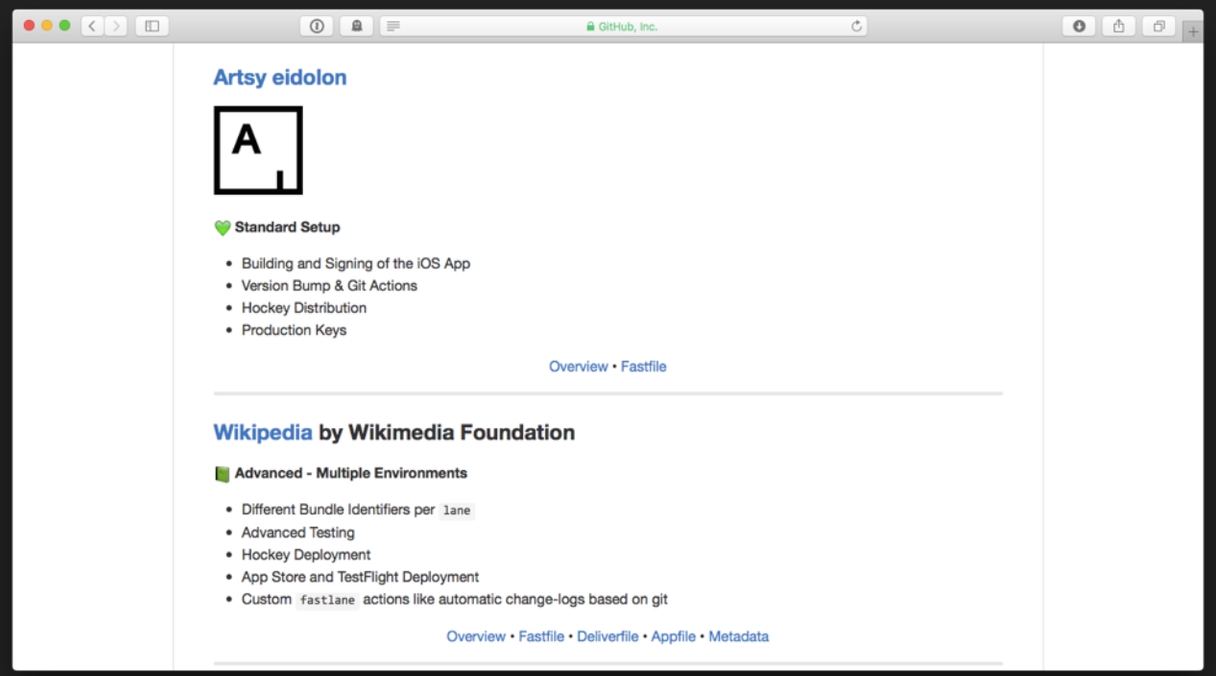


RECAP

Used Fastlane to smooth out:

- Provisioning Profiles (Sigh)
- Building (Gym)
- Submitting (Pilot & Deliver)
- Metadata (Snapshot & Deliver)

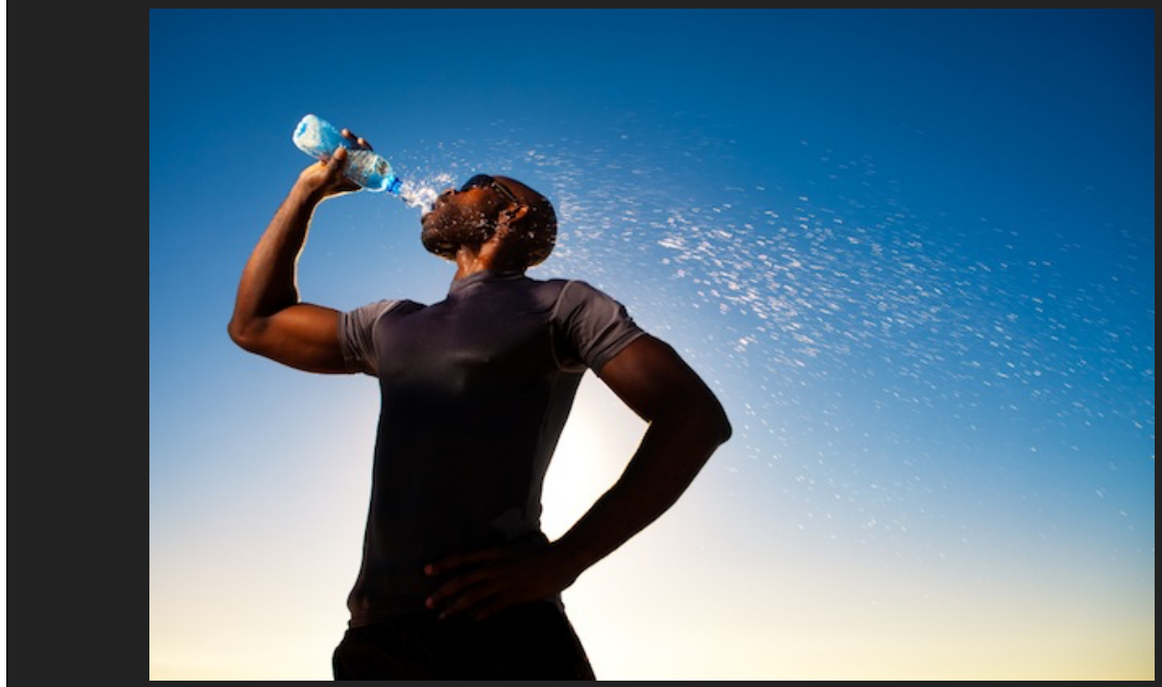
HTTPS://GITHUB.COM/FASTLANE/EXAMPLES



So I gave you 4 scenarios that can be automated or assisted using Fastlane.

On the Fastlane repo, you can see examples of how other companies are using it as well.

WHEW.



http://www.radiantpeach.com/tag/drink-water/

This is a reminder for me to drink water.

MY WORKFLOW

- Come up with scenario
  - “I need to get a beta build to Testflight.”
- First try to accomplish what I want reiteratively using the Command Line
- Next I move that to my Fastfile
- If arguments are constant, I move them to my *file.
- Optionally, I move it to my CI implementation.

GOTCHAS

No technology is perfect and Fastlane is by far the best suite for tackling these issues that I've encountered.
But it's not perfect.

GOTCHA #1: WHICH DIRECTORY AM I IN?

- Fastfile actions run in $(SRCROOT)/fastlane/
- Tools run in $(SRCROOT)/

GOTCHA #2: RUBY/SIMULATOR IS SLOOOOOOOOOOW

- We use Mac Minis for CI at work and was running into an error where the simulator wasn't starting up about 1 out of 20 builds.
- https://github.com/fastlane/fastlane/issues/1844
- And environmental variable was created (FASTLANE_XCODE_LIST_TIMEOUT) to put a deliberate timeout wait.

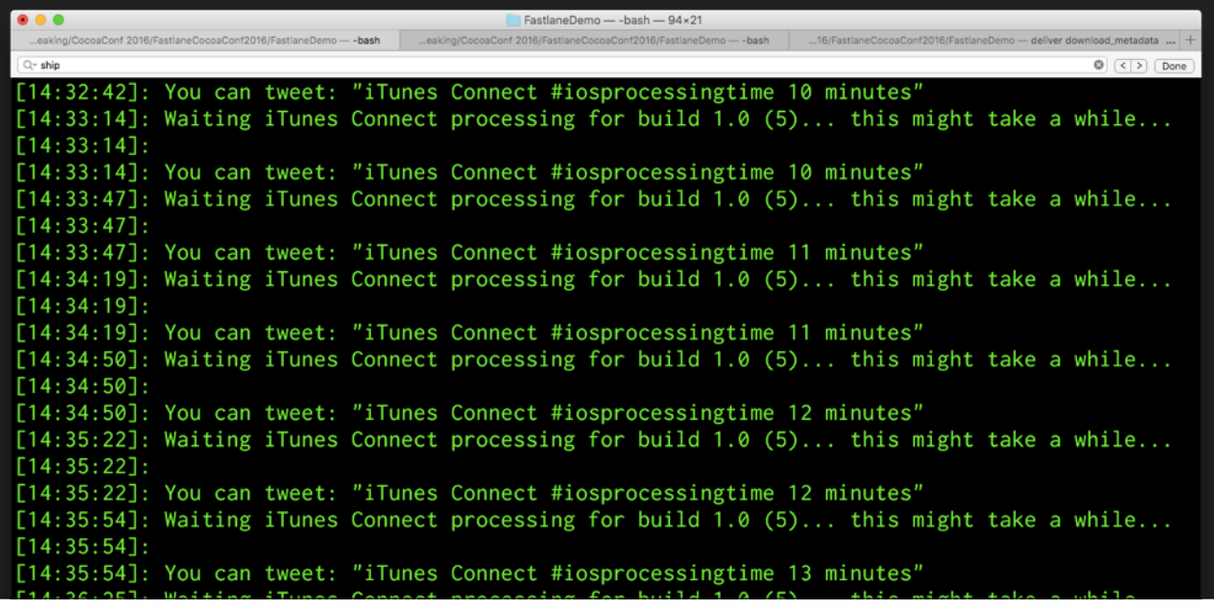
GOTCHA #3: IS IT YOU? FASTLANE? APPLE?

- Behind a firewall at work, Pilot stopped working.
- Cryptic error message was given.
- Issue was filed on Github.
- Google turned up nothing.
- Turns out it was Apple!

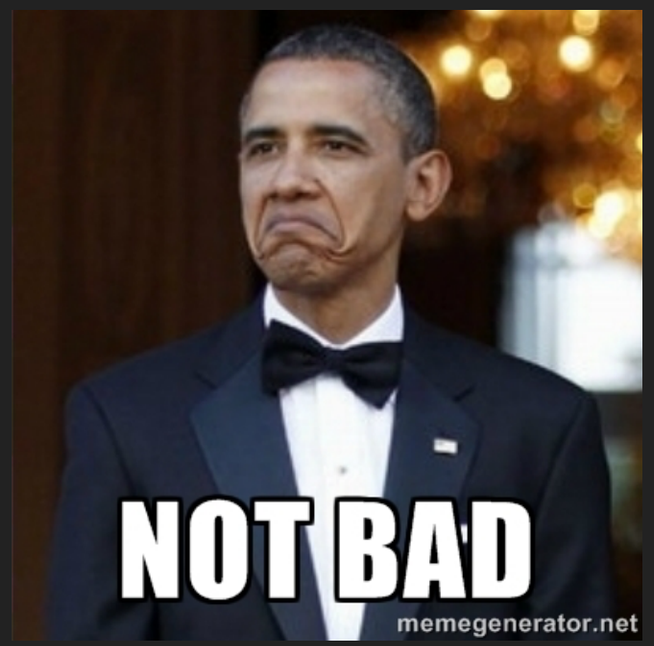
Introducing another tool adds another layer of doubt and makes troubleshooting just that little bit harder.

GOTCHA #4: ITUNES CONNECT IS SLOOOOOOOOOOOW

Deliver doesn't have a `skip_submission` flag so you have to wait forever.



OVERALL?



Really, these issues are very slight compared to the complexity of the issues it solves.

So, we talked about the bad parts but what about the great parts?

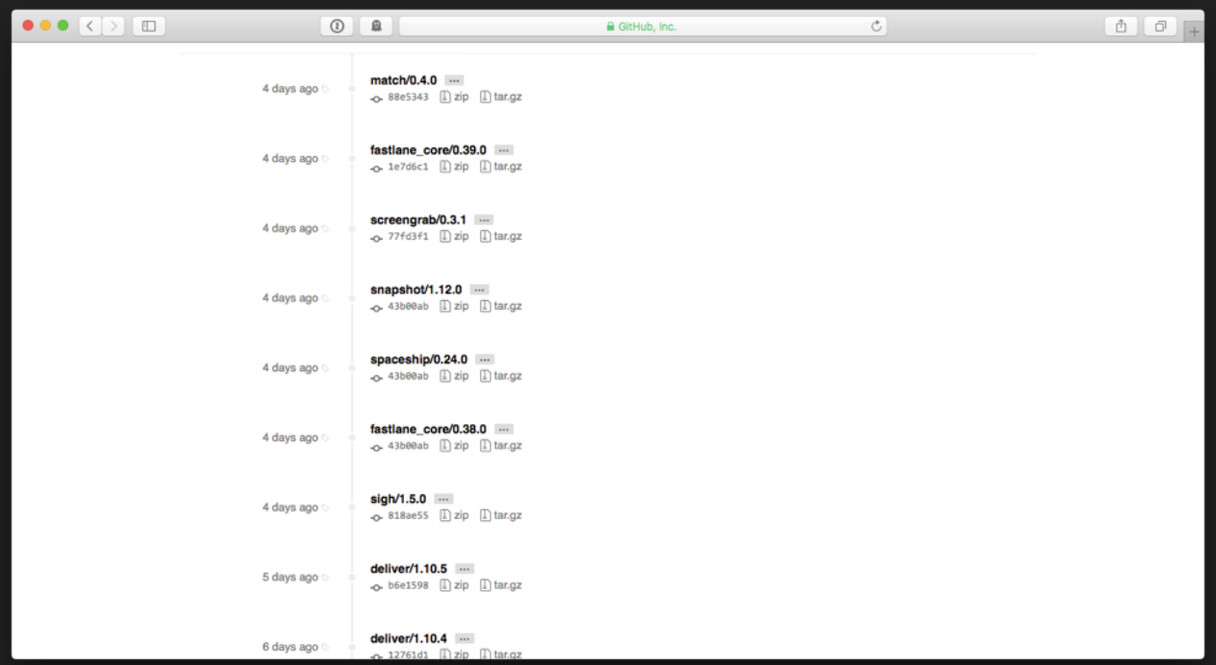
NIIIIIIIIIIICE #1: SNAPSHOT RESET_SIMULATORS

Have you seen this? Happens after you upgrade or have two Xcode versions installed.
Snapshot’s reset simulator function fixes that!

NIIIIIIIIIIICE #2: WELL SUPPORTED

This is the team behind Fastlane.
Fastlane was acquired by Twitter and now has an excellent support system.
File an issue and you'll at least get an response.

NIIIIIIIIIIICE #3: CONSTANTLY UPDATED



In addition to promptly responding to issues, the tool is constantly updated which is helpful.

BONUS:
CUSTOM ACTION

CUSTOM ACTIONS

- Think of it as a private_lane that you can share, reuse.
- Get it kicked off by running $ fastlane new_action
- Creates a file in fastlane/actions

```
module Fastlane
  module Actions
    module SharedValues
       …
    end
    class KillSimulatorAction < Action
        …
    end
  end
end
```

This is the structure of the file.

```
module Fastlane
  module Actions
    module SharedValues
       …
    end
    class KillSimulatorAction < Action
        …
    end
  end
end
```

In Shared Values, you define the key for the values you’d like to make universal

```
module Fastlane
  module Actions
    module SharedValues
       …
    end
    class KillSimulatorAction < Action
        …
    end
  end
end
```

A Class is created which is a subclass of Action
In here is where there is a structure that I'll go over.

```
class KillSimulatorAction < Action
  def self.run(params)
    sh(‘killall simulator’)
    Actions.lane_context[SharedValues::MY_SHARED_VAR
    IABLE] = ”my_val”
  end
end
```

The meat of is in self.run(params).

Here is where you’ll do your action.
If I want to make something universal, I’ll pass it to lane_context with the key defined before.

```
class KillSimulatorAction < Action
  def self.description
    "Kills the simulator"
  end
end
```

The next methods are all around documentation.
In this case, it's a short description.

```
class KillSimulatorAction < Action
 def self.available_options
   # Define all options your action supports.
   …
 end
end
```

Available options let’s you set up a config item which are used to express what arguments you want to be passed in.

```
FastlaneCore::ConfigItem.new()

key: :api_token,
env_name: "FL_KILL_SIMULATOR_API_TOKEN",
description: "API Token for KillSimulatorAction",
```

You'll use the ConfigItem object and pass in parameters that explain what the arguments are, if they are required, and what to do if they aren't fulfilled.

```
class KillSimulatorAction < Action
  def self.output
    …
  end
end
```

If you’re going to provide shared values, you document them here.

```
class KillSimulatorAction < Action
  def self.return_value
    …
  end
end
```

If you method provides a return value, you can describe here what it does here

```
class KillSimulatorAction < Action
  def self.is_supported?(platform)
    platform == :ios
  end
end
```

Finally, you can put limitations on your action for certain platforms or versions of the target os.

```
$ fastlane action

| kill_simulator  | Kills the simulator | @jacobvo |
```

So the result is that you'll see if if you type in fastlane action.

IN CONCLUSION...

- iOS Development can have it's rough spots
- Fastlane can help automate most of them no matter your development environment size
- Any 3rd party tool can introduce complications but...
- Fastlane is an excellent and well-maintained tool

QUESTIONS
ANSWERS

THINGS I DON'T KNOW ABOUT:

- Jenkins, Travis, Circle CI, Xcode bots
  - We use Bamboo at work
- How well this works with Extensions, Watch Apps, or Mac Apps
- BMWs

THANK YOU!

@JACOBVO

JACOB@SUSHIGRASS.COM