12 Continuous Integration & Build Servers Written by Evana Margain Puig

In the previous chapter, you learned about **Fastlane**, one of the best automation tools for your local environment used by developers worldwide. You also learned how to build, test, take screenshots and release your app with Fastlane.

Fastlane is a great tool and one that you'll often encounter in your Android development career. But most companies take it one step further. Instead of making developers do automation tasks on their local machines, they delegate them to **build servers**.

This chapter will teach you how to use a free and open-source build server hosted by **GitHub** called **GitHub Actions**. With its help, you'll take your automation skills to the next level. Even better, you'll free your computer from executing tasks locally so you can continue developing while the server does its job.

The benefits of CI

Before diving into build servers, it's important to understand two common terms: **Continuous integration** and **Continuous delivery**, also known as **CI/CD**. These two terms are often a source of confusion, and many developers have trouble understanding the difference between them.

- **Continuous integration**: A development practice where you **continuously** merge code into the main repository. Once you merge the code, a series of automated tasks occur, such as testing or linting the code.

- **Continuous delivery**: The process of delivering useful and working code to the final user in a consistent timeframe. Depending on the project and number of developers, a company can choose a CD pipeline that runs as often as once a day or infrequently as every two weeks.

Finally, take a look at build servers.

- **Build servers**: Dedicated instances of virtual machines that can build, test and run your code. In Android, the server you use will compile Android code on a version you specify when configuring it.

Advantages of using a build server

At this point, you may be thinking there's not much of a difference between build servers and what you learned in the last chapter. Here are some examples of the advantages build servers provide:

- They don't run on your computer, so you can keep working while the remote server validates the code.

- All developers on a team can commit their code to the server and then ensure all the code is working. Have you ever been in a situation where a developer swears the code is working on their computer, but there's no way to run it on another computer? Continuous integration solves this important problem because the build server acts as the single source of truth.

- Build servers can have higher specs than many local machines, especially laptops. These specs put less pressure on development machines and give you stable build times. If you've ever worked on a large app that takes more than ten minutes to build, it is easy to appreciate how this is an advantage.

You'll notice many other characteristics as you learn how to use build servers. Keep reading to level up your CI/CD skills.

Which CI/CD provider to use?

There are many CI/CD providers out there. Like most things in tech, and life, each has its advantages and disadvantages. Here’s a list of the some of the most commonly used CI/CD providers in the Android platform. This list is not exhaustive, but a good starting point when looking for a provider.

GitHub Actions

- **Advantages**: If you have your project repository on GitHub, this is a great option. The best part is that the free plan is generous, and for that reason, you’ll use it for this tutorial.

- **Disadvantages**: For larger business critical projects, you may want to get a paid service so you can get support. Since the community creates actions in **GitHub Actions**, they may have more errors, bugs or limitations than a paid service.

Bitrise

- **Advantages**: Bitrise has become popular in the mobile world. It’s stable and has a free tier, so you can use it for small projects and start paying as you grow your app. The interface is easy to use, so new users can get used to working with it quickly.

- **Disadvantages**: Some tests are problematic to run, and updates can break your tests.

CircleCI

- **Advantages**: CircleCI is another commercial platform like Bitrise. It also has a free tier and custom pricing for self-hosted projects.

- **Disadvantages**: It’s a bit more technical and can be harder to use if you’re newer to the platform.

Jenkins

- **Advantages**: Jenkins comes from the historical roots of Android in Java. It’s been in the software industry for several years. Many developers have used it at some point. For the same reason, it has a large development community that provides many plugins and maintenance while still being free and open source.

- **Disadvantages**: It requires the most technical knowledge. Jenkins has to be self-hosted or administered by someone on your team in a cloud provider through a Docker machine. If you're just getting started, the learning curve can be a very frustrating experience.

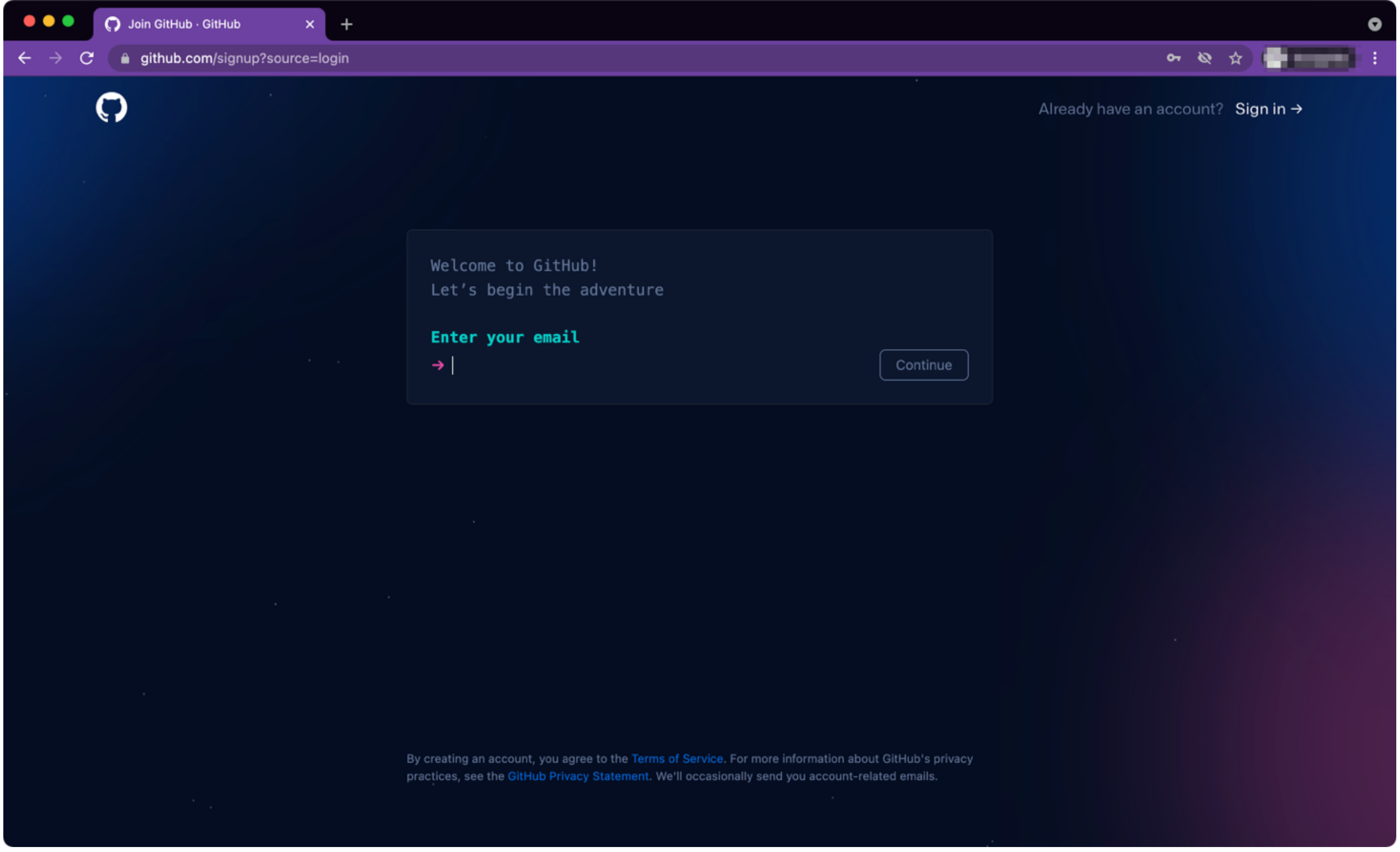
GitHub Actions

You are going to use GitHub Actions as the CI tool for this chapter. Before using GitHub Actions, you need to upload the project to GitHub. If you already know how to do this, create a repo with PodPlay and skip the next section. Otherwise, follow along.

Uploading PodPlay to GitHub

You didn't come to this chapter to only read theory, and at this point, you're probably eager to create your own CI/CD pipeline. To work through this part of the chapter, you'll need a GitHub account and to upload your **PodPlay** code to a repository there.

If you don't have one already, start by creating a new account on their site by visiting http://www.github.com in your favorite browser and clicking **Sign Up**. They have a very interactive interface. Follow the steps until you have an account.

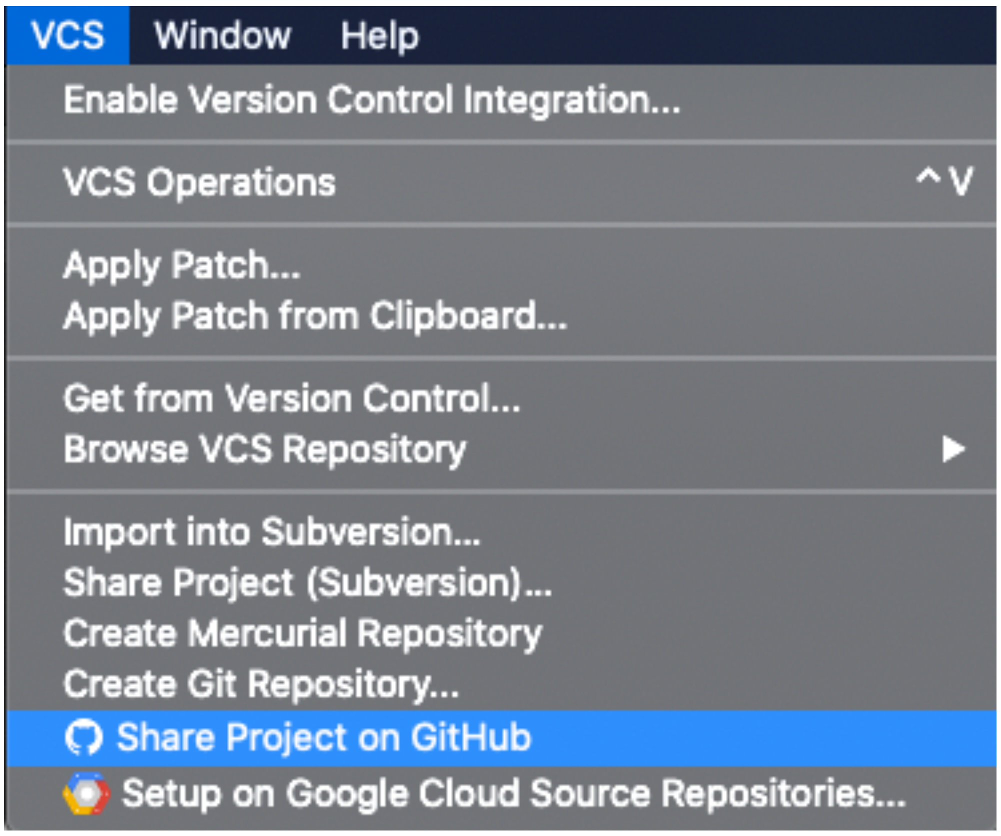

Join GitHub · GitHub
github.com/signup?source=login
Already have an account? Sign in →
Continue

Welcome to GitHub!
Let's begin the adventure

Enter your email
→ |

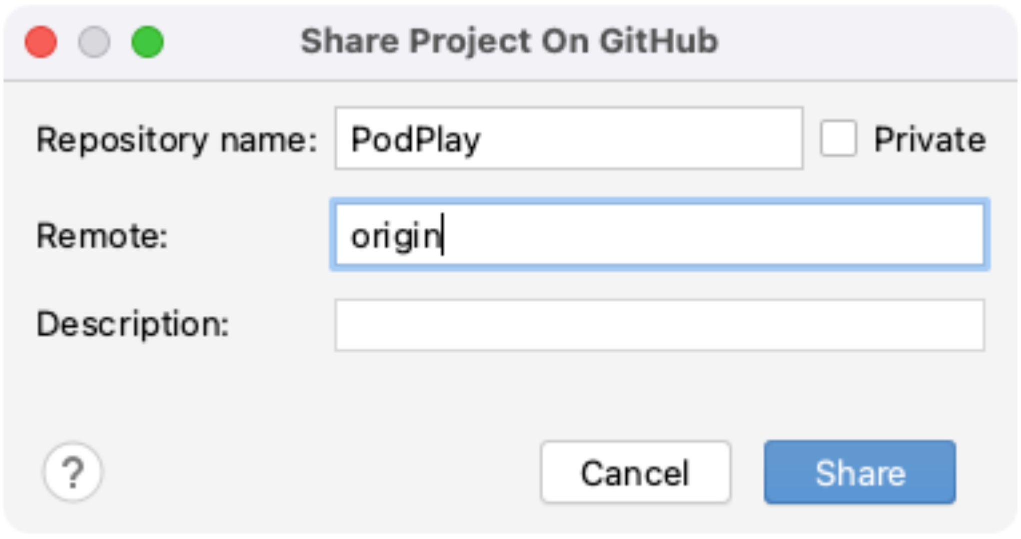
By creating an account, you agree to the Terms of Service. For more information about GitHub's privacy practices, see the GitHub Privacy Statement. We'll occasionally send you account-related emails.

Once you have your account, log in. Copy the starter project of this chapter to your computer's Desktop, or some other location where you can work with it. Open it in Android Studio and click the top menu **VCS ▸ Share Project on GitHub**.

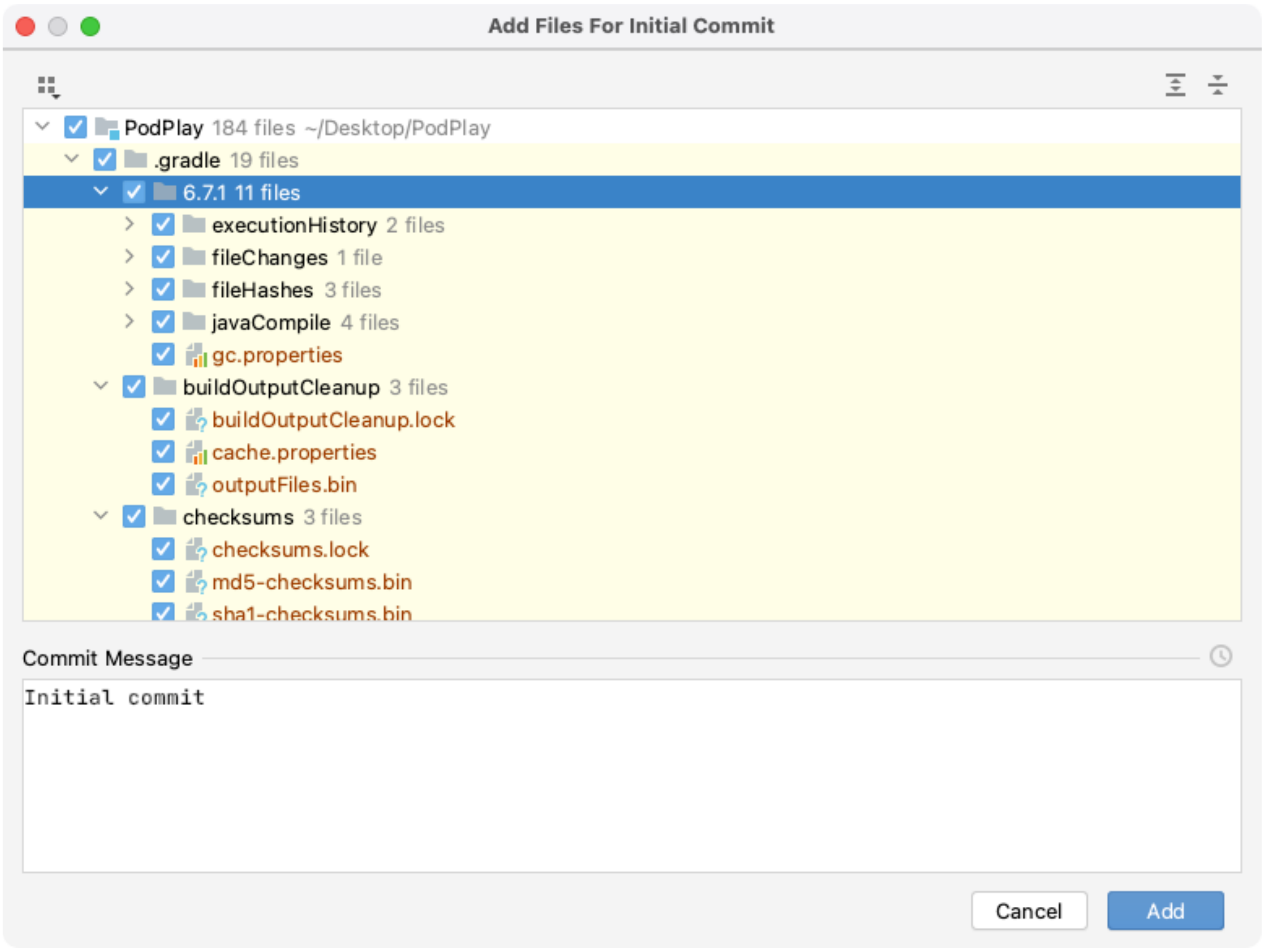

VCS
Window
Help
Enable Version Control Integration...
VCS Operations ^V
Apply Patch...
Apply Patch from Clipboard...
Get from Version Control...
Browse VCS Repository
Import into Subversion...
Share Project (Subversion)...
Create Mercurial Repository
Create Git Repository...
Share Project on GitHub
Setup on Google Cloud Source Repositories...

If you've already set up Git in your Podplay project, you'll have to use **Git ▸ GitHub ▸ Share Project on GitHub**.

If you haven't already logged in to GitHub through Android Studio, do so by using **Add Account ▸ Log In via GitHub**. Then leave the defaults in the following dialog and click **Share**.



Another dialog will show you all the files it will upload to GitHub. Accept everything as it is and click **Add**.


Add Files For Initial Commit

PodPlay 184 files ~/Desktop/PodPlay
.gradle 19 files
6.7.1 11 files
executionHistory 2 files
fileChanges 1 file
fileHashes 3 files
javaCompile 4 files
gc.properties
buildOutputCleanup 3 files
buildOutputCleanup.lock
cache.properties
outputFiles.bin
checksums 3 files
checksums.lock
md5-checksums.bin
sha1-checksums.bin

Commit Message

```
Initial commit
```

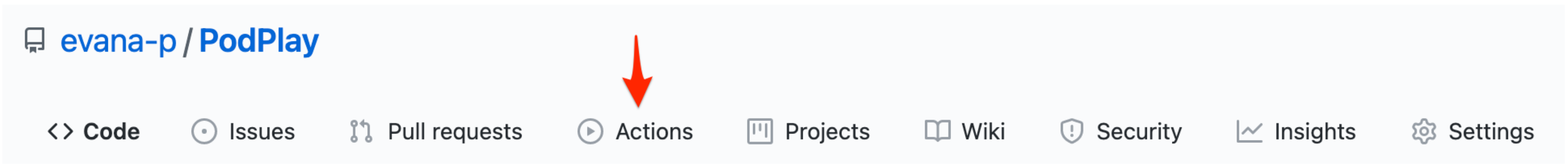
The upload may take a couple of minutes, depending on your internet connection. You can see the progress on the bottom right corner of Android Studio. Once it's done, an alert will show in that same bottom right corner, indicating it was successfully shared. It will also show a blue link named **PodPlay** that will take you to the repo.

Successfully shared project on GitHub
PodPlay

Open the repo with the link on the alert and take a look. You'll notice the files you had on your computer are now shared on GitHub.

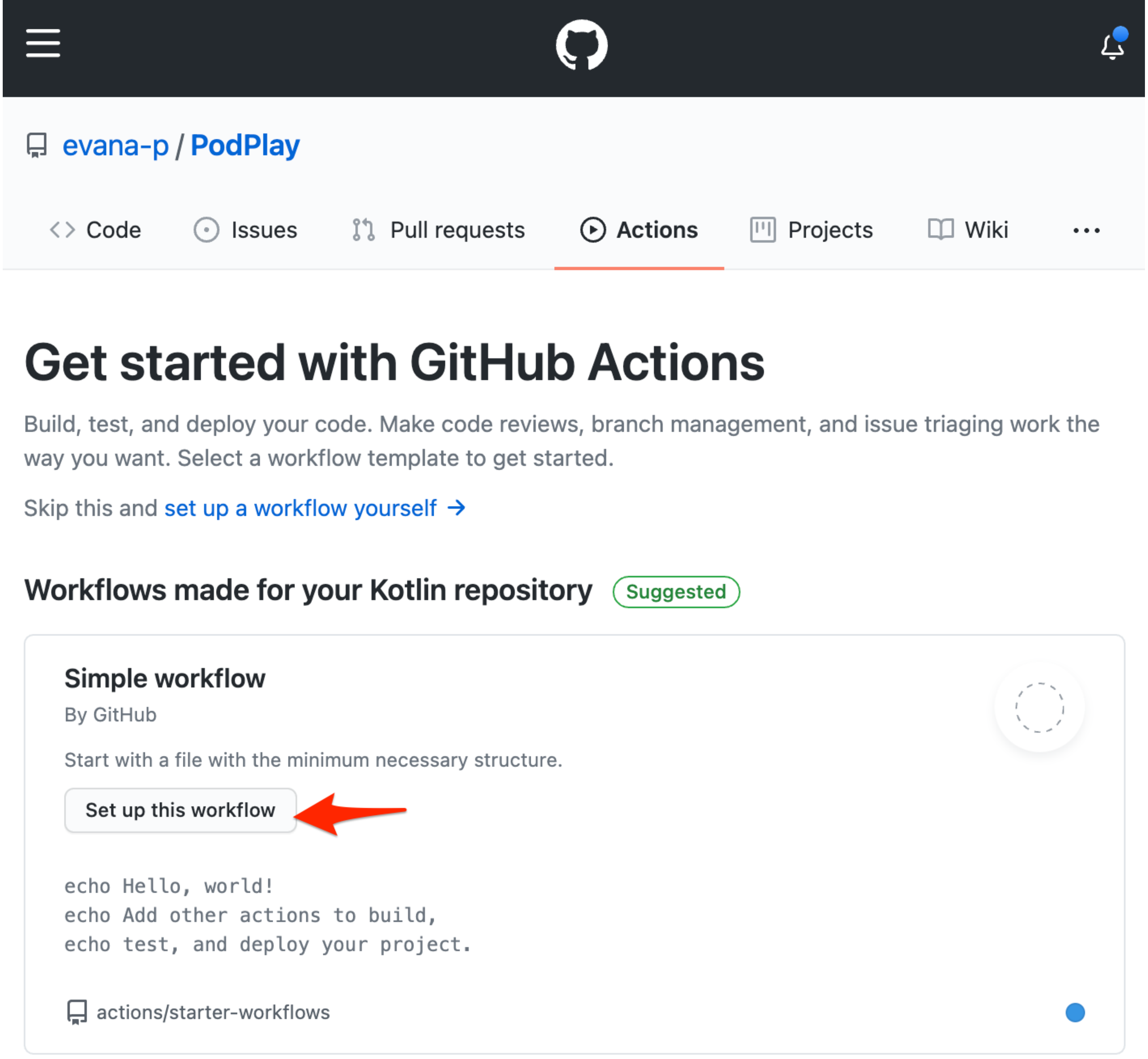
Getting started with GitHub Actions

Go to the main page of the repository. In the horizontal menu below the repo's name, click the button labeled **Actions**.



The page that opens will give you a brief introduction to **GitHub Actions**. It will also suggest some pre-made flows to get you started right away.

Clicking **Set up a workflow yourself** and **Set up this workflow** suggested for the Kotlin repository gives you the same base script. In case GitHub changes this in the future, make sure to click **Set up this workflow**, as shown in the image below.


evana-p / PodPlay

Code · Issues · Pull requests · Actions · Projects · Wiki · ...

Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow template to get started.

Skip this and set up a workflow yourself →

Workflows made for your Kotlin repository Suggested

Simple workflow
By GitHub

Start with a file with the minimum necessary structure.

Set up this workflow

```
echo Hello, world!
echo Add other actions to build,
echo test, and deploy your project.
```

actions/starter-workflows

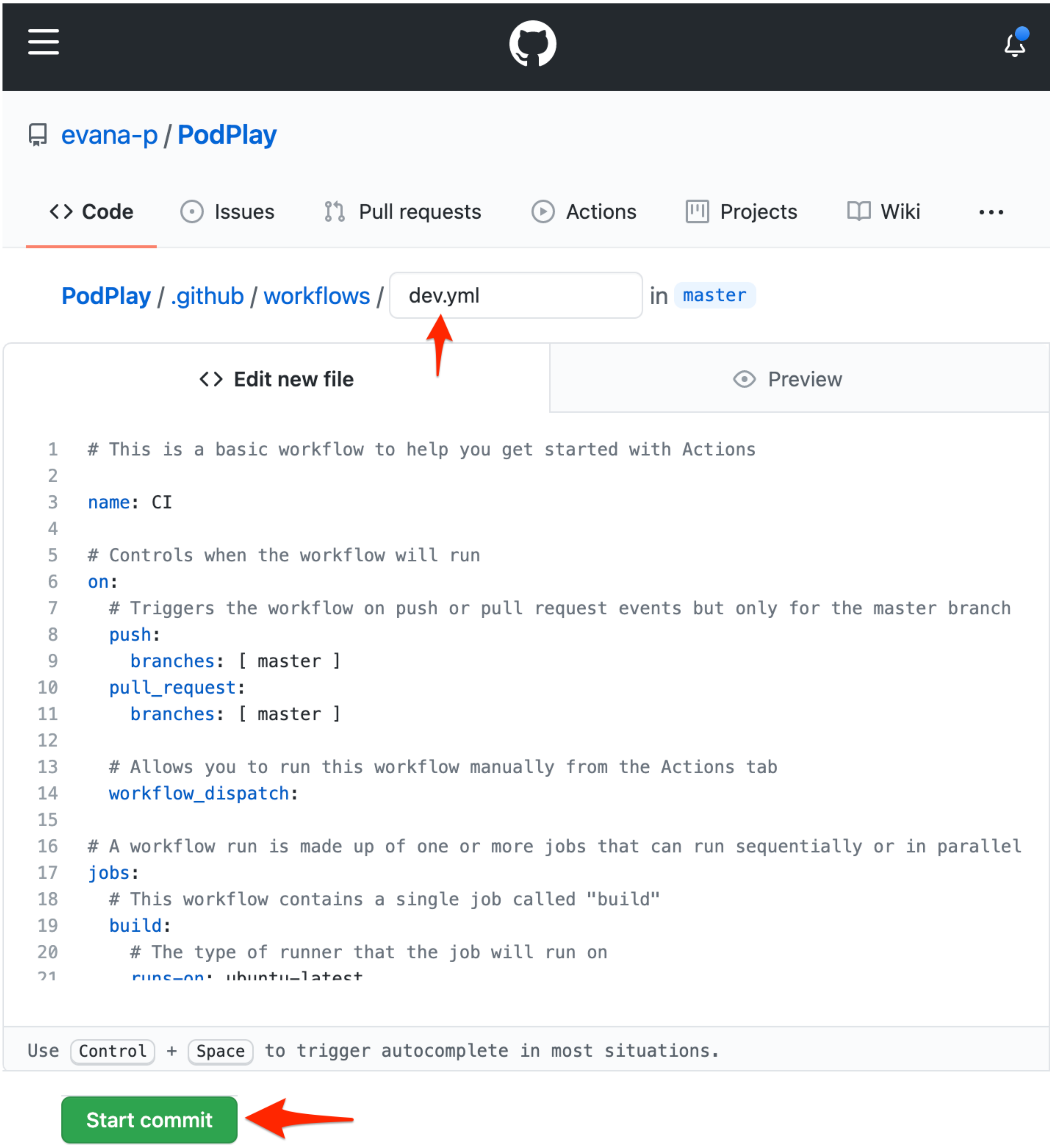
Then you’ll get a **blank.yml** for the workflows. **YAML**, the file ending in **.yml**, is the format used by all CI providers. You’ll specify the instructions for your different flows in this file.

The template you get is the basic workflow GitHub provides. You can read the comments identified with the **#** sign to see what each part does. Don’t worry if you don’t understand it right now. You’ll learn everything you need in this chapter.

In the following sections, you’ll learn more about what each part means and customize it for **PodPlay**.

Change the name to **dev.yml** and leave the rest of the file exactly as it is.

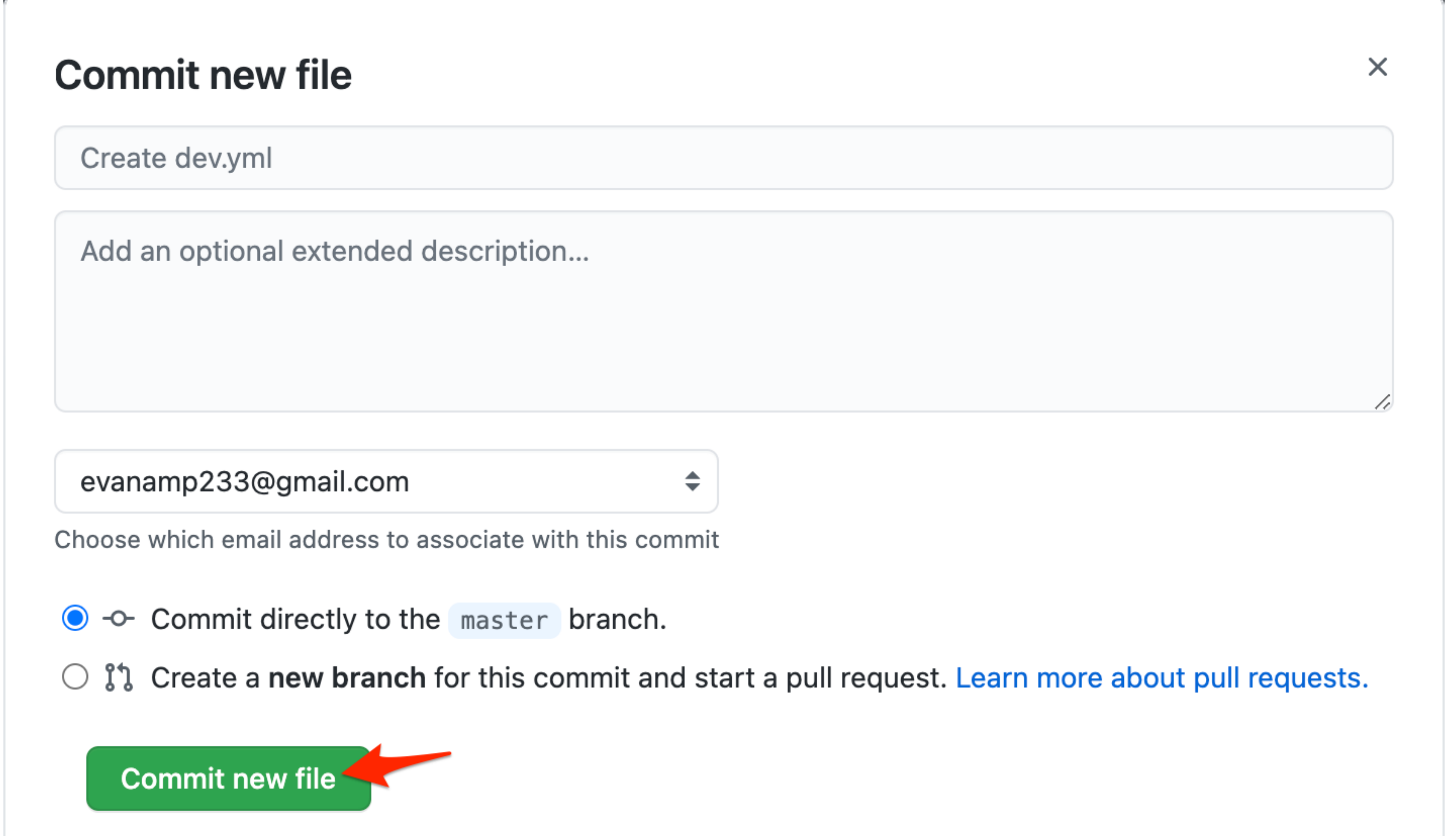
Click **Start commit** to add the new **YAML** file to the repository.



```yaml
# This is a basic workflow to help you get started with Actions

name: CI

# Controls when the workflow will run
on:
  # Triggers the workflow on push or pull request events but only for the master branch
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

# A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest
```

Use Control + Space to trigger autocomplete in most situations.

GitHub will ask you for a commit title and description like it does when you commit changes from your local computer to the repository. Leave the default options and click **Commit new file**.


Commit new file

Create dev.yml

Add an optional extended description...

evanamp233@gmail.com

Choose which email address to associate with this commit

[✓ Commit directly to the `master` branch.]

Create a **new branch** for this commit and start a pull request. Learn more about pull requests.

YAML building blocks

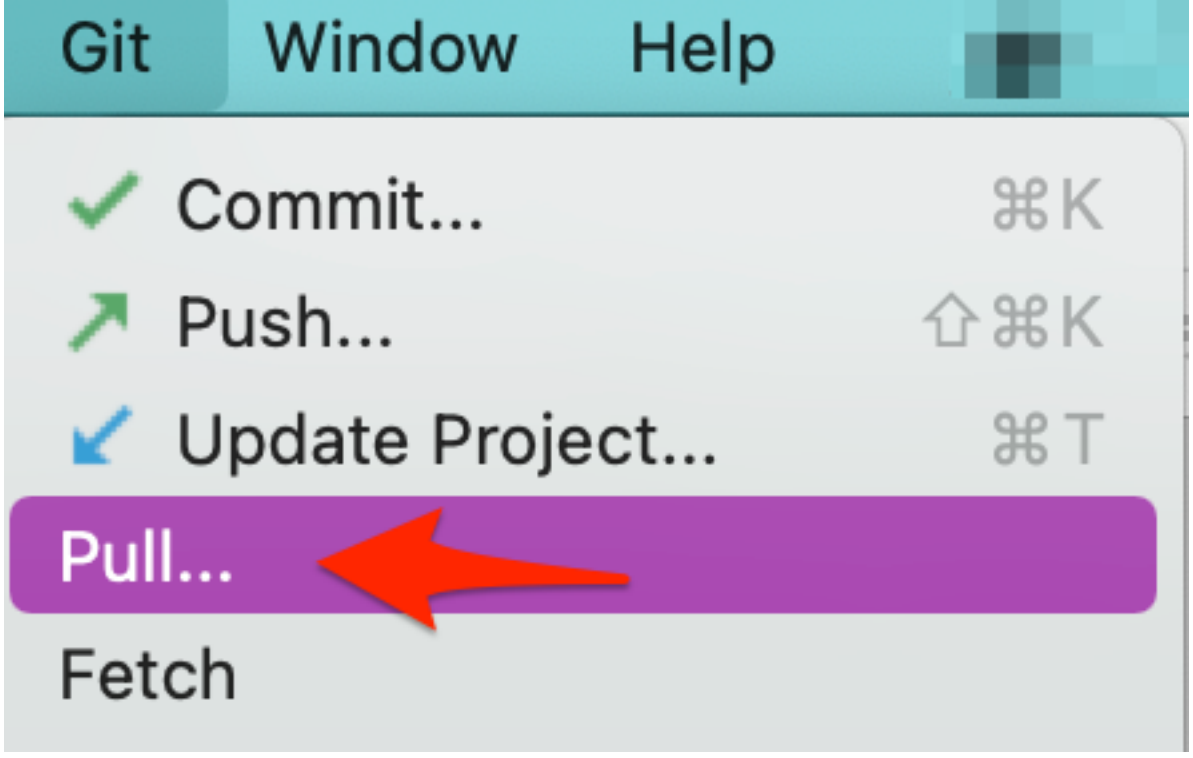
A build server can execute many key **actions** through the YAML files. GitHub has a catalog of pre-made actions that you can integrate into your flows.

Before integrating these custom flows, it's important to learn how to create one by yourself.

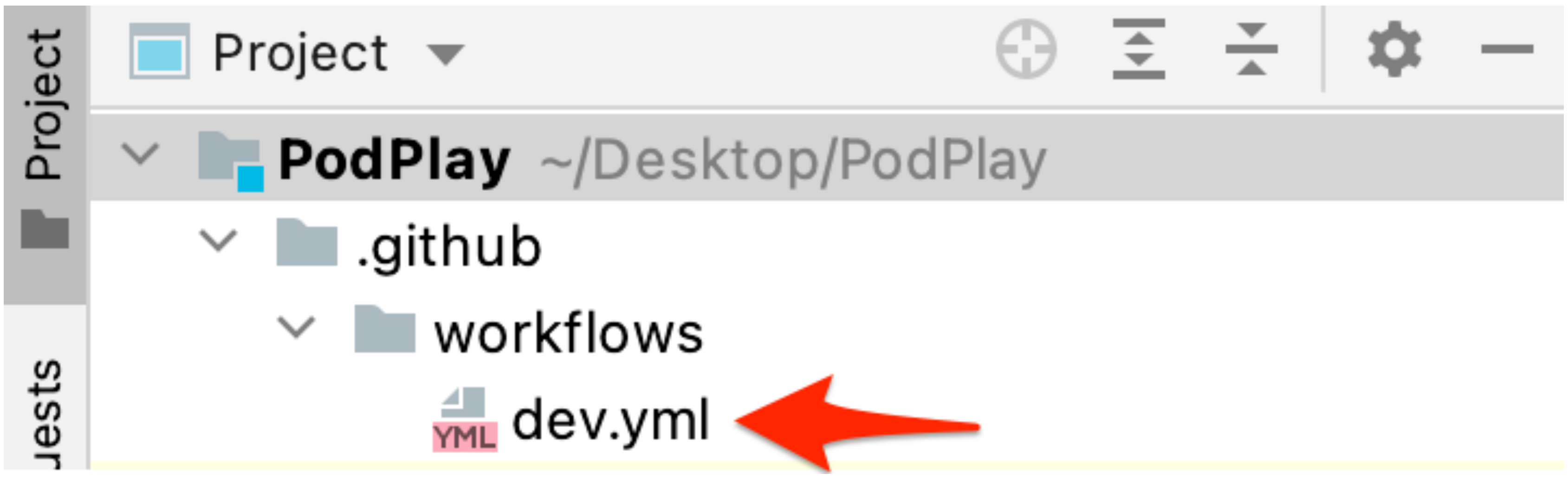
> **Note**: Although you're learning how to do this with **GitHub Actions**, most of the steps are the same across CI providers.

At this point, you'll edit and change **PodPlay**'s YAML file. Go back to Android Studio and pull the changes in master. Click the top menu **Git ▸ Pull** and clicking **Pull** in the dialog that appears.

Right now, you only have one branch, master, so that will get all the changes.



Now switch to the Project view of the file navigator in Android Studio. Navigate to **.github ▸ workflows**. There you'll see your newly added **dev.yml** ready for you to edit.



Name

This is the name at the top of the YAML file in line three. You can rename it to anything you want. Preferably, choose a name that will help you identify what that particular flow does.

In a production app, you may have YAML files for your different environments or processes. For example:

- Triggering a build for the QA team to test an added feature.
- Creating a Release Candidate for beta users to test.
- Running automated tests once a day to ensure your new code didn't break anything.

Each of those files should have a name that tells your team what it does.

Triggers

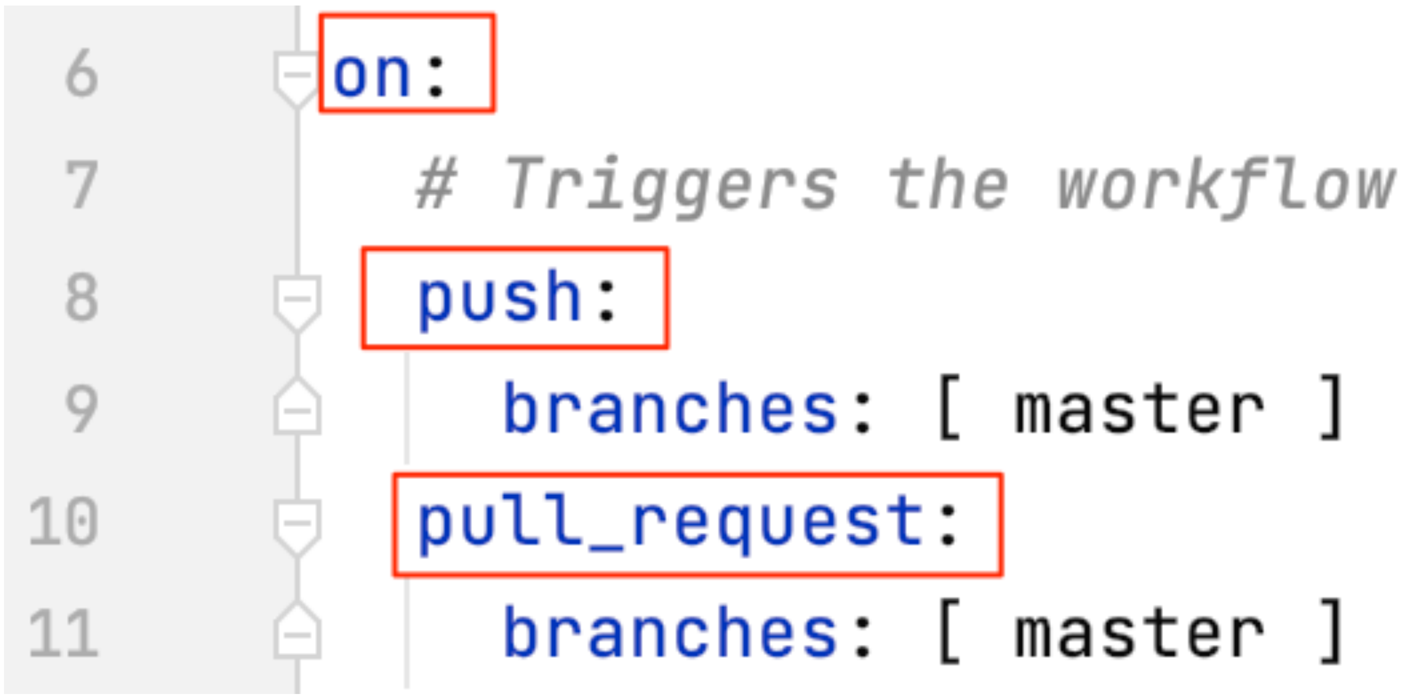
As the word suggests, **Triggers** are situations that cause a workflow to run.

If you're automating a process, you don't want to trigger it manually every time. With this keyword, you can define triggers that start any workflow every time something happens. Examples include:

- Opening pull requests.
- Merging pull requests.
- Every time someone pushes code in the repository.
- Creating new labels.

You can find a detailed list of other possible triggers in the GitHub Actions documentation at https://docs.github.com/en/actions/reference/events-that-trigger-workflows.

The keyword that identifies triggers is **on:**. Look at the YAML file you created and opened in Android Studio. In one of the first lines, you'll see this keyword, followed by two triggers, **push** and **pull_request**, as shown below:



```
on:
  # Triggers the workflow
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
```

Branch triggers

The most common trigger is when a specific action happens on a branch. In this example, you'll use a **push to the master branch**. This kind of trigger may also occur when you merge a pull request in a branch.

In your **dev.yml** look for the following in your file:

```
on:
  # Triggers the workflow on push or pull request events but only for the m
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
```

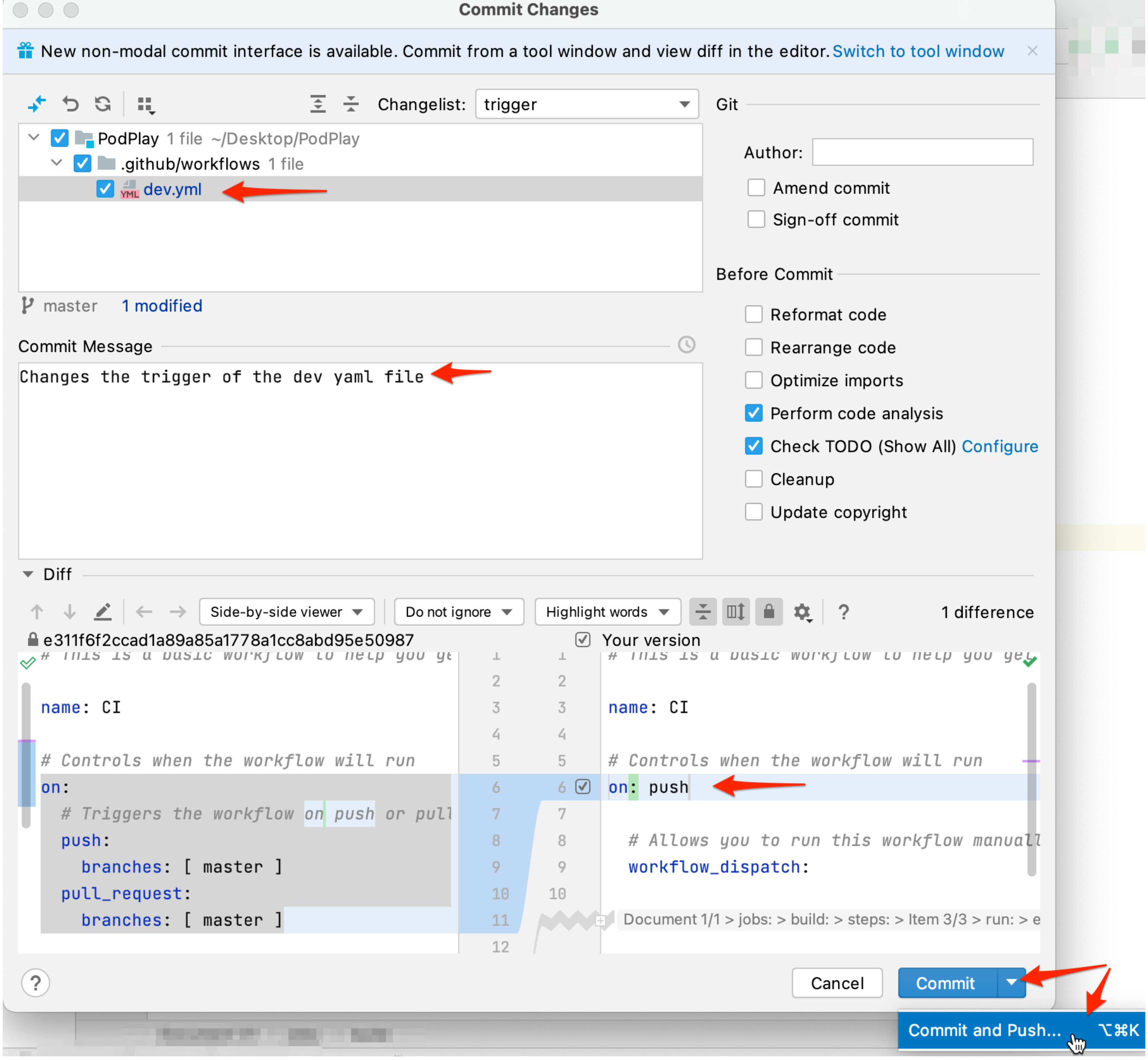
Replace it with the following:

```
on:
  push:
    branches:
      - 'master'
```

That causes it to trigger every time you **push** to any branch.

> **Note**: Some build server providers that charge based on the number of builds and/or the build duration. A build trigger that is frequently hit under those circumstances could lead to a very large bill. If you have a small or limited hosting budget you may want to choose a trigger that is hit less frequently.

Now, push the changes to trigger your first workflow. In Android Studio's top menu, go to **Git ▸ Commit** and add a commit message as in the image below. Finally, click the small arrow next to the commit button and **Commit and Push...**.



```
Changes the trigger of the dev yaml file
```

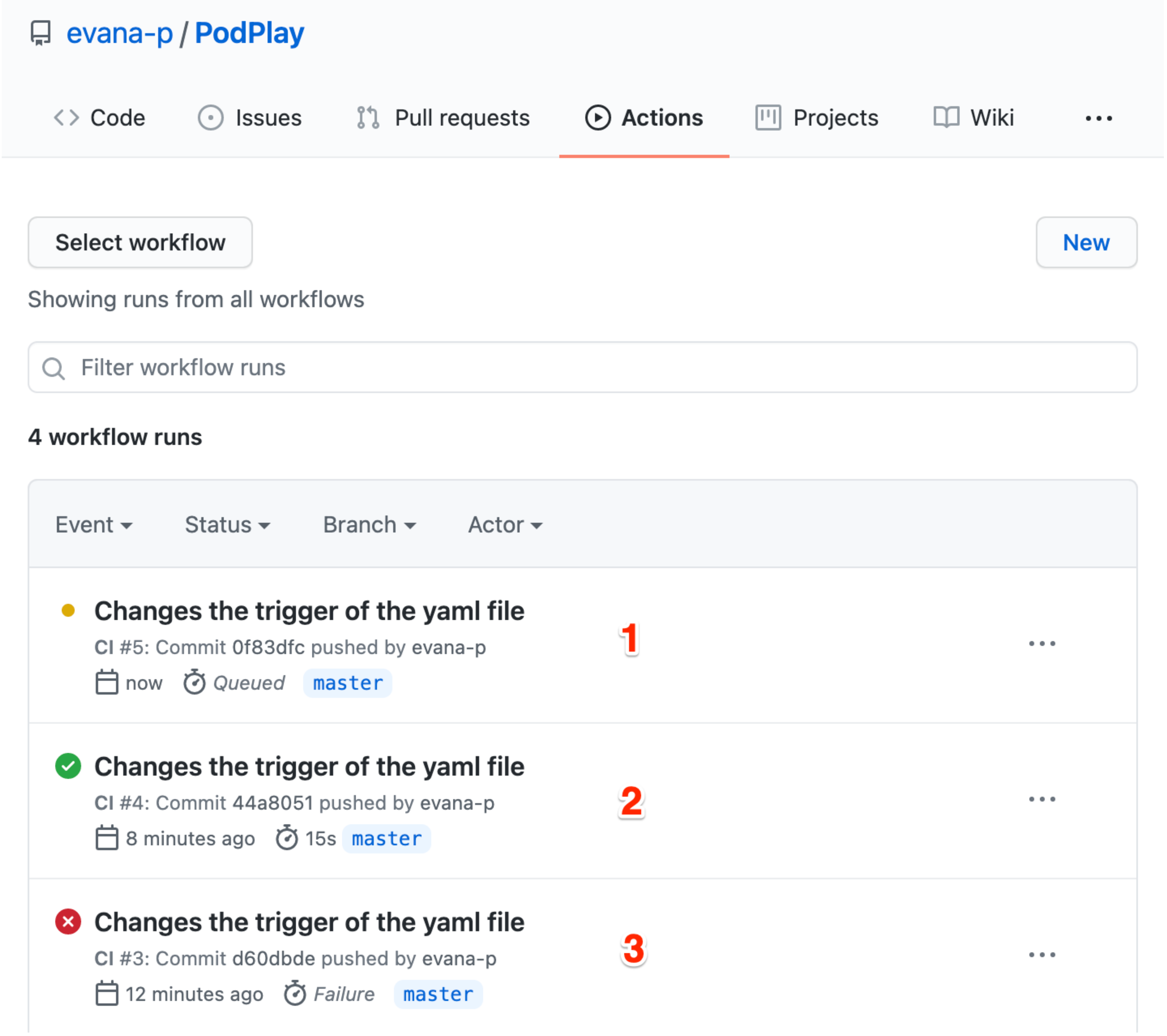
```
# This is a basic workflow to help you ge

name: CI

# Controls when the workflow will run
on:
  # Triggers the workflow on push or pull
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
```

```
# This is a basic workflow to help you get

name: CI

# Controls when the workflow will run
on: push

  # Allows you to run this workflow manuall
  workflow_dispatch:
```

Once you click to go back to the **GitHub Actions** page and look at the flows, you'll see one of three signs:

1. A **yellow dot** if the flow is still running, which happens for a couple of seconds. For some larger apps, it may even take a couple of hours.
2. A **green checkmark** if the flow ran correctly.
3. A **red cross** if you had an error.

Look at the image below for examples of each case:


evana-p / PodPlay

Select workflow

Showing runs from all workflows

4 workflow runs

Changes the trigger of the yaml file
CI #5: Commit 0f83dfc pushed by evana-p
now Queued master

Changes the trigger of the yaml file
CI #4: Commit 44a8051 pushed by evana-p
8 minutes ago 15s master

Changes the trigger of the yaml file
CI #3: Commit d60dbde pushed by evana-p
12 minutes ago Failure master

Ignoring branches

In the same way that you can trigger a flow when something happens, you can also ignore branches. Imagine a feature branch where developers are making tons of changes, but none of them is final. You can ignore those branches so developers don't overload the build server with any minor changes they make to their branches.

The syntax is the same as the last example but with `branches-ignore` instead of `branches`.

```
on:
  push:
    branches-ignore:
      - 'feature/**'
```

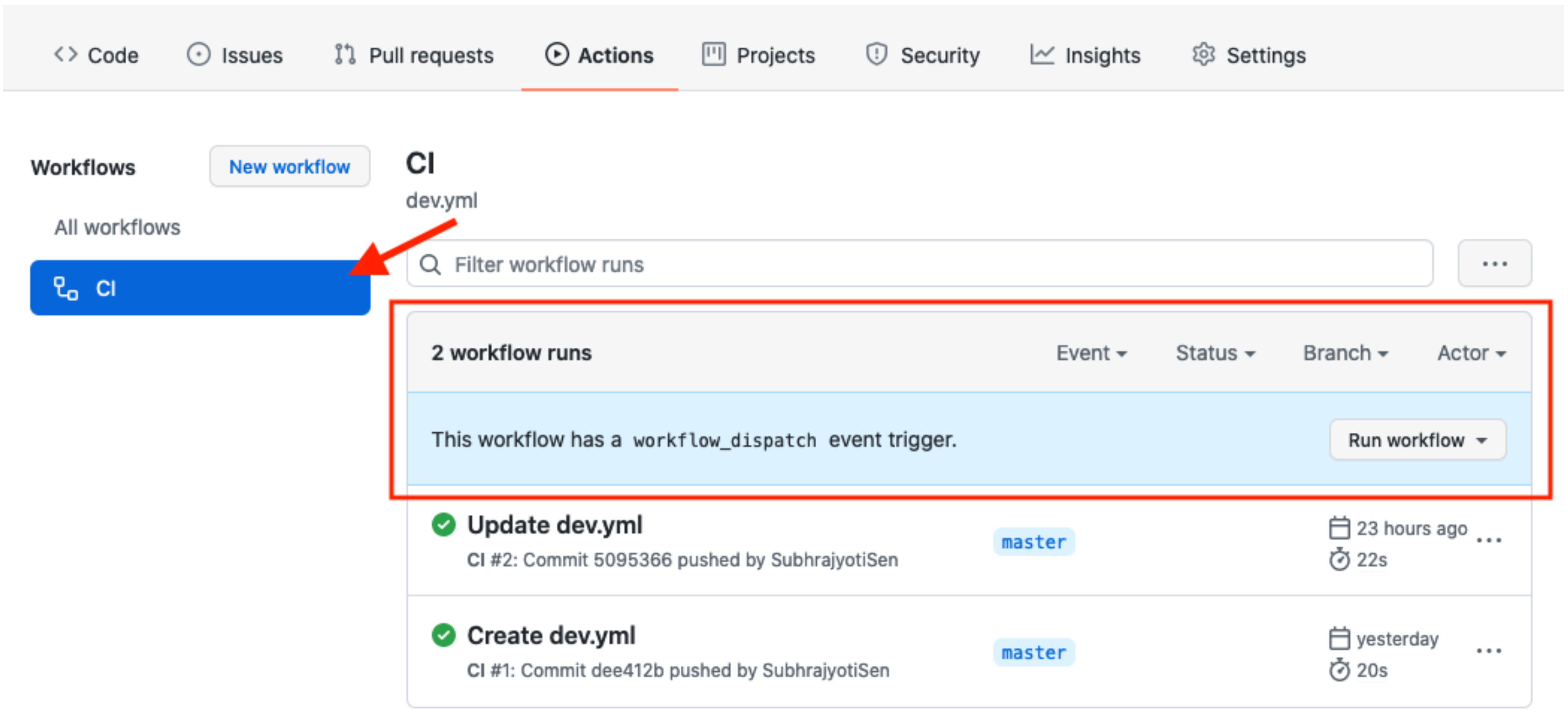
Since you don't have any feature branches, you don't need to add the above code to your YAML file.

Workflow dispatch

In the YAML file GitHub Actions generated for you, the next line reads `workflow_dispatch:` with nothing else. This line tells GitHub Actions that you want to run the flow manually directly from their site.

On the same page where you looked at the results, you'll notice a column titled **Workflows**. Click the one named **CI**.

A light blue square will appear showing a message, **This workflow has a workflow_dispatch event trigger.**, with a button to **Run workflow**. Click it to manually trigger the workflow like it gets triggered when you push something to master.



CI
dev.yml

2 workflow runs

This workflow has a `workflow_dispatch` event trigger.

Update dev.yml
CI #2: Commit 5095366 pushed by SubhrajyotiSen
master
23 hours ago
22s

Create dev.yml
CI #1: Commit dee412b pushed by SubhrajyotiSen
master
yesterday
20s

In some special cases, you'll want to trigger the build manually. Here are two examples when you might do this:

1. You want to validate a build is working without opening Android

Studio.
2. You need to re-rerun a build that failed for something out of your control, like network availability, a timeout or an error downloading a plugin.

In other cases, you may want to prevent someone on your team from running a build unless something very specific happens, so you don't add the `workflow_dispatch` step. This is particularly important when you're dealing with flows that push to production under certain circumstances. You don't want to risk someone manually triggering that build and pushing code to production.

Jobs and steps

The next part of the code is `jobs`. Jobs are a group of actions that run together independently of other jobs to complete a particular action.

You can have as many jobs as you want, but try to keep it short and simple. Otherwise, it becomes complicated for other developers to contribute to the workflow.

Jobs can be anything you like, but as most build flows are similar, you'll likely have these jobs:

1. **Setting up the project.**

- Checking out the code from the repository.
- Installing the dependencies on the remote machine.
- Downloading assets from your app's repository.

2. **Building the app.**

- Basically, the same things you do on your local machine when testing. This can be either for a debug, release or any other build variant you have defined.

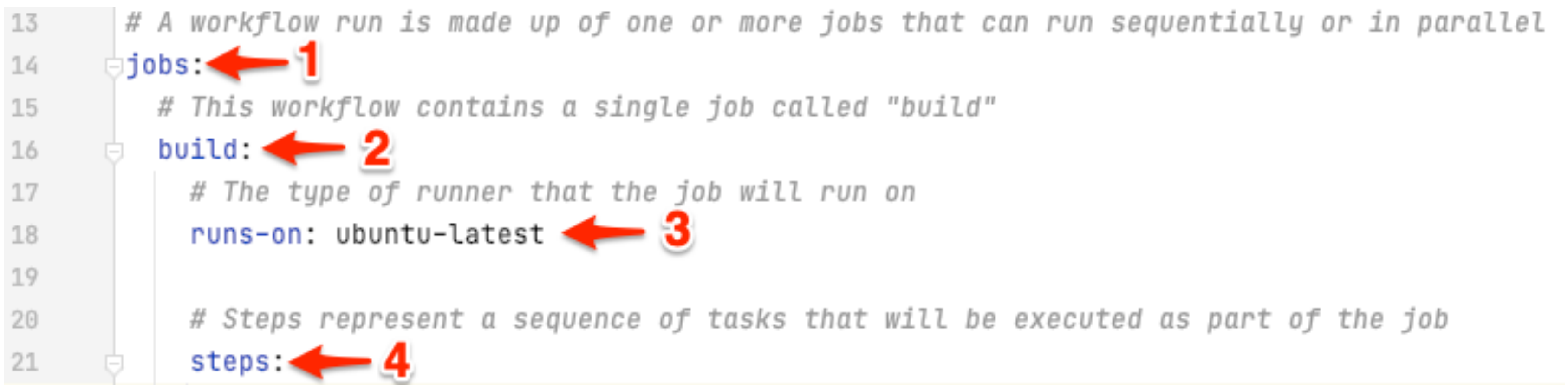
3. **Running tests.**

- As mentioned, developers often overlook this important step. You can run any type of test, but automated and unit tests are the most common.

4. **Depending on the flow, creating a beta, internal or public release.**

- You can release the app to a testing service that will distribute it to your desired testers. Testing services include App center, TestFlight and TestFairy.
- Releasing directly to the Google Play Console and creating an open, closed or internal test.
- Releasing to the Play Store to all your users, as long as you're sure this is a tested and working version.

In the sample project, you'll notice a line called `jobs`. Look at the image below to see what each of the following lines means. The numbers correlate to numbered explanations below:



```
# A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    # Steps represent a sequence of tasks that will be executed as part of the job
    steps:
```

1. The start of the `jobs` definition.

2. You have a job named `build`. Unlike other lines in the YAML file, you can give `build` any name you want. You'll have more than one in most projects, but the sample only provides `build` as a starting point.

3. Next, the YAML file specifies what type of remote server the build `runs-on`. For Android builds, you'll commonly use `ubuntu-latest` or `macos-latest`. Note that you can switch the word **latest** with a specific version like `macos-11`.

4. With those two things defined, you can start listing the steps of that specific job by inputting the line of code `steps`.

Creating a job

Your current sample file only has one job. To practice, you'll now create another job that will run your tests.

First, create the job below the final step of the existing `build` job at the end of the file. Name this new step `test`.

```
  test:
```

Make sure to ident the job name with two spaces, so it's at the same level as the other job `build`. Indentation is very important in YAML files, and your whole workflow will break if an item is incorrectly indented. You can use a tool like http://www.yamllint.com/ to validate your YAML files.

Setup tasks, commands & actions

As mentioned earlier, jobs run several steps. If you look at the sample file, each step starts with a hyphen and can consist of one or more lines. They continue until you encounter the next hyphen, a new job or the end of the file.

Steps can be one of three types: setup tasks, commands and actions. Take a look at each in detail.

Setup tasks

If you have a complex workflow, setup tasks will help you define a specific job's dependencies and other technical aspects. You already learned about one setup task, the `runs-on` keyword. As you learned before, this tells the virtual machine what kind of environment you want your flow to run on.

Another useful setup task is the keyword `needs`, which tells a job to wait until the server finishes the named job.

In the job you created, set up the environment as `ubuntu-latest` by pasting the following below `test:`:

```
    runs-on: ubuntu-latest
```

> **Note**: The indentation for this item is four spaces.

This is a setup task because it happens before the actual steps and gets everything ready for your job to run seamlessly.

Commands

Commands are instructions you give to the build server's terminal and run in an automated way. Whenever you run a manual Android build, as shown at the beginning of the chapter, you give instructions to the terminal to execute things.

A command is precisely that: You give the terminal an instruction, but the server automatically does it.

Some common examples include:

- Printing log messages so you can customize the output after you review how the build went. Log messages can be: debug, warning and error messages.
- Making the workflow start or stop depending on specific conditions.
- Setting environment variables.
- Creating paths in the virtual machine for files your build may need to create.

Now, create a command in your new `test` job, with the code below:

```
    steps: #1
```

```
      - name: Run a log message #2
        run: echo This is my job for running tests! #3
```

Here's a code breakdown:

1. Unlike the setup tasks, commands and actions start with the `steps` keyword. There are four spaces of indentation.
2. The hyphen that indicates a step is starting, followed by the keyword `name`. The name itself can be any string of your choice. In this case, you chose `Run a log message` as the name for that step. Be sure to use simple and understandable names to label what your flow is doing in each part. There are six spaces of indentation before the hyphen.
3. The keyword `run` without a hyphen because it's still part of the previous step. This keyword indicates what the step will do. In this case, it will print `This is my job for running tests!` in your output. There are eight spaces of indentation.

Actions

Explaining what an action is might seem redundant, but think of it this way. Action is a single command in your workflow. It's the smallest unit in your YAML file: Workflow > Jobs > Steps > Actions.

An action is similar to a method in programming. It contains a series of commands you can reuse as a single unit.

Take a look at an example by using the checkout action in your test. Add the following code to your YAML file:

```
      - uses: actions/checkout@v2 #1

      - name: set up JDK #2
        uses: actions/setup-java@v1 #3
        with:
          java-version: 11 #4
```

```
    - name: Unit tests
      run: ./gradlew test #5
```

There is a lot going on here, so let’s break it down:

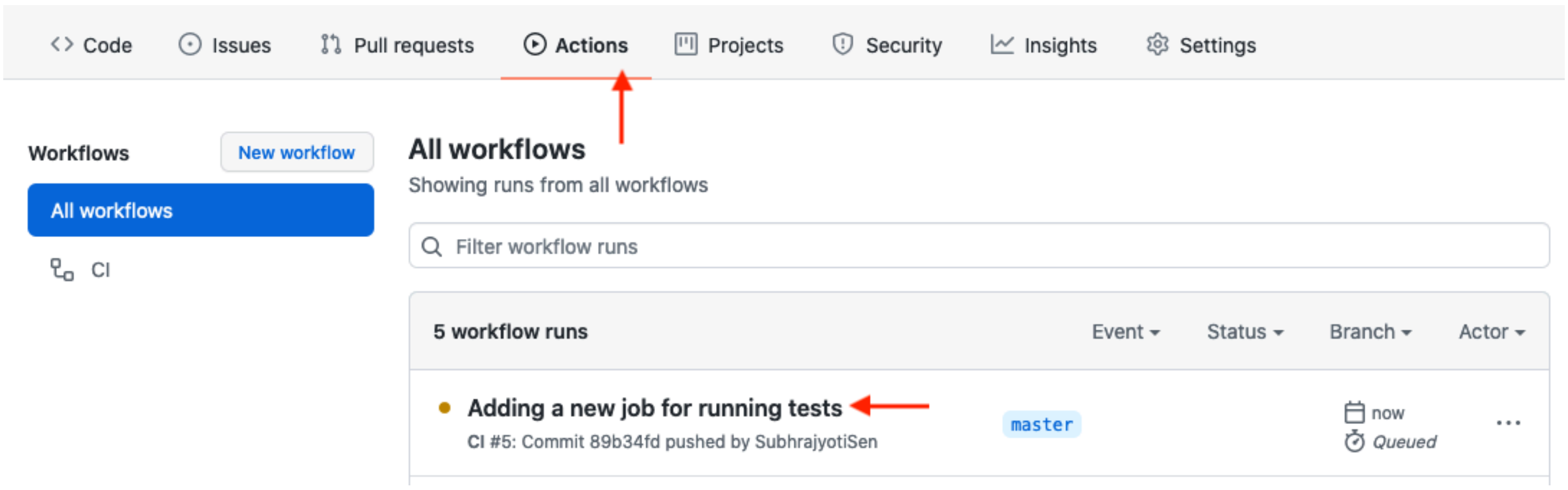
1. You’ll use the `checkout` action often because most of the time, you need to check it out in the remote machine to do anything with your app. `@v2` refers to version 2 of the action. This designation is useful when there are multiple versions of an action, and you want to use a specific version.
2. The name of the step that you’ll run once you check out the code.
3. `setup-java@v1` installs the correct version of Java on the build server.
4. You specify the Java version to use with the `with` keyword.
5. A command in gradle to run tests.

Your complete job is as follows:

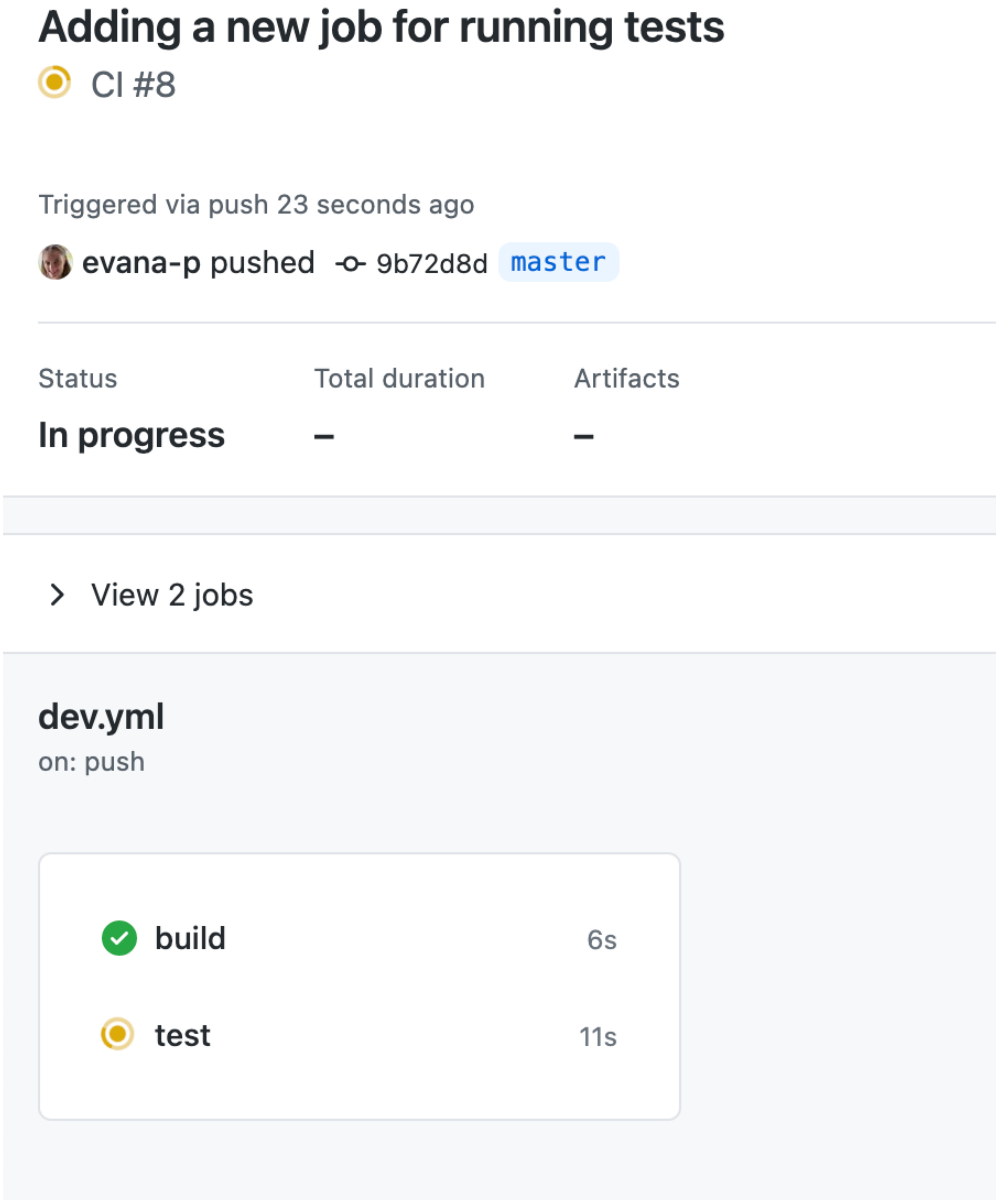
```
test:
  runs-on: ubuntu-latest
  steps: #1
    - name: Run a log message #2
      run: echo This is my job for running tests! #3

    - uses: actions/checkout@v2

    - name: set up JDK #2
      uses: actions/setup-java@v1 #3
      with:
        java-version: 11 #4

    - name: Unit tests
      run: ./gradlew test
```

With this code added, you’re ready to see your new job in action. Commit and push your code with the commit message `Adding a new job for` `running tests`. Go to your GitHub repository to the actions section.

It will take up to an hour to run because of the many build variants you created throughout the book. Since this is not a service you're paying for, servers are not as fast. As long as you see the yellow dot to the left of the run, it's still in progress.



While you wait, look at a couple of the details inside the workflow run. Click the bold text of the build as shown above, and you'll see the details page as follows:



Adding a new job for running tests

CI #8

Triggered via push 23 seconds ago

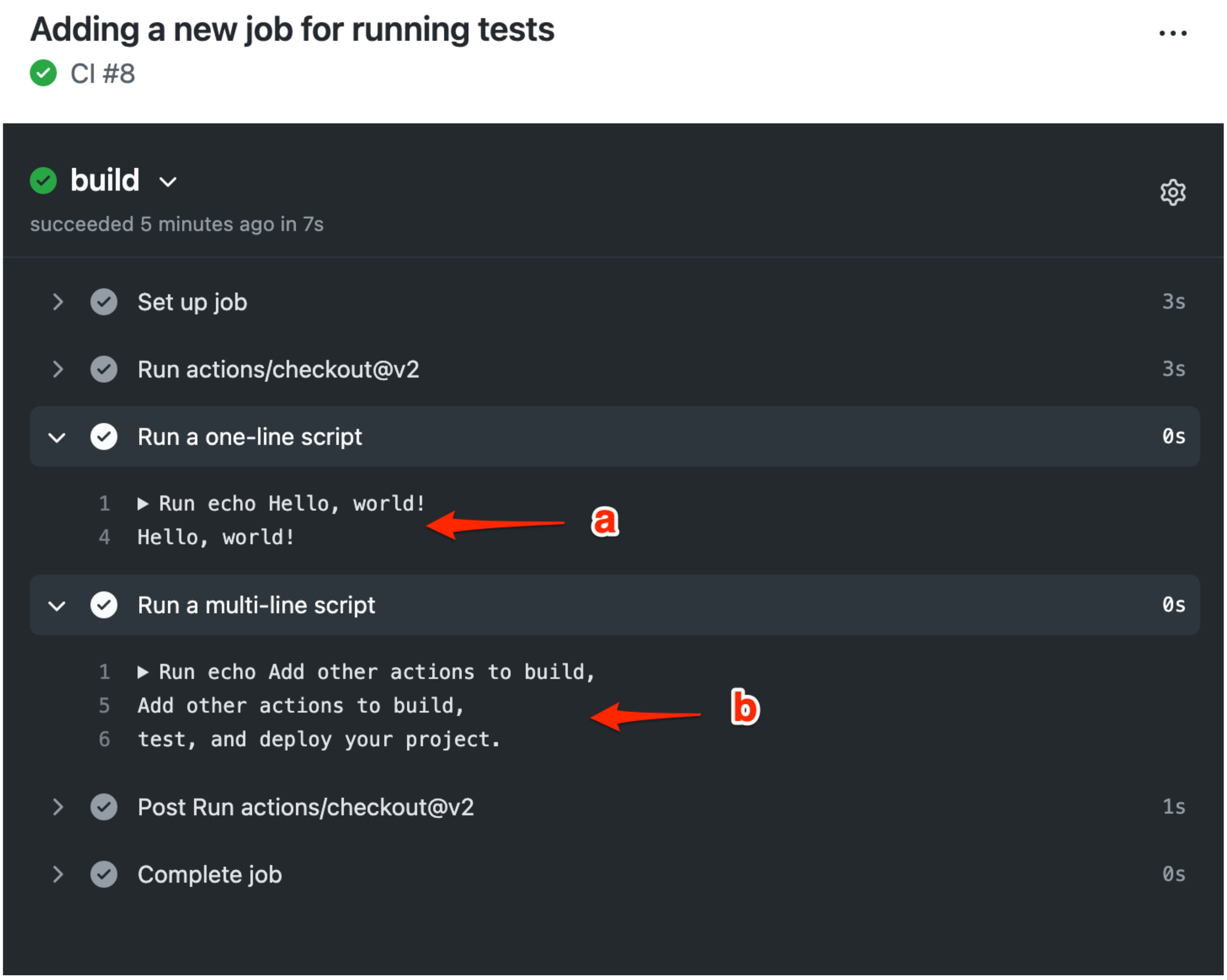
evana-p pushed 9b72d8d master

Status	Total duration	Artifacts
In progress	–	–

dev.yml
on: push

✓ build	6s
● test	11s

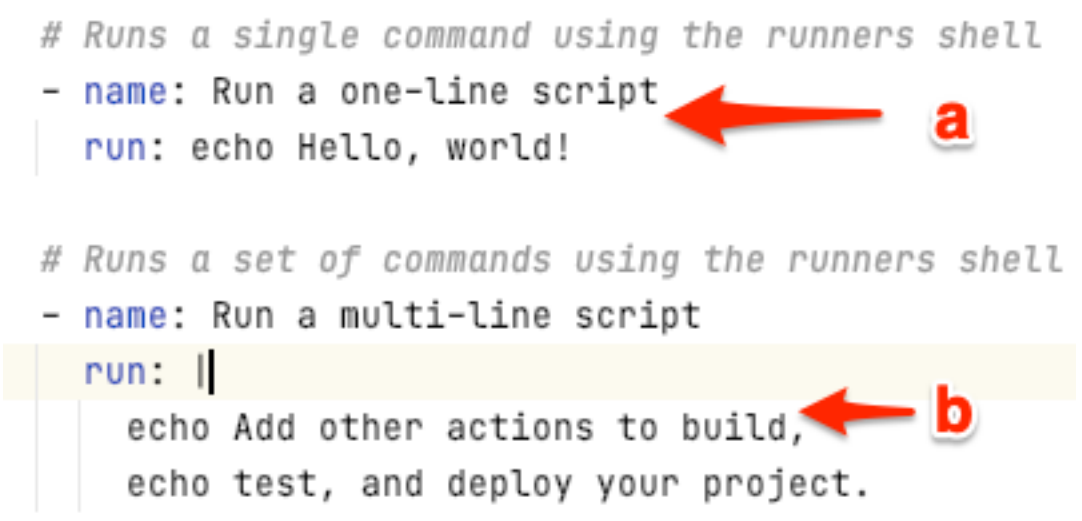
Click each of the jobs to see the execution's details. In the image above, the build job is already finished, and the test job is still ongoing.


Adding a new job for running tests
CI #8

build
succeeded 5 minutes ago in 7s

```
Set up job                          3s
Run actions/checkout@v2             3s
Run a one-line script               0s
    ▶ Run echo Hello, world!
    Hello, world!
Run a multi-line script             0s
    ▶ Run echo Add other actions to build,
    Add other actions to build,
    test, and deploy your project.
Post Run actions/checkout@v2        1s
Complete job                        0s
```

You can see the inspection of the build job and each of the steps signaled by an arrow and their corresponding name. See if you can identify the relationship of the steps with your yaml file. Look at the image below to identify where the `a` and `b` steps come from.



```yaml
# Runs a single command using the runners shell
- name: Run a one-line script
  run: echo Hello, world!

# Runs a set of commands using the runners shell
- name: Run a multi-line script
  run: |
    echo Add other actions to build,
    echo test, and deploy your project.
```

Great! Now you have your first workflow running on Github Actions. These build servers are powerful, and you can customize them to do many of your everyday tasks.

Key points

- CI is short for **Continuous integration**. It refers to the process of continually merging your code into a repository. The server takes care of testing, linting and other helpful things in your code.
- CD is short for **Continuous delivery**, the process of delivering to production as often as possible with the help of a build server. This process is not covered in detail here because it's a topic on its own, but the same build server can take care of it.
- **Build servers** are powerful machines that release your time and resources while saving your project from human error when automating tasks.
- There are many **CI/CD** providers with their own advantages and disadvantages.
- In this book, you tried **GitHub Actions** as your build server because it's free and easy to use.
- You need to upload a project to a GitHub repo to use GitHub Actions in it.
- GitHub Actions provides pre-made workflows that match your needs and get you started with a YAML template.
- **YAML** files are the base of Continuous Integration Servers, and the language and commands are very similar across build servers.
- You can set up **triggers**, such as when you create a pull request or push code to the repository, to automatically start your workflows.
- **Workflow Dispatch** refers to manually triggering a workflow without the trigger happening automatically.
- YAML files require a series of keywords to work. The most important are **jobs**, **steps** and **actions**.
- It's easy to monitor and debug your workflows through the GitHub Actions tab on the repository's webpage.

Where to go from here?

Congratulations! You just created your first CI pipeline.

In this chapter, you learned the basics you need to continue mastering CI and even dive into CD. Don’t be afraid to get deeper into the topic.

Reinforce your CD knowledge by taking a look at this tutorial: https://www.raywenderlich.com/19407406-continuous-delivery-for-android-using-github-actions.

Also, check out this amazing tutorial if you want to learn about continuous Integration if you want to learn how to sign and deploy your Android Apps with GitHub Actions: https://www.raywenderlich.com/10562143-continuous-integration-for-android.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.