Article
Eduard Lebedyuk · Mar 5, 2022 6m read

Running InterSystems IRIS in a FaaS mode with Kubeless

> Function as a service (FaaS) is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app. Building an application following this model is one way of achieving a "serverless" architecture, and is typically used when building microservices applications.
>
> Wikipedia

FaaS is an extremely popular approach to running workloads in the cloud, allowing developers to focus on writing code.

This article will show you how to deploy InterSystems IRIS methods in a FaaS way.

Install Kubernetes

First of all, install **Kubernetes 1.16**. There are a lot of guides available so that I won't be copying them here, but I'm using minicube. With minicube to run kubernetes, it's enough to execute this command:

```
minikube start --kubernetes-version=v1.16.1
```

Install kubeless

Next, we will install kubeless. kubeless is a Kubernetes-native serverless framework that lets you deploy small bits of code without worrying about the underlying infrastructure plumbing. It leverages Kubernetes resources to provide auto-scaling, API routing, monitoring, troubleshooting, and more.

```
kubectl create ns kubeless
kubectl create -f https://github.com/kubeless/kubeless/releases/download/v1.0.8/kubel
ess-v1.0.8.yaml
kubectl get pods -n kubeless
```

Output should be something like this:

```
NAME                                  READY  STATUS   RESTARTS  AGE
kubeless-controller-manager-666ffb749-26vhh  3/3   Running  0      83s
```

You also need to install a kubeless client (on the same instance you have kubectl). You can get it here. Installation on Linux is as simple as:

```
sudo install kubeless /usr/local/bin/kubeless
```

Test kubeless

First, let's deploy a simple Python function to check that kubeless works.

Create *test.py*:

```
def hello(event, context):
  return event['data']
```

To read more about function environment check this doc, generally function accepts two arguments - event and context with this data:

```
event:
  data:                                        # Event data
    foo: "bar"                                 #
 The data is parsed as JSON when required
  event-id: "2ebb072eb24264f55b3fff"           # Event ID
  event-type: "application/json"               # Event content type
  event-time: "2009-11-10 23:00:00 +0000 UTC"  # Timestamp of the event source
  event-namespace: "kafkatriggers.kubeless.io" # Event emitter
  extensions:                                  # Optional parameters
    request: ...                               # Reference to the request received
    response: ...                              # Reference to the response to send

           # (specific properties will depend on the function language)
context:
    function-name: "pubsub-nodejs"
    timeout: "180"
    runtime: "nodejs6"
    memory-limit: "128M"
```

Now we can deploy our hello function by specifying our file with a function and a runtime:

```
kubeless function deploy hello --runtime python3.7 --from-
file test.py --handler test.hello
kubeless function ls hello
```

And let's test it:

```
kubeless function call hello --data 'Hello world!'
```

You should receive *Hello World!* as an answer.

Add IRIS config

Next we need to add an InterSystems IRIS function handler, to do that open kubeless config for edit:

```
kubeless get-server-config
kubectl get -n kubeless configmaps -o yaml > configmaps.yaml
kubectl edit -n kubeless configmaps
```

Add this entry to runtime-images array and save:

```
{"ID": "iris","depName": "","fileNameSuffix": ".cls","versions": [{"images": [{"image
": "eduard93/kubeless-iris-
runtime:latest","phase": "runtime"}],"name": "iris2022.1","version": "2022.1"}]}
```

Restart kubeless controller for the changes to take effect.

```
kubectl delete pod -n kubeless -l kubeless=controller
```

Build IRIS function CRD and publish it

Now let's write our first function in InterSystems IRIS:

```
Class User.Test {

ClassMethod hi(event, context) As %Status
{
    if $isObject(event) {
        write event.Text + event.Text
    } else {
        write "HELLO FROM IRIS"
    }
    quit $$$OK
}
}
```

Next, we need to build a function CRD:

Here's our template:

function.yaml

And we need to fill:

- *name*: function name (for kubeless)
- *handler*: class.namemethod (for InterSystems IRIS)
- *function body*: add at the end (don't forget tabs!)

So our CRD looks like this:

functiondemo.yaml

This can be easily automated. On Linux execute:

```
sed 's/!name!/iris-
demo/; s/!handler!/User_Test.hi/' function.yaml > function_demo.yaml
sed  's/^/     /'  User.Test.cls >> function_demo.yaml
```

And on Windows (PowerShell):

```
Get-Content function.yaml | ForEach-Object { $_ -replace "!handler!", "User_Test.hi"
-replace "!name!", "iris-demo" } | Set-Content function_demo.yaml
"    " + [string]((Get-Content User.Test.cls)  -join "`r`n    ") | Add-
Content function_demo.yaml
```

Now we need to publish our CRD in kubeless:

```
kubectl apply -f function_demo.yaml
```

Test IRIS function

First, let's see that the function is deployed and ready (can take a few minutes the first time):

```
kubeless function ls
```

And now call it:

```
kubeless function call iris-demo --data '{"Text":123}'
```

If you're on Windows, call the function like this (same for all other calls with escaped double quotes):

```
kubeless function call iris-demo --data '{\"Text\":123}'
```

Anyway, the response should be *456* since *123* is a number.

HTTP access

kubeless also offers HTTP access. To test this, use the kubectl proxy command:

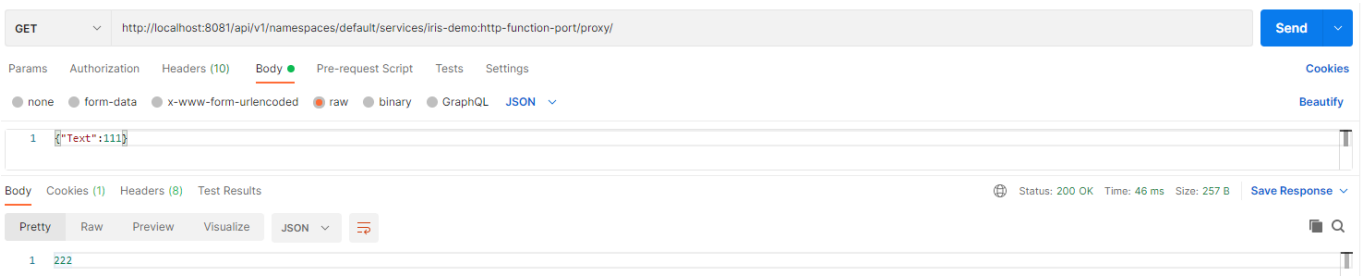
```
kubectl proxy -p 8081
```

Next, send this request using your preferred REST API client:

```
GET http://localhost:8081/api/v1/namespaces/default/services/iris-demo:http-function-
port/proxy/

{"Text":111}
```

Here's how it looks like in Postman:



Next, let's publish it on the internet.

There are two approaches. Preferably configure ingress as described here.

Additionally you can patch function service:

```
kubectl get svc
kubectl patch svc iris-demo -p '{"spec": {"type": "LoadBalancer"}}'
kubectl get svc
```

Clean up

To remove a deployed function call:

```
kubectl delete -f function_demo.yaml
```

Conclusion

While this is undoubtedly a proof-of-concept and not a production-grade solution, this approach demonstrates that it's possible to run InterSystems IRIS workloads using the serverless, FaaS approach.

Links

- Minicube
- Kubeless
- InterSystems IRIS runtime

#Cloud #Docker #InterSystems IRIS

Source URL:https://community.intersystems.com/post/running-intersystems-iris-faas-mode-kubeless