記事
Shintaro Kaminaka · 2020年12月10日 20m read

Open Exchange

GitHub Actions を使って GKE に InterSystems IRIS Solution をデプロイする

以前紹介した記事 (お読みいただいたでしょうか) では、GitHub とパーフェクトに統合する CircleCI のデプロイシステムについてカバーしました。 なのにどうしてさらに掘り下げる必要があるのか？ それは、GitHub には GitHub Actions という独自の CI/CD プラットフォームがあり、詳しく見ておく価値があるからです。 GitHub Actions を使えば、外部のサービスを使用する必要はありません。

この記事では、GitHub Actions を使って InterSystems Package Manager のサーバー部分である ZPM-registry を Google Kubernetes Engine (GKE) にデプロイする、ということを試したいと思います。

どのシステムもそうですが、ビルド / デプロイのシステムも結局は「これを実行して、そこに移動したら、あれを実行する」といった感じの流れになります。 GitHub Actions では、こうようなアクションはそれぞれが 1 つ以上のステップで構成されるジョブとして扱われ、集合的にはワークフローとして知られています。 GitHub は、ワークフローの説明を探して *.github/workflows* ディレクトリの YAML ファイル (拡張子が .yml または .yaml のファイル) を検索します。 詳細は、GitHub Actions の主要コンセプトをご覧ください。

これから実行するアクションはすべて ZPM-registry リポジトリのフォーク内で実行されます。 この記事では、このフォークを "*zpm-registry*" と呼び、そのルートディレクトリーを "*<rootrepodir>* " と呼びます。 ZPM アプリケーションそのものに関する詳細は、Introducing InterSystems ObjectScript Package Manager および The Anatomy of ZPM Module: Packaging Your InterSystems Solution をご覧ください。

すべてのコードサンプルは、簡単にコピー & ペーストできるよう、こちらのリポジトリに保管されています。 必須事項は、CircleCI ビルドで GKE の作成を自動化すると題した記事に記載されている内容と同じです。

過去の記事を読まれていること、Google アカウント を既にお持ちであること、そして前回の記事で「Development」と名付けたプロジェクトを作成されているという前提で進めていきます。 この記事では、その ID を <PROJECTID>として表示しています。 下の例を実行する場合は、ご自身のプロジェクトの ID に変更してください。

Google には無料のティアがありますが、Google クラウドプラットフォームは有料ですのでご注意ください。 経費の管理にはお気を付けください。

ワークフローの基本

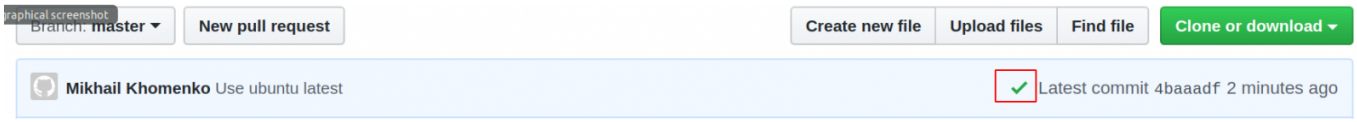
それでは、始めましょう。

以下はシンプルですが使いものにならないワークフローの例です。

```
  $ cd <rootrepodir>
$ mkdir -p .github/workflows
$ cat <rootrepodir>/.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!
```

リポジトリにプッシュするときは、「Greeting」と名付けたジョブを実行する必要があります。このジョブは唯一のステップとしてウェルカムフレーズを出力します。 このジョブは、GitHub でホストされる仮想マシン「Runner」で実行されます。同マシンには Ubuntu の最新バージョンがインストールされています。
このファイルをリポジトリにプッシュした後に「Code GitHub」タブを開けば、すべて順調に実行されたことを確認できるはずです。



ジョブが失敗したら、緑のチェックマークの代わりに赤の「X」が表示されます。
詳細を見るには、緑のチェックマークと「Details」を順にクリックします。
もしくは、直接「Actions」タブに移動することもできます。

ワークフローの構文に関する完全な詳細は、Workflow syntax for GitHub Actions と題したヘルプドキュメントをご覧ください。

リポジトリにイメージビルド用の Dockerfile が含まれている場合は、「Hello world」のステップをもっと役に立つものに置き換えることができます。こちらは starter-workflows から選んだ 1 例です。

```
steps:
- uses: actions/checkout@v2
- name: Build the Docker image
run: docker build . --file Dockerfile --tag my-image:$(date +%s)
```

新しいステップ "uses: action/checkout@v2" が追加されていることに注目してください。 "checkout" という名前からも、リポジトリをクローンするステップであるのは分かりますが、詳細はどこを見ればいいのか？

CircleCI については、便利なステップがたくさんありますが、書き直す必要はありません。
その代わりに、Marketplace という共有リソースから取り入れることができます。
そこで実行したいアクションを探します。「By actions」とマーキングされているアクション (カーソルを合わせると、"Creator verified by Github" と表示されるアクション) をおすすめします。

ワークフローの "uses" 句には、独自のモジュールを記述する代わりに、既成のモジュールを使用する意図が示されています。

アクションの実装そのものは、どの言語でも記述できますが、JavaScript が推奨されます。 JavaScript (または TypeScript) で書かれたアクションは、直接 Runner のマシンで実行されます。
他のかたちで実装する場合は、ユーザーが指定する Docker コンテナが希望する内部環境で実行されますが、もちろんその速度は少し遅くなります。
アクションの詳細は、そのままのタイトルが付けられた記事 About actions をお読みください。

checkout アクションは TypeScript で書かれています。 また、この記事の例で使う Terraform アクション は、Docker Alpine で起動される標準的な Bash シェルスクリプトです。

クローンしたリポジトリに Dockerfile がありますので、修得した知識を応用してみましょう。 ZPM レジストリのイメージをビルドし、Google Container Registry にプッシュします。
並行して、このイメージを実行する Kubernetes クラスターを作成します。これには Kubernetes のマニフェストを使用します。

私たちの計画を GitHub が理解できる言語で記述すると以下のようになります
(見やすくするために行をたくさん省いて俯瞰したものですので、このコンフィグは使わないでください)。

```
  name: ワークフローの説明
# Trigger condition. In this case, only on push to ‘ master ’ branch
on:
  push:
  branches:
  - master
# ここではすべての後続のジョブとそれらの各ステップで
# 使用可能な環境変数について説明しています
# これらの変数は GitHub Secrets ページで初期化できます
# それらを参照する目的で “ ${{ secrets }} ” を追加しています
env:
  PROJECTID: ${{ secrets.PROJECTID }}

# ジョブリストの定義 ジョブ / ステップの名前はランダムなもので構いませんが
# 有意義なほうがベターです
jobs:
  gcloud-setup-and-build-and-publish-to-GCR:
  name: gcloud ユーティリティをセットアップし、ZPM イメージをビルドし、Container Registry に公開する
  runs-on: ubuntu-18.04
  steps:
  - name: チェックアウト
  - name: gcloud cli をセットアップする
  - name: gcloud を認証情報ヘルパーとして使用するよう Docker を設定する
  - name: ZPM イメージをビルドする
  - name: ZPM イメージを Google Container Registry に公開する

  gke-provisioner:
  name: Provision GKE cluster
  runs-on: ubuntu-18.04
  steps:
  - name: チェックアウト
  - name: Terraform 初期化
  - name: Terraform 検証
  - name: Terraform プラン
  - name: Terraform 適用

  kubernetes-deploy:
  name: Kubernetes マニフェストを GKE クラスタにデプロイする
  needs:
  - gcloud-setup-and-build-and-publish-to-GCR
  - gke-provisioner
  runs-on: ubuntu-18.04
  steps:
  - name: チェックアウト
  - name: プレースホルダーをステートフルセットのテンプレートに記載の値に置換する
  - name: gcloud cli をセットアップする
  - name: Kubernetes のマニフェストを適用する
```

これが実用的な config のスケルトン (骨格だけのコード)

で、その筋肉、つまり各ステップで実行されるアクションは含まれていません。
アクションはシンプルなコンソールコマンドで実行できます ("run"、複数のコマンドがあれば "run ¦")。
```
 - name: gcloud を認証情報ヘルパーとして使用するよう Docker を設定する
 run: ¦
 gcloud auth configure-docker
```

アクションは "uses" を使ってモジュールとして起動することもできます。
```
 - name: チェックアウト
 uses: actions/checkout@v2
```

デフォルトでは、すべてのジョブが並列して実行され、各ステップが順番に処理されます。 しかし、"needs" を使えば、そのジョブを残りのジョブが完了するまで待機するジョブとして指定できます。
```
 needs:
- gcloud-setup-and-build-and-publish-to-GCR
- gke-provisioner
```

ちなみに、このように待機するジョブが GitHub の Web インターフェイスに表示されるのは、待機される側のジョブが実行されるときだけです。

"gke-provisioner" ジョブには過去の記事の中で検証した Terraform が記述されています。 GCP 環境で操作する場合の事前設定は、便宜上、別の markdown file で繰り返されます。
以下のリンクもご利用いただくと便利です。

- Terraform Apply Subcommand documentation
- Terraform GitHub Actions repository
- Terraform GitHub Actions documentation

"kubernetes-deploy" ジョブには、"Apply Kubernetes manifests" と呼ばれるステップがあります。
今回は、Deploying InterSystems IRIS Solution into GCP Kubernetes Cluster GKE Using CircleCI という記事に記載されている通りにマニフェストを使用しますが、少しだけ変更を加えます。

過去の記事で使った IRIS アプリケーションは、ステートレスでした。
つまり、ポッドを再起動したら、すべてのデータがデフォルトの場所に戻っていたのです。
これは良いことで、必要になることも多々あります。しかし、ZPM レジストリでは、ポッドを何回再起動する必要があっても、読み込まれたパッケージを何とかして保存する必要があります。
デプロイすれば出来るのですが、もちろんそれには制限があります。

ステートフルなアプリケーションには、StatefulSet のリソースを選択する方が無難です。
メリットとデメリットは、Deployments vs. StatefulSets の GKE ドキュメンテーションに関するトピックおよび Kubernetes Persistent Volumes with Deployment and StatefulSet と題したブログ記事に記載しています。

StatefulSet のリソースは、リポジトリの中にあります。 注目したいのは、以下の部分です。
```
 volumeClaimTemplates:
- metadata:
 name: zpm-registry-volume
 namespace: iris
 spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 10Gi
```

このコードは、単一の Kubernetes ワーカーノードによってマウント可能な 10GB の読み取り / 書き込みディスクを作成します。 このディスク (およびその中のデータ) はアプリケーションの再起動後も残ります。 StatefulSet 全体を削除しても残ります。そのためには正しい Reclaim Policy を設定する必要がありますが、この記事ではカバーしていません。

ワークフローを起動させる前に、GitHub Secrets に変数をあといくつか追加しておきましょう。

以下のテーブルはこれらの設定の意味を説明するものです (サービスアカウントキー も含まれています):

名前	意味	例
GCRLOCATION	グローバル GCR ロケーション	eu.gcr.io
GKECLUSTER	GKE クラスタ名	dev-cluster
GKEZONE	イメージを格納するゾーン	europe-west1-b
IMAGENAME	イメージのレジストリ名	zpm-registry
PROJECTID	GCP プロジェクト ID	possible-symbol-254507
SERVICEACCOUNTKEY	GitHub が GCP に接続する際に使用する JSON key。重要: Base64 でエンコードされている必要があります (下の注意点をお読みください)	ewogICJ0eXB...
TFSERVICEACCOUNTKEY	Terraform が GCP に接続する際に使用する JSON key (下の注意点をお読みください)	{…}

SERVICEACCOUNTKEY において、JSON-key に、例えば、key.json のような名前が付いている場合は、下のコマンドを実行します。

```
$ base64 key.json ¦ tr -d '\n'
```

TFSERVICEACCOUNTKEY について、その権限は　CircleCI ビルドで GKE の作成を自動化する と題した記事にて説明してあります。

SERVICEACCOUNTKEY のちょっとした注意点: 私がやってしまったように、base64 フォーマットに変換するのを忘れてしまうと、以下のような画面が表示されます。

ワークフローの主要部分を確認し、必要な変数を追加したところで、ワークフローの完全版を検証する準備が整いました (*<rootrepodir>/.github/workflow/workflow.yaml*)。

```
 name: ZPM レジストリのイメージを構築し、GCR にデプロイする。 GKE を実行。 ZPM レジストリを GKE で実行する。
on:
 push:
 branches:
 - master
# 環境変数。
# ${{ secrets }} は GitHub -> Settings -> Secrets より取得されます
```

```
# ${{ github.sha }} はコミットハッシュです
env:
  PROJECTID: ${{ secrets.PROJECTID }}
  SERVICEACCOUNTKEY: ${{ secrets.SERVICEACCOUNTKEY }}
  GOOGLECREDENTIALS: ${{ secrets.TFSERVICEACCOUNTKEY }}
  GITHUBSHA: ${{ github.sha }}
  GCRLOCATION: ${{ secrets.GCRLOCATION }}
  IMAGENAME: ${{ secrets.IMAGENAME }}
  GKECLUSTER: ${{ secrets.GKECLUSTER }}
  GKEZONE: ${{ secrets.GKEZONE }}
  K8SNAMESPACE: iris
  STATEFULSETNAME: zpm-registry

jobs:
  gcloud-setup-and-build-and-publish-to-GCR:
  name: gcloud ユーティリティをセットアップ、ZPM イメージを構築、Container Registry に公開する
  runs-on: ubuntu-18.04
  steps:
  - name: チェックアウト
  uses: actions/checkout@v2

  - name: gcloud cli をセットアップする
  uses: GoogleCloudPlatform/github-actions/setup-gcloud@master
  with:
  version: '275.0.0'
  serviceaccountkey: ${{ secrets.SERVICEACCOUNTKEY }}

  - name: gcloud を認証情報ヘルパーとして使用するよう Docker を設定する
  run: |
  gcloud auth configure-docker

  - name: ZPM イメージを構築する
  run: |
  docker build -t ${GCRLOCATION}/${PROJECTID}/${IMAGENAME}:${GITHUBSHA} .

  - name: ZPM イメージを Google Container Registry に公開する
  run: |
  docker push ${GCRLOCATION}/${PROJECTID}/${IMAGENAME}:${GITHUBSHA}

  gke-provisioner:
  # Inspired by:
  ## https://www.terraform.io/docs/github-actions/getting-started.html
  ## https://github.com/hashicorp/terraform-github-actions
  name: GKE クラスタをプロビジョンする
  runs-on: ubuntu-18.04
  steps:
  - name: チェックアウト
  uses: actions/checkout@v2

  - name: Terraform 初期化
  uses: hashicorp/terraform-github-actions@master
  with:
  tfactionsversion: 0.12.17
  tfactionssubcommand: 'init'
  tfactionsworkingdir: 'terraform'

  - name: Terraform 検証
  uses: hashicorp/terraform-github-actions@master
  with:
```

```
tfactionsversion: 0.12.17
tfactionssubcommand: 'validate'
tfactionsworkingdir: 'terraform'

- name: Terraform プラン
uses: hashicorp/terraform-github-actions@master
with:
tfactionsversion: 0.12.17
tfactionssubcommand: 'plan'
tfactionsworkingdir: 'terraform'

- name: Terraform 適用
uses: hashicorp/terraform-github-actions@master
with:
tfactionsversion: 0.12.17
tfactionssubcommand: 'apply'
tfactionsworkingdir: 'terraform'

kubernetes-deploy:
name: Kubernetes マニフェストを GKE クラスタにデプロイする
needs:
- gcloud-setup-and-build-and-publish-to-GCR
- gke-provisioner
runs-on: ubuntu-18.04
steps:
- name: チェックアウト
uses: actions/checkout@v2

- name: プレースホルダーをステートフルセットのテンプレートに記載の値に置換する
working-directory: ./k8s/
run: |
cat statefulset.tpl |\
sed "s|DOCKERREPONAME|${GCRLOCATION}/${PROJECTID}/${IMAGENAME}|" |\
sed "s|DOCKERIMAGETAG|${GITHUBSHA}|" > statefulset.yaml
cat statefulset.yaml

- name: gcloud cli をセットアップする
uses: GoogleCloudPlatform/github-actions/setup-gcloud@master
with:
version: '275.0.0'
serviceaccountkey: ${{ secrets.SERVICEACCOUNTKEY }}

- name: Kubernetes マニフェストを適用する
working-directory: ./k8s/
run: |
gcloud container clusters get-credentials ${GKECLUSTER} --zone ${GKEZONE} --project ${PROJECTID}
kubectl apply -f namespace.yaml
kubectl apply -f service.yaml
kubectl apply -f statefulset.yaml
kubectl -n ${K8SNAMESPACE} rollout status statefulset/${STATEFULSETNAME}
```

リポジトリにプッシュする前には、terraform-code を github-gke-zpm-registry リポジトリの Terraform ディレクトリーから取得し、プレースホルダーを *main.tf* のコメントに記載されている通りに置換し、*terraform/* ディレクトリーに配置する必要があります。 Terraform は、リモートのバケットを使用しますが、このバケットは最初から CircleCI ビルドで GKE の作成を自動化する と題した記事で言及されている通りに作成されている必要があることを覚えておきましょう。

また、Kubernetes-code は github-gke-zpm-registry リポジトリの K8S ディレクトリーから取得され、*k8s/* ディレクトリーの中に配置されている必要があります。
これらのコードのソースは、スペースを節約するためにこの記事では省いています。

そして、以下のようにすればデプロイをトリガーできます。

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

フォークされている ZPM リポジトリに変更内容をプッシュしたら、説明したステップの実装を確認できます。

ここまで、ジョブの数は 2 つしかありませんが、 3 つ目の "kubernetes-deploy" は、その 2 つのジョブに依存しており、それらが完了した後に表示されます。
Docket イメージのビルドと公開には少し時間がかかります。

また、結果は GCR コンソールで確認できます。

"Provision GKE cluster" ジョブは、最初の実行時に GKE クラスターを作成するので、最初だけ完了までの時間が少し長くなります。
数分間、待機中の画面が表示されます。

やっと完了したときには、思わず嬉しくなります。

Kubernetes のリソースも喜んでいます。

```
  $ 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
```

他の内容の確認は、Running ステータスになるまで待機することをおすすめします。

```
  $ 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
```

ディスクまでが喜んでいます。

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

そして、一番喜んでいるのは ZPM レジストリです ("*kubectl -n iris get svc*" の External-IP 出力を使用しました):

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

ログイン / パスワードを HTTP で処理しているのは残念ですね。今後の記事の中で何とかしたいと思います。

ちなみにですが、エンドポイントについてはソースコード
を見ていただければ、詳細が書かれていますので、*XData UrlMap* セクションをご覧ください。

このリポジトリは、それ自体にパッケージをプッシュすればテストできます。 GitHub
のダイレクトリンクだけをプッシュする便利な機能があります。 InterSystems ObjectScript の数式ライブラリ
で試してみましょう。 これをローカルマシンから実行します。

```
  $ 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"]}]
```

ポッドを再起動して、データがきちんと配置されていることを確認します。

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

実行中のポッドを待ちます。 そして、うまく処理されると以下が表示されます。

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

それでは、この数式パッケージをリポジトリからローカルの IRIS インスタンスにインストールしましょう。 ZPM クライアントが既にインストールされているものを選びます。

```
  $ 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
```

おめでとうございます！
いらなくなった GKE クラスタは、忘れずに削除しておきましょう。

まとめ

InterSystems のコミュニティには GitHub Actions のリファレンスがそれほど多くありません。
見つかったのは、エキスパート @mdaimor の メンション 1 つのみです。 ですが、GitHub Actions はコードを GitHub に保管するディベロッパーにとって非常に重宝すると思われます。 ネイティブアクションは JavaScript でしかサポートされていませんが、これはディベロッパーの多くがコードを使ってステップを説明することに慣れており、そうすることが望ましいということかもしれません。 いずれにしても、JavaScript に詳しくない方は Docker アクションを使えばいいと思います。

GitHub Actions の UI に関して、使っているうちに不便に感じたことがいくつかあり、知っておくべきだと思うことを紹介しておきます。

ジョブのステップが完了するまで、その状況を確認できない。 "Terraform apply" のステップのように、クリックすることができない。
失敗したワークフローは再実行できる一方で、成功したワークフローを再実行する方法が見つからなかった。
2 つ目のポイントの次善策として、次のコマンドを使います。
$ git commit --allow-empty -m "trigger GitHub actions"

これに関する詳細は、StackOverflow に投稿されている How do I re-run Github Actions? (Github Actions はどうすれば再実行できるか？) という質問をご覧ください。

#DevOps #Docker #GitHub #Kubernetes #クラウド #コンテナ化 #InterSystems IRIS #Open Exchange
InterSystems Open Exchangeで関連アプリケーションを確認してください

ソースURL:https://jp.community.intersystems.com/post/github-actions-%E3%82%92%E4%BD%BF%E3%81%A3%E3%81%A6-gke-%E3%81%AB-intersystems-iris-solution-%E3%82%92%E3%83%87%E3%83%97%E3%83%AD%E3%82%A4%E3%81%99%E3%82%8B