Deploy without credentials with GitHub Actions and OIDC

This article was fetched from an rss feed(08/10/2021)

There's been some talk on Twitter recently about a new feature emerging on GitHub Actions. It allows an action to mint an OpenID Connect (OIDC) token, which can then be used to deploy artifacts into other systems and clouds.

I'll give you a bit of context, then show you the AWS and GCP story, followed by how I integrated this with OpenFaaS so that a set list of users on GitHub could deploy to my OpenFaaS cluster without having to give them any credentials.

What's the point of federation?

I rememeber seeing a keynote in 2018 at KubeCon where Kelsey Hightower demoed an integration between two services with one of them running on AWS and the other running on GCP. Normally that would mean producing a number of secrets and configuring them on either side. His demo was different because there was a level of trust between the two sides, fedration if you will.

The Oxford dictionary describes "federation" as:

> the action of forming states or organizations into a single group with centralized control.

Centralised control within a single group sounds useful and just like Kelsey showed us GCP and AWS working in harmony, later on I'll show you GitHub Actions deploying to OpenFaaS without any shared credentials.

> This video is definitely worth a watch, even if you were there live. He explains how serverless platforms play well with Kubernetes, integrating with managed cloud services.

Kelsey was probably using GCP's Workload Identity Federation technology.

AWS has a similar technology with its IAM (Identity Access Management), IdP (Identity Provider) and Secure Token Service (STS).

Why could credential sharing be an anti-pattern?

For some systems, credentials are tied to a human identity. We've all probably created a GitHub Personal Access Token PAT) once or twice in our time. These tokens are equivalent to us taking actions, and most of the time have very coarse-level permissions, i.e. "read/write to all repos".

If I were to use a PAT in an integration, then I left the company, my access token would still be in use, and my identity would be tied to that. On the other hand, if I left and deactived my account, the integration would stop working.

Even when API tokens and service accounts decouple identity from access tokens, there's still the need to share, store and rotate these secrets which presents a risk. The less of this we do, the lower the risk of something going wrong.

What does it look like?

Here's what the author of the GCP integration said:

Before federation:

1. Create a Google Cloud service account and grant IAM permissions
2. Export the long-lived JSON service account key
3. Upload the JSON service account key to a GitHub secret

After:

1. Create a Google Cloud service account and grant IAM permissions
2. Create and configure a Workload Identity Provider for GitHub
3. Exchange the GitHub Actions OIDC token for a short-lived Google Cloud access token

In short, the token and identity that GitHub Actions provides is enough to deploy to GCP or AWS when configured in this way. That means using the SDK, CLIs, Terraform and other similar tooling. It could probably also be made to work with Kubernetes authentication and authorization.

The GCP example

The GCP example is where I learned the most about how this works, since GitHub haven't documented the API yet.

> This GitHub Action exchanges a GitHub Actions OIDC token into a Google Cloud access token using Workload Identity Federation. This obviates the need to export a long-lived Google Cloud service account key and establishes a trust delegation relationship between a particular GitHub Actions workflow invocation and permissions on Google Cloud.

From reading the source code:

The Action is provided with two variables: `ACTIONS_ID_TOKEN_REQUEST_URL` and `ACTIONS_ID_TOKEN_REQUEST_TOKEN`

By posting `ACTIONS_ID_TOKEN_REQUEST_TOKEN` to `ACTIONS_ID_TOKEN_REQUEST_URL`, you receive a JSON response containing an undocumented `count` variable and a `value` property

Next, the resulting JWT token is used with Google's STS service to create a short-lived token for Google Cloud.

Then from what I read, that token can be used with the `gcloud` CLI and so forth.

Some prior configuration is required with a service account on the Google side.

See Seth Vargo's example here: @google-github-actions/auth

The AWS example

Aidan Steele appears to have done the heavy lifting here.

Aidan writes:

> GitHub Actions has new functionality that can vend OpenID Connect credentials to jobs running on the platform. This is very exciting for AWS account administrators as it means that CI/CD jobs no longer need any long-term secrets to be stored in GitHub. But enough of that, here's how it works:

He also mentions:

> At the time of writing, this functionality exists but has yet to be announced or documented. It works, though!

That's important to note, because right when I was in the middle of demoing an integration with OpenFaaS, the team changed a URL for where the tokens are issued from. It's probably not a good idea to put this straight into production, until its announcement, if it should make it to GA. I hope it does!

Aidan shares an example of the JWT he received from the OIDC endpoint if you'd like to poke around.

If you're an AWS customer, try Aidan's example here: AWS federation comes to GitHub Actions

> After writing this article, I reached out to Aiden who told me this was a feature he requested in early 2020, you can see the GitHub Support request here.

Making it work with OpenFaaS

Now both of the examples we've seen give you the ability to access the APIs of Google Cloud or AWS - that's very powerful, but also very broadly scoped.

I wondered if I could follow the same principles and make an authentication plugin for OpenFaaS Pro that would act in the same way.

First of all, I wrote a tiny HTTP server in Go, and deployed it on my machine to print out webhooks.

```
package main

import (
        "fmt"
        "io"
        "log"
        "net/http"
        "os"
)

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "OK")
        fmt.Fprintf(os.Stdout, "Method: %s Path: %s \nHeaders:\n", r.Method, r.URL.Path)

        for k, v := range r.Header {
                fmt.Fprintf(os.Stdout, "%s=%s", k, v)
        }

        if r.Body != nil && r.ContentLength > 0 {
                defer r.Body.Close()
                fmt.Fprintf(os.Stdout, "\nBody (%d bytes):\n", r.ContentLength)

                io.Copy(os.Stdin, r.Body)
        }
        fmt.Fprintf(os.Stdout, "\n")

}

func main() {
        http.HandleFunc("/", handler)
        log.Println("Starting server on 8080")
        log.Fatal(http.ListenAndServe(":8080", nil))
}
```

> Want to learn patterns and practices for writing idiomatic Go including microservices, CLIs and building Go via GitHub Actions? See my new book: Everyday Go

Then I ran an inlets tunnel which gives me a secure, private and self-hosted way to receive webhooks. I didn't want this token going through the Ngrok or Cloudflare tunnel shared servers.

```
inlets-pro http client --token=$TOKEN \
  --url=wss://minty-tunnel.exit.o6s.io \
  --upstream http://127.0.0.1:8080 \
  --license-file $HOME/.inlets/LICENSE \
  --auto-tls=false
```

Getting the first OIDC token

Next I wrote an Action to dump out environment variables. I was wondering if the token would already be minted and available.

Note that the `id-token: write` permission must be given in order to obtain an OIDC token from the built-in endpoint.

```
name: federate

on:
  push:
    branches:
    - '*'

jobs:
  auth:

    # Add "id-token" with the intended permissions.
    permissions:
      contents: 'read'
      id-token: 'write'

    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
        with:
          fetch-depth: 1
      - name: Dump env
        run: env
```

After running it, I noticed that it wasn't, but two things caught my eye:

- `ACTIONS_ID_TOKEN_REQUEST_URL`
- and `ACTIONS_ID_TOKEN_REQUEST_TOKEN`

So I decided to post that token to the given URL. Then with some searching around, I saw the Google Aaction did the same.

> Another bonus to hitting "." is that you get the search from VSCode which is easier to use and faster IMHO than using the GitHub UI
>
> This runs on your local machine, so no need to pay for or wait for a Codespace to launch. pic.twitter.com/Lbq8UnSs9E
>
> — Alex Ellis (@alexellisuk) October 6, 2021

Rather than searching in the UI, I hit `.` which is a new GitHub UI shortcut. It opens VSCode in a client-side only experience that has a much better search capability.

So then I ran the action:

```
name: federate

on:
  push:
    branches:
    - '*'

jobs:
  auth:

    # Add "id-token" with the intended permissions.
    permissions:
      contents: 'read'
      id-token: 'write'

    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
        with:
          fetch-depth: 1
      - name: Dump env
        run: env
      - name: Post the token
        run: |
          OIDC_TOKEN=$(curl -sLS "${ACTIONS_ID_TOKEN_REQUEST_URL}&audience=minty.exit.o6s.io" -H "User-Agent: actions/oidc-client" -H "Authorization: Be

          curl -i -s --data-binary "$OIDC_TOKEN" \
            https://minty.exit.o6s.io/github-oidc
```

Bear in mind that the `aud` field should always be set to the URL that will consume the token, in this case: `minty.exit.o6s.io`.

Within the microservice, after validating the JWT against the issuer's public key, you should validate then that the aud field is set as expected.

See also: JWT RFC 7519

Checking out the fields

To my surprise, it worked.

I saw a response like this:

```
{
  "count: 3118,
  "value": "VALID_JWT_TOKEN",
}
```

Next, I extracted the .value property and threw away the count. I still don't know what that integer means, I hadn't run that many token requests for instance.

Pasting the token into JWT.io gave me an output much like Aidan's.

```
{
  "actor": "aidansteele",
  "aud": "https://github.com/aidansteele/aws-federation-github-actions",
  "base_ref": "",
  "event_name": "push",
  "exp": 1631672856,
  "head_ref": "",
  "iat": 1631672556,
  "iss": "https://token.actions.githubusercontent.com",
  "job_workflow_ref": "aidansteele/aws-federation-github-actions/.github/workflows/test.yml@refs/heads/main",
  "jti": "8ea8373e-0f9d-489d-a480-ac37deexample",
  "nbf": 1631671956,
  "ref": "refs/heads/main",
  "ref_type": "branch",
  "repository": "aidansteele/aws-federation-github-actions",
  "repository_owner": "aidansteele",
```

```
  "run_attempt": "1",
  "run_id": "1235992580",
  "run_number": "5",
  "sha": "bf96275471e83ff04ce5c8eb515c04a75d43f854",
  "sub": "repo:aidansteele/aws-federation-github-actions:ref:refs/heads/main",
  "workflow": "CI"
}
```


I don't think these tokens can be abused once they have expired, but I decided not to share one of mine with you verbatim.

I found these fields interesting:

- actor - who triggered this action? Do we want them to access OpenFaaS?
- iss - who issued this token? We can use this URL to get their public key and then verify the JWT
- repository_owner - who owned the repo? Is it part of our company organisation?

Token exchange

Now I started to look into how Google did its token exchange and came across a lesser-known OAuth2 grant_type called *token exchange*.

It turns out that various IdPs like Okta, Keycloak and Auth0 consider it experimental, and it may need additional configuration to enable it.

You can read about the OAuth 2.0 Token Exchange in its draft Internet Engineering Task Force (IETF) paper: rfc8693

The `grant_type` field should be populated as `urn:ietf:params:oauth:grant-type:token-exchange` and you should be aware of the many other fields prefixed with `urn:ietf:params:oauth`. They are not all required in the spec, but in some of the IdPs I looked into, many of the optional fields were listed as required.

Don't you love standardisation?

What I did instead

Any IdP can be used with OpenFaaS, even those which do not support Token Exchange, so I decided to try validating the OIDC token from GitHub directly.

Here's the way it works:

- The `iss` field maps to `https://token.actions.githubusercontent.com`
- By adding the OIDC configuration endpoint to the path `/.well-known/openid-configuration`, we get a JSON bundle back with various URLs that we can use to then download the public key of the server
- The public key can be used to verify the JWT token
- If the token is good, and hasn't expired, then we know it came from GitHub Actions

Here's the result from the OIDC Discovery URL URL as specified in the OIDC Spec

```
curl -s https://token.actions.githubusercontent.com/.well-known/openid-configuration
```


```
{
  "issuer": "https://token.actions.githubusercontent.com",
  "jwks_uri": "https://token.actions.githubusercontent.com/.well-known/jwks",
  "subject_types_supported": [
    "public",
    "pairwise"
  ],
  "response_types_supported": [
    "id_token"
  ],
  "claims_supported": [
    "sub",
    "aud",
    "exp",
    "iat",
    "iss",
    "jti",
    "nbf",
    "ref",
    "repository",
    "repository_owner",
    "run_id",
    "run_number",
    "run_attempt",
    "actor",
    "workflow",
    "head_ref",
    "base_ref",
    "event_name",
    "ref_type",
    "environment",
    "job_workflow_ref"
  ],
  "id_token_signing_alg_values_supported": [
    "RS256"
  ],
  "scopes_supported": [
    "openid"
  ]
}
```


The field we care about is `jwks` which we can follow to download the public key used to sign JWT tokens.


```
{
  "keys": [
    {
      "n": "zW2j18tSka65aoPgmyk7aUkYE7MmO8z9tM_HoKVJ-w_alYIknkf7pgBeWWfqRgkRfmDuJa8hATL20-bD9cQZ8uVAG1reQfIMxqxwt3DA6q37Co41NdgZ0MUTTQpfC0JyDbDwM_ZIzis1
      "kty": "RSA",
      "kid": "DA6DD449E0E809599CECDFB3BDB6A2D7D0C2503A",
      "alg": "RS256",
      "e": "AQAB",
      "use": "sig",
      "x5c": [ "MIIDrDCCApSgAwIBAgIQBJyUm+htTmG6lfzlIyswTjANBgkqhkiG9w0BAQsFADA2MTQwMgYDVQQDEyt2c3RzLXZzdHNnaHJ0LWdoLXZzby1vYXV0aC52aXN1YWxzdHVkaW8uY29t
      ],
      "x5t": "2m3USeDoCVmc7N-zvbai19DCUDo"
    },
  ]
}
```

Now the `kid` present on this URL is also sent over in the OIDC token from GitHub Actions, but in the header rather than the body that Aidan shared on his blog.

After matching the `kid` from the token to the JWKS payload, you can find the correct public key and verify the OIDC token sent to you. That's what the OpenFaaS plugin does.

But at this point all we've done is allow anyone using a GitHub Action to deploy to our cluster. So we are not quite done yet, because I want to restrict that to just my friends and colleagues.

I updated the inlets tunnel so it pointed at my OpenFaaS instance running on KinD:

```
kubectl port-forward -n openfaas deploy/gateway 8080:8080

inlets client .. \
  --upstream http://127.0.0.1:8080
```

I could have also deployed the [tunnel into KinD using the helm chart](), but didn't need this to be permanent.

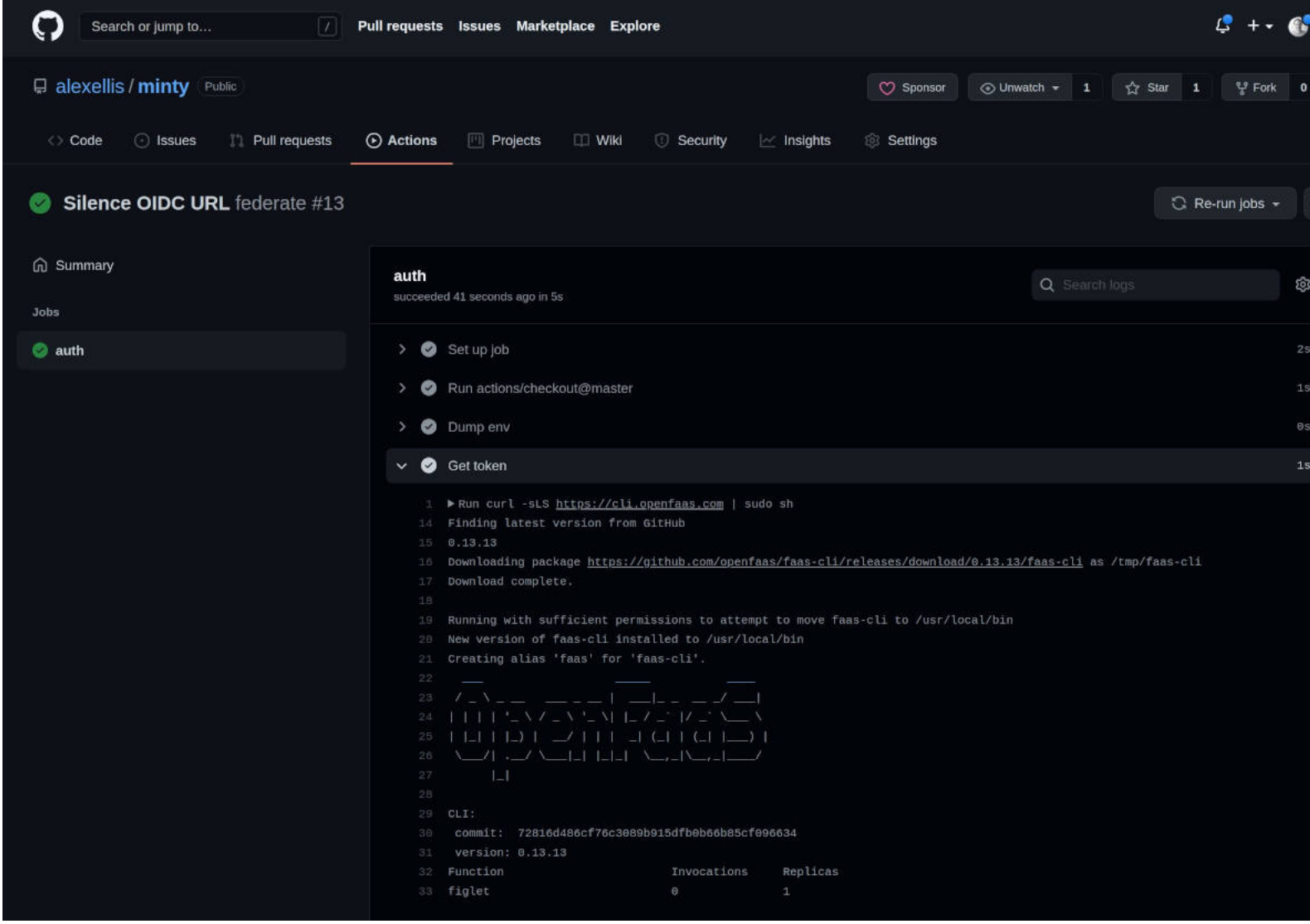
Next up, I changed GitHub Action to install the OpenFaaS CLI and to run `faas-cli list` using the token which I'd extacted into an environment variable.

```
name: federate

on:
  workflow_dispatch:
  push:
    branches:
    - '*'

jobs:
  auth:

    # Add "id-token" with the intended permissions.
    permissions:
      contents: 'read'
      id-token: 'write'

    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
        with:
          fetch-depth: 1
      - name: Install faas-cli
        run: curl -sLS https://cli.openfaas.com | sudo sh

      - name: Get token and use the CLI
        run: |
          OIDC_TOKEN=$(curl -sLS "${ACTIONS_ID_TOKEN_REQUEST_URL}&audience=minty.exit.o6s.io" -H "User-Agent: actions/oidc-client" -H "Authorization: Be
          JWT=$(echo $OIDC_TOKEN | jq -j '.value')
          export OPENFAAS_URL=https://minty.exit.o6s.io/

          faas-cli list --token "$JWT"
```

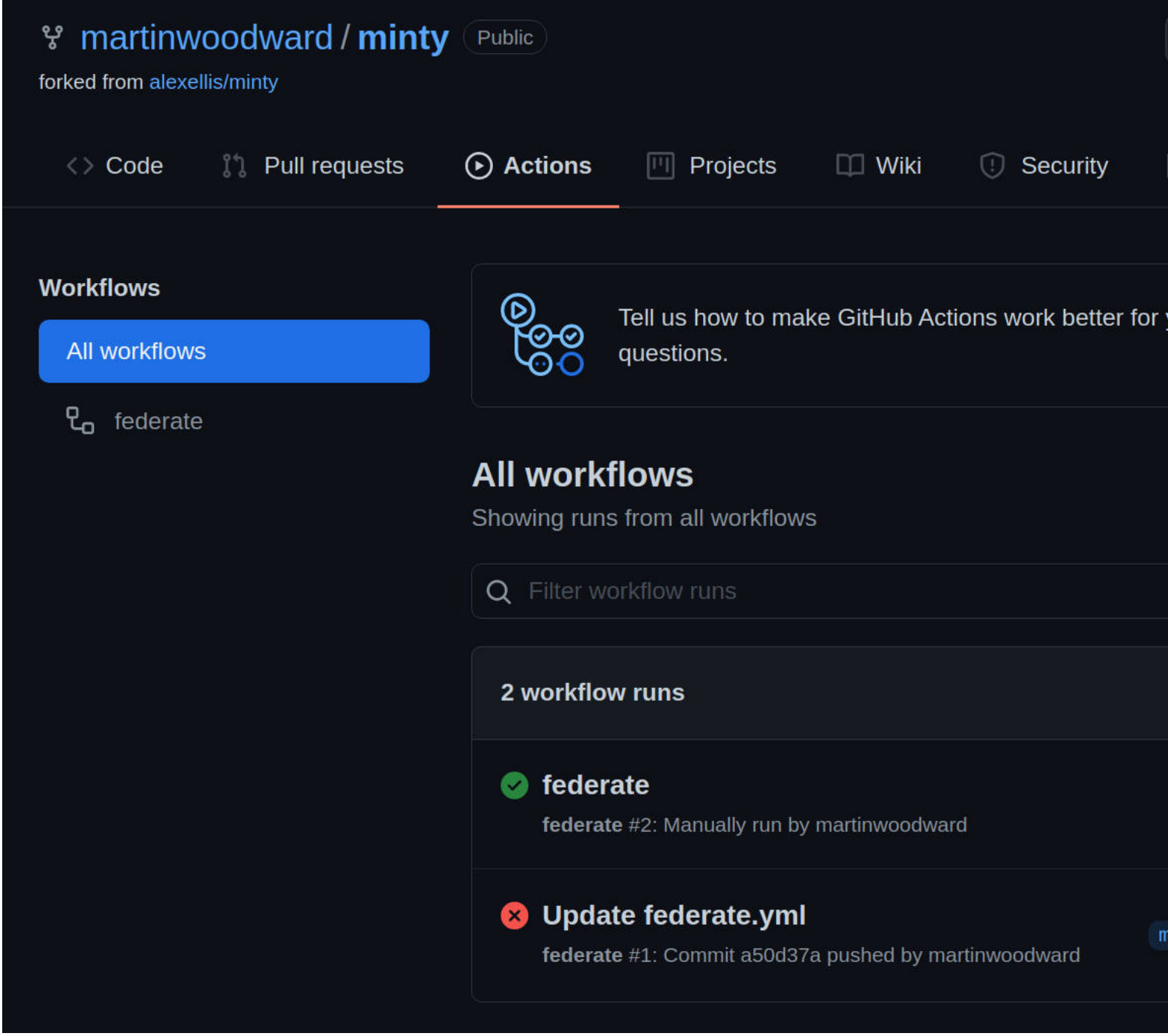
Normally we'd run `faas-cli login` followed by `list`, `deploy` and so forth, but here we're using `--token`. OpenFaaS Pro supports various OAuth2 flows, you can read more here: [OpenFaaS Single Sign-On]()

To my surprise, it worked!

There was just a little bit more work to do, I needed to write an Access Control List (ACL) and use one of GitHub Actions' fields to authorize just my friends. I used the `actors` field and enabled anyone in that field to be an admin for the OpenFaaS REST API.



Then I needed some help from a tester. [Martin Woodward, Director of DevRel at GitHub]() was only to happy to help.



He first got "unauthorized" then I added him to the ACL and it worked.

Later on, Lucas Roesler a core contributor to OpenFaaS deployed a function of his own and it worked as expected. Lucas is very knowledgeable about OAuth and OIDC, both of which have a lot of moving parts, so he is a good person to run things by when I'm doing something new.

The OIDC Specification is also very readable and many client libraries exist for various languages. You should probably use these whenever you can instead of writing your own.

> https://t.co/6csQKTbtd1
>
> No copying secrets or anything!!! https://t.co/3Ljji9QZ4t pic.twitter.com/RRUBaljKSM
>
> — Lucas Roesler (@TheAxeR) October 6, 2021

Why do I like this so much?

In the past I built a complex PaaS called OpenFaaS Cloud which integrated tightly with GitHub and GitLab, but those were much earlier times and predated GitHub Actions. OpenFaaS Cloud took you from commiting code into a linked repository, to having a live endpoint within a few seconds.

However, it was quite a lot of code to maintain and required many additional components. After all, they do say that you can have a secure system or a simple one. There are some parts I think we did well like the multi-user UI, secret support and user experience. But users wanted to customise everything, or build with their existing CI system.

If you'd like to know more about what we built back then, the culimation is probably this conference talk from 2019: KubeCon: OpenFaaS Cloud + Linkerd: A Secure, Multi-Tenant Serverless Platform - Charles Pretzer & Alex Ellis

With the new OIDC configuration, GitHub Actions and multiple namespace support in OpenFaaS, you can get very close to a multi-tenant, highly-integrated and portable serverless platform.

But that's not all.

Taking it further

The authorization rules could also be enhanced further so that Lucas could only deploy to a set of mapped namespaces and we wouldn't bump into each other that way. Perhaps we'd set up a central shared OpenFaaS server and we could just deploy whatever we needed there.

This has all been done on Kubernetes so far, but faasd has become a promising alternative way to run OpenFaaS. It uses containerd, but has no clustering support meaning it can run very well on a 5 USD VM, a work hypervisor or even on an edge compute device like a Raspberry Pi. All the OpenFaaS Pro components will work on faasd since they are not specific to Kubernetes, but I've not had a customer ask for that *yet*.

OpenFaaS functions are container images, and GitHub Actions has really impressive support for building them with Docker including caching, multi-arch support with buildx. It can be tricky to put all this together for the first time, so in my eBook Serverless For Everyone Else, I give a reference example.

How I would expect this plug-in to be used would be to: to get a commit event, build a set of multi-archtecture images, push them to GitHub Container Registry, and then trigger a deployment using the OIDC federation. If the cluster was public, just the OpenFaaS gateway URL would need to be shared, which is non-confidential data. If the function is going to an edge device or a private cluster, then an inlets tunnel would be needed to access the gateway.

GitLab also offers OIDC tokens during CI jobs when used in combination with Hashicorp Vault. As long as the proper issuer is configured in the new openfaas plugin, then any valid issuer would work in the same way as GitHub Actions. One thing I really enjoy is writing code once and then reusing it again.

I'd like to offer the OpenFaaS GitHub Actions federation to the community to try out and give feedback. Ping me on OpenFaaS Slack to try it out.

> If you can't wait for that, OpenFaaS Pro already has SSO support with any OIDC-compatible IdP like Azure LDAP, Auth0, Okta and Keycloak. Customers also get access to scale to zero for better efficiency, Kafka event integration, retries with exponential backoff and a chance to influence the future roadmap.

My Go application and the GitHub Actions workflow I shared are both available on my GitHub account.