Article
Mikhail Khomenko · Jan 5, 2022 8m read

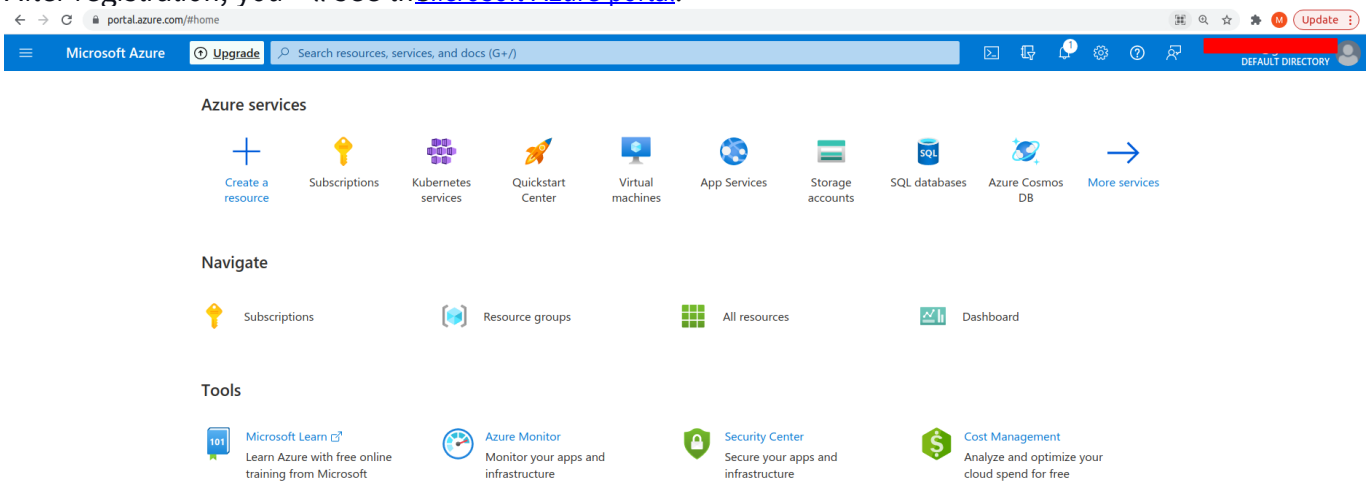
Deploy IRIS Application to Azure Using CircleCI

We ' ve already considered how to run an IRIS-based application in GCP Kubernetes in Deploying InterSystems IRIS Solution into GCP Kubernetes Cluster GKE Using CircleCI. Additionally, we ' ve seen how to run an IRIS-based application in AWS Kubernetes in Deploying a Simple IRIS-Based Web Application Using Amazon EKS. Now, let ' s look at how to deploy an application to the Azure Kubernetes Service (AKS).

Azure

For this article, we ' ll use an Azure free subscription. You can find pricing details on their pricing page.

After registration, you ' ll see the Microsoft Azure portal:



The portal is handy, but we won ' t use it in this article. Instead, let ' s install the Azure command-line interface. The most recent version at the moment of writing is 2.30.0.

```
$ az version
{
  "azure-cli": "2.30.0",
  "azure-cli-core": "2.30.0",
  "azure-cli-telemetry": "1.0.6",
  "extensions": {}
}
```

Now let ' s log in to Azure:

```
$ az login
```

CircleCI Pipeline

We ' re going to set up AKS and install an IRIS application using the power of CI/CD with CircleCI. This means that we take a GitHub-based project, add a couple of pipeline files along with infrastructure as code, push changes back to GitHub, and check the results in a friendly CircleCI UI.

With a GitHub account, it ' s effortless to create an integration with CircleCI. For more information, see this article on Seamless integration with GitHub.

Let ' s take an updated version of the project we ' ve already used in Deploying InterSystems IRIS Solution into GCP Kubernetes Cluster GKE Using CircleCI — namely, secured-rest-api. Open it, click Use this Template, and create a version in a new repository. We ' ll refer to code samples located there throughout this article.

Clone a repository locally and create a .circleci/ directory with a couple of files:

```
$ tree .circleci/
.circleci/
??? config.yml
??? continue.yml
```

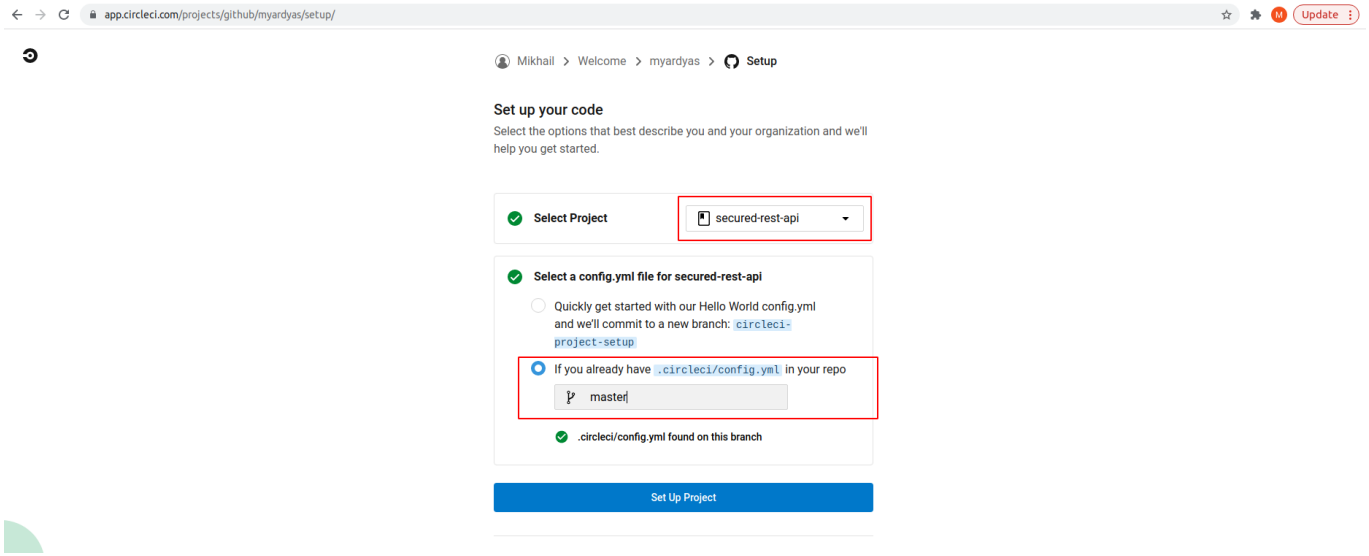
We use a dynamic configuration and path filtering to enable the pipeline to run as a whole or just its parts depending on which files have changed. In our case, we run a Terraform job only when Terraform code changes. The first file, config.yml, is simple. We call a second part, .circleci/continue.yml, and pass a particular Boolean parameter if the Terraform code is up-to-date.

```
$ cat config.yml
version: 2.1
# Enable CircleCI's dynamic configuration feature
setup: true
# Enable path-based pipeline
orbs:
  path-filtering: circleci/path-filtering@0.1.0
workflows:
  Generate dynamic configuration:
    jobs:
      - path-filtering/filter:
          name: Check updated files
          config-path: .circleci/continue.yml
          base-revision: master
          mapping: |
            terraform/.* terraform-job true
```

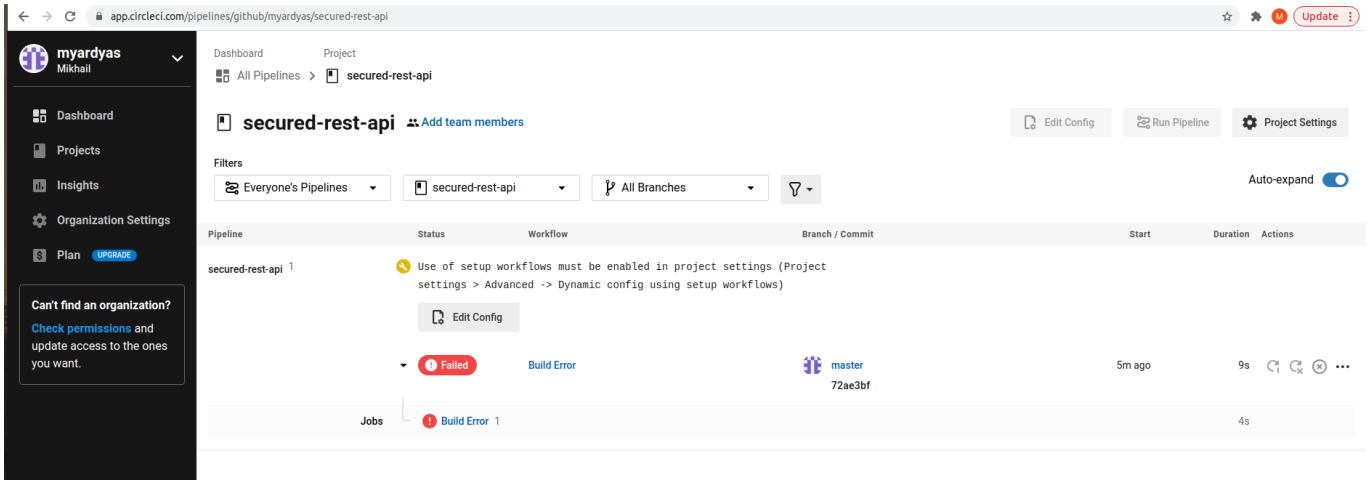
Before discussing a second file, continue.yml, let ' s add this secured-rest-app project to CircleCI and push updates with .circleci/config.yml to GitHub:

```
$ git add .circleci/config.yml
$ git commit -m "Add circleci config.yml"
$ git push
```

Then, open the CircleCI Projects page, choose your project, and click Set Up Project.



Follow the provided recommendation and enable Setup Workflow (for more information, see Getting started with dynamic config in CircleCI):



Now we ' re ready to continue with the second file, continue.yml. Its structure is as follows:

- Version indicates the CircleCI pipeline version.
- Parameters is a variable to decide if Terraform should be running or not.
- Orbs are parts of configuration created by others which we could reuse.
- Executors are docker images with an Azure command line for some of our jobs.
- Jobs are the actual deployment steps.
- Workflows are logic to run a pipeline with or without Terraform.

The Jobs section contains the following jobs:

- Build and push Docker image to ACR: This job runs inside a docker image with the az command-line tool installed. It logs in to Azure and builds and pushes an image to the Azure Container Registry (ACR).

- Terraform: This job uses a Terraform orb and creates an infrastructure. See the section on Terraform below for details.
- Setup packages: This job installs an IRIS application and a couple of service applications. See the Setup Packages section below for details.

Terraform

For infrastructure creation, we ' re going to use an infrastructure as code approach and leverage the power of Terraform. Terraform speaks with AKS using its Azure plugin. It ' s handy to use an AKS Terraform module that plays as a wrapper and simplifies resource creation.

You can find an example of creating an AKS resource with Terraform in Creating a Kubernetes Cluster with AKS and Terraform. Here, we enable Terraform to manage all resources for demo purposes and simplicity, that is, assign an Owner role. Terraform as an application connects to Azure using Service Principal. So, to be more accurate, we assign an owner role to Service Principal as described in Create an Azure service principal with the Azure CLI.

Let ' s run a couple of commands on a local machine. Save the Azure subscription ID in an environment variable:

```
$ export AZ_SUBSCRIPTION_ID=$(az account show --query id --output tsv)
$ az ad sp create-for-
rbac -n "Terraform" --role="Owner" --scopes="/subscriptions/${AZ_SUBSCRIPTION_ID}"
…
{
  "appId": "<appId>",
  "displayName": "<displayName>",
  "name": "<name>",
  "password": "<password>",
  "tenant": "<tenant>"
}
```

You can later find appId and tenantId listing Service Principals and looking for the display name Terraform:

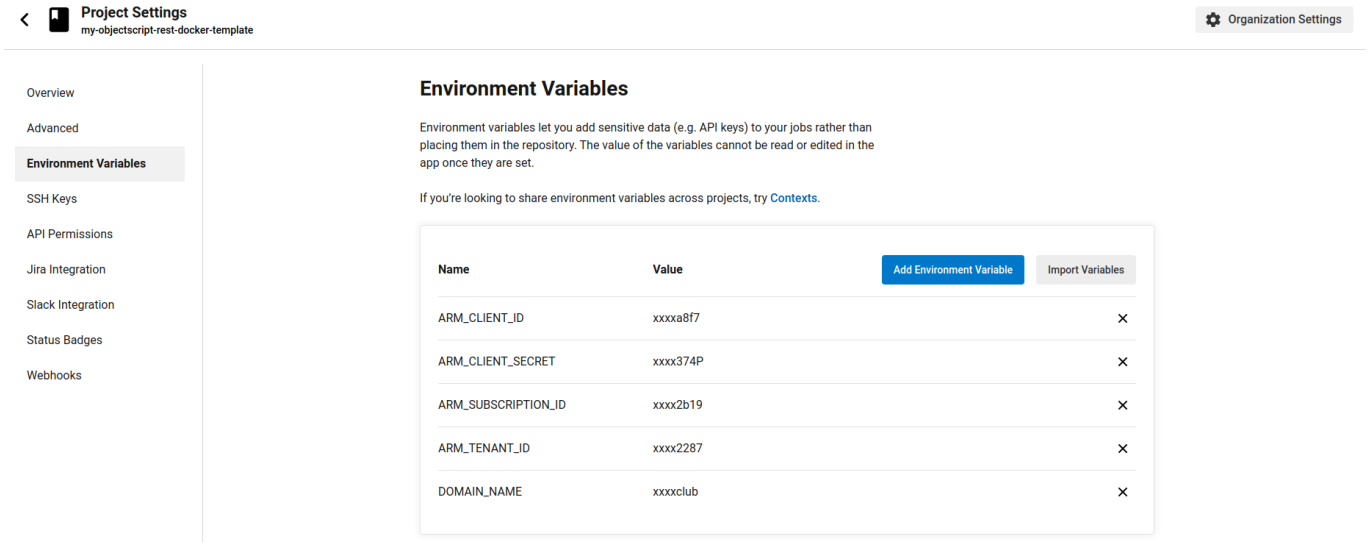
```
$ az ad sp list --display-
name "Terraform" | jq '.[] | "AppId: \(.appId), TenantId: \(.appOwnerTenantId)"'
```

But you can ' t see the password this way. If you forget your password, the only way is to reset credentials.

In a pipeline, for AKS creation, we use a publicly available Azure Terraform module and Terraform version 1.0.11.

Set the environment variables in the CircleCI project settings with the retrieved credentials that Terraform uses for connections to Azure. Also, set the DOMAINNAME environment variable. This tutorial uses the demo-iris.myardyas.club domain name, but you ' ll use your registered domain name. We use this variable in a pipeline to enable external access to the IRIS application. The mapping of CircleCI variables with the az create-for-rbac command is as follows:

```
ARM_CLIENT_ID: appId
ARM_CLIENT_SECRET: password
ARM_TENANT_ID: tenant
ARM_SUBSCRIPTION_ID: Value of environment variable AZ_SUBSCRIPTION_ID
DOMAIN_NAME: your domain name
```



To enable the Terraform Remote state, we use Terraform state in Azure Storage. To achieve this, let ' s run these commands on a local machine.

```
$ export RESOURCE_GROUP_NAME=tfstate
$ export STORAGE_ACCOUNT_NAME=tfstate14112021 # Must be between 3 and 24 characters i
n length and use numbers and lower-case letters only
$ export CONTAINER_NAME=tfstate

# Create resource group
$ az group create --name ${RESOURCE_GROUP_NAME} --location eastus

# Create storage account
$ az storage account create --resource-group ${RESOURCE_GROUP_NAME} --name ${STORAGE_
ACCOUNT_NAME} --sku Standard_LRS --encryption-services blob

# Enable versioning. Read more at https://docs.microsoft.com/en-
us/azure/storage/blobs/versioning-overview
$ az storage account blob-service-properties update --account-
name ${STORAGE_ACCOUNT_NAME} --enable-versioning true

# Create blob container
$ az storage container create --name ${CONTAINER_NAME} --account-
name ${STORAGE_ACCOUNT_NAME}
```

The Terraform code we put in the Terraform directory. It's divided into three files:

- provider.tf is to set the Azure plugin version and a path to remote storage for saving the Terraform state.
- variables.tf is input data for the Terraform module.
- main.tf is the creation of the actual resources.

We create an Azure resource group, public IP, Azure container registry, and so on. For networking and the Azure Kubernetes service, we leverage publicly available Terraform modules.

Setup Packages

What we ' re going to install into the newly created AKS cluster is located in the helm directory. The descriptive Helmfile approach enables us to define applications and their settings in the helmfile.yaml file.

Run the setup with the single command helmfile sync. The command installs an IRIS application and two additional applications, cert-manager and ingress-nginx, allowing us to call an application from the outside. For more information, see the releases section on GitHub.

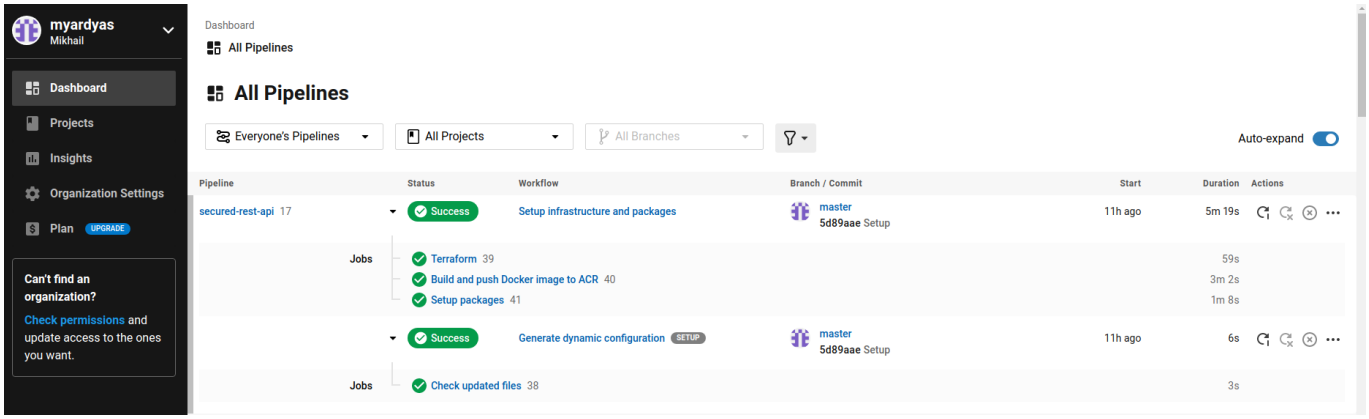
We install the IRIS application using a Helm chart similar to that described in Automating GKE creation on CircleCI builds. For simplicity, we use deployment. That means data doesn ' t persist during the pod ' s restart. For persistence, you should use Statefulset or, better, Kubernetes IRIS Operator (IKO). You can find an example of IKO deployment in the iris-k8s-monitoring repository.

Running the Pipeline

When you ' ve added the .circleci/, terraform/ and helm/ directories, push them into GitHub:

```
$ git add .
$ git commit -m "Setup everything"
$ git push
```

If everything is okay, you see a screen in the CIrcleCI UI that ' s similar to the following:



Setting the A-record in the Domain Registrar

One more thing is the creation of a binding by A-record between a public IP created in Azure by Terraform and your domain name in your Domain Registrar console.

Let ' s connect to a cluster:

```
$ az aks get-credentials --resource-group demo --name demo
```

Define a public IP-address exposed by ingress-nginx:

```
$ kubectl -n ingress-nginx get service ingress-nginx-
controller -ojsonpath='{.spec.loadBalancerIP}'
x.x.x.x
```

Set this IP in your domain registrar (GoDaddy, Route53, GoogleDomains, and so on) like this:

```
YOUR_DOMAIN_NAME = x.x.x.x
```

Now, wait for some time until the DNS change is propagated around the world and you can check the result:

```
$ dig +short YOUR_DOMAIN_NAME
```

The response should be x.x.x.x.

Testing

Assuming that the domain name is demo-iris.myardyas.club, we can perform manual testing. Note that we ' ve used the letsencrypt staging issuer, so let's omit certificate checking here. In production, we should replace the issuer with lets-encrypt-production here. Also, it ' s worth setting your email here instead of at example@gmail.com.

```
$ curl -sku Bill:ChangeMe https://demo-iris.myardyas.club/crudall/_spec | jq .
…
```

Create a person:

```
$ curl -ku John:ChangeMe -XPOST -H "Content-Type: application/json"  https://demo-
iris.myardyas.club/crud/persons/ -d '{"Name":"John Doe"}'
```

Check to see if a person was created:

```
$ curl -sku Bill:ChangeMe https://demo-iris.myardyas.club/crud/persons/all | jq .
[
  {
    "Name": "John Doe"
  }
...
```

Conclusion

That ' s it! You've seen how a Terraform and CircleCI workflow creates a Kubernetes cluster in an Azure cloud. For our IRIS installation, we used the most straightforward Helm chart. For production, you should extend this chart, at least deployment should be replaced by Statefulset, or you should use IKO.

Don ' t forget to remove created resources when you no longer need them. Although Azure has free tier and AKS is free, you pay for resources designed to run an AKS cluster.

#Azure #DevOps #Kubernetes #InterSystems IRIS

Source URL:https://community.intersystems.com/post/deploy-iris-application-azure-using-circleci