

# Fixing the Auto-Generated Helm Chart

In this lesson, we will fix the auto-generated Helm chart by adding dependencies for MongoDB. Moreover, we will perform checks to confirm that the application is running.

## We'll cover the following

- Adding DB environment variable
- Adding MongoDB to the Helm chart
- Defining values for the dependencies
- Monitor the build activities
- Inspect Pods
- Describe pods
- Modifying the probePath entry
- Final checks for the application

Even though the code of the application is small, I will save you from going through it. Instead, I'll let you know right away what's missing and what parts of the chart need to be added or modified.

## Adding **DB** environment variable #

First of all, the application requires an environment variable called **DB**. The code is using it to obtain the address of the database. That brings us to the first thing missing in the chart generated by Jenkins X; **there is no definition for MongoDB**.

The first step is to open *charts/go-demo-6/templates/deployment.yaml* in your favorite editor. That's where the Deployment for the application is defined, and that's where we need to add the variable.

Please locate the code that follows.

```
...  
    imagePullPolicy: {{ .Values.image.pullPolicy }}  
    env:  
...  

```



Now, add the `env` section with the `name` set to `DB` and the value `{{ template "fullname" . }}-db`. The final version of the snippet listed above should be as follows.

```
...
  imagePullPolicy: {{ .Values.image.pullPolicy }}
  env:
  - name: DB
    value: {{ template "fullname" . }}-db
...
```

Save the file.

## Adding MongoDB to the Helm chart #

Next, we need to add MongoDB to the Helm chart that was created for us. Now, we could start writing Helm templates for the MongoDB StatefulSet and a Service. We could spend time trying to figure out how to replicate data between its replicas, and probably a few other things that might not be obvious from the start.

However, we should know better. We should know that there is already a Helm chart that does that and more. There are quite a few charts we could use, but we'll go with [mongodb from the stable channel](#).

**So, how can we add MongoDB chart to the one we already have?** The answer is `dependencies`. We can make our application depend on MongoDB charts by creating a `requirements.yaml` file.

```
echo "dependencies:
- name: mongodb
  alias: go-demo-6-db
  version: 5.3.0
  repository: https://kubernetes-charts.storage.googleapis.com
  condition: db.enabled
" | tee charts/go-demo-6/requirements.yaml
```

The only things worth noting in that file are `alias` and `condition`.

- The former (`alias`) is set to the value that will create a Service with the same name as the environment variable `DB` that we just added to `deployment.yaml`.
- The latter (`condition`) will allow us to disable this dependency. We'll see later why we might want to do that.

There's only one more thing missing. We should probably customize the MongoDB chart to fit our use case. I won't go through all the values we could set. You can explore them yourself by executing `helm inspect values stable/mongodb` or by visiting [project README](#). Instead, we'll define only one, mostly as an exercise on how to define values for the dependencies.

## Defining values for the dependencies #

So, let's add some new entries to `values.yaml`.

```
echo "go-demo-6-db:
  replicaSet:
    enabled: true
" | tee -a charts/go-demo-6/values.yaml
```

We set only `replicaSet.enabled` to `true`. The important thing to note is that it is nested inside `go-demo-6-db`. That way, Helm will know that the variable is not meant for our application (*go-demo-6*), but for the dependency called (aliased) `go-demo-6-db`.

Now, that we know how to add dependencies to our applications, we should push the changes to GitHub, and check whether that solved our issue.

```
git add .
git commit \
  --message "Added dependencies"
git push
```

## Monitor the build activities #

Next, we need to wait until the new release is deployed to the staging environment. We'll monitor the activity of the new build and wait until its finished.

```
jx get activity \
  --filter go-demo-6 \
  --watch
```

The output, limited to the new build, is as follows:

STEP	STARTED	AGO	DURATION	STATUS
...				
vfarcic/go-demo-6/master #2	2m51s		2m43s	Succeeded Version: 1.0.421
meta pipeline	2m51s		20s	Succeeded
Credential Initializer	2m51s		0s	Succeeded

Credential Initializer Nlnj2	2m51s	0s	Succeeded	
Working Dir Initializer	2m51s	1s	Succeeded	
Place Tools	2m50s	1s	Succeeded	
Git Source Meta Vfaric Go Demo 6 Master R ...	2m49s	4s	Succeeded	<a href="https://github.com/vf">https://github.com/vf</a>
Git Merge	2m45s	1s	Succeeded	
Merge Pull Refs	2m44s	0s	Succeeded	
Create Effective Pipeline	2m44s	2s	Succeeded	
Create Tekton Crds	2m42s	11s	Succeeded	
from build pack	2m30s	2m22s	Succeeded	
Credential Initializer Jk7k5	2m30s	0s	Succeeded	
Working Dir Initializer 4vrhr	2m30s	1s	Succeeded	
Place Tools	2m29s	1s	Succeeded	
Git Source Vfaric Go Demo 6 Master Releas ...	2m28s	4s	Succeeded	<a href="https://github.com/vf">https://github.com/vf</a>
Git Merge	2m24s	1s	Succeeded	
Setup Jx Git Credentials	2m23s	0s	Succeeded	
Build Make Build	2m23s	20s	Succeeded	
Build Container Build	2m3s	3s	Succeeded	
Build Post Build	2m0s	1s	Succeeded	
Promote Changelog	1m59s	6s	Succeeded	
Promote Helm Release	1m53s	14s	Succeeded	
Promote Jx Promote	1m39s	1m31s	Succeeded	
Promote: staging	1m34s	1m26s	Succeeded	
PullRequest	1m34s	1m25s	Succeeded	PullRequest: <a href="https://">https://</a>
Update	9s	1s	Succeeded	
Promoted	9s	1s	Succeeded	Application is at: h

The **Promoted** step is the last one in that build. Once we reach it, we can stop monitoring the activity by pressing *ctrl+c*.

## Inspect Pods #

Let's see whether we got the Pods that belong to the database and, more importantly, whether the application is indeed running.

```
kubectl --namespace jx-staging get pods
```

The output is as follows:

NAME	READY	STATUS	RESTARTS	AGE
jx-go-demo-6-...	0/1	Running	5	5m
jx-go-demo-6-db-arbiter-0	1/1	Running	0	5m
jx-go-demo-6-db-primary-0	1/1	Running	0	5m
jx-go-demo-6-db-secondary-0	1/1	Running	0	5m



Please note that it might take a minute or two after the application pipeline activity is finished for the application to be deployed to the staging environment. If the output listed above does not match what you see on the screen, you might need to wait for a few moments and re-run the previous

command.

The good news is that the database is indeed running. The bad news is that the application is still not operational. In my case, it has already restarted five times, and 0 containers are available.

## Describe pods #

Given that the problem is this time probably not related to the database, the logical course of action is to describe the Pod and see whether we can get a clue about the issue from the events.

```
kubectl --namespace jx-staging \  
describe pod \  
-l app=jx-go-demo-6
```

The output, limited to the message of the events, is as follows.

```
...  
Events:  
... Message  
... -----  
... Successfully assigned jx-go-demo-6-fdd8f6644-xx68f to gke-jx-rocks-default-pool-119fec1e-v7p7  
... MountVolume.Setup succeeded for volume "default-token-cxn5p"  
... Readiness probe failed: Get http://10.28.2.17:8080/: dial tcp 10.28.2.17:8080: getsockopt: con  
... Back-off restarting failed container  
... Container image "10.31.245.243:5000/vfarcic/go-demo-6:0.0.81" already present on machine  
... Created container  
... Started container  
... Liveness probe failed: HTTP probe failed with statuscode: 404  
... Readiness probe failed: HTTP probe failed with statuscode: 404
```

This time, liveness and readiness probes are failing. Either the application inside that Pod is not responding to the probes, or there's some other problem we have yet to discover.

Now, unless you went through the code, you cannot know that the application does not respond to requests on the root path. If we take a look at the Pod definition, we'll see that `probePath` is set to `/`. Jenkins X could not know which path could be used for the probe of our application. So, it set it to the only sensible default it could, and that's `/`.

## Modifying the `probePath` entry #

We'll have to modify the `probePath` entry. There is already a variable that allows us

to do that instead of fiddling with the Deployment template.

```
cat charts/go-demo-6/values.yaml
```

If you go through the values, you'll notice that one of them is `probePath` and that it is set to `/`.

Please edit the `charts/go-demo-6/values.yaml` file by changing `probePath: /` entry to `probePath: /demo/hello?health=true`. Feel free to use your favorite editor for that and make sure to save the changes once you're done. Next, we'll push the changes to GitHub.

```
git add .  
  
git commit \  
  --message "Added dependencies"  
  
git push
```

Now we have another round of waiting until the activity of the new build is finished.

```
jx get activity \  
  --filter go-demo-6 \  
  --watch
```

Once the new build is finished, we can stop watching the activity by pressing `ctrl+c`. You'll know it is done when you see the `Promoted` entry in the `Succeeded` status or simply when there are no `Pending` and `Running` steps.

## Final checks for the application #

*What do you think?* Is our application finally up-and-running? Let's check it out.

```
kubectl --namespace jx-staging get pods
```

The output is as follows:

NAME		READY	STATUS	RESTARTS	AGE
jx-go-demo-6-...	1/1	Running	0	39s	
jx-go-demo-6-db-arbiter-0	1/1	Running	0	11m	
jx-go-demo-6-db-primary-0	1/1	Running	0	11m	
jx-go-demo-6-db-secondary-0	1/1	Running	0	11m	

To be on the safe side, we'll send a request to the application. If we are greeted back, we'll know that it's working as expected.

```
curl "$STAGING_ADDR/demo/hello"
```



The output shows `hello, world`, thus confirming that the application is up-and-running and that we can reach it.

Before we proceed, we'll go out of the `go-demo-6` directory.

```
cd ..
```



---

In the next lesson, we will discuss the reasoning behind all the work we just did.