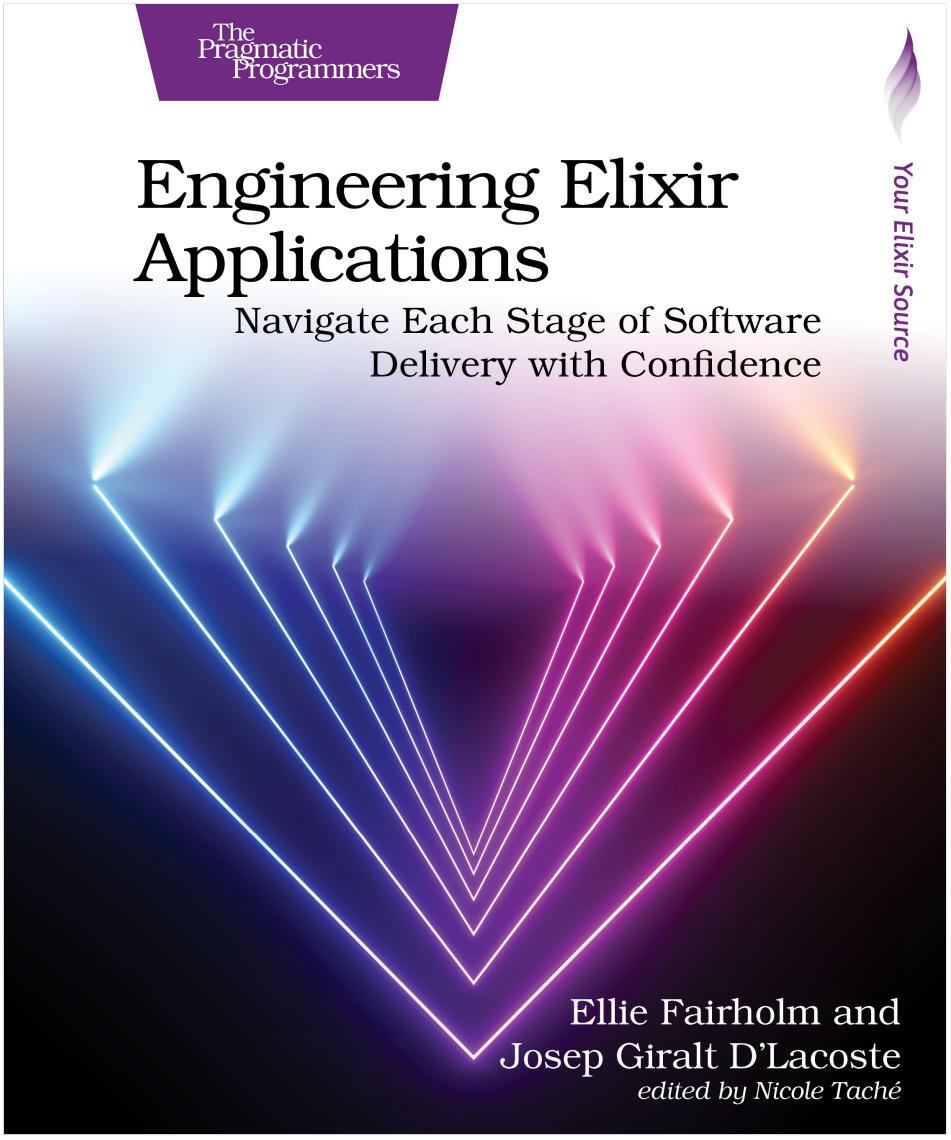

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Copyright © The Pragmatic Programmers, LLC.

CHAPTER 4

Set Up Integration Pipelines with GitHub Actions

In Chapter 2, we explained how turning the project management of a software product into a development task improves team efficiency because it empowers the whole team and shortens the feedback loop between developers and product owners. In this chapter, you're going to learn how to further shorten this feedback loop and adopt proactive habits for testing your code by implementing a continuous integration (CI) pipeline with GitHub Actions.

A CI pipeline helps you ensure that your code works as you expect it to. It streamlines the delivery process of your application by seeking out bugs and/or issues *before* they're reported. We believe that the CI pipeline must be implemented as soon as possible when delivering a software product to ensure your code is reliable from the start.

Closely related to the CI pipeline is the concept of "continuous improvement." The Japanese term for continuous improvement is *kaizen*. By using a CI pipeline, you're following what's known as the *Kaizen principle*—the idea that small, ongoing changes can lead to significant improvements. Exercising this principle is essential in software development because it encourages better quality and more reliable code. You should always aim to ship less code, more often.

After reading this chapter, you'll understand the mandatory steps that an Elixir application must follow to safely ship to production. You'll also learn the different triggers for a GitHub Action that will run your CI pipeline, and you'll build on your understanding of Docker by pushing your Docker image to the GitHub container registry in the CI. By the end of this chapter, you'll close three of the CI/CD GitHub issues that you created in Chapter 2.

• Click HERE to purchase this book now. discuss

As with the previous chapter, we'll continue following our same development approach of manually implementing the necessary steps first and then automating them. Let's get started and implement the mandatory steps that each CI pipeline for an Elixir application must follow.

Mandatory CI Steps for a CI Pipeline

When implementing a CI pipeline, you must always distinguish between mandatory and nonmandatory steps. Mandatory steps are the minimum requirements that your application must meet before being shipped. Nonmandatory steps are nice-to-haves that don't affect the basic running of your application. In the case of an Elixir application, the mandatory steps are to ensure that: the code compiles, already compiled files and previously fetched dependencies are cached, the tests pass, the code is properly formatted, the code has been analyzed by Dialyzer, and that no unused dependencies exist. Let's go through each step, one by one, starting with code compilation.

Code Compilation

The first, and most important, step is to ensure your code compiles properly. As we mentioned earlier, rather than jumping straight into automation, first try to compile your code locally. To do so, use the `mix compile` command. To ensure that you compile your code from an initial, untouched state, as your CI would do, remove your `_build` directory from the project you created in the previous chapter and then compile your code by running `rm -rf _build && MIX_ENV=test mix compile` as we've done here:

```
$ rm -rf _build && MIX_ENV=test mix compile
==> file_system
Compiling 7 files (.ex)
...
```

You'll notice we've set the `MIX_ENV` to be `test`. This is the environment you'll use when running your workflow. Great. You know that your code successfully compiles. A handy way to check whether the last terminal command was successful or not is to use the bash command `echo $?`. If you run it, you'll see that your terminal prints `0`, indicating a success.

Now that you know how to successfully compile your code, you can go ahead and implement your GitHub Action. However, we don't recommend doing this just yet. Instead, we advise that you also figure out a way to make your desired result fail. This is an important step in CI pipeline creation as it removes any false positives in your code. We recommend that you consistently follow this approach:

1. Make a test that fails locally.
2. Ensure that the test fails in the CI.
3. Fix the test so that it passes locally.
4. Validate that the test passes in the CI.

To make a test that fails locally, create a warning in your code by adding an unused module attribute to your lib/kanban_web.ex file. You can see how we've done so here:

```
# in lib/kanban_web.ex

defmodule KanbanWeb do
  @unused_attr ""
end
```

Now, compile your code again, but this time using the --warnings-as-errors flag as we've done here:

```
$ MIX_ENV=test mix compile --warnings-as-errors
Compiling 1 file (.ex)
warning: module attribute @unused_attr was set but never used
  lib/kanban_web.ex:2

Compilation failed due to warnings while using the --warnings-as-errors
option
```

As the name suggests, the --warnings-as-errors option turns any code warnings into errors and means that if your code has any warnings when trying to be compiled, the compilation will fail. This is a useful option in CI pipelines as it preserves the integrity of your codebase. If you don't use this flag, the daily development workflow of your application might become unhealthy. Systems deteriorate pretty quickly if you start neglecting them, and so you should always try to fix a warning as soon as it arises. To do this, you must ensure that the CI doesn't let you deliver any code that has a warning.

Now that you know how to successfully and unsuccessfully compile your codebase, let's move on to creating the GitHub Action.

Add the mix compile Step to Your CI workflow

GitHub Actions are always defined in a YAML file within the .github/workflows directory. There are eight main, first-level keys that can be used in a GitHub workflow file, which you can see in detail in their documentation.[1] We'll only be focusing on the following four keys in this chapter:

1. https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions

- name: gives a name, or ID, which will be used to refer to your workflow. This name is displayed in the Actions tab of the repository.
- on: defines the triggers that will produce a CI job, such as pushing to a repository or opening a pull request.
- env: defines environment variables that will be available in your CI job.
- jobs: defines a list of "actions" that will be executed in the CI pipeline. Each job can have an ID, a name, a runner that specifies on which machine the job is run, environment variables, a set of steps, and a set of services.

To create your workflow, create a new .github/workflows folder at the root of your project and inside that folder a file called ci_cd.yaml and then paste the following code snippet into the file. We'll then go through each of the file components.

```
# in ci_cd.yaml

name: CI/CD Elixir

on:
  push:
  workflow_dispatch:

jobs:
  ci:
    runs-on: ubuntu-latest
    name: Compile
    env:
      MIX_ENV: test
    steps:
      - uses: actions/checkout@v4

      - name: Setup Elixir
        uses: erlef/setup-beam@v1.17.3
        with:
          version-file: .tool-versions
          version-type: strict

      - name: Get dependencies
        run: mix deps.get

      - run: mix compile --warnings-as-errors
```

As you can see, we've named the workflow CI/CD Elixir. CI/CD stands for Continuous Integration/Continuous Deployment. This chapter will only focus on the CI part. You'll implement the CD part in Chapter 7. We then configured the on key of the workflow so it's triggered either by the workflow_dispatch event, which means you can manually trigger the workflow, or by the push event, meaning each time you push to your remote repository. You'll tweak these on events later, as it's not very cost-effective to trigger a workflow on each push, but

we're using that event for now so that you can test the workflow. Lastly, we've used the jobs key to define an action with the ID ci and name Compile. The id is what you could use later in the workflow if you wanted to refer to that specific job, whereas the name is what will appear in the logs of the run to refer to that specific part of the workflow. The machine used to run your workflow is called a runner. We've specified that the job will run on an ubuntu-latest machine, use the MIX_ENV=test environment variable, and have the following steps:

1. Check out your project code in the runner's machine.
2. Install Elixir on the runner's machine using the .tool-versions file you created in Chapter 1.
3. Get your project's dependencies.
4. Compile your project's code.

A step is either a shell script, defined by the key run, or a predefined action, defined by the key uses. In the earlier example, we've used both keys. A predefined action is an open-source, reusable unit of code that performs a particular task, such as checking out your code or installing a programming language and making it available in your runner's path, as you can see in the first two steps of the previous code snippet. GitHub has a marketplace[2] where you can search for all different kinds of actions. When installing the project dependencies, we used the run key to execute the command mix deps.get without having to use a predefined action. This is because the previous "Setup Elixir" step installs Erlang on the runner's machine, which means that all subsequent job steps can run any mix command as normal.

You'll see that the second step sets the Elixir version by reusing the .tool-versions file that you created in Chapter 1 to define the versions of the tools your application needs. By using the same tool to set up your environment and your CI pipeline, you ensure environment integrity by making it so that your application is always being run under almost all of the same conditions.

Okay, great! You've defined your workflow. Now add and commit both your ci_cd.yaml and lib/kanban_web.ex files with a message of your choice and then push your new commit to your remote repository. Your workflow will now be running in your GitHub account. Many developers choose to see the results of their jobs in the GitHub UI. However, we recommend using the GitHub CLI as it allows you to see the workflow output directly in your terminal rather than

2. https://github.com/marketplace?type=actions

• Click HERE to purchase this book now. discuss

having to switch between your terminal and the browser. We'll look at the basic GitHub CLI commands in the next section.

Choosing Your Branch

The workflow in this section only works if it's run on the `main` branch. This is because the `workflow_dispatch` trigger only works on the default branch. Pushing to `main` in this instance isn't an issue as you're setting up a pipeline for a project still in its development phase and you're working by yourself. If you were to be working in a team or setting up a CI pipeline for an existing project with multiple developers, you should test it by either implementing this pipeline using branches or a replica project in your personal account.