
Universe2020

Automate your workflows with GitHub Actions and GitHub Packages

@stebje @rwnfoo / Workflow Automation

Universe 2020

Steffen Bjerkenås

@stebje
Solution Architect
Professional Services

Rowena Foo

@rwnfoo
Implementation Engineer
Professional Services

Workshop Outcomes

GitHub Actions

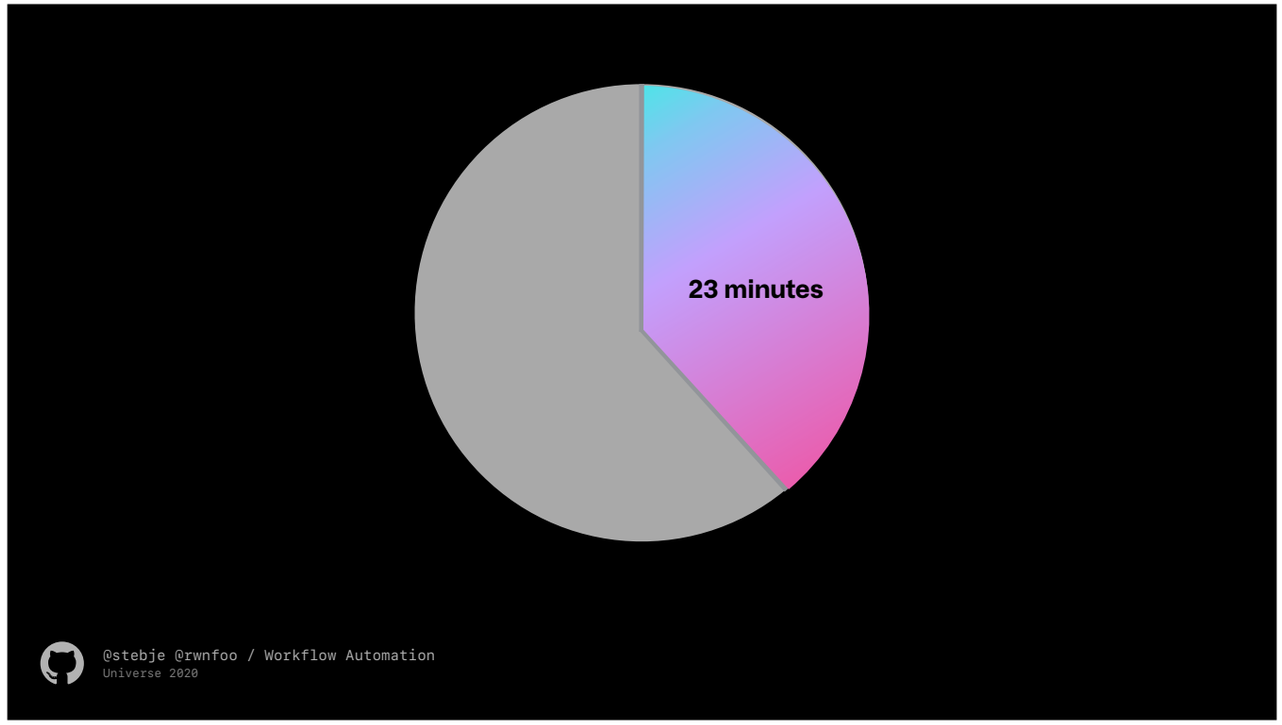
GitHub Secret store

GitHub Packages

How automation can help improve your own Software Development Lifecycle

The building blocks of GitHub Actions and GitHub Packages

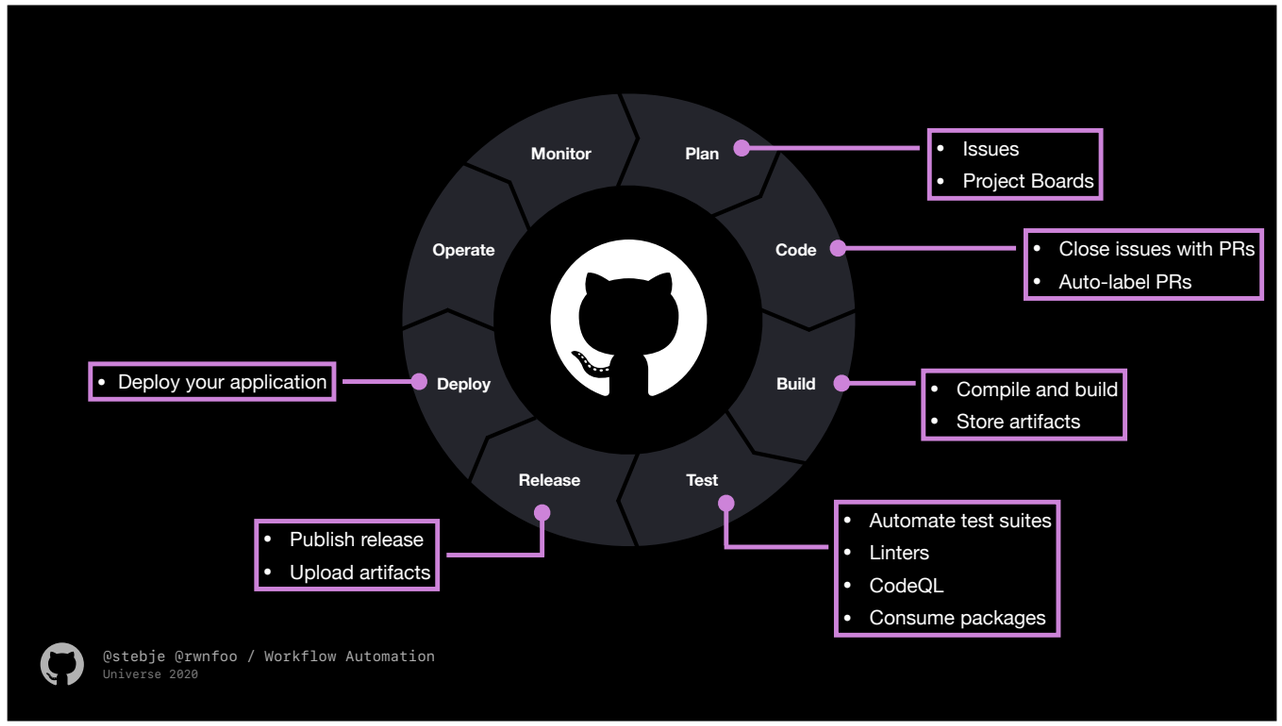
Build your own workflow! 🚀 🚀



If you were here during Universe, you might have heard that it takes about 23 minutes to get into that zen flow, 23 minutes to be in the zone while you're cranking work away.
Once you're distracted by something else, it takes another 23 minutes to enter it again.

So it's imperative to reduce the noise and reduce tasks that forces developers to context switch.

This diagram shows the entire SDLC which many of you may be familiar with.
Along this lifecycle, developers may have to switch between different tools and tasks, why not try and implement automation at every phase?
Let the bots do the work for you so you can focus on developing code and just generally make life easier!



In short, GitHub Actions allow you to build CI and CD capabilities directly in your repository instead of using CI/CD integrations. This gives you the freedom to automate and accelerate your processes all along your software development life cycle

Let’s dive into Actions and maybe you’ll have a better idea

GitHub Actions

@stebje @rwnfoo / Workflow Automation

Universe 2020

What are GitHub Actions?

- Automate, customize, and execute your software development workflows
- Stored as code
- Live logs and visualized workflow execution
- Matrix Builds
- Option to use GitHub-hosted runner or use your own (Self-hosted runner)

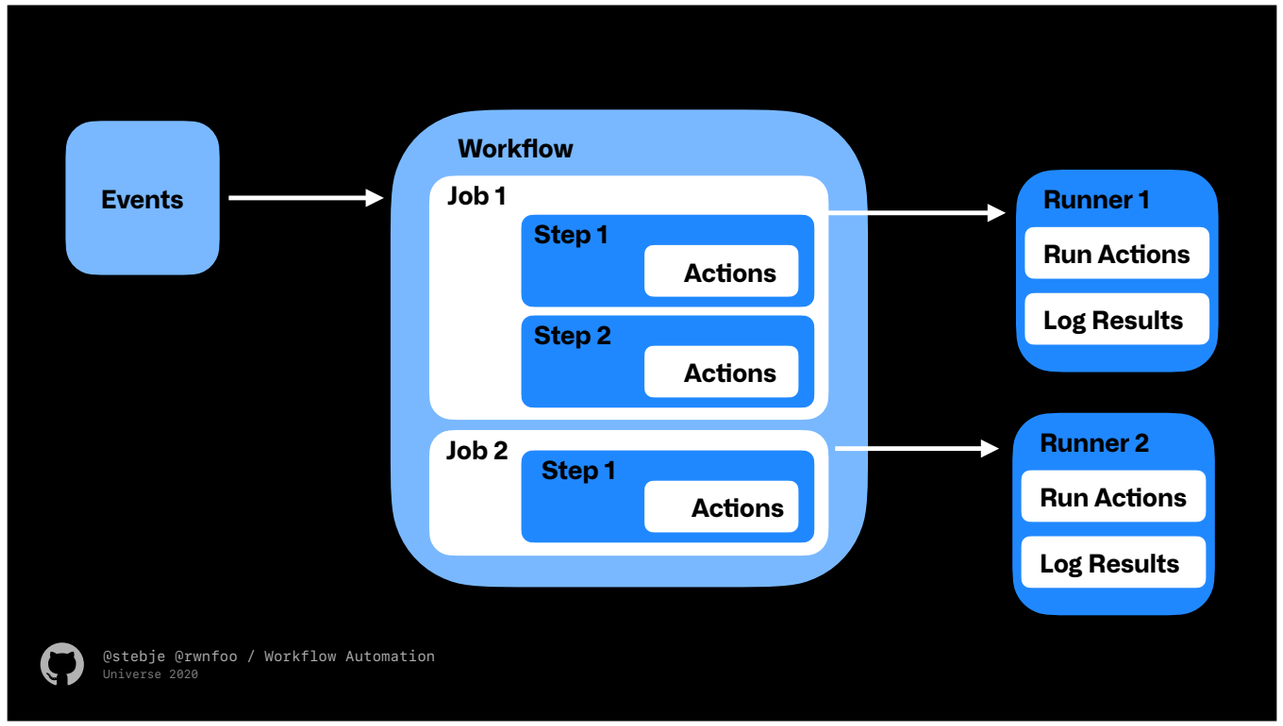
At a high level, GitHub Actions allow you to automate, customize and execute your workflows

Stored as code: you can version control, share, reuse it to allow for easy collaboration

Live logs increases transparency for developers and provide instant feedback for developers; reduces silos and developers can avoid switching to another tool to check the logs
And I'm sure by now, you may have heard of the announcement during universe - you can now see a visualization of your workflow in your repository

You can use a build matrix if you want your workflow to **run tests** across multiple combinations of operating systems, platforms, and languages all in the same workflow.

You also have the option to our own serves, called GitHub hosted runners or use your own, which is what we called self-hosted runners



At a high level, here are the components of GitHub Actions

An event is a specific activity that triggers a workflow.

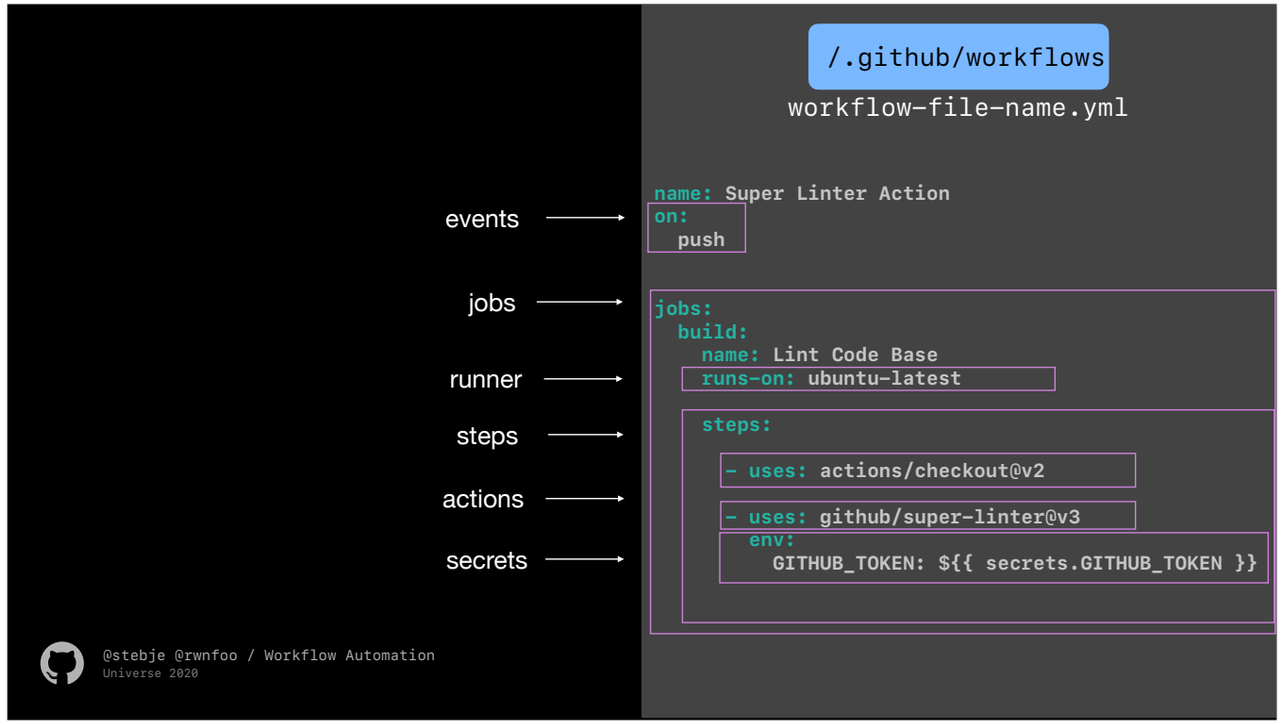
Workflows are made up of one or more jobs.

A job contains a set of steps, each step is an individual task that can run commands in a job.
A step can be either an action or a shell command.

Actions are then referenced in the step.
It is the smallest building block of a workflow.
The Action that you reference is also portable, meaning you can

A runner is the environment where the job is executed. Each step in a job executes on the same runner, allowing the actions in that job to share data with each other.
You can use a runner hosted by GitHub, or you can host your own.

A runner listens for available jobs, runs one job at a time, and reports the progress, logs, and results back to GitHub.



Let’s take a closer look into how GitHub Actions take place in practice

You would start off with a YAML file called workflow files. Workflow files codifies your process

They are stored in a folder in your repository called .github/workflows

In your workflow file, you would have the following syntax where you would declare the events ……

If you are familiar with jenkins file or Travis.yml, the syntax is pretty similar

events →

jobs →

runner →

steps →

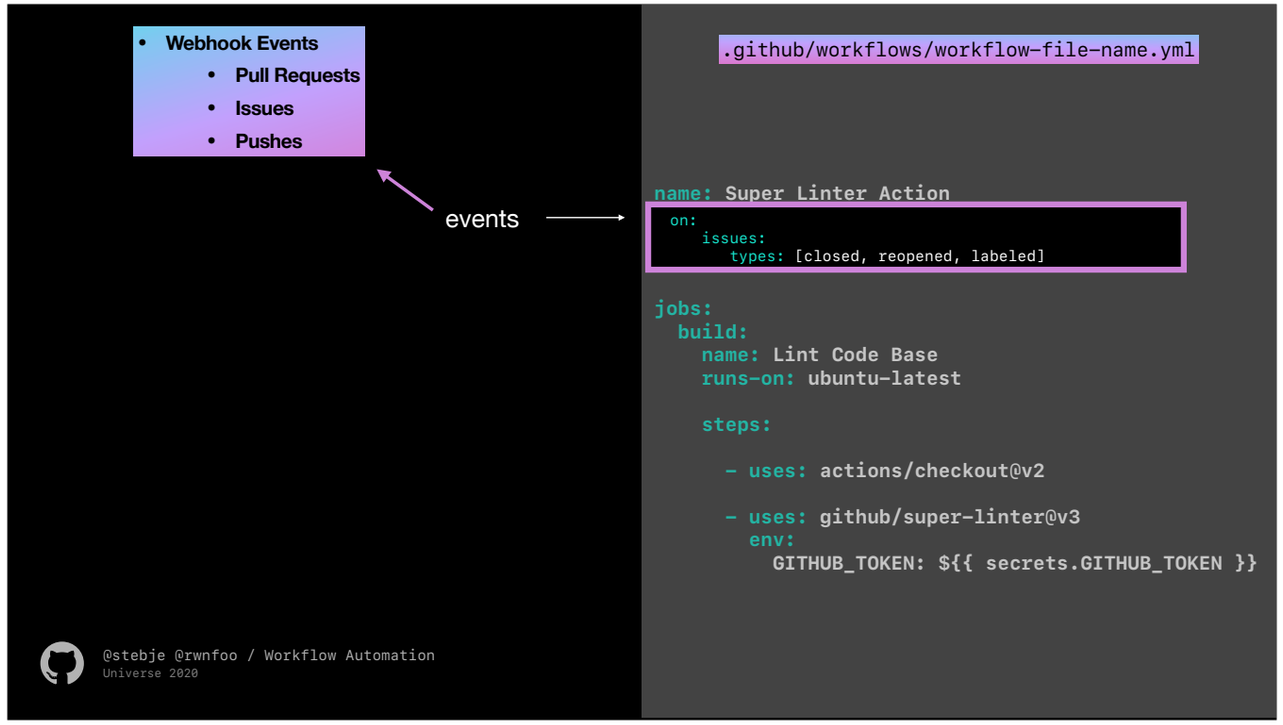
actions →

secrets →

`/.github/workflows`

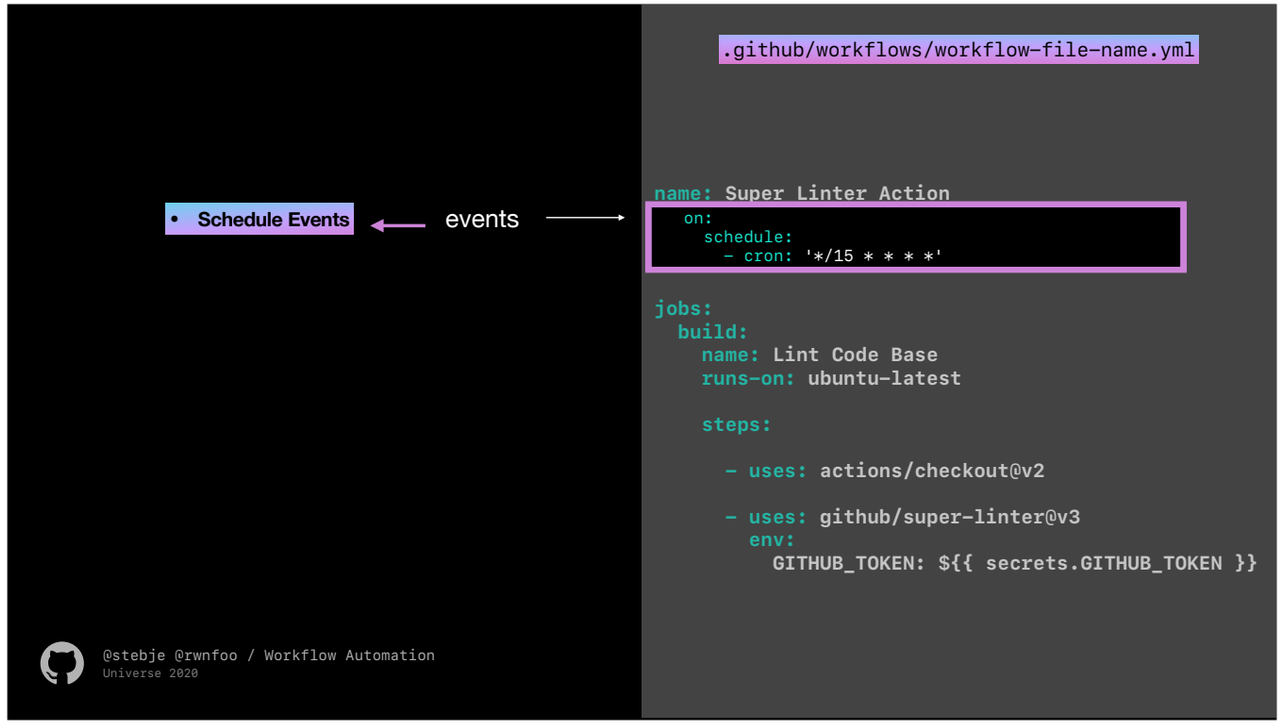
`workflow-file-name.yml`

```yaml
name: Super Linter Action
on:
  push

jobs:
  build:
    name: Lint Code Base
    runs-on: ubuntu-latest

    steps:

      - uses: actions/checkout@v2

      - uses: github/super-linter@v3
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```



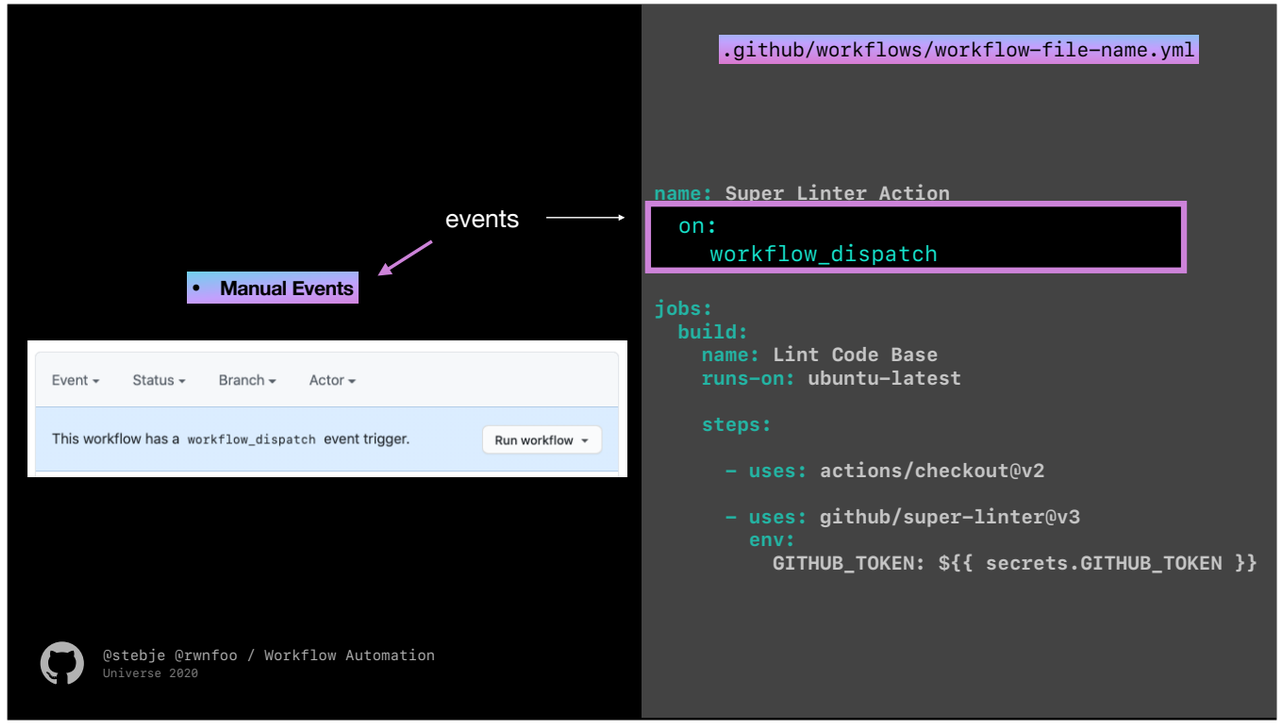
The first type of events are GitHub webhook events. For example, when someone creates a pull requests, or labels an issue or pushes to the repository

Manual Events
Let’s say none of those work for you, and you’d like something more manual -BYou can use the workflow_dispatch event, your action now comes with a button that allows you to run the action manually



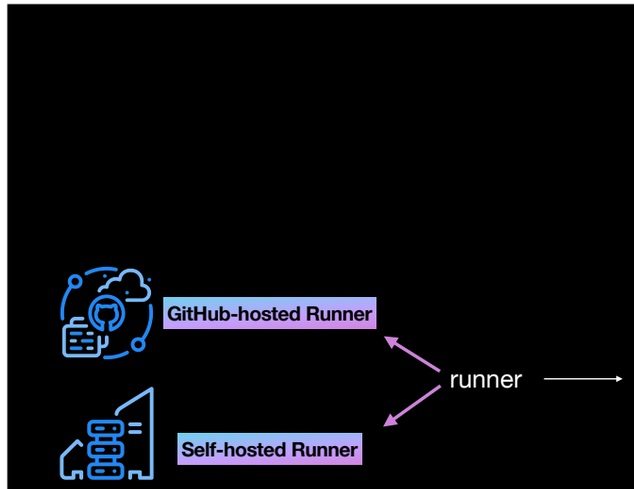
The first type of events are GitHub webhook events. For example, when someone creates a pull requests, or labels an issue or pushes to the repository

Manual Events
Let’s say none of those work for you, and you’d like something more manual -BYou can use the workflow_dispatch event, your action now comes with a button that allows you to run the action manually



The first type of events are GitHub webhook events. For example, when someone creates a pull requests, or labels an issue or pushes to the repository

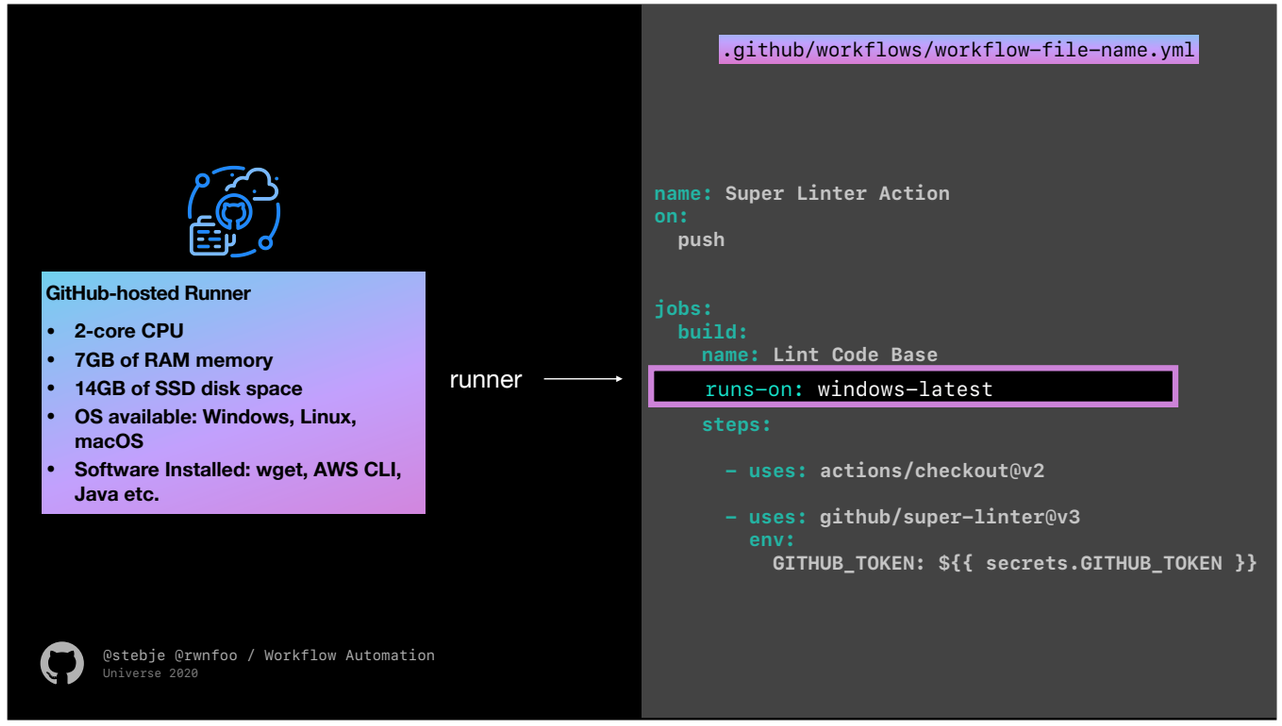
Manual Events
Let’s say none of those work for you, and you’d like something more manual -BYou can use the workflow_dispatch event, your action now comes with a button that allows you to run the action manually



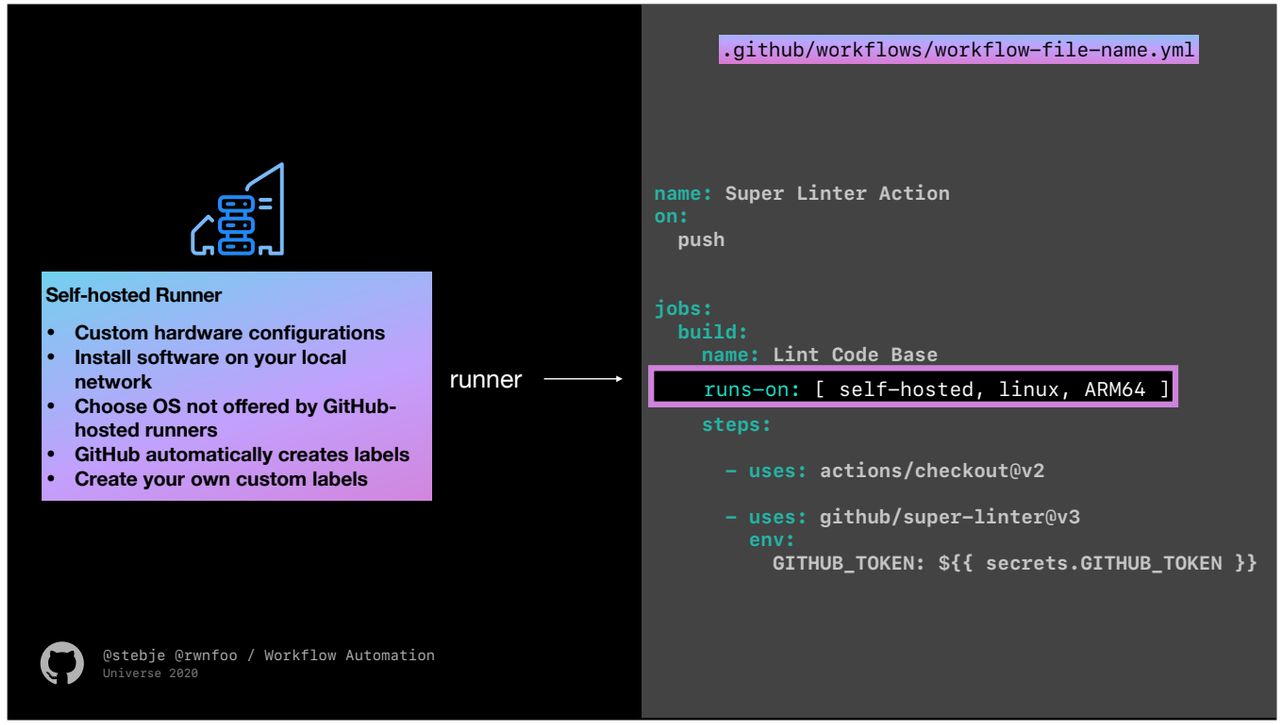
@stebje @rwnfoo / Workflow Automation
Universe 2020

.github/workflows/workflow-file-name.yml

```yaml
name: Super Linter Action
on:
  push

jobs:
  build:
    name: Lint Code Base
    runs-on: ubuntu-latest

    steps:

      - uses: actions/checkout@v2

      - uses: github/super-linter@v3
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```



GitHub Hosted runners:
comes with the following hardware configurations and operating systems
and also tons of software installed such as…



Self Hosted runners:
On the other hand, you might want to go with self-hosted runners if you require other types of configurations, for example if you need more processing power or memory of your hardware to run larger jobs
This is a good option if running actions in your own private network is important to you

GitHub creates labels for your self hosted runners automatically

Responsible for the costs of your own servers

Reusable units of code that can be referenced in a workflow

GitHub runs them in Node.js runtime or in Docker containers

3 types of actions you can reference:

- A public repository
-
-

actions →

.github/workflows/workflow-file-name.yml

```yaml
name: Super Linter Action
on:
  push

jobs:
  build:
    name: Lint Code Base
    runs-on: ubuntu-latest

    steps:

    - uses: actions/checkout@v2
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Reusable units of code that can be referenced in a workflow

GitHub runs them in Node.js runtime or in Docker containers

3 types of actions you can reference:

- A public repository
- The same repository as your workflow (local actions)
-

actions →

.github/workflows/workflow-file-name.yml

```yaml
name: Super Linter Action
on:
  push

jobs:
  build:
    name: Lint Code Base
    runs-on: ubuntu-latest

    steps:

      - uses: ./action-a
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Reusable units of code that can be referenced in a workflow

GitHub runs them in Node.js runtime or in Docker containers

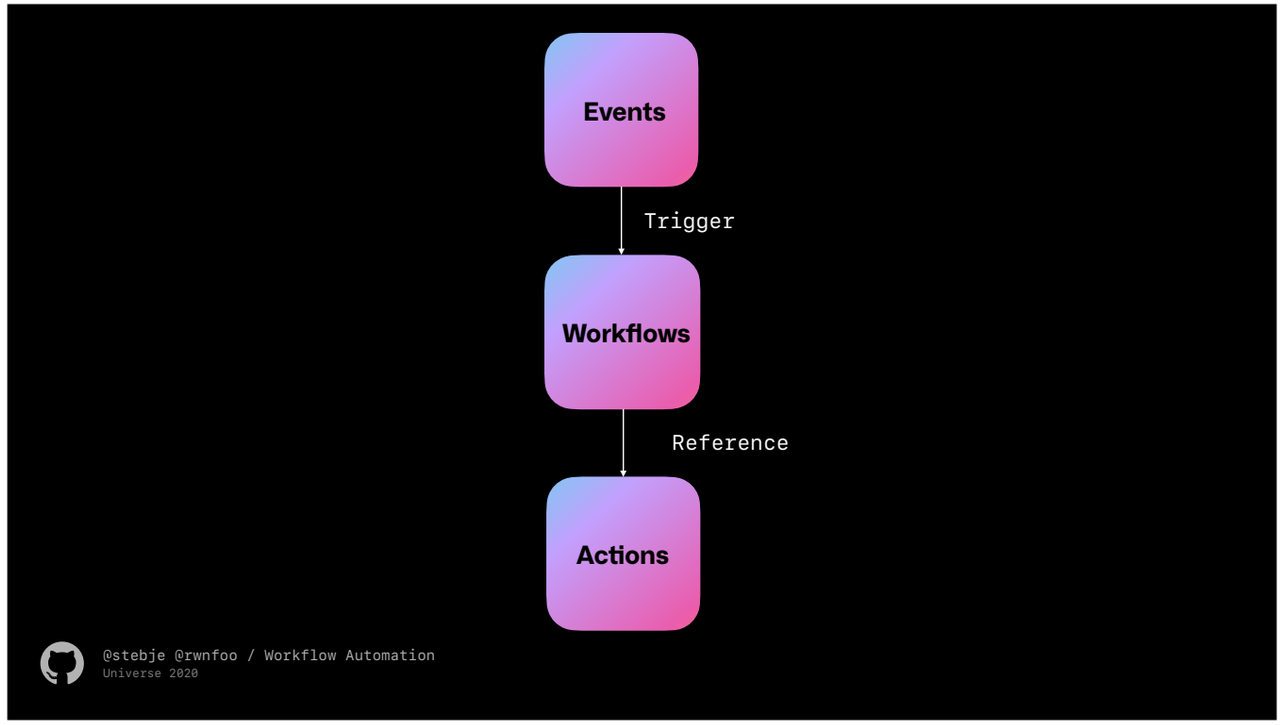
3 types of actions you can reference:

- **A public repository**
- **The same repository as your workflow (local actions)**
- **A published Docker container image on Docker Hub**

actions →

`.github/workflows/workflow-file-name.yml`

```yaml
name: Super Linter Action
on:
  push

jobs:
  build:
    name: Lint Code Base
    runs-on: ubuntu-latest

    steps:

      uses: docker://alpine:3.8
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

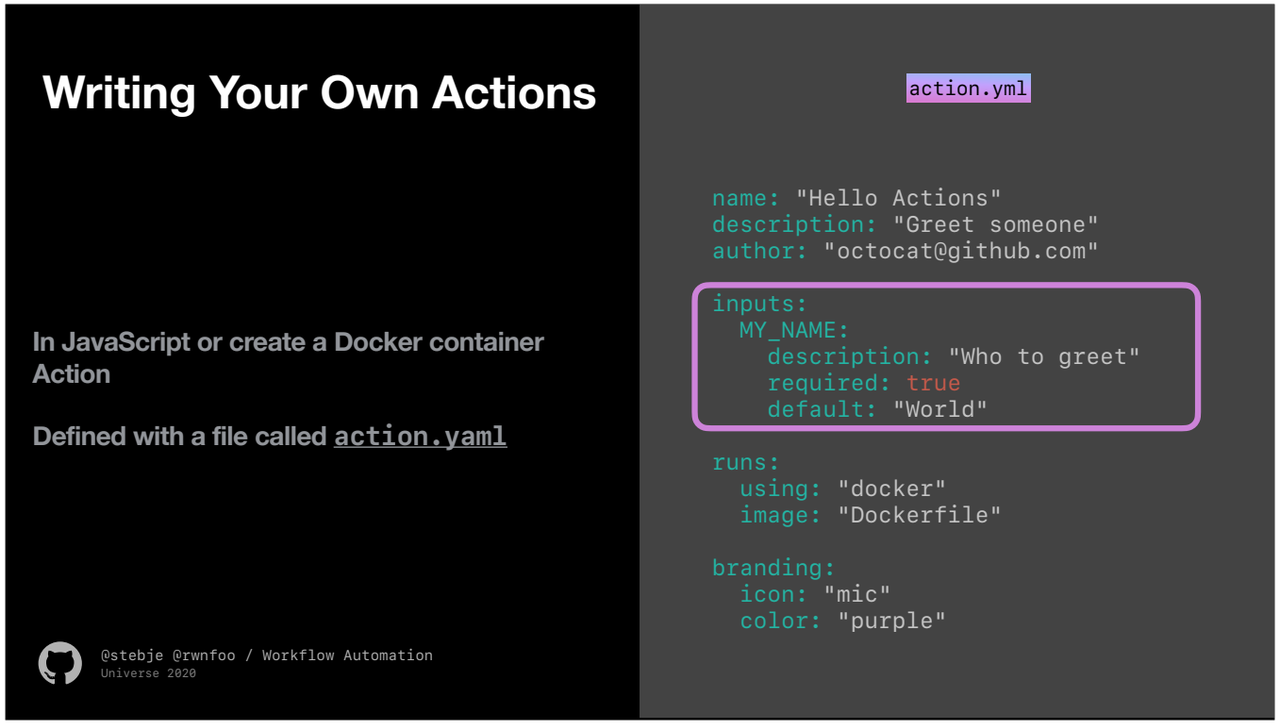


When people say GitHub Actions - it is usually in reference to all three components. It is not limited to the last component of Actions itself

If you’re trying to build your Action, we will talk a little bit about that later on but that’s not going to be the focus of this workshop

Your primary learning goal from today’s workshop is just to learn how to edit the workflow YAML file, and we will be using existing Actions that were built by others

But just to entertain your curiosity, let’s dip our toes a little bit into understanding what it’s like to write your own actions



You can choose to write them in JavaScript or create a container Action

Docker Actions:
Because a container allows you to have specific environments, like specific versions of dependencies and tools,
A Docker container actions is the ideal option if your actions must run in a specific environment configuration.

Anyone who is consuming your action does not need to worry about any of the dependencies, which is great!

Javascript Actions:
On the other hand, JavaScript actions can run directly on a runner machine
Using a JavaScript action simplifies the action code therefore it executes faster than a Docker container action.

action.yml
Actions are defined with a metadata file called action.yml.
The data in the metadata file defines the inputs, outputs and main entrypoint for your action.

Inputs (optional to declare)

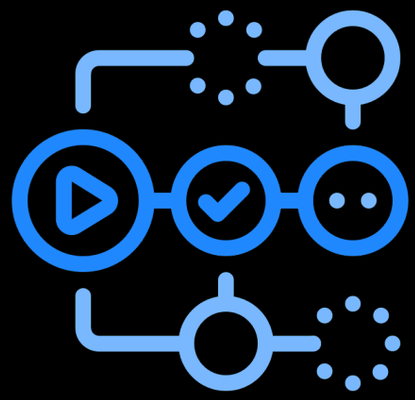
In this example, my action.yaml file has an input parameter
Input parameters allow you to specify data that the action expects to use during runtime.
GitHub stores input parameters as environment variables.

inputs. required
boolean that tells this action if this input is required or not

outputs (optional)
One thing you don’t see here, is an output parameter.
Output parameters allow you to declare data that can be used later in a workflow
If you don’t want to declare the output parameter in the action.yaml file, you can also declare it in the workflow file.

Best Practices for Actions

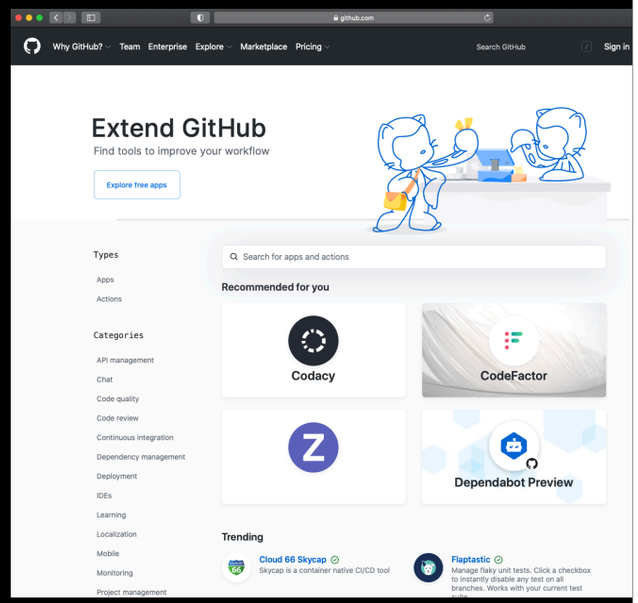
- Chainable actions vs. monolithic actions
- Versioning
- Documentation: `README.md` & `CONTRIBUTING.md`
- Proper `action.yml` metadata
- Add your action to the Marketplace!



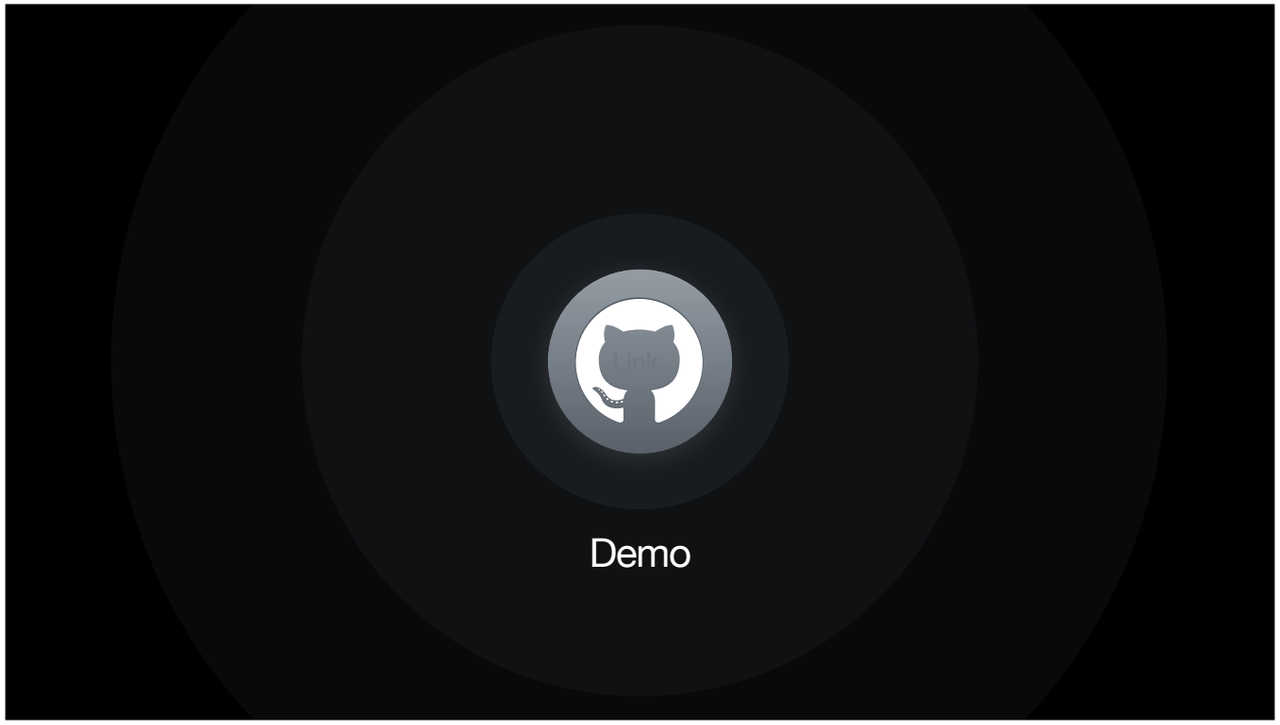
If you are thinking of writing and creating your own action, here are some things you want to consider

we recommend using release management to control how you distribute updates. users should not be referencing an action's default branch

Third-Party Actions



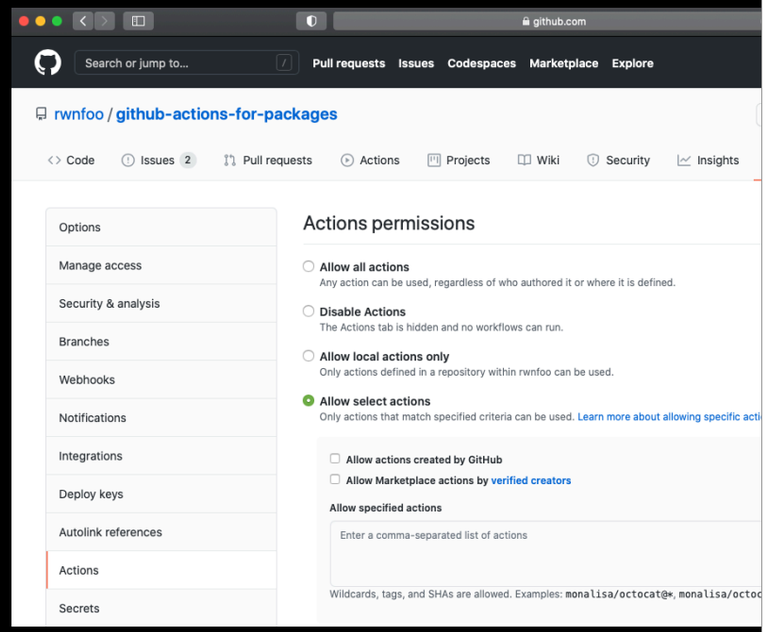
@stebje @rwnfoo / Workflow Automation
Universe 2020



Workflow syntax in help docs: https://docs.github.com/en/free-pro-team@latest/actions/reference/workflow-syntax-for-github-actions#about-yaml-syntax-for-workflows

https://docs.github.com/en/free-pro-team@latest/actions/creating-actions

Managing Permissions



@stebje @rwnfoo / Workflow Automation
Universe 2020

GitHub Secrets

@stebje @rwnfoo / Workflow Automation

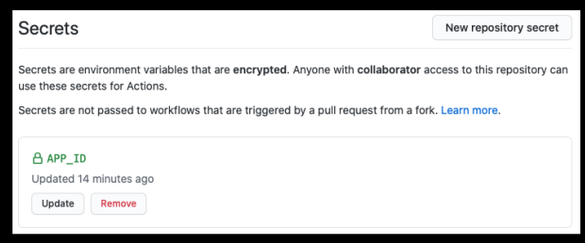
Universe 2020

GitHub Secrets

Built-in secret store

Encrypted

API support

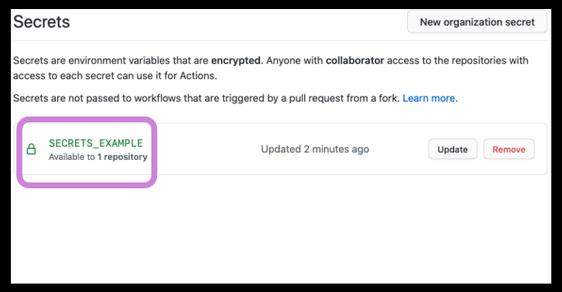


GitHub Actions come with a built-in secret store.
Secrets are encrypted environment variables that you create in a repository or organization

Secrets are encrypted before they reach GitHub, and remain encrypted until you use them in a workflow.

you can manage secrets through the API, you can create, update, delete, and retrieve information about encrypted secrets.

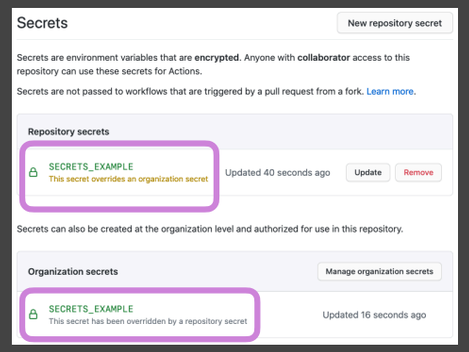
Organization Secrets



- Secrets managed at the organization level
- Share across repositories
- Can be scoped to specific repositories

@stebje @rwnfoo / Workflow Automation
Universe 2020

Repository Secrets



- Scoped to a repository
- Can override organization secrets
- Available on free plan

Organization-level secrets let you share secrets between multiple repositories so you don’t need to create duplicate secrets. Update just 1 place instead of multiple places

Using secrets in a workflow

All secrets can be accessed using the same syntax

`${{ secrets.<SECRET_NAME> }}`

Every workflow creates a `GITHUB_TOKEN` secret

- Authenticate on behalf of GitHub App
- Fixed scope and permissions

`${{ secrets.GITHUB_TOKEN }}`

Marketplace Actions exist for other secret stores

`.github/workflows/workflow-file-name.yml`

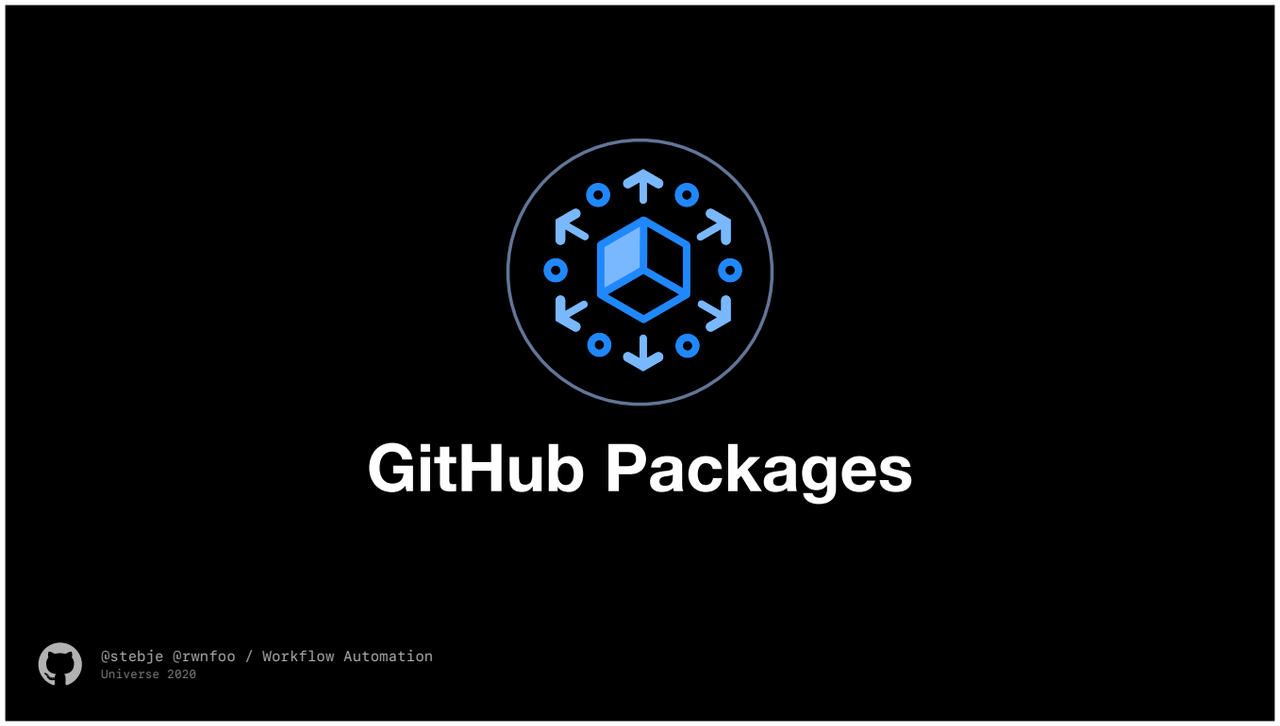
```yaml
name: Pull request labeler

on:
- pull_request

jobs:

  triage:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/labeler@v2
      repo-token: ${{ secrets.GITHUB_TOKEN }}

    - uses: myAction@v1
      with:
        mySecret: ${{ secrets.MY_SECRET }}
```

Once you store your secret either at the repository or organization level, you can accessed it by referencing the name of your secret name using the this syntax

On the other hand,
GitHub also automatically creates a token that you can use in your workflow
It does this by installing a GitHub App on your repository, and this token is used to authenticate on behalf of the GitHub App installed on your repository
So this means that the token's permission is limited to the repository that contains your workflow

When you run your workflow, GitHub fetches this token for the job, and the token expires when the job is finished

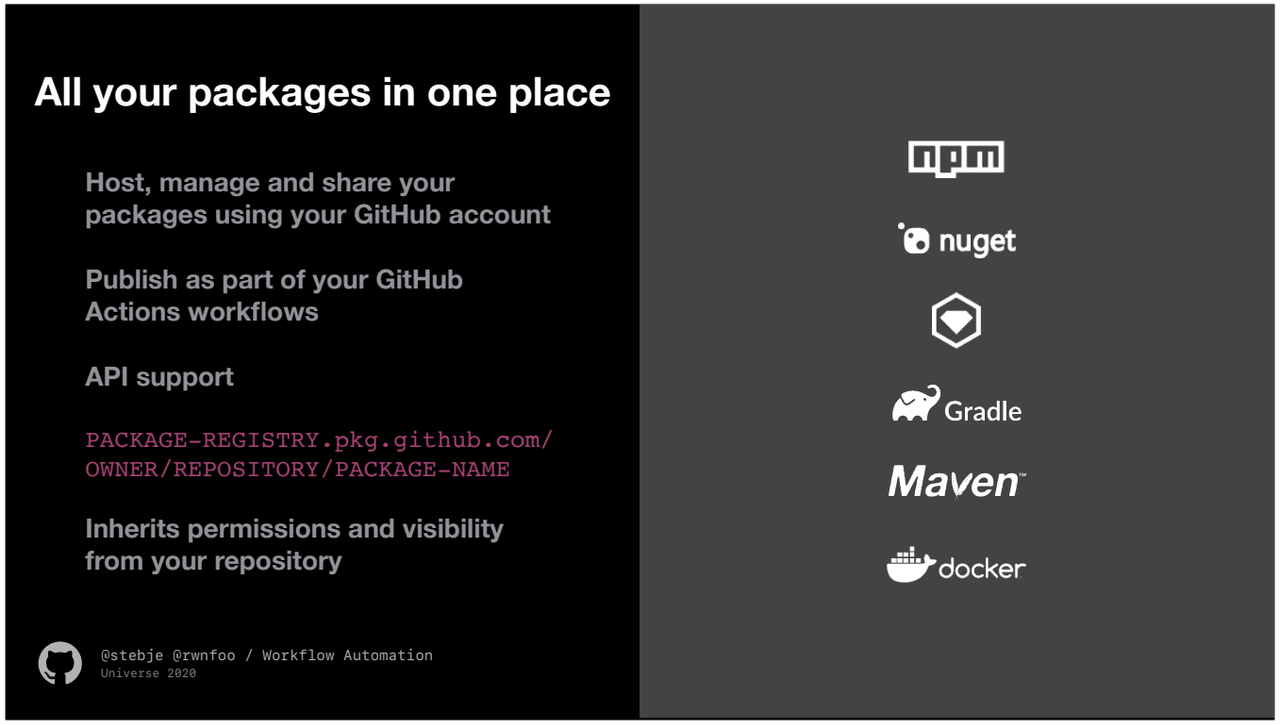
If you already have existing secret stores, like Azure key vault or HashiCorp Vault, you can definitely continue to use these secret stores using their Actions on the marketplace



We’ve looked at how to build up an automated pipeline using GitHub Actions.

Another aspect of software development is how to package and distribute your code, either as an open source project, inner source project, or in a smaller team, so that others can reuse your code including its metadata such as dependencies.

Most people are familiar with Package managers and use that actively as part of their SDLC, and for many of these package managers you can also use GitHub Packages.



Most will have familiarity with different package managers, like the ones listed on this slide.

Host, manage and share your packages using your GitHub Account
- No need to manage multiple account, and keep your packages together with your source code
- Leverage existing features such as repo access permissions to ease who can access your packages
- Can host multiple packages in the same repo

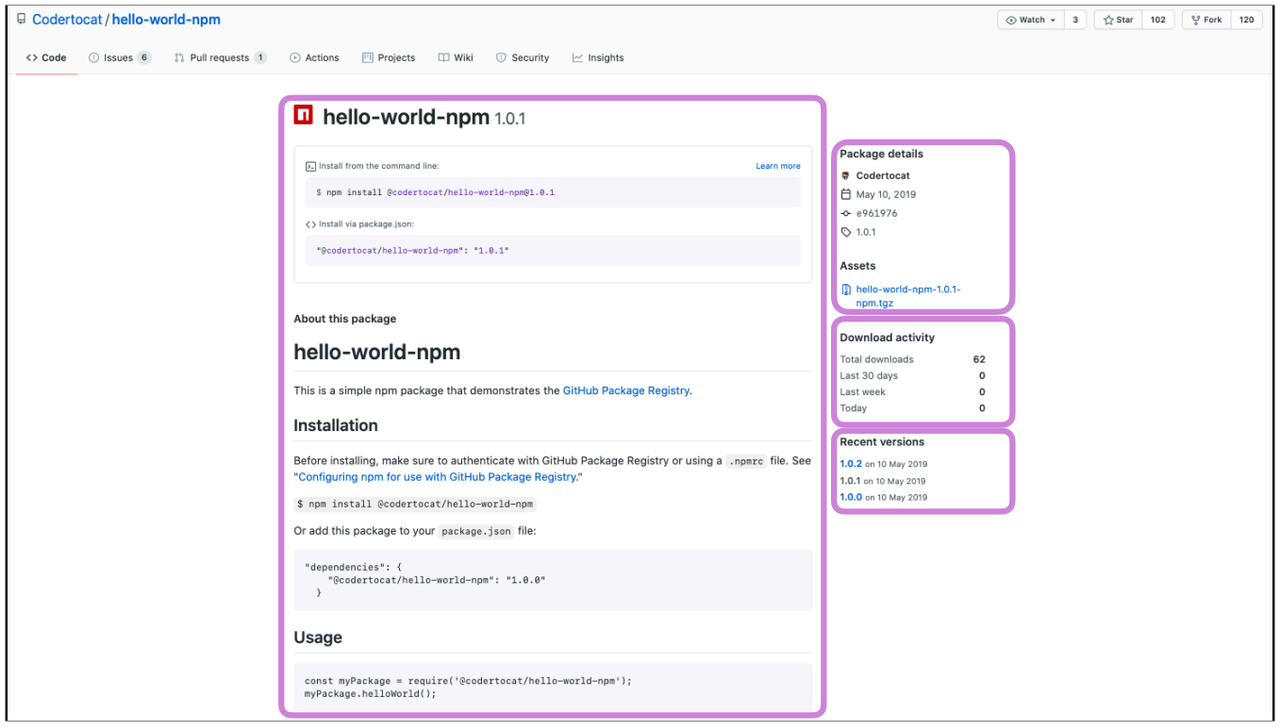
Publish as part of your GitHub Actions workflow
- Build and publish your package using a community action or build your own
- You can also build a workflow that triggers on a package event:
    - registry_package event for GitHub Actions (listen to when a package is published or updated)

API support
- Supported by our GraphQL API (read and delete)
- REST API (v3) and our GraphQL API (v4)
    - You can retrieve info about a package, end even delete specific package versions using GraphQL

Inherits permissions and visibility from your repository
- E.g. if I package my source code from a private repo, then the package will also be private, and only users with the appropriate permissions will be able to access that package
- Same goes for public repos, this will be a public package

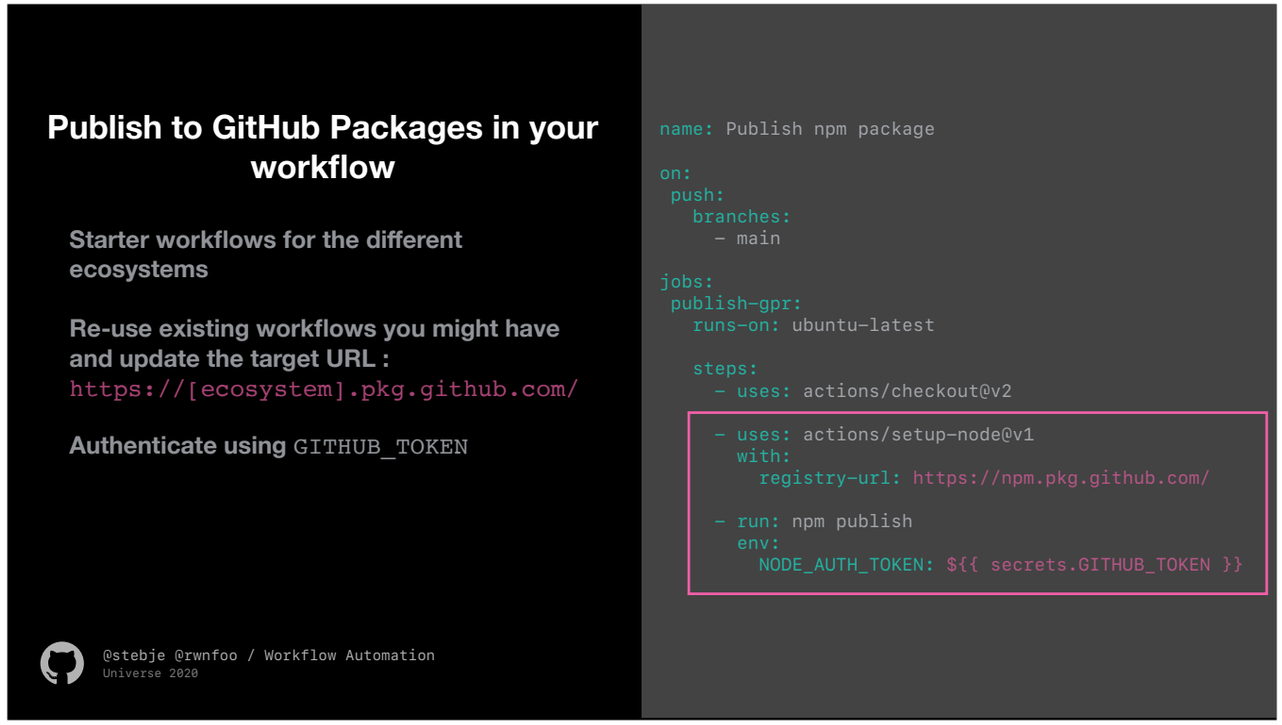
Note on missing support for Python, PHP, and generic packages
- Check the public roadmap

Note on Docker, will get to that in a later slide



Things to mention about the GUI

- Similar metadata view as you would find on npm
- Code snippets
- Initiates the package description from the README by default
- Right hand side
    - Creator of the package
    - Creation date/time
    - SHA fo the corresponding commit
    - Version number
    - Tarball file with the packaged code
    - Download activity
    - Version overview



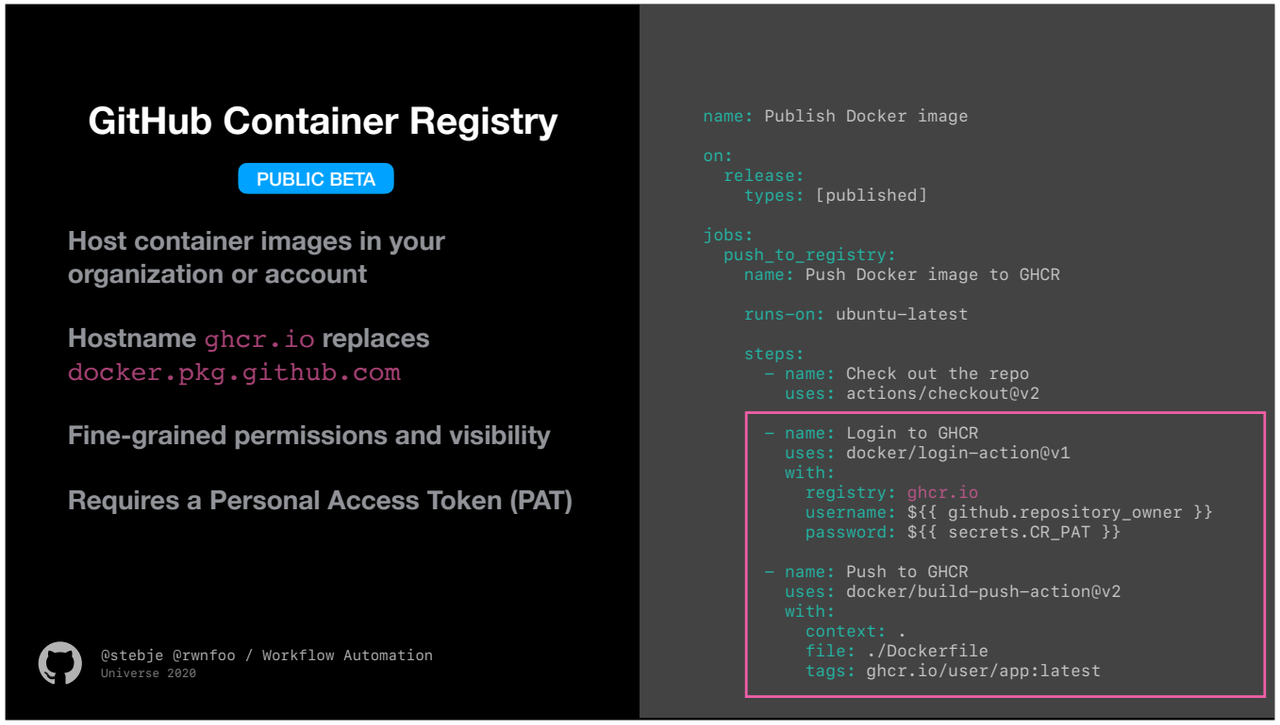
So, how to use GitHub Packages from within your workflow?
- It is pretty easy, and if you're coming from an existing package manager like npm you could quickly switch over to using Package by using one of the existing starter workflows and updating the target URL.

Starter workflows for different ecosystems
- node, maven, gradle, …
- As shown earlier, you can easily access these starter workflows via the Actions tab in your repo

Re-use existing workflows you might have
- Target URL for GitHub Packages must be updated if you're migrating from your existing package manager
- Or, you can of course publish to both registries in the same workflow!

Authenticate using GITHUB_TOKEN
- The auto-created GITHUB_TOKEN already has the appropriate permissions to publish packages, no additional credentials necessary

(See code snippet on the right)
- This workflow is publishing an npm package using one public Action which sets up my node environment including the target URL
- The second step runs a npm command directly in the workflow to publish my package using the GITHUB_TOKEN as a credential
    - GitHub hosted runners come with a preconfigured set of software, and since npm is one of them I can run npm commands directly in my workflow



We talked about Packages, what about containers?

Host container images in your organisation or account
- You'd also want to host your containers together with your code. This was already possible before using GitHub Packages, but has been separated into Container Registry to address the specific needs for containers and was released as public beta on September 1st.
- For those not familiar with containers;
    - …lightweight, executable bundle of software e.g. tools, libraries
    - …fast to spin up and deploy, often contrasted with the alternative of using VMs which is heavier and slower due to bundling of the OS
    - …standalone, i.e. "containerized"

New hostname
- If you used GitHub Packages Docker Registry before, you must change the target URL

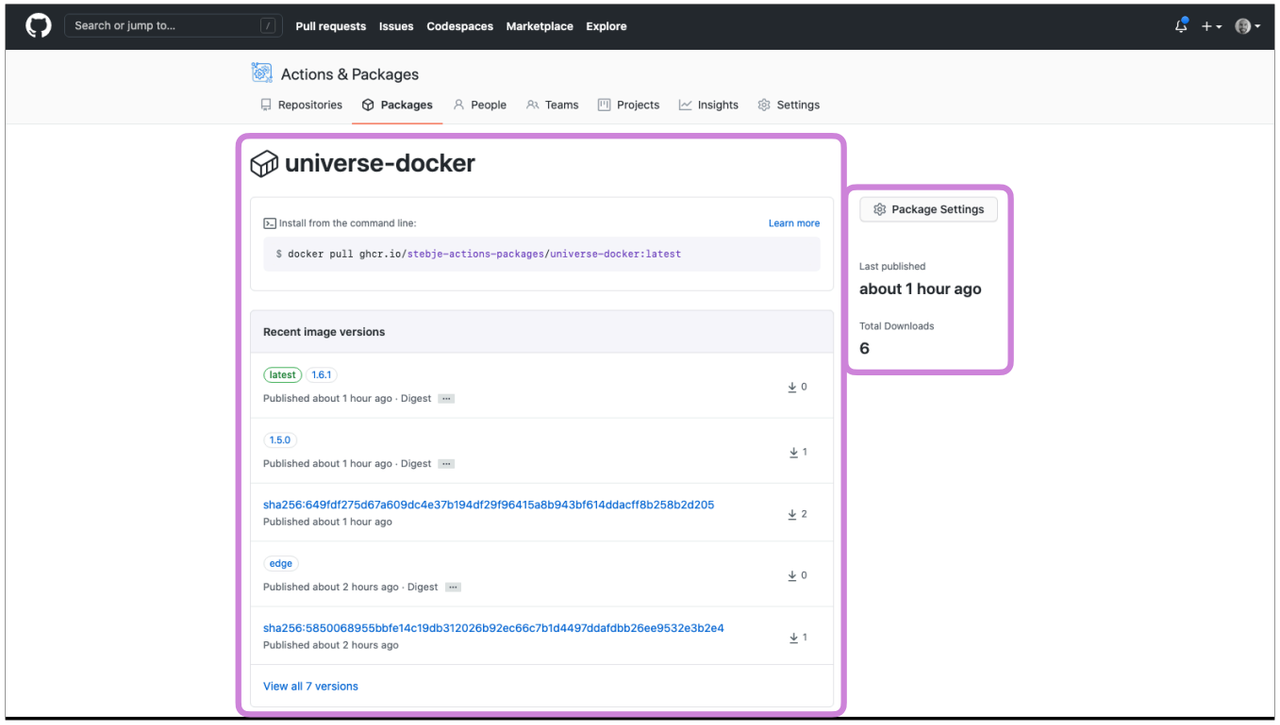
Fine-grained permissions and visibility
- Control whether container image is public or private
- Invite teams or users to your container image

Requires a PAT
- As opposed to Actions, GHCR requires a PAT, GITHUB_TOKEN cannot be used
- PAT Requires the appropriate permissions (read/write/delete packages)

(Code snippet on the right)
- Two actions, both published by Docker
- The first one logs into GHCR using the repository owner as username and (as pointed out) a PAT as password
- The second one builds pushes the container image, taking a few arguments

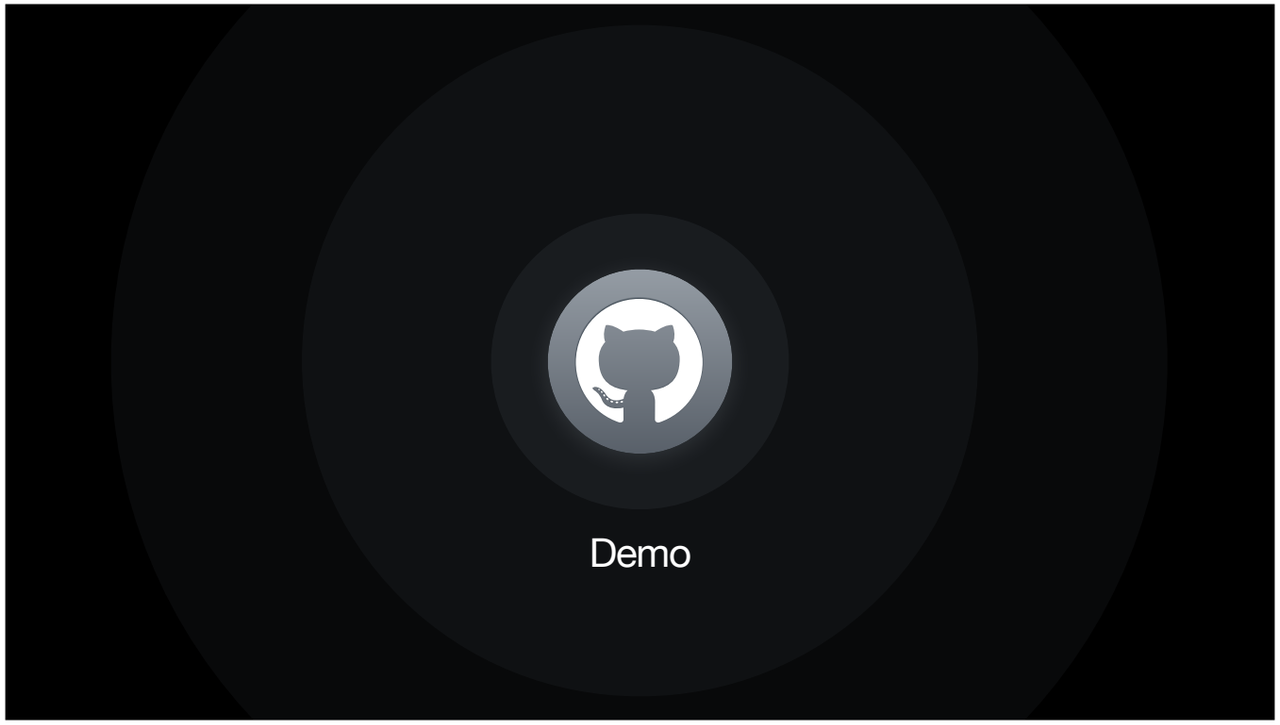
  - Context: which context to use, default is the git context
  - File: Where to find the Dockerfile
  - Tags: which tags to add to your container image

Additional remarks
- Anonymous access for public container images
- Can of course also push and pull images from the terminal
- Can publish to multiple container registries in the same workflow, e.g. to both DockerHub and GHCR



Things to mention about the GUI

- Similar metadata view as for packages
- Code snippets
- Versions
- Package settings
- Will show more during the upcoming demo



DEMO

- Show a workflow publishing an npm package
- Show the workflow file “Publish GPR”
    - Point out the trigger and the actions used
    - Point out running commands directly in your workflow file: https://github.com/actions/virtual-environments
- Show the package.json file
    - Point out the name and version
- Edit the package.json file
    - Update index.js code comment, save

```
git add .
git commit -S -m “<message>”
npm version patch
git log —oneline
git push
```

- Check that the workflow spins up
    - Workflow visualiser
- After run is finished, show the package
    - Show the metadata
    - Deleting versions
    - Updating package description
- Finally, install and use the npm package in a new node project
- Create a new folder in VSCode

```
npm init -y
npm install <package URL>
touch index.js
## check that the dependency is added to package.json—
## Add to index.js:
const <name> = require(“<package name>”)
## invoke the exported function
```

Show a workflow publishing an Docker image

- Show the workflow file “Publish GHCR”
    - Point out the trigger and the actions used
- Create and publish a new release from the tag created earlier
- Check that the workflow spins up
- After run is finished, show the container image
    - Show the metadata
    - Granting access
    - Updating visibility

Final remarks

- Accomplished all of these activities without leaving GitHub
- Most tasks taken care of by my automation
- Don’t have to context switch and lose time jumping between tools

Why GitHub Packages?

Your packages and containers together with your source code

Reduce overhead of managing multiple accounts

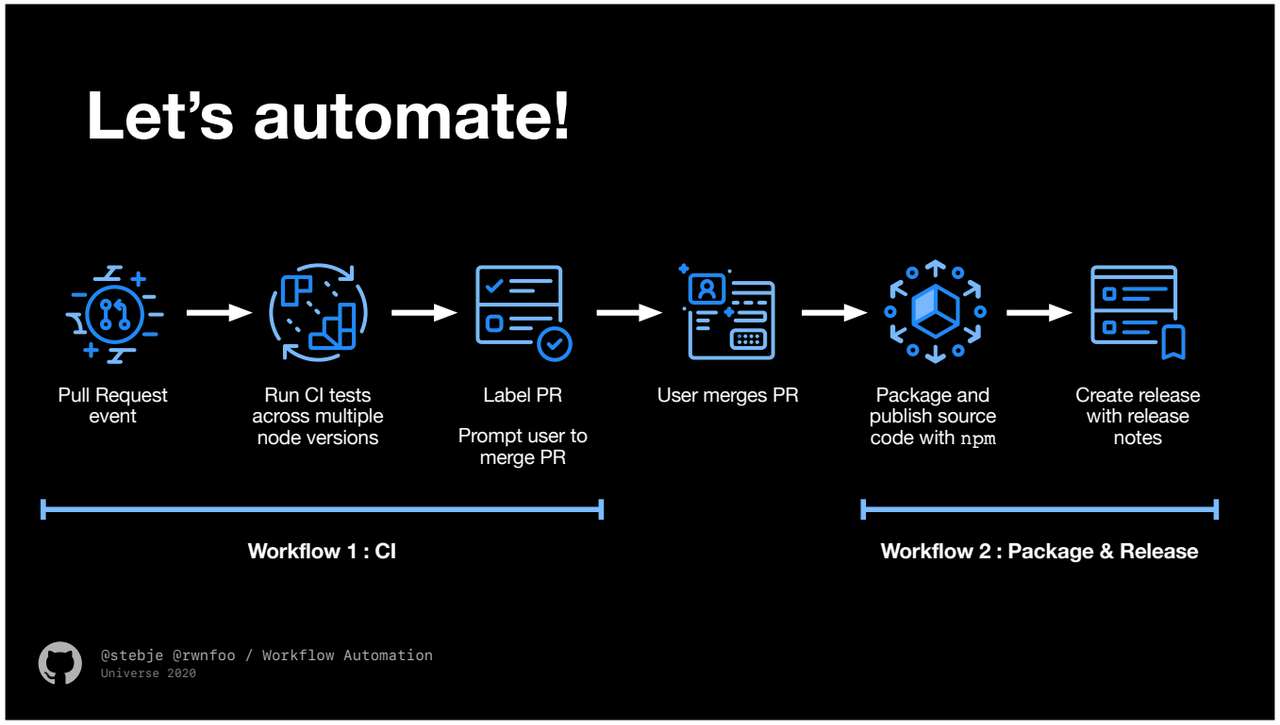
Ease sharing of code internally and externally

Integration with GitHub Actions

🎉 Hands-on 🎉

@stebje @rwnfoo / Workflow Automation

Universe 2020



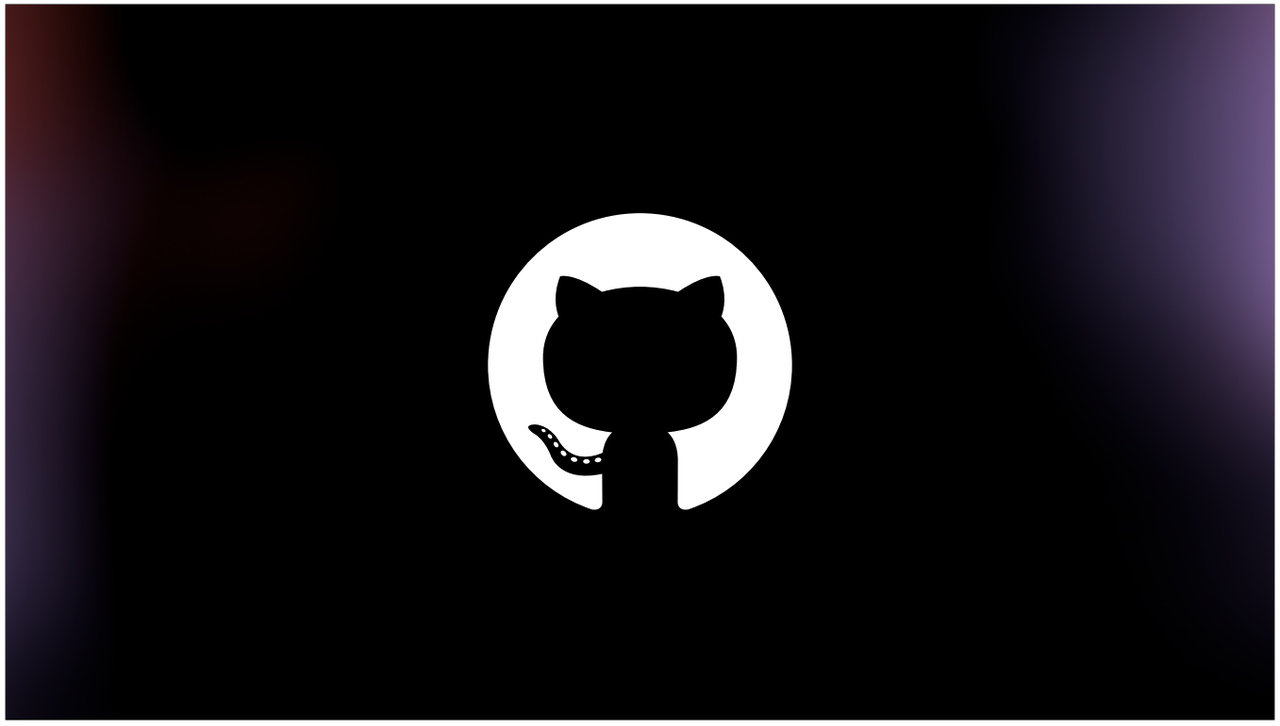
Workshop repo: https://github.com/githubuniverseworkshops/workshop-automate-your-workflow

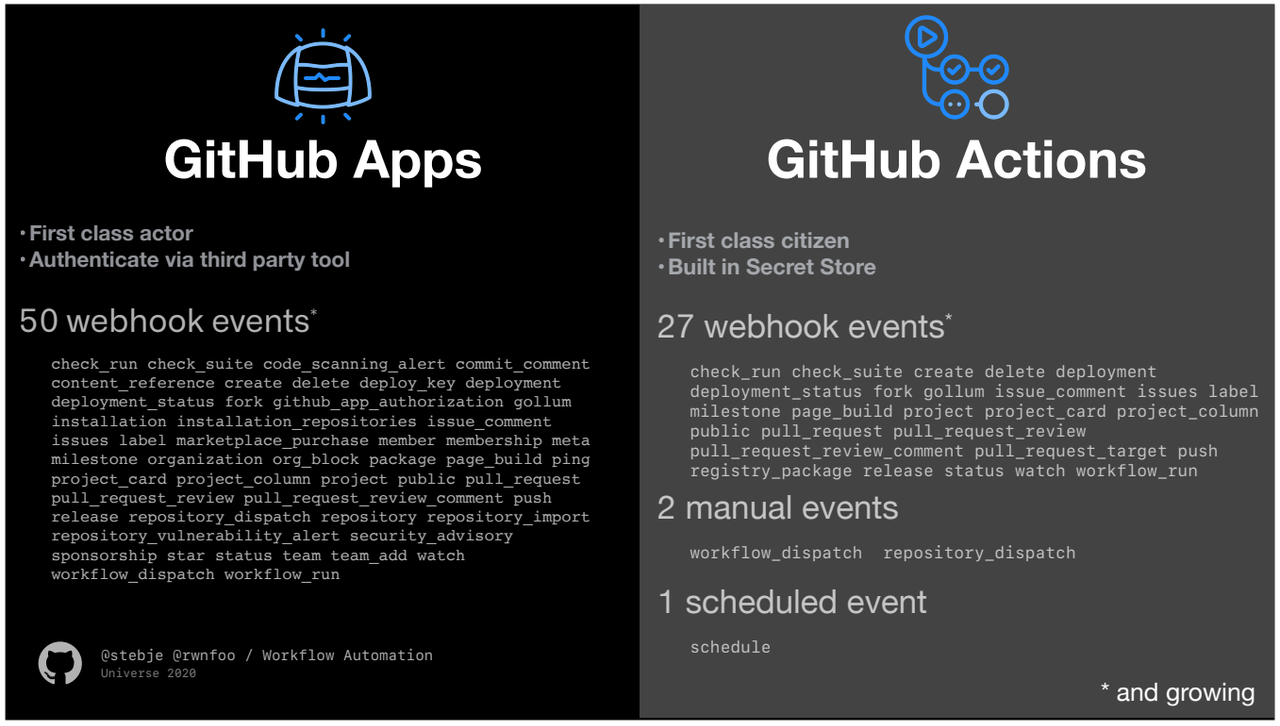
Summary

- Save time and reduce risk of human error by automating your SDLC
- Trigger your workflows on Events
- Use community Actions or build your own!
- Using Secrets
- Publish your code with GitHub Packages

Summary

- The future for Actions
  https://github.com/github/roadmap/projects/1?card_filter_query=actions
- Changelog
  https://github.blog/changelog/





Use GitHub App to authenticate Jenkins in order to run Jenkins pipelines

Actions allow you to run pipelines natively, e.g. whenever code is checked in

Strengths of GitHub Actions and GitHub Apps
While both GitHub Actions and GitHub Apps provide ways to build automation and workflow tools, they each have strengths that make them useful in different ways.

GitHub Apps:

Run persistently and can react to events quickly.
Work great when persistent data is needed.
Work best with API requests that aren't time consuming.
Run on a server or compute infrastructure that you provide.

GitHub Actions:

Provide automation that can perform continuous integration and continuous deployment.
Can run directly on runner machines or in Docker containers.
Can include access to a clone of your repository, enabling deployment and publishing tools, code formatters, and command line tools to access your code.
Don't require you to deploy code or serve an app.
Have a simple interface to create and use secrets, which enables actions to interact with third-party services without needing to store the credentials of the person using the action.

https://docs.github.com/en/free-pro-team@latest/actions/creating-actions/about-actions#comparing-github-actions-to-github-apps