

GitHub Actions IN ACTION

Michael Kaufmann
Rob Bos
Marcel de Vries



GitHub Actions IN ACTION

Michael Kaufmann
Rob Bos
Marcel de Vries



MANNING

GitHub Actions in Action

1. [welcome](#)
2. [1 Introduction to GitHub Actions](#)
3. [2 Hands-on: My first Actions Workflow](#)
4. [3 Workflows](#)
5. [4 GitHub Actions](#)
6. [5 Runners](#)
7. [6 Self-hosted runners](#)
8. [7 Managing your self-hosted runners](#)
9. [8 Continuous Integration \(CI\)](#)

welcome

Thank you for purchasing the MEAP for *GitHub Actions in Action*. We hope it will give you a kick-start into what is in our opinion the best and most flexible workflow platform for engineers.

The book consists of three parts. Part 1 explains the basics of GitHub actions. It uses a practical, hands-on approach – but it covers all basics with simple examples. Part 2 explains the workflow runtime in depth. This knowledge will help you completely understand the platform's architecture and security considerations. Part 3 covers the whole topic of CI/CD – the main use case for GitHub Actions. This part contains more complex real-life examples. It also dives into security, compliance, and optimizing performance and costs.

Please let us know your thoughts in the [liveBook Discussion forum](#) on what's been written so far [and what you'd like to see in the rest of the book]. Your feedback will be invaluable in improving *GitHub Actions in Action*.

Thanks again for your interest and for purchasing the MEAP!

—Michael Kaufmann, Rob Bos, Marcel de Vries

In this book

[welcome](#) [1 Introduction to GitHub Actions](#) [2 Hands-on: My first Actions Workflow](#) [3 Workflows](#) [4 GitHub Actions](#) [5 Runners](#) [6 Self-hosted runners](#) [7 Managing your self-hosted runners](#) [8 Continuous Integration \(CI\)](#)

1 Introduction to GitHub Actions

This chapter covers the following topics

- Getting an introduction to the GitHub universe
- Understanding what GitHub Actions and Workflows are
- Learning about the possibilities of GitHub Actions that go beyond CI/CD pipelines
- Understanding licenses and pricing for GitHub and GitHub Actions

GitHub is more than just a platform for hosting and sharing code. It has become the beating heart of the open-source community, with millions of developers from all over the world collaborating on projects of every type and size. Founded in 2008, GitHub has since grown to host over 200 million repositories and 100 million users, with a staggering 3.5 billion contributions made in the last year alone.

And now, with GitHub Actions, developers have access to a powerful and flexible toolset for automating their workflows, from *Continuous Integration (CI)* and *Continuous Deployment (CD)* to custom automation tasks and beyond. GitHub Actions is much more than just a CI/CD tool – it's a comprehensive automation platform that can help streamline your entire development workflow.

This book will show you how to make the most of GitHub Actions and take your development process to the next level. It is for everyone that wants to learn more about GitHub Actions – from complete beginners to already advanced users that want to bring their knowledge to the next level. You will learn how to use Actions effectively and secure, and it brings a lot of real-world examples for using it for CI/CD scenarios.

1.1 An introduction to the GitHub universe

At the core of GitHub lies the essential component of version control, namely *git*. This system has played a significant role in transforming the way in

which software is developed and is widely considered the standard for the versioning of code, which in this case does not just refer to program code. It includes infrastructure, configuration, documentation, and many other types of files. Git has risen to prominence due to its remarkable flexibility, which stems from its classification as a distributed version control system rather than a central one. As a result, developers can work while disconnected from the central repository, utilizing the full functionality of the version control system, and later synchronize changes with another repository. The efficacy of git's distributed architecture is attributed to its ability to store snapshots of files with changes in its database.

GitHub has extended beyond its function as a hosting platform for git and has evolved into a comprehensive DevOps platform that supports collaborative coding through asynchronous means, such as pull requests and issues. The platform's capabilities have expanded into six broad categories, which include:

- collaborative coding,
- planning and tracking,
- workflows and CI/CD,
- developer productivity,
- client applications, and
- security.

These categories encapsulate the key features that GitHub offers, making it a versatile and comprehensive DevOps platform that supports various stages of software development.

From its inception, GitHub has prioritized a developer-centric approach, resulting in a platform that places utmost importance on webhooks and APIs. Developers can leverage either the REST API or the graph API to manipulate all aspects of the GitHub platform. Authentication is also a straightforward process, and developers can use GitHub as an identity provider to access their applications. This user-friendly approach facilitates seamless integration with other tools and platforms, making GitHub a versatile option not only for open-source projects but also for commercial products. GitHub's extensive ecosystem, which boasts over 100 million users, comprises the entire open-source community, who collaborate to expand and enrich its functionality.

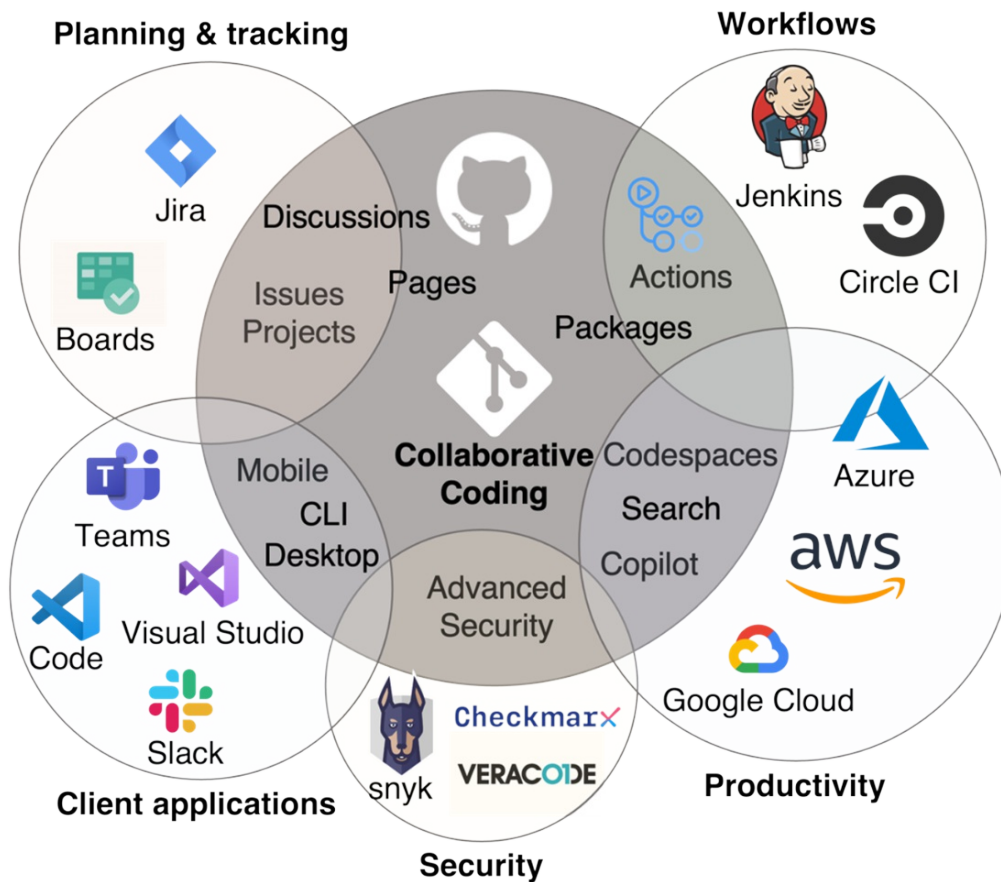
So, to understand the vastness of the GitHub ecosystem, one must also take into account the various integrations available:

- **Planning and tracking:** In addition to issues and milestones, GitHub offers GitHub Discussions, an entire forum for collaboration on ideas. Furthermore, GitHub Projects is a flexible planning solution that is fully integrated with issues and pull requests, and it supports nested backlogs, boards, and roadmaps. Moreover, GitHub integrates seamlessly with other popular planning and tracking solutions such as Azure Boards and Jira.
- **Client applications:** GitHub provides a fully-featured code editor that can be accessed directly in the browser. It also offers mobile applications for both iOS and Android platforms, enabling teams to collaborate from anywhere. Additionally, there is a cross-platform desktop application and an extensible CLI available. Furthermore, GitHub integrates smoothly with popular client applications such as Visual Studio, Visual Studio Code, and Eclipse. Moreover, it seamlessly integrates with popular chat platforms such as Slack and Teams.
- **Security:** GitHub provides a comprehensive solution for ensuring software supply-chain security, which includes several key features. For example, it generates Software Bills of Material (SBoMs) to keep track of all the components that are included in your software. And, with the Dependabot functionality, GitHub can alert you whenever vulnerabilities are detected in any of the dependencies you're using. Furthermore, GitHub can scan your repository to detect secrets, and it boasts a sophisticated code analysis engine called CodeQL. The platform also supports integrations with other security tools like Snyk, Veracode, or Checkmarx, and it can be integrated into Microsoft's Defender for DevOps.
- **Developer Productivity:** In GitHub, developers can quickly create a customized containerized development environment using GitHub Codespaces. This allows new developers to be productive right away. Additionally, Copilot, an AI-powered assistant, can generate code based on the context of comments or other code. This can significantly increase productivity, with reports of up to 50% gains. GitHub also offers code search, the command palette, and other features that can further enhance developer productivity.

- **Workflows and CI/CD:** In the world of continuous integration and continuous delivery (CI/CD), GitHub is a popular platform that enjoys widespread support from most CI/CD tools in the market. Furthermore, GitHub provides a secure integration with all the major cloud providers for CI/CD workflows using Open ID Connect (OIDC). This ensures a secure and streamlined experience for developers who rely on cloud-based services. Additionally, GitHub Packages is equipped with a robust package registry that supports a wide range of package formats, providing a powerful and versatile tool for developers to manage and distribute their code packages.

GitHub Actions serves as the automation engine for the GitHub ecosystem (see *Figure 1.1*). It allows users to automate various tasks, with a vast library of over 18,000 actions available in the marketplace. From issue triaging to automatic documentation generation, there is a building block – called **Action** – available to address nearly any task. With GitHub Actions, users can easily and securely automate their workflows.

Figure 1.1 The GitHub ecosystem has thousands of integrations



That's why GitHub Actions is more than just CI/CD. It is an automation engine that can be used to automate any kind of manual tasks in engineering, and it is already used by millions of developers worldwide. It can be used to not only automate GitHub – but the entire GitHub Universe.

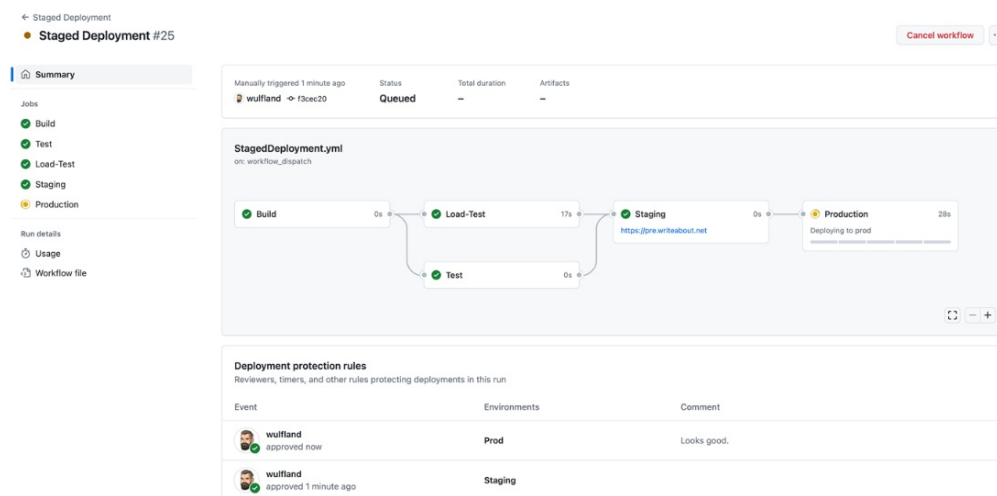
1.2 What are GitHub Actions and Workflows

GitHub Actions is both the name of the workflow engine and the name of an individual, reusable, and easily sharable workflow step within GitHub. This can lead to confusion. Workflows are composed of YAML files that are stored in a specific repository location (`.github/workflows`). In *Chapter 3, GitHub Action Workflows*, you will gain a comprehensive understanding of GitHub Action workflows and the YAML syntax. *Triggers* initiate the workflow, and one or more *jobs* are included in the workflow. Jobs are executed on a workflow runner, which can be a machine or container with an installed runner service. GitHub offers runners with Linux, macOS, and Windows operating systems in various machine sizes, but you can also host

your own runners. In *Part 2, Workflow Runtime*, you will learn about runners and the essential security measures to consider when hosting your own runners. Jobs execute in parallel by default, but the `needs` property can be used to chain jobs together. This enables you to fan out your workflow and run multiple jobs in parallel while waiting for all parallel jobs to complete before proceeding.

Environments in GitHub Actions provide a way to protect jobs by defining protection rules such as manual approvals, wait timers, and protected secrets. With this, you can create visual workflows that track for example your entire release pipeline, giving you complete control over your deployment process. Please refer to *Figure 1.2* for an example of a workflow with environments and approvals.

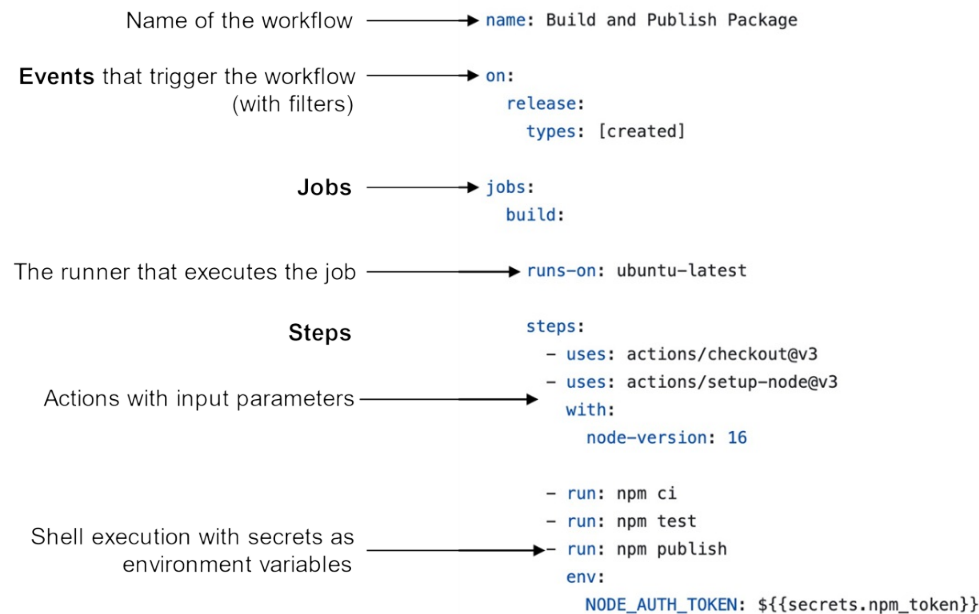
Figure 1.2 A GitHub Workflow with environments and approvals



A job is comprised of one or more *steps* that are executed sequentially. A step can take the form of a command line, script, or reusable step that is easily shareable, known as a GitHub Action. These Actions can be authored in JavaScript or TypeScript and executed in a NodeJS environment. Additionally, it is possible to run containers as Actions or create composite Actions that serve as a wrapper for one or multiple other Actions. Actions are covered in more depth in *Chapter 4, GitHub Actions*.

In *Figure 1.3* you can see an overview of the basic elements that make up a workflow and their syntax.

Figure 1.3 The basic syntax and elements that make up a GitHub Actions workflow



You will learn more about workflow syntax, YAML, GitHub Actions, and authoring and debugging workflows and Actions in *Chapter 3, GitHub Action Workflows*, and in *Chapter 4, GitHub Actions*.

1.3 GitHub Actions - More than CI/CD pipelines

GitHub workflows are intended to automate various tasks. In addition to pushing code, there are numerous triggers available. A workflow can be activated when a label is added to an issue, when a pull request is opened, or when a repository is starred.

In *Listing 1.1* you can find an example workflow that applies labels to opened or edited issues based on the content of the body of the issue.

Listing 1.1 A sample GitHub Actions workflow that can help triage GitHub issues

```
name: Issue triage
on:
  issues:
    types: [opened, edited]

jobs:
```

```

triage:
  runs-on: ubuntu-latest
  steps:
    - name: Label issue
      run: |
        if (contains(github.event.issue.body, 'bug')) {
          echo '::add-labels: bug';
        } else if (contains(github.event.issue.body, 'feature')) {
          echo '::add-labels: feature';
        } else {
          echo 'Labeling issue as needs-triage';
          echo '::add-labels: needs-triage';
        }

```

This is just an example to show you the power of GitHub actions.

GitHub does not automatically download or clone your repository when a workflow is executed. In many automation scenarios, the repository's code or files may not be required, and the workflow can be completed much faster without cloning the repository. If you intend to utilize GitHub Actions for CI/CD purposes, the first step in a job should be to download the code by utilizing the checkout action.:

```

steps:
  - name: Checkout repository
    uses: actions/checkout@v3

```

This action will clone your repository, allowing you to build and test your solution.

In *Part 3, CI/CD with GitHub Actions*, you will learn in depth how to use GitHub Actions for CI/CD in a secure and compliant way.

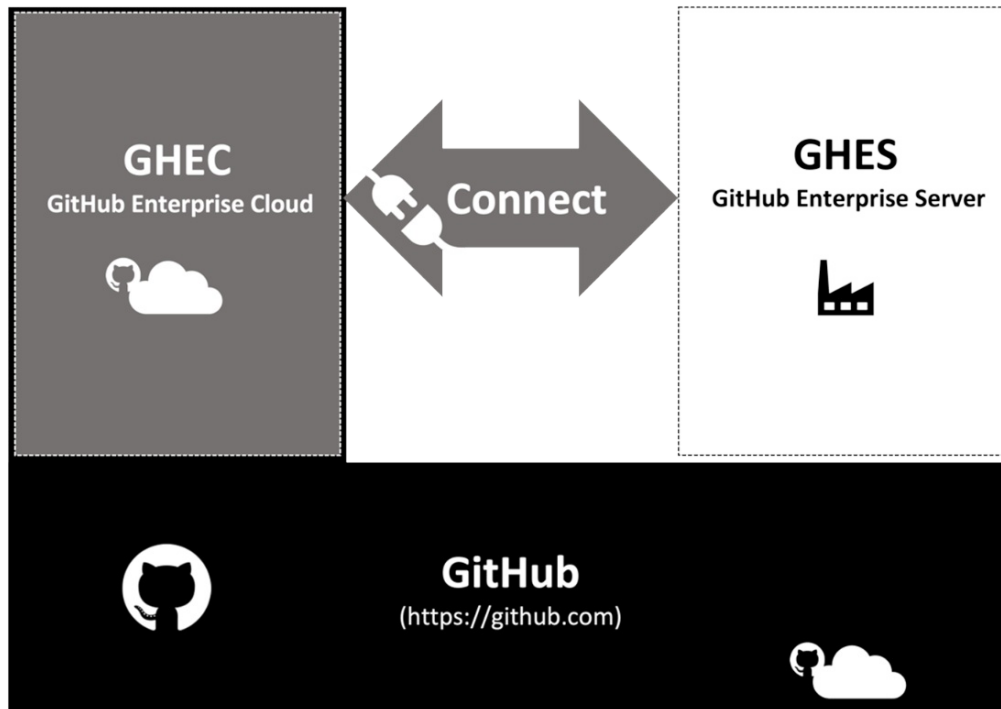
1.4 Hosting and pricing for GitHub and GitHub Actions

GitHub (<https://github.com>) is hosted in data centers located in the United States. Signing up for GitHub is free and provides users with unlimited private and public repositories. While many features on GitHub are available for free on open-source projects, they may not be available for private

repositories.

Enterprises have a variety of options for hosting GitHub (see *Figure 1.4*).

Figure 1.4 GitHub Enterprise Cloud, GitHub Enterprise Server, and GitHub Connect



1.4.1 GitHub Enterprise Cloud

GitHub **Enterprise Cloud (GHEC)** is a Software as a Service (SaaS) offering from GitHub, and it is fully hosted on its cloud infrastructure in the United States. GHEC provides additional security features and supports single sign-on for users. With GHCE, users can host private and public repositories, including open-source projects within their enterprise environment.

GHEC guarantees a monthly uptime Service Level Agreement (SLA) of 99.9%, which translates to a maximum downtime of 45 minutes per month.

1.4.2 GitHub Enterprise Server

The **GitHub Enterprise Server (GHES)** is a system that can be hosted

anywhere, either in a private data center or in a cloud environment like Azure, AWS, or GCP. Using GitHub Connect, it is possible to connect to GitHub.com, which enables the sharing of licenses and the use of open source on the server.

GHEC is based on the same source as GHEC, which means that all features eventually become available on the server a few months later. However, some features provided in the cloud must be managed independently on GHEC. For instance, runners in GitHub Actions require self-hosted solutions, whereas the cloud provides GitHub-hosted runners.

Managed Services are also available that provide hosting for GHEC, including in an Azure data center within the user's region. This approach ensures full data residency and eliminates the need to manage the servers personally. Some Managed Services also include hosting for managed GitHub Actions runners.

1.4.3 Pricing for GitHub

It is important to Understand the pricing model of GitHub and GitHub Actions when you start playing around with them, to not accidentally burn though all you free minutes.

GitHub's pricing model is based on a monthly per-user billing system and consists of three tiers: Free, Team, and Enterprise (see *Figure 1.5*).

Figure 1.5 Overview of GitHub pricing triers

Free	Team	Enterprise
\$ 0 per user/month	\$ 4 per user/month	\$ 21 per user/month
<ul style="list-style-type: none"> ✓ Unlimited public and private repositories ✓ Public repositories: <ul style="list-style-type: none"> ✓ Actions free ✓ Packages free ✓ Private repositories: <ul style="list-style-type: none"> ✓ 2,000 Actions minutes ✓ 500MB Package storage ✓ Dependency graph ✓ Dependabot 	<ul style="list-style-type: none"> ✓ 3,000 GitHub Actions minutes ✓ 2GB Package storage ✓ Access to Codespaces ✓ Protected branches ✓ Codeowners ✓ Advances pull request features 	<ul style="list-style-type: none"> ✓ 50,000 GitHub Actions minutes ✓ 50GB Package storage ✓ Server and Cloud ✓ GitHub Connect ✓ Single sign-on (SAML, LDAP) ✓ IP allow list ✓ Enterprise Managed Users ✓ SCIM ✓ Auditing / Policies <p>Available add-ons:</p> <ul style="list-style-type: none"> ✓ Premium Support ✓ Advanced Security

Public repositories, and therefore open-source projects, are entirely free of charge and offer many features such as Actions, Packages, and various security features. Private repositories are also available for free, but with limited functionality, including 2,000 Action minutes and 500 MB of storage per month.

To collaborate on private repositories with advanced features such as protected branches, codeowners, and enhanced pull request features, a team license is required. This license also includes access to Codespaces, although this feature requires a separate payment. Additionally, the team tier provides 3,000 free Action minutes per month and 2GB of monthly storage for packages.

Free and Team tiers are only available on GitHub.com. If users require GitHub Enterprise Cloud or Server, the GitHub enterprise license must be purchased. This license includes all enterprise features, such as single sign-on, user management, auditing, and policies, along with 50,000 Action minutes and 50GB of storage for packages per month. It also allows for the purchase of additional add-ons, such as Advanced Security or Premium Support.

1.4.4 Pricing for GitHub Actions

Hosted runners are provided for free to users with public repositories. The amount of storage and monthly build minutes available to users depends on

their GitHub edition, as shown in *Table 1.1*:

Table 1.1 Included storage and minutes for the different GitHub editions

GitHub edition	Storage	Minutes	Max concurrent jobs
GitHub Free	500 MB	2,000	20 (5 for macOS)
GitHub Pro	1 GB	3,000	40 (5 for macOS)
GitHub Free for organizations	500 MB	2,000	20 (5 for macOS)
GitHub Team	2 GB	3,000	60 (5 for macOS)
GitHub Enterprise Cloud	50 GB	50,000	180 (50 for macOS)

If you have purchased GitHub Enterprise through your Microsoft Enterprise Agreement, it is possible to link your Azure Subscription ID to your GitHub Enterprise account. This will allow you to use Azure Billing to pay for additional GitHub Actions usage beyond what is already included in your GitHub edition.

It is important to note that jobs running on Windows and macOS runners consume more build minutes than those running on Linux. Windows consumes build minutes at a rate of 2x and macOS consumes build minutes at a rate of 10x, meaning that using 1,000 Windows minutes would use up 2,000 of the minutes included in your account, while using 1,000 macOS minutes would use up 10,000 minutes included in your account. This is due to the higher cost of build minutes on these operating systems.

Users can pay for additional build minutes on top of what is included in their GitHub edition, with the following build minute costs for each operating system:

- Linux: \$0.008
- macOS: \$0.08
- Windows: \$0.016

These prices are for the standard machines with 2 cores.

The charges for extra storage are uniform for all runners, set at \$0.25 per GB.

In *Chapter 5*, Runners, you will learn in more detail, how minutes and extra storage are calculated.

If you are a customer who is billed monthly, your account is subject to a default spending limit of \$0 (USD), which restricts the use of extra storage or build minutes. However, if you pay by invoice, your account is given an unrestricted spending limit by default.

If you set a spending limit above \$0, any additional storage or minutes utilized beyond the included amounts in your account will be invoiced until the spending limit is reached. By setting up a spending limit, the Enterprise Administrators will receive e-mail notifications at reaching 75%, 90%, and 100% of the spending limit, on top of the default notifications for utilizing the same percentages of the included minutes in their monthly plan.

You won't incur any costs when using self-hosted runners since you provide your own computing resources.

It is important that you are aware of the costs when playing around with workflows. Especially if you try certain triggers. Best is to just use public repos for training purposes – in this case the workflows are free of charge in any case.

1.5 Conclusion

In this chapter, you have learned about the GitHub ecosystem and the myriad of possibilities it offers for automating tasks, extending beyond just CI/CD, using GitHub Actions. You have become familiar with key terms and concepts related to Workflows and Actions, enabling you to better navigate and utilize these features. Additionally, you have explored the hosting options and pricing models available for both GitHub and GitHub Actions.

Moving forward, the next chapter will provide an opportunity for practical application, as you embark on writing your first workflow. This initial exercise will serve as a useful foundation before delving further into the syntax and nuances of GitHub Action Workflows, which will be covered in *Chapter 3*.

1.6 Summary

- The GitHub universe consists of a vast ecosystem of products, partners, and communities around the areas of collaborative coding, planning, and tracking, workflows and CI/CD, developer productivity, client applications, and security.
- GitHub Actions is a workflow engine to automate any kind of manual tasks in engineering in the GitHub ecosystem beyond CI/CD.
- GitHub Action *workflows* are YAML files in a repository in the folder `.github/workflows` and contain *triggers*, *jobs*, and *steps*.
- A *GitHub Action* is reusable workflow step that can be easily shared through the GitHub marketplace.
- *GitHub Actions* are free for public repositories and paid per minute for private ones if you use the GitHub hosted runners; but you have free included Actions minutes in all GitHub pricing tears.
- *Private runners are always free – but the pricing for hosted runners varies depending on the machine size and type.*

2 Hands-on: My first Actions Workflow

This chapter covers the following topics

- Creating a new workflow
- Using the workflow editor
- Using actions from the marketplace
- Running the workflow

Before we dive into the details of the workflow and YAML syntax in *Chapter 3, GitHub Workflows*, it's a good idea to familiarize ourselves with the workflow editor, gain some practical experience in creating a workflow, and test it out to see it in action. This hands-on approach will help us better understand the concepts and give us the ability to quickly try something out, if it is unclear. Don't worry if there are parts of the workflow syntax that you don't understand yet— we'll be covering those in detail in the upcoming chapters.

2.1 Creating a new workflow

Let's begin this hands-on lab by signing into your GitHub account. Then, follow the link <https://github.com/new> to create a new repository. To ensure you have unlimited Action minutes, create a new public repository in your user profile and name it *ActionInAction*. Initialize the repository with a readme so that we can retrieve the files in the workflow later on. Lastly, click on the *Create repository* button to complete the process (refer to Figure 2.1).

Figure 2.1 Creating a new repository

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Repository template

No template ▾

Start your repository with a template repository's contents.

Owner *



wulfland ▾

Repository name *

ActionsInAction

✓ ActionsInAction is available.

Great repository names are short and memorable. Need inspiration? How about [expert-journey](#)?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:



Add a README file

This is where you can write a long description for your project. [Learn more about READMEs](#).

Add .gitignore

.gitignore template: None ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files](#).

Choose a license

License: None ▾

A license tells others what they can and can't do with your code. [Learn more about licenses](#).

This will set `main` as the default branch. Change the default name in your [settings](#).

ⓘ You are creating a public repository in your personal account.

Create repository

The repository

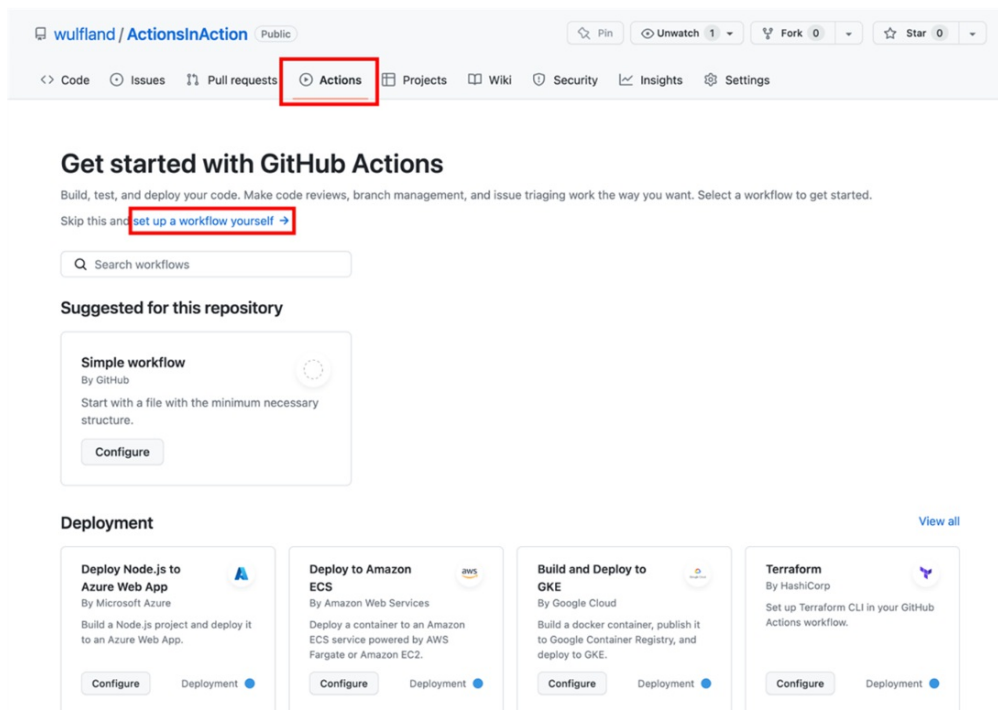
You can find companion repositories in the GitHub Organization

<https://github.com/GitHubActionsInAction> If you have already cloned the companion repository <https://github.com/GitHubActionsInAction/Part1>, you can also create a new workflow in this repository instead of creating a new one.

Now, let's navigate to the *Actions* tab inside the repository. If this is a new repository and there are no workflows set up yet, you will automatically be redirected to the new Action page (*Actions/new*). This is the same page you

would land on if you clicked the *New workflow* button in the workflow overview page, which is displayed if there are workflows in the repository. The new workflow page presents a plethora of templates for different languages and scenarios. You can certainly explore these available templates, but for our first workflow, we want to create the workflow ourselves. To proceed, simply click on the corresponding link as illustrated in *Figure 2.2*.

Figure 2.2 Set up a new workflow in the workflow editor

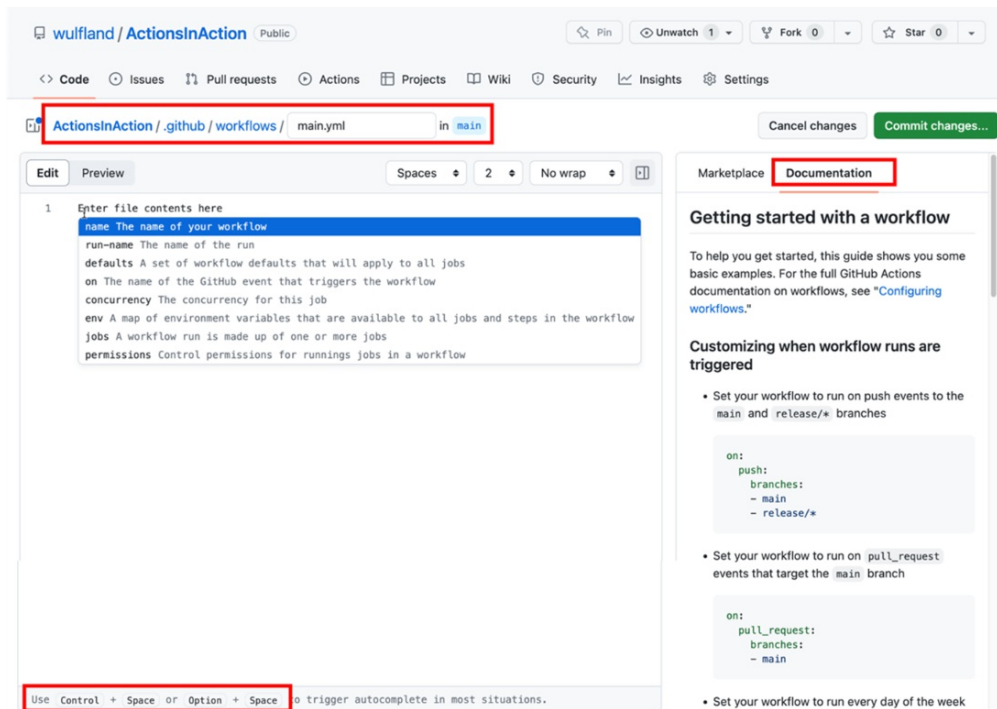


An empty workflow will be created and opened in the workflow editor.

2.2 Using the workflow editor

It's worth noting that a workflow is essentially a YAML file that resides in the folder `.github/workflows`. You can modify the filename as necessary from the top of the editor window. On the right side of the editor, you'll find the marketplace as well as the workflow documentation. The documentation provides valuable guidance to get you started. Moreover, the editor supports auto-complete when you use the `Ctrl+Space` keyboard shortcut. To give you a better idea of the key components of the editor, please refer to *Figure 2.3*.

Figure 2.3 The workflow editor



To begin, let's change the filename of the workflow file to `MyFirstWorkflow.yml`. Once that's done, click into the editor and open the auto-complete using `Ctrl+Space`. From the list of valid elements, choose `name`. The auto-complete feature will automatically add `name:` including the correct spacing to the file. Next, name the workflow *My First Workflow* and hit enter to start a new line.

Now, let's add triggers that will initiate the workflow. Begin a new line and press `Ctrl+Space` once again. From the options presented, select `on` and then `push`. Auto-complete will generate the following line, which will start the workflow upon any push in any branch:

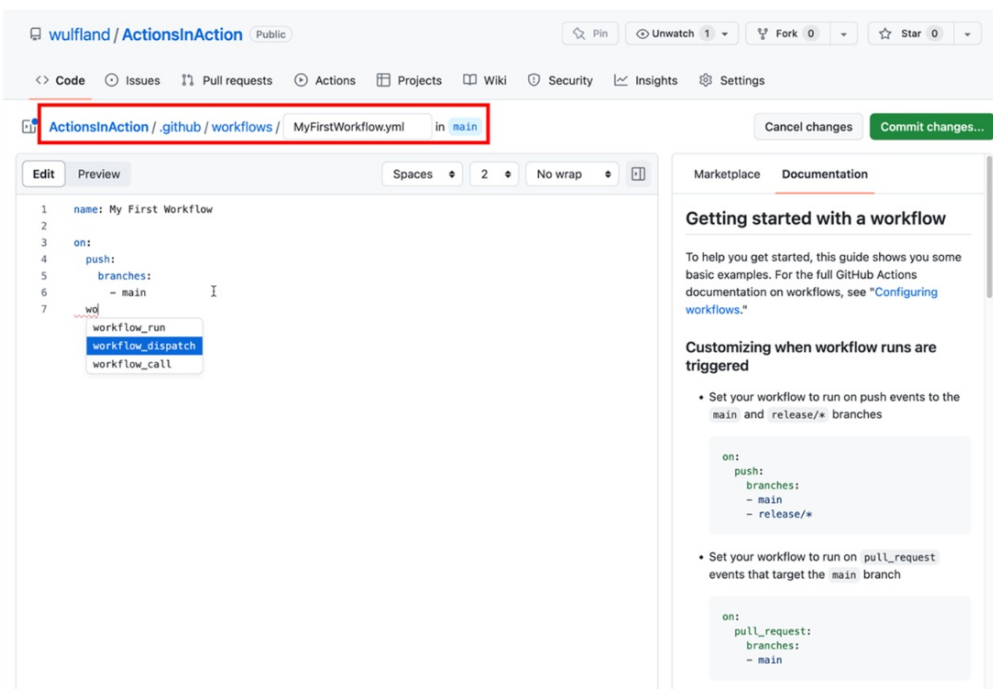
```
on: [push]
```

Suppose you want to trigger the workflow from only certain branches. In that case, you need to add additional parameters to the push trigger. First, delete the `[push]` and press enter to start a new line. Use the tab key to get the correct indentation. Next, press `Ctrl+Space` again, select `push`, and notice how auto-complete now functions differently. It will automatically create a

new line and offer all the available options for the push trigger. From there, choose branches and add the main branch, as shown in the documentation.

Create a new line with the same indentation as the push trigger and add a `workflow_dispatch` trigger, which will enable you to trigger the workflow manually. At this point, your workflow should resemble the one depicted in *Figure 2.4*.

Figure 2.4 Naming the workflow and adding triggers



To add a job to the workflow, create a new line in the workflow file with no indentation (same as name and on). Use auto-complete to write `jobs:` and move to the next line. Note that auto-complete won't work here, as the name of the job is expected. Enter `MyFirstJob:`, press enter to start a new line, and tab to indent one level. Auto-complete should work again now. Choose `runs-on` and enter `ubuntu-latest`, which will execute the job on the latest Ubuntu machine hosted by GitHub.

Next, add a step to the job. If you choose steps from auto-complete, it will insert a small snippet with a YAML array that you can use to enter your first step. For example, you can output `Hello world` to the console using `run` and `echo` (see Listing 2.1).

Listing 2.1 The first step outputs Hello World to the console

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

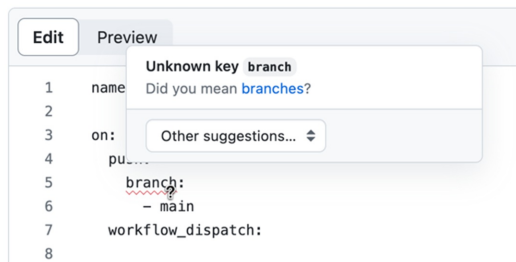
    steps:
      - run: echo "👋 Hello World!"
```

Error checking in the editor

It's important to note that if there are errors in your workflow file, the editor will mark the corresponding parts and you can hover over it with the mouse to get additional information and other suggestions (see Figure 2.5).

The editor will highlight structural errors, unexpected values, or even conflicting values, such as an invalid shell value for the chosen operating system.

Figure 2.5 Editor highlighting errors in the file and providing suggestions

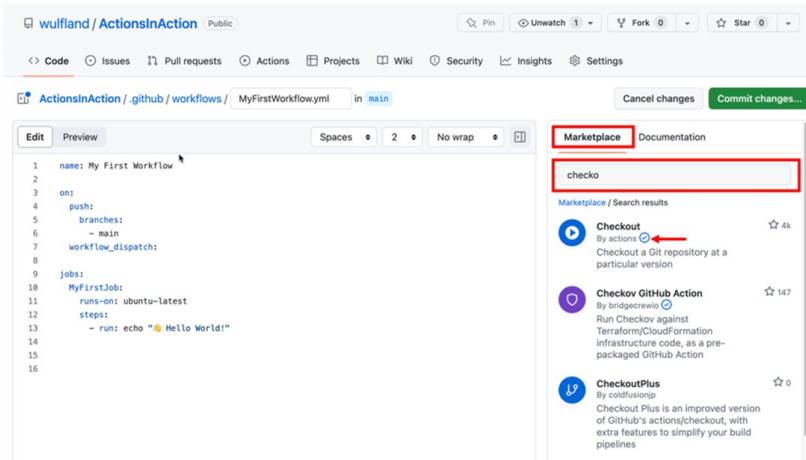


In the next step we will add a GitHub Action from the marketplace.

2.3 Using actions from the marketplace

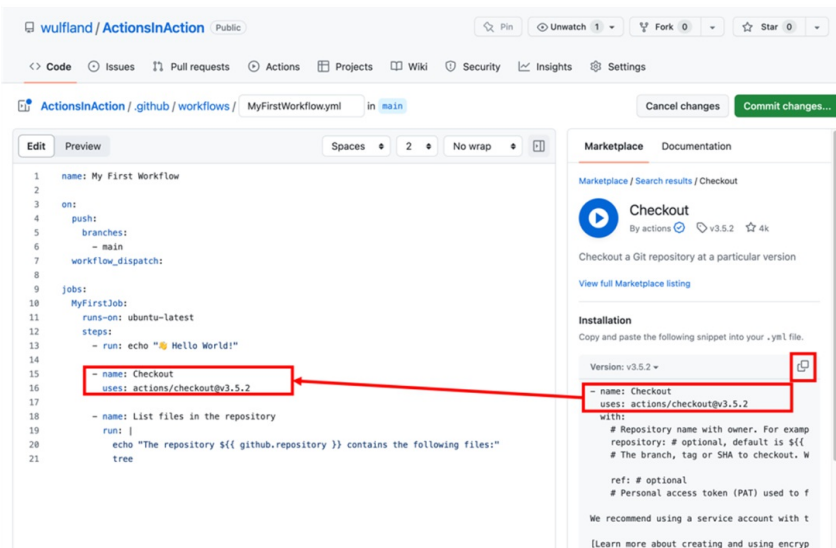
In the right pane, next to documentation, you can find the marketplace for GitHub Actions. To locate the Checkout action from GitHub Actions, start by typing checkout in the search bar (see Figure 2.6). Please note that the author of the Action is not GitHub, but *actions*, and that it has a blue badge with a checkmark, indicating that the author is a verified creator.

Figure 2.6 Searching in the marketplace from within the editor



If you click on the marketplace listing, you will be taken to a page with more details about the action. You can also copy the template using the copy button (see *Figure 2.7*) or copy parts of the YAML code snippet provided in the *Installation* section. The parameters for the action are under the `with:` property. They are all optional, so you can delete them all or just copy over `name:` and `uses:`. Paste the action as a step to the workflow like illustrated in *Figure 2.7*.

Figure 2.7 Adding the action from the marketplace to the workflow



As a last step, we add a script that displays the files in the repository using the `tree` command. Use the name property to set the name that is displayed in the workflow log. In this step we use a multi-line script using the pipe

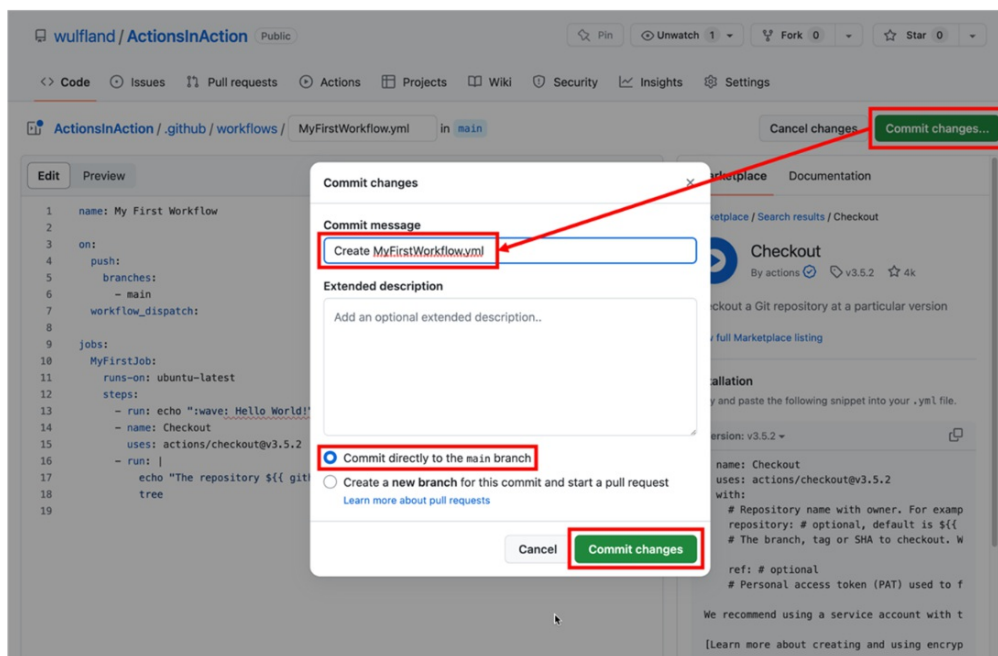
operator `|` and a two-blank indentation for the script. In the first line we output the name of the repository. We use an expression for that. We then use the `tree` command to output the files in the repository (see *Listing 2.2*):

Listing 2.2 Run a multiline script to display all files in the repository

```
- name: List files in repository
  run: |
    echo "The repository ${github.repository} contains the fo
    tree
```

If the editor does not indicate any errors, commit the workflow now to your main branch (see *Figure 2.8*).

Figure 2.8 Committing the new workflow file



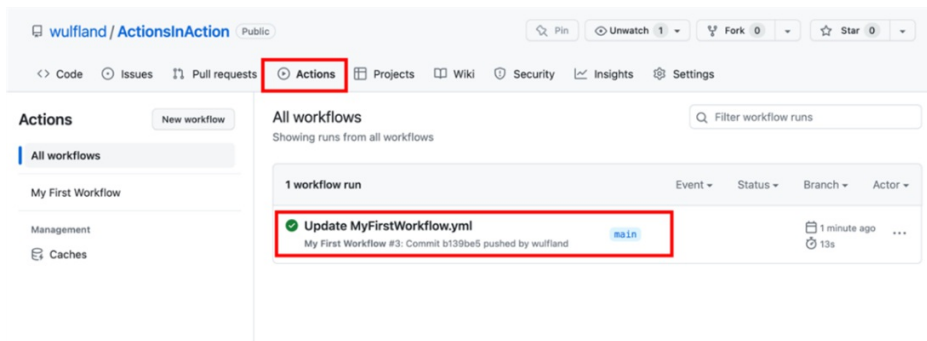
This will automatically trigger a workflow run because of the push trigger.

2.4 Running the workflow

The workflow will start automatically because of the push trigger on the main branch. To observe the workflow run, navigate to the *Actions* tab (see *Figure 2.9*). In the case of a push trigger, the name of the workflow run corresponds

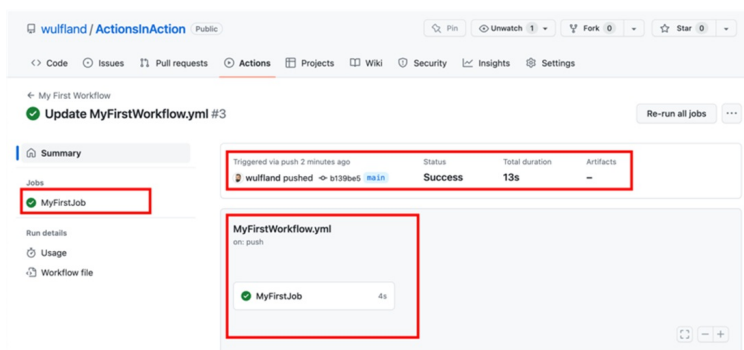
to the commit message. Additionally, you can view the branch on which the workflow was executed, as well as the time and duration of the run. Clicking on the workflow run will provide you with more detailed information.

Figure 2.9 The workflow runs in the Action tab



Within the workflow run overview page, you will come across a detail pane situated at the top, providing information about the trigger, status, and duration of the workflow. On the left-hand side, you will find a list displaying the jobs, while the workflow designer is located in the center (see *Figure 2.10*). Clicking on a specific job will redirect you to the corresponding job details page.

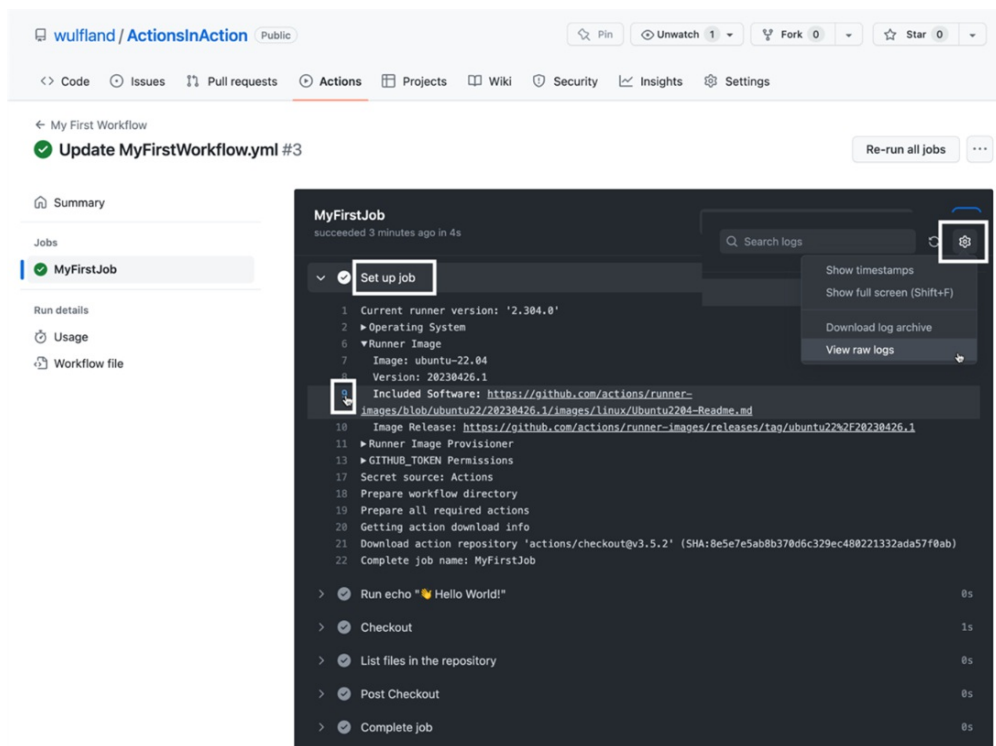
Figure 2.10 The workflow run overview



On the job details page, you will discover a log that allows you to track the progress of the running workflow. Each step within the workflow has its own collapsible section for easy navigation. Additionally, you will notice an additional Set up job section, providing additional details about the runner image, operating system, installed software, and workflow permissions.

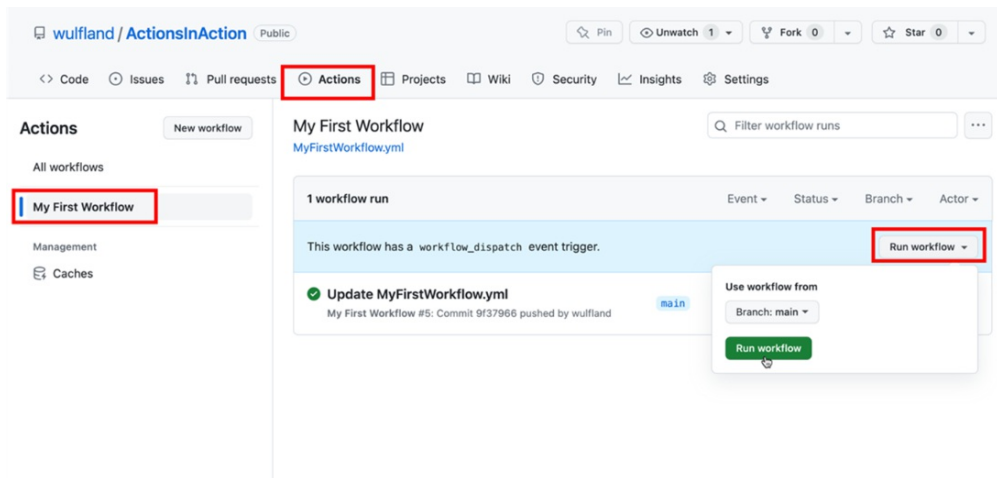
Each line in the workflow log is equipped with a deep link, enabling you to directly access a specific line within the log. In the top-right corner, you will find a settings menu where you can choose to display timestamps in the log or download the entire log for further analysis (see *Figure 2.11*).

Figure 2.11 The job details containing the workflow run log



With the inclusion of the `workflow_dispatch` trigger in your workflow, you now have the ability to manually run the workflow. To initiate the workflow manually, return to the **Actions** tab and select the workflow from the left-hand side, as illustrated in *Figure 2.12*. Once selected, you will encounter a **Run workflow** menu that you can utilize to trigger the workflow.

Figure 2.12 Triggering a workflow manually



While the workflow is starting, go to the workflow overview page and the job details page to observe the workflow in real time.

2.5 Conclusion

In this chapter, you have familiarized yourself with the workflow editor and have gained practical experience in creating and executing a workflow. You have also explored the documentation and incorporated a GitHub Action from the marketplace.

In the upcoming chapter, you will delve into the intricacies of YAML and workflow syntax. The chapter will provide comprehensive insights into advanced concepts, including expressions and workflow commands.

2.6 Summary

- New workflows are created under Actions/new.
- The workflow editor contains documentation and the marketplace.
- The workflow editor helps you writing the workflow with syntax-highlighting, auto-complete, and error checking.
- You can simply copy and paste actions from the marketplace into you workflow to use them.
- The workflow has a live-log with deep-linking that provides all the information for the workflow run.

3 Workflows

This chapter covers the following topics

- Understanding YAML and the YAML syntax
- Learning the basics of the workflow syntax
- Understanding workflow triggers, expressions, and contexts
- Getting an introduction into advanced workflow concepts like workflow commands
- Learning best practices for authoring and debugging workflows

Now that you have gained the first practical experience, it is time to fully understand the syntax for workflows.

Workflows are written in YAML. That's why it is really important to fully understand YAML before writing workflows.

3.1 YAML

YAML stands for *YAML Ain't Markup Language* and is a data-serialization language optimized to be directly writable and readable by humans. It is a strict superset of JSON but with syntactically relevant newlines and indentation instead of braces. In the next sections we go through all the YAML elements that are important for writing workflows.

3.1.1 YAML Basics

YAML files are text files and have a `.yaml` or `.yml` extension. Because YAML uses indentation instead of braces these text files can be versioned very well with git as changes are always per line.

YAML files can have different encodings – but GitHub uses UTF-8 for the workflows. You can write comments in YAML by prefixing text with a hash (`#`):

```
# A full-line comment in YAML
key:value # A in-line comment
```

Comments can occur anywhere in a line.

3.1.2 Data types

In YAML you have various data types available. There are simple (scalar) data types and more complex collection types.

Scalar types

In YAML you can assign a value to a variable with the following syntax.

```
key: value
```

The key is the name of the variable. Depending on the data type of value the type of the variable will be different. In *Listing 3.1* you can see the syntax for all basic data types: integer, float, string, Boolean, and datetime. Please note that in the listing the key is just the name of the variable. So, `age: 42` will assign the value 42 to an integer variable called age.

Listing 3.1 assigning basic scalar types to variables in YAML

```
integer: 42
float: 42.0
string: a text value
boolean: true
null value: null
datetime: 1999-12-31T23:59:43.1Z
```

Types in YAML

Types in YAML are more complex. For example, the datetime format – called timestamp in YAML - can be written in multiple forms. But I barely see this relevant for authoring workflows. If you want to learn more about types in YAML please see the documentation on <https://yaml.org/type>.

Note that keys *and* values *can contain spaces and do not need quotation!*

You can quote keys and values with single or double quotes, but you only have to do so if they contain some special characters or if the characters would indicate to YAML a wrong data type. Double quotes use the backslash as the escape pattern, single quotes use an additional single quote for this:

```
'single quotes': 'have ''one quote'' as the escape pattern'
"double quotes": "have the \"backslash \" escape pattern"
```

Especially for writing scripts in YAML workflows, it is important to understand this.

String variables can also span multiple lines using the pipe operator and a four spaces indentation. The multi-line text block can also contain line breaks and empty lines and continues until the next element:

```
literal_block: |
    Text blocks use four spaces as indentation. The entire
    block is assigned to the key 'literal_block' and keeps
    line breaks and empty lines.

    The block continuous until the next YAML element with the sam
    indentation as the literal block.
```

This makes writing complex scripts in YAML workflows much easier than in other formats where you must quote variables.

Collection types

In YAML there are two different collection types: nested types called maps and lists – also called sequences.

Maps use two spaces of indentation and the same syntax as assigning variables:

```
parent:
  key1: value1
  key2: value2
  child:
    key1: value1
```

Since YAML is a superset of JSON you can also use the JSON syntax to put

maps in one line:

```
parent: {key1: value1, key2: value2, child: {key1: value1}}
```

A sequence is an ordered list of items and has a dash before each line:

```
sequence:  
  - item1  
  - item2  
  - item3
```

You can also write this in one line using the JSON syntax:

```
sequence: [item1, item2, item3]
```

Learn more about YAML

This is just the tip of the iceberg and there is so much more you can learn about YAML. For working with GitHub Action workflows many topics are not relevant. Topics like file directives (---), tags, the different syntax variations for scalar types, like datetime or decimal, and folded literal block (with > instead of |) are not needed for writing workflows effectively. If you want to dive deeper in the YAML syntax you can go to:

<https://yaml.org/spec/1.2.2/#13-terminology>.

This is enough YAML knowledge to understand the workflow syntax.

3.2 The workflow syntax

The first element in a workflow file is typically the name of the workflow. The workflow can have a different name than the workflow file itself. In the example in *Chapter 2, Hands-on: My first Actions Workflow*, the workflow file is named `MyFirstWorkflow.yml` – but the workflow itself is named `My First Workflow`. The name is set using the `name` property:

```
name: My First Workflow
```

This is just a convention. You could also start the workflow file with one of the other valid root elements. The `name` property is typically followed by the

triggers that start the workflow.

You also might want to add a comment on top of the workflow to document what the workflow does.

3.3 Events and triggers

There are three categories of triggers:

- Webhook triggers
- Scheduled triggers
- Manual triggers

All triggers follow the key `on:` in the workflow file.

3.3.1 Webhook triggers

Webhook triggers start the workflow based on an event in GitHub. This can be a `git push` to the repository:

```
on: push
```

It can also be a pull request in the repo:

```
on: [push, pull_request]
```

Most webhook triggers can be configured to only start the workflow on certain conditions. You can, for example, start a workflow only when pushing to certain branches, or pushing when certain files in a path (paths) have been updated. The following example will only trigger the workflow, when files in the `doc` folder have changed and the changes are pushed to the main branch or a branch starting with `release/`:

```
on:
  push:
    branches:
      - 'main'
      - 'release/**'
    paths:
```

- 'doc/**'

Note

The character `*` is a special character in YAML so you have to quote all strings that contain values with wildcards.

There are many webhook triggers available. For example, you could run a workflow on an issues event. Supported activity type filters are: opened, edited, deleted, transferred, pinned, unpinned, closed, reopened, assigned, unassigned, labeled, unlabeled, locked, unlocked, milestoned, and demilestoned. Any time one of these events happen to an issue, it will trigger the workflow to run.

You can also run a workflow when your repository is starred (watch), a branch_protection_rule is created, edited, or deleted, or when you repository visibility is changed from private to public. For a complete list of the events that can trigger workflows, please refer to:

<https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows>.

3.3.2 Scheduled triggers

Schedule triggers allow you to start a workflow at a scheduled time. They use the same syntax as cron jobs. The syntax consists of five fields that represent the minute (0 – 59), hour (0 – 23), day of month (1 – 31), month (1 – 12 or JAN – DEC) and day of week (0 – 6 or SUN-SAT). You can use the operators in *Table 3.1*:

Table 3.1 Operators for scheduled events

Operator	Description
*	Any value
,	Value list separator if you specify multiple values
-	Range of values (from – to)
/	Step values

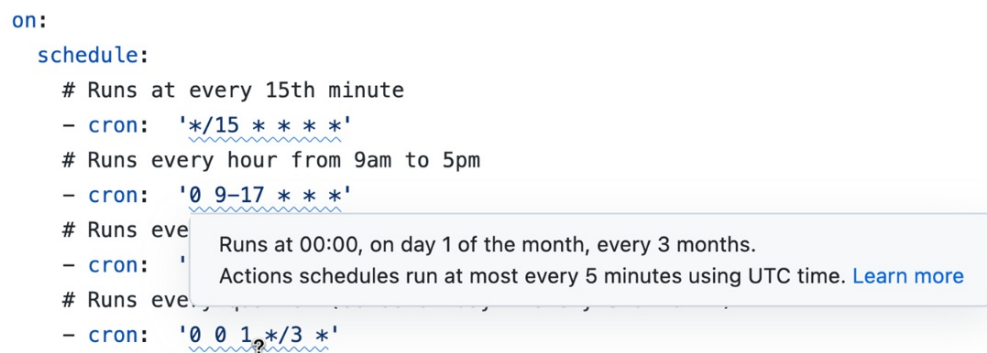
In *Listing 3.2* you can find some examples of scheduled triggers and when and how often they would be triggered.

Listing 3.2 Examples for scheduled workflows

```
on:
  schedule:
    # Runs at every 15th minute
    - cron: '*/*15 * * * *'
    # Runs every hour from 9am to 5pm
    - cron: '0 9-17 * * *'
    # Runs every Friday at midnight
    - cron: '0 0 * * FRI'
    # Runs every quarter (00:00 on day 1 every 3rd month)
    - cron: '0 0 1 */3 *'
```

As you can see in the examples, you can combine multiple schedule triggers in the same workflow, which can be helpful if you have a combination of multiple timings. The workflow designer is a great help when writing scheduled triggers as it will translate the cron job syntax into a human readable string (see *Figure 3.1*).

Figure 3.1 The workflow editor translates the cron job syntax into a human readable string



```
on:
  schedule:
    # Runs at every 15th minute
    - cron: '*/*15 * * * *'
    # Runs every hour from 9am to 5pm
    - cron: '0 9-17 * * *'
    # Runs eve
    - cron: '0 0 1 */3 *'
```

3.3.3 Manual triggers

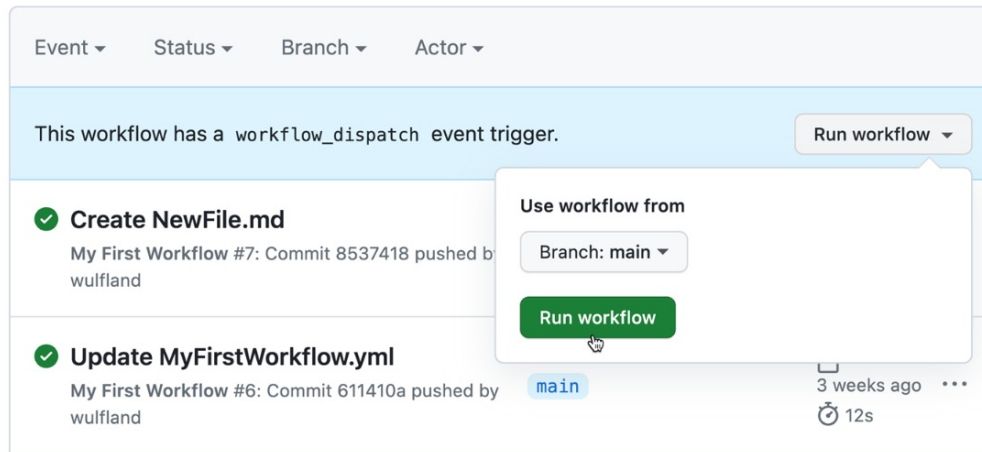
Manual triggers allow you to start a workflow manually. To do this using the GitHub UI or CLI you can use the `workflow_dispatch` trigger:

```
on: workflow_dispatch
```

The trigger always accepts one input: the branch the workflow runs on. The value defaults to the default branch of the repository – normally `main`.

In the GitHub UI you can trigger the workflow with the dialog displayed in *Figure 3.2*.

Figure 3.2 Triggering a workflow manually



You can also trigger the workflow using the GitHub CLI, either by *name*, *id*, or *filename* relative to `.github/workflow`:

```
$ gh workflow run WORKFLOW_FILENAME
```

The name of the workflow might contain blanks. That means you must quote it on the command line. The workflow id can be obtained by running `gh workflow list`. But the most practical approach is normally the name of the workflow file.

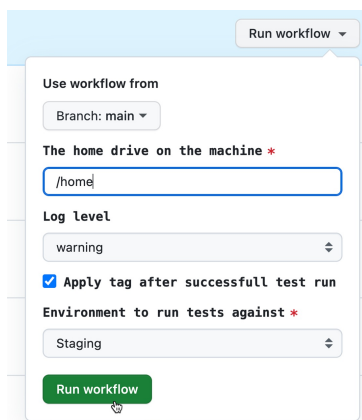
You can configure custom input arguments for a manual workflow start. The inputs can be required, optional, or you provide default values. They can be of the type string, Boolean, or choice. For choice you provide a fix list of values that are allowed. There is also the special type *Environment* that will give you a choice field over all environments found in the repository. (Environments have to be created manually in the repository. You will learn more about secrets and Environments later in this chapter). In *Listing 3.3* you can see an example that provides different custom inputs for a manual trigger.

Listing 3.3 Custom inputs for the workflow_dispatch trigger

```
workflow_dispatch:
  inputs:
    homedrive:
      description: 'The home drive on the machine'
      required: true
    logLevel:
      description: 'Log level'
      default: 'warning'
      type: choice
      options:
        - info
        - warning
        - debug
    tag:
      description: 'Apply tag after successfull test run'
      required: true
      type: boolean
    environment:
      description: 'Environment to run tests against'
      type: environment
      required: true
```

If the workflow is triggered through the user interface, the inputs are entered in a generated form like in *Figure 3.3*.

Figure 3.3 Providing custom-defined input when starting a workflow in the UI

A screenshot of the GitHub Actions 'Run workflow' form. At the top, there is a 'Run workflow' button with a dropdown arrow. Below it, the form is titled 'Use workflow from' and shows 'Branch: main' in a dropdown. The first input is 'The home drive on the machine' with a red asterisk, containing the text '/home/'. The second input is 'Log level' with a dropdown menu showing 'warning'. The third input is a checkbox labeled 'Apply tag after successfull test run' which is checked. The fourth input is 'Environment to run tests against' with a red asterisk and a dropdown menu showing 'Staging'. At the bottom, there is a green 'Run workflow' button.

If you trigger the workflow using the CLI, it will prompt you for the inputs. Alternatively, you can pass the inputs to the command using the `-f` (`--field`) argument:

```
$ gh workflow run MyFirstWorkflow.yml -f homedrive=/home -f logLe
```

In the case that you already have the input in JSON format, you can pipe it into the command using the standard input together with the `--json` switch:

```
$ echo '{"homedrive":"/home", "environment":"Staging", "tag":"tru
```

In the workflow the values of the inputs can be accessed using the `inputs` context:

```
steps:
  - run: |
      echo "Homedrive: ${ inputs.homedrive }"
      echo "Log level: ${ inputs.logLevel }"
      echo "Tag source: ${ inputs.tag }"
      echo "Environment ${ inputs.environment }"
      name: Workflow Inputs
      if: ${ github.event_name == 'workflow_dispatch' }
```

You will learn more about context and expression syntax in the next section of this chapter.

Another manual trigger is the `repository_dispatch` trigger that can be used to start all workflows in the repository that listen to that trigger using the GitHub API. This trigger can be used for integration scenarios with other systems.

If added to a workflow, the trigger can have one or more event types that can then specified when calling the API if you only want to trigger certain workflows:

```
on:
  repository_dispatch:
    types: [event1, event2]
```

The API endpoint is

`https://api.github.com/repos/{owner}/{repo}/dispatches` and you provide the event type in the following way:

```
$ curl \
  -X POST \
  -H "Accept: application/vnd.github.v3+json" \
```

```
https://api.github.com/repos/{owner}/{repo}/dispatches \
-d '{"event_type":"event1"}
```

You can also pass in additional JSON as a `client_payload`:

```
{
  "event_type": "event1"
  "client_payload": {
    "passed": false,
    "message": "Error: timeout"
  }
}
```

The payload can then be access through the `github.event` context:

```
- run: |
  echo "Payload: ${ toJSON(github.event.client_payload) }"
  name: Payload
  if: ${ github.event_name == 'repository_dispatch' }
```

There are multiple ways that you can call the GitHub API. You can use *curl*, like in the example above. You can use the GitHub CLI:

```
$ gh api -X POST -H "Accept: application/vnd.github.v3+json" \
/repos/{owner}/{repo}/dispatches \
-f event_type=event1 \
-f 'client_payload[passed]=false' \
-f 'client_payload[message]=Error: timeout'
```

There is also an SDK for many programming languages called *octokit*. Here is an example on how to call the dispatch API in JavaScript:

```
await octokit.request('POST /repos/{owner}/{repo}/dispatches', {
  owner: '{owner}',
  repo: '{repo}',
  event_type: 'event1'
  client_payload: {
    passed: "false",
    message: "Error: timeout",
  },
})
```

If you want to learn more on working with the GitHub API, please refer to:

<https://docs.github.com/en/rest/guides/getting-started-with-the-rest-api>.

Workflow triggers are very important. If you chose the right triggers and configure them right, you need less complex workflow logic. But before we learn more about expressions and context we should first have a look at the main workflow elements: workflow jobs and steps.

3.4 Workflow jobs and steps

The logic of the workflow is configured in the jobs section. Every job is executed on a *runner*. The runner can be self-hosted, or you can pick one from the cloud. There are different versions available in the cloud for all platforms. If you want to always use the latest version you can use `ubuntu-latest`, `windows-latest`, or `macos-latest`. You'll learn more about runners in *Chapter 5, Runners*, and in *Chapter 6, Self-hosted runners*.

3.4.1 Workflow jobs

Jobs are a YAML map and not a list – and they run in parallel per default. You can chain them in a sequence by depending a job on the successful output of one or multiple other jobs using the `needs` keyword. *Listing 3.4* shows an example of 4 jobs. Two that run in parallel after the first job, and a last one that runs after the two parallel have finished:

Listing 3.4 Chaining of jobs

```
jobs:
  job_1:
    runs-on: ubuntu-latest
    steps:
      - run: "echo Job: ${ github.job }"

  job_2:
    runs-on: ubuntu-latest
    needs: job_1
    steps:
      - run: "echo Job: ${ github.job }"

  job_3:
    runs-on: ubuntu-latest
```

```

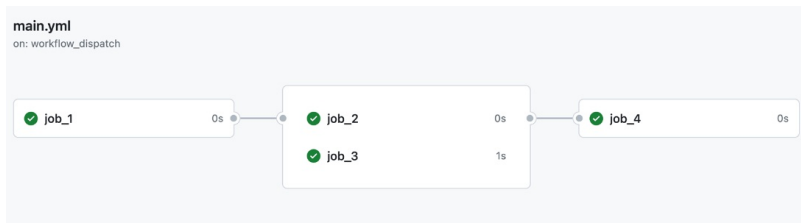
needs: job_1
steps:
  - run: "echo Job: ${github.job}"

job_4:
  runs-on: ubuntu-latest
  needs: [job_2, job_3]
  steps:
    - run: "echo Job: ${github.job}"

```

The resulting workflow would look like in *Figure 3.4*:

Figure 3.4 Visual representation of chained workflow jobs in GitHub



3.4.2 Workflow steps

A job contains a sequence of steps, and each step can run a command. Steps are always executed one after the other:

```

steps:
  - name: Install Dependencies
    run: npm install
  - run: npm run build

```

The name property is optional and defines how the step is displayed in the workflow log.

Literal blocks allow you to write multi-line scripts in one workflow step. If you want the workflow to run in a different shell than the default shell, you can configure it together with other values like the working-directory:

```

- name: Clean install dependencies and build
  run: |
    npm install
    npm run build
  working-directory: ./temp

```

```
shell: bash
```

The following shells are available (see *Table 3.2*):

Table 3.2 Available shells in GitHub workflows

Parameter	Description
bash	Bash shell. The default shell on all non-Windows platforms with a fallback to <code>sh</code> . When specified on Windows, the bash shell included with Git is used.
pwsh	PowerShell Core. Default on the Windows platform.
python	The python shell – allows you to run python scripts
cmd	Windows only! The windows command prompt.
powershell	Windows only! The classical Windows PowerShell.

The default shell on non-windows systems is *bash* with a fallback to *sh*. The default on windows is *pwsh* with a fallback to *cmd*.

You can also configure a custom shell with the with the syntax `shell: command [options] {0}`. The placeholder `{0}` will be replaced with the script you provide. Here is an example for running a *perl* script:

```
- run: print %ENV  
  shell: perl {0}
```

You will learn more about shells in *Chapter 5, Runners*.

3.4.3 Using GitHub Actions

Most of the time you want to use reusable steps – called *GitHub Actions*. You can reference an action using the `uses` keyword and the following syntax:

```
{owner}/{repo}@{ref}
```

The `{owner}/{repo}` is the path to the actions repository in GitHub. If you have multiple actions in a repository the syntax is the following:

```
{owner}/{repo}/{path}@{ref}
```

But in this case the action cannot be published to the marketplace.

The reference `{ref}` is the version of the action. It is a git reference to the point in time in the history of changes. The reference can be all kind of valid git references. A *tag*, a *branch*, or an individual commit referenced by its SHA-1 value. The most common is using tags for explicit versioning with major and minor versions.

```
# Reference a version using a tag
```

- uses: actions/checkout@**v3**
- uses: actions/checkout@**v3.5.2**

```
# Reference the current head of a branch
```

- uses: actions/checkout@**main**

```
# Reference a specific commit
```

- uses: actions/checkout@**8e5e7e5ab8b370d6c329ec480221332ada57f0ab**

If your action is in the same repository as the workflow, you can use a relative path to the action:

```
uses: ../github/actions/my-action
```

If the action has defined inputs, you can specify them using the `with` property:

- name: My first Action step
 uses: ActionsInAction/HelloWorld@v1
 with:
 WhoToGreet: Mona

Inputs can be optional or required. You can also set environment variables for steps using the `env` property:

- uses: ActionsInAction/HelloWorld@v1
 env:
 GITHUB_TOKEN: \${ secrets.GITHUB_TOKEN }
 WhoToGreet: Mona

You can also set variables for the entire workflow, or a job and they will automatically be available to the action.

Every docker container stored in a container registry like Docker Hub or GitHub Packages can be used as a step in the workflow using the syntax `docker://{image}:{tag}`:

```
uses: docker://alpine:3.8
```

This is very handy if you want to integrate existing solutions in docker into your workflows. Only limitation is that the container registry must be accessible for the workflow without credentials.

In *Chapter 4, GitHub Actions*, you will learn how to author GitHub Actions and you will learn how they work internally.

3.4.4 The strategy matrix

Jobs can be run with different configurations using the matrix strategy. The matrix can be a one-dimensional array and the workflow will execute one job for each value in the array. Furthermore, the matrix can consist of multiple arrays and the workflow will execute a job for all combinations of all values in the matrix. You can think of this as nested for-loops over all arrays.

The keys in the matrix can be anything and you refer to them using the expression `${{ matrix.key }}`. You can choose if the matrix should abort execution when an error occurs in one of the jobs in the matrix, or if it should continue executing the other jobs using the `fail-fast` property. The maximum number of jobs that run in parallel can be set using `max-parallel`. *Listing 3.5* shows an example that will run the same job for the NodeJS versions 12, 14, and 16 on ubuntu and macOS.

Listing 3.5 Executing jobs with different configurations

```
jobs:
  job_1:
    strategy:
      fail-fast: false
```

```

max-parallel: 3
matrix:
  os_version: [macos-latest, ubuntu-latest]
  node_version: [12, 14, 16]

name: My first job
runs-on: ${{ matrix.os_version }}
steps:
  - uses: actions/setup-node@v3.6.0
    with:
      node-version: ${{ matrix.node_version }}

```

This code will result in 6 jobs with all combinations and the workflow output will look like *Figure 3.5*. The job name will be suffixed with the values of the matrix.

Figure 3.5 Output of a job with multiple configurations



There is also the possibility to include or exclude some values for specific configurations. Please refer to <https://docs.github.com/en/actions/using-jobs/using-a-matrix-for-your-jobs> for the latest documentation.

3.5 Expressions and Contexts

You have already seen some expressions – in the first hands-on, when we had a look at manual triggers and the matrix strategy. An expression has the following syntax:

```
${{ <expression> }}
```

Expression can access context information and combine them with operators.

There are different objects available that provide context information, like `matrix`, `github`, `env`, `vars`, `needs`, `runner`, or `input`. With `github.sha`, for example, you can access the commit SHA that had triggered the workflow. With `runner.os` you can get the operating system of the runner and with `env` you can access environment variables. For a complete list see of all context object and all properties please refer to:

<https://docs.github.com/en/actions/reference/context-and-expression-syntax-for-github-actions#contexts>.

There are two possible syntaxes to access context properties:

```
context['key']  
context.key
```

The latter, the property syntax, is the more common.

Depending on the format of the key you might have to use the first option. This might be the case if the key starts with a number or contains special characters other than dash (-) and underscore (_).

Expressions are often used in the `if` object to run jobs or steps on different conditions. The following example will only execute the job `deploy` if the workflow was triggered by a push to `main`:

```
jobs:  
  deploy:  
    if: ${{ github.ref == 'refs/heads/main' }}  
    runs-on: ubuntu-latest  
    steps:  
      - run: echo "Deploying branch ${{ github.ref }}"
```

The expression must return `true` or `false` and can be used on steps and jobs to control the flow of the workflow by conditionally executing them.

To write expressions and compare context with static values you can use of the operators from *Table 3.3*.

Table 3.3 Operators for expressions

--	--

Operator	Description
()	Logical grouping
[]	Index
.	Property de-reference
!	Not
< , <=	Less than, less than or equal
> , >=	Greater than, greater than or equal
==	Equal
!=	Not equal
&&	And
	Or

GitHub offers a set of built-in functions that you can use in expressions. They can help you searching in strings, formatting output, or working with arrays. See *Table 3.4* for a list of available functions.

Table 3.4 Built-in functions in GitHub for expressions

Function	Description
contains(search, item)	Returns true if search contains item. Examples: contains('Hello world', 'llo') returns true. contains(github.event.issue.labels.*.name, 'bug') returns true if the issue related to the event has a label bug .
startsWith(search, item)	Returns true when search starts with item.
endsWith(search, item)	Returns true when search ends with item.
format(string, v0, v1, ...)	Replaces values in the string. Example: format('Hello {0} {1} {2}', 'Mona', 'the', 'Octocat') returns 'Hello Mona the Octocat'.
join(array, optS)	All values in array are concatenated into a string. If you provide the optional

	separator optS , it is inserted between the concatenated values. Otherwise, the default separator ',' is used.
toJSON(value)	Returns a pretty-print JSON representation of value.
fromJSON(value)	Returns a JSON object or JSON data type for value.
hashFiles(path)	Returns a single hash for the set of files that matches the path pattern.

There are also some special functions to check the status of the current job. In the following example, the step displayed would only be executed if a previous step of the jobs has failed:

steps:

```
...
- name: The job has failed
  if: ${{ failure() }}
```

For a list of available function to check the status of the job see *Table 3.5*.

Table 3.5 Functions to check status of the workflow job

Function	Description
success()	Returns true if none of the previous steps have failed or been cancelled.
always()	Returns true even if a previous step was cancelled and causes the step to always get executed anyway.
cancelled()	Returns only true if the workflow was canceled.
failure()	Returns true if a previous step of the job had failed.

You can use the * syntax to apply *object filters* for arrays and objects. If you have an array of objects called fruits with the following values:

```
fruits=[
  { "name": "apple", "quantity": 1 },
```

```
{ "name": "orange", "quantity": 2 },  
{ "name": "pear", "quantity": 1 }  
]
```

The filter `fruits.*.name` returns the array `["apple", "orange", "pear"]` and the filter `fruits.*.quantity` returns `[1, 2, 1]`.

Expressions are a powerful way to control the flow and execution of your workflow and you will learn more examples in the rest of the book.

3.6 Workflow commands

Workflow steps and actions can communicate with the workflow and the runner machine using *workflow commands*. They can be used to write messages to the workflow log, pass values to other steps or actions, set environment variables, or write debug messages.

Workflow commands use the `echo` command with a specific format, or they are invoked by writing to a specific environment file.

```
echo "::workflow-command parameter1={data},parameter2={data}::{co
```

If you are using JavaScript, the *toolkit* (<https://github.com/actions/toolkit>) provides a lot of wrappers that can be used instead of using `echo` to write to `stdout`. For example, if you want to log an error the workflow log, you can use the following `echo` command:

```
- run: echo "::error file=app.js,line=1::Missing semicolon"
```

With the toolkit you can achieve the same in the following form:

```
core.error('Missing semicolon', {file: 'app.js', startLine: 1})
```

For a complete list of available workflow commands please refer to:

<https://docs.github.com/en/actions/using-workflows/workflow-commands-for-github-actions>.

In the subsequent sections you will learn some examples of useful workflow

commands.

3.6.1 Writing a debug message

You can print debug message to the workflow log. To see the debug messages set by this command in the log, you must create a variable named `ACTIONS_STEP_DEBUG` with the value `true`. You will learn later in this chapter how to set variables. The syntax is:

```
::debug::{message}
```

Debug messages are extremely useful to being able to debug your workflows without cluttering the log if you are not debugging.

3.6.2 Creating error or warning messages

The same way you can create warning and error messages and print them to the log. The messages will create an annotation, which can associate the message with a particular file in your repository. Optionally, your message can specify a position within the file.

```
::warning file={name},line={line},endLine={el},title={title}::{me}
::error file={name},line={line},endLine={el},title={title}::{mess}
```

The parameters are the following:

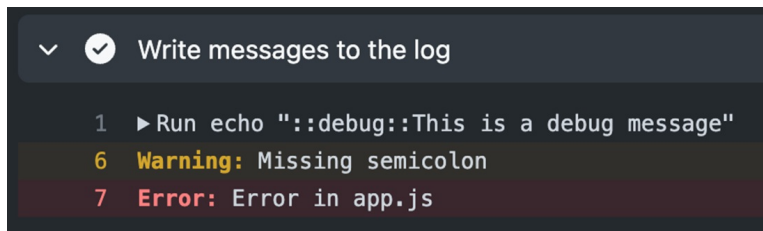
- *Title*: a custom title for the message
- *File*: the filename that raised the error or warning
- *Col*: Column / character number, starting at 1
- *EndColumn*: End column number
- *Line*: The line number in the file starting with 1
- *EndLine*: End line number

Here is an example how these two commands can be used:

```
echo "::warning file=app.js,line=1,col=5,endColumn=7::Missing sem
echo "::error file=app.js,line=1,col=5,endColumn=7::Error in app.
```

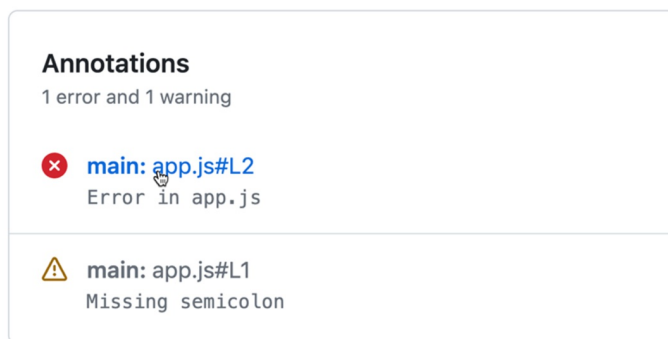
You can see the output of these commands in the log in *Figure 3.6*.

Figure 3.6 Warning and error messages in the workflow log



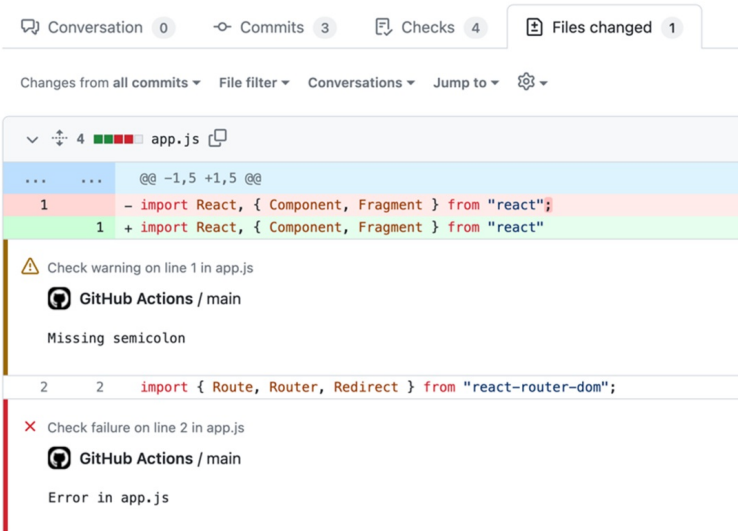
The annotations will be added to the workflow overview page and the link to the file is clickable (see Figure 3.7).

Figure 3.7 Annotations in the workflow overview page



The link will redirect you to the corresponding line in the file if it is part of the source commit of the workflow. If the workflow is associated with a pull request, then you can see the messages on the correct lines in the Files changed tab (see *Figure 3.8*).

Figure 3.8 Warning and error messages shown as pull request decorations



3.6.3 Passing output to subsequent steps and jobs

The syntax to pass output values to subsequent tasks is different. Instead of using a workflow command with echo you have to pipe a name-value pair to the *environment file* `$GITHUB_OUTPUT`:

```
echo "{name}={value}" >> "$GITHUB_OUTPUT"
```

The operator `>>` appends the name-value pair to the end of the file. The path and filename of the file are stored in the environment variable `$GITHUB_OUTPUT`. You can access the output using the output property of the step in the steps context:

- name: Set color
id: **color-generator**
run: echo "**SELECTED_COLOR**=green" >> "\$GITHUB_OUTPUT"
- name: Get color
run: echo "\${{ steps.**color-generator**.outputs.SELECTED_COLOR }}"

Outputs are Unicode strings and cannot exceed one MB in size. The total of all outputs in a workflow run cannot exceed 50 MB.

If you want to mask output in the log you can use `::add-mask::{value}`. This will mask the output in the log - even when you pass the value to other steps or jobs. The value will be preserved – only the output is masked. You

can find an example in *Listing 3.6* demonstrating that.

Listing 3.6 Masking secret values across multiple steps

```
on: push
jobs:
  generate-a-secret-output:
    runs-on: ubuntu-latest
    steps:
      - id: sets-a-secret
        name: Generate, mask, and output a secret
        run: |
          the_secret=$((RANDOM))
          echo "::add-mask::$the_secret"
          echo "secret-number=$the_secret" >> "$GITHUB_OUTPUT"
      - name: Use that secret output (protected by a mask)
        run: |
          echo "the secret number is ${ steps.sets-a-secret.outp
```

3.6.4 Environment files

During the execution of a workflow, the runner generates temporary files that you can manipulate to perform certain actions. The output file was one example. The paths to these files are exposed via environment variables – in this case `$GITHUB_OUTPUT`.

Another use case for environment file is setting an environment variable for subsequent steps in a job. The corresponding environment file is `$GITHUB_ENV`. And again, you just append another name-value pair to the end of the file:

```
echo "{environment_variable_name}={value}" >> "$GITHUB_ENV"
```

Note that the name is case sensitive! Here is a complete example how to set an environment variable in one step and access it in a subsequent step using the `env` context:

```
steps:
  - name: Set the value
    id: step_one
    run: |
      echo "action_state=yellow" >> "$GITHUB_ENV"
```

```
- run: |  
    echo "${{ env.action_state }}" # This will output 'yellow'
```

For a complete reference on environment files, please refer to:

<https://docs.github.com/en/actions/using-workflows/workflow-commands-for-github-actions?tool=bash#environment-files>

Another example for environment files is adding a job summary in a workflow.

3.6.5 Job summaries

You can set some custom Markdown for each workflow job. The rendered markdown will then be displayed on the summary page of the workflow run. You can use job summaries to display content, such as test or code coverage results, so that someone viewing the result of a workflow run doesn't need to go into the logs or an external system.

Job summaries support GitHub flavored markdown. But since Markdown is HTML, you can also output html to the job summary file.

Add results from your step to the job summary by appending markdown to the following file:

```
echo "{markdown content}" >> $GITHUB_STEP_SUMMARY
```

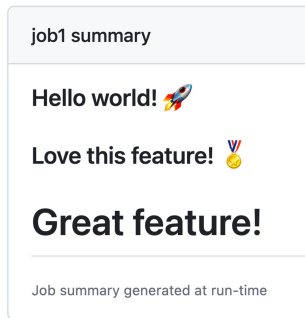
The steps are isolated and restricted to 1 MiB (1.04858 MB). It is isolated so that malformed Markdown from a single step cannot break Markdown rendering for subsequent steps. Only 20 steps can write to the summary, the output of any step after that will not be visible.

Here is an example that adds markdown and plain html to the job summary:

```
- run: echo '### Hello world! :rocket:' >> $GITHUB_STEP_SUMMARY  
- run: echo '### Love this feature! :medal_sports:' >> $GITHUB_ST  
- run: echo '<h1>Great feature!</h1>' >> $GITHUB_STEP_SUMMARY
```

The result looks like *Figure 3.9*.

Figure 3.9 Markdown and HTML displayed on the workflow summary page






If you have more complex scenarios or you are authoring your action in JavaScript anyway, then you can use the *toolkit* (<https://github.com/actions/toolkit>) function `core.summary` to write tables or links. *Listing 3.7* shows an example of that.

Listing 3.7 Writing a job summary using the toolkit

```
- name: Write Summary from Action
  uses: actions/github-script@v6.1.0
  with:
    script: |
      await core.summary
      .addHeading('Test Results')
      .addTable([
        [{data: 'File', header: true}, {data: 'Result', header: true}],
        ['foo.js', 'Pass ✓'],
        ['bar.js', 'Fail ✗'],
        ['test.js', 'Pass ✓']
      ])
      .addLink('View staging deployment!', 'https://github.com')
      .write()
```

The result will look like in *Figure 3.10*.

Figure 3.10 Job summary created by the toolkit

job2 summary	
Test Results	
File	Result
foo.js	Pass 
bar.js	Fail 
test.js	Pass 
View staging deployment!	
Job summary generated at run-time	

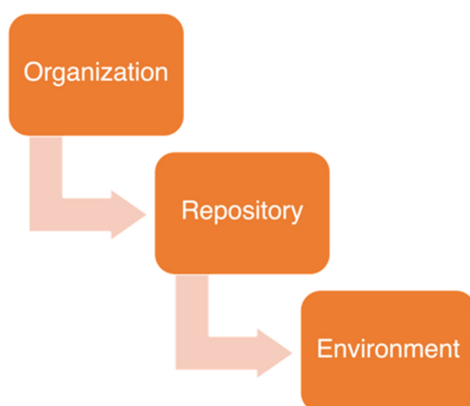
3.7 Secrets and variables

You can create configuration variables for use across multiple workflows by defining them on one of the following levels:

- Organization level
- Repository level
- Environment level

The three levels work like a hierarchy: you can override a variable or secret on a lower level by providing a new value to the same key. *Figure 3.11* illustrates the hierarchy.

Figure 3.11 The hierarchy for configuration variables and secrets



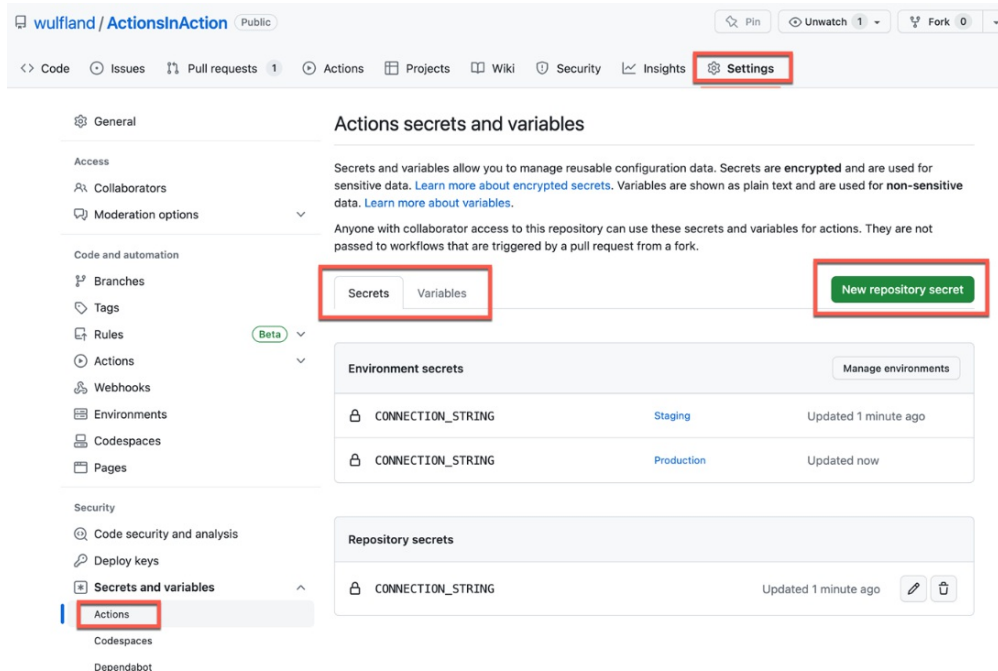
A special form of configuration variables are secrets. They are stored encrypted and are only decrypted at runtime. They are also protected and masked in the workflow log.

Secrets can be accessed using the secret context and variables using the vars context. Here is an example of how you can pass secrets and variables to a GitHub Action:

```
- name: Set secret and var as input
  uses: ActionsInAction/HelloWorld@v1
  with:
    MY_SECRET: ${ secrets.secret-name }
    MY_VAR: ${ vars.variable-name }
```

Secrets and variables can be set using the UI or CLI. You must be part of the admin role to do this. In the UI you can do this under **Settings | Secrets and variables | Actions** on the corresponding hierarchy level. In a repository you can set secrets with write access – but you have to use the CLI to do so as you have no access to the settings. There you can switch using the tabs between **Secrets** and **Variables** and you will find the **New repository secret** button (`/settings/secrets/actions/new`) or **New repository variable** button (`/settings/variables/actions/new`) that you can use to create new entries (see *Figure 3.12*).

Figure 3.12 Setting secrets and variables using the GitHub UI



When creating secrets or variables, please take into account the Naming

conventions for secrets and variables.

Naming conventions for secrets and variables

Secret names are not case-sensitive, and they can only contain normal characters ([a-z] and [A-Z]), numbers ([0-9]), and the underscore (_). They must not start with GITHUB_ or a number.

Best practice is to name the secrets with upper-case words separated by the underscore character.

Secrets and variables for organizations work the same way. Create the secret or variable under **Settings | Secrets and variables | Actions**. New organization secrets or variables can have an access policy to

- All repositories,
- Private repositories, or
- Selected repositories.

When choosing **Selected repositories**, you can grant access to individual repositories.

If you prefer the GitHub CLI, you can use `gh secret` or `gh variable` to set to create new entries:

```
$ gh secret set secret-name
```

```
$ gh variable set var-name
```

You will be prompted for the secret or variable value or you can read the value from a file, pipe it to the command, or specify it as the body (-b or --body):

```
$ gh secret set secret-name < secret.txt
```

```
$ gh variable set var-name --body config-value
```

If the entry is for an environment, you can specify it using the `--env (-e)` argument. For organization secrets you set the visibility (`--visibility` or `-v`) to `all`, `private`, or `selected`. For selected you must specify one or more

repos using `--repos (-r)`:

```
$ gh secret set secret-name --env environment-name
```

```
$ gh secret set secret-name --org org -v private
```

```
$ gh secret set secret-name --org org -v selected -r repo
```

3.8 Workflow permissions

A special secret is the `GITHUB_TOKEN` secret. It is automatically created by GitHub and it can be accessed through the `github.token` or the `secrets.GITHUB_TOKEN`. The token can be accessed by a GitHub action, even if the workflow does not provide it as an input or environment variable. The token can be used to authenticate the workflow when accessing GitHub resources. The default permissions can be set to permissive (read and write) or restricted (read only) – but the permissions can be adjusted in the workflow. You can see the workflow permissions in the workflow log under **Set up job | GITHUB_TOKEN Permissions**. It is best practice to always explicitly set the permissions your workflow needs. All other permissions will be set to none automatically. The permissions can be set for an individual job or the entire workflow.

Here is an example of a workflow that will apply labels to pull requests depending on the files that are changed. The workflow needs read permissions for content to read the configuration and it needs write permissions for pull-requests to apply the label. All other permissions will be non:

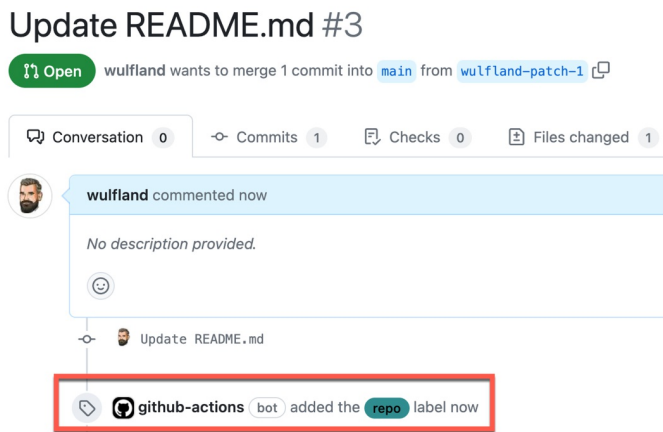
```
on: pull_request_target
```

```
permissions:
  contents: read
  pull-requests: write
```

```
jobs:
  triage:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/labeler@v4
```

Actions performed with the `GITHUB_TOKEN` will be in the history as performed by the *github-actions bot* (see *Figure 3.13*). They will also not trigger new workflow runs to avoid infinite loops by recursive workflow runs.

Figure 3.13 Actions performed with the `GITHUB_TOKEN` will be in the log as performed by the *github-actions bot*.



The default access for the `GITHUB_TOKEN` is restricted. This grants read permission for contents and metadata. You could set the default to read and write, but the recommended way is to have this restricted and grant permissions on workflow or job level. In *Chapter 10, security*, you'll learn more about the security implications of the permissions for the `GITHUB_TOKEN`.

When authoring workflow, you should be aware of the permissions your workflow will need. You should also keep in mind what will happen when the workflow runs from a fork. Private repositories can configure if pull requests from forks are able to run workflows or not. The maximum permissions for the `GITHUB_TOKEN` in workflows triggered from a fork will always be read for all individual permissions.

3.9 Authoring and debugging workflows

The workflow designer is a great help when authoring workflows, as you have experienced in Chapter 2, Hands-on: My first Actions Workflow. Auto-complete, error checking, and the integration of the documentation and the

marketplace in the UI are a great help when writing your workflow.

If you start in a greenfield repository, it is best to just write your workflows in the main branch. However, if you have to create the workflow in a repository that developers are working in, you don't want to get in their way. It is possible to write workflows in a branch and merge them back to the main branch using a pull request - however, some triggers might not work as expected. If you want to run your workflow manually using the `workflow_dispatch` trigger, you first must merge the workflow with the trigger back to main or use the API to trigger the workflow. After that you can author the workflow in a branch and select the branch when triggering the workflow through the UI.

If your workflow needs webhook triggers like `push`, `pull_request`, or `pull_request_target`, it is best to create the workflow in a fork of the repository. This way you can test and debug the workflow without interfering with the developers work, and once you are done you can merge it back to the original repository.

The workflow designer in the web can help a lot when authoring GitHub Actions – but an even better experience is provided by the Visual Studio Code Extension GitHub Actions: <https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-github-actions>.

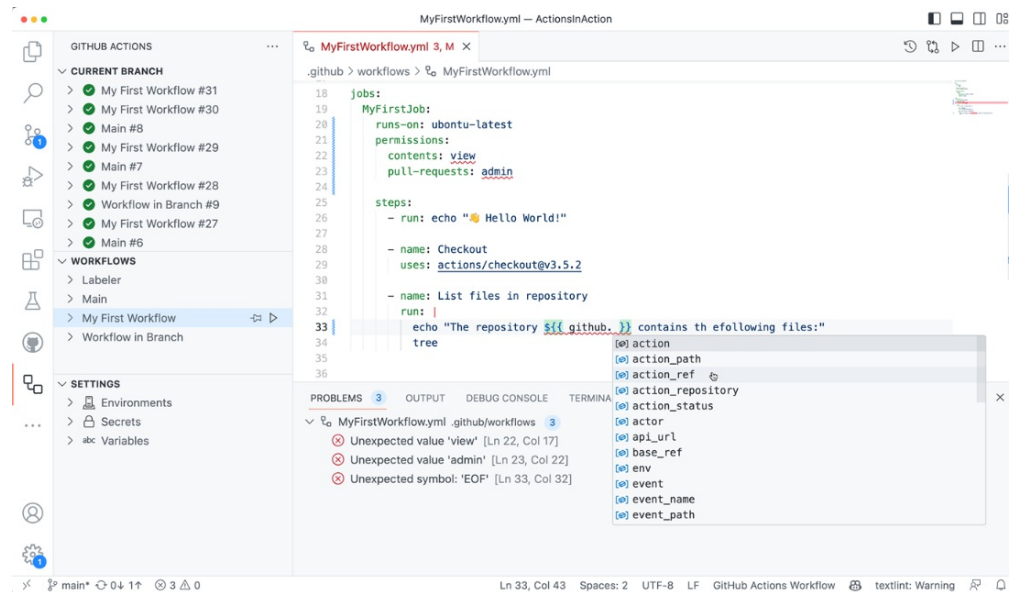
The extension provides the following features:

- Managing workflows and monitoring workflow runs
- Manually triggering workflows
- Syntax highlighting for workflows and expressions
- Integrated documentation
- Validation and code completion
- Smart validation

Especially the smart validation is a great help. It supports code completion for referenced actions and reusable workflows and will parse parameters, inputs, and outputs for referenced actions and provides validation, code completion, and inline documentation. Together with GitHub Copilot this increases quality and speed for authoring workflows tremendously.

Figure 3.14 shows some of the most important features of the extension.

Figure 3.14 The Visual Studio Code Extension for GitHub Actions



There is also a GitHub Action available that can lint all your workflow in your repo: <https://github.com/devops-actions/actionlint>. It can surface a lot of mistakes – for example if you use potentially untrusted inputs in scripts like the `github.head_ref`. The linter can also run on pull requests and annotate you changes in workflow files. You can add the linter as a step to your workflow after checking out the repository.

```
jobs:
  main:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
    steps:
      - uses: actions/checkout@v3
      - uses: devops-actions/actionlint@v0.1.2
```

In general, it is normally the best approach to first run and debug deployment scripts locally or on a virtual machine first and move them to the workflow when you know they will work. But even then, you might experience strange behavior. In this case you can enable debug logging by adding a variable `ACTIONS_STEP_DEBUG` to your repository and set the value to `true`. This will

add a very verbose output to your workflow log and all debug messages from all actions will be displayed. If your issue is related to a runner, you can activate additional logs the same way by setting a variable `ACTIONS_RUNNER_DEBUG` to `true`. In *Chapter 6, Self-hosted Runners*, you will learn more about self-hosted runners and logging. If you want to learn more about debug logging, please refer to: <https://docs.github.com/en/actions/monitoring-and-troubleshooting-workflows/enabling-debug-logging>

3.10 Conclusion

In this chapter you have learned the important things on YAML and the workflow syntax you need to know to start authoring workflows. In the next chapter you will learn how to author and share your own GitHub Actions.

3.11 Summary

- YAML is a text-based data-serialization language optimized to be directly writable and readable by humans and it is a strict superset of JSON with syntactically relevant newlines and indentation instead of braces.
- There are three types of events that can trigger workflows: webhook triggers, scheduled triggers, manual triggers.
- Jobs run in parallel per default if they do not depend on other jobs, whereas steps run in a sequence.
- A workflow step can be a command line that is executed in a shell or a reusable action.
- You can store configuration variables and secrets on organization, repository, or environment level and access them in your workflow.
- The `GITHUB_TOKEN` can be used to authenticate the workflow when accessing GitHub resource and you can set the permissions in a job or workflow.
- You can author your workflows in a branch – but sometimes it's better to create the workflow in a fork to not get in the way with developing the application.

4 GitHub Actions

This chapter covers the following topics:

- Types of actions
- Authoring actions
- Hands-on: my first Docker container action
- Sharing actions
- Advanced action development

Now that we have explored the YAML and workflow syntax in detail, this chapter will dive into the core building block of GitHub Actions – the reusable and sharable actions themselves that give the product its name.

The chapter will cover the different types of actions and some tips to get started writing your first actions. We will cover this in detail in a hands-on lab, where you can follow along step by step. Additionally, the chapter will cover sharing actions in the marketplace and internally, and some advanced topics for action authors.

4.1 Types of Actions

There are three different types of actions:

- *Docker container actions*
- *JavaScript actions*
- *Composite actions*

Docker container Actions only run on Linux whereas JavaScript and composite Actions can be used on any platforms.

All actions have in common, that they are defined by a file `action.yml` (or `action.yaml`) that contains the metadata for the action. This file cannot be named differently, meaning an action must reside in its own repository or folder. The `run` section in the `action.yml` file defines what type of action it

is.

4.1.1 Docker container actions

The docker container actions contain all their dependencies and are therefore very consistent. They allow you to develop your actions in any language – the only restriction is that it has to run on Linux. Docker container actions are slower than JavaScript actions because of the time retrieving or building the image and starting the container.

Docker container actions can reference an image in a container registry like Docker Hub or GitHub Packages. It can also build a Dockerfile at runtime that you provide with the action files. In this case you specify Dockerfile as the image name.

You can pass inputs of the action to the container by either specifying them as arguments to the container or setting them as environment variables.

Listing 4.1 shows an example of an `action.yml` for a container action.

Listing 4.1 An example `action.yml` file for a Docker container action

```
name: 'Your name here'
description: 'Provide a description here'
author: 'Your name or organization here'
inputs:
  input_one:
    description: 'Some info passed to the container'
    required: false
  input_two:
    default: 'some default value'
    description: 'Some info passed to the container'
    required: false
runs:
  #A
  using: 'docker'
  image: 'docker://ghcr.io/wulfland/container-demo:latest'
  #B
  args:
    - ${ inputs.input_one }
    - ${ inputs.input_two }
  #C
  env:
    VARIABLE1: ${ inputs.input_one }
    VARIABLE2: ${ inputs.input_two }
```

Later in this chapter we will provide you with a hands-on lab that gives you the possibility to create your own Docker container action and pass in inputs and process outputs in subsequent steps.

4.1.2 JavaScript actions

JavaScript actions run directly on the runner and are executed in NodeJS. They are faster than Docker container actions and they support all operating systems. Normally, there are two NodeJS versions supported. Older versions will be deprecated at some point. This means you have to maintain your actions and update to newer versions from time to time. That is not necessary for Docker-container-based actions as the container contains all its dependencies.

JavaScript actions support TypeScript – as TypeScript compiles to normal JavaScript code. That's why it is best practice to develop your actions in TypeScript to have static typing, enhanced tooling, better readability and maintainability, and earlier error detection. Keep in mind that the action must contain all dependencies in the repository. This means you have to commit the `node_modules` folder and all transpiled TypeScript code.

In JavaScript as well as TypeScript actions you can use the toolkit (<https://github.com/actions/toolkit>) to easily access input variables, write to the workflow log, or set output variables.

If you want to start writing JavaScript actions in TypeScript you can use the following template: <https://github.com/actions/typescript-action>. It will get you started quickly.

Listing 4.2 shows an example for a TypeScript action running on NodeJS 16.

Listing 4.2 An example for a TypeScript action.yml file

```
name: 'Your name here'
description: 'Provide a description here'
author: 'Your name or organization here'
inputs:
  input_one:
    required: true
```

```
    description: 'input description here'
    default: 'default value if applicable'
runs:
  using: 'node16'
  main: 'dist/index.js'
```

4.1.3 Composite actions

The third type of actions are the composite actions. They are nothing more than a wrapper for other steps or actions. You can use them to bundle together multiple run commands and actions, or to provide default values for other actions to the users in your organization.

Composite actions just have steps in the runs section of the `action.yml` file – like you would have in a normal workflow. You can access input arguments using the `inputs` context. Output parameters can be accessed using the `outputs` of the step in the `steps` context. *Listing 4.3* shows an example of a composite action and how you can process inputs and outputs.

Listing 4.3 An example for a composite action

```
name: 'Hello World'
description: 'Greet someone'
inputs:
  who-to-greet:
    description: 'Who to greet'
    required: true
    default: 'World'
outputs:
  random-number:
    description: "Random number"
    value: ${{ steps.random-number-generator.outputs.random-id }}
runs:
  using: "composite"
  steps:
    - run: echo "Hello ${{ inputs.who-to-greet }}."
      shell: bash

    - id: random-number-generator
      run: echo "random-id=$(echo $RANDOM)" >> $GITHUB_OUTPUT
      shell: bash

    - run: echo "Goodbye $YOU"
```

```
shell: bash
env:
  YOU: ${{ inputs.who-to-greet }}
```

Note that if you use `run:` in composite actions, the `shell` parameter is required. In normal workflows it is optional. Keep in mind that your action might run on different operating systems. Bash is a shell that is most likely available on all of them.

4.2 Authoring Actions

If you want to start authoring actions on your own, you first must decide what kind of action you want to use. If you already know NodeJS and TypeScript then this is probably your natural choice. If not, you have to balance the effort of learning a new language and ecosystem with the fact that you have the toolkit in JavaScript actions and that Docker container actions are slower to start up.

Composite actions can be used to wrap recurring scenarios together. This is very useful in an enterprise context, but there are also some actions in the marketplace that do this. If you write bash scripts, composite actions are also a simple solution you might consider.

If you already have a solution that runs in a container, then it is probably very easy to port it to GitHub Actions.

4.2.1 Getting started

Independent of the type of action you want to write – the best thing is to get started with a template. You can find templates for all kind of actions under <https://github.com/actions/>:

- JavaScript: <https://github.com/actions/javascript-action>
- TypeScript: <https://github.com/actions/typescript-action>
- Docker container: <https://github.com/actions/hello-world-docker-action>
- An example for a composite action: <https://github.com/actions/upload-pages-artifact>

The composite action is just an example – the others are template repositories, and you can generate a new repository directly from the template and modify the files there.

Depending on your technical background, you might have a different choice for tools and approaches. If you are familiar with GitHub Actions and REST but not with TypeScript, you might want to try out a solution first in a workflow using the *actions/github-script* action. This action is pre-authenticated and has a reference to the toolkit. This action allows you to validate fast if your solutions works and you can later move the code to the TypeScript action template.

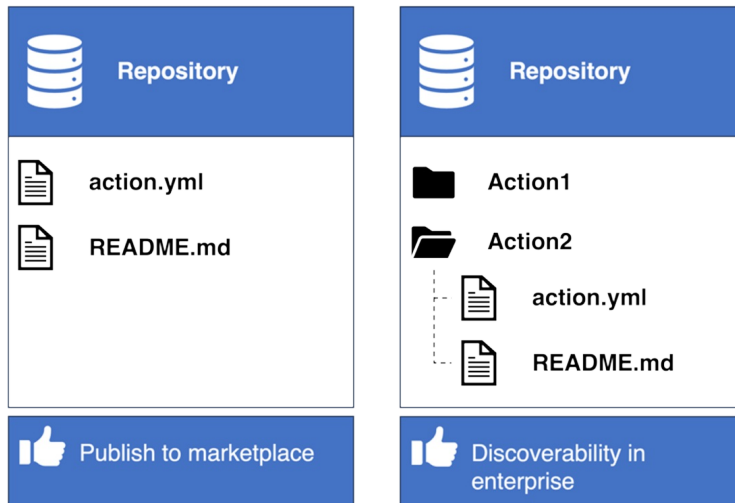
Make sure to pick a toolset and approach that fits your background and that allows you to get fast feedback and iterate in short cycles on your solution.

4.2.2 Storing actions in GitHub

Actions are files located in GitHub. GitHub uses the `action.yml` file to discover actions. Since you cannot change that name, this means your actions must either reside in their own repository or in a folder. Storing them in folders allows you to have multiple actions in one repository. This can be better for easy discoverability in an enterprise context if you just want to publish a few composite actions. It's also a valid solution if some actions belong together and share the same dependencies and versioning.

The downside is that you cannot publish these actions in the marketplace. If you want to publish your actions to the marketplace you must store them in their own, public repository and the `action.yml` must be in the root of the repository. The other downside is, that you have to version all actions together if they reside in the same repository. *Figure 4.1* shows a comparison of storing actions in a repo or in folders.

Figure 4.1 Actions can be stored in a repository or a folder



The recommended way is storing each action in its own repository and have its own lifecycle. In an enterprise context you can store all your actions in a separate organization. This helps with the discoverability and management.

4.2.3 Compatibility with GitHub Enterprise Server

When writing actions – especially if you plan to share them publicly – try to keep them compatible with GitHub Enterprise Server. There are still many customers that run GitHub on premises. This means that you cannot hardcode any URLs to GitHub APIs. For the GitHub REST API you can use the `GITHUB_API_URL` environment variable and for the GitHub GraphQL API you can use the `GITHUB_GRAPHQL_URL` environment variable. This way you don't have to hardcode the URL and stay compatible with GitHub Enterprise server deployments.

4.2.4 Release management

It is important to have a proper release management for your action in place. Best practice is to use tags together with GitHub releases (see <https://docs.github.com/en/repositories/releasing-projects-on-github>) together with semantic versioning. Using GitHub release is required if you want to publish your action to the marketplace.

Since you will learn more about semantic versioning and how you can automate release management for GitHub Actions in *Chapter 8, Continuous*

Integration (CI), we will not cover this here in depth. But when you are starting to author actions. You should make sure to include the following from the beginning:

- Create a tag with a semantic version for every version of the action that you want to publish.
- Mark the version latest if you publish the action to the marketplace.
- Create a CI build that tests your action before releasing it.
- Make sure to add additional tags for major version and update these tags if you provide a security or bug fix. For example, if you have a version v3.0.0 – also provide a version v3 and update v3 to a new commit in case you release a version v3.0.1 with an important fix.

In the following hands-on lab, you will create a basic docker container action with a workflow that will test the action on any change to one of the files.

4.3 Hands-on: a docker container action in action

In this hands-on lab you will create a docker container action that uses input and output parameters. Furthermore, you will create a CI build that tests the action every time a change is made to one of the files.

Hands-on:

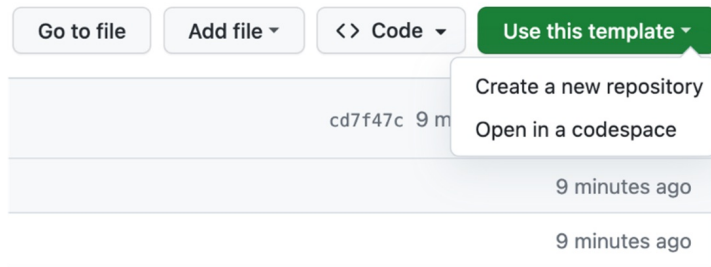
The following instruction can also be followed on:

<https://github.com/GitHubActionsInAction/ActionInAction>. This allows you to also copy and paste the values to the files without having to type them.

4.3.1 Use the template to create a new repository

In the repository <https://github.com/GitHubActionsInAction/ActionInAction>, click on **Use this template** and select **Create new repository** (see *Figure 4.2*).

Figure 4.2 Create a new repository from the template



Pick your GitHub username as the owner and enter `MyActionInAction` as the repository name. Make the repository public and click **Create repository from template** (see *Figure 4.3*)

Figure 4.3 Create a public repo for the action

Create a new repository from ActionInAction

The new repository will start with the same files and folders as [GitHubActionsInAction/ActionInAction](#).

Owner *



GitHubActionsInAction ▾

Repository name *

/ MyActionInAction ✓

Great repository names are short and memorable. Need inspiration? How about [sturdy-octo-robot?](#)

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.



Include all branches

Copy all branches from GitHubActionsInAction/ActionInAction and not just main.



You are creating a public repository in the GitHubActionsInAction organization.

Create repository from template

4.3.2 Create the Dockerfile for the action

The action will use a docker container to execute a script. We will create this docker container using a *Dockerfile*. Create a new file called `Dockerfile` and add the following content:

```
# Container image that runs your code
```

```
FROM alpine:latest

# Copies entrypoint.sh from your repo to the path '/' of the cont
COPY entrypoint.sh /entrypoint.sh

# Make the script executable
RUN chmod +x entrypoint.sh

# Executes '/entrypoint.sh' when the docker container starts up
ENTRYPOINT ["/entrypoint.sh"]
```

Commit the file to the main branch.

The Dockerfile defines the docker container for the action. It uses the latest alpine image and copies a local script – that yet has to be created – to the container and marks it executable (`chmod +x`). The container will then execute the script.

You could also use an existing image, but we want to build everything from scratch so that we know what the container exactly does.

4.3.3 Create the action.yml file

GitHub identifies actions by looking for an `action.yml` manifest that defines the action. Create a new file called `action.yml`. Add the content from *Listing 4.4* to the file and replace the placeholder `{GitHub username}` with your GitHub username.

Listing 4.4 Writing the action.yml file that defines the action

```
name: "{GitHub username}'s Action in Action"
description: 'Greets someone and returns always 42.'
inputs:
  who-to-greet: # id of input
    description: 'Who to greet'
    required: true
    default: 'World'
outputs:
  answer: # id of output
    description: 'The answer to everything (always 42)'
runs:
  using: 'docker'
```

```
image: 'Dockerfile'
args:
  - ${{ inputs.who-to-greet }}
```

Commit the file to the main branch.

This action file defines the action and the input and output parameters. The runs section is the part that defines the action type – in this case we use docker together with a Dockerfile instead of an image. We pass the input to the container as an argument (args).

4.3.4 Create the entrypoint.sh script

The script that is executed in the container is called `entrypoint.sh` in our Dockerfile. Create the file and add the following content:

```
#!/bin/sh -l

echo "Hello $1"
echo "answer=42" >> $GITHUB_OUTPUT
```

This simple script writes `Hello` and the input `who-to-greet`, that was passed in as the first argument (`$1`) to the container, to the standard output. It also sets the output parameter to `42`.

Commit the file to the main branch.

4.3.5 Create a workflow to test the container

The action is now ready to be used. To see it in action, we'll create a workflow that uses it locally. Create a new file called `.github/workflows/test-action.yml` and add the content from *Listing 4.5*.

Listing 4.5 Testing an action in a local workflow

```
name: Test Action
on: [push]

jobs:
  test:
```

```

runs-on: ubuntu-latest
steps:
  - name: Checkout repo to use the action locally
    uses: actions/checkout@v3.5.3

  - name: Run my own container action
    id: my-action
    uses: ./
    with:
      who-to-greet: '@wulfland'

  - name: Output the answer
    run: echo "The answer is ${ steps.my-action.outputs.answ

  - name: Test the container
    if: ${ steps.my-action.outputs.answer != 42 }
    run: |
      echo "::error file=entrypoint.sh,line=4,title=Error in
      exit 1

```

In this workflow we use the local version of the action (`uses: ./`). In this case it is required to check out the repository first using the checkout action. This is not necessary if you reference an action by a git reference (`action-owner/action-name@reference`). To access the output parameters, you have to set the `id` property of the step. The outputs can then be accessed using the step context (`step.name-of-step.outputs.name-of-output`).

The workflow will automatically run because of the push trigger after committing the file. Inspect the output – how the container is created, how it writes the greeting to the workflow log, and how the output is passed to the next step (see *Figure 4.4*).

Figure 4.4 Output of the action in the test workflow

```
Run my own container action

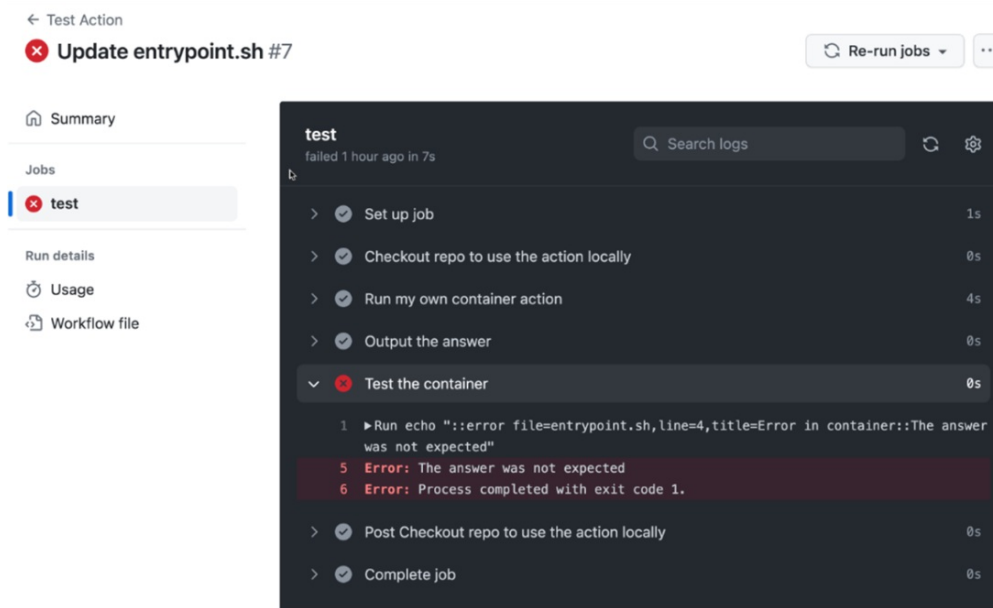
1 ▶ Run ./
4 ▶ Building docker image
26 /usr/bin/docker run --name a6c8cf0162cc8c9b95700c7a26e69b717c010_
-e "GITHUB_REPOSITORY_OWNER_ID" -e "GITHUB_RUN_ID" -e "GITHUB_RUN
GITHUB_HEAD_REF" -e "GITHUB_BASE_REF" -e "GITHUB_EVENT_NAME" -e
"GITHUB_WORKSPACE" -e "GITHUB_ACTION" -e "GITHUB_EVENT_PATH" -e "
"RUNNER_NAME" -e "RUNNER_ENVIRONMENT" -e "RUNNER_TOOL_CACHE" -e "
"/var/run/docker.sock":"/var/run/docker.sock" -v "/home/runner/wo
"/home/runner/work/ActionInAction/ActionInAction":"/github/worksp
27 Hello @wulfland

Output the answer

1 ▶ Run echo "The answer is 42"
4 The answer is 42
```

The last step of the workflow will only run if the output does not have the expected value. The step will write an error message to the log and fail the workflow by returning a non-zero return value using `exit`. To test this, just set the value in `entrypoint.sh` to another value and commit the changes. The workflow will be triggered and fail with a message like displayed in Figure 4.5.

Figure 4.5 Fail the workflow if the action returns the wrong value



Make sure to reset the value again in case you also want to try out sharing the action to the marketplace.

4.4 Sharing Actions

Actions are the core building blocks for workflows, and they are built in way that is it easy to reuse and share them. You can share actions internally in your organization from within private repositories or publicly in the GitHub marketplace.

4.4.1 Sharing actions in your organization

You can grant GitHub actions access to private repositories in your organization. Per default, workflows cannot access other repositories. But by granting permissions for GitHub Actions it is easy to share actions and *reusable workflows* within your organization.

Reusable Workflows

Reusable workflows are building blocks like actions and can also be shared in your organization – but not in the marketplace. Reusable workflows use the `on: [workflow_call]` trigger that you can also use to define inputs and outputs. Reusable workflow can contain multiple jobs that are executed on different runners. The calling workflow will use the keyword `uses` instead of `runs-on` on a job level in the same way you use it for actions on the step level (path in git plus a reference – or a local path if your repository is checked out. Please refer to the documentation: <https://docs.github.com/en/actions/using-workflows/reusing-workflows>

You will learn more about reusable workflows in action in *Chapter 9, Improving workflows performance and costs*.

Compared with composite actions, reusable workflows give you control over multiple jobs and environments that can run on different runners and have interdependencies. Composite actions are always executed in one job and only give you control over the steps inside the job.

To grant access to GitHub actions and reusable workflows in a repo, you can go to *Settings | Actions* in the repository. In the section *Access* you can grant access to repositories in your organization or enterprise (see *Figure 4.6*).

Figure 4.6 Allowing access to actions and reusable workflows in private repositories

Access

Control how this repository is used by GitHub Actions workflows in other repositories. [Learn more about allowing other repositories to access to Actions components in this repository.](#)

☐ **Not accessible**

Workflows in other repositories cannot access this repository.

☒ **Accessible from repositories in the 'GitHubActionsInAction' organization**

Workflows in other repositories that are part of the 'GitHubActionsInAction' organization can access the actions and reusable workflows in this repository. Access is allowed only from private repositories.

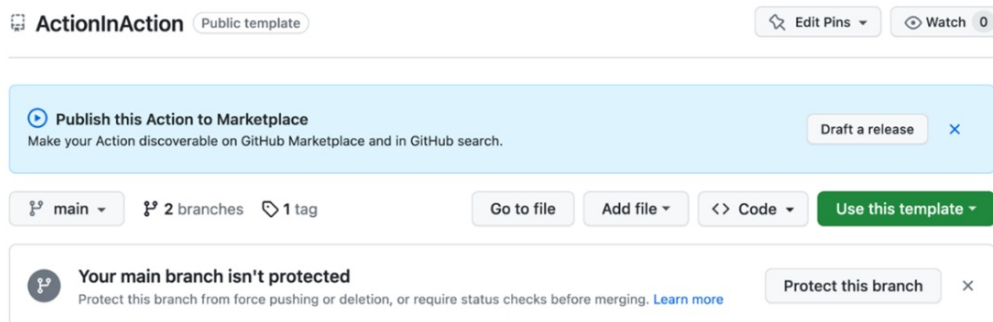
Save

This has to be configured for each repository, that contains actions or reusable workflows.

4.4.2 Sharing actions publicly

GitHub will automatically detect if you have an `action.yml` file in your repository and propose to draft a release to publish it to the marketplace (see *Figure 4.7*).

Figure 4.7 GitHub automatically detects if you have an `action.yml` file in the root of the repository



When creating a release, you will find a new section Release Action in the dialog. You must accept the *GitHub Marketplace Developer Agreement* before being able to publish a release to the marketplace (see *Figure 4.8*).

Figure 4.8 You must accept the GitHub Marketplace Developer Agreement before being able to publish a release

Release Action

☐ Publish this release to the GitHub Marketplace 
 GitHubActionsInAction must [accept the GitHub Marketplace Developer Agreement](#) before publishing an Action.

Once you have accepted the agreement you can select the checkbox. GitHub will then check your action and give you guidance on important properties for your action:



- **Name:** the name must be unique.
- **Description:** The action should have a description what it does.
- **Branding:** The action should have an icon and a color. GitHub will give you a list of available colors and icons.
- **Readme:** The action should contain a README.md file.


The check looks like in *Figure 4.9* if you try it with the action that you have created in the hands-on lab earlier.

Figure 4.9 GitHub will check the properties of your action



Release Action


☒ Publish this Action to the GitHub Marketplace
 Your Action will be discoverable in the Marketplace and available in GitHub search.

 action.yml
 

 Improve your Action by adding labels for icon and color.

✓ Name	@wulfland's Action in Action
✓ Description	Greets someone and returns always 42.
⚠ Icon	See list of available icons.
⚠ Color	See list of available colors.

 README
 

 A README exists.

To add an icon and color, pick one from each list and add them to the action.yml file like this:


```
branding:
  icon: 'alert-triangle'
  color: 'orange'
```

A list of the current available icons and colors can be found on <https://docs.github.com/en/actions/creating-actions/metadata-syntax-for-github-actions#branding>.

You can now draft a release by picking a tag or creating a new one. Pick one or two categories for the marketplace that will define where the action will be listed.

Note the feature to automatically create release notes for your release. It will pick up your pull requests and first-time contributors and automatically create good release notes as shown below in *Figure 4.10*.

Figure 4.10 Creating a release with release notes that will be published to the marketplace

The screenshot shows the GitHub Actions release creation interface. At the top, there are two dropdown menus for 'Primary Category' (set to 'Learning') and 'Another Category' (set to 'Choose an option'). Below these are two buttons: 'v1.0.0' and 'Target: main'. A message states: 'Excellent! This tag will be created from the target when you publish this release.' Below this is a text input field containing 'v1.0.0'. There are two tabs: 'Write' (active) and 'Preview'. The 'Write' tab shows a rich text editor with a toolbar containing icons for bold, italic, link, list, quote, code, mention, link, and undo. A 'Generate release notes' button is on the right. The text area contains the following content:
What's Changed
* Solution by @wulfland in <https://github.com/GitHubActionsInAction/ActionInAction/pull/1>

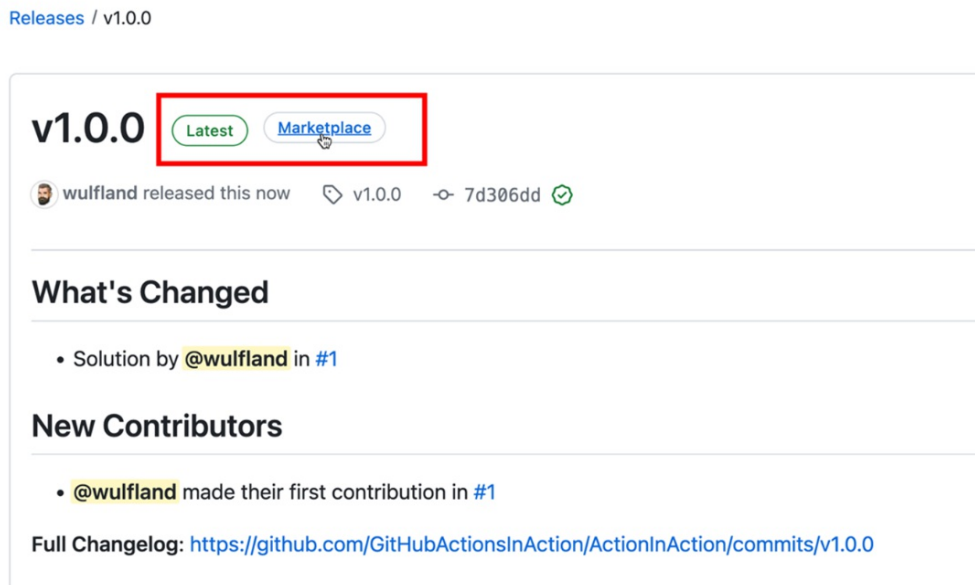
New Contributors
* @wulfland made their first contribution in <https://github.com/GitHubActionsInAction/ActionInAction/pull/1>

Full Changelog: <https://github.com/GitHubActionsInAction/ActionInAction/commits/v1.0.0>
Below the text area is a file upload section with a message: 'Attach files by dragging & dropping, selecting or pasting them.' and a button with a plus icon. Below that is another section with a message: 'Attach binaries by dropping them here or selecting them.' and a downward arrow icon. At the bottom, there is a checkbox labeled 'Set as a pre-release' with the text 'This release will be labeled as non-production ready' below it. At the very bottom are two buttons: 'Publish release' (green) and 'Save draft' (grey).

The result will look like in Figure 4.11. In the screenshot you can see that the


release contains a link to the marketplace. It also contains a label indicating that the release is the latest release. This makes it the default in the marketplace listing.

Figure 4.11 A release that is listed in the marketplace



If you follow the link, it will take you to the listing that looks like *Figure 4.12*. You will see the README.md, the versions, contributors, and links to your repository. This page is also the place where you can delist your action from the marketplace if you want to stop sharing it.

Figure 4.12 The marketplace listing of the GitHub action



GitHub Action

@wulfland's Action in Action

v1.0.0 Latest version

Use latest version

Choose a version

v1.0.0

v1.0.0

Star 0

Contributors

Categories

Learning

Links

GitHubActionsInAction/ActionInAction

Open issues 0

Pull requests 0

Report abuse

@wulfland's Action in Action is not certified by GitHub. It is provided by a third-party and is governed by separate terms of service, privacy policy, and support documentation.

Hands-on: a docker container action in action

In this hands-on lab you will create a docker container action that uses input and output parameters. Furthermore, you will create a CI build that tests the action every time y change is made to one of the files.

The lab consists of the following parts:

1. Use the template to create a new repo
2. Create the dockerfile for the action
3. Create the action.yml file
4. Create the entrypoint.sh file
5. Create a workflow to test the action

Use the template to create a new repo

In this repository, under [Code](#), click on 'Use this template' and select [Create new repository](#).

Go to file

Add file -

<> Code

Use this template -

Create a new repository

Open in a codespace

Once the action is published to the marketplace, you can also find it from within the workflow editor (see *Figure 4.13*).

Figure 4.13 The action will be discoverable in the workflow editor

ActionInAction / .github / workflows / test-mp.yml in main

Cancel changes Commit changes...

Edit Preview Spaces 2 No wrap


```

1 name: Test Action in Marketplace
2 on: [workflow_dispatch]
3
4 jobs:
5   test:
6     runs-on: ubuntu-latest
7     steps:
8
9       - name: Run my own container action
10         id: action
11         uses: GitHubActionsInAction/ActionInAction@v1.2.1
12         with:
13           who-to-greet: '@wulfland'
14
15       - name: Output the answer
16         run: echo "The answer is ${ steps.action.outputs.answer }"
17

```

Marketplace Documentation

Marketplace / Search results /
@wulfland's Action in Ac...



@wulfland's Action in Action

v1.2.1 0

Greets someone and returns always 42

View full Marketplace listing

Installation

Copy and paste the following snippet into your .yaml file.

Version: v1.2.1

```

- name: @wulfland's Action in Action
  # You may pin to the exact commit or
  # uses: GitHubActionsInAction/ActionIn
uses: GitHubActionsInAction/ActionInA
with:
  # Who to greet
  who-to-greet: # default is World

```

If you want to try this out, you can modify your workflow – or create a new one in another repository – and pick the version from the marketplace like

you would use any other action.

```
name: Test Action in Marketplace
on: [workflow_dispatch]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:

      - name: Run my own container action
        id: action
        uses: GitHubActionsInAction/ActionInAction@v1.2.1
        with:
          who-to-greet: '@wulfland'

      - name: Output the answer
        run: echo "The answer is ${ steps.action.outputs.answer }
```

But make sure to delist your action again from the marketplace to not clutter the marketplace unnecessary with actions that you don't have any intent to maintain them. In the marketplace offering you will find a **Delist** button in the right top corner to do so.

4.5 Advanced Action development

If you want to build actions, you will probably need to interact with GitHub. GitHub has two different APIs that you can use:

- **REST API:** Use the REST API to create integrations, retrieve data, and automate your workflows. The rest API is easy to use because you send a simple request and get a response. And yet it is very powerful and you can automate with it everything. See <https://docs.github.com/en/rest> for the complete documentation.
- **GraphQL API:** The GitHub GraphQL API offers more precise and flexible queries than the GitHub REST API. It is better suited for complex scenarios where you have to control the flow of data and the amount of data being transmitted. One example would be paging of big lists. It is more complicated because you have to specify in your request the data and fields that should be included in the result. See

<https://docs.github.com/en/graphql> for the complete documentation.

There is an SDK available called *Octokit* (see <https://github.com/octokit>). The SDK is supported by GitHub, and it is available for the following languages:

- JavaScript and TypeScript
- C# .NET
- Ruby
- Terraform

But there are many third-party libraries available. For Java, Erlang, Haskell, python, Rust, and many more. You can find a complete list under <https://docs.github.com/en/rest/overview/libraries>.

The SDKs are a good starting point to learn how to authenticate using the GitHub token and performing actions in GitHub from within your code.

4.6 Best practices

When authoring actions that you want to share – publicly or within your organization – there are some best practices that you should follow:

- **Small and focused:** Keep the action small and focused and adhere to the *Single Responsibility Principle*. An action should do one thing well and not many things mediocrely. To avoid this problem, try not to create “Swiss army knives” that have many inputs and can do a lot of different things.
- **Write tests and a test workflow:** Make sure to have sufficient tests for your code - and a test workflow that runs the action as an action. Good tests will give you the confidence to release frequently.
- **Semantic versioning:** Use semantic versioning with your releases to indicate what has changed. Use multiple tags and update the major versions with patches if you fix a bug. For example: if you release a version v3.0.0 – also add a tag v3 for the current major version. If you provide a bugfix (v3.0.1), move the tag v3 to the fixed version.
- **Documentation:** Make sure you have good documentation and a proper

README.md that helps the users of your action to understand what it does and how it is supposed to be used. Give one or more concrete examples of how the actions is supposed to be used. Also provide documentation on how people can contribute changes.

- **Proper action.yml metadata:** Provide good metadata in your action.yml and especially for your inputs and outputs. Try to avoid required inputs and provide default values whenever possible. This will make it much easier to consume your action.
- **SDKs:** Use the toolkit (github.com/actions/toolkit) or the other SDKs to interact with GitHub and the APIs.
- **Publish the Action:** Last but not least – publish the action to the marketplace to make it discoverable and to potentially get other people to contribute to it or provide you with feedback.

4.7 Conclusion

In this chapter you have learned what actions are and some tips and best practices if you want to start writing and sharing actions.

This is the end of part 1 and you have now a good understanding of GitHub Actions workflows, the workflow syntax, and writing GitHub Actions. In part 2 we will now dive deep into how runners execute your workflows and the security implications of this before we cover the more practical part of using actions for CI/CD in part 3.

4.8 Summary

- There are three types of GitHub Actions: Docker container actions, JavaScript actions, and Composite actions
- Docker container actions only run on Linux and not on Windows or macOS
- Docker container actions can retrieve an image from a Docker library like Docker Hub or build a Dockerfile
- JavaScript actions run directly on the runner using NodeJS and are faster than container actions
- Composite actions are a wrapper for other steps or actions

- You publish actions to the marketplace by placing them in their own repository and publishing a GitHub release
- You can share actions internally by granting access to workflows in your organization in a private repository
- You can use the *octokit* SDK to interact with the GitHub APIs in your actions

5 Runners

This chapter covers

- Getting to know GitHub runners
- What does the runner service do
- Using GitHub hosted runners
- Analyzing utilization of GitHub hosted runners
- When to use self-hosted runners

The runtime of GitHub Actions is provided by a service that is called *Runners*. Runners are standalone instances that continuously ask GitHub if there is work for them to execute. They provide the runtime for your job definitions: they will execute the steps defined in the job for you and provide information about the outcome back to GitHub, as well as the logs and any data uploaded to GitHub, for example artifacts and cache information.

In this chapter we will focus on the runners that GitHub hosts for you as a service. These are called **GitHub hosted runners** and come with certain compute power, preinstalled software, and are maintained with the latest security- and Operating System (OS) updates. Since GitHub does all the maintenance for you, there is a cost attached to using these runners. Depending on your plan, you will have a certain amount of action minutes included for free. See **Paragraph 5.4, GitHub hosted runners**, for more information.

5.1 Targeting a runner

Job definitions have to specify a set of labels they want to use for the GitHub service to find a match when a job is queued to be executed. See **Listing 5.1, An example of targeting multiple labels to run the job**. A job has to target at least one runner label and can target multiple labels if needed. The GitHub hosted runners have several default labels available that indicate for example the operating system of the runner.

Listing 5.1 An example of targeting multiple labels to run the job

```
jobs:
  example-job:
    runs-on: [ubuntu-latest]
    steps:
      run: echo 'Job is running on ${ runner.os }'
```

GitHub will use the list of labels to find a runner that is online and that is ready to handle jobs. For a job to find a runner, all labels in the `runs-on` array need to match.

You can also install the runner yourself, in your own environment, we call those ‘self-hosted runners’. Since you define where the service is being hosted (local machine, cloud, etc.), you are already paying for that compute. GitHub is not charging you for self-hosted runners or for parallel job executions. With self-hosted runners you can add extra labels associated with the runner as well. There is always one extra label that is added to the self-hosted runner so that users can differentiate it from the GitHub runners. The value of that label is `self-hosted`. This is available next to the label that indicates the OS and the bitness of the environment. You can find more information about self-hosted runners in **Chapter 6: Self-hosted runners**.

5.2 Queuing jobs

A job can be queued in many different ways. See **Chapter 3** for ways to trigger a job to be queued. When the event is triggered, GitHub will start queueing the relevant jobs from the workflow and will start searching for an available runner that has the correct labels (and is available for your repository). For GitHub hosted runners, the queuing of the job will fail if there are no runners available with the requested label(s) within 45 minutes. For self-hosted runners the job will stay queued until a matching runner comes online. The maximum duration of being queued is 24 hours. If there is no runner available within this period, the job will be terminated. The most common reason the workflow does not start is because the runner label does not exist or is not available for the current repository. It could be for example that the label for a self-hosted runner is used, which does not exist on GitHub hosted runners.

5.3 The runner application

The runner application is based on .NET core and can be installed on a virtual machine, a container, or any other environment that can run .NET core code. That means it can be installed on Linux, Windows, MacOS operating systems, as well as on X86, X64 and ARM processors. This allows you the flexibility of hosting it where it makes sense to you: whether it is on a full-fledged server (physical or virtual) or on a containerized environment. You can run it in an AWS Lambda, an Azure Function, or in Kubernetes. The application itself can be installed as a service and has configuration options to start when the environment starts, to only run on demand, or to run ephemeral. Configuring a runner as ephemeral means that the runner will only handle one single job, after which it will stop asking for more work. That gives you the opportunity to clean up after each run, or to completely destroy the environment and start up new environments as needed.

The source code of the runner is open source, so you can see how it works and can even contribute issues and pull requests to make the service even better. The release notes of the runner contain important information about upcoming changes, like we have seen for example with the planned deprecation of ‘set-output’ and ‘save-state’: actions and scripts that used these calls, got warnings in the months before the actual deprecation. You can look at the source code and follow along with the updates from here: <https://github.com/actions/runner>.

The runner service will execute job definition and handles things like:

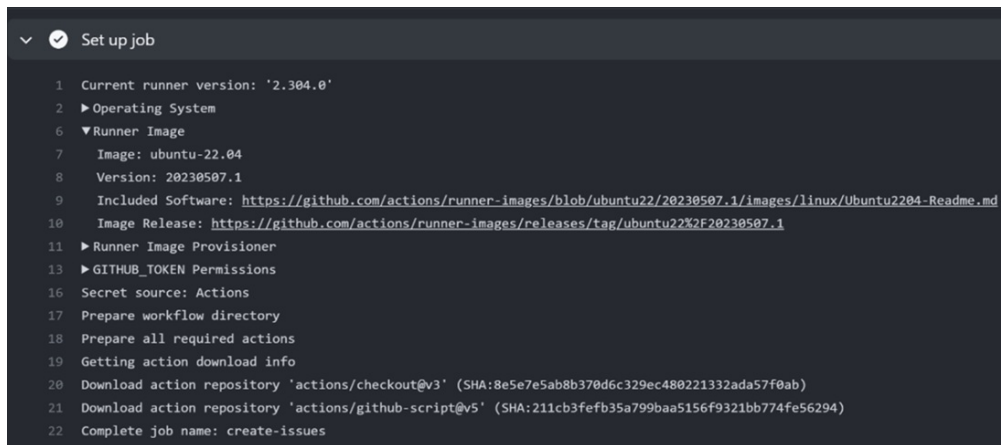
- Downloading action repositories.
- Writing the logs back to GitHub for later retrieval.
- Up- and downloading artifacts to and from GitHub.
- Reading and writing to the cache service provided by GitHub.

5.4 GitHub hosted runners

GitHub hosts runners for their users to enable them quickly started using GitHub Actions. That means that GitHub hosts the environments that execute the runner service, makes sure the OS is secured, continuously updated, as

well as installing all security updates. Any tool that they provide on the environment also needs to be updated with the latest versions and security fixes. What is installed on the environment can be found in this public repository: <https://github.com/actions/runner-images>. You can find for each job execution what version of the environment was used by checking the execution logs. See Figure 5.1, Set up job step with information about the environment.

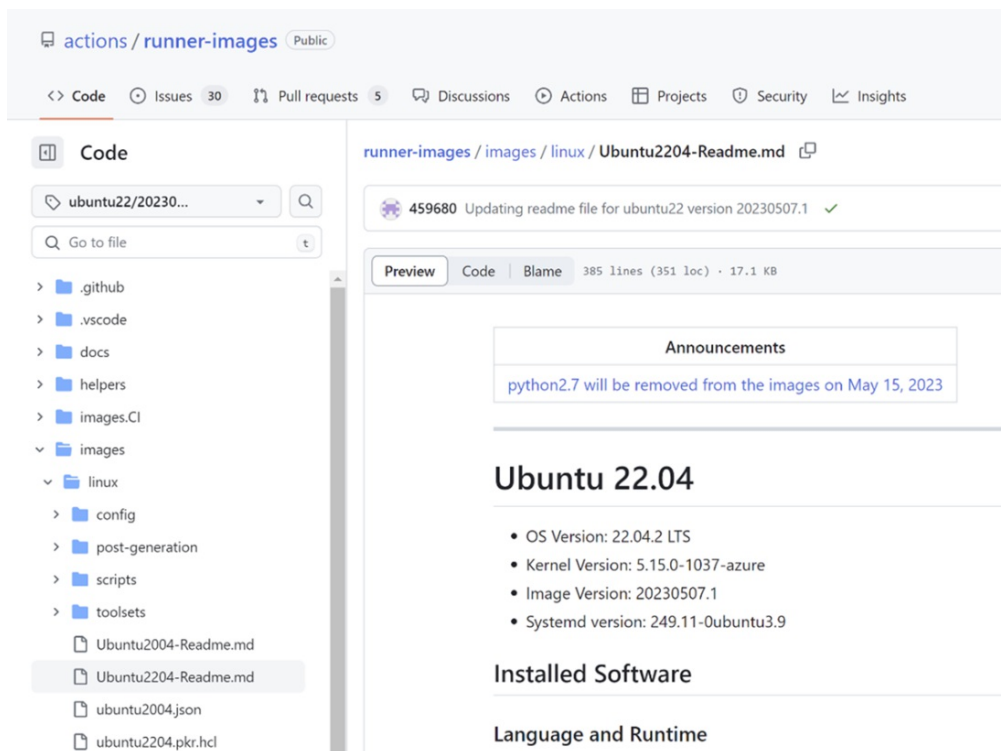
Figure 5.1 Set up job step with information about the environment



```
1 Current runner version: '2.304.0'
2 ▶ Operating System
6 ▼ Runner Image
7 Image: ubuntu-22.04
8 Version: 20230507.1
9 Included Software: https://github.com/actions/runner-images/blob/ubuntu22/20230507.1/images/linux/Ubuntu2204-Readme.md
10 Image Release: https://github.com/actions/runner-images/releases/tag/ubuntu22%2F20230507.1
11 ▶ Runner Image Provisioner
13 ▶ GITHUB_TOKEN Permissions
16 Secret source: Actions
17 Prepare workflow directory
18 Prepare all required actions
19 Getting action download info
20 Download action repository 'actions/checkout@v3' (SHA:8e5e7e5ab8b370d6c329ec480221332ada57f0ab)
21 Download action repository 'actions/github-script@v5' (SHA:211cb3fefb35a799baa5156f9321bb774fe56294)
22 Complete job name: create-issues
```

In the <https://github.com/actions/runner-images> repository you find the list of installed software, the versions that were used during installation as well as any information about deprecated versions of software on the environment. An example of the information from the used environment can be found in **Figure 5.2: Information about the runner image.**

Figure 5.2 Information about the runner image



The images are updated on a weekly basis, or more often when needed. The version is linked to the date (in ISO format) of the Monday of the week the image was created, and starts with version 0, for example '20230417.0'. If there are any extra updates needed during the week (normally only to fix broken software deployments or security updates), they update the version number but not the date, for example 20230417.1, 20230417.2, and so on. New versions are gradually rolled out based on the US time zone in California, as most of the engineering teams responsible are located in that time zone. In case any deployment issues arise, they can quickly mitigate the problem. For example by stopping the rollout, by reverting back to a previous version, or rolling out a fix.

5.5 Hosted operating systems

GitHub hosts three different operating systems for you to choose from:

- Linux based (Ubuntu)
- Windows base
- macOS based

For each operating systems GitHub usually hosts the two or three most recent versions, which can be targeted with the label for that specific version. See **Table 5.1: Overview of supported runner Operating Systems**. You can always find the latest version in the documentation here:

<https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners#supported-runners-and-hardware-resources>

Table 5.1 Overview of supported runner Operating Systems

Operating System	Version label available
Ubuntu	ubuntu-20.04
	ubuntu-22.04
Windows	windows-2019
	windows-2022
macOS	macos-12
	macos-11

Next to the version labels, there is always a ‘latest’ version of each operating system available:

- ubuntu-latest
- windows-latest
- macos-latest

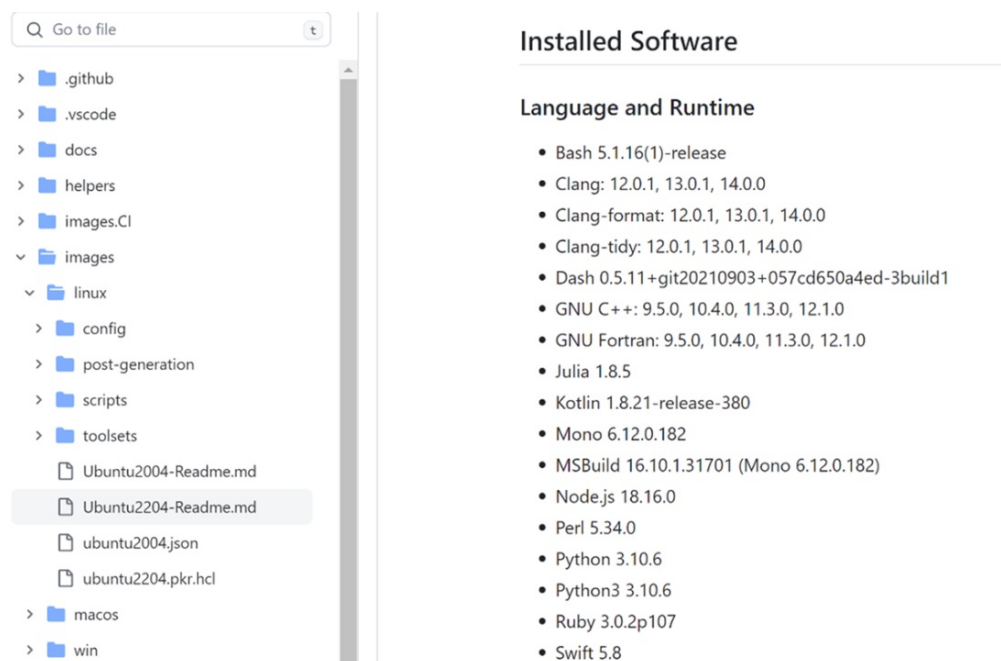
These labels are there for your convenience. It is up to GitHub to decide what version that ‘latest’ means at any given time. Any changes to the meaning of ‘latest’ is communicated up front through the runner-images repository, as well as deprecation warning messages in the action logs. In the past we have seen changes to the latest version being communicated up to six months before they started to mean the new version. Right before the new version becomes latest, GitHub also flips the meaning of ‘latest’ for a percentage of the runners, and carefully checks their telemetry for any big spikes in errors coming from the change.

5.6 Installed software

A lot of software comes preinstalled with the GitHub hosted runners. It

includes the operating systems' build-in tools and shells. For example, Ubuntu and macOS runners include ``grep``, ``find``, and ``which``, among other default tools. The software list is available in the runner-images repositories, and is dependent on the operating system itself, as not everything is available for Linux, Windows as well as MacOS. GitHub works together with the user community to define what software will get installed on the environment. They focus on the most used SDK's, shells, package ecosystems, etc. If you have the need for software that is missing, you can create an issue in the runner-images repository and propose it for adoption. Since it is GitHub who then is responsible for installation, maintenance, and security, it is up to them to decide if they think it is worth the effort of including the new software on the environment. See **Figure 5.3: Partial list of preinstalled software on an Ubuntu runner** for a part of the installed languages.

Figure 5.3 Partial list of preinstalled software on an Ubuntu runner



It is not recommended to assume that a specific versions of an SDK (or other software) is always installed on the runners by default. It's up to GitHub to decide when a version is updated to a newer version, which could potentially break your job definition. When a version is being deprecated, GitHub announces that up front, and will start generating warnings in the runner logs to urge users to start upgrading. We have seen this for example with the

deprecation of Node 12 in favor of Node 16: large amounts of GitHub Actions were still using the older version, and a lot of jobs started to fail because of it. Usually this means that the latest LTS (Long-Term Support) release is supported.

When you know your job is dependent on having for example Node 14 installed, then specify that in the job definition itself. See Listing 5.2: Define node version needed for an example.

Listing 5.2 Define node version needed

```
steps:
  name: Install node with correct version
  uses: actions/setup-node@v3
  with:
    node-version: 14

  name: build your node application
  run: |
    npm install
    npm run build
```

There are setup actions available for widely used SDK's and maintained by GitHub in their 'actions' organization, for example:

- actions/setup-dotnet
- actions/setup-java
- actions/setup-go
- actions/setup-node
- actions/setup-python

By specifying the version you need, the job will always have the right version available, which saves you time and errors when the default environment is updated to the latest LTS of that SDK. For popular versions, the last three versions are also cached on the runner image. So, when the LTS version of Node on the runners is 18, versions 16 and 14 are stored in the 'opt/hostedtoolcache' folder of the GitHub hosted runner. The actions that can switch between versions know about this common folder and will use the version for the corresponding folder when told to do so. Switching to the

correct version will not require a full download to save execution time. If the version is no longer in the ‘hostedtoolscache’ directory, the setup actions will download it from the corresponding GitHub repository and install it from there.

5.7 Default shells

What the default shell used for your steps in your job depends on the operating system:

- Windows: pwsh (PowerShell core)
- Linux: bash
- macOS: bash

You can always check if the OS you are using has other shells installed as well. For example, each GitHub hosted operating system has the following already pre-installed for you:

- bash (on Windows, the bash shell included with Git for Windows is used)
- pwsh (PowerShell code)
- python

You can then specify the shell to use for each run step as seen in **Listing 5.3**.

Listing 5.3 Specifying the shell

```
steps:  
  run: echo "Hello world"  
  shell: pwsh
```

You can also make the desired shell the **default** for all jobs in the workflow as shown in **Listing 5.4**. Using that will set the default shell for any step in every job in the workflow to your value. If a single step still needs a different shell, you can use the `shell` keyword at the step level to override the default.

Listing 5.4 Specify the default shell for all jobs


```
name: example-workflow
on:
  workflow_dispatch:
default:
shell: pwsh
```

5.8 Installing extra software

If the software you need is not installed on the runner environment, there are lots of actions available on the public marketplace that will install the software for you. Do be aware of the security implications of these actions: they download binaries from somewhere (often they download from the GitHub releases of their corresponding repositories) and start installing it on the runner. There are actions that perform the download themselves, as well as actions that download and execute an installation script from a vendor (for example through an npm package). Verify those actions beforehand and follow best practices for using them, like pinning their version with a commit SHA hash for the version you have checked. For more information on version pinning, see Chapter 3.

5.9 Location and hardware specifications of the hosted runners

GitHub hosted runners are either hosted by GitHub directly (Linux + Windows runners are hosted in Microsoft Azure) or by a third party (for macOS runners). Currently there is no option to define in which region the runners are hosted. If you have data residency requirements, you will have to create a setup for self-hosted runners in the region of your choice.

The default Linux and Windows based runners are hosted on Standard_DS2_v2 in Microsoft Azure. That means they have the following specs available:

- 2-core processors (x86_64)
- 7 GB RAM
- 14 GB of hard storage

macOS based runners have these specs available:

- 3-core processors (x86_64)
- 14 GB RAM
- 14 GB of hard disk storage

Next to the default runners there are also more powerful macOS runners in case you need extra compute to speed up your jobs. This can be very helpful if you have CPU / RAM intensive workloads that hit the limits of the default runners. Read chapter 7 for more information about finding the resources used in your runners.

The extra large macOS runners can be targeted with the following labels: `macos-12-x1` or `macos-latest-x1`. These runners have 12 core CPU's available and for the rest the same specs as the normal macOS runners.

5.10 Concurrent jobs

Depending on the plan you are in there are some limitations on the amount of jobs that can run at the same time. See **Table 5.2: Overview of maximum concurrent jobs**.

Table 5.2 Overview of maximum concurrent jobs

GitHub plan	Total concurrent jobs	Maximum concurrent macOS jobs
Free	20	5
Pro	40	5
Team	60	5
Enterprise	5000	50

5.11 Larger GitHub hosted runners

When the hardware specs for the normal hosted runners are not enough for your workload, you can use larger GitHub hosted runners. Larger runners are only available in GitHub Enterprise Cloud and not on the server. With these runners you can control how much hardware capacity you give the runners

(CPU, RAM, and disk space) and how many runners can be spun up on demand for you (see **Figure 5.4, Creating custom hosted runners with more hardware options**). The maximum amount of concurrent jobs for these runners can be configured from 1 to 250 per configuration. For the entire organization only 500 of these runners can be active at the same time. That means 500 concurrent jobs can be executed at the same time on this type of runners.

Figure 5.4 Creating custom hosted runners with more hardware options

Runners / Create custom hosted runner

Name

Runner image

Ubuntu version

20.04

Runner size

4-cores - 16 GB RAM - 150 GB HDD

✓ 4-cores
16 GB RAM - 150 GB HDD

8-cores
32 GB RAM - 300 GB HDD

16-cores
64 GB RAM - 600 GB HDD

32-cores
128 GB RAM - 1200 GB HDD

64-cores
256 GB RAM - 2048 GB HDD

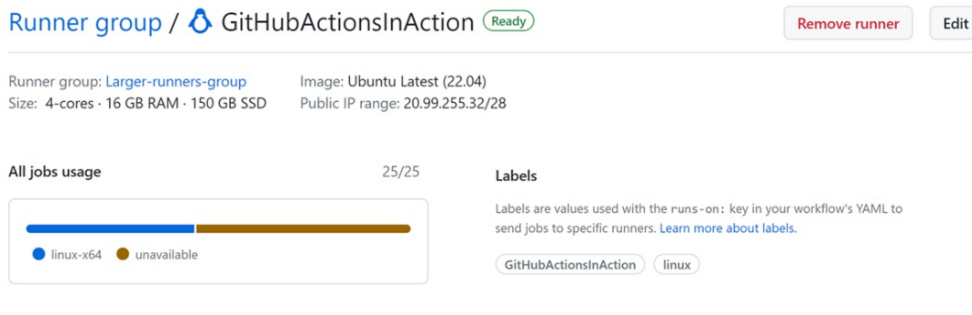
Maximum runners

1

Runners will not auto-scale above the maximum. Use this setting to limit your cost.

After creating the runners and adding them to a runner group, you can target them either with a label for their OS ('linux' or 'windows') or for the runner configuration you created (without spaces). See **Figure 5.5: Larger GitHub hosted runners** for an example.

Figure 5.5 Larger GitHub hosted runners



This type of runners also allows you to assign a static public IP address range, which will be unique to your configuration. That means that no one else will have a runner executing with a public IP address in this range. The runners will get assigned a public IP address from a reserved range based on the configuration group. That gives you the opportunity to use that range for allowing connectivity into your resources (like an API endpoint, or a database). The IP address is reserved for the configuration for 30 days. If the group is not used in the last 30 days, the address range is removed cannot be recovered. In that case you can edit the configuration and let it provision a new IP range for you.

Note that you can provision a maximum of 10 larger runner configurations with IP address ranges **per organization**, and another 10 that are shared across the entire Enterprise.

5.12 GitHub hosted runners in your own Azure Virtual Network

It's also possible to let GitHub host their Linux or Windows runners inside of your own virtual network in Azure. That means you configure a virtual network in your Azure subscription in such a way that you can connect from the runners to your own private resources, and still let GitHub manage the virtual machines, including the software and runner that is installed on them. Billing of those runners will go through the normal billing process, as the only thing hosted on your Azure subscription, are the Virtual Network, a Network security group and the Network interfaces that GitHub uses for the virtual machines. An example of the resource group in Azure is shown in Figure 5.6. The setup of these runners is configuring a normal runner group




in your organization or enterprise, and linking that to a preconfigured virtual network in Azure with a list of inbound and outbound networking rules that can be found in the GitHub documentation.

Figure 5.6 Bring your own Azure virtual network

Resources Recommendations (2)

Filter for any field... Type equals all X Location equals all X Add filter

Showing 1 to 3 of 3 records. Show hidden types ⓘ

<input type="checkbox"/> Name ↑↓	Type ↑↓	Location ↑↓
<input type="checkbox"/>  githubactionsrunner_938da66f04fb4c11_06ecbb50	Network Interface	North Europe
<input type="checkbox"/>  runners-nsg	Network security group	North Europe
<input type="checkbox"/>  runners-vnet	Virtual network	North Europe

5.13 Billing of GitHub hosted runners

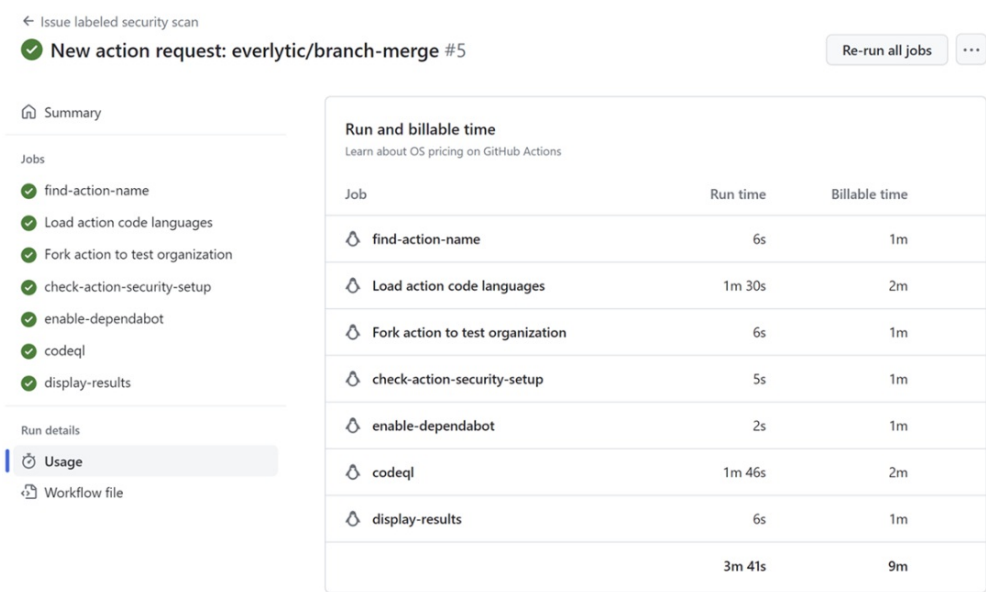
For GitHub hosted runners, GitHub Actions is billed at the minute level **per job** (self-hosted runners are free). If your job takes 4 minutes and 30 seconds, you will be billed 5 minutes of action minutes for that **job**. See **Figure 5.7, Example of action minutes usage of a workflow**, for an example of the job overview.

In **Table 5.1, Example of billable time**, you can see the calculation of billable time if I used a private repository for this workflow. This example would costs me 8 minutes instead of the 3m 37s that the total runtime has been as you can see in the column ‘billable time’.

This example shows why it can be worthwhile to have sequential steps in a job, instead of running everything in parallel jobs. Running everything in parallel can save you time, to get feedback faster back to a developer, but can also cost more action minutes. Take this into account when creating workflows: depending on the trigger used, you might not need to run everything in parallel to get fast feedback to a developer. An example of this is running on a pull request trigger: a pull request is often an asynchronous event that gives you more time to run all the checks you need to allow the pull request to be merged. Therefore, you do not need the fastest run duration

and have time to run steps as a sequence, instead of running them in parallel across more than one job.

Figure 5.7 Example of action minutes usage of a workflow



Depending on the OS of the hosted runner, there is also a multiplier calculated on top of the time you use the runner (see **Table 5.3: Breakdown of costs for action minutes based on dual core processors**). Billing only applies to workflow runs in private or internal repositories. Runs in public repositories are for free for the default hosted runners (See **Paragraph 5.4 GitHub Hosted Runners**).

Table 5.3 Breakdown of costs for action minutes based on dual core processors

OS	Per minute rate	Multiplier	Description
Ubuntu	\$ 0.08	x1	Base unit for calculations
Windows	\$ 0.16	x2	Additional hosting and licensing cost
macOS	\$ 0.80	x10	More hardware requirements and licensing cost

For larger runners (see **Paragraph 5.11 Larger GitHub hosted runners**) the calculation is based on the default (2 or 3 core runner) with a multiplier for

the number of cores the larger runner has. So, if the larger Windows based runner has 32 cores, the action minutes on this runner will be $32/2 = 16$ times more expensive than the run on the default Windows runner.

Depending on the plan you use with the account, you get several action minutes for free each month. The free action minutes included in the plan are only available for the standard dual core processor based GitHub hosted runners (as well as the default 3 core processor variant for macOS). Runs on larger runners will **not** count against this free entitlement. The list of included minutes and storage per plan can be found in **Table 5.4: Action minutes and storage included per plan**.

Table 5.4 Action minutes and storage included per plan

Plan	Storage	Minutes (per month)
GitHub Free	500 MB	2,000
GitHub Pro	1 GB	3,000
GitHub Free for organizations	500 MB	2,000
GitHub Team	2 GB	3,000
GitHub Enterprise Cloud	50 GB	50,000

The storage used by a repository is the total storage used by GitHub Actions artifacts and GitHub Packages. Storage is calculated based on hourly usage and is rounded up to the nearest MB per month. For that reason it is recommended to look at the amount and size of artifacts generated in each run. Check if you really need to retain those artefacts for the default 90 day period. The retention period for artifacts can be set as a default value at the Enterprise and Organization level, or be configured on a per repository basis. See **Figure 5.8: Artifact retention settings at the organization level** for an example how you can configure the retention period.

Figure 5.8 Artifact retention settings at the organization level.

Artifact and log retention

Choose the default repository settings for artifacts and logs.

Artifact and log retention

100 days

Save

Your enterprise has set a maximum limit of 400 days. [Learn more.](#)

Let's look at an example how storage is calculated. Note that prices for the storage in Actions and Packages are combined. You store an artefact of 100 MB when running a workflow. Five hours after running the workflow you delete it's history. That means we have stored the 100 MB for 5 hours. This needs to be calculated against the total amount of hours in a month, which can be calculated as 744 hours (in a month with 31 days). For the two hours we can calculate the Mb-Hours as $5 * 100 = 500$ MB-hours. That means that the price of 500 MB for that duration can be calculated as MB-Hours divided by the hours in a month. That will be $500 / 744 = 0,672$ MB-Months. This number will be rounded up to the nearest MB before billing, so that means we'll need to pay for 1 MB. Prices for the storage in Actions and Packages are \$0.248 for storing 1 GB of data for the entire month (of 31 days).

5.14 Analyze usage of GitHub hosted runners

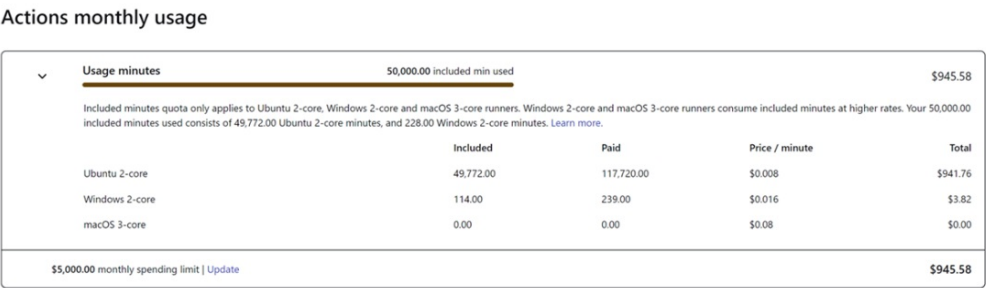
You can get insights into the usage of GitHub Actions at the following levels:

- Enterprise
- Organization
- Personal user account

At each level you can go into *settings* | *billing* and get insights into the action minutes being used in the current billing period. You will need to have 'Admin' access for the level you request this information for, or be in the 'billing manager' role. You can see the overall usage in **Figure 5.9: Billing and usage information for GitHub Actions**. In it you can find when the billing period resets (in this example in 30 days), the monthly free minutes

included in your plan, and see the split between the different GitHub hosted runner types. If you have configured a monthly spending limit you will also see how far along the usage for the current billing period is.

Figure 5.9 Billing and usage information for GitHub Actions



To get detailed information on a per repository and per workflow basis, you can request the usage report. A selection screen will offer you to choose from the following periods to get the usage information for:

- Last 7 days
- Last 30 days
- Last 90 days
- Last 180 days

A link to download the report in comma separated values (CSV) will be send to your email address. Generating the report can take up to a couple of hours. The information included in the CSV can be found in **Table 5.5: Overview of columns in the usage report**. Be aware that there is currently no way to set up automatic reporting for you spending on GitHub.

Table 5.5 Overview of columns in the usage report

Column	Description
Date	Information is grouped per day (based on UTC)
Product	Either ‘Actions’ or ‘Shared storage’
SKU	‘Compute + OS’ for Actions and ‘Shared storage’ for the storage results
Quantity	Number of units used on that date
Unit Type	Either action minutes or GB per day (for ‘Shared storage’)

Price Per Unit (\$)	Cost per unit
Multiplier	Multiplier on the action minutes (Windows and macOS are more expensive)
Owner	Owner of the repository (organization or user)
Repository Slug	The short name of the repository the workflow belongs to
Username	The user that triggered the workflow
Actions Workflow	Path to the workflow file inside of the repository

5.15 Self-hosted runners

In addition to GitHub hosted runners it is also possible to host your own runners. Those runners are under your control regarding installation and configuration. That also means it is **your** responsibility to keep the environments maintained, updated, as well as secured properly. Self-hosted runners can be helpful if you need more control over the environment, like for example running them in your own network so they can communicate with your internal environment, like connecting to a database, or other internal/private service. When you need hardware or software that is not available from the GitHub hosted runners, self-hosted runners can be an option as well: you can install them anywhere you need it. Most often the use-case we see for self-hosted runners is having a runner inside of your company firewall, licensed software that need to be installed, or adding more powerful hardware combinations, like a GPU enabled environment.

There are several security related aspects to be aware of when using self-hosted runners, which you will learn about in **Chapter 6** where we dive deeper into setting up you own runner.

5.16 Summary

In this chapter you have learned about:

- The runner application provides the runtime of the jobs and executes the

steps in your job definition

- The differences between GitHub hosted runners and self-hosted runners
- How to target GitHub hosted runners with either the latest version of that runner or provide a version specific label
- The differences between the different hosting environments across the provided OS-es, like providing a different default shell and installed tools
- How to install extra software on the runners that is not available by default, or specify a version that you rely on
- You can create larger hosted runners to give your jobs more hardware to execute your jobs on, potentially making your jobs more efficient
- How billing of GitHub hosted runners works and how to get insights into the biggest users of your action minutes and storage

6 Self-hosted runners

This chapter covers

- Setting up self-hosted runners
- Securely configuring your runners
- Using ephemeral runners
- Choosing autoscaling options
- Setting up autoscaling with actions-runner-controller

In **Chapter 5, Runners**, we have seen how we can use GitHub hosted runners, for which purposes they can be used, as well as how billing works for those hosted runners. You can also install your own runners in your own environments. These are then called *self-hosted runners*. Creating self-hosted runners gives you full control over their execution environment, like placing it inside of the company network, or adding specific hardware or software capabilities. Self-hosted runners can also be beneficial from a cost perspective, since you do not need to pay any Action minutes to GitHub for jobs that run on self-hosted runners. There is, of course, a ‘cost’ associated with hosting, setup, and system administrative tasks that you will have to do to keep the environments you host the runners on up to date and secure.

One example where self-hosted runners can be beneficial is that you can run a self-hosted runner inside of your company network so the runtime can connect to a database service to run certain integration test or deploy into your production environment that cannot be accessed from outside the company perimeter. Maybe you need to have a GPU enabled machine for certain jobs. Or perhaps you have a need for certain (larger) Docker containers: installing a self-hosted runner on a machine that already has those containers downloaded and pre-cached can save a lot of time and network bandwidth.

6.1 Setting up self-hosted runners

Setting up a self-hosted runner can be done by installing the runner application and following the steps from the documentation for the OS that will be hosting the service. The service itself is Open Source and can be found in the following repository: <https://github.com/actions/runner>. This repository also hosts the releases of the application as well. The application is based on .NET Core runtime and can be executed on a large number of Operating Systems and processor types. For example, from x86 / x64 to ARM processors, and on Linux, Windows and macOS. That means you can even run the service inside of a Docker container, or on a Raspberry Pi!

The supported operating systems for self-hosted runners can be found in **Table 6.1: Overview of supported Operating Systems for self-hosted runners**. For the current list of supported systems, check the documentation at <https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners#supported-architectures-and-operating-systems-for-self-hosted-runners>.

Table 6.1 Overview of supported Operating Systems for self-hosted runners

Operating system	Supported
Linux	Red Hat Enterprise Linux 7 or later
	CentOS 7 or later
	Oracle Linux 7
	Fedora 29 or later
	Debian 9 or later
	Ubuntu 16.04 or later
	Linux Mint 18 or later
	openSUSE 15 or later
	SUSE Enterprise Linux (SLES) 12 SP2 or later
Windows	Windows 7 64-bit
	Windows 8.1 64-bit
	Windows 10 64-bit
	Windows Server 2012 R2 64-bit
	Windows Server 2019 64-bit
macOS	macOS 10.13 (High Sierra) or later

To get started installing the runner you will need to have an environment that

is supported by the .NET core version (see the docs <https://github.com/actions/runner> for the current version). .NET core does not need to be preinstalled, the runner is self-contained. It also includes the two most recent versions of the node binaries it supports, as most of the public actions will need node to execute. For running the checkout action you will need to have a recent version of git installed.

If you want to run Docker based actions, you will also need to have Docker installed and you need to install the runner on a Linux machine. Windows and macOS are not supported for running Docker based actions.

The environment also needs to be able to connect either to GitHub or a self-hosted GitHub Enterprise Server. On Linux you will also need to have an account to run the service as 'root', so you will need sudo privileges. On Windows you will need to have 'administrative' privileges to configure the runner as a service. Installing the service is done by downloading the runner and executing the configuration to tell it the following information:

- To which GitHub service does this runner need to connect? It can either be github.com or against your own GitHub server. This cannot be changed after installation.
- For which hierarchical level is this runner created? A runner can be linked to an entire Enterprise, for a specific Organization, or for a specific Repository. This setting cannot be changed after the installation.
- A configuration token that is used for the installation. The token can be generated by a user (it is shown in the GitHub UI by default) and is only valid for 1 hour. You can only use a token one time and only during installation. You can create an installation token through the REST API on demand by sending a POST request to <https://api.github.com/orgs/<ORG>/actions/runners/registration-token>. The token in the result will also be valid for only 1 hour. The expiration date is also present in the response.
- Name of the runner, will default to the hostname. Cannot be changed afterwards
- Runner group to place this runner in, will default to the runner group named 'default'. This can be changed afterwards, as the runner itself has no idea what group it belongs to after the installation: this is all stored

on the GitHub side. With runner groups you can allow a group of runners to be used on certain repositories. This will be explained in more detail in Chapter 7.

- The labels that will be associated with this runner. You can add more labels through the UI or API later on as the runner itself has no ideas of the labels that are assigned to it. That configuration is stored on the GitHub side so it can be used from that end to find the appropriate runner to send the job to when queued. There is no upper limit on the amount of labels you can add, so you can be as specific as you prefer. The only restriction is that the label cannot be longer than 256 characters and cannot contain spaces.

See Listing 6.1: Installation script for creating a runner on Linux for an example of downloading the runner software from a GitHub release and extracting it to get started. Listing 6.2 then contains the script for configuring the runner for an organization with only the default token that is present in the GitHub UI. This token is valid for one hour.

Listing 6.1 Installation script for creating a runner on Linux

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

# Download the latest runner package
$ curl -o actions-runner-linux-x64-2.305.0.tar.gz -L https://github.com/actions/runner/releases/download/v2.305.0/actions-runner-linux-x64-2.305.0.tar.gz

# Optional: Validate the hash
$ echo "737bdcef6287a11672d6a5a752d70a7c96b4934de512b7eb283be6f51" | sha256sum --check

# Extract the installer
$ tar xzf ./actions-runner-linux-x64-2.305.0.tar.gz
```

Listing 6.2 Configure and start the runner

```
# Create the runner and start the configuration experience
$ ./config.sh --url https://github.com/devops-actions --token ABO

# Last step, run it!
$ ./run.sh
```

Some extra configurations parameters that are not required are:

- **work:** overwrite the default location where the downloaded work will be stored. Defaults to the ‘_work’ directory relative to the runner application directory.
- **Replace:** indicate if you want to replace an existing runner with the same name. Defaults to false.

On Windows the configuration script will ask you if you want to execute the runner as a service, so that it will start with the start of the environment. On Linux you will have to configure the service yourself using the `svc.sh` script. See Listing 6.3 for an example.

Listing 6.3 Install the runner as a service on Linux

```
# Install the service, parameter USERNAME is optional to run as a
sudo ./svc.sh install

# Start the service
sudo ./svc.sh start

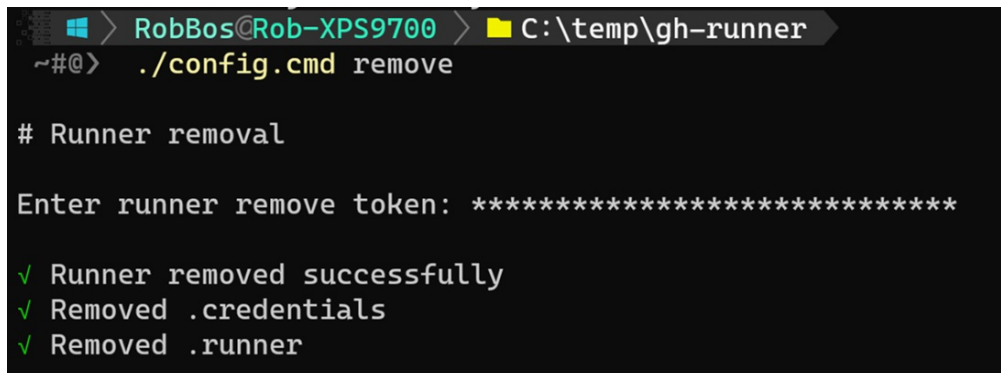
# Check the status of the service
sudo ./svc.sh status

# Stop the service
sudo ./svc.sh stop

# Uninstall the service
sudo ./svc.sh uninstall
```

For removing and deregistering the service on Windows, you can run the config command again with the ‘remove’ parameter. The token needed to deregister is the same type of token as with the installation: it’s a one-time token, generated specifically for the (de)registration at that level in the GitHub environment (Enterprise / Organization / Repository). The token that you use has to come from the same configuration point that you used for the registration, or else the removal command will fail. So get a token from the same enterprise, organization, or repository where you registered the runner. See Figure 6.1: Deregister and removing a runner.

Figure 6.1 Deregister and removing a runner

A terminal window with a dark background. The title bar shows 'RobBos@Rob-XPS9700' and the current directory is 'C:\temp\gh-runner'. The user has entered the command './config.cmd remove'. The output shows a confirmation message, a prompt for a remove token (which is masked with asterisks), and three green checkmarks indicating successful removal of the runner, its credentials, and the runner directory.

```
RobBos@Rob-XPS9700 C:\temp\gh-runner
~#> ./config.cmd remove

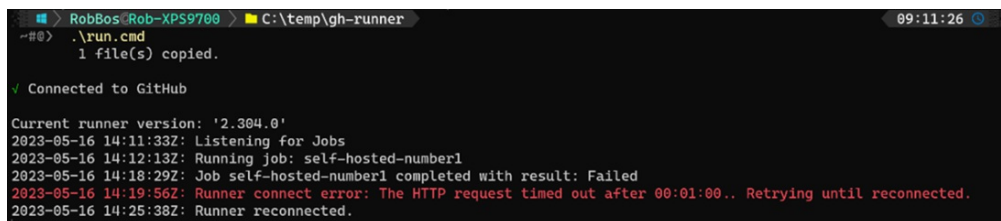
# Runner removal

Enter runner remove token: *****

✓ Runner removed successfully
✓ Removed .credentials
✓ Removed .runner
```

After configuring the service, you can either start the process as a service (so that it will always be running and ready to receive work) or start it as a one-time process. As a one-time process it will announce itself to GitHub, wait for the work to come in, and then stop. It will also not be started together with the operating system when not configured as a service. An example of a running service that is waiting for work and then executing a job can be found in Figure 6.2: Runner service is executing work.

Figure 6.2 Runner service is executing work

A terminal window with a dark background. The title bar shows 'RobBos@Rob-XPS9700' and the current directory is 'C:\temp\gh-runner'. The user has entered the command './run.cmd'. The output shows the runner connecting to GitHub, displaying its version, and then listening for jobs. It receives a job named 'self-hosted-number1', which it completes with a 'Failed' result. It then shows a 'Runner connect error' due to a timeout and finally 'Runner reconnected'.

```
RobBos@Rob-XPS9700 C:\temp\gh-runner
~#> ./run.cmd
1 file(s) copied.

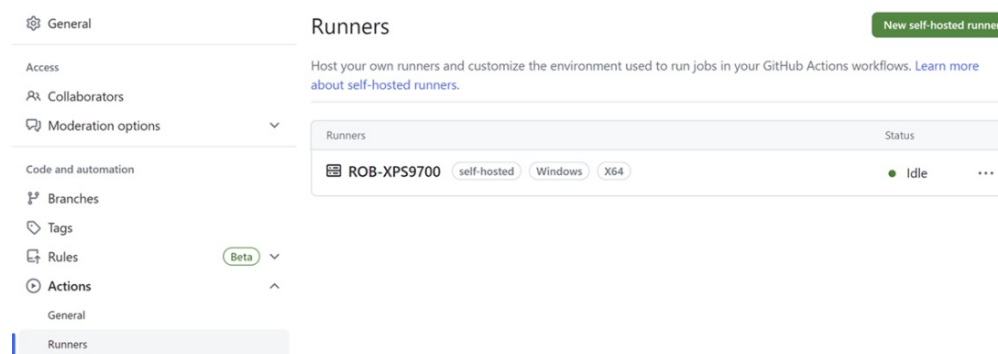
✓ Connected to GitHub

Current runner version: '2.304.0'
2023-05-16 14:11:33Z: Listening for Jobs
2023-05-16 14:12:13Z: Running job: self-hosted-number1
2023-05-16 14:18:29Z: Job self-hosted-number1 completed with result: Failed
2023-05-16 14:19:56Z: Runner connect error: The HTTP request timed out after 00:01:00.. Retrying until reconnected.
2023-05-16 14:25:38Z: Runner reconnected.
```

If the runner is configured as a service you can also check it's connectivity back to GitHub by running the following command: `.\run.cmd --check --url <url> --pat <personal access token>`. You need a Personal Access Token (PAT) because the runner does not have this authentication information available to connect back to the URL.

The runner will show up in the runner list at the corresponding level it was created for (Enterprise/Organization/Repository) under Settings à Actions à Runners. See Figure 6.3: Runner overview. In this view you can search for runners with a certain label by using the search box and using for example this search query: `label:self-hosted`.

Figure 6.3 Runner overview



6.1.1 Runner communication

The way the runner communicates with GitHub, is by setting up an outgoing https connection. The communication is created as what is called ‘a long poll connection’: it asks GitHub if there is work queued to be executed for this specific runner, and then waits for 50 seconds for a response, before the connection is severed. Immediately after closing the connection, a new connection is started that does the same thing, and so on until the runner is completely stopped. The nice part about this setup is that you can configure the runner anywhere, as long as the firewall is open for **outgoing** connections over port 443. There is no inbound connection to be made from GitHub back into your network.

The runner itself has no knowledge of the GitHub side of the connection. For example, it does not know for which repositories it is configured to run, the GitHub organizations that can use it, or if it has been setup on the Enterprise or Repository level. It only knows the GitHub url it needs to use to ask for work. There is no GitHub user associated with the runner itself. A runner also has no idea what kind of environment it is running in. During installation, it checks the type of operating system that is used (Linux / Windows / MacOS), the CPU architecture of the environment (x64, ARM32, ARM64) and sends that to GitHub as labels that can be used for jobs to ‘target’ a runner. The labels can later be changed on the GitHub side, since the runners have no idea what labels are assigned to them.

The runner installation will create two files that are important for its

communication back to GitHub. In Listing 6.4 you'll find the content of the `.runner` file in the application folder of the installed runner. As you can see it is stored as a JSON file with settings for the agent Id and Name, together with the settings for the runner group (pool) it was configured with. Here you also find the server being used and the GitHub url that was used during configuration. If you move the runner between runner groups this information will not be updated, it's only written when configuring the runner. The `githubUrl` property does have an owner/repo in the url but this is only used for asking the GitHub environment for work.

Listing 6.4 Content of `.runner` file

```
{
  "agentId": 23,
  "agentName": "ROB-XPS9700",
  "poolId": 1,
  "poolName": "Default",
  "serverUrl": "https://pipelines.actions.githubusercontent.com/f",
  "githubUrl": "https://github.com/GitHubActionsInAction/demo-act",
  "workFolder": "_work"
}
```

In Listing 6.5 you can find the content of the `.credentials` file where a longer lived authentication token is stored after the runner is registered to GitHub with the registration token in the config command.

Listing 6.5 Content of the `.credentials` file

```
{
  "scheme": "OAuth",
  "data": {
    "clientId": "21ecc1ca-2d1a-4c44-abcd-309480c44a33",
    "authorizationUrl": "https://vstoken.actions.githubusercontent.com/f",
    "requireFipsCryptography": "True"
  }
}
```

The OAuth credential that are used to authenticate the connection to GitHub with are stored in the file `.credentials_rsaparams`, which is encrypted on Windows with an RSA private key with 2048 bit length encryption and can

only be read on the local machine. On Linux this file is **not encrypted** and can be copied over to another machine and start the runner process there. The file is needed for runners that are expected to reboot (for example after upgrading) and then register themselves again. It is also used to refresh the long polling connection that times out after 50 minutes.

The one thing you can do with these credentials, is execute the runner service and wait for an incoming job to execute. Having this file available for reading from the user that is used to execute the runner is considered a security risk. The job could read all the information and start a new runner elsewhere with the same configuration. This setup is there for backwards compatibility reasons and are a know risk. The recommended configuration for the runners is using the Just-in-Time (JIT) setup discussed in a later chapter. The JIT setup uses the same files, but the token used for configuration is only valid once.

Since the runner communication is an outgoing channel from the runner to the GitHub environment, there are events that happen when the communication stops. When there is no communication from the self-hosted runner to GitHub for more than 14 days, the runner will be removed from the listing and will need to be reconfigured before it is allowed to reconnect. When the runner is configured as ephemeral, it will be removed after 1 day of non-communication.

To be able to communicate with GitHub you must ensure that certain hosts can be reached from the runner environment. You can find the full list in the documentation here: <https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners#communication-between-self-hosted-runners-and-github>. Some interesting hosts are shown in Table 6.2: Hosts that the runner needs to be able to reach.

Table 6.2 Hosts that the runner needs to be able to reach

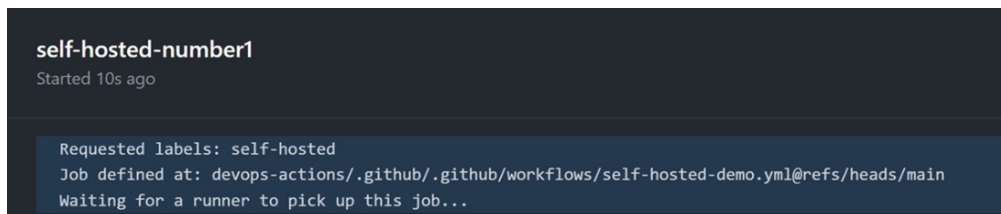
Purpose	Hosts
Essential operations	github.com api.github.com

	*.actions.githubusercontent.com
Downloading actions	codeload.github.com
Up/downloading job summaries and logs	actions-results-receiver-production.githubapp.com productionresultssa*.blob.core.windows.net
Runner version updates	objects.githubusercontent.com objects-origin.githubusercontent.com github-releases.githubusercontent.com github-registry-files.githubusercontent.com
Up/downloading artifacts and cache	*.blob.core.windows.net

6.1.2 Queued jobs

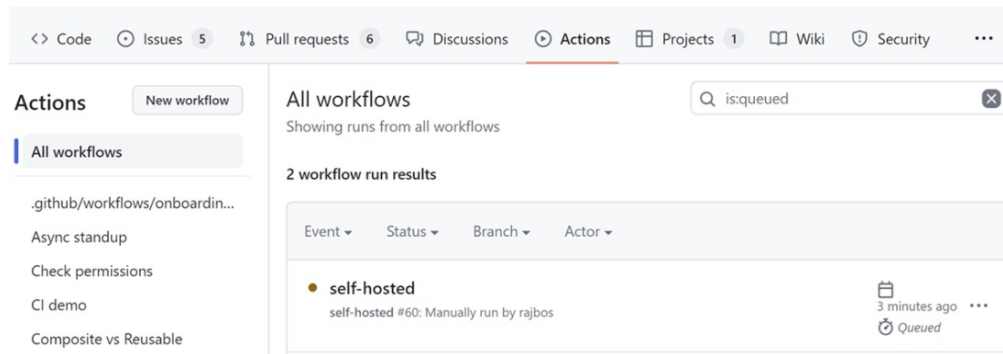
When a job is queued for a certain combination of labels, the job will stay in the queue if there is no runner online that matches all the labels the job is targeting. An example of a queued job with the labels that where targeted can be found in Figure 6.4: Queued job waiting for a runner to become active with the self-hosted label. The maximum duration of being queued for self-hosted runners is 24 hours. If there is no runner available within this period, the job will be terminated and a cancelation message is send to the user that triggered the job. Currently there is no API or User Interface to get an overview of all the jobs that are queued for either the Enterprise/Organization/Repository level.

Figure 6.4 Queued job waiting for a runner to become active with the self-hosted label.



You can only load that per workflow using the API, or for an entire repository like shown in Figure 6.5: Overview of queued workflows for a repository, where the overview has been filtered using the ‘is queued’ query.

Figure 6.5 Overview of queued workflows for a repository



6.1.3 Updating self-hosted runners

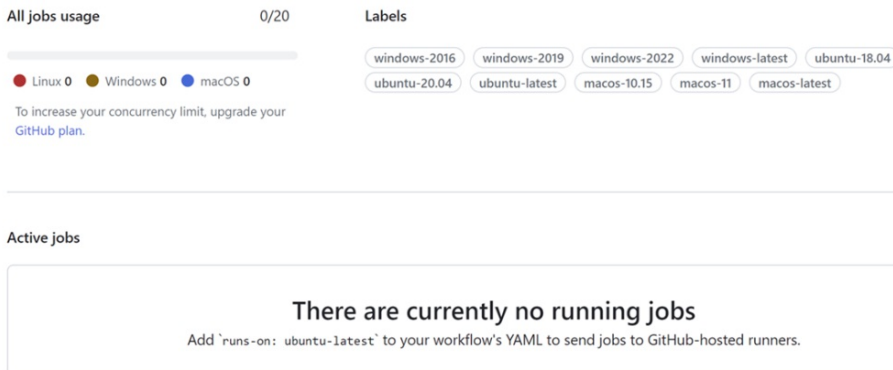
Self-hosted runners will automatically check with each job they execute if there is a new version of the runner available, by either calling the public GitHub repository on <https://www.github.com/actions/runner> where all runner releases are stored, or calling the GitHub Enterprise Server if you are using that. If the runner has not been used for seven days, it will also check for updates and run them if needed. New releases are created by GitHub when needed, which has been almost once a month in the past. The updates contain both fixes and updates. In case you host your runners in a locked down environment with for example no direct internet connection, you will need to make sure to keep the runners up to date yourself by pulling in updates regularly in your setup environment. In that case, also configure the runner with the `disable-autoupdate` parameter.

6.1.4 Available runners

You can find out which runners already have been configured for your Enterprise/Organization/Repository by going into the *Settings* of that level and then to *Actions | Runners*. An example is given in Figure 6.5: Available runners. Here we can see which GitHub hosted runners are available, as well as the labels that are available for those types of runners. If there are any runners executing a job, they will be visible in the 'Active jobs' panel.

Figure 6.6 Available runners

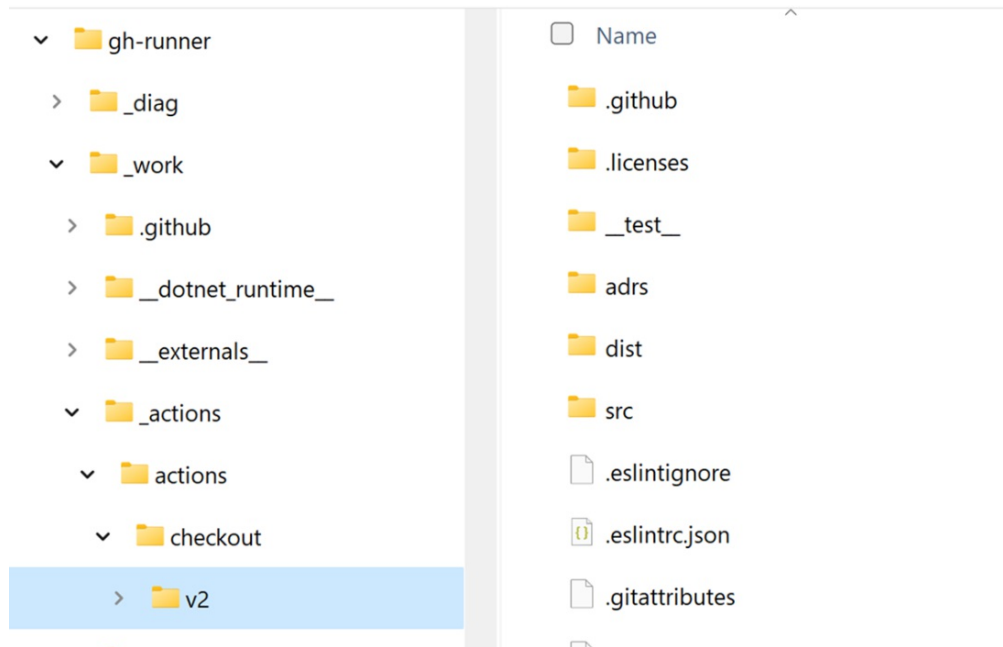
Runners / GitHub-hosted runners



6.1.5 Downloading actions & source code

When there is work queued for a self-hosted runner, the runner will first download the definition of the work that needs to be done from GitHub and then starts executing it. It will download the job definition and the extract all GitHub Actions statements that are included directly in the job definition. The next step is to download the repositories of the actions that are needed by going to the GitHub API and download the correct version of the action repo as a zip file. Each action (and version used) will be stored in the subfolder `_work_actions\actions\<action-name>\<version-reference>`, so that it only needs to be stored on disk once per job. See Figure 6.7: Runner action folder on disk for a screenshot of the runner folder on disk, with the `actions/checkout` action downloaded with a `v2` folder as the version tag. Here you can also see that the entire repository is downloaded, but not as a git repo (the `.git` folder is missing). That also means that every version you use in the job that is executed, will get its own version folder as well.

Figure 6.7 Runner action folder on disk



If the action is running a Docker container, the runner will either download the Docker image or start building the included *Dockerfile*, depending on the setup of the action. Using a prebuild image can significantly save time executing the action, since it will skip the time needed to build the action. Also note that the image will be built for **every single run** that the runner executes.

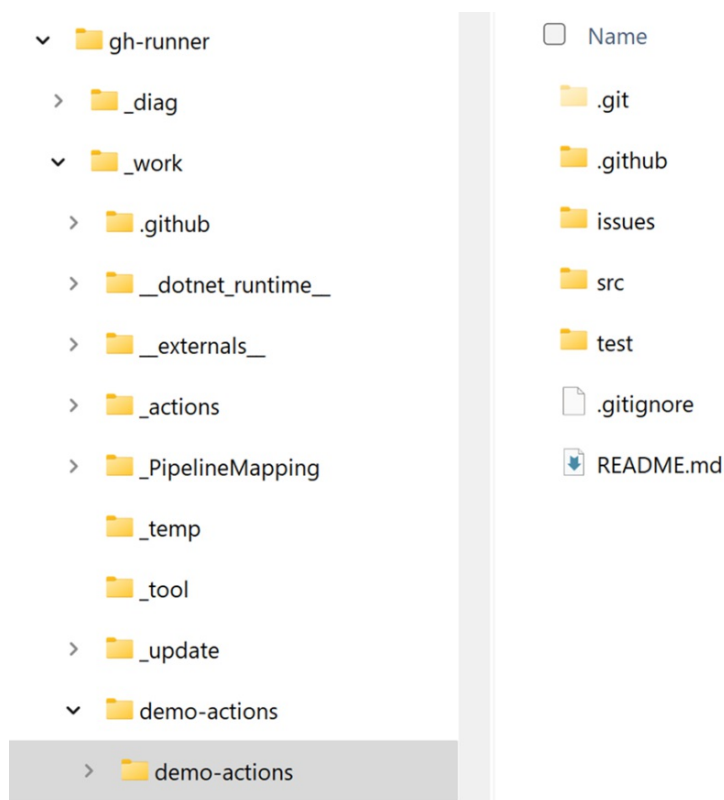
The runner uses the URL that was entered during the installation of the application for downloading the actions. It will suffix this URL the actions' using statement to get a link to the action repository it needs to download. This means it will use www.github.com when connected to GitHub in the cloud, or the URL to your GitHub Enterprise server when connected to a server.

In the case of composite actions or reusable workflows, the runner will download the definition to make sure it exists, but only expands these configurations and download those actions if and when the step or job is executed. This way the runner only downloads what is needed.

Keep in mind that the folder `_work_actions\actions\` will be cleaned at the start of each job the runner executes, to prevent any issues when an action stores data in these folders that might get overwritten during the job execution.

When downloading repositories with the actions/checkout action, a new directory is created in the `_work` folder with the name of the repository where the executing workflow is defined and then a folder with the name of the repository that is checked out. Usually these are the same, so in the example in Figure 6.8: Actions/checkout folder creation you end up with `'demo-actions/demo-actions'`, as that is the repository we are working with. You can also see that this is an actual git repository, as the `.git` folder is there. This gives you the option to switch branches if needed, create new commits and push them back upstream, or work with any tool that uses the git repo information, like for example the GitHub CLI that uses this to execute commands like creating issues and pull requests from the current repository.

Figure 6.8 Actions/checkout folder creation



6.1.6 Runner capabilities

The runner gets its capabilities from the environment it is installed in: if there is software installed in the runner, the job that is executed can make use of that software. The environment defines the compute power the runner has,

depending on how much RAM, CPU, and network capabilities it has. If there is a GPU available for the environment, the runner will automatically pick that up as well. If you want to execute a Docker based action, you will need to install Docker on the host. Be aware that a runner can only run a single job at the same time. It is possible to install multiple runners at the same environment, but from a security perspective that is not recommended: concurrent jobs can then influence and interfere with each other, since the runner will have access to the entire environment.

A best practice for indicating runner capabilities is to add them as labels to the runners so that users can target the capabilities they need. An example could be that there is a GPU available, then add the label `gpu`. You can also run with a default `self-hosted` label on all runners, and if you need a runner with more RAM available, target the runners for example with the label `x1`. You could even go so far as to have labels for both large RAM (`ram-x1`) and another one for disk size (`disk-x1`). This will also guide users into thinking about what they actually need and specify that with the labels they target: a simple linter job should not have to run on a runner with 64GB of RAM available if it does not need that much power. To make this message even more clear you can make use of internal billing for Action minutes used, with your own cost per minute for the different runner types. See **Chapter 7: Managing self-hosted runners** for more example on internal billing.

6.1.7 Self-hosted runner behind a proxy

Proxy support is available for self-hosted runners. You can either use the standard environment variables (`https_proxy`, `http_proxy`, and `no_proxy`) to pass in the information or use an `.env` file in the runner application folder, containing the information shown in Listing 6.6. If you are also using Docker based actions, you also need to update the Docker configuration by adding the proxy settings to the `~/.docker/config.json` file.

Listing 6.6 Proxy configuration in `.env`

```
https_proxy=http://proxy.local:8080
no_proxy=example.com,myserver.local:443
https://username:password@proxy.local
```

6.1.8 Usage limits of self-hosted runners

Even though GitHub does not restrict the amount of concurrent jobs executed on self-hosted runners, or enforces the normal time outs for jobs, there are still some limits to be aware of when using self-hosted runners:

- The total workflow duration cannot be longer than 35 days. This includes job execution and time spent on waiting and approvals.
- Maximum queue time for a job on self-hosted runners is 24 hours. If the job has not started executing within this timeframe, it will be terminated.
- A job matrix can generate a maximum of 256 jobs per workflow run. If it generates more, the workflow run will be terminated and fails to complete.
- No more than 500 workflow runs can be queued in a 10 second interval per repository. Additionally queued jobs will fail to start.

6.1.9 Installing extra software

You are in full control of what you install on the self-hosted runners. After installation and adding it to the `$PATH`, you can use the software in your workflow definitions. You can either preinstall the software on the runner or install it on-demand. In general, you do not want to make your job definitions dependent on a specific type of runner to give you more freedom to switch runners. For the job itself it should not matter where it is running: if the job is self-contained, it will install the software it needs itself, like for example downloading the latest version of the Node and installing it. If the job needs it, it can specify the dependency itself:

steps:

- name: Install Node with version
 uses: actions/setup-node@v3
 with:
 version: 18.*
- uses: actions/checkout@v3
- name: use the CLI
 run: node --version # check the installed version

If you decide to start preinstalling software on the runner itself, like in a virtual machine setup, the general recommendation is to try and keep your runners as uniformly as possible. What we often see is that different user groups (e.g., teams) have different needs. When the runner definition starts to diverge, it can become unclear to the users what they can expect of the self-hosted runners. The best practice is then to add the installed software / capability as a label to the runner, so that the users can specify the right label to target the right runner. Keep in mind that jobs will only be queued on a runner if **all** the labels on the job match. An example would be:

```
runs-on: [self-hosted, gh-cli, kubectl]
```

This job can only run on a runner that has all three labels. Some of the software that we have seen used the most are system tools that are often used in jobs:

- The GitHub CLI
- Libraries that help you work with json or yaml, like ‘jq’ or ‘powershell-yaml’.
- Cloud specific CLI/ SDK’s, or other tools (AWS’s CDK, Azure CLI, etc.)
- SDK’s for the coding languages used the most in the company
- Caching the most used Docker images to save bandwidth costs and time to download the images
- Container tooling (Docker, BuildX, Buildah) and Kubernetes tooling (Helm, kubectl, etc.)
- Mobile Application tooling (Android studio, XCode)

Another option for this setup is to have a list of container images that your users can configure themselves when they need it. They then configure the use of the image with the container keyword on the job level (See Listing 6.7 for an example). All the steps in that job will run inside of the container, with any tool that you have installed in that container as well.

Listing 6.7 Run the entire job in you own container

```
jobs:  
  run-in-container:
```

```
runs-on: ubuntu-latest
container: alpine:3.1.2
steps:
  - uses: actions/checkout@v3
```

We often get the question how to get the same images for the runner as the VM image that GitHub uses for their hosted runners. Due to licensing reasons GitHub cannot distribute so called ‘golden images’ that already have everything pre-installed. They do give you the installation scripts to run and build your own image from the source code in the runner images repository. You can find the scripts to get started from this repository:

<https://github.com/actions/runner-images/blob/ubuntu22/20230611.1/docs/create-image-and-azure-resources.md>. All the pre-requisites to get started can be found in the same documentation.

6.1.10 Runner service account

The runner gets the rights to its environment from the way it was installed. On Windows you can configure it to run as a service with a certain service account. It will then have access to everything on the environment that the service account has access to, including any networking access.

For Linux and macOS the default setup is to run the service as root. You can configure it to use a non-root account. Be aware that this often causes some issues with actions or jobs that run inside of a container on non-ephemeral runners. The container runs with its own account setup, which is often root. The GITHUB_WORKSPACE folder will get mounted inside of the container. When the steps executed inside the container change a file or folder in the workspace, those files will get the root level access attached to them as well. Any subsequent cleanup of those files afterwards on the runner will fail if the runner is not executing as root.

6.1.11 Pre- and post-job scripts

The runner service can be setup with an environment variable that holds the path to a script that can either run as a step at the beginning of a job or as the

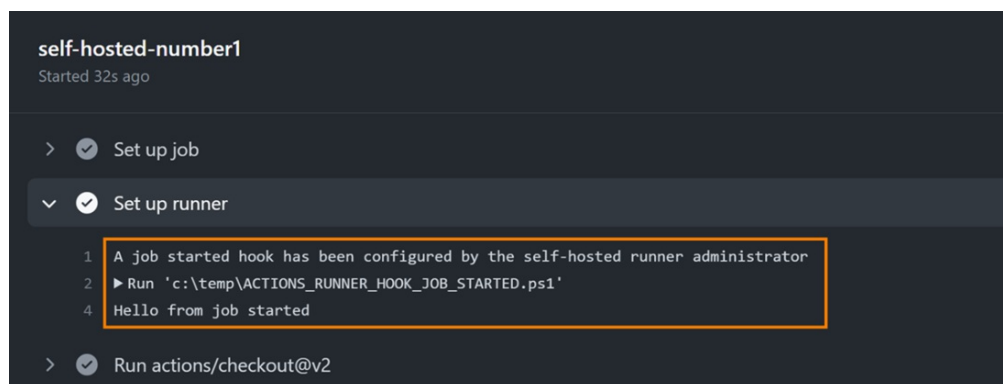
last step of the job. This can be used to prepare the runner environment with internal configurations. We have used this for configuring default read-only accounts to internal package managers and Docker registries. To configure the pre- and post-job scripts you need to save a script in a location the runner account will have access to, and then configure the corresponding environment variables for each hook:

- ACTIONS_RUNNER_HOOK_JOB_STARTED
- ACTIONS_RUNNER_HOOK_JOB_COMPLETED

Another option is storing these values as key-value pairs in a `.env` file inside of the runner application directory. The value of the settings needs to be the full path to the script that can be executed. If the runner account does not have access to that path, the setup runner will fail.

When the startup hook is configured, it will show up on the logs of the jobs that are executed on that runner as an extra step at the beginning of the job. In Figure 6.9: Setup runner job an example is shown. Note that the extra step runs **after** downloading all the action definitions. The job completed hook does the same thing, but at the end of the job as a last step.

Figure 6.9 Setup runner job



The environment variables can be set at any time, including after the installation, as long as they have been set before the next job executes. Any changes during a job execution will not be used.

The scripts are executed synchronously for the job run, just as a normal step. If the exit code for the script is non-zero, the step will fail, and the job will

stop executing. Additionally, these scripts will not have a timeout applied to them from the runner, so if needed you will need to configure a timeout handler inside of the script itself. The scripts also have access to the default variables as they are treated as a normal step in the job. That means you have access to variables like the `GITHUB_WORKSPACE` or `GITHUB_TOKEN`.

6.1.12 Adding extra information to your logs

There is support for showing extra information to your logs by placing a file called `.setup_info` in the runner's application folder. See listing 6.8 for the contents that GitHub uses for hosting their runners. The information is grouped with a tile for the group, which will result in grouped information in the *set up job* step in each run on this runner. The result is shown in figure 6.10. Note the use of `\n` for adding breaks in the output and start a new line.

Listing 6.8 Contents of the `.setup_info` file on GitHub hosted runners

```
[
  {
    "group": "Operating System",
    "detail": "Ubuntu\n22.04.2\nLTS"
  },
  {
    "group": "Runner Image",
    "detail": "Image: ubuntu-22.04\nVersion: 20230702.1.0\nInclud
  },
  {
    "group": "Runner Image Provisioner",
    "detail": "2.0.238.1"
  }
]
```

Figure 6.10 Results of the `.setup_info` file

```
✓ Set up job 5s
1 Current runner version: '2.305.0'
2 ▼ Operating System
3   Ubuntu
4   22.04.2
5   LTS
6 ▼ Runner Image
7   Image: ubuntu-22.04
8   Version: 20230702.1.0
9   Included Software: https://github.com/actions/runner-  

images/blob/ubuntu22/20230702.1/images/linux/Ubuntu2204-Readme.md
10  Image Release: https://github.com/actions/runner-images/releases/tag/ubuntu22%2F20230702.1
11 ▼ Runner Image Provisioner
12  2.0.238.1
```

6.1.13 Customizing the containers during a job

With the keyword **container**, users can specify that their job will run inside of a Docker container. The runner has default setups for the ‘docker create’ and ‘docker run’ commands it executes to get the container setup and running. You can overwrite the default commands with your own custom JavaScript file that runs when a job is assigned to the runner, but before the runner starts executing the job. This allows you to add custom volume mounts, configure your private container registry, or always run with a sidecar container. To configure the customization, store a reference to the script you want to run in the `ACTIONS_RUNNER_REQUIRE_JOB_CONTAINER` environment variable, or store this configuration in an `.env` file in the runners’ application folder as a key-value pair, where the value is the path to the JavaScript file.

Be aware that the script will run synchronously and thus will block the execution of the job until the script completes. There is also no timeout for the script, so you will need to handle a timeout mechanism inside of the script. The script will run in the context of the runner service, with the corresponding system and networking access.

The following configuration commands are available:

- `prepare_job`: Called when a job is started.
- `cleanup_job`: Called at the end of a job.
- `run_container_step`: Called once for each container action in the job.
- `run_script_step`: Runs any step that is not a container action.

Each command has its own definition file, with the filename being the name of the command and the json file extension. Another option is to use an `index.js` file that can trigger the correct command when it is called. Examples for setting up projects for docker, hooklib, and k8s can be found in this example repository from GitHub: <https://github.com/actions/runner-container-hooks>.

6.2 Security risks of self-hosted runners

Running jobs on self-hosted runners comes with a risk as well. The self-hosted runner might have too much access into your network and could be used for network traversal attacks (travel to other machines in the network, either for reconnaissance, or execute an attack and encrypt all files it has access to). On reused runners, data might be persisted on disk as well, leading to attacks like:

- *Cache poisoning*: overwriting `node_modules` at the runner level for example. The next job will use the dependency from the cache. This applies for any package manager's local caching system. An attacker can even prep your local Docker images with their own version, by mislabeling their version of a Docker image with a label you are using.
- Changing environment variables or things like SSH keys and configuration files for your package managers, like `.npmrc`, `.bashrc`, etc. This could be misused to let the package manager search for all packages on an endpoint controlled by an attacker, instead of using the default package managers URL.
- Overwriting tools in the `/opt/hostedtoolcache/` directory, which is the default storage for the actions like `setup-node`, `setup-java`, `setup-go`.
- The credentials used to register the runner with are always stored in the runner folder itself, which means it is accessible from inside a job. In the one of the next paragraphs about Just-in-time runners you will find a way to mitigate the risk of using these credentials to spin up a new runner in a different location.

As a general best practice, we always recommend to never run a job on a self-hosted runner without having full control over the job definition. Especially with public repos hosted on <https://github.com>, where any authenticated user

can craft a pull request to attack your setup, we cannot stress it enough to never run a job on your self-hosted runner, with access to your private network. GitHub protects you for these types of attacks by limiting the `GITHUB_TOKEN` for the `on: pull_request` trigger and by not running workflows automatically on incoming pull requests from new contributors as shown in Figure 6.11 Settings for running workflows from outside collaborators.

Figure 6.11 Settings for running workflows from outside collaborators

Fork pull request workflows from outside collaborators

Choose which subset of outside collaborators will require approval to run workflows on their pull requests.
from public forks.

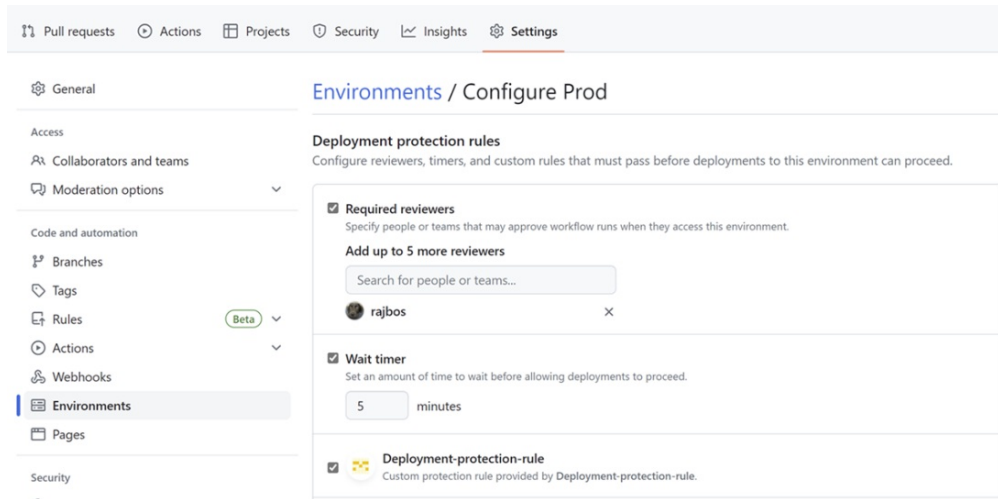
- ☐ Require approval for first-time contributors who are new to GitHub
Only first-time contributors who recently created a GitHub account will require approval to run workflows.
- ☐ Require approval for first-time contributors
Only first-time contributors will require approval to run workflows.
- ☒ Require approval for all outside collaborators

If you still have a need to run a job on your self-hosted runner, then either run it on a contained runner that is ephemeral (single use), does not have any networking connection options, and is only allowed to run after running stringent security checks, both manually and automated. You can for example run specific linters for GitHub Actions on your workflows to detect things like shell-injection attacks (running injected code from `run` commands). One of those linters is the `ActionLinter` from: <https://github.com/devops-actions/actionlint>, that will check for shell-injection attacks based on untrusted user input, like for example the title of an issue, the name of a branch, or the body of a pull request, etc.

Another way to protect your workflows and thus self-hosted runners is to have environment protection rules that allow a job to only run when for example a (manual) approval is given, or when custom checks (environment protection rules) have completed successfully. You can even configure an environment to only allow jobs to run when they come from a certain branch. In Figure 6.12: Environment protection rules you can find an example where a custom protection rule has been configured by using a GitHub App that will run the checks. Additionally, GitHub already blocks workflows to run when

coming from a fork, or from a new contributor to the repository.

Figure 6.12 Environment protection rules



6.3 Single use runners

There are three different runtime options for setting up self-hosted runners:

- Environments that are continuously available to run new jobs (running as a service). That means that the same machine is always ready to handle a queued job.
- Ephemeral runners that only are available for executing a **single job** and shuts down when that job is completed.
- Ephemeral runners with just-in-time tokens: only available for a single job and the token to register the runner with can only be used once.

Our recommendation is to use ephemeral runners with Just-in-time tokens whenever possible, because of the security concerns of persisting data from job1 that then can be (mis)used in job2 on the same runner. The GitHub hosted runners are configured the same way to protect data being leaked between customers. With this setup you also get a new fresh runner with every job, so there is less change of becoming dependent on a specific runner that has some files cached or software preinstalled. You are now required to specify all the tools you need to execute, in your job definition. This highly increases portability of your workloads as well.

6.3.1 Ephemeral runners

You configure an ephemeral runner by adding the `--ephemeral` parameter to the runner configuration script. This will let the runner to be online, waiting for a job to run. When a single job has been executed, the runner will deregister itself, and stop running. Not a single extra job will land on that runner. Be aware that the environment for the runner itself will still linger around, depending on the solution. For example, if you install this ephemeral runner on a virtual machine (VM), the VM will still be up and running, even though the runner itself deregistered itself from the GitHub environment and stopped itself from running. You can use the `ACTIONS_RUNNER_HOOK_JOB_COMPLETED` hook to handle the completion of the job, and for example clean up the VM (and spin up a new VM to handle new incoming jobs the same way).

6.3.2 Just-in-time runners

The token that is used to register self-hosted runners is always valid for an hour and is stored on the runner itself and available from inside a job. That makes it possible to steal these credentials and start a new runner with the same credentials in a different location. If you want to make this setup more secure by limiting the exposure of that ‘long lived’ credential, then you can use ‘Just-in-time’ runner configuration (JIT). The JIT runners work the same as with the ephemeral setup: The validity duration of the installation token is the only difference (one hour vs one **time** usage).

To get the configuration needed to register a new runner with the JIT configuration, you need to make an API call to the following endpoint (shown in Listing 6.9: Creating JIT runners):

`/orgs/{org}/actions/runners/generate-jitconfig`. The response can be used in the script to start up the runner. Instead of `--ephemeral` you call the script as follows: `./run.sh --jitconfig ${encoded_jit_config}`. The encoded JIT configuration value is only valid for one installation of a self-hosted runner, and it cannot be reused.

The new just-in-time runner will only accept one single job execution. On completion of that job, it is automatically removed from the Enterprise /

Organization / Repository level for which it was created, and the service stops running. It is still your own responsibility to clean up the runner and prevent reuse of the same environment. For that you can use the `ACTIONS_RUNNER_HOOK_JOB_COMPLETED` hook to handle the completion of the job.

Listing 6.9 Creating a JIT runner

```
curl --location 'https://api.github.com/orgs/GitHubActionsInActio
--header 'X-GitHub-API-Version: 2022-11-28' \
--header 'Content-Type: application/json' \
--header 'Authorization: Basic <encrypted token>' \
--data '{
    "name": "New JIT runner",
    "runner_group_id": 1,
    "labels": ["jitconfig"]
}'
```

6.4 Disabling self-hosted runner creation

Keep in mind that by default, every user with admin level access (enterprise, organization, or repository level) can get to the self-hosted runner screen and start installing a runner in their environment. To control this, it is possible to disable the creation of self-hosted runners at the enterprise or organization level. In Figure 6.13: Disable self-hosted runner at the organization level you can see the options you have at the organization level. This gives you more control over where a self-hosted runner can be created. At the organization level you can either allow for all repositories, disable it for all repositories, or enable the creation for specific repositories.

Figure 6.13 Disable self-hosted runner at the organization level

Runners

Choose which repositories are allowed to create repository-level self-hosted runners.

All repositories ▾

✓ All repositories

Repo-level self-hosted runners can be used by any repository in the organization

Selected repositories

Repo-level self-hosted runners can be used by specifically selected repositories

Disabled

Repo-level self-hosted runners are disabled for all repositories in the organization

ts and logs.

rn more.

On the enterprise level you can completely disable the creation of self-hosted runners for all organizations. The user interface for this can be seen in Figure 6.14: Disable self-hosted runners at the enterprise level. If you have Enterprise Managed User (EMU) organizations, then it is also possible to disable it for any repositories in the **personal** namespace that are in the user space for those organizations.

Figure 6.14 Disable self-hosted runners at the enterprise level

Runners

Choose which organizations are allowed to self manage self-hosted runners at the repository level.

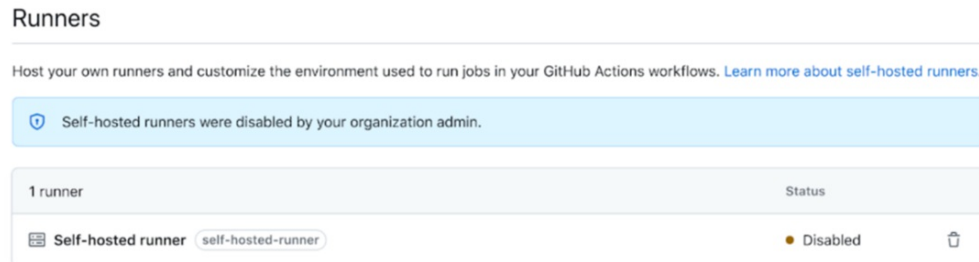
- ☐ **Disable for all organizations**
Repository-level runners will be disabled across all organizations in your Enterprise.
- ☐ **Disable for all Enterprise Managed User (EMU) repositories**
Repository runners will be disabled across all EMU personal namespaces in your organization.

Save

After disabling the creation of self-hosted runners, users will get the warning as shown in Figure 6.15: Self-hosted runner creation disabled. Any runners that have been created before these settings were enabled, will still be running, and executing jobs. You will need to check the organizations where you disallowed self-hosted runners and then remove the existing runners

manually. Do note that users can still create self-hosted runners for repositories created in their own user space.

Figure 6.15 Self-hosted runner creation disabled



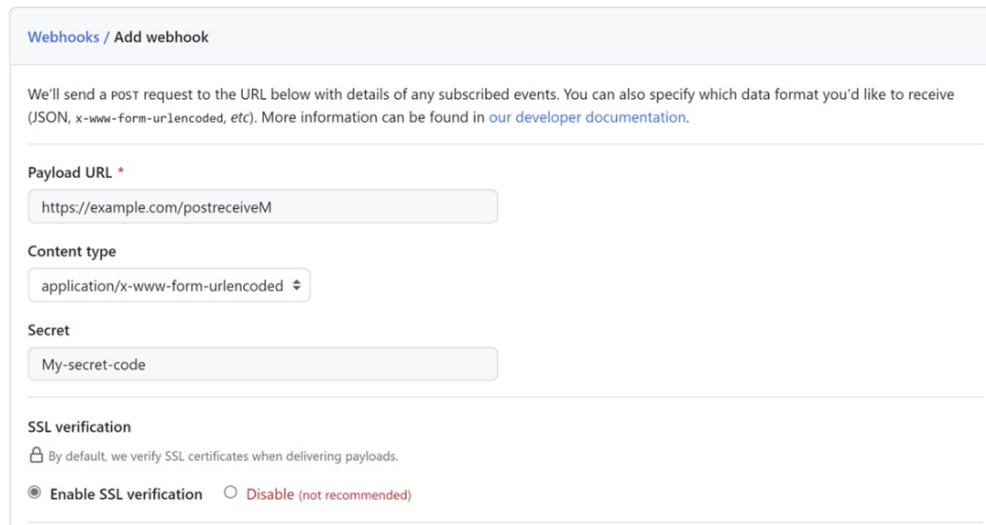
6.5 Autoscaling options

For setting up runners in an automated way, we recommend looking at the curated list of solutions in this repository:

<https://github.com/jonico/awesome-runners>. There are options to host runners on Amazon EC2 instances, AWS Lambdas, Kubernetes clusters, OpenShift, Azure VM's and you can of course setup an Azure scale set yourself as well. Some of the solutions will scale for you by itself, by using GitHub API endpoints to check for incoming jobs. Several solutions also support rules that let you scale up or down based on time of day (for example scale up between business hours, and down outside of business hours), or scale up and down based on the number/percentage of runners that are executing a job at the moment.

It's also possible to scale with a webhook in a GitHub App on the event **workflow_job**. This webhook is triggered every time a job is queued, waiting, in progress or completed. These events let you trigger the creation and deletion of a runner with for example the corresponding labels for that job. Using this webhook gives you full control over the runners that are available, including where to create them (for example in the correct network) or which hardware capabilities the runner will get. Setting up a webhook can be done at the organization or enterprise level, as shown in Figure 6.16: Scaling webhook setup.

Figure 6.16 Scaling webhook setup



The screenshot shows the 'Webhooks / Add webhook' interface. It includes a header with the title, a descriptive paragraph about sending POST requests, and four main input sections: 'Payload URL' with a text box containing 'https://example.com/postreceiveM', 'Content type' with a dropdown menu set to 'application/x-www-form-urlencoded', 'Secret' with a text box containing 'My-secret-code', and 'SSL verification' with radio buttons for 'Enable SSL verification' (selected) and 'Disable (not recommended)'.

6.5.1 Autoscaling with actions-runner-controller

The *actions-runner-controller* (ARC) solution is owned by GitHub and gives you an option to host a scalable runner setup inside your own Kubernetes cluster (a setup where multiple computers share the workload and scheduling is handled for you). If you have the option to host your own Kubernetes cluster for this and be in control how the cluster is utilized and scales, then we recommend this solution over others. Do note that ARC only supports Linux based Kubernetes nodes, so there is no option to run with Windows based nodes in your cluster. With ARC you get control over the Docker image that is executed so you can configure extra tools that are preinstalled by adding it to the container you configure. You also have control over the available hardware resources that the runner has by configuring the resource limits on the pod deployments. Since you manage and maintain the Kubernetes cluster, you have also control over what the runners can connect to, so you can really limit access to the internet for example, something that some enterprises do require. ARC runners are setup as ephemeral runners by default: the container will execute one job and exit the container. As it is using Kubernetes replica sets, Kubernetes will spin up a new container automatically.

Within ARC there are options to:

- Scale up and down based on a schedule.
- Scale based on percentage of runners that are busy executing a job (and then scale up / down by a configurable number of runners).
- Spin up new runners when on-demand, by listening to the API webhook that GitHub will trigger when a new pull-request is created.

The ARC solution supports creating runners at the Enterprise, Organization, as well as the Repository level, giving you the most flexibility of creating shared runners. You can also configure a scale set for Team A, and another one (with different scaling rules and a different even container image!) for Team B. By using a Helm chart to configure the scale set, you can let the configuration land in different Kubernetes namespaces, to give you more separation between them, as well as options for networking and limit access across namespaces.

Note that the ARC solution will spin up ephemeral runners, so any caching you want to do in the runners, will have to be done inside of the container images you use or rely on for example Kubernetes to cache the Docker containers you use. The images can be spun up using a rootless setup, making the setup a lot more secure (as breaking out of the container is harder when using rootless).

6.5.2 Communication in ARC

ARC lets you configure the communication with either a Personal Access Token (PAT) or a GitHub App. Since there is no GitHub App support to configure runners at the Enterprise level a PAT is required there. For all the other levels (Organization or Repository) we recommend using a GitHub App: that way you are not tied to a single user account and can really setup a fine grained App that can only be used to register runners, and nothing else. Usage of a PAT is discouraged as the PAT can impersonate everything the user can do instead of having fine grained control over the available scopes of the token. Additionally the engineer whose token is used, can leave the company and thus invalidate the PAT, leaving you with a broken setup that will take some time to figure out what happened. GitHub Apps also do not take up a license seat, saving on those costs as well.

With a GitHub App, you'll get an installation ID and a private key file (PEM file) that can be given to the ARC controller as a Kubernetes secret, that will be used to register and deregister runners. You can also use the GitHub App to be the receiving end for the webhooks available in GitHub to trigger a runner to be created whenever a job is queued.

Each time a new runner is requested, the GitHub App information will be used to get a fresh installation token from the GitHub API, and then the runner will be registered with that token.

6.5.3 ARC monitoring

There is very little monitoring for action runners in general, as you can see in **Chapter 7: Managing self-hosted runners**. The only user interface available is the one that shows you at each level which runners are available and if they are busy. Even the available API's are only showing that information: which runners are available, are they busy, and which labels are assigned. There is also no method to get that information out of ARC, like getting a count of available runners for a certain label, or get information about the percentage of runners that is busy at the moment. For monitoring purposes you will need to set something up yourself. You can use Kubernetes monitoring to check how many pods are up and running, and link that with dynamic scaling settings to see how you are doing, and then configure alerts if you are scaling up or down relatively fast. Alternatively you can create a workflow and use an action from the marketplace, for example <https://github.com/devops-actions/load-runner-info> and use that to get all the information of available runners, and determine the amount of runners available for a certain label (and alert if the count is lower than expected), or check how many runners are busy executing a job or not.

6.6 Conclusion

In this chapter you have seen what self-hosted runners are and when to use them as well as security risks and the different setup options you have available.

- Self-hosted runners can be configured in any environment that supports

.NET core, git, and node.

- Installing Docker is optional, but needed for running actions that are based on a Docker image.
- The self-hosted runner communicates with an outbound HTTPS connection, which makes installation in your network easier and more secure.
- You have a lot of configuration options for the runners to customize what happens before and after a job.
- The best way to setup a runner is by configuring it as ephemeral. Then it will only run a single job and deregister itself and not accept any more jobs. That gives you the option to clean up the environment and prevent a lot of security risks.
- There are several autoscaling options available, and the one that is managed and supported by GitHub is the actions runner controller. This can scale based on time, runner utilization, and just in time by configuring a webhook in GitHub that triggers whenever a workflow job is queued.

7 Managing your self-hosted runners

This chapter covers

- Managing runner groups
- Monitoring your runners
- Finding runner utilization and capacity needs
- Internal billing for action usage

When you start creating your self-hosted runners you will have the need to find out how and when your runners are being utilized, by which repositories and teams. With that information you can then both scale the runners appropriately and guide your users into better patterns of using them. There are options to segment runners into groups and only allow a group to be used by specific repositories, for example by a single team.

7.1 Runner groups

With runner groups you can segment your runners into different clusters and manage access to the runners in the group with specific options. You can use runner groups for example to segment the runners for the repos of a specific team, and make sure that they always have a specific number of runners available. Or make sure that a group of runners with a certain capability (for example GPU enabled) are only available to certain repositories and thus users. You do not want to run simple linting jobs on those expensive runners, so you better make sure to separate these runners from the default runners that have the ‘self-hosted’ label!

Runner groups can only be created at the Enterprise or Organization level and not at the repository level. When you navigate in the organization to Settings à Actions à Runner groups, you’ll find the overview of all your runner groups as shown in Figure 7.1: Runner groups. On the enterprise level you can find

runner groups under Settings à Policies à Actions and then the Runner groups tab. By design, there is always a group called *default* where new runners get registered unless you indicate otherwise in the configuration process. New groups can only be created using either the user interface or by using the REST API as shown in Listing 7.1.

Listing 7.1 Creating a new runner group

```
curl -L \
  -X POST \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer <YOUR-TOKEN>" \
  -H "X-GitHub-API-Version: 2022-11-28" \

  https://api.github.com/orgs/ORG/actions/runner-groups \

  -d '{"name":"gpu-group",
      "visibility":"selected",
      "selected_repository_ids":[123,456],
      "restricted_to_workflows": true,
      "selected_workflows":
        ["<ORG-NAME>/<REPONAME>/.github/workflows/<WORKFLOW>.yaml"]
    }'
```

In the overview depicted in Figure 7.1:Runner groups you can see how many runners are in each group, as well as the overall settings per group. Creating or editing a specific group will bring you to the settings as shown in Figure 7.2: Changing a runner group. You can configure if the runners in the group are available to be used by all repositories (or all organizations on the Enterprise level) or only a select subset of them. There is also an option to specify if the group can be used by public repositories or not. In Chapter 6: self-hosted runners we have shown the security implications of self-hosted runners. Especially for the use of self-hosted runners on public repositories it is crucial that you have a secure setup and don't let anyone create pull requests against your public repository that will directly run against your self-hosted runner! That is why this setting is not enabled by default.

Figure 7.1 Runner groups

Runner groups

Control access to your runners by specifying the repositories that are able to use your shared organization runners.

Group	Runners	
Default ⓘ All repositories, excluding public repositories	0	
BIG All repositories, excluding public repositories	1	...
Larger-runners-group Selected repositories (2), excluding public repositories	1	...
Test Selected repositories (1), excluding public repositories	0	...

Figure 7.2 Changing a runner group

Runner groups / BIG Remove group

Group name

BIG Save

Repository access

Selected repositories ▾ 4 selected repositories ⓘ

☐ Allow public repositories
Runners can be used by public repositories. Allowing self-hosted runners on public repositories and allowing workflows on public forks introduces a significant security risk. [Learn more about self-hosted runners.](#)

Workflow access

Control how these runners are used by restricting them to specific workflows. [Learn more about managing runner groups.](#)

All workflows ▾

Search runners

New runner ▾

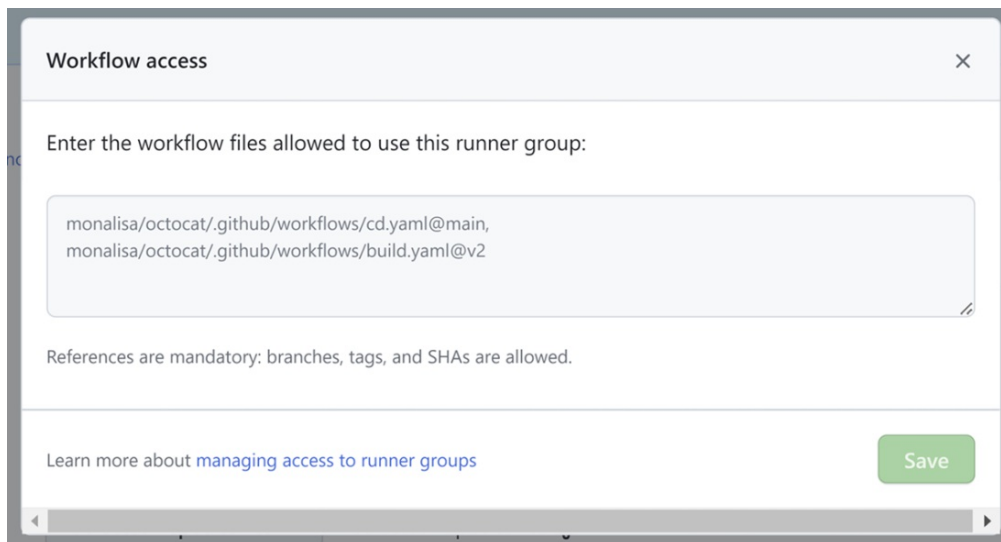
Runners	Status
test-16 test-16 BIG Runner group: BIG Public IP: Disabled	● Ready ...

You can even go a step further and configure the runner group to only be used for specific workflows, as shown below in Figure 7.3. This can be helpful if you have for example a runner with a GPU enabled, but you do not want every workflow in a repo to be able to run on that runner, as that could be a waste of resources. There can also be security reasons for separating your runners like this. You can configure one or more workflows that are allowed to use the runners in a group. Adding a specific reference to the workflow is required and has to be in the form of `<organization>/<repository>/.github/workflows/<filename>@<reference>`. Wildcards are not allowed. The reference can be any valid git reference, so

the name of a branch or tag will work, as well as a SHA hash of a commit.

Locking down a runner group to a workflow can be ideal for spinning up a runner on-demand by listening to a webhook. To achieve that, configure the group for a specific workflow and a specific revision, which will make that this run (and only this run) lands on the newly created runner. Configuration of the webhook has been shown in Chapter 6: Self-hosted runners. From automation in the webhook you can create a runner group on demand and lock it down to the workflow that triggered the runner creation as shown in Listing 7.1. Then create a new runner inside of the newly created runner group, which can now only be used by the correct workflow.

Figure 7.3 Locking a runner group to a specific workflow



Workflow access

Enter the workflow files allowed to use this runner group:

monalisa/octocat/.github/workflows/cd.yaml@main,
monalisa/octocat/.github/workflows/build.yaml@v2

References are mandatory: branches, tags, and SHAs are allowed.

Learn more about [managing access to runner groups](#)

Save

Do note that it is not possible to lock down a runner group directly to a specific team. You can only do that on the repository level, by configuring that the repository is allowed to use the runners in the group.

7.1.1 Assigning a runner to a runner group

The group a runner is part of will be configured by default on the creation of the runner. If you do not configure it, the runner will be added to the group named *Default*, which can be used by any repo at the level where the group exists (enterprise or organization). Listing 7.2 shows an example of configuring the runner group in the config script by passing in the name of

the group. A runner can only be assigned to a single group at the same time.

Listing 7.2 Adding a runner to a group during configuration

```
./config.sh --url <url> --token <token> --runnergroup <name of
```

When the runner has been created, you can still move it to another runner group by either using the REST API, or use the web interface as shown in 7.4: Moving the runner to a different group. The runner does not even need to be online to be able to move it. The runner will have the security setup immediately after saving the changes and can then be used from the repositories that have access to that group. Any running jobs will finish first with the security rules for the runners when the job started.

Figure 7.4 Moving the runner to a different group

Runners / Rob-XPS9700

Configuration: linux x64

Runner group: GPU enabled ▼

Move runner to group

×

Move runner to group

Default
BIG
GPU enabled
Larger-runners-group
Test
vnet-injected-runners-group

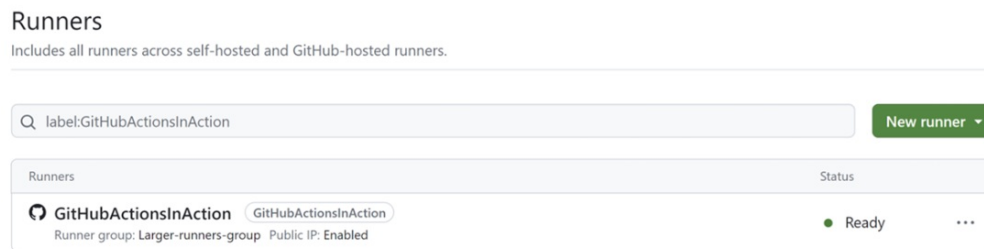
7.2 Monitoring your runners

You can view the available runners on the organization or repository level by going into 'Settings' à 'Actions' à 'Runners' or use the 'Runner groups' entry in the same menu. For the enterprise level runners, you can go to 'Enterprise settings' à 'Policies' à 'Actions' and then open the Runner tab or use the Runner groups tab. In the runner overview you find all runners that have been registered successfully with GitHub together with their status. A runner can be in one of four states here:

- Idle: Online and waiting for a job to execute.
- Active: Executing a job.
- Offline: no communication with the server, runner could be offline or updating to a newer version of the service.
- Ready: This state is used for GitHub hosted runners and indicates that there is no running online at the moment, but the setup is ready to spin up a runner on demand.

In the runner overview you can search for runners with a certain name, or use the search query as shown in Figure 7.5: Checking runner status to search for all runners that have a specific label.

Figure 7.5 Checking runner status



Searching can only be done on the part of the runner's name, or by specifying one or more labels to search on: `team-a label:linux label:x1`. An example of this is shown in Listing 7.3, where we search for a runner that has a name that has 'team-a' in the name and has both the labels mentioned in the search. Note that this search query is case-insensitive, and spaces have a meaning to break between search commands. Searching with for example wildcards on the name is not supported. Searching for a part of the label is also not supported.

The runner group overview gives an overview of the number of runners in that group as well as the security settings on the group, but does not give any indication of the status of the runners in the group. The search on this screen only allows you to search on a part of the name of the group.

Figure 7.6 Runner groups overview

Runner groups

Control access to your runners by specifying the repositories that are able to use your shared organization runners.

Q Search runner groups		New runner group
Group	Runners	
Default ⓘ All repositories, excluding public repositories	0	
Big runners Selected repositories (4), excluding public repositories	1	...
GPU enabled Selected repositories (0), excluding public repositories	1	...
Larger-runners-group Selected repositories (2), excluding public repositories	1	...
vnet-injected-runners-group Selected repositories (1), excluding public repositories	1	...

That means for monitoring of uptime and utilization you will need to implement your own solution.

7.2.1 What to monitor

What you want to monitor is dependent on the type of runners and the setup you have chosen. With for example actions-runner-controller, the autoscaling solution from GitHub as shown in Chapter 6, you need to monitor two important metrics:

1. Queue time of the jobs
2. Triggering of scaling up and down

If you have a solution of spinning runners up on demand, then the queue time of the jobs is the most important metric to keep track of. This will indicate if your runners are spinning up fast enough to prevent your users from waiting until their job is being started. Especially on *bursty* workloads (large amounts of jobs being queued at the same time), queue time can start to become longer rather quickly if your runners cannot spin up fast enough. Scaling down to fast is also not a great option, as that will potentially create a loop between scaling up and down constantly.

Keeping track of the number of concurrent jobs being executed is interesting from the perspective of knowing how many jobs, and therefor runners, you need at normal times, but be aware that the queuing of jobs can be very spikey, depending on your users. There are always user groups that have

nightly jobs scheduled, and other groups that schedule those jobs at the beginning of their workday. Depending on how geographically your user base is spread out, this can easily mean a big spike in the middle of the afternoon or evening. Your scaling or just in time solution needs to be able to handle these spikes gracefully, without scaling out of control just for that single user that is trying out the matrix strategy in their workflow for the first time and running it at maximum scale (256 jobs in one matrix) and scheduling those runs every 5 minutes. This can create some serious load on your runner setup (as well as the GitHub environment) and the question will be if this single spike will mean that all users have to wait for the queue to clear up, or if your solution is set up to handle these use cases.

Staying with the example of the recommended scaling solution of actions-runner-controller (ARC), you'll probably want to either configure this with the job queued webhook and spin up a runner on demand, or work with the deployment setup where you configure that you always have a certain amount of runners available, and let ARC handle the scaling up and down in case of need. In the second example, ARC will monitor your runners and check every period the amount of runners that are busy, and based on configurable rules will for example scale up if over the period of the last 10 minutes, 70 percent of the runners were busy executing a job. You can then indicate to scale up by a percentage of new runners. This can also mean that scaling up for a bursty load can take quite some time! Take the example that you have 50 runners available at any given time as a minimum. You have a rule that looks at 70% of runners busy that gets evaluated every 10 minutes. If the 70% busy threshold is reached, you scale up by 25% of runners. With this setup, one or more users schedule 100 jobs that take a while to run, let's say an hour. Scaling will happen after the first 10 minutes, where 25% times 50 runners = 12 new runners will be started. All existing and new runners are immediately busy executing jobs. It takes another 10 minutes to scale again. The rest of the example can be found in Table 7.1 scaling out runners. You can see that it takes 40 minutes with this setup to scale to a burst of new jobs getting queued, which are more than the runners you had available. It's up to you to define the needs of the organization, which can only be done by monitoring the use of your runners.

Table 7.1 Scaling out runners

Duration (mins)	Action	Number of runners	Jobs queued	Jobs running	Percentage busy
0	100 jobs get queued	50	50	50	100%
10	Scale out by 25%	62	38	62	100%
20	Scale out by 25%	77	23	77	100%
30	Scale out by 25%	96	4	96	100%
40	Scale out by 25%	120	0	100	83%

Depending on the time it takes to spin up a new runner, you can define a different strategy of scaling as well. If spinning up a runner is rather fast (less than a minute), then your users can most probably live with that delay. In that case it is advisable to work with the webhook and spin up runners on demand: every time a job is queued, a new runner is created. Spin them up as ephemeral and remove them on completion of the job. You can still have a pool of runners available on standby, and create new runners as the jobs come in: that way you can skip any larger startup time.

Another strategy for scaling is time based: if your users need the runners mostly during office hours, then you can spin up and down based on that. Create 100 runners at the start of the day and scale down at the end of the day. These strategies can be combined when using a solution like ARC by configuring multiple scaling rules.

7.2.2 Monitoring available runners using GitHub Actions

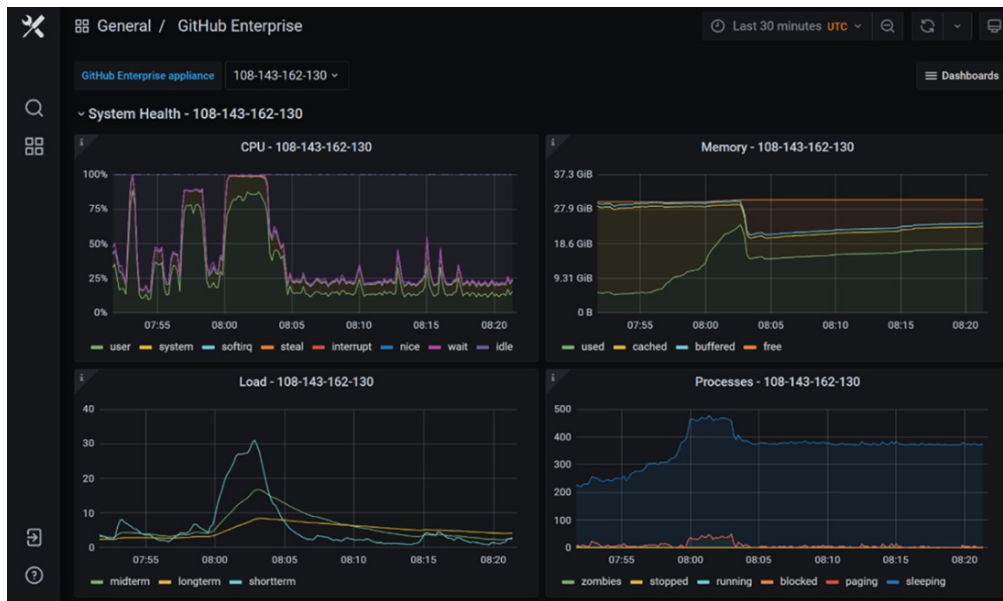
GitHub Actions is not meant for any sort of monitoring, as there are no guarantees that events are triggered immediately, or that cron schedules are followed on the second. There can always be some lag in triggering a workflow or a job. That said, since there are no out of the box solutions available from GitHub, you could utilize a workflow that runs and checks if the expected amount of runners is connected. If the amount of runners is less than a predefined number, you can trigger an alert into your tool of choice (for example Slack, Teams, or something else). One example is using the free <https://github.com/devops-actions/load-runner-info> action to get information about the amount of runners available. This action will for example give you a number of runners available per label. This can then be combined with your

own rules and your own notification channel to trigger an update to your team. An entire workflow example can be found in the readme of the action itself. The downside here is the information can only be loaded on a recurring schedule and cannot be retrieved real-time. Using this option can at least be the starting point for getting some insights into how your runners are being used, but not for scaling the runner setup on the fly.

7.2.3 Building a custom solution

Another option is to look at the free `github-actions-exporter` project and export the usage of actions on a schedule from the GitHub API into a monitoring solution of your choice: <https://github.com/Spendesk/github-actions-exporter> by using the OpenTelemetry output from the exporter. It can be used by default to export into Prometheus for example. Although the solution has not been touched and updated for a while, the basic premise and setup is still valid. After exporting the data you need into a type of storage, you can create your own dashboards, queries, and alerts. This will give you full control over the solution, but can take quite some time to have a working solution. You can think of tools like Grafana or Prometheus to build your own dashboards and alerts on top of the exported data. The downside here is again that the results will not be available real-time, only after the fact when you run a download cycle. The Prometheus setup does this every 30 seconds by default, which can cause some rate limiting issues. This method can still be very useful to get insights into the usage patterns of your runners. An example of a Grafana dashboard can be found in Figure 7.7 Example of a Grafana dashboard.

Figure 7.7 Example of a Grafana dashboard



7.2.4 Using a monitoring solution

There are several monitoring solutions available that integrate with GitHub Actions. DataDog has a paid GitHub integration that will pull information from the GitHub API and give you insights into your GitHub workflows, indicating how long a workflow run took, as well as the individual jobs and steps. For more information see <https://www.datadoghq.com/blog/datadog-github-actions-ci-visibility/>. One important metric it will show you is the queue time of the jobs. The DataDog integration does not retrieve any metrics on the runner level at the moment, like for example how many runners are available / busy at a point in time. The recommendation is to look at the queue time of your jobs, which is included in the DataDog integration.

This solution is also running on a cron schedule to retrieve the information using the GitHub API and will not give you real-time information. You can still learn a great deal of information from the usage patterns of your runners from this setup. This is very helpful when you get started running GitHub Action workflows at scale.

Alternatively, you can use a webhook at the organization or enterprise level to send notifications of jobs getting queued, starting and completing into a monitoring solution of your choice. This is the best solution for real-time information being available. An example of the hook configuration can be

seen in Figure 7.8. The webhook can be sent anywhere, as long as GitHub can reach that url. The payload of the webhook can be ingested by an application like Azure Log Analytics, Splunk, or any other tool that can visualize the JSON formatted data being send in. There is a Splunk App available that, amongst other visualizations, gives you information about the amount of workflows being triggered, as well as the job outcomes and duration. You can find more information on the app here: <https://splunkbase.splunk.com/app/5596>. The benefit of using the app is that the queries have been prewritten and can give you a first overview quite fast. The downside is that the out-of-the-box dashboards don't go far enough to properly manage your self-hosted runners. It does not show queue times for example. Adding your own custom dashboards on the data is straightforward if you are familiar with Splunk. The data that is used and the initial queries can be taken from the existing dashboards and can then be the base of your custom queries and alerts.

Figure 7.8 Configure a webhook to send information about jobs starting

The screenshot shows the 'Webhooks / Add webhook' page in GitHub. It includes a header with the breadcrumb 'Webhooks / Add webhook'. Below this is a descriptive paragraph: 'We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in our developer documentation.' The form contains several fields: 'Payload URL' with the value 'https://example.com/postreceive', 'Content type' set to 'application/x-www-form-urlencoded', and an empty 'Secret' field. A section titled 'Which events would you like to trigger this webhook?' offers three radio button options: 'Just the push event.', 'Send me everything.', and 'Let me select individual events.' (which is selected). Under the selected option, there are two checkboxes: 'Workflow jobs' (selected) and 'Workflow runs'. At the bottom, there is an 'Active' checkbox (checked) and a green 'Add webhook' button.

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in our developer documentation.

Payload URL *

https://example.com/postreceive

Content type

application/x-www-form-urlencoded

Secret

Which events would you like to trigger this webhook?

☐ Just the push event.

☐ Send me everything.

☒ Let me select individual events.

☒ **Workflow jobs**
Workflow job queued, waiting, in progress, or completed on a repository.

☐ **Workflow runs**
Workflow run requested or completed on a repository.

☒ **Active**
We will deliver event details when this hook is triggered.

Add webhook

7.3 Runner utilization and capacity needs

When you start creating your own runners the need for defining the capabilities for them will start to arise. Often, we see that people start with rather simple runners: maybe a dual core processor and 2GB of RAM. This is fine for most normal workflows, where you lint code, or build an application. For some projects, these hardware specs are not enough to complete your workload within a reasonable amount of time. If you are using modern working practices like CI/CD (discussed in the next chapters), you want your build validation to happen as fast as possible, so that the developers get fast feedback. If they have to wait long for a build to complete, they will start doing something else, which comes with the cost of context switching. Most of the time you can shorten the time the developer has to wait by adding more hardware capacity: give the runner more RAM or more CPU cores (or both). This can significantly speed up build times and shorten the feedback cycle for the developers. An example of making a multi-hour workflow job complete faster and the hidden developer costs for it, can be found in this post from GitHub: <https://github.blog/2022-12-08-experiment-the-hidden-costs-of-waiting-on-slow-build-times/>.

There is no golden rule available for finding out how much compute power a workflow job needs. You can monitor your runner environments for their utilization, which will give you a hint if adding more power will be of any help or not. If the entire job only uses 50% of your compute, then adding more resources will probably not have any impact. But if the usage spikes close to 90% utilization, then it might be worthwhile to try out a bigger runner.

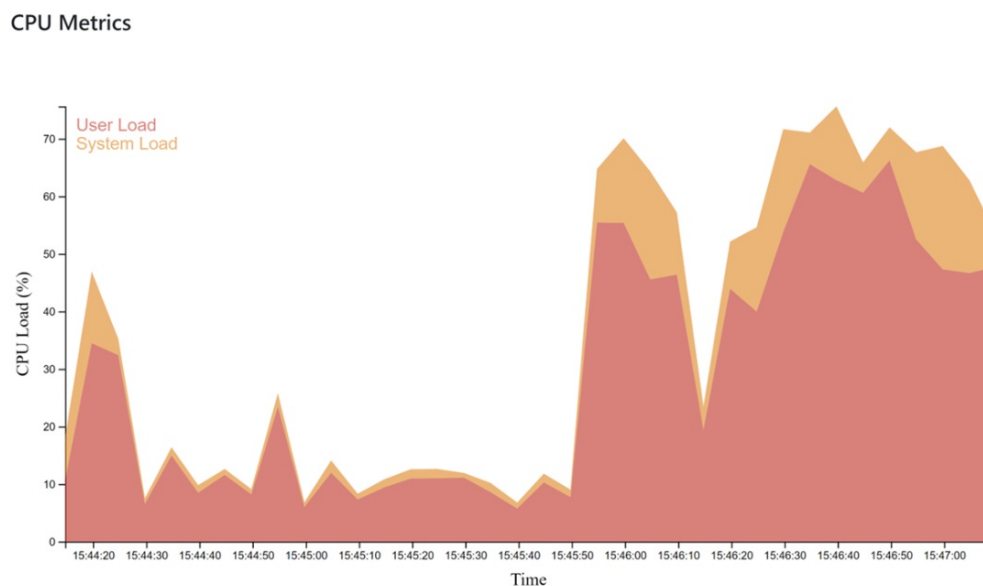
The same goes for jobs that execute on a runner with way too much power: running them on a smaller runner will probably take almost as much time, but frees up the larger runner for other workloads. It makes no sense to run a code linting job that takes 30 seconds to run, execute on a big 64 core runner with 32GB of RAM: that machine can probably be used more effectively.

Monitoring can be done by using your normal monitoring solutions in the form of agents that are installed on the runners and they sent data to your central monitoring server for reviewing after the fact. Depending on the

monitoring solution you can add additional data fields like the name of the runner, repository name, and workflow. With this information you can correlate the runner utilization to the workflow job that was executed.

Another option is to point your users to the telemetry action and use it in their jobs: <https://github.com/runforesight/workflow-telemetry-action>. The action will start logging information about step duration, CPU, RAM, disk IO, and network IO. At the completion of the job the information will be shown in MermaidJS charts in your workflow summary. An example of the CPU Metrics can be seen in Figure 7.9 CPU utilization of the runner. This action uses tracing of the metrics through NodeJS and therefor works across Ubuntu, Windows, and macOS based runners. It does not work on container-based jobs though.

Figure 7.9 CPU utilization of the runner



7.4 Monitoring network access

You need to be aware of what the runners are doing inside of the environment you have setup for them. By default, the runners need access to the internet to be able to download and run actions. If the action is based on a container, that image will need to be downloaded as well. As most container actions from the public marketplace use a local Dockerfile, this will need to

be built at runtime as well, with all the dependencies it needs. The default setup of the runner also includes an auto-update mechanism, so the runner will need internet access for that as well. If you are running GitHub Actions against Enterprise Server, the runner updates will be downloaded from the server itself.

The main reason to look at outgoing connections is for security purposes. You want to be aware of what actions and scripts are doing on your runners and see if that matches your expectations. For example: why would an action that is intended to lint your code for guidelines, need to connect to a third-party API endpoint? It would be weird if it did, as that does not match the expectations of a linter. As an action is built on top of an ecosystem like npm, attack vectors for the action are numerous. Therefore you need a way to monitor and limit networking access on top of vetting the actions before they are used by your end-users.

7.4.1 Monitor and limit network access

The runner service itself has no options for monitoring or limiting network access. The whole setup assumes internet access is available, and that the runner can always download the action repositories and all the necessary dependencies. That means you will need to setup your own monitoring solution. The options for this depend heavily on the platform and setup that is chosen. If you run the runner on a Virtual Machine in a cloud environment, you can setup networking monitoring and rules on that level. This will give you some insights, but stopping outgoing connections can become more cumbersome as the usage of your runners increases. Segmenting your runners into different networking segments can be done by deploying them differently and giving the runners labels that match the networking capabilities. You can also use the runner groups to configure them for certain repositories with only the access those repos need for their type of workloads.

Additionally, there are solutions of vendors like StepSecurity (<https://www.stepsecurity.io>) that can help you with monitoring the outgoing connections from your runners by installing an agent at runtime. That agent is called 'harden-runner' and is free to use for public repositories. For private repositories it is a paid product. The harden runner starts with an initial

testing phase to gather the connections being made by a job and logging those connections to the SaaS service of the product. After knowing and analyzing the connections that are made, you can then add an allow list to the workflow and lock down the connections it can make. The solution from StepSecurity works by leveraging a custom Linux DNS setup and needs sudo rights. That means it does not work on macOS or Windows runners. Container support is also not present at the time of writing. There is also support for the ARC setup where they leverage tooling on the Kubernetes cluster level, so that not every workflow needs to install the harden runner by itself. This greatly improves the usability for the end users. For the ARC support you do need to have a paid license.

An example of configuring the harden-runner action to analyze the outgoing network connections being made from the job can be found in Listing 7.3. From running this workflow you will learn that the setup-terraform action will download the binaries from <https://releases.hashicorp.com>, which is expected. Next to that you will also learn that running terraform version also makes an outgoing connection to <https://checkpoint-api.hashicorp.com>, as it is also checking if there is a newer version to download and log a warning if that is the case. The harden runner setup can then give you fine grained control over the connections you want to allow to be made. Shown in Listing 7.4 is an example where all outgoing connections will be blocked (and logged), except for the endpoints in the allow-list. The code used for the agent is written in Go and open sourced at <https://github.com/step-security/agent>.

Listing 7.3 Configure harden-runner

```
name: harden runner demo
on:
  workflow_dispatch:

jobs:
  demo:
    runs-on: ubuntu-latest
    steps:
      - name: Harden Runner
        uses: step-security/harden-runner@v2.1.0
        with:
          egress-policy: audit # TODO: change to 'egress-policy:
```

- uses: actions/checkout@v3
- uses: hashicorp/setup-terraform@v2
- run: terraform version

Listing 7.4 Harden-runner with block policy

```
- name: Harden Runner
  uses: step-security/harden-runner@v2.1.0
  with:
    egress-policy: block
    allowed-endpoints: >
      api.nuget.org:443
      github.com:443
```

You can also use your own networking setup to limit the outgoing connections from your runners. If you are using actions-runner-controller on Kubernetes as described in Chapter 6, then it is possible to use egress control using network policies around your runners to allow or deny certain traffic to connect to the internet, or limit it to certain endpoints. Tools to look at for this are for example Cilium and Calico.

If you host your runners in your own networking setup, it is possible to segment the networks for the runners, and only configure certain endpoints to be used. Having a pool of runners ready for each type, will create some overhead as you need to have a warm pool of runners available for each group. Next to that you need to handle scaling up and down for each pool yourself.

7.4.2 Recommended setup

There is a tradeoff between being very restrictive for your runners and what they are capable of doing in terms of connecting to external endpoints then your GitHub environment. Connections back to GitHub have to be made in any case, and next to that your users will want to use GitHub Actions and download them from a marketplace.

Our recommendation is to use a declarative style in your workflows like for

example StepSecurity uses, and have the users specifically configure to which endpoints they need to make connections to. This will prevent leaking out data to third party endpoints without being aware of it. With the block policy from StepSecurity, any extra connection that is made will be blocked initially and it will be logged centrally for your security team to keep track of new connections being requested. This will greatly improve your runners and workflows security!

7.5 Internal billing for action usage

Self-hosted runners come with setup costs, hosting costs, as well as maintenance costs. Even if you use them on GitHub Enterprise Cloud, the usage for self-hosted runners is not included in the usage reports. It can be very helpful to show teams how they have been using the runners over time and make them more aware of the costs of having them online all the time. Those costs can be split between hosting the machines as well as the amount of energy used and thus CO2 they generate. Both aspects can be used to let user think if they really need to use 5 jobs in parallel, or if it would be better to run the same steps in sequence (and use less concurrent machines doing so).

For the usage aspects you can either use the information already available in your monitoring tool (e.g., Splunk) and separate the information out per repository or team. If you don't have a monitoring tool in place, you can also use the <https://github.com/self-actuated/actions-usage> tool. This uses the GitHub Api to get the actions usage information per workflow, and get an overview like the example in Figure 7.10 Action minutes overview report. Most tools only call the GitHub API on the workflow level and calculate the duration of the entire workflow. It is possible to do the same on the job level, but that will not include extra information (like the used label for the job). That is why most tools do not make the extra API calls to load that information as well. This also means that it is harder to make the split between GitHub hosted and self-hosted runners, if you mix these in the same repository or workflow! You could take the extra step of getting the information on the job level, as that will include the commit SHA of the workflow definition. You can then download that version of the workflow and parse the definition yourself.

Figure 7.10 Action minutes overview report

Repo	Builds	Success	Failure	Cancelled	Skipped	Total	Average	Longest
devops-actions/load-available-actions	1961	1449	85	148	279	69h57m21s	2m8s	56m16s
devops-actions/load-runner-info	1411	1287	4	0	120	6h4m13s	15s	4m7s
devops-actions/issue-comment-tag	1404	1318	9	0	77	7h11m33s	18s	2m29s
devops-actions/json-to-file	837	829	7	0	1	3h12m38s	14s	2m18s
devops-actions/variable-substitution	668	624	29	5	10	11h10m36s	1m0s	54m21s
devops-actions/actionlint	392	297	69	0	26	57m58s	9s	1m43s
devops-actions/load-used-actions	353	312	40	0	1	1h20m45s	14s	56s
devops-actions/action-get-tag	256	228	27	0	1	1h9m32s	16s	2m4s
devops-actions/actionlint-testing-repo	179	132	47	0	0	8m23s	3s	16s
devops-actions/.github	79	67	11	1	0	9m28s	7s	3m12s
devops-actions/action-template	35	33	2	0	0	6m51s	12s	36s
devops-actions/actionlint-testing-repo2	26	14	12	0	0	1m16s	3s	7s
devops-actions/test-repo	25	25	0	0	0	39s	2s	7s
devops-actions/demo-actions	24	0	9	0	15	54s	2s	11s
devops-actions/docker-action-demo2	15	15	0	0	0	25s	2s	6s
devops-actions/docker-action-demo	15	15	0	0	0	24s	2s	7s
devops-actions/test-repo-bla	5	5	0	0	0	8s	2s	5s
devops-actions/docker-action-demo3	5	5	0	0	0	7s	1s	5s
Total usage: 101h33m11s (6093 mins)								

When you have the action minutes used per repository, you can calculate the price of the runs by multiplying the minutes with a predefined cost. Combine that with the used network traffic and you have a more complete picture of all the things the users are doing in their repositories and workflows.

Rolling up the repositories can be done to a team level or any other level if you add topics to the repository and use that for slicing the information into groups. Showing this information in for example a monthly report or in a dashboard, can help the users to become more aware of what they are actually doing in their workflows. We have seen examples where a repository of 300MB was cloned 4 times in the same workflow file, to get seven lines of shell script and execute it. The seven lines of script where in total 11 bytes. By doing a shallow clone of only the script, we saved 1.2GB of network traffic in every single workflow run (this workflow was used everywhere!).

7.6 Summary

In this chapter we have seen:

- How to monitor your runners for availability and how they are being used.
- Using the right size of runners and how to report back this information to your end users.
- Runners can be configured to only be used by certain repositories, by placing them into runner groups.
- Runners can be moved between runner groups, but can only be part of a single group.

- Monitoring your self-hosted runners is important to check if there are enough runners available for your users.
- Even by using scaling solutions, you still need to monitor for scaling actions, to see if you're not scaling in and out constantly, or if you are scaling up fast enough.
- Getting information out of GitHub on how your repositories are using the runners is not available out of the box. Existing open source solutions have their pro's and con's. They can be used to get started loading the information, but more specific information like runner labels can be needed to have a full overview.
- Make the use of reporting back the usage information to your users to make them think about how they are using your runners: should they really clone the repo every time for running a simple script, or can this be done more intelligently?

8 Continuous Integration (CI)

This chapter covers

- Achieving fast feedback with Continuous Integration (CI)
- Differentiating different types of integration workflows
- Defining the generic set of steps of every CI workflow
- Ensuring the integrity of the artifacts produced
- Creating a release that can trigger your Continuous Delivery (CD) workflows
- Setting up a CI workflow that prepares the deployment of a sample application

Continuous integration (CI) is a DevOps practice where you regularly merge code changes into the central repository and where automated builds and tests are run to check the correctness and quality of the code. CI aims to provide rapid feedback and identify and correct defects as soon as possible. CI relies on the source code version control system to trigger builds and tests at every commit.

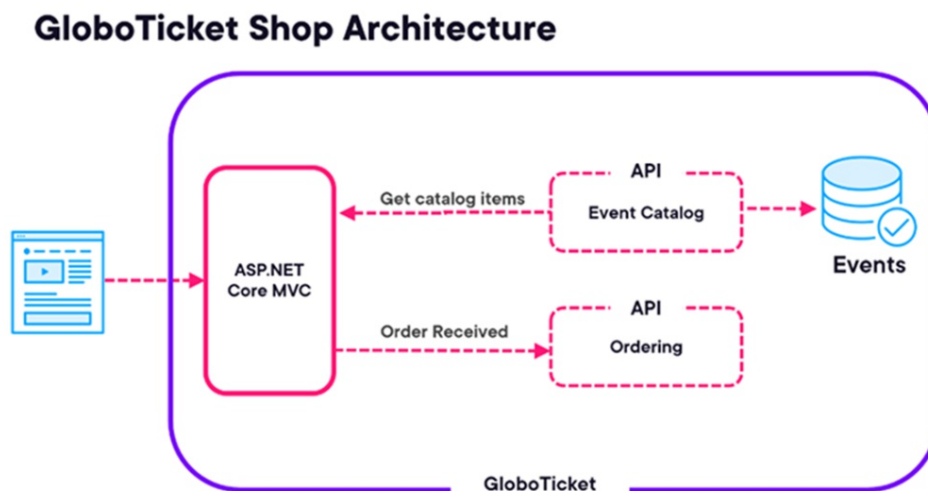
CI is comprised of a set of steps that delivers the output artifacts we need to run a system in production. Which set of steps are required depends on the programming language and tools you are using and the platforms you are targeting with your product. In this chapter, we will lay out the generic set of steps each Continuous Integration process typically entails and how you can setup GitHub actions to trigger on each commit and deliver this as a set of artifacts that can be picked up in a subsequent process of *Continuous Delivery (CD)*, where you deploy the product to preferably a production environment. CD is covered in Chapter 9, Continuous Deployment (CD).

8.1 GloboTicket, a Sample Application

In the next paragraphs, I will guide you on how you can build an application. We will cover how we will deploy this application to production in Chapter

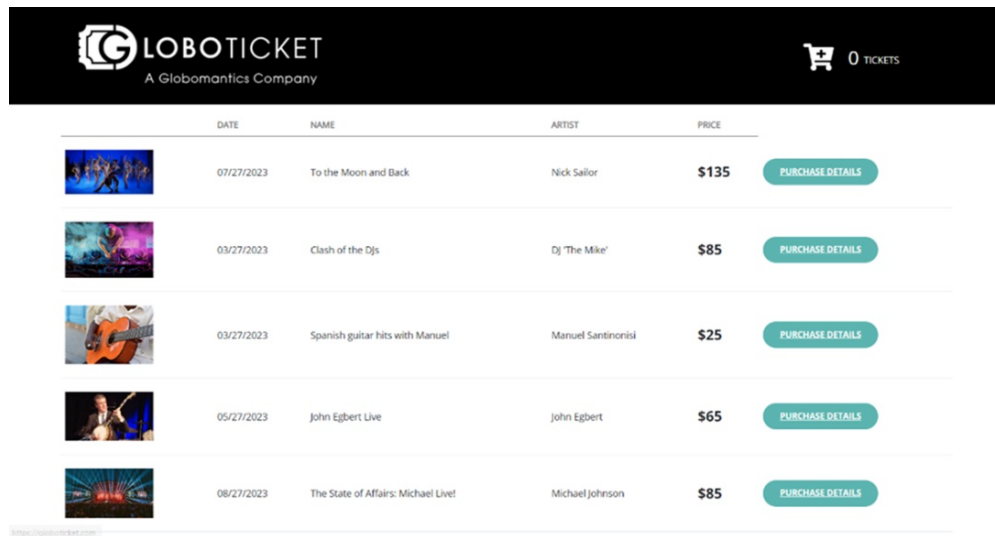
9. In order to give some real-world examples of how you can create a CI and CD workflow, we will use an application written in C# and deployed to the Azure Cloud. We picked a solution that can be deployed to a Kubernetes cluster since that is very common these days. Remember, the application is used to illustrate the concepts, and all steps and concepts we use to build and deploy an application to the cloud are applicable to any piece of software you have yourself. With GitHub actions, you can deploy any application to any infrastructure. The architectural diagram for the application is visualized in **Figure 8.1**. You see, we have a web application that shows the front end of the application, which is a web app. The application uses two API's. One is to retrieve the tickets that can be sold (Catalog), and the other is to register the orders that have been placed (ordering).

Figure 8.1 globoticket architecture.



The moment we deploy this application, you should see a website that shows you tickets you can buy to attend a concert. The deployed application is shown in Figure 8.2.

Figure 8.2 Globoticket home page



You can find the sample application yourself at the following location:
[https://github.com/ GitHubActionsInAction/Globoticket](https://github.com/GitHubActionsInAction/Globoticket)

The application is based on a microservices architecture and requires three containers to be deployed to a Kubernetes cluster.

8.2 Why Continuous Integration

The first mention of Continuous Integration dates to 1989 from a computer Software & Applications conference in Orlando. (Proceedings of the Thirteenth Annual International Computer Software & Applications Conference. Orlando, Florida. pp. 552–558.

doi:10.1109/CMPSAC.1989.65147) In the 90's software methodologies like Extreme programming also experimented with this concept. It really picked up popularity in the early 2000's right after the Agile Manifesto got traction. The agile manifesto is based on twelve principles, and the first principle states: *"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."*

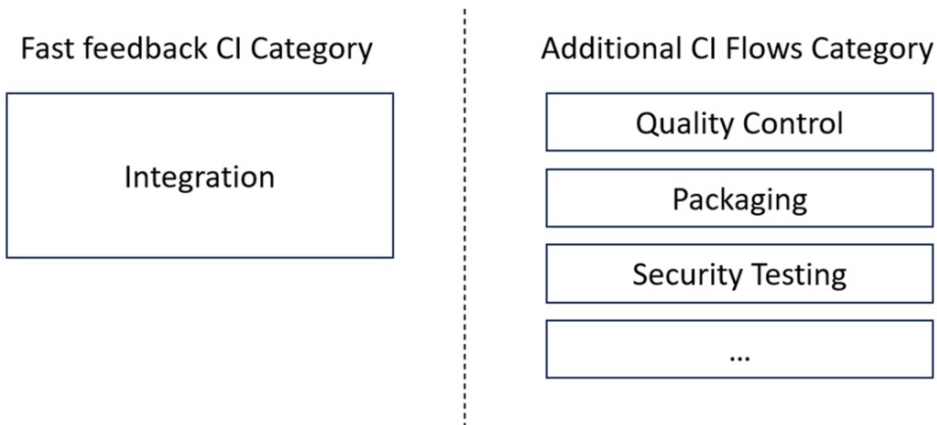
(<http://agilemanifesto.org/principles.html>) To get into the state of continuous delivery, we first need to get into a state where we ensure our codebase is always in a so-called "buildable state". Where in the past, we built our codebase infrequently and had to take a significant amount of time to integrate changes of many team members who committed changes over a period of time. When you implement CI you spend way less time on

integrating the software with the changes of others, and this ultimately reduces the waste that is spent in resolving code change conflicts. So, the bottom line is that we use CI to reduce waste in our software delivery process by integrating the software at each commit to the central repository instead of spending a lot of time fixing all integrations that would otherwise be accumulated over time.

8.3 Types of CI

Because we strive to integrate the software as soon as possible on each commit to the central repository, we also distinguish different types of CI. Each of these types of CI strives for different goals as part of the final goal of moving software to production as fast as possible. We can run these different types of CI in parallel with each other to produce results much faster and give the developer feedback as soon as possible. While working with many different companies, we see there is a common pattern in categories of workflows. First, you have the category that has the goal to create feedback on the integration as soon as possible. This is the primary reason for CI. The workflow needs to be as fast as possible. The second category has various goals that differ from this fast feedback CI. This can be the creation of packages for the final delivery of the software, it can be reporting on the quality of the software, it can be providing insights in security, etc. Because they have a different purpose, you can trigger them less frequently and can have slower response times regarding the feedback to the developers. This is represented in *Figure 8.3*.

Figure 8.3 Different types of CI

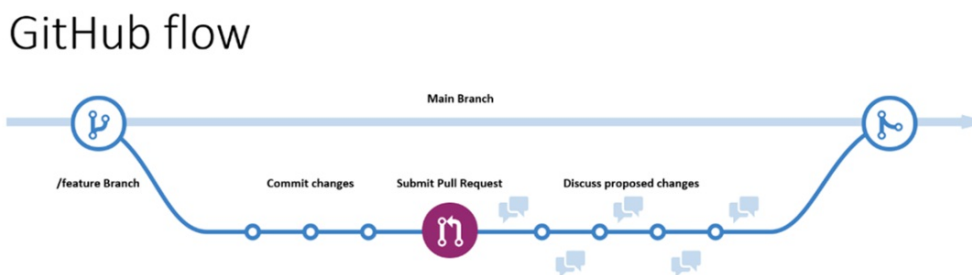


Let's have a deeper look at when we would trigger each type of workflow and what the goal of each type of workflow actually is. For this we will refer to the way we work with our code repository with a specific branch strategy. The strategy chosen is the most popular strategy that is known by the name GitHub Flow.

8.3.1 Branching strategy GitHub Flow

GitHub flow is the advertised way of working when you use git and deliver a software product the DevOps way. With GitHub Flow, you create a branch with the name feature/name-of-feature, and you commit your changes to that branch. When you think your feature is complete, you open a pull request where you solicit feedback and have the peer review of your code. After some discussion and final approval from your team members, the pull request is accepted, and the change is merged into the main branch and then deployed to production. This workflow is visualized in Figure 8.4:

Figure 8.4 Branching Strategy GitHub Flow



We can set up GitHub in such a way that you protect the main branch from any direct commits and that every change needs to come from a pull request. This way of working is encouraged since it gives a great way of controlling the quality of what goes into the main branch and you also enforce a four-eyes principle, which is needed in most compliance frameworks. Using this way of working also enables you to comply with regulations in industries with heavy governance. Enforcing this flow also enables you to set up action workflows that can receive very fast feedback on the work you do on the feature branch, and it ensures there is always a stable main branch that is always in a deployable state. In the remainder of this chapter, we describe what types of CI you can distinguish, and it can save you a lot of time and compute resources when you apply the various action workflows to particular steps in the GitHub Flow process. We assume GitHub Flow for the remainder of the book since it is the most used way of working nowadays. This way, all the examples we provide can be used immediately without many modifications.

8.3.2 CI for Integration

This CI process strives for validation if the software you just committed to the repo can be integrated into the source code. This entails compiling the code to the type of artifact you need for production. The integration CI strives to provide results as fast as possible, and you should strive for swift feedback to the developer. If failures occur, this implies the code is not integrated, and the developer typically takes action immediately to fix the integration problems. Errors that occur here are often of the type of compiler errors and warnings. This workflow is normally triggered on the feature branch in GitHub flow. The workflow on the main branch often entails more steps, to ensure complete validation of all we need before we want to deploy.

8.3.3 CI for Quality Control

This process validates the quality of the source code that was committed. This involves simple quality control checks like linting of the source code for readability, checking if the code has multiple duplications of the code, or if the code itself has not passed a set of maintainability metrics. There are also some more involved quality control checks, like validation if the code is

written securely and if the code delivers the functionality based on automated tests. In this process, you can include a variety of tools that will give you insights into the quality of the code currently in your source code repository. Tools you can think of that are typically part of a CI build for quality control are Linting Tools to check the syntax of the code against a set of rules, Code metric tools like SonarQube that provide insights into maintainability and other code smells, Unit testing tools that validate the overall functional workings of the code base, and security tools that can validate if your code might be vulnerable to all kinds of known ways to attack software. Typically, these builds take longer to complete and often result in work that is placed back on the backlog to fix in a later stage of the development process. This workflow is often triggered when you create a pull request, so it provides input for the reviewers and helps ensure quality in the main branch.

8.3.4 CI for Security Testing

This process is there to check if the software that is written is secure by default. It uses tools we know as Static Application Security Testing (SAST) and Dynamic Application Security Testing tools (DAST). GitHub itself also provides these tools as part of the tool suite and they are fully integrated with GitHub Actions and the user interface on the web. When you have GitHub Enterprise, you can buy the rights to use the tools as an add-on capability. This product is called Advanced Security, which is free for public repositories. With Advanced security, you can create a security testing workflow that does advanced scanning of the software on known vulnerabilities that might have been introduced in your own software. You can also, of course, use any other tools you can find in the market that can help you do security scanning on your software. Well-known vendors here are Snyk, Black Duck, and Mend. This CI type is also triggered at the moment of a pull request, so we ensure we don't bring new security vulnerabilities to the main branch. It also should be part of a regular schedule on the main branch since new vulnerabilities emerge in the software ecosystem without us needing to change our code. Having this on a schedule on the main branch ensures we always know the potential security issues that we ship to production. We also can decide to block releasing the software as part of this action workflow to ensure vulnerabilities of a certain severity level are always mitigated before release.

8.3.5 CI for Packaging

This process aims to produce the final artifacts to deliver the software to production. Here we can target multiple platforms and create builds that are optimized for production purposes. While previous builds can, for example, still include debug type of builds, these builds provide the clean artifacts we move to production. Removing debug information is often forgotten and can besides creating a larger size artifact also create a potential security risk. The end result of this build is that we get the final artifacts delivered to either an artifact store, e.g., the package management store, the container registry, or upload the artifacts to GitHub actions storage so that it can be picked up at a later moment by the Continuous Delivery actions workflow.

8.4 Generic CI Workflow Steps

Every CI Workflow has the same set of generic steps.

- Get the sources
- Building sources into artifacts and some initial checks that are very fast to execute.
- Publish results.

Let's have a look at these steps and see how we can optimize them for each type of CI.

8.4.1 Getting the Sources

You get the sources from your repo with the action *actions/checkout*. The action to get sources can also be tuned to what you actually need to get from the source repository. To speed up your workflows, it often makes a lot of sense not to fully clone the repository but get only the tip of the main branch. This can speed up the operation significantly, especially for repositories with a longer lifespan. You can also control which branches are retrieved and the depth of the repository you clone. To show an example, I created **listing 8.1**, where we only retrieve the tip of the main branch since this is often the only data you need to see, for example, if the source code compiles and integrates with what is in the current repository.

Listing 8.1 Checkout action

```
- uses: actions/checkout@v3
  with:
    ref: 'main' #not naming the ref will fetch the default branch
    fetch-depth: '1' #1 is default and 0 fetches the full depth o
```

If you want to get the repository with the full history, you can set the property `fetch-depth` to 0, this will get you the full history of the repo. This is only needed when you are going to traverse the history of the repo as part of the next steps in your CI. Sometimes other actions might need this, so it is good to know it is possible with only a simple change. The default for `fetch-depth` is 0.

8.4.2 Building the Sources Into Artifacts

Once you have the sources available, you can take steps to build the source into artifacts that you need to validate if the software is doing what it is supposed to do. A common practice is that you compile the sources into binary files or create container images that can be used to deploy the application.

We use the sample application to give you a concrete example of the next step in your workflow. This application first needs to be compiled, then we run the basic unit tests to validate the basic behavior of the application, after which we create container images that can be used for deployment.

Since the sample application is a .NET Core application, the step to compile the sources requires using a tool called the dotnet command line interface. Some tools are already installed on the GitHub hosted action runners. The dotnet tooling is a good example of this. To get a full list of the tools installed on the runners, visit the following location: <https://github.com/actions/runner-images>, as described in chapter 6. **Listing 8.2** shows how to compile the .NET code into binaries.

Listing 8.2 Compile .NET core code

```
- name: Setup .NET
  uses: actions/setup-dotnet@v3
```

```
with:
  dotnet-version: 6.0.x

- name: Restore dependencies
  run: dotnet restore

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

The code example in **Listing 8.3** only builds the code. The sample application will eventually run on a Kubernetes Infrastructure, so we must create a container image. Now, we can make a choice here always to build a container image, but this would significantly slow down the workflow compared to a simple build of the C# files in the project.

If we go back to the GitHub flow approach of branching, we can also divert this and make this part of the processing of the pull request. That way, you can suffice with only fast feedback if the sources are in good shape. The following workflow you need has a different purpose: to produce the required artifacts for us to deploy to an environment and validate if the code adheres to coding standards, license checks, etc., before it is accepted into the main branch. This is not needed for your feature branches, but only when you merge to the main branch. You can trigger the next workflow the moment you create the pull request.

Note

You might think this is a bad idea if your compile step takes a few hours because you are building a large code base. When Actions started, this was true, but nowadays, we have a caching option where we can decide what the cache key will be. When you share the key between jobs, you can use the cached artifacts and nicely separate the concerns of CI and the steps to create the final output for delivery. More on caching can be found in **Chapter 12**

8.4.3 Testing the Artifacts

The tests we run during the CI for Validation only involve tests that can show if the integration of the sources is successful. Preferably this would only involve the tests that can verify the impact of the change. Often this is not

easy to determine, and the most common tests you run in this step are the Unit Tests that are part of the sources you are building. In our case, this is .NET, and we can use the dotnet command line to kick off the tests. The result of the test run should indicate success or failure, which we can use in our other steps in the workflow as an indicator if we should continue our run. Any test tool you use can set the workflow state to failure by producing an exit status code other than 0.

For example, you can run the unit tests that are part of the sample application by running the dotnet command line `dotnet test`. It will produce an exit code `>0`, indicating the number of tests that have failed. If all tests pass, the command line will return 0, indicating success.

Listing 8.3 Use of command line to run tests

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

8.4.4 Test Result Reporting

By default, GitHub has no other way than the console output built-in reporting if it comes to test results. But especially when tests fail, you like to see a report of which test failed and which was successful. You can still get the data in the final workflow report by adding information to the job summary description. This is done by outputting data to the output variable available in your workflow run. It is called `$GITHUB_STEP_SUMMARY`.

You get things in the result summary by pushing any text in markdown format. This is then rendered in the output report of the job.

In **Listing 8.4** you can see an example of outputting text to this variable and the results that will be outputted to the job results page.

Listing 8.4 Using `$GITHUB_STEP_SUMMARY` to visualize test result output

```
name: "chapter 08: Generate job output using markdown"
on:
  workflow_dispatch:
jobs:
```

```

build:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout
      uses: actions/checkout@v3
    - name: Generate markdown
      run: |
        echo "## Test results" >> "$GITHUB_STEP_SUMMARY"
        echo "| **Test Name** | **Result**|" >> "$GITHUB_STEP_S
        echo "|--|--|" >> "$GITHUB_STEP_SUMMARY"
        echo "| validate numbers are > 0 |:white_check_mark: |"
        echo "| validate numbers are < 10 |:white_check_mark: |"
        echo "| validate numbers are odd |:x: |" >> "$GITHUB_ST
        echo "| validate numbers are even|:white_check_mark: |"

```

This will result in a summary that contains a nicely formatted table with the results. This is shown in **Figure 8.5**.

Figure 8.5 Summary results in markdown

The screenshot shows the GitHub Actions interface for a workflow named 'Generate-job-output-using-markdown.yml'. The 'build' job is highlighted as completed successfully. The summary section displays the following test results table:

Test Name	Result
validate numbers are > 0	✓
validate numbers are < 10	✓
validate numbers are odd	✗
validate numbers are even	✓

Job summary generated at run-time

There are various test tools available for different ecosystems. Some of them produce markdown reports that you can integrate with the workflow by

utilizing the GitHub step summary and pushing the data of such a file to this output. At the end of this chapter, we will show you how this is done with a concrete example of using a markdown logger that can be integrated with the dotnet tools.

8.4.5 The Use of Containers for Jobs

When we showed how to build sources into deployable artifacts, we could also have picked an alternative way to run those tools. Instead of relying on available tools on the GitHub runners or installing it during the run, you can also pick a container image yourself that you use to run your tools. You can select any image from the docker hub, refer to your container registry, and retrieve your container image to run a build. You only need to specify the container and set up a mounted volume to point to the sources you get from GitHub and where you place the produced output. Using container images versus relying on the available tools on the runners is a matter of preference. This can also sometimes be a way to circumvent issues with licensing of tools that require elaborate setup and setting licensing keys. One other advantage might be that you can also run the build locally with a docker command and don't need a separate setup for local work. You also have full control over the history. Even 10 years later, you are able to build the sources again as long as you don't delete the image.

Note

Since the container image you want to use needs to be downloaded, you can incur some slowdown at the start of the workflow.

You can also use images that come from a private repository. For this, you do need to provide the location and credentials to pull the image from the runner. In the listing below, you can see the exact same steps for how to build the dotnet application, but now it runs from a container image hosted on docker hub. You can execute actions like you are used to in any action workflow, the big difference is that you don't need to set up all kinds of tools and configuration, you can start directly with the task at hand.

As an example, you can see in **Listing 8.5** the exact same workflow as

previously shown to build the frontend of the globoticket application using the dotnet tools. Using the available Microsoft SDK container image saves you from installing and configuring the right tools and you can start the build process immediately.

Listing 8.5 Using a container for jobs

```
name: Build inside container
on:
  push:
    branches: [ main ]
  workflow_dispatch:
jobs:
  container-build-job:
    runs-on: ubuntu-latest
    container:
      image: mcr.microsoft.com/dotnet/sdk:6.0
    steps:
      - uses: actions/checkout@v3
      - name: Build
        run: dotnet build
      - name: Test
        run: dotnet test --no-build --verbosity normal
```

8.4.6 Multiple Workflows vs. Multiple Jobs, What to Choose?

An action workflow always has one or more jobs. These jobs are run in parallel and can have dependencies with other jobs. When a job is dependent on another job, these jobs are executed in sequence. A workflow is contained in one yaml file, and you can have more jobs in one file. You can create multiple workflows for a repository.

When should you choose a new workflow or a new job to do some work?

In most examples you see on the web and in examples given by GitHub itself, you often see multiple jobs in one file. While this is convenient in terms of keeping everything in one place, it also creates some issues.

The main issue is who will maintain the workflow file and who gets to review this file before the change is accepted? Especially in highly regulated organizations, there needs to be a strict separation of duties when making

changes that can impact the deployment to a production environment.

Another question is, what changes when I make changes to my software? When I make changes to the source code and its dependencies, this should only impact a small part of the system, not everything you have in terms of automation.

You can even boil this down to a very commonly used term in software development and part of the Solid Principles. In Solid, the ‘S’ stands for Single Responsibility, and we use this so we can keep changes to a minimum and make maintenance less hard and brittle over time. If you keep the work that needs to be done simple and you have clearly defined reasons when you want to run a particular workflow that has specific goals, you will end up with some more workflow files, that all have a single job. When you combine this with a well-defined branching strategy, you can very nicely use the different event types that we have in the development cycle as the moments you want to trigger a particular piece of automation.

When we come to Continuous Delivery, we often need to run automation on different machines. A job also has the ability to run on another machine. For this reason, it does make total sense to have multiple jobs, since each job can then execute on another machine. In this case, the jobs are a means to distribute the work, but the type of work is exactly the same, as we will see in the next chapter. Based on this, we propose a set of small workflow files with a specific purpose or goal. It keeps the cognitive load during maintenance on those files low, the group of people that need to review is also specific from an audit perspective. Please treat this as guidance, not a must-follow rule. If there is a reason to have multiple jobs, then please do so.

8.4.7 Parallel Execution of Jobs

In some situations, you might want to run a set of jobs in parallel that do the same thing but only with a few other parameters. An example of this would be building artifacts for various platforms like arm and x86. For this we can use the concept of matrix job strategy. With the matrix strategy you can use different variables to build the same code in a single job definition for different platforms and tooling. In **Listing 8.6** you can find an example of a

matrix strategy that builds the code on two platforms with three different node versions.

Listing 8.6 Using Matrix for parallel execution of Jobs

```
jobs:
  build:
    strategy:
      matrix:
        dotnet-version: [6, 7]
        processor: [x86, arm]
```

In this example, we would start parallel jobs for the following builds:

- dotnet version 6, processor x86
- dotnet version 6, processor arm
- dotnet version 7, processor arm
- dotnet version 7, processor x86

8.5 Preparing for Deployment

In your CI workflow that creates the final artifacts, you need to define where you want to store them for the next phase in the process, which is the Continuous Delivery phase. There are a few things that are important when we are preparing for release. We want to ensure we can trace back which change was made, by whom and how we can track this back from the environment we deployed to. We also want to ensure we use proper version numbering, and we want to ensure you can deploy the created artifacts in the most convenient way to the various environments.

Let's dive a bit deeper into traceability, versioning, and creating a GitHub Release. After we create a release, we then use GitHub package management to store our artifacts in GitHub Package management or the GitHub Container Registry.

8.5.1 Traceability of Source to Artifacts

When you work in more compliance-heavy organizations, you need to be

able to proof a certain change in the source code is tied to a requirement and that this particular source change is deployed in an environment. With GitHub you can make use of the fact that not only the actions are integrated with your source repository, but GitHub also provides ways to track requirements, defects, feature requests and more. This is all done with the use of GitHub issues.

When you commit source code to the repository, you can e.g. enforce you want the code to be validated before it is committed. For this Pull Requests are used and you can enforce them to be used by setting a branch policy. In your guidelines for approving a pull request, you can check that at least one issue is attached to the pull request so there is a traceable history to the requirement that was implemented with the code change. Unfortunately, branch policies don't have a way to enforce the required traceability to issues. So, you need to have the reviewer check themselves or create an action yourself to do this verification. Setting the branch policy is crucial here. You can set branch policies using the settings page, as shown in ***Figure 8.6***.

Figure 8.6 Branch Protection Rules

Branch protection rule

Branch name pattern *

main

Applies to 1 branch

main

Protect matching branches

☐ Require a pull request before merging

When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.

☐ Require status checks to pass before merging

Choose which [status checks](#) must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

☐ Require conversation resolution before merging

When enabled, all conversations on code must be resolved before a pull request can be merged into a branch that matches this rule. [Learn more about requiring conversation completion before merging.](#)

☐ Require signed commits

Commits pushed to matching branches must have verified signatures.

☐ Require linear history

Prevent merge commits from being pushed to matching branches.

When you use a pull request to merge the changes into the main branch, you can ensure there is always traceability to the requirements and that there is a four-eyes principle in place, which is a requirement in almost any governance framework you use for your compliance.

Figure 8.7 Tracing back to requirements

Update site.css #5

Merged vriesmarcel merged 2 commits into `main` from `feature/feature1` on Jul 16, 2023

Conversation 0 Commits 2 Checks 1 Files changed 2



vriesmarcel commented on Jun 24, 2023

Owner ...

fixes #2



1



vriesmarcel added 2 commits 8 months ago



Update site.css ...

Verified ✗ 486bd65



Update UnitTest1.cs ...

Verified ✓ e58a45a



vriesmarcel merged commit `6d7b6b8` into `main` on Jul 16, 2023

View details

Revert

1 check passed



vriesmarcel deleted the `feature/feature1` branch 7 months ago

Restore branch

With these comments in the commit messages, we now make it possible to track any change back to a requirement or change request defined as an issue.

You can see in **Figure 8.7** how you can refer in a pull request to an issue, to ensure we trace the changes back to the requirement.

8.5.2 Ensuring Delivery Integrity, the SBOM

With the attack on SolarWinds

(<https://msrc.microsoft.com/blog/2020/12/december-21st-2020-solorigate-resource-center/>), our industry became more aware of a new way of attacking our customers: via the CI/CD infrastructure we use as developers. This now imposes a new burden on us to validate if the software we created is actually the software we expected to deliver and validate if everything we used during the creation of the software was not tampered with. In May 2021, the president of the United States even ordered a presidential Executive Order that states software companies must help improve the nation's cyber security. This entails a way to validate the integrity of the software in production. You can find the executive order here <https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity> When we are running workflows that create artifacts, we want to ensure the integrity of those artifacts to ensure they are not tampered with after or during creation. This entails a set of

things, including the individual validation of the files used during creation as well as the assertion that the tools we used are not compromised.

This requires multiple layers of validation for each workflow. We need to check which actions we used with a workflow, and we need to check where the files we used are coming from. Also, when we produce an artifact that we will use later in our Delivery workflow, we need a way to transfer those files securely and easily. In section 8.5 Preparing for Deployment we will go into more detail about where you can store the artifacts before you start the deployment.

We also need to ensure this list of items we used while running a workflow. For this, the industry has defined a set of standards to create a so-called “bill of materials”. In full this is the Software Bill of Materials, in short called the SBOM. You can generate an SBOM in several ways. One for your repository dependencies. This one can be retrieved from the user interface when you go to the dependencies overview.

When you want to have an SBOM every time you create software artifacts, then you are better off using a GitHub action and making this part of your standard workflow that prepares artifacts before deployment. With GitHub Actions, you can create an SBOM by using several actions that are available in the marketplace. In **listing 8.7** you can see how to use the Microsoft SBOM generator action that generates an SBOM that is compliant with the NTIA specifications and delivers this in a SPDX format. SPDX is Software Package Data eXchange format. This is the open standard for communicating software bill of material information. It is good to note there are two competing standards, CycloneDX and SPDX. Microsoft and GitHub have chosen to use the SPDX standard.

Listing 8.7 Generate an SBOM using the Microsoft SBOM tool

```
name: Generate SBOM
run: |
    curl -Lo $RUNNER_TEMP/sbom-tool https://github.com/micros
    chmod +x $RUNNER_TEMP/sbom-tool
    $RUNNER_TEMP/sbom-tool generate -b ./buildOutput -bc . -p
```

Note that the example only shows you how to generate the SBOM. Normally

you also want to use this file as part of your release and you should upload it to the release as an artifact that is part of the release.

8.5.3 Versioning

One important part about releasing software that is often forgotten or an afterthought is the versioning of what you release. There are many ways we are creating version numbers in our industry, and in the last few years, you might have seen the industry is moving to a more standardized versioning. Two of the most used types of versioning are named semantic versioning and calendar versioning.

Semantic versioning

As the name implies, it means that we adhere to a set of semantics when we bump a version number. The basic idea behind semantic versioning is that based on the version number, you can tell if a new version of a package, library, image, or artifact is backward compatible. The thinking behind this and all the details can be found here: <https://semver.org>.

In a nutshell, the versioning works as follows:

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backward-compatible manner
- PATCH version when you make backward-compatible bug fixes

If you want the version number to be calculated based on your branches, you can use an action called `GitVersion` (see <https://gitversion.net/>). `GitVersion` is part of the `GitTools` action (see <https://github.com/marketplace/actions/gittools>). `GitVersion` looks at your git history and works out the Semantic Version of the commit being built. For `GitVersion` to function properly, you have to perform a so-called un-shallow clone. You do this by adding the `fetch-depth` parameter to the checkout action and setting it to 0. Next, install `GitVersion` and run the `execute` action. Set an

id if you want to get details of the semantic version. This is shown in Listing 8.8:

Listing 8.8 Use of GitVersion action

```
steps:
- uses: actions/checkout@v3
  with: fetch-depth: 0

- name: Install GitVersion
  uses: gittools/actions/gitversion/setup@v0.9.7
  with:
    versionSpec: '5.x'

- name: Determine Version
  id: gitversion
  uses: gittools/actions/gitversion/execute@v0.9.7
```

The calculated final semantic version number is stored as the environment variable `$GITVERSION_SEMVER`. You can use this, for example, as the input for the version of a package that you publish.

If you need to access details from GitVersion (such as major, minor, or patch), you can access them as output parameters of the `gitversion` task:

Listing 8.9 Use of version number by referring previous step

```
- name: Display GitVersion outputs
  run: | echo "Major: ${ steps.gitversion.outputs.major }"
```

It is also possible with Semantic versioning to indicate the quality of the build as part of the version number. You do this on pre-releases or alpha versions of a soon to be stable new version. It is common to use for this the notation: `v1.0.0-pre` or `v1.0.0-alpha`

Calendar versioning

As this name implies the version number is generated based on the calendar and the moment the workflow is executed. Depending on the release frequency of your application you can choose to include the date up until the

minute of release, or simply keep it to the date of today. In **Listing 8.10** you can see an example of how we can generate a calendar-based version. If we assume it is May 29th in the year 2023 then the output in the variable is 2023-05-29 and can be used in subsequent parts of the workflow by referencing the variable `$BUILD_VERSION` using the environment context.

Listing 8.10 Use of Calendar action

```
- name: Set Release Version
  run: echo "BUILD_VERSION=$(date --rfc-3339=date)" >> $GITHUB_EN

- name: use the variable
  run: echo ${ env.BUILD_VERSION }
```

8.5.4 Testing for Security with Container Scanning

In general, when you prepare artifacts that are going to be deployed to a production environment, you like them to be scanned if they are secure. When building containers, we can use various tools to run a validation if there are known vulnerabilities in the container image. I like to use the open-source tool provided by aqua security called Trivy. You can add Trivy scanning to your workflow by adding one additional step. In **Listing 8.11**, you can see how to use this action to scan your image and fail when it finds a vulnerability with the severity of Critical or High.

Listing 8.11 Adding container image scanning step

```
- name: Run Trivy vulnerability scanner
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: '${ env.containerRegistry }/${ env.imageRepository }'
    format: 'table'
    severity: 'CRITICAL,HIGH'
    exit-code: '1'
```

By adding this extra step, your workflow will fail when a vulnerability is found in the container image preventing you from pushing the image to the image registry. A best practice is always to scan before you push your image to the registry, preventing it from ever getting into an environment and hence preventing you from being breached. Adding security as early as possible in

the development cycle is often referred to as shifting left.

8.5.5 Using GitHub Package Management and Container Registry

Many organizations use artifact repositories to keep the artifacts in a safe place where we can pull them from when we move to the deployment phase. GitHub also provides capabilities to act as an artifact repository. This is called GitHub packages and is available for multiple package management solutions. You can find the supported artifacts in **table 8.1**.

Table 8.1 Supported Artifacts

Language	Description	Package format	Package client
JavaScript	Node package manager	package.json	npm
Ruby	RubyGems package manager	Gemfile	gem
Java	Apache Maven project management and comprehension tool	pom.xml	mvn
Java	Gradle build automation tool for Java	build.gradle or build.gradle.kts	gradle
.NET	NuGet package management for .NET	nupkg	dotnet CLI
N/A	Docker container management	Dockerfile	Docker

As the last step in your workflow, you can use the package manager that matches the ecosystem you are working on and push it to the GitHub Artifact registry.

You publish packages when you are building libraries that are used between projects or when you have a shared solution between various components or microservices. Packages are published and from there on used by other CI workflows. When you publish a package to an ecosystem like npm, NuGet or RubyGems it is a good practice to also create a release when you publish. This way it is clear you released a new version of your package, so others can pick it up. Creating a release is described further down in this chapter, since it can also be a source to start a deployment. You find more about creating a release in **paragraph 5.5.8**

GitHub also provides a container registry where you can store container images you create during your CI workflows. To authenticate against the package management capability, we need to extend our authorization token to include write permissions on packages. In **listing 8.12** you can see how to set these permissions and some examples of how to push a container image to the GitHub packages endpoint.

Listing 8.12 creating a container image and upload to GitHub

```
name: "chapter 08: create-container-and-push-frontend"
permissions:
  actions: write
  packages: write
  contents: read

on:
  push:
    branches: ["main"]
    paths:
      - 'frontend/**'
  workflow_dispatch:

jobs:
  build:
    uses: ../.github/workflows/create-container-and-push.yml
    with:
      imageRepository: 'frontend'
      containerRegistry: 'ghcr.io/githubactionsinaction'
      dockerfilePath: 'frontend/Dockerfile'
      namespace: 'globoticket'
    secrets:
      registryPassword: '${{ secrets.EXTENDED_ACCESSTOKEN }}
```

Because we need to create a container image for every service we have in our application, we used a reusable workflow that actually builds the container. You see the reference to `uses: ../github/workflows/create-container-and-push.yml`; this refers to the reusable action workflow that is shown in **listing 8.13**

Listing 8.13 reusable workflow that creates and pushes the container

```
name: "chapter 08: create-container-and-push"
permissions:
  actions: write
  packages: write
  contents: read
on:
  #define the input parameters for this workflow used in the work
  workflow_call:
    inputs:
      imageRepository:
        required: true
        type: string
      containerRegistry:
        required: true
        type: string
      dockerfilePath:
        required: true
        type: string
      namespace:
        required: true
        type: string
    secrets:
      registryPassword:
        required: true
  # the input parameters are also defined for a manual trigger
  workflow_dispatch:
    inputs:
      imageRepository:
        required: true
        type: string
        default: 'frontend'
      containerRegistry:
        required: true
        type: string
        default: 'ghcr.io/vriesmarcel'
      dockerfilePath:
        required: true
```

```

        type: string
        default: 'frontend/Dockerfile'
namespace:
    required: true
    type: string
    default: 'globoticket'
jobs:
  build:
    # we check out the sources, determine the version number and
    # login to the container registry
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
        with:
          fetch-depth: 0

      - name: Install GitVersion
        uses: gittools/actions/gitversion/setup@v0.10.2
        with:
          versionSpec: '5.x'

      - name: Determine Version
        id: gitversion
        uses: gittools/actions/gitversion/execute@v0.10.2

      - name: Login to GitHub
        uses: docker/login-action@v2
        with:
          registry: ghcr.io
          username: ${github.actor}
          password: ${secrets.registryPassword}
    # we use docker buildx to create a builder instance and the
    # build and push the image
    - name: select docker driver
      run: |
        docker buildx create --use --driver=docker-container
    # we use this action to determine the labels for the image
    - name: Docker meta
      id: meta
      uses: docker/metadata-action@v4
      with:
        images: actions-with-actions/globoticket
    # build and push the image to the container registry
    - name: Build and push
      uses: docker/build-push-action@v4
      with:

```

```
context: ${github.workspace}}
file: ${inputs.dockerfilePath}}
push: true
tags: ${inputs.containerRegistry}}/${inputs.imageRe
cache-from: type=gha
cache-to: type=gha,mode=max
labels: ${steps.meta.outputs.labels}}
```

You can see in the reusable action workflow that we push the resulting artifact to the GitHub artifact registry. You can do this in a similar way if you are pushing Packages that are from any of the supported package managers. When Pushing a package you also use the GitHub Token to authenticate against the package registry.

Linking the package to the repo

It is important to note that you need to link the package that you publish to the repository. Linking it back to the source repository enables it to also send events that you can use to trigger e.g. the release and deployment. You enable this by either creating the link in the GitHub portal using the page that you find when you look the details of the package. This is shown in **Figure 8.8**.

Figure 8.8 Link package to repo



Link this package to a repository

By linking to a repository, you can automatically add a Readme, link discussions, and show contributors on this page.

Connect Repository

Or link via Dockerfile

[Learn more](#)

To locally connect your container image to a repository, you must add this line to your Dockerfile:

```
LABEL org.opencontainers.image.source https://github.com/OWNER/REPO
```

Note: To connect a repository to your container image, the namespace for the repository and container image on GitHub must be the same. For example, they should be owned by the same user or organization.

You can also enable this link back to the source repository by providing the metadata during publication on the docker push action, or by adding the label to the docker image when you build it.

8.5.6 Using the Upload/Download Capability to Store Artifacts

In the case you are not using container images or packages and have a set of binaries or a zip file that you want to retain as part of your CI workflow, you can use an action called `actions/upload-artifact`. This action can take any set of arbitrary files and upload them to GitHub. Another workflow can then retrieve these files using the `actions/download-artifact` action.

When creating artifacts to deploy our sample application to a Kubernetes cluster, we need to produce a deployment descriptor file that references the newly created container during our CI. A way to do this is by using an action that can annotate an existing file you have in your repository and then outputting the altered results as an artifact we are going to store on GitHub. This can then later be retrieved by the deployment workflow.

Listing 8.14 shows a simple example of a workflow storing a file and retrieving it in a second job.

Listing 8.14 Upload Artifacts to GitHub

```
name: Upload and Download arbitrary artifacts

on:
  workflow_dispatch:
env:
  deploymentFile: 'file-I-want-to-use-in-deploy-phase.txt'
jobs:

  build:

    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: create a file we will use in next job
        run: |
          touch ${github.workspace}/${env.deploymentFile}

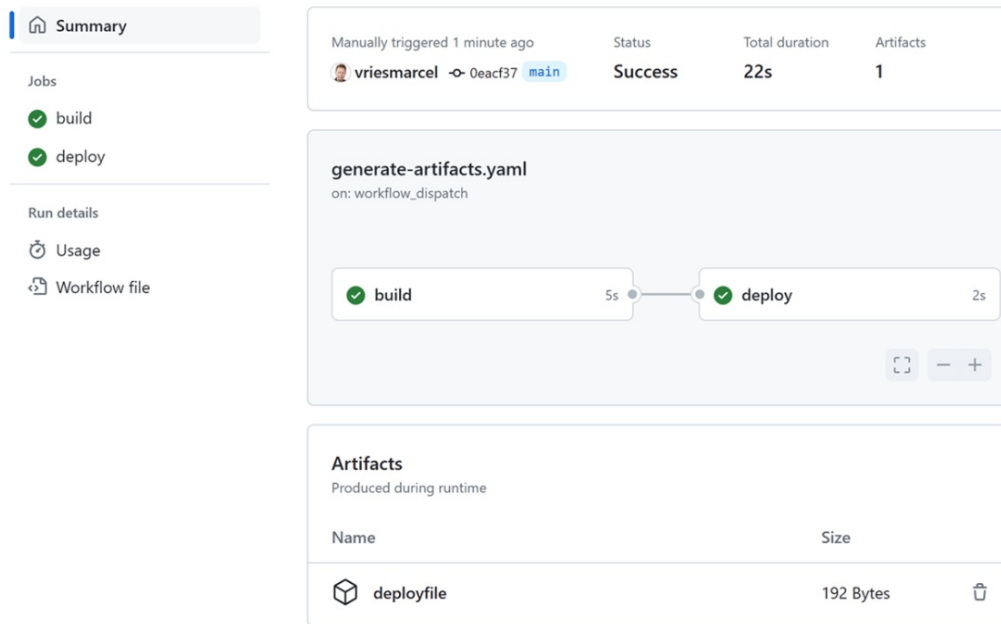
      - name: Upload a Build Artifact
        uses: actions/upload-artifact@v3
        with:
          name: deployfile
          path: ${github.workspace}/${env.deploymentFile}

  deploy:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - name: Download artifact from build job
        uses: actions/download-artifact@v3
        with:
          name: deployfile
      - name: show files downloaded
        run: |
          ls ${github.workspace}
```

The result of this workflow is that we uploaded a file, and we can also see this result in a second job that was started with the name deploy.

You can see the artifacts you create in the UI as shown in **figure 8.9**:

Figure 8.9 Artifact publishing



8.5.7 Preparing Deployment Artifacts

When you release your software, you want to get a fully prepared package that you can deploy. In the case with the example we are using, we need not only a set of containers in the container registry, but we also need a set of files that we use to run the deployment to the Kubernetes cluster. These are deployment files that contain a reference to the image we want to run.

To ensure you have a complete package that is good and traceable to the source and changes, it is preferred to prepare the deployment files as part of the CI workflow. In the case of the deployment of `globoticket`, this means we take the Kubernetes deployment file that we use as a template for the deployment and replace variables in this template file. After creating the containers and scanning them for known vulnerabilities, you then create the deployment file with the tags that got created while building the containers. After the replacement of the variables in the template, you can make this part of the artifacts that get pushed to the repo for another workflow to pick them up.

For transformation of existing files, we use the action `cschleiden/replace-`

tokens. This action has the option to specify a replacement token you like, and then replace this across a set of files. The example here is the tag of the container that will get pulled by Kubernetes with the tag created while creating the container.

Listing 8.15 shows how to prepare a Kubernetes deployment file.

Listing 8.15 Kubernetes template Deployment File

```
apiVersion: apps/v1
# Kubernetes deployment specification. We want to deploy our
# container frontend to the cluster, in the namespace globoticket
kind: Deployment
metadata:
  name: frontend
  namespace: globoticket
  labels:
    app: frontend
# We want to deploy 3 replicas of the frontend and deploy
# them with a rolling update strategy
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 0
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    # here is the specification of the container we want to deploy
    # we set the resource limits and requests according to best prac
    # we pull the image from github container registry, using the se
    # defined in the pullsecret
    spec:
      containers:
        - name: frontend
          image: ghcr.io/vriesmarcel/frontend:#{Build.version}#
          resources:
            requests:
              memory: "500Mi"
```



```

        cpu: "250m"
    limits:
        memory: "1Gi"
        cpu: "750m"
    env:
    - name: ASPNETCORE_ENVIRONMENT
      value: Development
    - name: ApiConfigs__EventsCatalog__Uri
      value: http://catalog:8080
    - name: ApiConfigs__Ordering__Uri
      value: http://ordering:8080
    ports:
    - containerPort: 80
    imagePullPolicy: Always
    imagePullSecrets:
    - name: pullsecret

```

When you look at the file, you see the markers that can be used to replace. In this case, I am going to replace the part that states `#{Build.version}#` with the number of the build that we are running. This is the same as the number we generate when we create the image and push the image to the registry. By ensuring these numbers are the same, you guarantee that you deploy exactly those images.

Replacement can now be done by pointing to this file and defining the replacement tokens and the variable for `Build.version`. How to do this is shown in **Listing 8.16**:

Listing 8.16 Replacing tokens

```

- name: Replace tokens
  uses: cschleiden/replace-tokens@v1.0
  with:
    files: '["${{github.workspace}}/${{env.deploymentfileFo
      /frontend.yaml"]'
  env:
    Build.version: ${{env.FRONTED_VERSION}}

```

After replacing the tokens in the file, you then upload them to the artifact store as described in the previous paragraph, so they can be retrieved the moment we want to run the deployment.

8.5.8 Creating a Release

Creating a release is the starting point of moving the created deployment artifacts to the outside world. It is the handoff moment to the CD Workflow that does the actual deployment. The deployment artifacts can be a set of packages that are going to be published, a set of container images to be pulled from a container registry.

You can create a release in GitHub by using the create release page in GitHub. When using this page, you are doing it manually, which can be a good practice if you want to separate duties of people who can create releases or not. This release defines what we want to release, and we prefer to add all artifacts that we deploy to be part of this release.

It is preferred to create a release using an action in the CI workflow. When using a branching strategy like GitHub flow or Trunk based development, you create a new release the moment you merged a change into the main branch. The main branch is the source to release to the production environments. How this is done is normally via a pull request that is merged, helping you to ensure compliance by providing good traceability and a 4 eyes principle before something can move to a production environment.

You can define that regardless of how the change moved to the main branch, the moment we detect a change, we want to trigger the CI workflows first and after all have completed and are successful, we want to create a release that in its turn will trigger the CD workflow that moves the software to a production environment, with the necessary steps based on the process you want to follow.

You can trigger the release, e.g., the moment a new container image is published by one of the previous workflows. **Listing 8.17** shows the workflow that is triggered by the publication of the container image, picks up the version number from the image and produces a file that is used for deployment to the Kubernetes cluster. This file is attached as an artifact that is part of the release, so it can be used by the CD workflow that we discuss in the next chapter.

Listing 8.17 Creating a release automatically

```

name: "chapter 08: create release"
permissions:
  actions: write
  packages: write
  contents: read
on:
  registry_package:
    types: [published]

env:
  deploymentFolder: 'deployment-automation'
  GH_TOKEN: ${ secrets.EXTENDED_ACCESSTOKEN } #required for gh
# only run this workflow when a package is published with a tag
# that is not empty. We cancel any other releases that are in pro
# before we create the GH Release, we need the latest version of
# we use these to patch the deployment files with the correct ver
# and then create a release with the version provided by the pack
jobs:
  release:
    if: github.event.registry_package.package_version.container_m
    concurrency:
      group: ${github.event.registry_package.package_version.con
      cancel-in-progress: true
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
      # get the versions of the images from the package registry
      - name: Retrieve latest image version frontend
        run: |
          export FRONTED_VERSION=
          $(gh api user/packages/container/frontend/versions |
            jq -r '.[0].metadata.container.tags[0]')
          echo "FRONTED_VERSION=$FRONTED_VERSION" >> $GITHUB_ENV
          export ORDERING_VERSION=
          $(gh api user/packages/container/ordering/versions |
            jq -r '.[0].metadata.container.tags[0]')
          echo "ORDERING_VERSION=$ORDERING_VERSION" >> $GITHUB_EN
          export CATALOG_VERSION=
          $(gh api user/packages/container/catalog/versions |
            jq -r '.[0].metadata.container.tags[0]')
          echo "CATALOG_VERSION=$CATALOG_VERSION" >> $GITHUB_ENV

      # patch the deployment files with the correct versions.
      # we do this for catalog, frontend and ordering
      - name: Replace tokens
        uses: cschleiden/replace-tokens@v1.0
        with:

```

```

    files: '["${{github.workspace}}/${{env.deploymentFolder
env:
    Build.version: ${env.CATALOG_VERSION}}

- name: Replace tokens
  uses: cschleiden/replace-tokens@v1.0
  with:
    files: '["${{github.workspace}}/${{env.deploymentFolder
env:
    Build.version: ${env.FRONTED_VERSION}}

- name: Replace tokens
  uses: cschleiden/replace-tokens@v1.0
  with:
    files: '["${{github.workspace}}/${{env.deploymentFolder
env:
    Build.version: ${env.ORDERING_VERSION}}

# create a release with the version provided by package pus
# contains the deployment files

- name: create a relase with version provided by package pu
  uses: softprops/action-gh-release@v1
  with:
    token: "${{ secrets.EXTENDED_ACCESSTOKEN }}"
    tag_name: "v${{github.event.registry_package.package_ve
    generate_release_notes: true
    files: |
      ${{github.workspace}}/${{env.deploymentfileFol
      ${{github.workspace}}/${{env.deploymentfileFolder}}/ordering.yaml
      ${{github.workspace}}/${{env.deploymentfileFolder}}/catalog.yaml

```

After running this workflow, you will find the release in the releases section of GitHub, and more important, an event is generated to signal a new release is created. It is also possible to use the GitHub API to add files to the release, you an think e.g. attaching the SBOM that we discussed in paragraph 8.5.2

This is also the reason this workflow uses a different token than the standard GitHub token that is available in the workflow. If we use the default token, the release will not trigger any new workflows that could e.g. take care of the deployment. The token that is stored in GitHub secrets provides the ability to trigger a new workflow as part of the publication process.

This way of working ensures you have a very clear and simple workflow that has a focus on creating the CI end result, a release. Now it becomes better maintainable and can be secured in terms of who is allowed to review the change before acceptance. It is a good practice to upload all files you need as part of the deployment process. That way the release becomes the container of all deliverables you need to execute a release and hence the perfect hand-over to the CD workflows we cover in the next chapter.

8.6 The CI Workflows for GloboTicket

Now that we have the concepts in place, let's start creating the CI workflows we need to get our GloboTicket application ready for deployment. GloboTicket has two API's and one front-end web application that needs to get deployed. If we take this application and design the CI workflows then we need the following:

- One workflow to validate the integration on each pull request
- The same workflow that validates the integration in the main branch
- One workflow that tests the API's or the front-end application using the available unit tests
- One workflow to check for known vulnerabilities in the committed sources and the dependencies in use
- One workflow that creates the artifacts ready for the deployment to a Kubernetes cluster

Let us go through these workflows one by one, so you get a full end-to-end view on how we can prepare everything for deployment to the cloud.

8.6.1 The Integration CI for API's and Front-end

This workflow will trigger the moment we commit a change to any feature branch. The first step is to get the sources and then we use the dotnet tools to compile the sources. This workflow only compiles the source so we know that what we committed integrates and compiles. This way we get as fast as possible feedback to the developer that is building a new feature on a feature branch. The action workflow for this CI is shown in *listing 8.18*

Listing 8.18

```
name: "chapter 08: Compile and Test fast feedback"
permissions:
  actions: write
  contents: read

on:
  workflow_dispatch:
  push:
    branches: [ "feature/*" ]
    paths:
      - 'frontend/**'
      - 'catalog/**'
      - 'ordering/**'

jobs:
  build:

    runs-on: ubuntu-latest

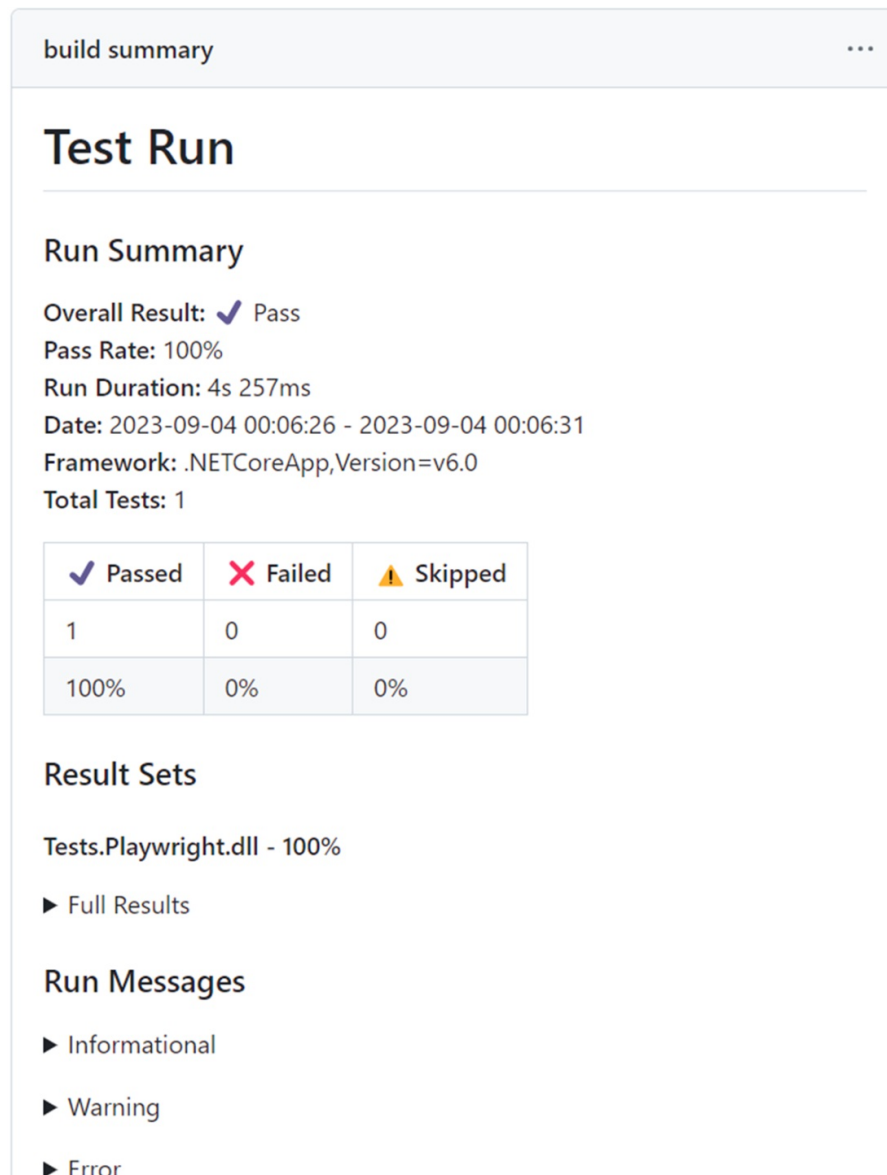
    steps:
      - uses: actions/checkout@v3
      - name: Setup .NET
        uses: actions/setup-dotnet@v3
        with:
          dotnet-version: 6.0.x

      - name: Restore dependencies
        run: dotnet restore
      - name: Build
        run: dotnet build --no-restore
```

8.6.2 The CI Workflows for Quality Control

This workflow aims to check if the software is still working according to requirements. This is validated by running the Unit Test projects that are part of the project. In dotnet this involves using the built-in test tools. To get the right test results in the output, it is possible to use a specific logger that can produce markdown output. This output file can then be output to the step results, so it shows up in the final report. This way, you get a nice report that is visible in the GitHub user interface.

Figure 8.10 Test result summary



After you run the workflow, you will see the results as shown here in **Figure 8.10**

The workflow for Quality Control on GloboTicket is shown in **Listing 8.19**. We run this workflow the moment we create a pull request. This provides input to the team of reviewers and the developer of the feature what the state of the feature is. It is fine to combine the first CI and this CI workflows into one, when the unit tests provide fast feedback. The moment this can take several minutes, it makes more sense to split them.

Listing 8.19

```

name: Compile and Test --fast feedback
permissions:
  actions: write
  contents: read
env:
  GH_TOKEN: ${ github.token }

on:
  workflow_dispatch:

  pull_request:
    branches: [ "main" ]
    paths:
      - 'frontend/**'
      - 'catalog/**'
      - 'ordering/**'
jobs:
  build:

    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Setup .NET
        uses: actions/setup-dotnet@v3
        with:
          dotnet-version: 6.0.x

      - name: add markdown report logger for frontend project
        run: dotnet add unittests/unittests.csproj package LiquidT

      - name: Test
        run: dotnet test --logger "liquid.md;logfile=testResult

      - name: Output the results to the actions jobsummary
        if: always()
        run: cat $(find . -name testResults.md) >> $GITHUB_STEP_SUM

```

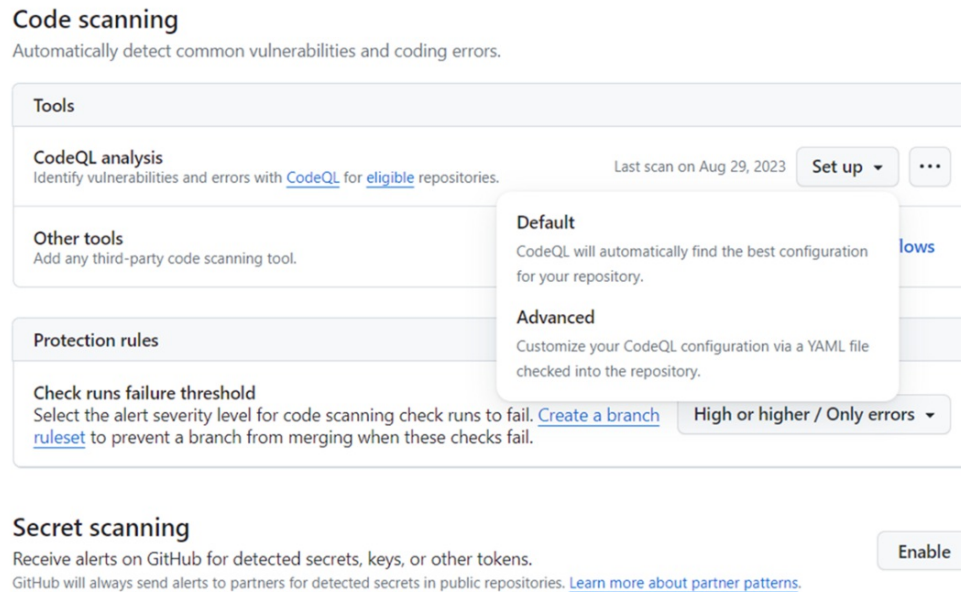
8.6.3 The CI Workflow for Security Testing

This workflow aims to periodically check the software on the main branch and the moment we push changes. The software will be checked on known vulnerabilities that are present in the software produced by the development team. This is done using the GitHub Advanced Security scanning tool, which

is reported back to the GitHub security dashboard in the UI. This is done by activating Advanced Security in the account and selecting the setup CodeQL Analysis.

This is shown in **Figure 8.11**

Figure 8.11 GitHub advanced security



When you run this workflow, you will find the CodeQL Analysis finds four known vulnerabilities in the code we have for GloboTicket. All with a severity of High! You can see the results in **Figure 8.12**.

Figure 8.12 Code Scanning Results

Code scanning

The screenshot displays the GitHub Code Scanning dashboard. At the top, a green status bar indicates "All tools are working as expected". Below this is a search bar with the text "is:open branch:main". The main section shows a list of four vulnerabilities, all marked as "High" severity. Each entry includes a checkbox, a shield icon, the vulnerability name, a "High" severity tag, and a "main" branch tag. The vulnerabilities are:

- Inefficient regular expression** (High): #3 opened 4 months ago • Detected by CodeQL in frontend/.../dist/query.validate.js:1394
- Inefficient regular expression** (High): #2 opened 4 months ago • Detected by CodeQL in frontend/.../dist/additional-methods.js:1092
- Inefficient regular expression** (High): #1 opened 4 months ago • Detected by CodeQL in frontend/.../dist/additional-methods.js:1092
- Log entries created from user input** (High): #4 opened 4 months ago • Detected by CodeQL in frontend/Controllers/CheckoutController.cs:51

At the bottom, a "ProTip!" message states: "The libraries and queries that power CodeQL are open-source. [Learn more](#)".

After scanning for known vulnerabilities in the code, the next step is to also scan for known vulnerabilities in the container images. For this, the workflow determines the latest version of the container images available and then runs the tools shown in **paragraph 5.5.4**. We can also extend this to use the same GitHub Security Dashboard, by configuring the Trivy security scanner to output a sarif format and then upload this to GitHub. SARIF (Static Analysis Results Interchange Format) is an OASIS Standard that defines an output file format. The SARIF standard is used to streamline how static analysis tools share their results.

This workflow will find multiple known vulnerabilities in the container images. Solving these vulnerabilities is easy to mitigate by changing the default base images used for .NET Core containers to Alpine instead of Ubuntu.

The result of the workflow will show up in the code scanning results, as shown in **Figure 8.12**

The workflow for Security is also triggered on a pull request since it takes some more time to complete. It is also set up to run on the main branch when there is a push and on a schedule, so we always keep an eye on potential new vulnerabilities. The listing for the workflow is shown in **listing 8.20**

Listing 8.20 Security testing

```

name: "chapter 08: Security Testing"

env:
  imageRepository: 'frontend'
  containerRegistry: 'ghcr.io/xpiritcommunityevents'
  dockerfilePath: 'frontend/Dockerfile'

on:
  workflow_dispatch:

jobs:
  # run the codeql analysis on the code
  # we use a matrix to run the analysis on multiple languages
  # we define the languages c# and javascript
  analyzecode:
    name: Analyze
    runs-on: ${{ (matrix.language == 'swift' && 'macos-latest') |
                  'ubuntu-latest' }}
    timeout-minutes: ${{ (matrix.language == 'swift' && 120) ||
                          360 }}
    permissions:
      actions: read
      contents: read
      security-events: write

    strategy:
      fail-fast: false
      matrix:
        language: [ 'csharp', 'javascript' ]

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Initialize CodeQL
        uses: github/codeql-action/init@v2
        with:
          languages: ${{ matrix.language }}

      - name: Autobuild
        uses: github/codeql-action/autobuild@v2

      - name: Perform CodeQL Analysis
        uses: github/codeql-action/analyze@v2
        with:
          category: "/language:${{matrix.language}}"

  # next we run the trivy vulnerability scanner on
  # our container images. This way we can find vulnerabilities

```

```

# in our container images. We determine the latest version of
# the images and use that version to scan. This is done with
# the GitVersion tool.
analyzecontainers:
  runs-on: ubuntu-latest
  permissions:
    actions: read
    contents: read
    security-events: write
    packages: read

  steps:
    - name: Checkout repository
      uses: actions/checkout@v3
      with:
        fetch-depth: 0
    #determine the version of the image
    - name: Install GitVersion
      uses: gittools/actions/gitversion/setup@v0.9.7
      with:
        versionSpec: '5.x'

    - name: Determine Version
      id: gitversion
      uses: gittools/actions/gitversion/execute@v0.9.7

    # use trivy to scan the container image
    - name: Run Trivy vulnerability scanner
      uses: aquasecurity/trivy-action@master
      with:
        image-ref: ${env.containerRegistry}/${env.imageRepos}
        format: 'sarif'
        output: 'trivy-results.sarif'
      env:
        TRIVY_USERNAME: ${github.actor}
        TRIVY_PASSWORD: ${secrets.GITHUB_TOKEN}

    # upload the results to the security tab in github
    # this is the same place where the codeql results are uploa

    - name: Upload Trivy scan results to GitHub Security tab
      uses: github/codeql-action/upload-sarif@v2
      with:
        sarif_file: 'trivy-results.sarif'

```

8.6.4 The CI Workflows for Container Image creation and

Publishing

This workflow gets triggered the moment the sources are pushed to the main branch.

It will only create and push the new images to the registry, and we let the container registry trigger the creation of a new release. When you don't use containers, this would also be the workflow that would create the release immediately after the creation of the artifacts and it would store those in the release so it can be used for the deployment.

The workflow that creates and publishes the containers uses the reusable workflow defined in *listing 8.15*. In *listing 8.16* you find the yaml to create the container.

8.6.5 Creating a release

The moment the container images are published, we can use that as a trigger to create a release. We create the Kubernetes deployment files that go with the release. We need to determine what version numbers the various containers have at the container registry and use the correct version numbers in the deployment descriptor files we need at deployment. It will pick up the version numbers from the container images that got published, so that is all in sync. It also publishes the deployment file as an artifact of the release so that can be used during deployment. The listing for the yaml is in *listing 8.19*

8.7 Bringing it all together

In this chapter we started by describing the goals of Continuous Integration. We defined the types of integration workflows we typically use and described how you can split up your CI workflows. We use the GitHubFlow branching strategy and have small action workflows that have a very specific tasks and trigger on specific points in the GitHubFlow process. The overall structure is shown in figure 8.13.

Figure 8.13 GitHub Flow and the various workflow triggers

code quality. The final step in CI is to package up the artifacts for Continuous Delivery (CD). The most appropriate handoff mechanism in GitHub is to use the release and package the artifacts for deployment as part of the release.

- Artifacts are stored as part of a workflow execution and can be made part of a release. The latter is a great moment to hand off to release and provide a version number.
- For GloboTicket we create container images and push them to the container registry. We also created deployment descriptors that are used to deploy the containers to the Kubernetes cluster. These files are created and stored in the release.