The Details: CI

Standardized testing for your code

Justin
Dec 1, 2021

15 8

Hello there, lovely subscribers. This week's post is going to dive deeper into **CI**. We're going to look at what types of tests developers run in CI, how they integrate with version control systems, and where teams set up CI (on your own server, or in the cloud).

The first "Details" post covered ETL, and the most recent one covered data warehouses. As a reminder, if there are any topics you want to see me write about, just let me know! These details posts are for paid subscribers only.

Refresher: What is CI?

Continuous Integration and Continuous Delivery (big words) are all about getting new code out into the wild as quickly and as securely as possible. CI is about the *secure* part – making sure that code is well tested and is going to perform on the stage exactly how you expect it to (or at least as close as we can get it).

Back in the day – especially before cloud delivered software became the norm – software upgrades were infrequent, difficult, and risky. It was hard to know if the new feature you built might cause something to break incidentally, and logistically, getting that new feature out to your users required some degree of coordination. Because of that, companies would *batch* new features and bug fixes together into big new releases.

As we started moving software to the cloud though, that all changed – upgrades started happening **automatically and without notification**. For example, when you're using Gmail in your web browser, you'll notice that Google is *constantly* changing and improving it, from how it looks to how you navigate around and even behind the scenes performance changes. You don't click "update," the stuff just continuously (see?) shows up.

> 🚨 **Confusion Alert** 🚨
>
> Plenty of software still follows more traditional update models. For example, when you want to update iOS on your phone or MacOS on your laptop, you need to actively choose to download and apply the update. Plus, Apple bundles new features and bug fixes together. The same is true for updating the apps on your phone. For cloud software though – especially B2B – the standard is becoming this permissionless update cycle.
>
> 🚨 **Confusion Alert** 🚨

This shift was powerful for software companies – it let them iterate extremely quickly on the product, get new features out into the wild near instantaneously, and gather product feedback through experimentation.

What *enables* this lightning quick delivery is a new approach to testing and deploying software, or what you've been hearing about with this CI/CD stuff. In a nutshell:

- **CI** is the process of extensively testing every new unit of code you want to deploy, automatically and quickly
- **CD** is the process of getting your new, now-tested in the hands of your users automatically and quickly

Another way of thinking about CI/CD is as a process of **automation**: the manual testing and deployment that used to take so much engineer time is now making its way into code and SaaS. In this post, we're going to dive deeper into both CI and CD, what they entail, and what engineering teams use to do them.

This post is going to go deeper into the CI part; but keep in mind that the two are intertwined.

Types of tests: what CI is running

An engineer's nightmare: after working on an exciting new feature for a week and quickly getting it out to users, bug reports start to roll in. It turns out the new feature broke an existing workflow and it's causing users to lose data (😨).

For anyone who has worked at a startup, you already know how common this is; CI is aimed at fixing this exact problem.

As engineering teams and codebases mature, they add in more and more tests to make sure that things will work as expected. That way, you can catch these errors *before* they hit your users. Of course, you'll never be able to catch *everything* – bugs are a fundamental reality of software, whether we like it or not. But tests can help.

So what exactly are these tests we're talking about? A few examples:

1. **Unit tests**

Unit tests are the broadest category, and refer to tests that check that a unit of code is doing what it's expected to. A unit test can be something as small as verifying that a function returns an intended data type, or as large as verifying the structure of an object generated from a set of functions. The philosophy behind them is ensuring that each logical unit of code produces exactly what you expect it to, and that keeping these tests low level makes it easier to understand what went wrong if the whole thing doesn't work.

2. **Linting and code rules**

These are tests that check code formatting and style. Developers want to maintain consistency across the codebase, and linters help enforce stylistic rules. Some common examples of linting rules are not allowing trailing spaces, setting a maximum number of characters per line of code, and removing any unused library imports at the top of files. Open source packages like ESLint for JavaScript allow engineers to set codebase-wide linting rules that you can enforce in CI.

3. **Misc. other tests**

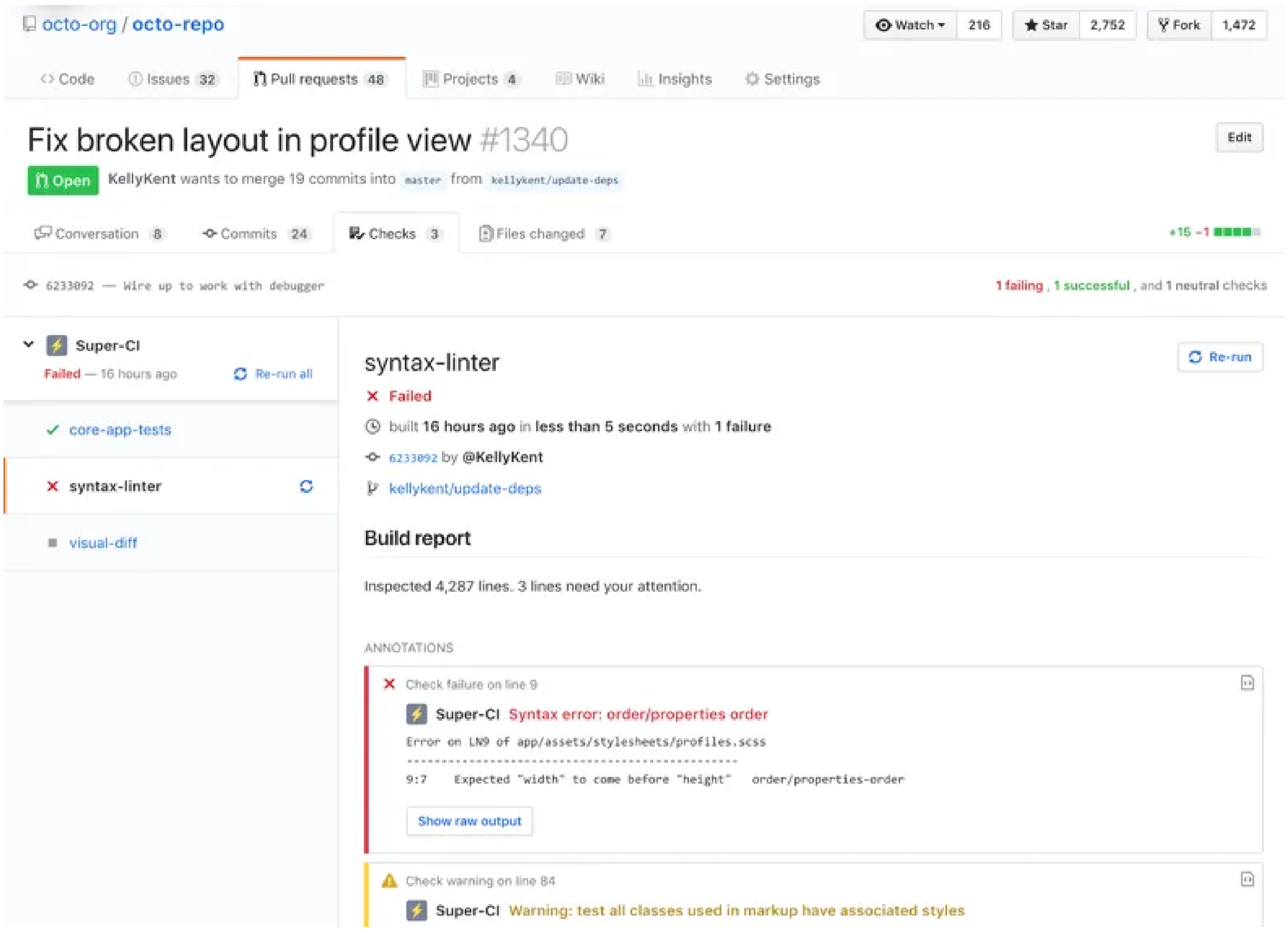
Depending on what your app looks like and what its unique constraints are, there are a variety of other tests that engineers implement as part of CI. One example is frontend tests: making sure the actual UI of your application looks and performs as expected, often achieved through something like Storyboard. Some tests try to mimic a user interacting with the software, and what clicks or inputs they might make: frameworks like Cypress help developers set these kinds of declarative tests ("do this, do that, should result in this") in their code.

CI runs on every Pull Request

Logistically, CI gets triggered every time a developer wants to merge in new code. In most cases, this means a Pull Request in GitHub (or whatever version control system you're using). A Pull Request is a GitHub concept that basically means, hey, I have new code and I want to merge it into the codebase. Each Pull Request gets its own name, description, and set of comments (your fellow developers telling you your code is shit). For more info on this workflow, check out the Technically post about how a feature gets shipped.

> 🔍 **Deeper Look** 🔍
>
> One of the main reasons that these checks get run once a Pull Request is opened – as opposed to locally, as a developer builds the feature – is that tests need to run in the *environment* that this software is eventually going to be deployed in. As such, part of CI is building that environment (spinning up the web server, starting up Docker, etc.). That being said, many tests – like linting – can and are run locally first.
>
> 🔍 **Deeper Look** 🔍

In new PRs (Pull Requests), you'll see a section for "checks" – these are your CI tests. After setting up your CI server to integrate with GitHub (more on that in a bit), it will run them automatically whenever you open a PR. GitHub provides a nice UI for showing the status and results of each test, powered by a public checks API behind the scenes.



Fix broken layout in profile view #1340

KellyKent wants to merge 19 commits into `master` from `kellykent/update-deps`

```
6233092 — Wire up to work with debugger
```

1 failing, 1 successful, and 1 neutral checks

Super-CI
Failed — 16 hours ago

syntax-linter
Failed
built 16 hours ago in less than 5 seconds with 1 failure
6233092 by @KellyKent
kellykent/update-deps

Build report

Inspected 4,287 lines. 3 lines need your attention.

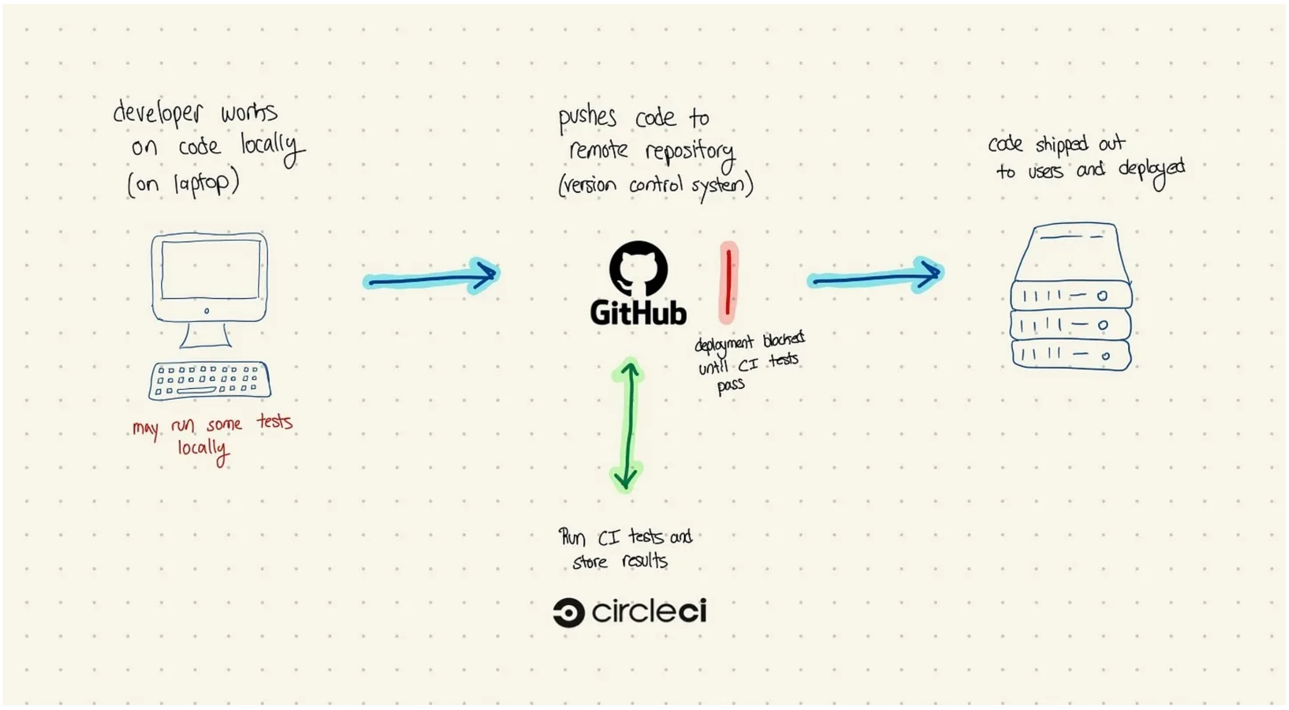
ANNOTATIONS

Check failure on line 9
Super-CI Syntax error: order/properties order

```
Error on LN9 of app/assets/stylesheets/profiles.scss
--------------------------------------------------
9:7    Expected "width" to come before "height"   order/properties-order
```

Check warning on line 84
Super-CI Warning: test all classes used in markup have associated styles

If the test passes, great – if not, and you designed it properly, it should give you a useful error message. In the above screenshot, it looks like a CSS test failed because the author of this code put a “height” property before a “width” one – and example of a linting test from above. With that information in hand, the developer will fix the problem, commit that new fix to the PR, and the checks will run again (and pass, hopefully).


developer works on code locally (on laptop)
may run some tests locally
pushes code to remote repository (version control system)
GitHub
deployment blocked until CI tests pass
Run CI tests and store results
circleci
code shipped out to users and deployed

By the time there's a good number of engineers working at a company, you might have 10+ of these groups of tests running on every PR. Inevitably, you run into "flaky tests" that seem to fail for no reason, logistical issues with tests not getting triggered, and other associated issues; in other words, you can have bugs in your CI too!

You need a server to run your CI

While your CI tests might get *triggered* by a Pull Request in GitHub, they're not *running* in GitHub (unless you're using their proprietary CI product, GitHub Actions). Instead, they need a standalone server to run on, and you can either set that up yourself or pay someone else to do it.

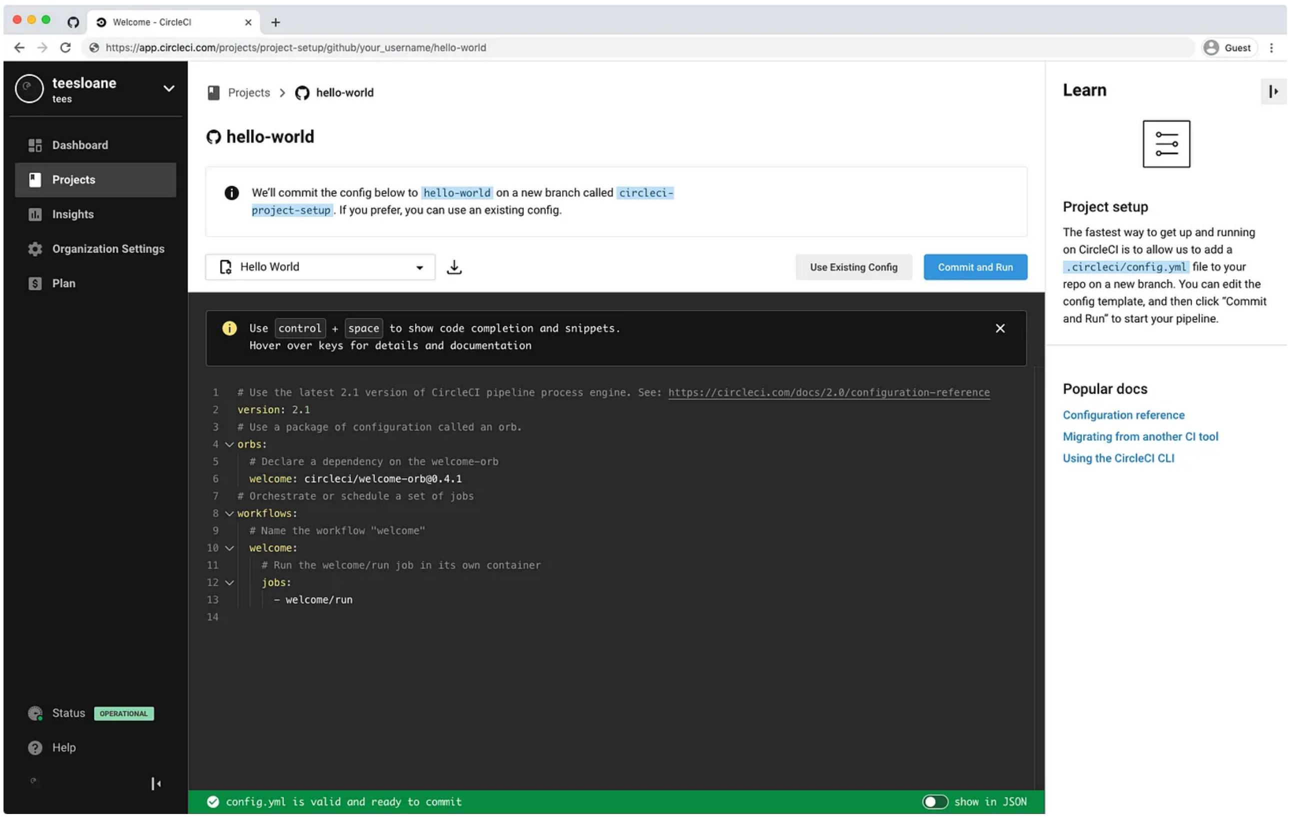
1. **Home grown CI servers**

For companies with high security requirements or deeply complex CI flows, they may choose to stand up their own server to run tests from. You can build *everything* from scratch, or use an open source product like Jenkins to build out some of the basic building blocks like automation flows and plugins for connecting to your version control system. You'll also need to set up a database to track the results of your tests so you can refer back to them on success / failure.

2. **Cloud solutions**

Most startups use a cloud hosted CI service instead of setting up their own servers. There are a bunch of these: [TravisCI](), [CircleCI](), [Harness](), [JFrog](), etc. The big cloud providers also have their own options, like [Azure Pipelines]().

These cloud solutions usually have two sides to them: the server itself for running your CI tests, and a nice web UI for integrating with version control and visually building pipelines. In CircleCI, you can write your test code in their UI, or pull it in from an existing repository:



We'll commit the config below to `hello-world` on a new branch called `circleci-project-setup`. If you prefer, you can use an existing config.

The fastest way to get up and running on CircleCI is to allow us to add a `.circleci/config.yml` file to your repo on a new branch. You can edit the config template, and then click "Commit and Run" to start your pipeline.

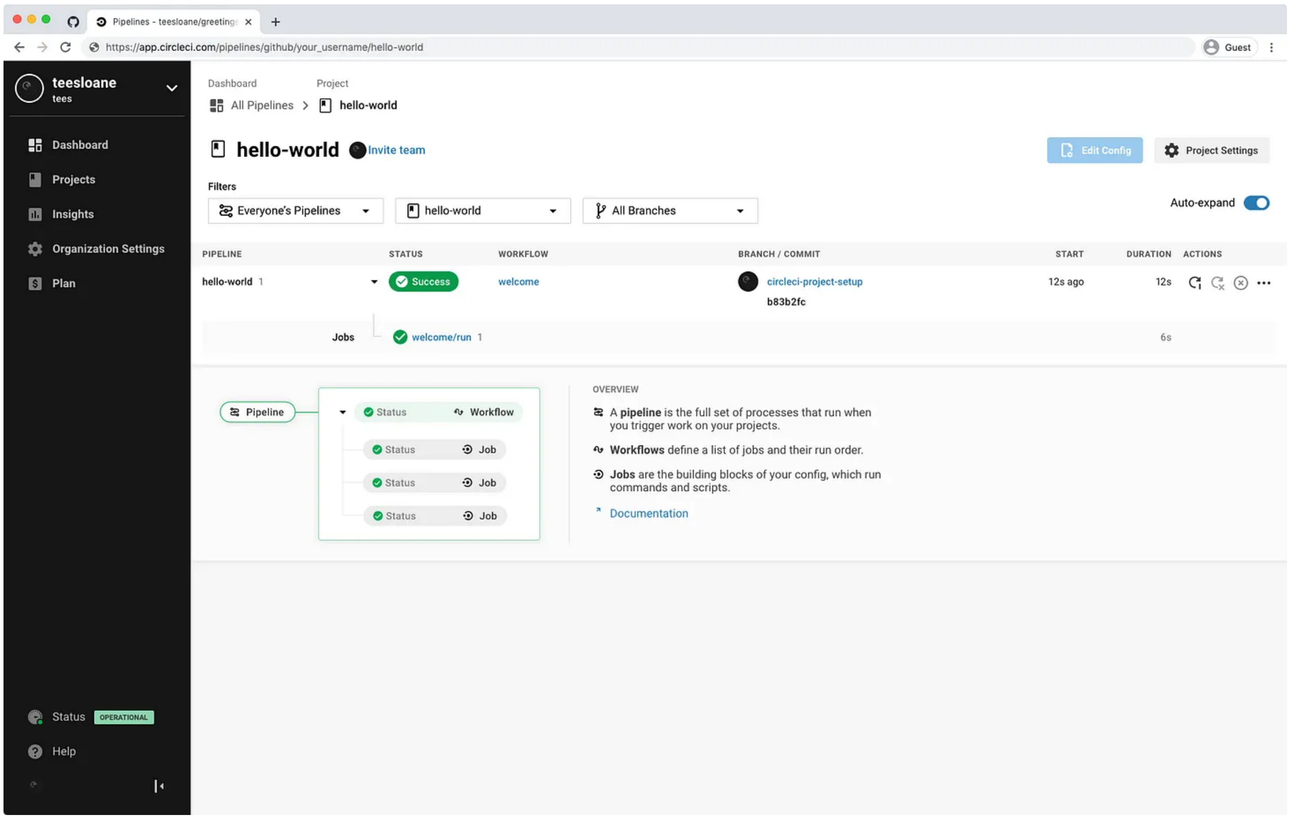
Use `control` + `space` to show code completion and snippets.
Hover over keys for details and documentation

```yaml
# 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

```

config.yml is valid and ready to commit

Behind the scenes, CircleCI is storing the results of every test run in addition to actually running those tests. That's really convenient because it allows engineering teams to look back on why a test failed, and see basics on error messages and the like. There's also a nice UI on top of that data for exploring it:



PIPELINE	STATUS	WORKFLOW	BRANCH / COMMIT	START	DURATION	ACTIONS
hello-world 1	Success	welcome	circleci-project-setup b83b2fc	12s ago	12s	
Jobs	welcome/run 1				6s	

OVERVIEW

A **pipeline** is the full set of processes that run when you trigger work on your projects.

Workflows define a list of jobs and their run order.

Jobs are the building blocks of your config, which run commands and scripts.

Documentation

The way to think about what CircleCI and other cloud hosted CI solutions provide is like this: you still need to write the code that tests your code (code on code), but CircleCI takes care of all of the *management* around that code. That includes a UI for setting things up and configuration, a database for storing pipeline results, and connections to popular version control systems.

8 Comments

Write a comment...

Mariana Oliveira Jan 4, 2022 Liked by Justin

Hi Justin, great article! Super helpful to understand these bits and pieces as a PM so I can make sure my team is running smoothly and identify any bottlenecks/opportunities they might be failing to communicate.

PS:

I think there's a small typo where you meant to write "Storybook" (it says Storyboard instead).

2 Reply Collapse •••

Tuan-Anh Dec 15, 2021 Liked by Justin

Hi Justin, would be great if you could write an article on data modeling in data warehouse, such as Kimball modeling dim, fact, cube, etc.

2 Reply Gift a subscription Collapse •••

6 more comments...