Extracted from:

Creating Software with Modern Diagramming Techniques

Build Better Software with Mermaid

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Creating Software with Modern Diagramming Techniques

Build Better Software with Mermaid

Ashley Peacock

Edited by Michael Swaine

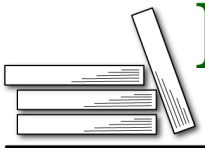
Creating Software with Modern Diagramming Techniques

Build Better Software with Mermaid

Ashley Peacock

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: L. Sakhi MacMillan
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-983-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2023

Devise a Plan of Action

Before we delve into creating an action, let's take a step back and look at what exactly we need to do to achieve our outcome. As a reminder, we want any Markdown files that are in a specific folder (in this example it will be `/docs`) to be uploaded to GitHub Pages, with any Mermaid markup converted to display the diagram as an image. We'll see it in action later, but Mermaid comes with a CLI that will do all of the heavy lifting for us. We can simply give the CLI a Markdown file containing Mermaid markup; it will generate SVGs from that markup and update the Markdown to reference the SVG rather than the Mermaid markup.

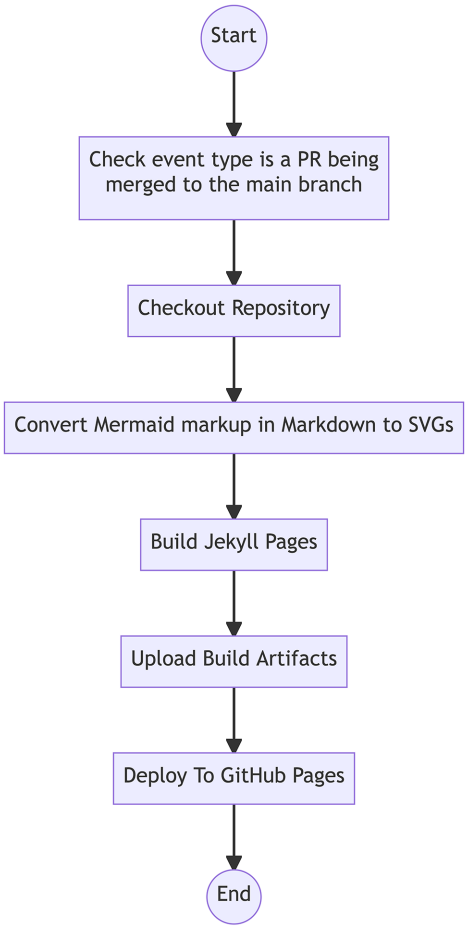
Here are what I believe to be the steps we need to take, and what better way to show them than in the diagram shown on page 6?

We'll go through each step in detail as we build the action, but I'll briefly touch on each one first:

1. We only want our action to trigger in certain scenarios. For our action, it's when code is pushed to the main branch (likely when a PR is merged, but you can commit straight to main too).
2. We can't do much without our repository, so we need to check out the repository so the action has access to it.
3. Then we need to find any Markdown files and convert any Mermaid markup in them to SVGs.
4. We'll be using Jekyll, which is a simple static site generator, to host our site on GitHub Pages. We're using this simply because GitHub has native support for Jekyll-based sites. The build step prepares the files as Jekyll expects them.
5. Penultimately, we upload the prepared build artifacts from the prior step ready to be deployed.
6. Finally, we use the uploaded build artifacts to deploy them to our GitHub pages site.

If you're not familiar with GitHub Actions, or GitHub Pages, it may not be immediately obvious how to do any of these steps. Don't worry, though; it'll be much simpler than it sounds, and we won't even need to write much actual code to achieve it!

You may have noticed that the preceding diagram is a very simple flowchart, of course created using Mermaid. This is a throwaway diagram, in that it's



just to visualize the steps I needed to take. Not all diagrams have to be a work of art or be complex. Sometimes it can help to just take a step back and visualize what it is you need to do.

Learn the Basics of a GitHub Action

Ready? Let's get started!

As opposed to other chapters where you completed an exercise at the end, this time treat this as a step-by-step tutorial to creating your first GitHub Action. As I take you through the steps, follow along using your own GitHub account.

First, we need to create a brand-new action. GitHub makes this super-simple; you go to a repository (or create a new one—remember it has to be public!),

click the Actions tab at the top, click the new workflow button, and finally select Simple Workflow. Each action you create can be thought of as a workflow, a series of steps to take to complete a given task.

You should now have something like the following:

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

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

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

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

    # Steps represent a sequence of tasks that
    # will be executed as part of the job
    steps:
      # Checks-out your repository
      # so your job can access it
      - uses: actions/checkout@v3

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

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

This is the basic structure for any GitHub Action and comes full of helpful comments. For anyone not familiar, the configuration is defined using YAML, which is a human-friendly data serialization format.

I'll add a little more information to some of the sections, starting with controlling when the workflow will run. This is defined by this section:

```
on:
  # Triggers the workflow on push or pull request events
  # but only for the "main" branch
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

Under `on`, we can list any number of GitHub Events[1] to trigger an action. In this case, it will trigger when code is pushed and when anything happens to a pull request (for example, opened or closed, but there are many more). We can further refine, for each event, when to trigger an action. In this example, the action will only get triggered on the "main" branch. GitHub's documentation contains all the possible events and the filter options for each event.

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

Next, we define the jobs we want to run for this action and what operating system to run them on:

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

Under jobs, you can list as many jobs as you like. In the example action, there's just one—build, which runs on Ubuntu. Other operating systems are available,[2] including Windows and Mac, but they're more expensive. You could run all your actions with a single job, but there are efficiency gains from defining multiple and parallelizing the jobs where possible, as any number of jobs can run in a single workflow in parallel.

A classic example of this would be deployments. We might want to run our test suite and at the same time build the Docker container ready for deployment. We could do them sequentially, but it would be a lot faster to run them in parallel. Similarly, if we're deploying multiple elements, for example a web app and a Kafka consumer, we can possibly do so in parallel.

Finally, each job can have any number of steps to complete that job, as shown in the example:

```
# Steps represent a sequence of tasks that
# will be executed as part of the job
steps:
  # Checks-out your repository
  # so your job can access it
  - uses: actions/checkout@v3

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

Unlike jobs, steps cannot run in parallel at this time and always run sequentially. Each step can run commands, same as you would in your terminal, or leverage another GitHub Action to run. GitHub itself provides many actions for you to use. In our first step we're using their checkout action, which checks out the repository's code, so that our action has access to the codebase. In the second step, it simply outputs some text by executing code

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

against the runner's shell. GitHub's documentation[3] is extensive and can explain every possible workflow configuration option. I've covered the basics we'll need.

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