
Article

[Mikhail Khomenko](#) · Feb 11, 2020 17m read

[Open Exchange](#)

Deploying InterSystems IRIS solution on GKE Using GitHub Actions

In an [earlier](#) article (hope, you ' ve read it), we took a look at the CircleCI deployment system, which integrates perfectly with GitHub. Why then would we want to look any further? Well, GitHub has its own CI/CD platform called GitHub Actions, which is worth exploring. With GitHub Actions, you don ' t need to rely on some external, albeit cool, service.

In this article we ' re going to try using GitHub Actions to deploy the server part of InterSystems Package Manager, [ZPM-registry](#), on Google Kubernetes Engine (GKE).

As with all systems, the build/deploy process essentially comes down to " do this, go there, do that, " and so on. With GitHub Actions, each such action is a job that consists of one or more steps, together known as a [workflow](#). GitHub will search for a description of the workflow in the YAML file (any filename ending in .yaml or .yml) in your .github/workflows directory. See [Core concepts for GitHub Actions](#) for more details.

All further actions will be performed in the fork of the [ZPM-registry repository](#). We ' ll call this fork "zpm-registry" and refer to its root directory as "<rootrepor> " throughout this article. To learn more about the ZPM application itself see [Introducing InterSystems ObjectScript Package Manager](#) and [The Anatomy of ZPM Module: Packaging Your InterSystems Solution](#).

All code samples are stored [in this repository](#) to simplify copying and pasting. The prerequisites are the same as in the article [Automating GKE creation on CircleCI builds](#).

We ' ll assume you ' ve read the earlier article and already have [Google account](#), and that you ' ve created a project named "Development," as in the previous article. In this article, its ID is shown as <PROJECTID>. In the examples below, change it to [the ID of your own project](#).

Keep in mind that Google isn ' t free, although it has [free tier](#). Be sure to [control your expenses](#).

Workflow Basics

Let ' s get started.

A simple and useless workflow file might look like this:

```
$ cd <rootrepor>
$ mkdir -p .github/workflows
$ cat <rootrepor>/.github/workflows/workflow.yaml
name: Traditional Hello World
on: [push]
jobs:
  courtesy:
    name: Greeting
    runs-on: ubuntu-latest
```

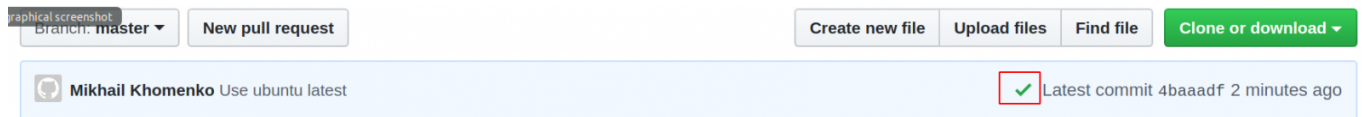
steps:

- name: Hello world

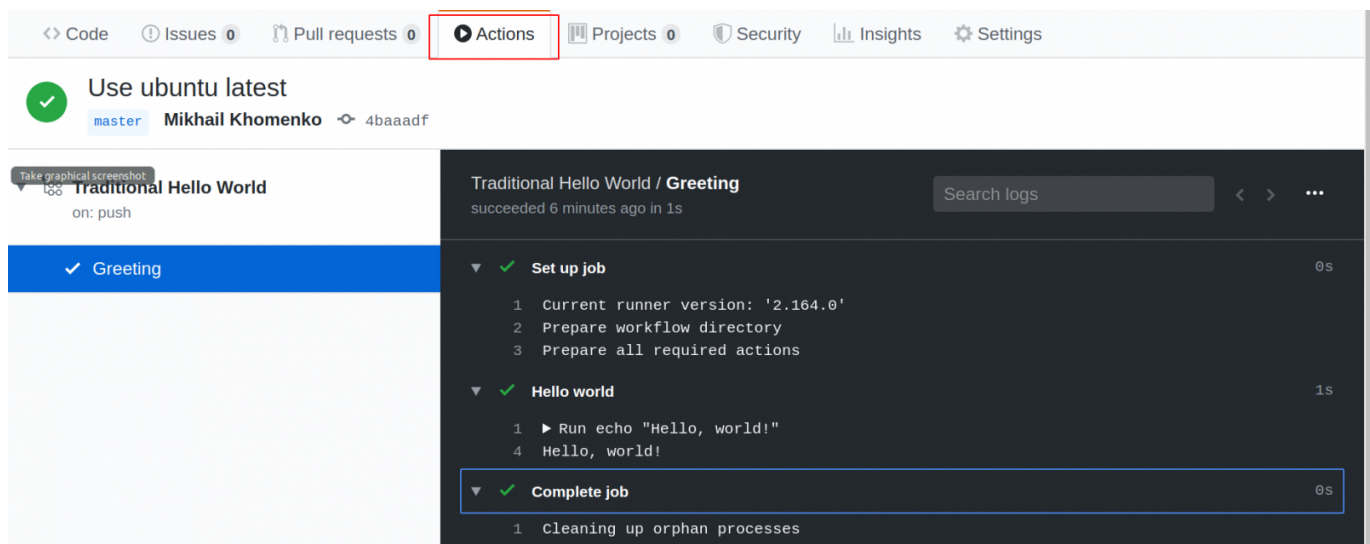
run: echo "Hello, world!"

When pushing to the repository, you need to execute a job named "Greeting," which consists of a single step: printing a welcome phrase. The job should run on a GitHub-hosted virtual machine called the Runner, with the latest version of Ubuntu installed.

After pushing this file to the repository, you should see on the Code GitHub tab that everything went well:



If the job had failed, you 'd see a red X instead of a green checkmark. To see more, click on the green checkmark and then on Details. Or you can immediately go to the Actions tab:



You can learn all about the workflow syntax in the help document [Workflow syntax for GitHub Actions](#).

If your repository contains a Dockerfile for the image build, you could replace the "Hello world" step with something more useful like this example from [starter-workflows](#):

steps:

- uses: actions/checkout@v2

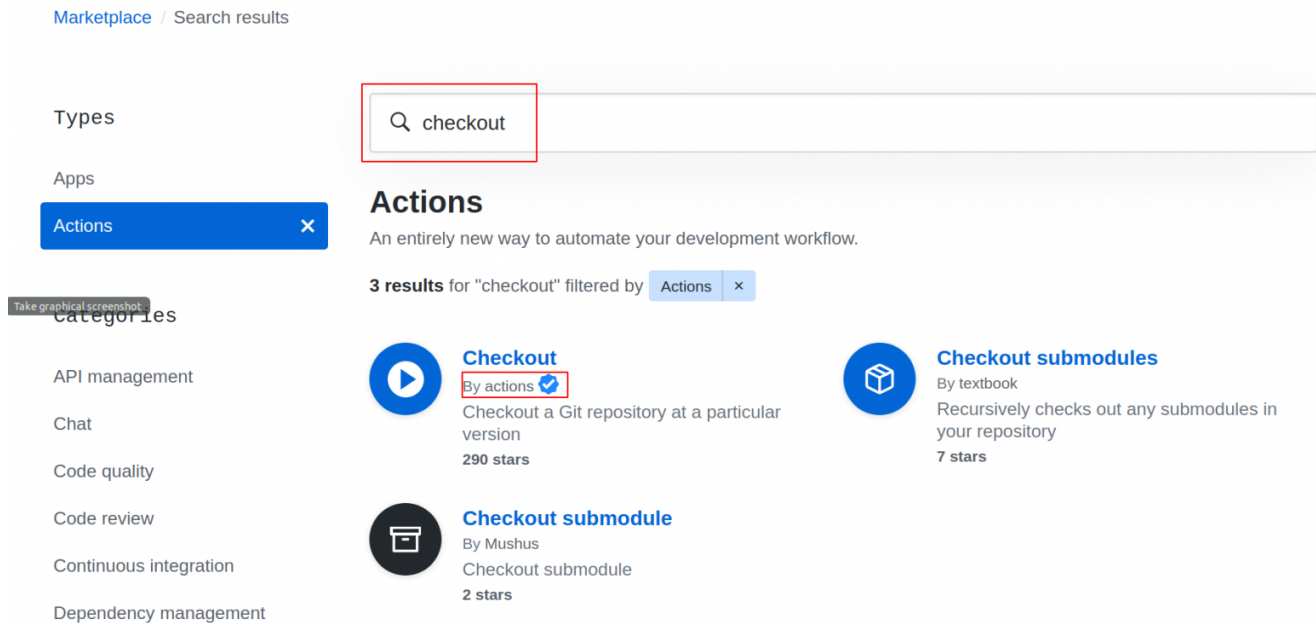
- name: Build the Docker image

run: docker build . --file Dockerfile --tag my-image:\$(date +%s)

Notice that a new step, "uses: action/checkout@v2", was added here. Judging by the name "checkout", it clones the repository, but where to find out more?

As in the case of CircleCI, many useful steps don ' t need to be rewritten. Instead, you can take them from the

shared resource called [Marketplace](#). Look there for the desired action, and note that it 's better to take those that are marked as "By actions" (when you hover over - "Creator verified by Github").



The "uses" clause in the workflow reflects our intention to use a ready-made module, rather than writing one ourselves.

The implementations of the actions themselves can be written in almost any language, but JavaScript is preferred. If your action is written in JavaScript (or TypeScript), it will be executed directly on the Runner machine. For other implementations, the Docker container you specify will run with the desired environment inside, which is obviously somewhat slower. You can read more about actions in the aptly titled article, [About actions](#).

The [checkout action](#) is written in TypeScript. And in our example, [Terraform action](#) is a regular bash script launched in Docker Alpine.

There 's a Dockerfile in our cloned repository, so let's try to apply our new knowledge. We ' ll build the image of the ZPM registry and push it into the Google Container Registry. In parallel, we ' ll create the Kubernetes cluster in which this image will run, and we ' ll use Kubernetes manifests to do this.

Here 's what our plan, in a language that GitHub understands, will look like (but keep in mind that this is a bird's eye view with many lines omitted for simplification, so don 't actually use this config):

name: Workflow description

Trigger condition. In this case, only on push to ' master ' branch

on:

push:

branches:

- master

Here we describe environment variables available

for all further jobs and their steps

These variables can be initialized on GitHub Secrets page

We add " \${ secrets } " to refer them

env:

PROJECTID: \${{ secrets.PROJECTID }}

Define a jobs list. Jobs/steps names could be random but
it 's better to have they meaningful

jobs:

gcloud-setup-and-build-and-publish-to-GCR:

name: Setup gcloud utility, Build ZPM image and Publish it to Container Registry

runs-on: ubuntu-18.04

steps:

- name: Checkout
- name: Setup gcloud cli
- name: Configure docker to use the gcloud as a credential helper
- name: Build ZPM image
- name: Publish ZPM image to Google Container Registry

gke-provisioner:

name: Provision GKE cluster

runs-on: ubuntu-18.04

steps:

- name: Checkout
- name: Terraform init
- name: Terraform validate
- name: Terraform plan
- name: Terraform apply

kubernetes-deploy:

name: Deploy Kubernetes manifests to GKE cluster

needs:

- gcloud-setup-and-build-and-publish-to-GCR
- gke-provisioner

runs-on: ubuntu-18.04

steps:

- name: Checkout
- name: Replace placeholders with values in statefulset template
- name: Setup gcloud cli
- name: Apply Kubernetes manifests

This is the skeleton of the working config in which there are no muscles, the real actions for each step. Actions can be accomplished with a simple console command ("run" or "run |" if there are several commands):

- name: Configure docker to use gcloud as a credential helper

run: |

gcloud auth configure-docker

You can also launch actions as a module with "uses":

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

By default, all jobs run in parallel, and the steps in them are done in sequence. But by using "needs", you can specify that one job should wait for the rest to complete:

needs:

```
- gcloud-setup-and-build-and-publish-to-GCR
- gke-provisioner
```

By the way, in the GitHub Web interface, such waiting jobs appear only when the jobs they 're waiting for are executed.

The "gke-provisioner" job mentions Terraform, which we examined in [the previous article](#). The preliminary settings for its operation in the GCP environment are repeated for convenience in a separate [markdown file](#). Here are some additional useful links:

- [Terraform Apply Subcommand documentation](#)
- [Terraform GitHub Actions repository](#)
- [Terraform GitHub Actions documentation](#)

In the "kubernetes-deploy" job, there is a step called "Apply Kubernetes manifests". We 're going to use manifests as mentioned in the article [Deploying InterSystems IRIS Solution into GCP Kubernetes Cluster GKE Using CircleCI](#), but with a slight change.

In the previous articles, [IRIS application](#) has been stateless. That is, when restarting the pod, all data is returned to its default place. This is great, and it 's often necessary, but for ZPM registry you need to somehow save the packages that were loaded into it, regardless of how many times you need to restart. Deployment allows you to do this, of course, but not without limitations.

For stateful applications, it 's better to choose the [StatefulSet](#) resource. Pros and cons can be found in the GKE documentation topic on [Deployments vs. StatefulSets](#) and the blog post [Kubernetes Persistent Volumes with Deployment and StatefulSet](#).

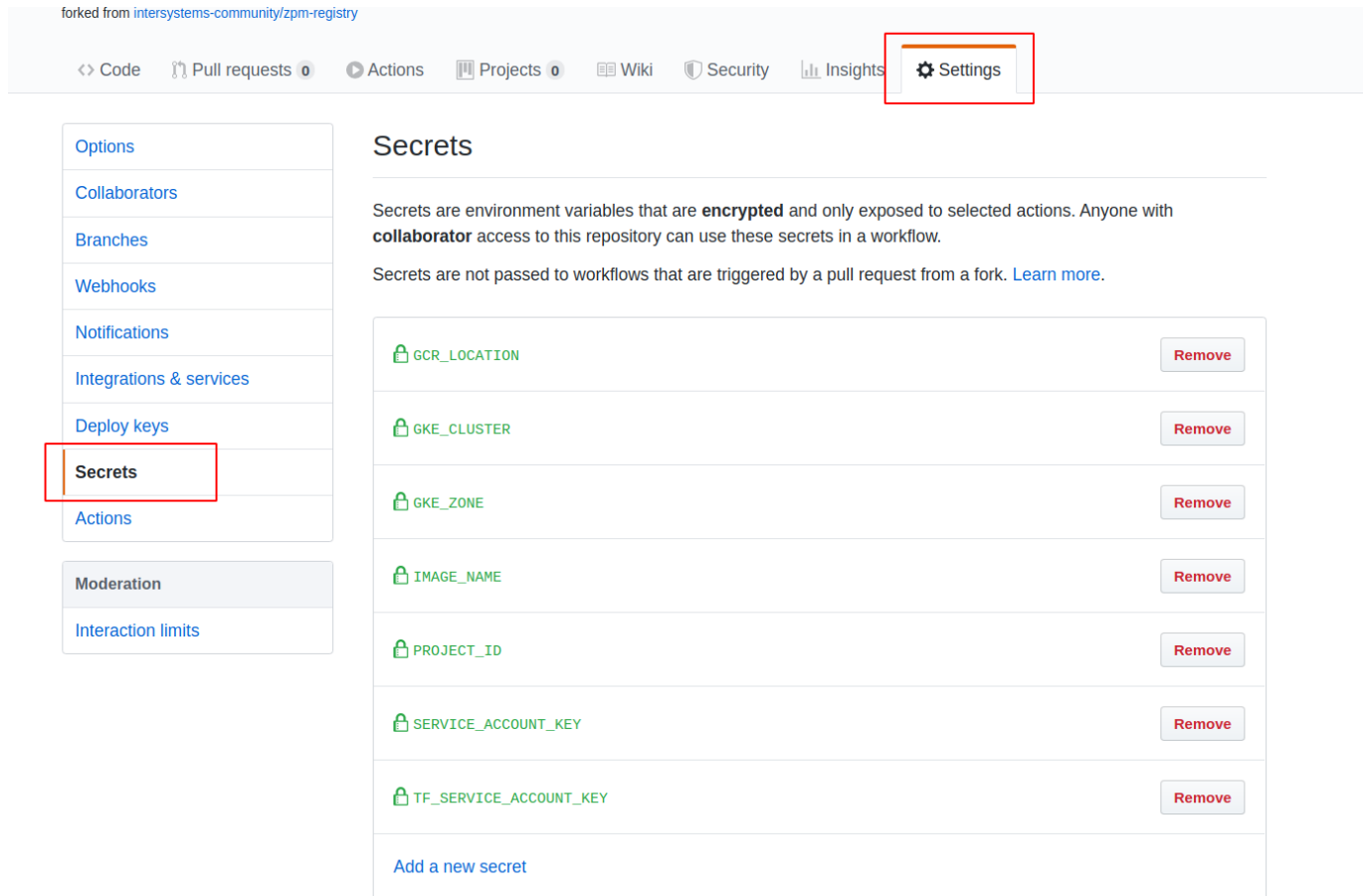
The StatefulSet resource is in [the repository](#). Here 's the part that 's important for us:

volumeClaimTemplates:

```
- metadata:
  name: zpm-registry-volume
  namespace: iris
spec:
  accessModes:
  - ReadWriteOnce
  resources:
  requests:
  storage: 10Gi
```

The code creates a 10GB read/write disk that can be mounted by a single Kubernetes worker node. This disk (and the data on it) will survive the restart of the application. It can also survive the removal of the entire StatefulSet, but for this you need to set the correct [Reclaim Policy](#), which we won't cover here.

Before breathing life into our workflow, let's add a few more variables to [GitHub Secrets](#):



The following table explains the meaning of these settings ([service account keys](#) are also present):

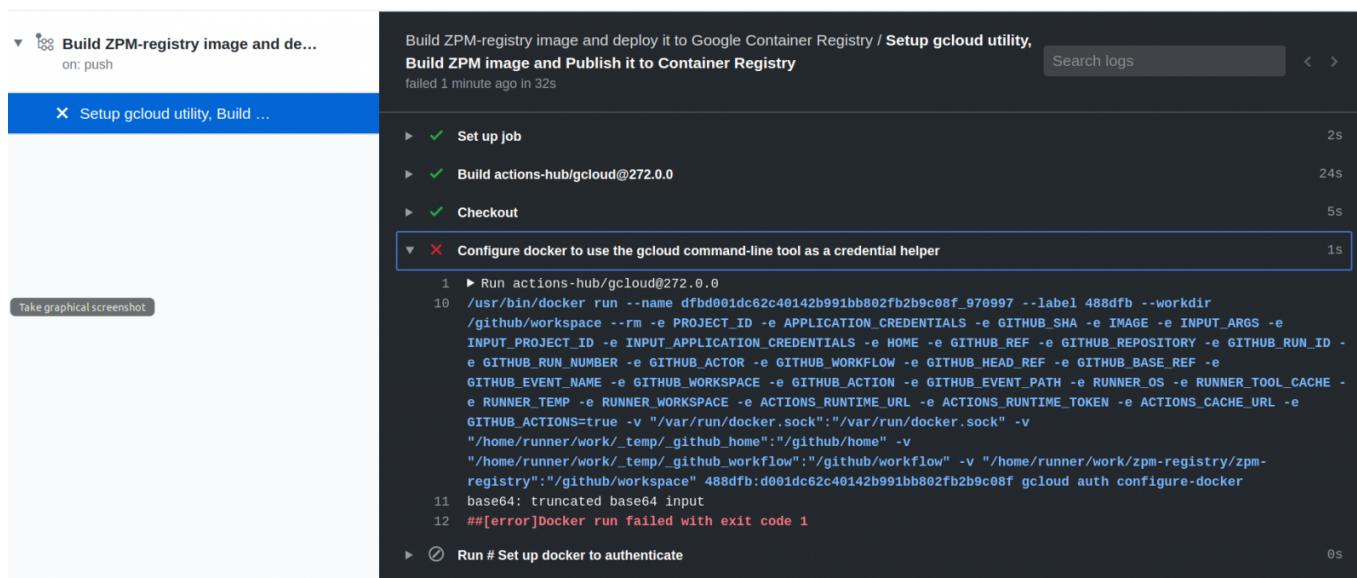
Name	Meaning	
GCR_LOCATION	Global GCR location	eu
GKE_CLUSTER	GKE cluster name	de
GKE_ZONE	Zone to store an image	eu
IMAGE_NAME	Image registry name	zpm
PROJECT_ID	GCP Project ID	po
SERVICE_ACCOUNT_KEY	JSON key GitHub uses to connect to GCP. Important: it has to be base64-encoded (see note below)	ev
TF_SERVICE_ACCOUNT_KEY	JSON key Terraform uses to connect to GCP (see note below)	{ ...

Name	Meaning	
		}

For `SERVICEACCOUNTKEY`, if your JSON-key has a name, for instance, `key.json`, run the following command:
`$ base64 key.json | tr -d '\n'`

For `TFSERVICEACCOUNTKEY`, note that its rights are described in [Automating GKE creation on CircleCI builds](#).

One small note about `SERVICEACCOUNTKEY`: if you, like me, initially forgot to convert it to base64 format, you 'll see a screen like this:



Now that we 've looked at the workflow backbone and added the necessary variables, we 're ready to examine the full version of the workflow ([<rootrepodir>/.github/workflow/workflow.yaml](#)):

name: Build ZPM-registry image, deploy it to GCR. Run GKE. Run ZPM-registry in GKE

on:

push:

branches:

- master

Environment variables.

`{{ secrets }}` are taken from GitHub -> Settings -> Secrets

`{{ github.sha }}` is the commit hash

env:

`PROJECTID: {{ secrets.PROJECTID }}`

`SERVICEACCOUNTKEY: {{ secrets.SERVICEACCOUNTKEY }}`

`GOOGLECREDENTIALS: {{ secrets.TFSERVICEACCOUNTKEY }}`

GITHUB_SHA: \${github.sha}
GCR_LOCATION: \${secrets.GCR_LOCATION}
IMAGE_NAME: \${secrets.IMAGE_NAME}
GKE_CLUSTER: \${secrets.GKE_CLUSTER}
GKE_ZONE: \${secrets.GKE_ZONE}
K8S_NAMESPACE: iris
STATEFULSET_NAME: zpm-registry

jobs:

gcloud-setup-and-build-and-publish-to-GCR:

name: Setup gcloud utility, Build ZPM image and Publish it to Container Registry

runs-on: ubuntu-18.04

steps:

- name: Checkout

uses: actions/checkout@v2

- name: Setup gcloud cli

uses: GoogleCloudPlatform/github-actions/[setup-gcloud@master](#)

with:

version: '275.0.0'

serviceaccountkey: \${secrets.SERVICEACCOUNTKEY}

- name: Configure docker to use the gcloud as a credential helper

run: |

gcloud auth configure-docker

- name: Build ZPM image

run: |

docker build -t

\${GCR_LOCATION}/\${PROJECT_ID}/\${IMAGE_NAME}:\${GITHUB_SHA} .

- name: Publish ZPM image to Google Container Registry

run: |

docker push

\${GCR_LOCATION}/\${PROJECT_ID}/\${IMAGE_NAME}:\${GITHUB_SHA}

gke-provisioner:

Inspired by:

<https://www.terraform.io/docs/github-actions/getting-started.html>

<https://github.com/hashicorp/terraform-github-actions>

name: Provision GKE cluster

runs-on: ubuntu-18.04

steps:

- name: Checkout

uses: actions/checkout@v2

- name: Terraform init

uses: hashicorp/[terraform-github-actions@master](#)

with:

tfactionsversion: 0.12.17

tfactionssubcommand: 'init'

tfactionsworkingdir: 'terraform'

- name: Terraform validate

uses: hashicorp/[terraform-github-actions@master](#)

with:

tfactionsversion: 0.12.17

tfactionssubcommand: 'validate'

tfactionsworkingdir: 'terraform'

- name: Terraform plan

uses: hashicorp/[terraform-github-actions@master](#)

with:

tfactionsversion: 0.12.17

tfactionssubcommand: 'plan'

tfactionsworkingdir: 'terraform'

- name: Terraform apply

uses: hashicorp/[terraform-github-actions@master](#)

with:

tfactionsversion: 0.12.17

tfactionssubcommand: 'apply'

tfactionsworkingdir: 'terraform'

kubernetes-deploy:

name: Deploy Kubernetes manifests to GKE cluster

needs:

- gcloud-setup-and-build-and-publish-to-GCR

- gke-provisioner

runs-on: ubuntu-18.04

steps:

- name: Checkout

uses: actions/checkout@v2

- name: Replace placeholders with values in statefulset template

working-directory: ./k8s/

run: |

cat statefulset.tpl | /

```
sed "s|DOCKERREPONAME|${GCR_LOCATION}/${PROJECTID}/${IMAGE_
NAME}|" | /
sed "s|DOCKERIMAGE_TAG|${GITHUB_SHA}|" > statefulset.yaml
cat statefulset.yaml
```

```
- name: Setup gcloud cli
uses: GoogleCloudPlatform/github-actions/setup-gcloud@master
with:
version: '275.0.0'
serviceaccountkey: ${ secrets.SERVICEACCOUNTKEY }

- name: Apply Kubernetes manifests
working-directory: ./k8s/
run: |
gcloud container clusters get-credentials ${GKE_CLUSTER} --zone
${GKE_ZONE} --project ${PROJECTID}
kubectl apply -f namespace.yaml
kubectl apply -f service.yaml
kubectl apply -f statefulset.yaml
kubectl -n ${K8S_NAMESPACE} rollout status
statefulset/${STATEFULSETNAME}
```

Before you push to a repository, you should take the terraform-code from the [Terraform directory of github-gke-zpm-registry repository](#), replace placeholders as noted in main.tf comment, and put it inside the terraform/ directory. Remember that Terraform uses a remote bucket that should be initially created as noted in [Automating GKE creation on CircleCI builds article](#).

Also, Kubernetes-code should be taken from the [K8S directory of github-gke-zpm-registry repository](#) and put inside the k8s/ directory. These code sources were omitted in this article to save space.

Then you can trigger a deploy:

```
$ cd <rootrepor> /
$ git add .github/workflow/workflow.yaml k8s/ terraform/
$ git commit -m " Add GitHub Actions deploy "
$ git push
```

After pushing the changes to our forked ZPM repository, we can take a look at the implementation of the steps we described:

Deploying InterSystems IRIS solution on GKE Using GitHub Actions

Published on InterSystems Developer Community (<https://community.intersystems.com>)

The screenshot shows the GitHub Actions interface for a workflow named "Build ZPM-registry image, deploy it to GCR. Run GKE. Run ZPM-registry in GKE". The workflow is triggered by a push event on the master branch. The workflow status is "All stages uncommitted". The workflow file is located at .github/workflows/build-zpm-registry.yml. The workflow is triggered by myardyas on the master branch, 15 seconds ago, with a duration of 14s.

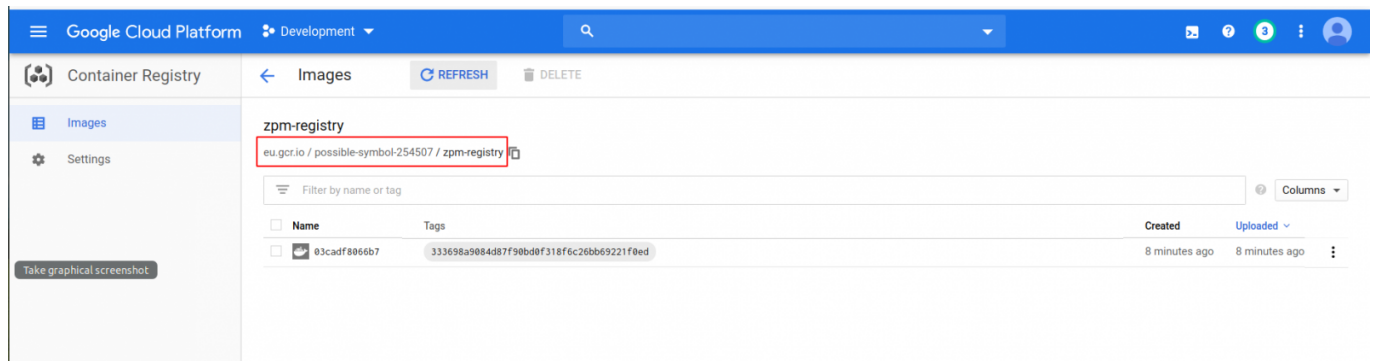
There are only two jobs so far. The third, "kubernetes-deploy", will appear after the completion of those on which it depends.

Note that building and publishing Docker images requires some time:

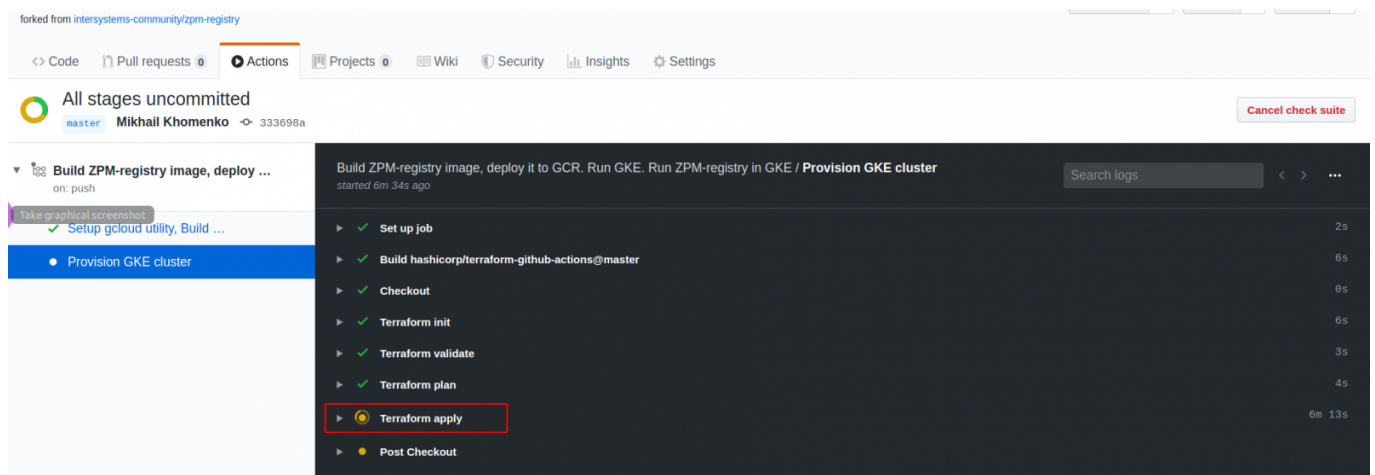
The screenshot shows the details of a workflow run for the workflow "Build ZPM-registry image, deploy it to GCR. Run GKE. Run ZPM-registry in GKE". The run is triggered by a push event on the master branch. The run status is "succeeded 3 minutes ago in 1m 43s". The run is triggered by Mikhail Khomenko on the master branch, 333698a. The run is triggered by myardyas on the master branch, 15 seconds ago, with a duration of 14s.

Job	Status	Duration
Set up job	✓	5s
Checkout	✓	6s
Setup gcloud cli	✓	8s
Configure docker to use the gcloud as a credential helper	✓	6s
Build ZPM image	✓	1m 5s
Publish ZPM image to Google Container Registry	✓	13s
Post Checkout	✓	8s
Complete job	✓	8s

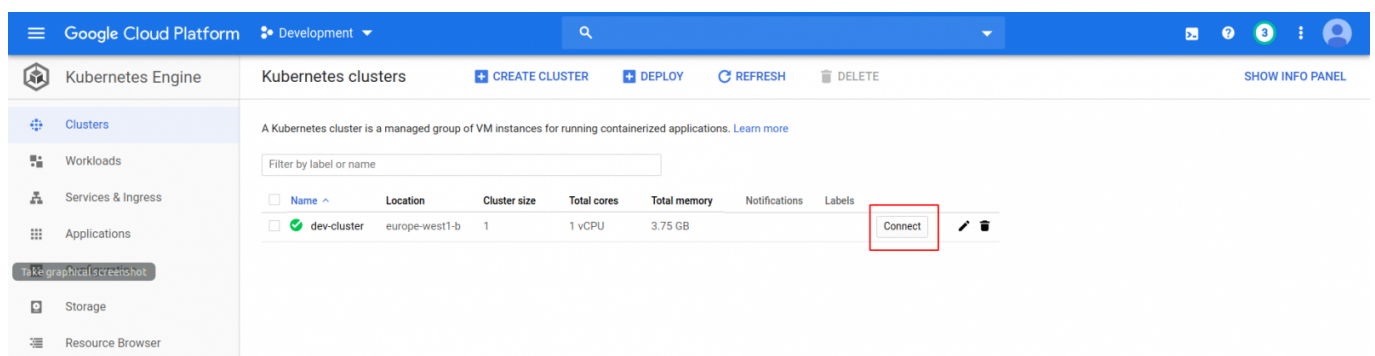
And you can check the result in the [GCR console](#):



The "Provision GKE cluster" job takes longer the first time as it creates the GKE cluster. You 'll see a waiting screen for a few minutes:



But, finally, it finishes and you can be happy:



The Kubernetes resources are also happy:

```
$ gcloud container clusters get-credentials <CLUSTERNAME> --zone  
<GKEZONE> --project <PROJECTID>  
$ kubectl get nodes
```

```
NAME STATUS ROLES AGE VERSION
gke-dev-cluster-dev-cluster-node-pool-98cef283-dfq2 Ready <none> 8m51s
v1.13.11-gke.23
```

```
$ kubectl -n iris get po
NAME READY STATUS RESTARTS AGE
zpm-registry-0 1/1 Running 0 8m25s
```

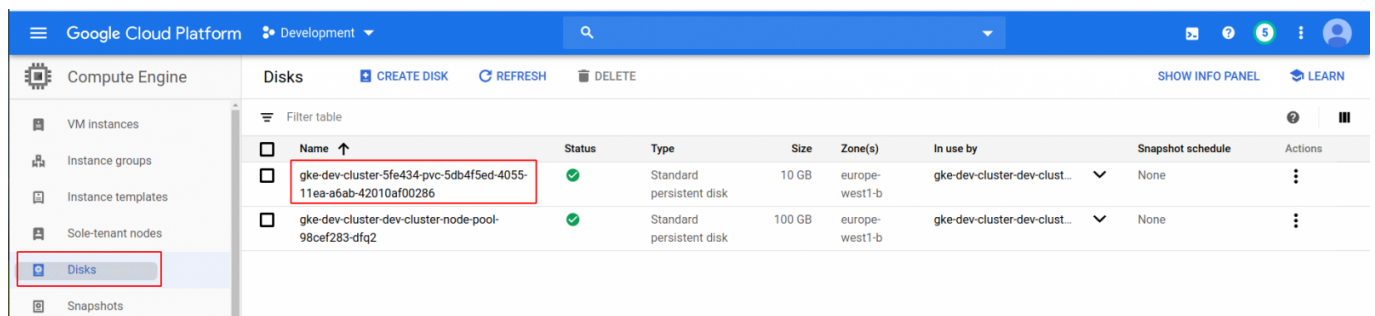
It's a good idea to wait for Running status, then check other things:

```
$ kubectl -n iris get sts
NAME READY AGE
zpm-registry 1/1 8m25s
```

```
$ kubectl -n iris get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
zpm-registry LoadBalancer 10.23.248.234 104.199.6.32 52773:32725/TCP
8m29s
```

Even the disks are happy:

```
$ kubectl get pv -oyaml | grep pdName
pdName: gke-dev-cluster-5fe434-pvc-5db4f5ed-4055-11ea-a6ab-42010af00286
```



Name	Status	Type	Size	Zone(s)	In use by	Snapshot schedule	Actions
gke-dev-cluster-5fe434-pvc-5db4f5ed-4055-11ea-a6ab-42010af00286	OK	Standard persistent disk	10 GB	europe-west1-b	gke-dev-cluster-dev-clust...	None	⋮
gke-dev-cluster-dev-cluster-node-pool-98cef283-dfq2	OK	Standard persistent disk	100 GB	europe-west1-b	gke-dev-cluster-dev-clust...	None	⋮

And happiest of all is the ZPM registry (we took the External-IP output of "kubectl -n iris get svc"):

```
$ curl -u system:SYS 104.199.6.32:52773/registry/ping
{"message":"ping"}
```

Handling the login/password over HTTP is a shame, but I hope to do something about this in future articles.

By the way, you can find more information about endpoints in [the source code](#): see the XData UriMap section.

We can test this repo by pushing a package to it. There 's a cool ability to push just a direct GitHub link. Let 's try with the [math library for InterSystems ObjectScript](#). Run this from your local machine:

```
$ curl -XGET -u system:SYS 104.199.6.32:52773/registry/packages/-/all  
[]  
$ curl -i -XPOST -u system:SYS -H "Content-Type: application/json" -d  
'{"repository": "https://github.com/psteiwer/ObjectScript-Math"}'  
'http://104.199.6.32:52773/registry/package'  
HTTP/1.1 200 OK  
$ curl -XGET -u system:SYS 104.199.6.32:52773/registry/packages/-/all  
[{"name": "objectscript-math", "versions": ["0.0.4"]}]
```

Restart a pod to be sure that the data is in place:

```
$ kubectl -n iris scale --replicas=0 sts zpm-registry  
$ kubectl -n iris scale --replicas=1 sts zpm-registry  
$ kubectl -n iris get po -w
```

Wait for a running pod. Then what I hope you 'll see:

```
$ curl -XGET -u system:SYS 104.199.6.32:52773/registry/packages/-/all  
[{"name": "objectscript-math", "versions": ["0.0.4"]}]
```

Let 's install this math package from your repository on your local IRIS instance. Choose the one where the ZPM client is already installed:

```
$ docker exec -it $(docker run -d intersystemsdc/iris-  
community:2019.4.0.383.0-zpm) bash
```

```
$ iris session iris  
USER>write ##class(Math.Math).Factorial(5)  
<CLASS DOES NOT EXIST> *Math.Math
```

```
USER>zpm  
zpm: USER>list
```

```
zpm: USER>repo -list  
registry  
Source: https://pm.community.intersystems.com  
Enabled? Yes  
Available? Yes  
Use for Snapshots? Yes  
Use for Prereleases? Yes
```

```
zpm: USER>repo -n registry -r -url http://104.199.6.32:52773/registry/ -user  
system -pass SYS
```



```
zpm: USER>repo -list
registry
Source: http://104.199.6.32:52773/registry/
Enabled? Yes
Available? Yes
Use for Snapshots? Yes
Use for Prereleases? Yes
Username: system
Password: <set>
```

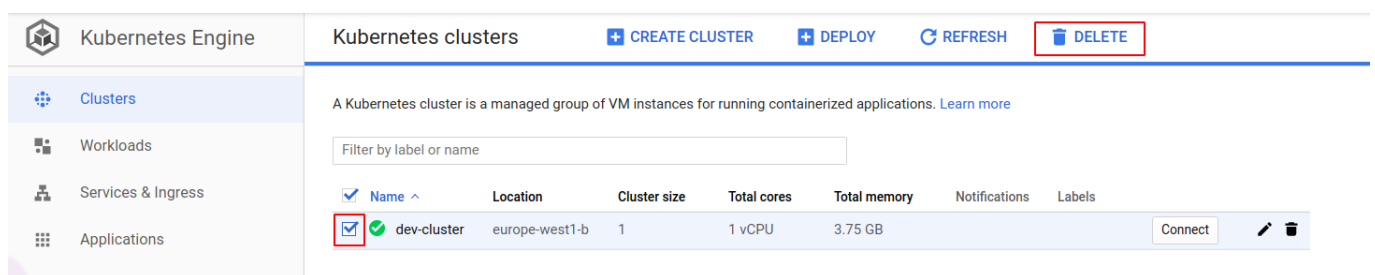
```
zpm: USER>repo -list-modules -n registry
objectscript-math 0.0.4
```

```
zpm: USER>install objectscript-math
[objectscript-math] Reload START
...
[objectscript-math] Activate SUCCESS
```

```
zpm: USER>quit
```

```
USER>write ##class(Math.Math).Factorial(5)
120
```

Congratulations!
Don ' t forget to remove the GKE cluster when you don ' t need it anymore:



Kubernetes Engine							
Kubernetes clusters							
+ CREATE CLUSTER + DEPLOY REFRESH DELETE							
A Kubernetes cluster is a managed group of VM instances for running containerized applications. Learn more							
Filter by label or name							
<input checked="" type="checkbox"/>	Name ^	Location	Cluster size	Total cores	Total memory	Notifications	Labels
<input checked="" type="checkbox"/>	dev-cluster	europe-west1-b	1	1 vCPU	3.75 GB		Connect

Conclusion

There are not many references to GitHub Actions within the InterSystems community. I found only [one mention](#) from guru [@mdaimor](#). But GitHub Actions can be quite useful for developers storing code on GitHub. Native actions supported only in JavaScript, but this could be dictated by a desire to describe steps in code, which most developers are familiar with. In any case, you can use Docker actions if you don ' t know JavaScript.

Regarding the GitHub Actions UI, along the way I discovered a couple of inconveniences that you should be aware of:

- You cannot check what is going on until a job step is finished. It ' s not clickable, like in the step "Terraform apply".
- While you can rerun a failed workflow, I didn ' t find a way to rerun a successful workflow.

A workaround for the second point is to use the command:

```
$ git commit --allow-empty -m "trigger GitHub actions"
```

You can learn more about this in the StackOverflow question [How do I re-run Github Actions?](#)

[#Best Practices](#) [#Cloud](#) [#Containerization](#) [#DevOps](#) [#Docker](#) [#GitHub](#) [#Kubernetes](#) [#InterSystems IRIS](#) [#Open Exchange](#)

[Check the related application on InterSystems Open Exchange](#)

Source

URL:<https://community.intersystems.com/post/deploying-intersystems-iris-solution-gke-using-github-actions>