



CI/CD for iOS: Build, Test, Deploy

The Theory

So... what is CI and what is CD?

Most developers think of CI as continuous integration and CD as continuous delivery. Integration is the action of integrating or validating, in a repeated (continuous) way. Usually that means integrating or validating every PR, commit or merge. To put it more simply, we run our tests repeatedly. In a similar way, Continuous Delivery means we generate working binaries in a repeated way. The repetition for CD doesn't need to be at the same cadence as CI.

If you still have questions, we can check the [Wikipedia article](#):

CI/CD bridges the gaps between development and operation activities and teams by enforcing automation in building, testing and deployment of applications.

To make it even more clear, consider this example. As developers, we know about testing because we all write tests for our classes and methods, components and frameworks. There is a point when it makes sense to test those components against existing (and new) tests in a continuous and automated way. Running those tests consistently will ensure the correctness of the code written, and improve its quality.

Similar to the way Xcode runs the compiler every time we build our applications, continuous integration checks for the correctness of our code. (I know you actually need to compile the app to run it, but hopefully you get the point).

In a similar way, continuous delivery automatically builds and delivers new versions of the application (or SDK). This can happen on every merge or at a particular time of day (hello night builds). Continuous delivery is CI's natural partner and next in line. Why is it next? Usually we perform the build and ship tasks after all tests are green and we can guarantee a minimum quality of our app.

Continuous delivery can become a challenging aspect of app development. The costs are high in time and knowledge. CD is by far one of the most complex and brittle parts of the development process. CD can be well worth the cost because of how useful it is once you get it working. For many development teams, building and shipping an app to TestFlight or the AppStore can be stressful. Imagine how much less stressful it would be if you managed to automate that process. As a bonus, you have to document and transfer knowledge to create the delivery pipeline. This scales your team up a level, from relying on a person, to using a set of documents/code that is available to everyone. The result is a better and more robust delivery process that cannot be stopped by a single point of failure.

Delivery is usually performed by some tool on top of our friend Xcode. In this case we will use Fastlane, a tool that leverages xcodebuild. Fastlane is a group of Ruby scripts and plugins for automation, while xcodebuild is a command line tool you use to interact with your Swift or ObjC project.

There is no doubt about why CD is so complex. In our case building and shipping an iOS application involves: provisioning profile, certificate, knowing the target of the project you want to build, push notification certificates, uploading to the chosen store. True, some of the work is now automated but there are still many, many steps.

Why is CI/CD important?

As a developer I like working on a codebase that has tests. Why? Tests define the behaviour of the written code. I can refactor¹ with confidence. If tests are green, it is likely that I did not break anything. If tests are not green, that means I need to check what might have been broken or overlooked. In Xcode it is well integrated with CMD+U, but it can also be slow. Delegating your tests to your CI counterpart can be a good alternative. When you add a commit into an existing, open PR, CI can pick it up and run all the tests for you.

Why are tests important?

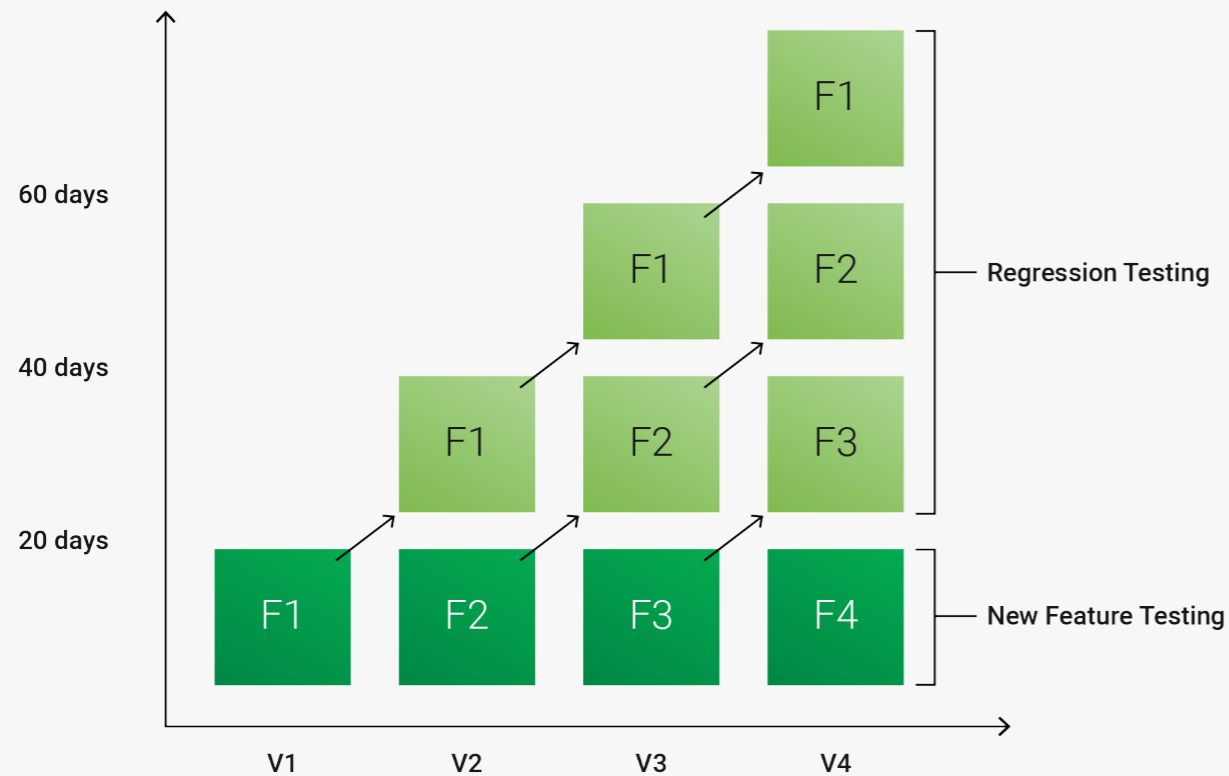
If you're trying to validate MVP, it is possible that you do not need tests. That is a very narrow use case, though. If you know what you are building you should have tests. If your engineering team is made up of five or more people and you don't have tests, that is a red flag.

1 – Refactoring means rewriting existing code but not changing how that code interacts with external actors

If your Engineering Manager tells you tests are not worth it, that is another red flag. Testing is like insurance. If you don't use it, your house could burn and you may not be able to cover the cost of repairing it.

Note: The need for tests can be a heated topic. In this book we will describe different viewpoints on the subject.

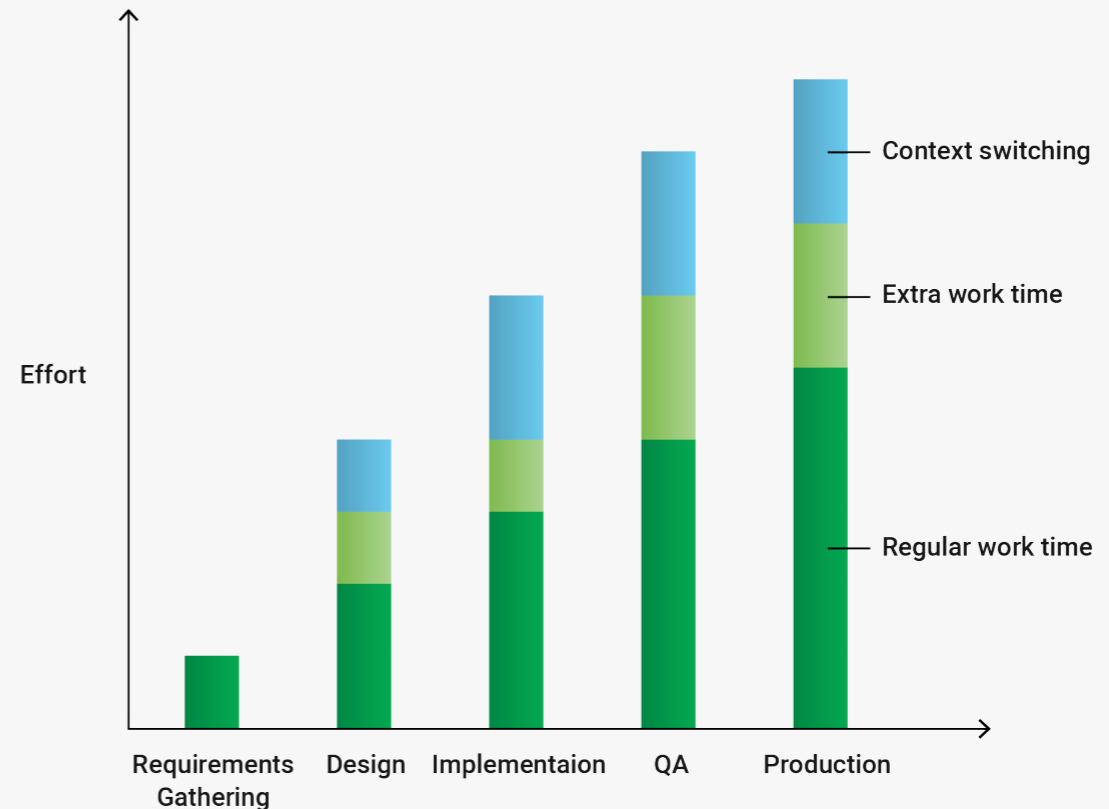
Tests increase confidence, and that confidence increases over time. For example, we may want to roll out our small app in four releases. For every release we make tests mandatory. On every new release we increase the amount of tests because new code = new tests. What happens at the end of the fourth release? We have tests that validate old parts of the shipped app while we used new tests for the newer parts. We call these regression tests. Regression tests cover existing parts of the app and verify that we did not break anything.



As I mentioned earlier, controlling cost is an important reason to have tests. Code shipped to production passes through different stages of development: requirements gathering, design, implementation, QA, and production. Catching bugs during the requirements stage can make a huge impact on reducing costs. Creating the wrong set of requirements is as bad as writing bad code.

For example: a requirement says the mortgage rate is 2%, but that rate should be 3%. It will not matter if the code follows the requirement, the rate will be wrong. A bug is a bug.

As a bug passes through each phase, it becomes more and more expensive, and it is possible that the cost will increase over time. Because you do not have the context you need, you will invest more time in fixing an unfamiliar part of the codebase. If the person who wrote it is not on the team anymore, that cost spikes! Having tests will help you better understand your codebase so you make fewer errors. Catching bugs earlier in the cycle reduces the cost of development. If the bug affects a critical part of the system you could also lose users, conversion, or money for the business.



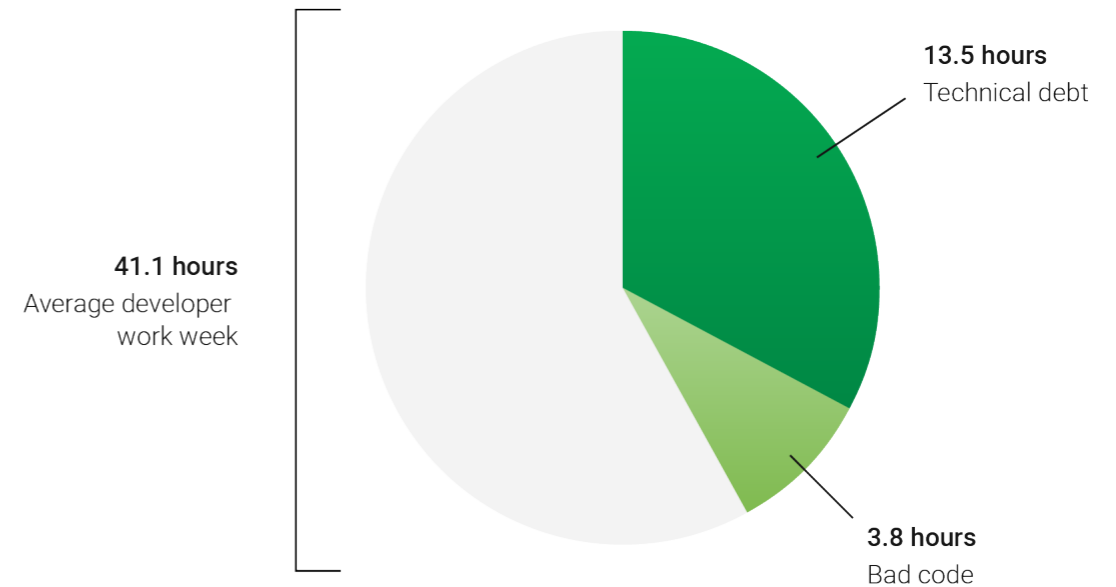
Does testing improve code quality?

The short answer is “yes”. A longer answer might be “yes, but indirectly in some cases”. Not having tests and ending up with bad code creates technical debt. The easier it is to work with your codebase and make changes to it, the more productive your team can be. No one can avoid all bad code or technical debt; we can reduce it by testing. For some organizations, code that is missing tests, is considered technical debt by default. Working with untested code is more difficult, more error prone, and more unpredictable.

Not trying to get into trouble here, but it is similar to the way that strongly-typed languages are less error prone than typeless languages. Some typeless languages are developing typed alternatives, like JavaScript and TypeScript.

The first step of our CI/CD integration is making sure we can lint and build our app, but we will talk about that later on. For now we will skip to the second step, which is running the tests against the app. Tests come from a need to validate written code against something. To prove that your code works, you need a way to see that it works, or you need one or more tests to validate that the code behaves “as expected”. In most Agile workflows tickets define tasks and establish “acceptance criteria”. Acceptance criteria is a set of rules that is used by someone other than the developer to validate that the code does what it is supposed to do. Verifying acceptance criteria is not the only way to test;

The Developer Work Week

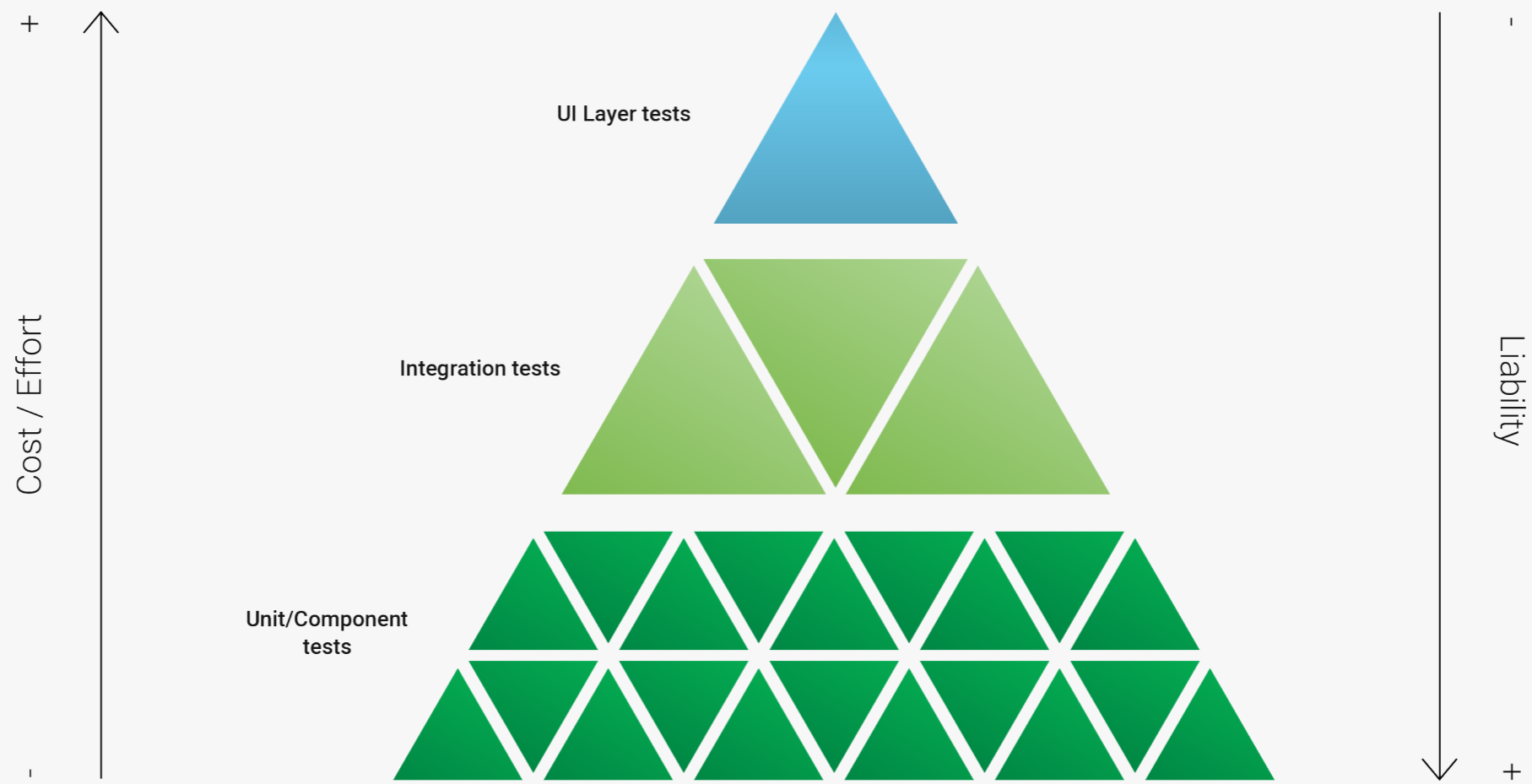


This image comes from the [Stripe Developer Coefficient from 2018](#).

you can also write tests to make sure there is high enough code coverage. Code coverage is a percentage over the total amount of code that is “covered” by test cases for that code. For example, a method that uses if/else, would need two tests to verify that it is fully tested (2 cases, 2 tests).

Why is my CI/CD setup important?

Assume you convinced your manager to substantially increase the number of tests your team runs. Assume also that all your colleagues are onboard with increasing test coverage. You now have not only unit tests, but integration tests, and UI (or smoke) tests. These three types of tests create a “testing pyramid”.



The testing pyramid shows the level of cost, complexity, and time it takes to write and maintain tests. You can use the pyramid to help you decide on the amount of code coverage you need. At the bottom, there is fast and cheap testing. As we rise to the top, testing becomes more complex and time-consuming, and therefore, more error prone. That is a big reason that people do not like testing, or consider it a waste of time. Tests need to keep running and working for them to return value.

And that is exactly why you want a CI/CD system. As your codebase grows, the time it takes to run tests will increase from mere minutes, and may even begin to take hours. Of course, most devs are not willing to wait hours to merge things, right? Teams develop strategies to mitigate long wait times. Devs want confidence and they want speed! There are many different strategies to choose from. If you have enough machine power, you can run tests in parallel. It might be that not every test is needed all the time. In that case, you can run tests based on the specific changes that were made in the codebase, and run the whole suite less often.

That is why having a good CI/CD system empowers the team instead of dragging it down. As your app becomes more successful, it will become bigger and bigger, and require more power. Builds will start to take longer, and what used to be a ten-minute build starts to hit the pain threshold of 50 to 60 minutes. That is why parallel tests are key to building and shipping applications quickly.

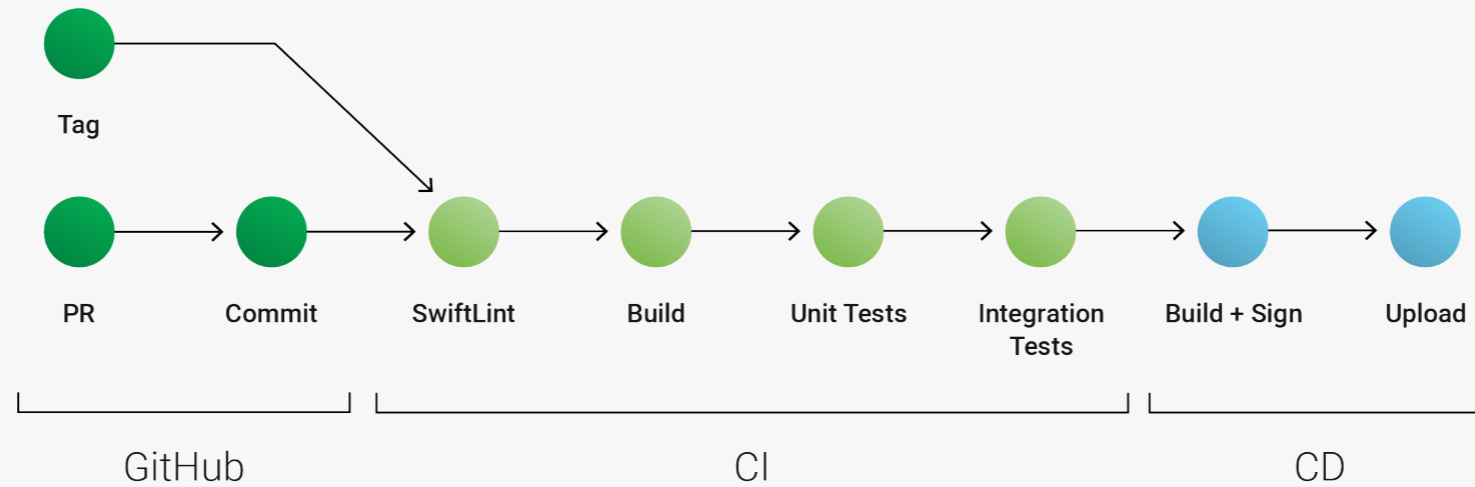
What does a CI/CD workflow look like?

What are the ideal steps to include in our workflow? First, we want to guarantee some minimum level of correctness to not waste resources, that's achieved with linters. Then we take the correctness checks and static analyzer tests. That's the CI part of the equation. CI: Lint, Build, Test.

Then we bundle our application, sign it with our certificate, and send it to the chosen store. This can be the Store, Testflight, a third party like TestFairy or Firebase App Distribution. Taking advantage of your company's enterprise account can also be helpful. An enterprise account is a different type of developer account that a company can use for shipping internal or external apps. It allows you to generate a new set of certificates and sign the app to deliver internally or externally without any other service. It is a great option if your user test base is big!

Many teams fear CD because they think they have to ship binaries to users at the same cadence as its CI pipeline. That is not true!

CD: build for delivery, sign, upload (with options to ship).



Shipping betas to testers or even internal users of the company can be decoupled from the CI cadence and can be done less frequently. Shipping to end users is the same process but signed with a different certificate, making it a different app. Decoupling cadence from CI means you can choose when and how you deliver to your testers and when you deliver to your users. It is the same process happening at different points in time and with different actions. For example, creating a tag in master may trigger the delivery process to the AppStore, The options are endless, really.

This completes the cycle, from a merged PR to a new version of your app in your device in minutes. The time will vary depending on the size of the project.

Show me the code

Building the CI/CD pipeline with CircleCI

Using the sample app

The app we are using for this exercise uses more than one NASA API to get several different images. Each API provides different images and conditions than the others. The idea is to have 2 or 3 screens where we can review images from each NASA API. We want to store them so that we can access and display them later. Our goal is to create something similar to a like button or favorites icon.

The basic element is a router to hold the navigation logic. The router should be unit testable. Each screen will have View Models that use MVVM architecture for screens. We will use classes to hold the networking details and some state. This project is a simple example, so it does not use complex architecture like RIBs, Clean, or VIPER.

For this sample project, we will be using some CocoaPods libraries that we would not usually need to. Alamofire is great, but the Apple networking library is really very good as well.

Starting with Xcode

We will use a basic application that we will be supercharging for our demonstration. We will start in Xcode by creating a new application that does not use CocoaPods. We will not be using any SwiftUI this time (sorry about that). We will create a good old Swift application, with tests.

We will name it “Nasa”. It is a cool name for a cool app, and we will find an API that will bring us some images from outer space, or maybe just from Earth.

With the basic project created in Xcode, we can add CocoaPods. We will use Bundler, a developer tool, to add the CocoaPods. Bundler lets you define what libraries you want to use, and what versions of them you want.

Create a Gemfile file: ``vi Gemfile``

Enter the commands we need for right now:

```
source 'https://rubygems.org'
```

```
gem 'cocoapods'
```

The next step is installing our dependencies. This can take some time.

```
bundle install
```

The command generates a Gemfile.lock to assist any future user of the project by installing exactly the same library versions you are using right now. This command makes it easy for anyone to start using the project. It is great for teams, especially teams implementing CI.

Next, we need to run all commands preceded by ``bundle exec``, to make use of the Gemfile we just created. When you run ``bundle exec pod --version`` you should get a result in the range of 1.10.1. For the example, the result should match the contents of Gemfile.lock.

Here is the interesting part: ``bundle exec pod init``. What happened? We used the power of CocoaPods tools to inspect our project and create a Podfile file, which is similar to a Gemfile:

```
# Uncomment the next line to define a global
platform for your project

# platform :ios, '9.0'

target 'Nasa' do

    # Comment the next line if you don't want to
    use dynamic frameworks

    use_frameworks!

    # Pods for Nasa

    target 'NasaTests' do

        inherit! :search_paths

        # Pods for testing

    end

    target 'NasaUITests' do

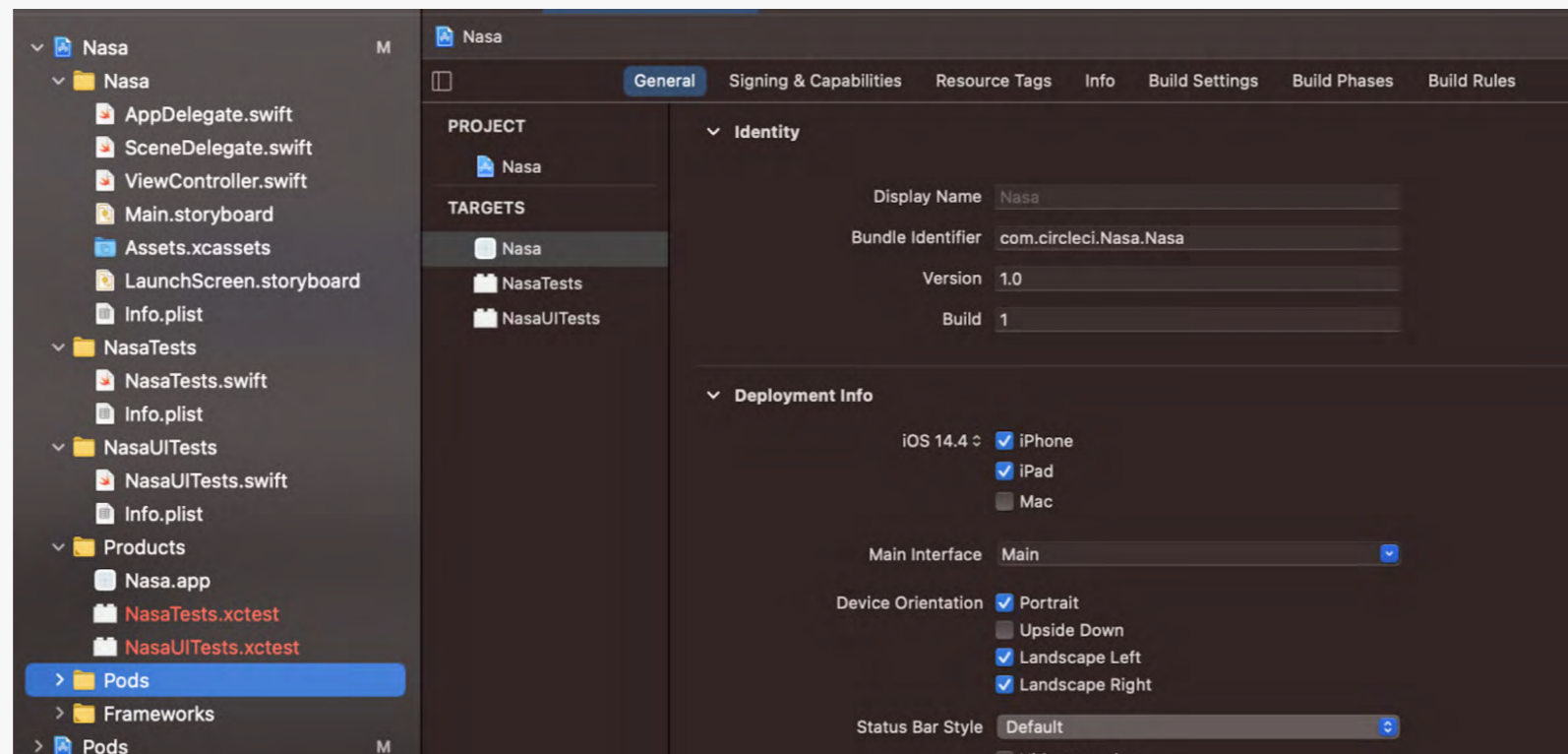
        # Pods for testing

    end

end
```

In this case we want to use iOS 13 as default, so uncomment that line and set the platform version to 13. We also see three targets: `Nasa`, `NasaTests`, and `NasaUITests`. Those are different “states” that we can use to run different configurations. For this project, there are two sets of tests for the target `Nasa` and the main target of our project, also named `Nasa`.

We still do not have any .xcworkspace available to us. We can use the recently created `Podfile` by using `bundle exec pod install`.



The results:

- A 'Podfile.lock' file is created for us in similar fashion to Gemfile.lock
- A 'Pods' folder has been created (where our new libraries are going to be)
- We have a Nasa.xcworkspace that will become our main working file now. This is the file we want to open Xcode with so we can keep working. It contains all the magic that CocoaPods uses to build and run the app

Once the installation is finished, use the Xcode shortcut 'CMD + B' to build our app.

Here is a recap of our project so far. We have a working folder named `Nasa` and two other folders for tests. One folder is for regular tests and one is for UI tests. The main config is listed in the Project section, with the main target and the target for tests listed below that.

Setting up an iOS app on CircleCI and Github

Our first example repo for setup will be hosted: <https://github.com/CI-CD-for-iOS/guide-setup>

Now that we have the basics working, we can use GitHub to connect to our CircleCI instance. Go to <https://circleci.com/vcs-authorize/> and log in with your Github Account. After choosing an organization, select the repository you want to try.

First, create a ``.circleci/config.yml`` file inside the project repo. Note that `config.yml` is created inside a "hidden" folder on our terminal. That is what the `."` stands for at the beginning of the "circleci" folder.

This is a good starting point:

```
# Use the latest 2.1 version of CircleCI pipeline process engine. See: https://circleci.com/docs/2.0/configuration-reference
version: 2.1

# Use a package of configuration called an orb.
orbs:
  # Declare a dependency on the welcome-orb
  welcome: circleci/welcome-orb@0.4.1

# Orchestrate or schedule a set of jobs
workflows:
  # Name the workflow "welcome"
  welcome:
    # Run the welcome/run job in its own container
    jobs:
      - welcome/run
```


PIPELINE	STATUS	WORKFLOW	BRANCH / COMMIT	START	DURATION	ACTIONS
guide-test 3	Success	welcome	main ab3bb2f pod install + spm install	28m ago	20s	🔄 🗑️ 📄 ⋮
Jobs	Success	welcome/run 3			18s	

Now we have some keywords in a .yml file. If we commit and push this file, some changes will happen and be shown on our project's CircleCI page.

welcome/run Success Rerun ⋮

Duration / Finished: 18s / 34m ago | Queued: 0s | Executor: Docker Medium | Branch: main | Commit: b563690, 2a583d0, ab3bb2f | Author & Message: pod install + spm install

STEPS TESTS ARTIFACTS

1 / 80 parallel runs

- Spin up environment (17s)
- Preparing environment variables (0s)
- Congratulations! (0s)
- Next Steps (0s)
- Help Topic: Examples - Tutorials, Sample Configs, Cookbook (0s)
- Help Topic: Overview and Concepts (0s)
- Help Topic: Using the CLI (0s)

Inside the workflow there is more detail. Take a few minutes to familiarize yourself with the steps list, the test tab, and the artifacts tab. The page shows that the Executor is Docker and that it ran on the main branch. It also shows the number of commits, along with information about the author and the commit.

We are building for iOS, so we may want to review **some documentation about Mac OS build environments** before moving forward.

Updating config.yml with MacOS values

Start by changing our config.yml so that it uses a Mac machine. To do that, we need to add the `macos` and `xcode` values to our yml. Next, we will make a couple of changes and add test and build information.

Test command

```
xcodebuild test -workspace Nasa.xcworkspace -scheme Nasa -destination 'platform=iOS Simulator,name=iPhone 12,OS=14.4'
```

Build command

```
xcodebuild -workspace Nasa.xcworkspace -scheme Nasa
```

Using these two commands we can test and build our application. We will execute these two commands one after another, as if they depended on each other.

```
version: 2.1

jobs: # a basic unit of work in a run

  build: # runs not using `Workflows` must have a `build` job as
entry point

  macos: # indicate that we are using the macOS executor

  xcode: 12.4.0

  steps: # a series of commands to run

    - checkout # pull down code from your version control
system.

    - run:

      # run our tests using xcode's cli tool `xcodebuild`

      name: Run Unit Tests

      command: xcodebuild test -workspace Nasa.xcworkspace -scheme
Nasa -destination 'platform=iOS Simulator,name=iPhone
12,OS=14.4'

    - run:

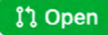

      # build our application

      name: Build Application




      command: xcodebuild -workspace Nasa.xcworkspace -scheme Nasa
```

In GitHub, CircleCI is now included as part of our PR:



Setting config.yml #2

 Open wolffan wants to merge 1 commit into `main` from `move-config-yml-to-nasa` 


Conversation 0 Commits 1 Checks 0 Files changed 22


 wolffan commented 1 minute ago  

No description provided.


  all ready to start ● 35939d3


Add more commits by pushing to the `move-config-yml-to-nasa` branch on `CI-CD-for-iOS/guide-test`.




 **Some checks haven't completed yet** [Hide all checks](#)

1 pending check

-  **ci/circleci: build** Pending — CircleCI is running your tests [Details](#)

 **This branch has no conflicts with the base branch**
Merging can be performed automatically.

Merge pull request  You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

build Running Rerun ...

Duration: 59s | Queued: 0s | Executor: macOS Medium | Branch: move-config-yml-to-nasa | PR / Commit: #2 / 35939d3 | Author & Message: all ready to start

STEPS TESTS ARTIFACTS

1 / 15 parallel runs

- Spin up environment (4s)
- Preparing environment variables (0s)
- Checkout code (1s)
- Install dependencies (18s)
- Run Unit Tests (33s)

Click **Details** to review what is happening in CircleCI. Note that we are now using the Mac instance to build and run our unit tests. Navigating between GitHub to CircleCI and back using the connection between the tools facilitates an effective workflow.

Add more commits by pushing to the `move-config-yml-to-nasa` branch on `CI-CD-for-iOS/guide-test`.

All checks have passed Hide all checks
1 successful check

ci/circleci: build — Your tests passed on CircleCI! Details

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request ▼ You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

When everything is green we can merge our PR right away, but remember that it is best that another team member reviews it before the merge. We are skipping this step in this project, to simplify the process. Reviews are important, though, and you can require them on Github.

We chose to put Pods inside the project for this demonstration, so a `Pods` folder is listed. When you are working on a different project, you can either check those pods in, or have them pulled as a step in your CI process. CircleCI will cache them for you if you do not include them in your project. This decision is a mix of technical and personal choice. I like having any code from the repository ready to be used, but for another project that might not be an option. Other solutions, like Carthage, cache not only the code, but the compiled binaries you use (other libraries, for example), so that the project takes less time to compile. For other projects, there are solutions available on CocoaPods with **Rome**, or other tools like **Tuist**.

Note: For this project, we have not applied .gitignore. Xcode and Swift development projects have personal files in the repository. Most of the time, we do not want them there. We chose to include the Pods folder, but not all teams will want to. For those teams, the /Pods folder would be a good candidate to go into the .gitignore file. There are open source solutions for this process: <https://github.com/github/gitignore>. We included the swift .gitignore and updated the project so the configuration files will not be included in our PRs.

For the next part of the demonstration, we will use a new project to show how to manage multiple projects in the same repo. This project also contains tests. Like the previous project, our new project has 1 target for the tests (we will create a new one for UI tests). The difference is that this project has both unit tests and integration tests.

Using mocking and stubbing in integration testing

For the more complete project: <https://github.com/CI-CD-for-iOS/guide-test>.

Integration tests make sure that two or more components work together successfully. Setting up integration tests includes “mocking” and “stubbing” project components. Mocking can be done with a tool. You can also mock a component by creating a subclass (typical for Swift) and changing the inner implementation.

Mocking components

Mocking means creating a fake version of an external or internal service that can stand in for the real one, helping your tests run more quickly and more reliably. A mock behaves like the real service but in a much simpler way. When your implementation interacts with an object’s properties, instead of its function or behavior, you can use a mock.

Stubbing a class

Stubbing, like mocking, means creating a stand-in, but a stub only mocks the behavior, not the entire object. You can use a stub when your implementation only interacts with a certain behavior of the object. If you find you are relying on multiple stubs for testing, it may be a symptom that something in your code is not right. For example, it might not have enough injection, or the components are too big.

The internet is full of [reasons for integration testing](#).



This “code” may have been 100% unit tested, but without integration testing, it does not work at all.

Configuring fastlane for launching tests

We want to write and organize tests for our project around our CircleCI setup. The long command that we saw before is going to become annoying before long. We want an easier way to launch our tests, so we encourage using Fastlane with CircleCI. The combination simplifies the setup and automation of the build, test, and deploy process..

Go to our /wallet project:

(``cd wallet``) and run the command

``bundle exec fastlane init``. Remember to use ``bundle exec`` in front of the commands.

```
[✓] 🚀
[✓] Looking for iOS and Android projects in current directory...
[19:49:08]: Created new folder './fastlane'.
[19:49:08]: Detected an iOS/macOS project in the current directory: 'ETHRadar.xcworkspace'
[19:49:08]: -----
[19:49:08]: --- Welcome to fastlane 🚀 ---
[19:49:08]: -----
[19:49:08]: fastlane can help you with all kinds of automation for your mobile app
[19:49:08]: We recommend automating one task first, and then gradually automating more over time
[19:49:08]: What would you like to use fastlane for?
1. 📸 Automate screenshots
2. 🛩️ Automate beta distribution to TestFlight
3. 🚀 Automate App Store distribution
4. 🛠️ Manual setup - manually setup your project to automate your tasks
```

Fastlane gives you 4 options. I will describe 2 and 3 a bit later, but for now select the fourth option.

We have accomplished so much! For now, we can skip ``/fastlane/Appfile`` and work with ``/fastlane/Fastfile``.

```
[19:50:30]: -----
[19:50:30]: --- Setting up fastlane so you can manually configure it ---
[19:50:30]: -----
[19:50:30]: Installing dependencies for you...
[19:50:30]: $ bundle update
[19:51:12]: -----
[19:51:12]: --- ✅ Successfully generated fastlane configuration ---
[19:51:12]: -----
[19:51:12]: Generated Fastfile at path `./fastlane/Fastfile`
[19:51:12]: Generated Appfile at path `./fastlane/Appfile`
[19:51:12]: Gemfile and Gemfile.lock at path `Gemfile`
[19:51:12]: Please check the newly generated configuration files into git along with your project
[19:51:12]: This way everyone in your team can benefit from your fastlane setup
[19:51:12]: Continue by pressing Enter ↵

[19:52:25]: fastlane will collect the number of errors for each action to detect integration issues
[19:52:25]: No sensitive/private information will be uploaded, more information: https://docs.fastlane.tools/#metrics
[19:52:25]: -----
[19:52:25]: --- fastlane lanes ---
[19:52:25]: -----
[19:52:25]: fastlane uses a `Fastfile` to store the automation configuration
[19:52:25]: Within that, you'll see different lanes.
[19:52:25]: Each is there to automate a different task, like screenshots, code signing, or pushing new releases
[19:52:25]: Continue by pressing Enter ↵

[19:52:31]: -----
[19:52:31]: --- How to customize your Fastfile ---
[19:52:31]: -----
[19:52:31]: Use a text editor of your choice to open the newly created Fastfile and take a look
[19:52:31]: You can now edit the available lanes and actions to customize the setup to fit your needs
[19:52:31]: To get a list of all the available actions, open https://docs.fastlane.tools/actions
[19:52:31]: Continue by pressing Enter ↵

[19:52:34]: -----
[19:52:34]: --- Where to go from here? ---
[19:52:34]: -----
[19:52:34]: 📸 Learn more about how to automatically generate localized App Store screenshots:
[19:52:34]: https://docs.fastlane.tools/getting-started/ios/screenshots/
[19:52:34]: 🧑‍🔬 Learn more about distribution to beta testing services:
[19:52:34]: https://docs.fastlane.tools/getting-started/ios/beta-deployment/
[19:52:34]: 🚀 Learn more about how to automate the App Store release process:
[19:52:34]: https://docs.fastlane.tools/getting-started/ios/appstore-deployment/
[19:52:34]: 🛡️ Learn more about how to setup code signing with fastlane
[19:52:34]: https://docs.fastlane.tools/codesigning/getting-started/
[19:52:34]:
[19:52:34]: To try your new fastlane setup, just enter and run
[19:52:34]: $ fastlane custom_lane
```

We have a barebones configuration that shows what we can do.

```
default_platform(:ios)

platform :ios do

  desc "Description of what the lane does"

  lane :custom_lane do

    # add actions here: https://docs.fastlane.
tools/actions

  end

end
```

We will use the name in between ‘:’ and ‘do’ to create commands we can run from outside the file. In fact, we will run the commands from our CircleCI file.

First we need to migrate our 2 commands (test and build) to fastlane.

In fastlane, the equivalent of test is ``scan``. The build equivalent is ``gym`` (gym is the other name given to ``run_test``). You can find more information in the great [fastlane documentation](#).

The updated fastfile

```
lane :test do

  # add actions here: https://docs.fastlane.tools/
actions

  scan(

    workspace: "ETHRadar.xcworkspace",

    scheme: "ETHRadar",

    clean: true,

    destination: "platform=iOS
Simulator,name=iPhone 12,OS=14.4"

  )

end
```

Go to your folder and run ``bundle exec fastlane test``. We added the “clean” option, which will help avoid any external factors affecting your test. The option is configurable, so you do not have to use it, but it is worth giving it a try.

Now that we have the fastfile and the modified config.yml to call it, we can do another push.

Previous screenshots were from a computer, but these are from the CircleCI log. The format for each is similar,

There are different files running, with 9 tests total. Ideally we want some more detail on our PRs, but it is not available on GitHub.

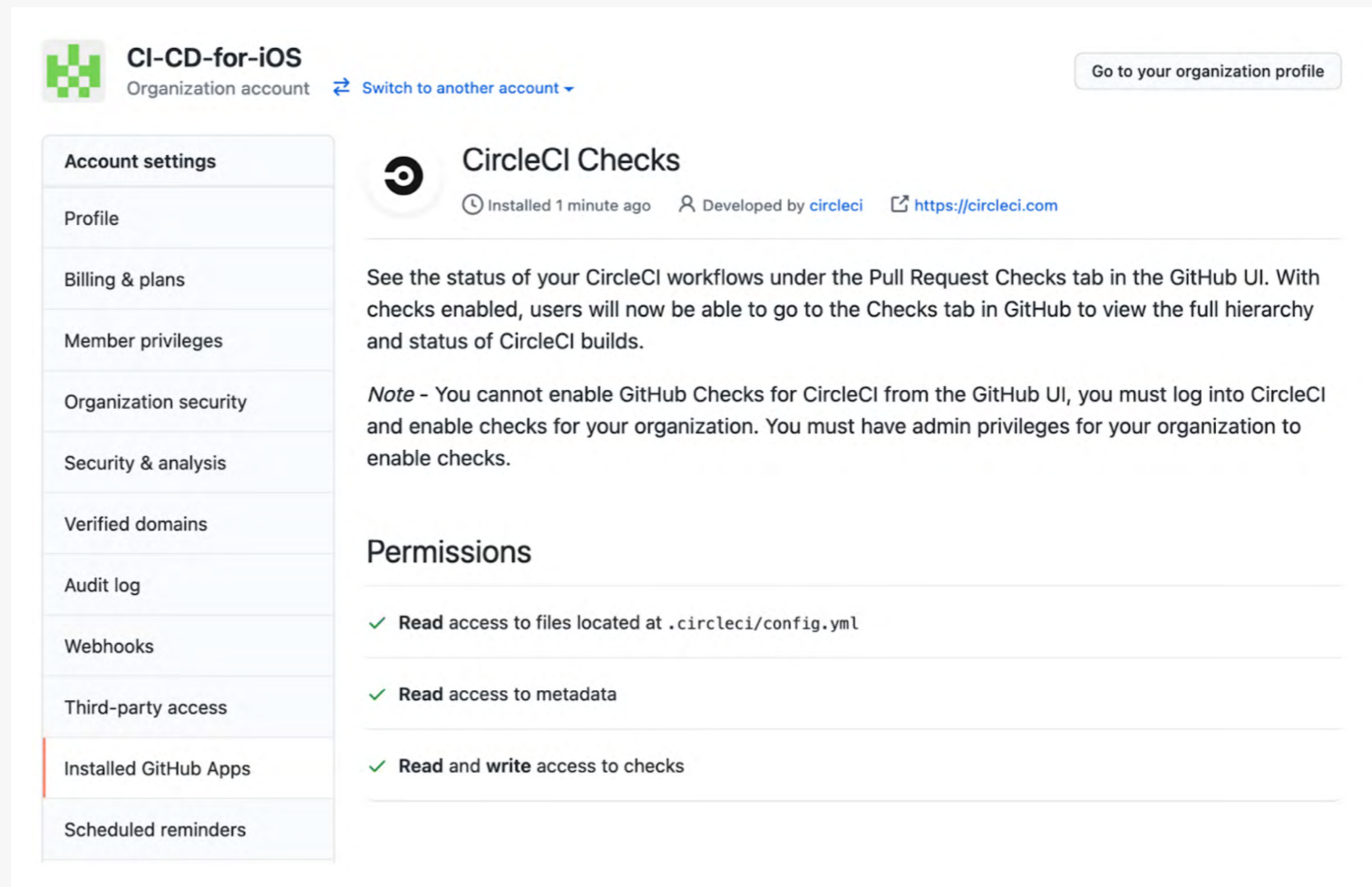
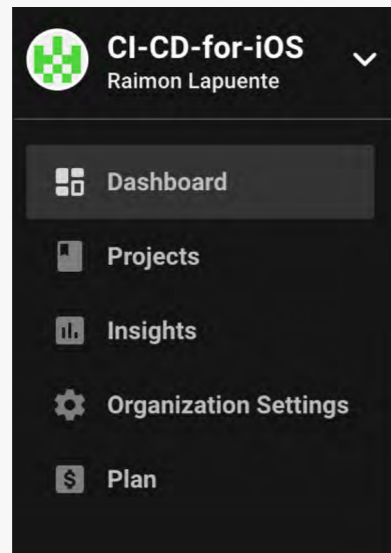
To set it up, go to the CircleCI website and navigate to the VCS section of the CircleCI Organization Settings.

```
[21:09:18]: ▶ Test Suite ETHRadarTesting.xctest started
[21:09:18]: ▶ DateHelpersTests
[21:09:18]: ▶   ✓ testGetTimeReturnsDoubleDigits (0.005 seconds)
[21:09:18]: ▶   ✓ testGetTimeReturnsSimpleDigits (0.001 seconds)
[21:09:18]: ▶ ETHRadarTesting
[21:09:18]: ▶   ✓ testExample (0.000 seconds)
[21:09:18]: ▶   ⚠ testPerformanceExample measured (0.000 seconds)
[21:09:18]: ▶   ✓ testPerformanceExample (0.658 seconds)
[21:09:18]: ▶ IntegrationStorageTests
[21:09:18]: ▶   ✓ testRetrivalKeepsDataStructure (0.003 seconds)
[21:09:18]: ▶   ✓ testStorageWhenNotEmpty (0.002 seconds)
[21:09:18]: ▶   ✓ testStorgeIsEmpty (0.001 seconds)
[21:09:18]: ▶ TokenViewModelTests
[21:09:18]: ▶   ✓ testChatViewModelShowsCorrectCoinName (0.001 seconds)
[21:09:18]: ▶   ✓ testChatViewModelShowsUnknownCoinName (0.001 seconds)
[21:09:18]: ▶ Executed 9 tests, with 0 failures (0 unexpected) in 0.672 (0.827) seconds
[21:09:18]: ▶
[21:09:19]: ▶ 2021-03-25 21:09:19.548 xcodebuild[1260:7681] [MT] IDETestOperationsObserverDebug: 133.021 elapsed -- Testing started completed.
[21:09:19]: ▶ 2021-03-25 21:09:19.548 xcodebuild[1260:7681] [MT] IDETestOperationsObserverDebug: 0.000 sec, +0.000 sec -- start
[21:09:19]: ▶ 2021-03-25 21:09:19.548 xcodebuild[1260:7681] [MT] IDETestOperationsObserverDebug: 133.021 sec, +133.021 sec -- end
[21:09:20]: ▶ Test Succeeded
+-----+
|      Test Results      |
+-----+
| Number of tests      | 9 |
| Number of failures   | 0 |
```

You will be asked to enter your GitHub password, and you may need permissions.

When you return to the GitHub settings page, CircleCI Checks has been added as a new GitHub App.

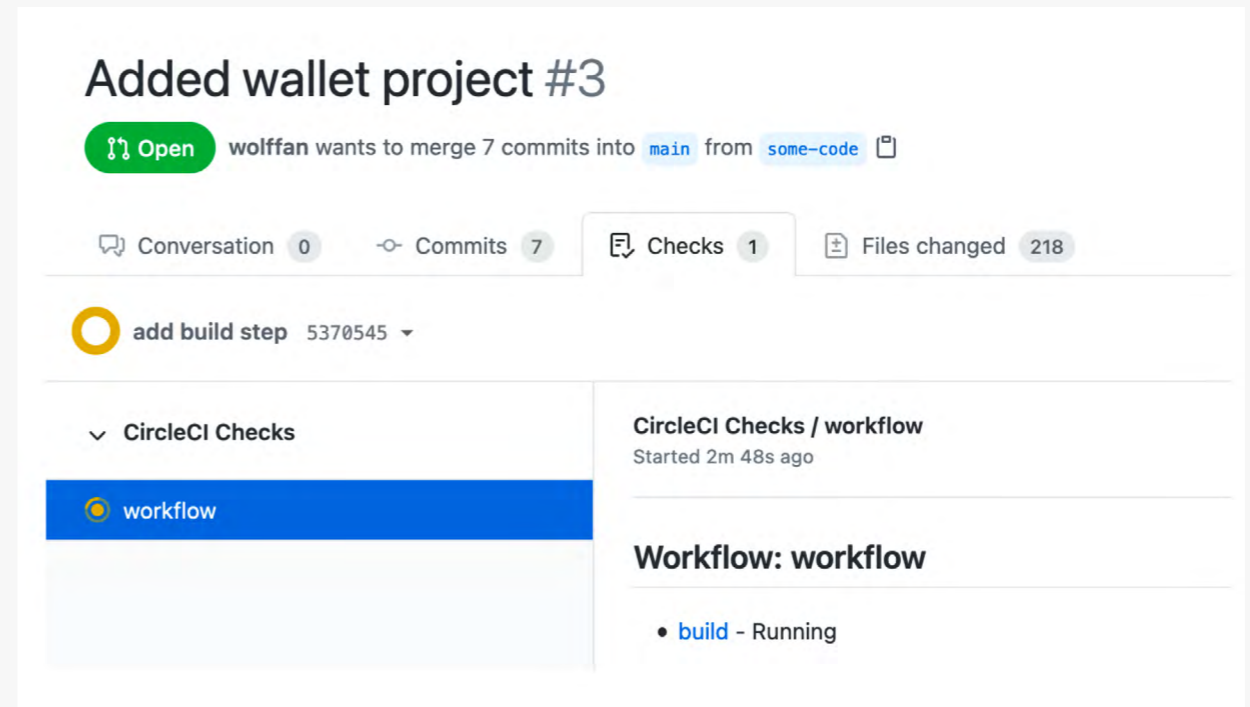
There is only 1 job running, but in our config.yml we said we would build and test. Something is going on. To find out what is happening, we need to look at our config.yml.



The `config.yml` states that `build`, jobs not using `Workflows` must have a `build` job as entry point. We need to configure our project with the proper workflows to take full advantage of the GitHub checks. This configuration lets us visualize the activity clearly and shows our project progress on CI.

Ideally we should do some checks even before we run our tests. We could do some linting, perhaps using Danger or a similar tool, or maybe make sure some non-functional checks are passed. Then we want to run tests, and maybe after that we can work on making a build for release. To do this, we need to **transform our current `config.yml` file to something that works in “steps”**.

With the help of an orb (prebaked configuration yml that you or someone else put together) we can create some more quality checks before we run our tests. We will also introduce a linter and a non-functional review using SwiftLint + Danger + **CircleCI quality checks**.



The screenshot shows a GitHub pull request titled "Added wallet project #3". It indicates that "wolffan" wants to merge 7 commits into the `main` branch from a branch named `some-code`. The interface includes navigation tabs for Conversation (0), Commits (7), Checks (1), and Files changed (218). Below the tabs, there is a section for "add build step" (ID: 5370545) with a dropdown menu showing "CircleCI Checks" expanded to reveal a "workflow" step. To the right, the "CircleCI Checks / workflow" section shows it started 2m 48s ago. Underneath, the "Workflow: workflow" section lists a single job: "build - Running".

Note: To run orbs someone from your team needs to approve them in the CircleCI security settings for your organization.


```
version: 2.1
orbs:
  ios-quality-checks: storytel/ios-quality-checks@1.0.1
jobs:
  danger:
    macos:
      xcode: 12.4.0
    steps:
      - checkout
      - run:
          command: bundle install
      - run:
          command: bundle exec danger
  test:
    macos:
      xcode: 12.4.0
    steps:
      - checkout
      - run:
          name: Install dependencies
          command: bundle install
      - run:
          name: Run Unit Tests
          command: cd wallet && bundle exec fastlane test
```

The initial native code will not actually work because we first need to configure SwiftLint and Danger. Please bear with me.

We define our jobs first, and then, below, we create the sequence of jobs that need to run in order.

As expected, both Danger and Lint failed. One because we did not have a Dangerfile configured and the other because we didn't consider linting in the first place.

```
build:
  macos:
    xcode: 12.4.0
  steps:
    - checkout
    - run:
      name: Install dependencies
      command: bundle install
    - run:
      name: Run Unit Tests
      command: cd wallet && bundle exec fastlane build

workflows:
  checks_for_deploy:
    jobs:
      - danger
      - ios-quality-checks/swiftlint:
          path: wallet/
      - test
```

Here is a super quick setup checklist:

- Add gem 'danger' to our Gemfile
- Run `bundle install`
- To initiate `bundle exec danger init`
- Follow instructions in terminal
- Remember to add your Token into the CircleCI Project Settings in Environment Variables

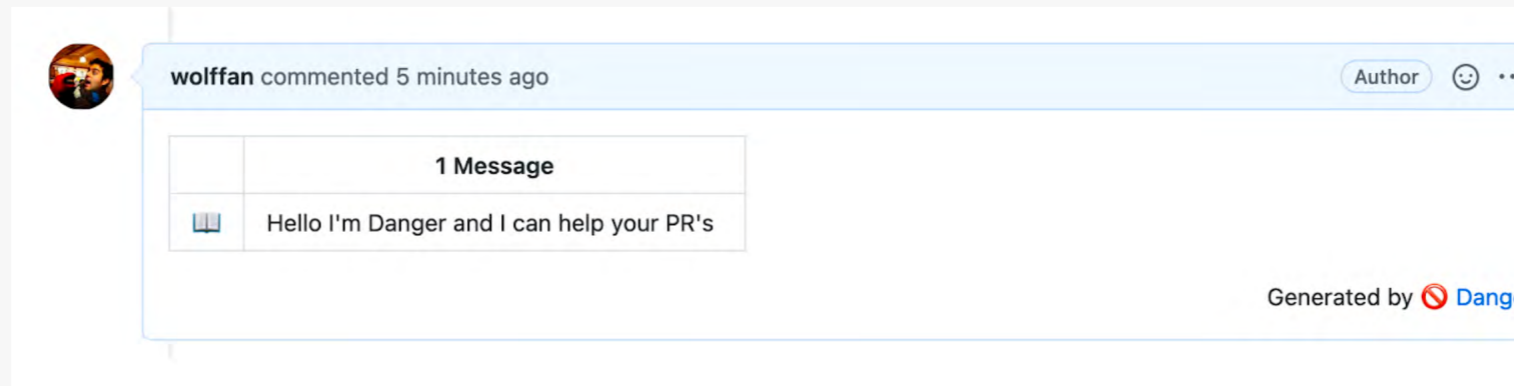
There are some additional concerns for CircleCI, which are covered in the [Danger Getting Started Guide](#).

Setting up tokens

If "Only build pull requests" cannot be enabled for your project, Danger *can* still work by relying on the CircleCI API to retrieve PR metadata. Using the API requires an API token.

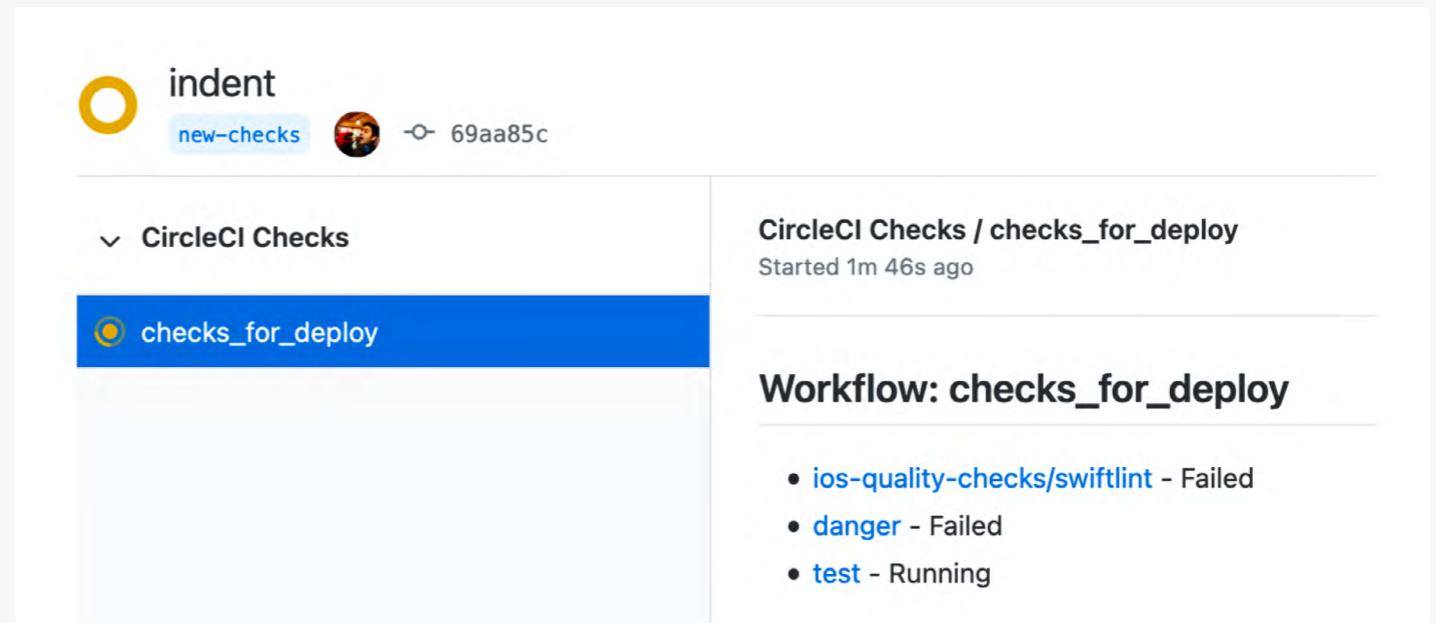
- Go to your project > Settings > API Permissions. Create a token with scope "view-builds" and a label like "DANGER_CIRCLE_CI_API_TOKEN".
- Settings > Environment Variables. Add the token as a CircleCI environment variable, which exposes it to the Danger process.

The result is just a message, not actual rules that were applied to this PR. Although I created a token for myself this time, I recommend creating a new user for the robot to give feedback.



Now we need to fix our code to have a happy linter. It is not best practice of course, but we could also just bend some rules to make it pass.

Notice that the GitHub workflow checks are shown. Which is helpful, because it gives you insight into what is happening without having to move from your PR.



The screenshot shows a GitHub pull request interface for a repository named 'indent'. The pull request is titled 'new-checks' and is associated with commit '69aa85c'. Under the 'CircleCI Checks' section, a workflow named 'checks_for_deploy' is highlighted. To the right, the details for this workflow are shown, including the title 'CircleCI Checks / checks_for_deploy', the start time 'Started 1m 46s ago', and the workflow name 'Workflow: checks_for_deploy'. The workflow status is 'Failed', and the failed steps are 'ios-quality-checks/swiftlint', 'danger', and 'test'.

indent
new-checks 69aa85c

CircleCI Checks

checks_for_deploy

CircleCI Checks / checks_for_deploy
Started 1m 46s ago

Workflow: checks_for_deploy

- ios-quality-checks/swiftlint - Failed
- danger - Failed
- test - Running

Configuring Danger

For configuring Danger, the [Getting Started](#) guide is a helpful resource.

We have already made some errors.

```
/root/project/wallet/ETHRadar/Chat.swift  
❌ Line 29: Force casts should be avoided.
```

Other errors show inside `/Pods`. Because we do not own this folder we will just create a `.swiftlint.yml` file and mark the `/Pods` folder as disabled for linting.

We also disabled some linting rules inside `keyboardExtension.swift` to show that by using `// swiftlint:disable force_cast` we can disable the linter for those lines. There is a closing `// swiftlint:enable force_cast`.

It is time to review our workflows. They also run in parallel to maximise speed.

Duration	Branch	Commit
🕒 4m 2s	🔗 new-checks	🔑 e01621b
🌩 test		3m 51s
✅ ios-quality-checks/swiftlint		20s
✅ danger		1m 25s

All checks are green and we can now move forward.

Note: We used a default version of lint and a basic version of Danger. These are complex tools that can support almost any workflow or rule you can imagine.

Creating functional tests

We have done some testing, but before we can ship our app, we need a bit more confidence that our application is indeed ready for users. We could write more unit and integration tests, but we can also add other types of tests to our repertoire.

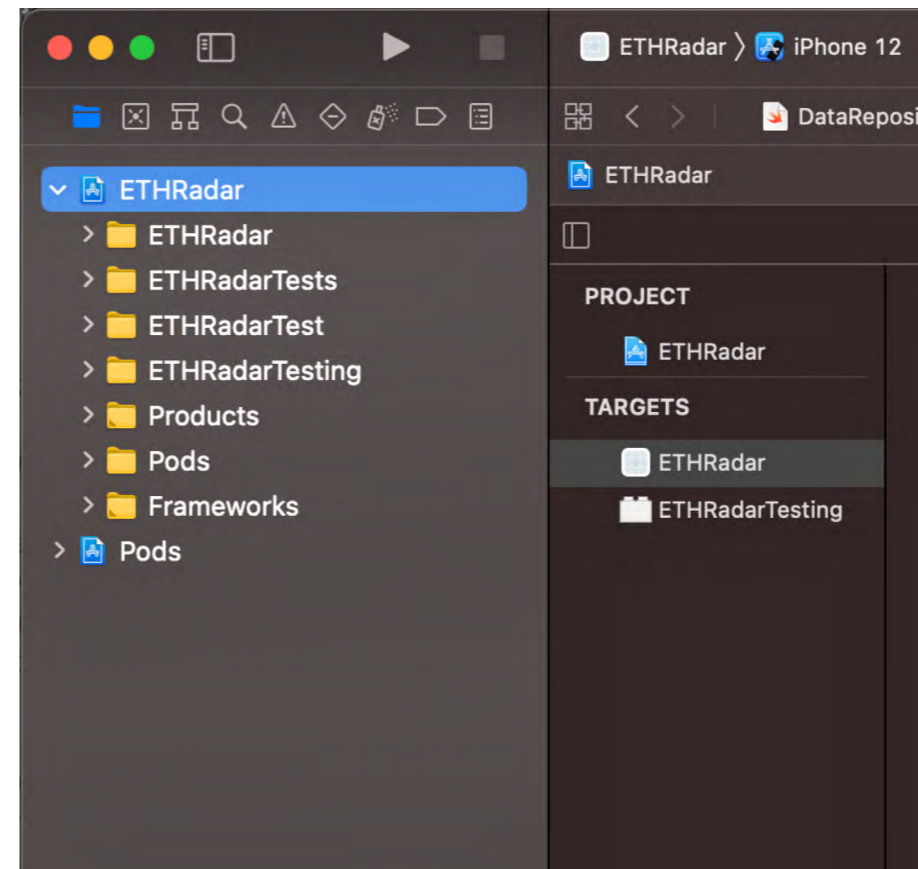
Unlike unit tests, which are small-scale tests for parts of our code, or integration tests, which are tests that interact with device features like geolocation, functional and UI tests ensure the app runs and functions as it should for an end user.

In the next example we will show how to create functional tests, using the tools Xcode provides by default. Not only can we simulate user touches, we can simulate text input and other interactions as well. We will inspect the screen, navigate to another screen, then go back to the original one, simulating what a user would do. As you can see, unlike unit or integration tests, functional or UI tests are less specific but allow a wide range of testing strategies.

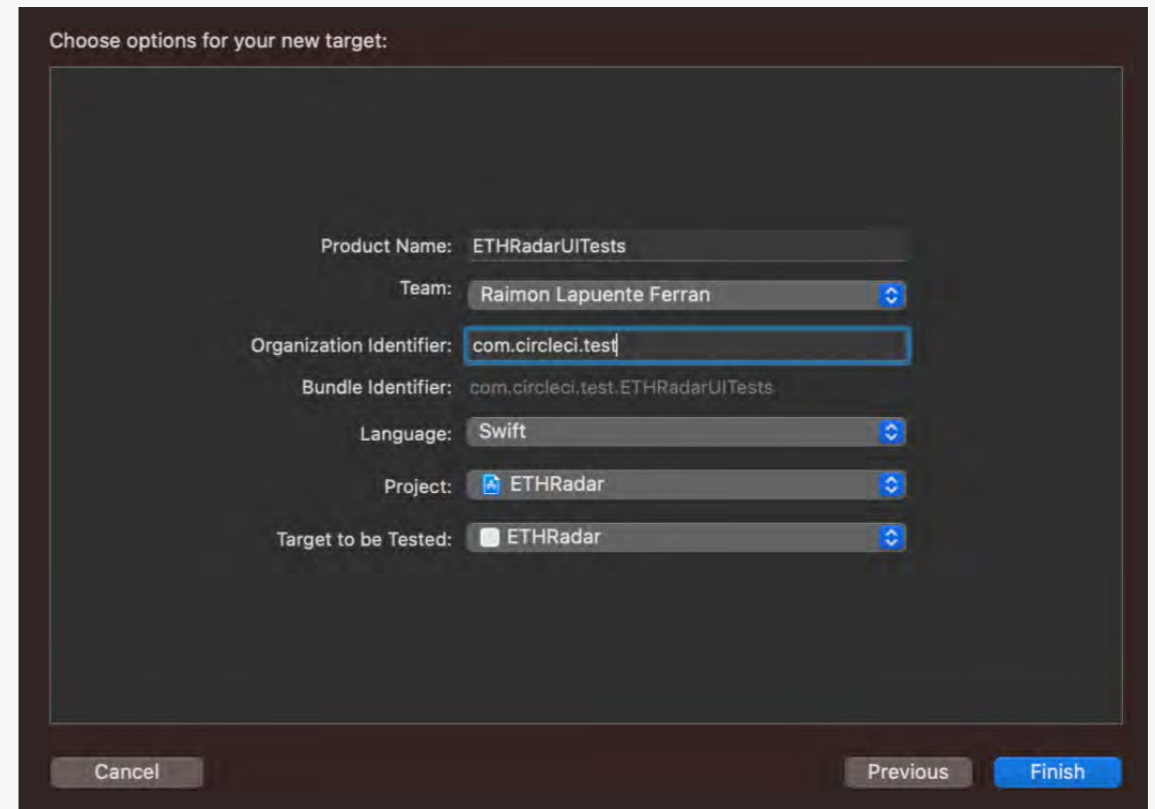
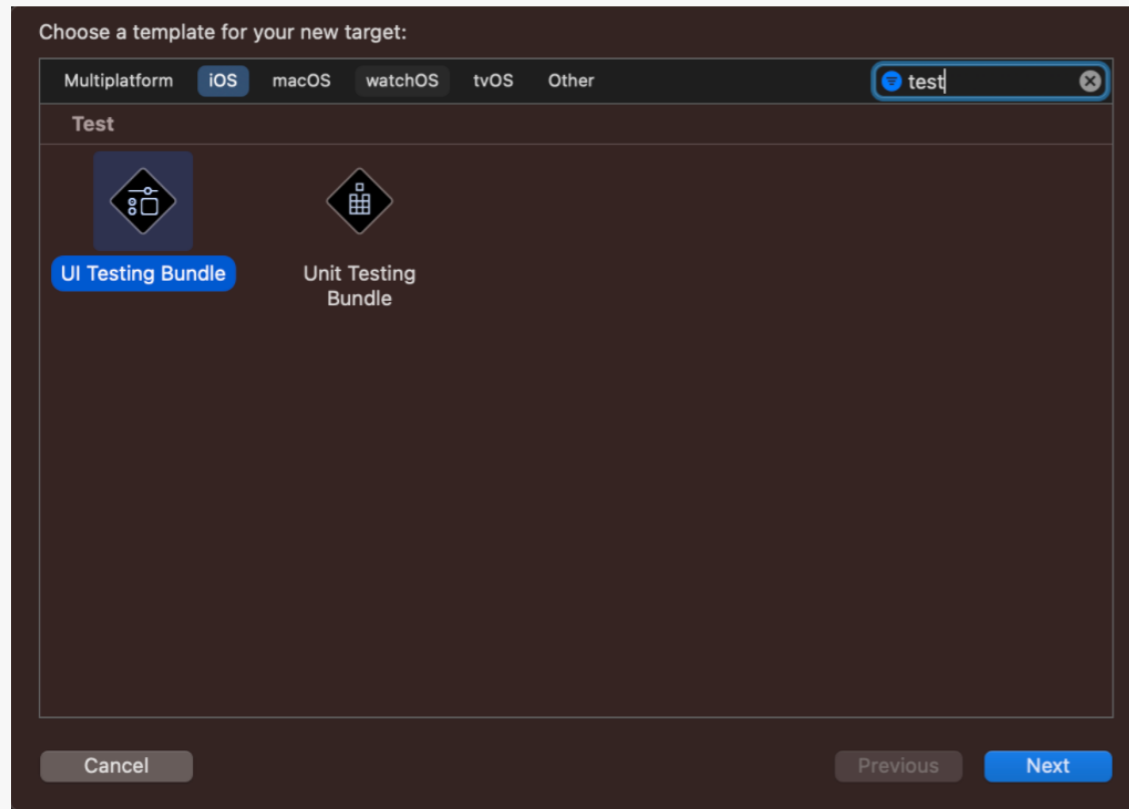
We will be using the UI test suite to hit the same endpoints that the app hits regularly. Because these endpoints belong to external services, we do not want to hit them for real. Stubbing solves this problem, and some Swift libraries allow you to inject json files into your tests by copy/pasting the network responses. Two of these libraries are [MockingJay](#) and [OHHTTPStubs](#).

Running the UI tests

To create the UI tests we need a new target specifically for them. The new target will allow us to run them separately, if we choose to, in a simpler way.



Scroll all the way down the Xcode window and Click the “+” button under in the Targets section. You will be prompted to select the kind of target. Then, search for a test. There will be 2 options to choose from; select the UI test option. Leave the default responses in the rest of the fields.

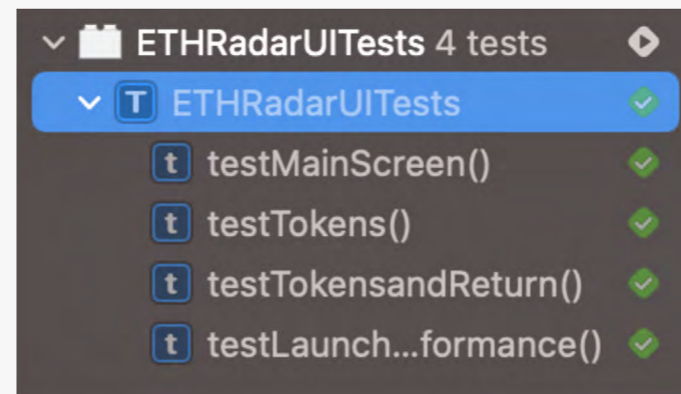


When we open the ETHRadarUITests file, we can create a new function or rename the example function. Keep the cursor inside the function and get ready for some magic. At the bottom left of the Xcode window, there is a **Record** button. Clicking **Record** launches our app, records our touches and transforms them into code. It is just that easy to automate functional testing. Give it some time to compile and launch the app, then watch as it records our touches and keyboard strokes.

And as if by magic, our tests have passed.

Note: Because it is not really magic, I had to do some working around a bit and some waiting for the external server response. For me, it was time and effort well spent.

```
25
26 func testMainScreen() throws {
27     // UI tests must launch the application that they test.
28     let app = XCUIApplication()
29     app.launch()
30
31 }
32
33
34 func testLaunchPerformance() throws {
35     if #available(macOS 10.15, iOS 13.0, tvOS 13.0, *) {
36         // This measures how long it takes to launch your application.
37         measure(metrics: [XCTApplicationLaunchMetric()]) {
38             XCUIApplication().launch()
39         }
40     }
41 }
42 }
43
```

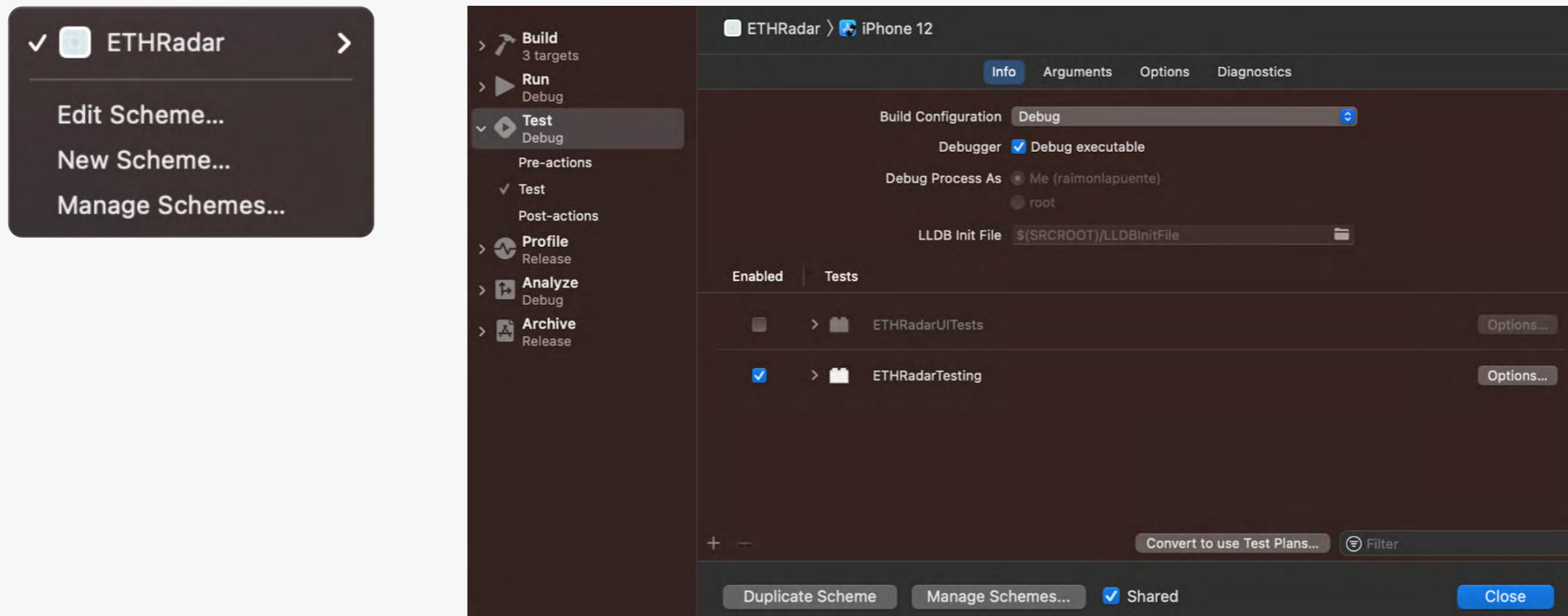


Adding new tests to CircleCI

Now we are ready to add a new step into our workflow in CircleCI. To do this, we will create a new scheme. Schemes are a way of building targets with particular conditions within a project. Our new scheme will effectively separate unit and integration testing from UI tests.

There are default schemes, and it might be helpful to understand a little bit about them.

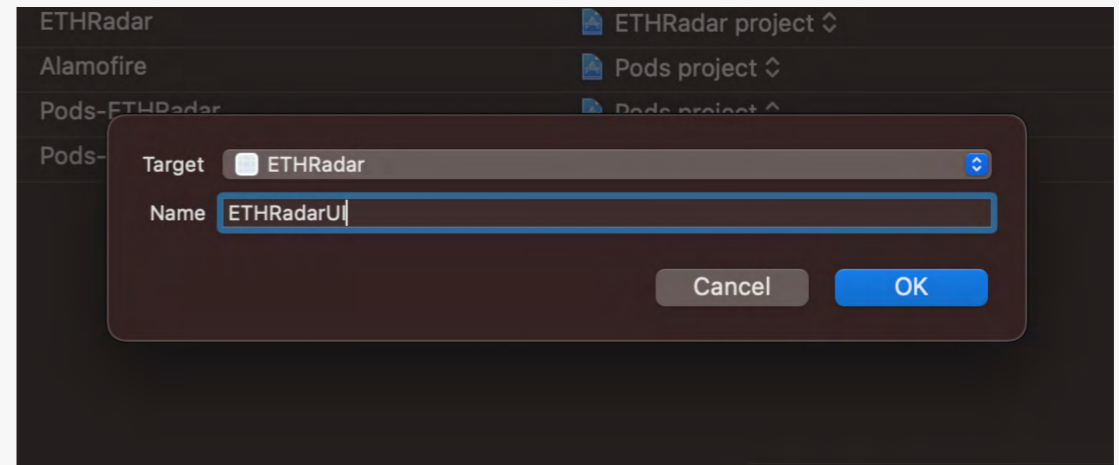
Click **Edit Scheme...** and navigate to **Test**. Our UITests target is not selected. We have 2 options: we can add the UITests target to the Tests scheme; or we can create a new scheme that runs just UITests. The second option is a cleaner solution, especially if you have more tests that the 4 we have in this example.



Create a new ETHRadarUI scheme just for our UITests target. test. And we Check the box to enable it. Add it by clicking the **+** button. From the dropdown at the top of the Xcode window, select **ETHRadarUI**. Press **CMD+U** to trigger the test.

For the automation on CircleCI, we need to:

- Create a copy of the test job in the circle.yml file
 - Rename the copied job **UITest**
- Change the fastlane command to use **UITest**
 - Create the fastlane job
- Go to the Fastfile inside wallet folder, and duplicate the test lane
 - Rename the copied lane to **UITest**
- Select the **ETHRadarUI** scheme



```
workflows:
  checks_for_deploy:
    jobs:
      - danger
      - ios-quality-checks/swiftlint:
          path: wallet/
      - test
      - UITest
        requires:
          - test
```

`config.yml`

```
lane :UITest do
  # add actions here: https://docs.fastlane.tools/actions
  scan(
    workspace: "ETHRadar.xcworkspace",
    scheme: "ETHRadarUI",
    clean: true,
    destination: "platform=iOS Simulator,name=iPhone 12,OS=14.4"
  )
end
```

`Fastfile`

We want the test to run only if the previous tests pass, so we will need to specify on our workflow that UITests depends on the original Test scheme.

And if we head over to CircleCI what should expect something like this.



Before closing the section there is a last bit you might be interested in. We want to store our test results in CircleCI, because it will be easier for us to monitor our test results. We need to update the fastlane scan script to include ``junit`` format for test results, code coverage, and output folder. For the config.yml we want to flag that folder as a folder with the test results.

Return to CircleCI and you will discover that our tests are being recognised and shown in the UI.

The image shows a screenshot of the CircleCI interface for a test job. At the top, the job name 'test' is displayed next to a green 'Success' badge. Below this, a table provides details about the test execution:

Duration / Finished	Queued	Executor
4m 5s / 5m ago	0s	macOS Medium ⓘ

At the bottom of the screenshot, there are three tabs: 'STEPS', 'TESTS 9', and 'ARTIFACTS'. The 'TESTS 9' tab is currently selected and highlighted with a green background.

Deploying with CircleCI and fastlane

Our sample project has reached its final phase. Our last task is to configure the delivery of our application using fastlane and CircleCI. We began with nothing more than an idea, and we are nearly ready to upload our app for beta distribution.

Although much of the process can be automated from Xcode, it is a good idea to familiarise yourself with the basics of Apple code signing by reviewing this [guide](#).

Why use a different tool than Xcode?

Working as a team, it is harder to share credentials easily using Xcode, and also, you may not want to give everyone the same level of credentials. Xcode is not great for CI and many CI systems will need more flexibility and power. Many people prefer a tool that streamlines the certificate/security language and process, which Xcode does not do. As a result, we choose to use fastlane, as do many other teams. Some really big teams may have a custom solution, but for most teams, fastlane is a great choice.

The 3 things we need to get started are:

- Private key
- Certificates
- Provisioning profiles

These items are required to sign the app. They have a complicated relationship with each other, so managing this process in a big project can be cumbersome. Luckily, fastlane has a tool (`match`) to help us automate the process.

Note: fastlane recommends that you use an account created exclusively for this purpose. They do not recommend using a regular account.

In the `wallet/` folder, start with `bundle exec fastlane match` and follow the instructions in this tutorial. You will need to keep your Apple credentials handy, because the process involves creating the files mentioned earlier, and storing them on a shared git repository. Later CircleCI or any developer can pull those files from the repository to build and ship the app. More than one tool can be used to store those credentials: git, s3, or google Cloud.`

Now that fastlane has set up Matchfile we can execute `bundle exec fastlane match development` to create development certs. Continue to follow the fastlane instructions. To create the development certs, you will need to give a passphrase to the repository for security. Make sure to store that passphrase properly. In our case, we will use the passphrase “testCircle”.

Because our app does not exist yet, fastlane will complain. There are other fastlane commands we can use instead that will allow us to complete this task.

Type `bundle exec fastlane produce`. Remember that any fastlane interaction with the store will require your credentials. You can specify them on the Matchfile.

Note: fastlane produce caused an error, so we ended up manually creating an app on the portal. It did create the bundle, though.

```
[12:36:03]: Cloning remote git repo...
[12:36:03]: If cloning the repo takes too long, you can use the `clone_branch_directly` option in match.
[12:36:04]: Checking out branch master...
[12:36:04]: Enter the passphrase that should be used to encrypt/decrypt your certificates
[12:36:04]: This passphrase is specific per repository and will be stored in your local keychain
[12:36:04]: Make sure to remember the password, as you'll need it when you run match on a different machine
[12:36:04]: Passphrase for Match storage: *****
[12:45:21]: Type passphrase again: *****
[12:45:27]: 🗝️ Successfully decrypted certificates repo
[12:45:27]: Verifying that the certificate and profile are still valid on the Dev Portal...

-----
Please provide your Apple Developer Program account credentials
The login information you enter will be stored in your macOS Keychain
You can also pass the password using the `FASTLANE_PASSWORD` environment variable
See more information about it on GitHub: https://github.com/fastlane/fastlane/tree/master/credentials_manager
-----
Username: █
```

```
[13:08:42]: To not be asked about this value, you can specify it using 'app_identifier'
[13:08:42]: App Identifier (Bundle ID, e.g. com.krausefx.app): com.circleci.wallet
[13:09:14]: To not be asked about this value, you can specify it using 'app_name'
[13:09:14]: App Name: Circle test
[13:09:21]: Creating new app 'Circle test' on the Apple Dev Center
[13:09:22]: Created app 77VC6MK32W
[13:09:23]: Finished creating new app 'Circle test' on the Dev Center
[13:09:25]: Creating new app 'Circle test' on App Store Connect
```

Remember to monitor the instructions from the terminal. If you are not specifying your details on Matchfile, you will be prompted to enter the same information multiple times.

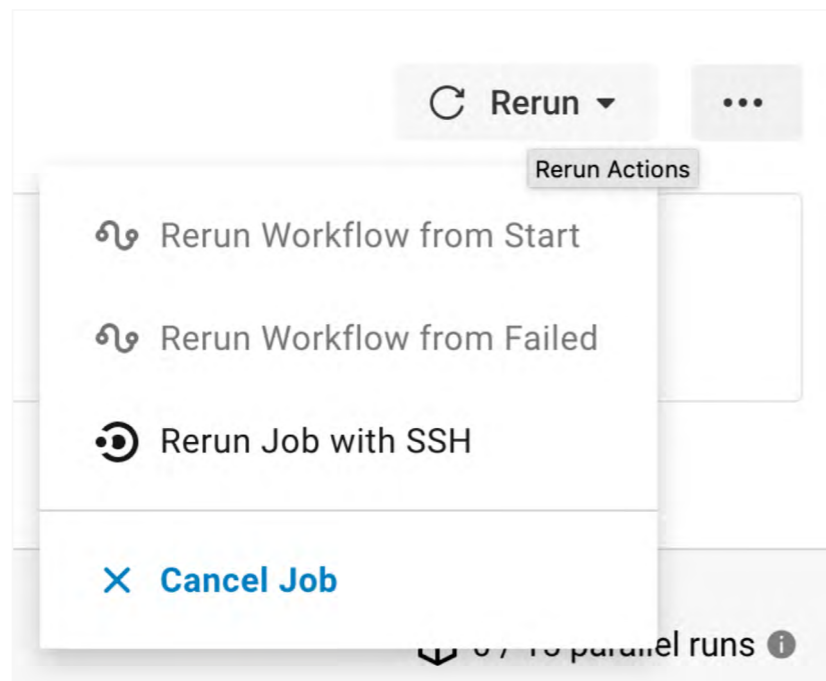
The result of this process is that all keys are created now and are available to use.

Now that we have the certificates ready, go back to the Fastfile. Create a new line and use the fastlane tools sigh and cert to build the app for beta distribution.

```
[13:14:05]: Verifying certificates...
[13:14:05]: No existing profiles found, that match the certificates you have installed locally! Creating a new provisioning profile for you
[13:14:06]: Creating new provisioning profile for 'com.circleci.wallet' with name 'match Development com.circleci.wallet' for 'ios' platform
[13:14:14]: Downloading provisioning profile...
[13:14:14]: Successfully downloaded provisioning profile...
[13:14:14]: Installing provisioning profile...
/var/folders/wt/nhnsjd0953993dh_7drzmnzw0000gn/T/d20210328-52283-p8j2cv/profiles/development/Development_com.circleci.wallet.mobileprovision
[13:14:14]: Installing provisioning profile...
[13:14:15]: 🔒 Successfully encrypted certificates repo
[13:14:15]: Pushing changes to remote git repo...
[13:14:17]: Finished uploading files to Git Repo [https://github.com/CI-CD-for-iOS/matchRepo.git]
```

Debugging failed jobs

It seems like something is not right, so we will take this opportunity to use a handy feature of CircleCI: ssh debugging. When you rerun with SSH on the UI, you can ssh into the instance to find out exactly what is happening so you can troubleshoot on the fly.



To use ssh to debug this job and find out why it failed, we need an ssh key.

`ls -al ~/.ssh` will reveal if your machine already has a key. If there is an ssh key, there will be a file named `id_rsa.pub`.

If you do not have an ssh key, [follow these instructions](#).

When you rerun the job with ssh, a message box will pop up.

You can now SSH into this box if your SSH public key is added:

```
$ ssh
```

Enter:

```
$ ssh -p 54782 162.221.90.194
```

Running this command grants us access to the machine running the job. Once we ssh in, our code will be inside the project folder. You can do whatever you would on your local machine. Go ahead and run `fastlane` inside that ssh machine. You will discover that we are missing some bits from `fastlane` certificate tools in our configuration. From previous experience, I am guessing that we are most likely missing some env vars.

```

lane :beta do
  match(
    type: "appstore",
    readonly: true,
    git_basic_authorization:
Base64.strict_encode64("r.lapuente@gmail.com:84e9c2bd398d27d8af8ba11c57
e0d3c9082fbf25"),
    generate_apple_certs: false,
    verbose: true
  )

  gym(
    workspace: "ETHRadar.xcworkspace",
    scheme: "ETHRadar",
    export_method: "app-store",
    output_directory: "binaries",
    output_name: "CircleTest.ipa",
    verbose: true,
    configuration: "Release"
  )

  api_key = app_store_connect_api_key(
    key_id: "75LQYZ4AY9",
    issuer_id: "69a6de79-b776-47e3-e053-5b8c7c11a4d1",
    key_filepath: "../AuthKey_75LQYZ4AY9.p8",
    duration: 1200, # optional (maximum 1200)
  )

```

SSH debugging is great, but it is not as good as new 2FA requirements from Apple for access to some APIs. To create an app store connect API key and use it in fastlane, use this [guide](#).

When I arrived at this point the first time I built this project, something was still not right, because match could not import the repo. To avoid this, [validate your match configuration](#) with CircleCI.

Note: When using `match` inside CircleCI it is better to use the ssh url on your Matchfile. In the development computer I used, https was better while working locally on the computer. There is no downside to changing the urls while you are developing. Just make sure the ssh url is checked into the repo for CircleCI.

```
    in_house: false # optional but may be required if using match/sigh
  )
  upload_to_testflight(
    api_key: api_key,
    skip_waiting_for_build_processing: true)
end
```

After the key is uploaded, verify that the Fastfile will correctly:

- Download the certs using match
- Build the app
- Upload to testflight

```
ship:
  macos:
    xcode: 12.4.0
  steps:
    - checkout
    - run:
        name: Install dependencies
        command: cd wallet && bundle install
    - run:
        name: Build and ship
        command: cd wallet && bundle exec fastlane beta
```

We will also want to store the binaries in CircleCI so that they are available to download manually.

When things go well we can observe the different steps as they run; downloading the certificates, building the app, then archiving the app. As a bonus, the ipa we are generating will be stored in the “binaries” folder, so it will be captured by CircleCI and stored for us.

```
Successfully exported and compressed dSYM file
Successfully exported and signed the ipa file:
```

```
-----
--- Step: upload_to_testflight ---
-----
Creating authorization token for App Store Connect API
Ready to upload new build to TestFlight (App: 1560574870)...
Going to upload updated app to App Store Connect
This might take a few minutes. Please don't interrupt the script.
```

Because we created only a development certificate with match, the Testflight will fail. We need to call `bundle exec fastlane match appstore` to generate certificates that can go to Testflight. We then need to call the beta lane again with `fastlane bundle exec fastlane beta`. Of course, Testflight is always a bit picky so we need to add the icon.

Finally, after all that work we successfully shipped our app to testflight! And our process will ship the app every time we complete a full testing cycle. The important part to remember is that we don't “need” to use that shipped app. We can ship as many apps as we want, and choose what versions we make available to the end users or testers.

We are able to submit as many versions as we want to Apple. In fact, it is a good practice to actually ship things daily, and we could do that with a CircleCI job that runs at night or on every PR merge (a topic for another book). Once we have CircleCI set up and the scripts clear, we can mix and match with all the power offered by the tools we are using.

App Store Connect

Dear Raimon Lapuente,

The following build has completed processing:

Platform: iOS
App Name: Circle test 2
Build Number: 1
Version Number: 1.0
App SKU: 12345
App Apple ID: 1560574870

You can now use this build for TestFlight testing or submit it to the App Store.

If you have any questions regarding your app, click [Contact Us](#) in App Store Connect.

Regards,

The App Store team


```
-----  
--- Step: upload_to_testflight ---  
-----  
Creating authorization token for App Store Connect API  
Ready to upload new build to TestFlight (App: 1560574870)...  
Going to upload updated app to App Store Connect  
This might take a few minutes. Please don't interrupt the script.  
iTunes Transporter successfully finished its job  
-----  
--- Successfully uploaded package to App Store Connect. It might take a few minutes until it's visible online. ---  
-----
```

Note: The example project we are using has some differences than what is documented. Because we ported an older Xcode project it is possible that some unexpected problems from Xcode 8 or 9 were carried over. My solution was to disable Automatic signing, and instead, pick the default Provisioning profile created by match. That made the build successful for me. There are still some limitations on the key we created distributing automatically to testers, but the app is uploaded.

```
INFO [2021-03-30 08:07:58.52]: -----  
INFO [2021-03-30 08:07:58.52]: Successfully uploaded package to App Store Connect. It might take a few minutes until it's visible online.  
INFO [2021-03-30 08:07:58.52]: -----  
INFO [2021-03-30 08:07:58.52]: Successfully uploaded the new binary to App Store Connect  
INFO [2021-03-30 08:07:58.52]: If you want to skip waiting for the processing to be finished, use the `skip_waiting_for_build_processing` option  
INFO [2021-03-30 08:07:58.52]: Note that if `skip_waiting_for_build_processing` is used but a `changelog` is supplied, this process will wait for the build t  
DEBUG [2021-03-30 08:07:58.52]: App Platform (ios)  
INFO [2021-03-30 08:07:58.54]: Waiting for processing on... app_id: 1560574870, app_version: 1.0, build_version: 1, platform: IOS  
WARN [2021-03-30 08:07:58.89]: Read more information on why this build isn't showing up yet - https://github.com/fastlane/fastlane/issues/14997  
INFO [2021-03-30 08:07:58.89]: Waiting for the build to show up in the build list - this may take a few minutes (check your email for processing issues if th  
INFO [2021-03-30 08:08:29.28]: Waiting for the build to show up in the build list - this may take a few minutes (check your email for processing issues if th  
INFO [2021-03-30 08:08:59.80]: Waiting for App Store Connect to finish processing the new build (1.0 - 1) for IOS  
INFO [2021-03-30 08:09:30.73]: Waiting for App Store Connect to finish processing the new build (1.0 - 1) for IOS  
INFO [2021-03-30 08:10:02.00]: Successfully finished processing the build 1.0 - 1 for IOS  
WARN [2021-03-30 08:10:02.00]: Using App Store Connect's default for notifying external testers (which is true) - set `notify_external_testers` for full cont  
INFO [2021-03-30 08:10:02.00]: Distributing new build to testers: 1.0 - 1
```

Learning more about your project with Insights

The Insights dashboard is a great way to understand more about your project, and it is provided by default as a part of CircleCI. The dashboard shows an overview of the health and usage statistics of repository build processes like credit usage, success rates, pipeline duration, and other pertinent information. What do you have to do? Nothing. Insights does it for you.

Enter a workflow name to discover data about time and resources used (\$\$ credits). Test insights show you potentially bad tests and areas to improve your project. This valuable information can reduce costs and help you make better engineering decisions, without the additional cost of using another external service to collect and analyze it.

The screenshot displays the CircleCI Insights dashboard. At the top, there are four summary cards: 'Workflow runs' (63), 'Total workflow duration' (2h 14m 52s), 'Total credits consumed' (6,881), and 'Overall success rate' (21%). Below this is a table of workflow runs with columns for Project, Workflow, Total Credits, Total Duration, Runs, and Success Rate. The 'Top 10 most failed tests' section shows a message that all tests passed. The 'Top 10 slowest tests' section lists tests with their job names, durations, runs, and success rates.

PROJECT	WORKFLOW	TOTAL CREDITS	TOTAL DURATION	RUNS	SUCCESS RATE
guide-test	welcome	10	56s	3	100%
guide-test	Build Error	40	4m 22s	22	0%
guide-test	workflow	3,899	1h 20m 50s	26	31%
guide-test	checks_for_deploy	2,932	48m 44s	12	17%

TEST NAME	JOB	DURATION(P95)	RUNS	SUCCESS RATE
testLaunchPerformance	UITest	37s	1	100%
testMainScreen	UITest	6s	1	100%
testTokens	UITest	3s	1	100%
testTokensandReturn	UITest	3s	1	100%
testPerformanceExample	test	666 ms	1	100%
testExample	test	16 ms	1	100%

Conclusion

Let me take a moment to recap our progress. We started this experiment by learning just the basics — what CI and CD are and why they are important. We explored the reasons that testing is so important to this process.

We set up a sample iOS application on CircleCI and Github, and configured fastlane for launching tests. We created a suite of tests that includes unit tests, integration tests, and UI tests, which give us speed, precision, and stability.

We discovered the secrets to deploying with CircleCI and fastlane, creating the certificates needed to sign and ship the app to the AppStore. We learned how to debug failed jobs from the command line, and how to gain actionable insights about why those jobs failed. As we've seen, this does not have to be difficult. It makes sense to start mobile app projects the right way: with testing and CI/CD built in from the beginning. Our CI/CD process keeps things ready for teams to work together doing what they do best, and letting the CI system automate the most essential tasks. We can ship apps with increased confidence and significantly reduce the amount of time we dedicate to repetitive tasks.

Before we wrap up, here are some useful resources that could help you begin your continuous integration journey:

- [Mobile and browser testing on CircleCI: simple setup, easy management, increased confidence](#)
- [Mobile integrations with orbs](#)
- [Hello World examples](#)
- [Sample config.yml files](#)