Parallel Processing using Expansions

This task demonstrates running multiple Jobs based on a common template. You can use this approach to process batches of work in parallel.

For this example there are only three items: *apple*, *banana*, and *cherry*. The sample Jobs process each item by printing a string then pausing.

See using Jobs in real workloads to learn about how this pattern fits more realistic use cases.

Before you begin

You should be familiar with the basic, non-parallel, use of Job.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

For basic templating you need the command-line utility `sed`.

To follow the advanced templating example, you need a working installation of Python, and the Jinja2 template library for Python.

Once you have Python set up, you can install Jinja2 by running:

```
pip install --user jinja2
```

Create Jobs based on a template

First, download the following template of a Job to a file called `job-tmpl.yaml`. Here's what you'll download:

application/job/job-tmpl.yaml  Copy application/job/job-tmpl.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: process-item-$ITEM  labels:    jobgroup: jobexamplespec:  template:    metadata:      name: jobexample
```

```
# Use curl to download job-tmpl.yaml
curl -L -s -O https://k8s.io/examples/application/job/job-tmpl.yaml
```

The file you downloaded is not yet a valid Kubernetes manifest. Instead that template is a YAML representation of a Job object with some placeholders that need to be filled in before it can be used. The `$ITEM` syntax is not meaningful to Kubernetes.

Create manifests from the template

The following shell snippet uses `sed` to replace the string `$ITEM` with the loop variable, writing into a temporary directory named `jobs`. Run this now:

```
# Expand the template into multiple files, one for each item to be processed.
mkdir ./jobs
for i in apple banana cherry
do
  cat job-tmpl.yaml | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml
done
```

Check if it worked:

```
ls jobs/
```

The output is similar to this:

```
job-apple.yaml
job-banana.yaml
job-cherry.yaml
```

You could use any type of template language (for example: Jinja2; ERB), or write a program to generate the Job manifests.

Create Jobs from the manifests

Next, create all the Jobs with one kubectl command:

```
kubectl create -f ./jobs
```

The output is similar to this:

```
job.batch/process-item-apple created
job.batch/process-item-banana created
job.batch/process-item-cherry created
```

Now, check on the jobs:

```
kubectl get jobs -l jobgroup=jobexample
```

The output is similar to this:

```
NAME                   COMPLETIONS   DURATION   AGE
process-item-apple     1/1           14s        22s
process-item-banana    1/1           12s        21s
process-item-cherry    1/1           12s        20s
```

Using the `-l` option to kubectl selects only the Jobs that are part of this group of jobs (there might be other unrelated jobs in the system).

You can check on the Pods as well using the same label selector:

```
kubectl get pods -l jobgroup=jobexample
```

The output is similar to:

```
NAME                           READY     STATUS      RESTARTS   AGE
process-item-apple-kixwv       0/1       Completed   0          4m
process-item-banana-wrsf7      0/1       Completed   0          4m
process-item-cherry-dnfu9      0/1       Completed   0          4m
```

We can use this single command to check on the output of all jobs at once:

```
kubectl logs -f -l jobgroup=jobexample
```

The output should be:

```
Processing item apple
Processing item banana
Processing item cherry
```

Clean up

```
# Remove the Jobs you created
# Your cluster automatically cleans up their Pods
kubectl delete job -l jobgroup=jobexample
```

Use advanced template parameters

In the first example, each instance of the template had one parameter, and that parameter was also used in the Job's name. However, names are restricted to contain only certain characters.

This slightly more complex example uses the Jinja template language to generate manifests and then objects from those manifests, with a multiple parameters for each Job.

For this part of the task, you are going to use a one-line Python script to convert the template to a set of manifests.

First, copy and paste the following template of a Job object, into a file called `job.yaml.jinja2`:

```
{% set params = [{ "name": "apple", "url": "http://dbpedia.org/resource/Apple", },
                 { "name": "banana", "url": "http://dbpedia.org/resource/Banana", },
                 { "name": "cherry", "url": "http://dbpedia.org/resource/Cherry" }]
%}
{% for p in params %}
{% set name = p["name"] %}
{% set url = p["url"] %}
---
apiVersion: batch/v1
kind: Job
metadata:
  name: jobexample-{{ name }}
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
      - name: c
        image: busybox:1.28
        command: ["sh", "-c", "echo Processing URL {{ url }} && sleep 5"]
      restartPolicy: Never
{% endfor %}
```

The above template defines two parameters for each Job object using a list of python dicts (lines 1-4). A `for` loop emits one Job manifest for each set of parameters (remaining lines).

This example relies on a feature of YAML. One YAML file can contain multiple documents (Kubernetes manifests, in this case), separated by `---` on a line by itself. You can pipe the output directly to `kubectl` to create the Jobs.

Next, use this one-line Python program to expand the template:

```
alias render_template='python -c "from jinja2 import Template; import sys; print(Template(sys.stdin.read()).render());"'
```

Use `render_template` to convert the parameters and template into a single YAML file containing Kubernetes manifests:

```
# This requires the alias you defined earlier
cat job.yaml.jinja2 | render_template > jobs.yaml
```

You can view `jobs.yaml` to verify that the `render_template` script worked correctly.

Once you are happy that `render_template` is working how you intend, you can pipe its output into `kubectl`:

```
cat job.yaml.jinja2 | render_template | kubectl apply -f -
```

Kubernetes accepts and runs the Jobs you created.

Clean up

```
# Remove the Jobs you created
# Your cluster automatically cleans up their Pods
kubectl delete job -l jobgroup=jobexample
```

Using Jobs in real workloads

In a real use case, each Job performs some substantial computation, such as rendering a frame of a movie, or processing a range of rows in a database. If you were rendering a movie you would set `$ITEM` to the frame number. If you were processing rows from a database table, you would set `$ITEM` to represent the range of database rows to process.

In the task, you ran a command to collect the output from Pods by fetching their logs. In a real use case, each Pod for a Job writes its output to durable storage before completing. You can use a PersistentVolume for each Job, or an external storage service. For example, if you are rendering frames for a movie, use HTTP to `PUT` the rendered frame data to a URL, using a different URL for each frame.

Labels on Jobs and Pods

After you create a Job, Kubernetes automatically adds additional labels that distinguish one Job's pods from another Job's pods.

In this example, each Job and its Pod template have a label: `jobgroup=jobexample`.

Kubernetes itself pays no attention to labels named `jobgroup`. Setting a label for all the Jobs you create from a template makes it convenient to operate on all those Jobs at once. In the first example you used a template to create several Jobs. The template ensures that each Pod also gets the same label, so you can check on all Pods for these templated Jobs with a single command.

Note:

The label key `jobgroup` is not special or reserved. You can pick your own labelling scheme. There are recommended labels that you can use if you wish.

Alternatives

If you plan to create a large number of Job objects, you may find that:

- Even using labels, managing so many Jobs is cumbersome.
- If you create many Jobs in a batch, you might place high load on the Kubernetes control plane. Alternatively, the Kubernetes API server could rate limit you, temporarily rejecting your requests with a 429 status.
- You are limited by a resource quota on Jobs: the API server permanently rejects some of your requests when you create a great deal of work in one batch.

There are other job patterns that you can use to process large amounts of work without creating very many Job objects.

You could also consider writing your own controller to manage Job objects automatically.

Indexed Job for Parallel Processing with Static Work Assignment

FEATURE STATE: `Kubernetes v1.24 [stable]`

In this example, you will run a Kubernetes Job that uses multiple parallel worker processes. Each worker is a different container running in its own Pod. The Pods have an *index number* that the control plane sets automatically, which allows each Pod to identify which part of the overall task to work on.

The pod index is available in the annotation `batch.kubernetes.io/job-completion-index` as a string representing its decimal value. In order for the containerized task process to obtain this index, you can publish the value of the annotation using the downward API mechanism. For convenience, the control plane automatically sets the downward API to expose the index in the `JOB_COMPLETION_INDEX` environment variable.

Here is an overview of the steps in this example:

1. **Define a Job manifest using indexed completion**. The downward API allows you to pass the pod index annotation as an environment variable or file to the container.
2. **Start an `Indexed` Job based on that manifest**.

Before you begin

You should already be familiar with the basic, non-parallel, use of Job.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.21.

To check the version, enter `kubectl version`.

Choose an approach

To access the work item from the worker program, you have a few options:

1. Read the `JOB_COMPLETION_INDEX` environment variable. The Job controller automatically links this variable to the annotation containing the completion index.
2. Read a file that contains the completion index.
3. Assuming that you can't modify the program, you can wrap it with a script that reads the index using any of the methods above and converts it into something that the program can use as input.

For this example, imagine that you chose option 3 and you want to run the rev utility. This program accepts a file as an argument and prints its content reversed.

```
rev data.txt
```

You'll use the `rev` tool from the `busybox` container image.

As this is only an example, each Pod only does a tiny piece of work (reversing a short string). In a real workload you might, for example, create a Job that represents the task of producing 60 seconds of video based on scene data. Each work item in the video rendering Job would be to render a particular frame of that video clip. Indexed completion would mean that each Pod in the Job knows which frame to render and publish, by counting frames from the start of the clip.

Define an Indexed Job

Here is a sample Job manifest that uses `Indexed` completion mode:

application/job/indexed-job.yaml Copy application/job/indexed-job.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: 'indexed-job'spec:  completions: 5  parallelism: 3  completionMode: Indexed  template:    spec:      res
```

In the example above, you use the builtin `JOB_COMPLETION_INDEX` environment variable set by the Job controller for all containers. An init container maps the index to a static value and writes it to a file that is shared with the container running the worker through an emptyDir volume. Optionally, you can define your own environment variable through the downward API to publish the index to containers. You can also choose to load a list of values from a ConfigMap as an environment variable or file.

Alternatively, you can directly use the downward API to pass the annotation value as a volume file, like shown in the following example:

application/job/indexed-job-vol.yaml Copy application/job/indexed-job-vol.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: 'indexed-job'spec:  completions: 5  parallelism: 3  completionMode: Indexed  template:    spec:      res
```

Running the Job

Now run the Job:

```
# This uses the first approach (relying on $JOB_COMPLETION_INDEX)
kubectl apply -f https://kubernetes.io/examples/application/job/indexed-job.yaml
```

When you create this Job, the control plane creates a series of Pods, one for each index you specified. The value of `.spec.parallelism` determines how many can run at once whereas `.spec.completions` determines how many Pods the Job creates in total.

Because `.spec.parallelism` is less than `.spec.completions`, the control plane waits for some of the first Pods to complete before starting more of them.

You can wait for the Job to succeed, with a timeout:

```
# The check for condition name is case insensitive
kubectl wait --for=condition=complete --timeout=300s job/indexed-job
```

Now, describe the Job and check that it was successful.

```
kubectl describe jobs/indexed-job
```

The output is similar to:

```
Name:              indexed-job
Namespace:         default
Selector:          controller-uid=bf865e04-0b67-483b-9a90-74cfc4c3e756
Labels:            controller-uid=bf865e04-0b67-483b-9a90-74cfc4c3e756
                   job-name=indexed-job
Annotations:       <none>
Parallelism:       3
Completions:       5
Start Time:        Thu, 11 Mar 2021 15:47:34 +0000
Pods Statuses:     2 Running / 3 Succeeded / 0 Failed
```

```
Completed Indexes: 0-2
Pod Template:
  Labels:  controller-uid=bf865e04-0b67-483b-9a90-74cfc4c3e756
           job-name=indexed-job
  Init Containers:
   input:
    Image:      docker.io/library/bash
    Port:       <none>
    Host Port:  <none>
    Command:
      bash
      -c
      items=(foo bar baz qux xyz)
      echo ${items[$JOB_COMPLETION_INDEX]} > /input/data.txt

    Environment:  <none>
    Mounts:
      /input from input (rw)
  Containers:
   worker:
    Image:      docker.io/library/busybox
    Port:       <none>
    Host Port:  <none>
    Command:
      rev
      /input/data.txt
    Environment:  <none>
    Mounts:
      /input from input (rw)
  Volumes:
   input:
    Type:       EmptyDir (a temporary directory that shares a pod's lifetime)
    Medium:
    SizeLimit:  <unset>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  4s    job-controller  Created pod: indexed-job-njkjj
  Normal  SuccessfulCreate  4s    job-controller  Created pod: indexed-job-9kd4h
  Normal  SuccessfulCreate  4s    job-controller  Created pod: indexed-job-qjwsz
  Normal  SuccessfulCreate  1s    job-controller  Created pod: indexed-job-fdhq5
  Normal  SuccessfulCreate  1s    job-controller  Created pod: indexed-job-ncslj
```

In this example, you run the Job with custom values for each index. You can inspect the output of one of the pods:

```
kubectl logs indexed-job-fdhq5 # Change this to match the name of a Pod from that Job
```

The output is similar to:

```
xuq
```

Running Automated Tasks with a CronJob

This page shows how to run automated tasks using Kubernetes CronJob object.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:
  - iximiuz Labs
  - Killercoda
  - KodeKloud
  - Play with Kubernetes

Creating a CronJob

Cron jobs require a config file. Here is a manifest for a CronJob that runs a simple demonstration task every minute:

application/job/cronjob.yaml  Copy application/job/cronjob.yaml to clipboard

```
apiVersion: batch/v1
kind: CronJobmetadata:  name: hellospec:  schedule: "* * * * *"  jobTemplate:    spec:      template:        spec:          contai
```

Run the example CronJob by using this command:

```
kubectl create -f https://k8s.io/examples/application/job/cronjob.yaml
```

The output is similar to this:

```
cronjob.batch/hello created
```

After creating the cron job, get its status using this command:

```
kubectl get cronjob hello
```

The output is similar to this:

```
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        <none>          10s
```

As you can see from the results of the command, the cron job has not scheduled or run any jobs yet. Watch for the job to be created in around one minute:

```
kubectl get jobs --watch
```

The output is similar to this:

```
NAME                COMPLETIONS   DURATION   AGE
hello-4111706356    0/1                      0s
hello-4111706356    0/1           0s         0s
hello-4111706356    1/1           5s         5s
```

Now you've seen one running job scheduled by the "hello" cron job. You can stop watching the job and view the cron job again to see that it scheduled the job:

```
kubectl get cronjob hello
```

The output is similar to this:

```
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        50s             75s
```

You should see that the cron job `hello` successfully scheduled a job at the time specified in `LAST SCHEDULE`. There are currently 0 active jobs, meaning that the job has completed or failed.

Now, find the pods that the last scheduled job created and view the standard output of one of the pods.

Note:

The job name is different from the pod name.

```
# Replace "hello-4111706356" with the job name in your system
pods=$(kubectl get pods --selector=job-name=hello-4111706356 --output=jsonpath={.items[*].metadata.name})
```

Show the pod log:

```
kubectl logs $pods
```

The output is similar to this:

```
Fri Feb 22 11:02:09 UTC 2019
Hello from the Kubernetes cluster
```

Deleting a CronJob

When you don't need a cron job any more, delete it with `kubectl delete cronjob <cronjob name>`:

```
kubectl delete cronjob hello
```

Deleting the cron job removes all the jobs and pods it created and stops it from creating additional jobs. You can read more about removing jobs in garbage collection.

Run Jobs

Run Jobs using parallel processing.

Running Automated Tasks with a CronJob

Coarse Parallel Processing Using a Work Queue

Fine Parallel Processing Using a Work Queue

Indexed Job for Parallel Processing with Static Work Assignment

Job with Pod-to-Pod Communication

Parallel Processing using Expansions

Handling retriable and non-retriable pod failures with Pod failure policy

---

Coarse Parallel Processing Using a Work Queue

In this example, you will run a Kubernetes Job with multiple parallel worker processes.

In this example, as each pod is created, it picks up one unit of work from a task queue, completes it, deletes it from the queue, and exits.

Here is an overview of the steps in this example:

1. **Start a message queue service.** In this example, you use RabbitMQ, but you could use another one. In practice you would set up a message queue service once and reuse it for many jobs.
2. **Create a queue, and fill it with messages.** Each message represents one task to be done. In this example, a message is an integer that we will do a lengthy computation on.
3. **Start a Job that works on tasks from the queue**. The Job starts several pods. Each pod takes one task from the message queue, processes it, and exits.

Before you begin

You should already be familiar with the basic, non-parallel, use of Job.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

You will need a container image registry where you can upload images to run in your cluster.

This task example also assumes that you have Docker installed locally.

Starting a message queue service

This example uses RabbitMQ, however, you can adapt the example to use another AMQP-type message service.

In practice you could set up a message queue service once in a cluster and reuse it for many jobs, as well as for long-running services.

Start RabbitMQ as follows:

```
# make a Service for the StatefulSet to use
kubectl create -f https://kubernetes.io/examples/application/job/rabbitmq/rabbitmq-service.yaml
```

```
service "rabbitmq-service" created
```

```
kubectl create -f https://kubernetes.io/examples/application/job/rabbitmq/rabbitmq-statefulset.yaml
```

```
statefulset "rabbitmq" created
```

Testing the message queue service

Now, we can experiment with accessing the message queue. We will create a temporary interactive pod, install some tools on it, and experiment with queues.

First create a temporary interactive Pod.

```
# Create a temporary interactive container
kubectl run -i --tty temp --image ubuntu:22.04
```

```
Waiting for pod default/temp-loe07 to be running, status is Pending, pod ready: false
... [ previous line repeats several times .. hit return when it stops ] ...
```

Note that your pod name and command prompt will be different.

Next install the `amqp-tools` so you can work with message queues. The next commands show what you need to run inside the interactive shell in that Pod:

```
apt-get update && apt-get install -y curl ca-certificates amqp-tools python3 dnsutils
```

Later, you will make a container image that includes these packages.

Next, you will check that you can discover the Service for RabbitMQ:

```
# Run these commands inside the Pod
# Note the rabbitmq-service has a DNS name, provided by Kubernetes:
nslookup rabbitmq-service
```

```
Server:        10.0.0.10
Address:    10.0.0.10#53

Name:    rabbitmq-service.default.svc.cluster.local
Address: 10.0.147.152
```

(the IP addresses will vary)

If the kube-dns addon is not set up correctly, the previous step may not work for you. You can also find the IP address for that Service in an environment variable:

```
# run this check inside the Pod
env | grep RABBITMQ_SERVICE | grep HOST
```

```
RABBITMQ_SERVICE_SERVICE_HOST=10.0.147.152
```

(the IP address will vary)

Next you will verify that you can create a queue, and publish and consume messages.

```
# Run these commands inside the Pod
# In the next line, rabbitmq-service is the hostname where the rabbitmq-service
# can be reached.  5672 is the standard port for rabbitmq.
export BROKER_URL=amqp://guest:guest@rabbitmq-service:5672
# If you could not resolve "rabbitmq-service" in the previous step,
# then use this command instead:
BROKER_URL=amqp://guest:guest@$RABBITMQ_SERVICE_SERVICE_HOST:5672

# Now create a queue:

/usr/bin/amqp-declare-queue --url=$BROKER_URL -q foo -d

foo
```

Publish one message to the queue:

```
/usr/bin/amqp-publish --url=$BROKER_URL -r foo -p -b Hello

# And get it back.

/usr/bin/amqp-consume --url=$BROKER_URL -q foo -c 1 cat && echo 1>&2

Hello
```

In the last command, the `amqp-consume` tool took one message (`-c 1`) from the queue, and passes that message to the standard input of an arbitrary command. In this case, the program `cat` prints out the characters read from standard input, and the echo adds a carriage return so the example is readable.

Fill the queue with tasks

Now, fill the queue with some simulated tasks. In this example, the tasks are strings to be printed.

In a practice, the content of the messages might be:

- names of files to that need to be processed
- extra flags to the program
- ranges of keys in a database table
- configuration parameters to a simulation
- frame numbers of a scene to be rendered

If there is large data that is needed in a read-only mode by all pods of the Job, you typically put that in a shared file system like NFS and mount that readonly on all the pods, or write the program in the pod so that it can natively read data from a cluster file system (for example: HDFS).

For this example, you will create the queue and fill it using the AMQP command line tools. In practice, you might write a program to fill the queue using an AMQP client library.

```
# Run this on your computer, not in the Pod
/usr/bin/amqp-declare-queue --url=$BROKER_URL -q job1  -d

job1
```

Add items to the queue:

```
for f in apple banana cherry date fig grape lemon melon
do
  /usr/bin/amqp-publish --url=$BROKER_URL -r job1 -p -b $f
done
```

You added 8 messages to the queue.

Create a container image

Now you are ready to create an image that you will run as a Job.

The job will use the `amqp-consume` utility to read the message from the queue and run the actual work. Here is a very simple example program:

application/job/rabbitmq/worker.py  Copy application/job/rabbitmq/worker.py to clipboard

```
#!/usr/bin/env python

# Just prints standard out and sleeps for 10 seconds.
import sys
import time
print("Processing " + sys.stdin.readlines()[0])
time.sleep(10)
```

Give the script execution permission:

```
chmod +x worker.py
```

Now, build an image. Make a temporary directory, change to it, download the Dockerfile, and worker.py. In either case, build the image with this command:

```
docker build -t job-wq-1 .
```

For the Docker Hub, tag your app image with your username and push to the Hub with the below commands. Replace `<username>` with your Hub username.

```
docker tag job-wq-1 <username>/job-wq-1
docker push <username>/job-wq-1
```

If you are using an alternative container image registry, tag the image and push it there instead.

Defining a Job

Here is a manifest for a Job. You'll need to make a copy of the Job manifest (call it `./job.yaml`), and edit the name of the container image to match the name you used.

application/job/rabbitmq/job.yaml  Copy application/job/rabbitmq/job.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: job-wq-1spec:  completions: 8  parallelism: 2  template:    metadata:      name: job-wq-1    spec:      
```

In this example, each pod works on one item from the queue and then exits. So, the completion count of the Job corresponds to the number of work items done. That is why the example manifest has `.spec.completions` set to `8`.

Running the Job

Now, run the Job:

```
# this assumes you downloaded and then edited the manifest already
kubectl apply -f ./job.yaml
```

You can wait for the Job to succeed, with a timeout:

```
# The check for condition name is case insensitive
kubectl wait --for=condition=complete --timeout=300s job/job-wq-1
```

Next, check on the Job:

```
kubectl describe jobs/job-wq-1
```

```
Name:             job-wq-1
Namespace:        default
Selector:         controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
Labels:           controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
                  job-name=job-wq-1
Annotations:      <none>
Parallelism:      2
Completions:      8
Start Time:       Wed, 06 Sep 2022 16:42:02 +0000
Pods Statuses:    0 Running / 8 Succeeded / 0 Failed
Pod Template:
  Labels:       controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
                job-name=job-wq-1
  Containers:
   c:
    Image:      container-registry.example/causal-jigsaw-637/job-wq-1
    Port:
    Environment:
      BROKER_URL:       amqp://guest:guest@rabbitmq-service:5672
      QUEUE:            job1
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen  LastSeen   Count    From    SubobjectPath    Type      Reason              Message
  ---------  --------   -----    ----    -------------    -------   ------              -------
  27s        27s        1        {job }                   Normal    SuccessfulCreate    Created pod: job-wq-1-hcobb
  27s        27s        1        {job }                   Normal    SuccessfulCreate    Created pod: job-wq-1-weytj
  27s        27s        1        {job }                   Normal    SuccessfulCreate    Created pod: job-wq-1-qaam5
  27s        27s        1        {job }                   Normal    SuccessfulCreate    Created pod: job-wq-1-b67sr
  26s        26s        1        {job }                   Normal    SuccessfulCreate    Created pod: job-wq-1-xe5hj
  15s        15s        1        {job }                   Normal    SuccessfulCreate    Created pod: job-wq-1-w2zqe
  14s        14s        1        {job }                   Normal    SuccessfulCreate    Created pod: job-wq-1-d6ppa
  14s        14s        1        {job }                   Normal    SuccessfulCreate    Created pod: job-wq-1-p17e0
```

All the pods for that Job succeeded! You're done.

Alternatives

This approach has the advantage that you do not need to modify your "worker" program to be aware that there is a work queue. You can include the worker program unmodified in your container image.

Using this approach does require that you run a message queue service. If running a queue service is inconvenient, you may want to consider one of the other job patterns.

This approach creates a pod for every work item. If your work items only take a few seconds, though, creating a Pod for every work item may add a lot of overhead. Consider another design, such as in the fine parallel work queue example, that executes multiple work items per Pod.

In this example, you used the `amqp-consume` utility to read the message from the queue and run the actual program. This has the advantage that you do not need to modify your program to be aware of the queue. The fine parallel work queue example shows how to communicate with the work queue using a client library.

Caveats

If the number of completions is set to less than the number of items in the queue, then not all items will be processed.

If the number of completions is set to more than the number of items in the queue, then the Job will not appear to be completed, even though all items in the queue have been processed. It will start additional pods which will block waiting for a message. You would need to make your own mechanism to spot when there is work to do and measure the size of the queue, setting the number of completions to match.

There is an unlikely race with this pattern. If the container is killed in between the time that the message is acknowledged by the `amqp-consume` command and the time that the container exits with success, or if the node crashes before the kubelet is able to post the success of the pod back to the API server, then the Job will not appear to be complete, even though all items in the queue have been processed.

Fine Parallel Processing Using a Work Queue

In this example, you will run a Kubernetes Job that runs multiple parallel tasks as worker processes, each running as a separate Pod.

In this example, as each pod is created, it picks up one unit of work from a task queue, processes it, and repeats until the end of the queue is reached.

Here is an overview of the steps in this example:

1. **Start a storage service to hold the work queue.** In this example, you will use Redis to store work items. In the previous example, you used RabbitMQ. In this example, you will use Redis and a custom work-queue client library; this is because AMQP does not provide a good way for clients to detect when a finite-length work queue is empty. In practice you would set up a store such as Redis once and reuse it for the work queues of many jobs, and other things.
2. **Create a queue, and fill it with messages.** Each message represents one task to be done. In this example, a message is an integer that we will do a lengthy computation on.
3. **Start a Job that works on tasks from the queue**. The Job starts several pods. Each pod takes one task from the message queue, processes it, and repeats until the end of the queue is reached.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

You will need a container image registry where you can upload images to run in your cluster. The example uses Docker Hub, but you could adapt it to a different container image registry.

This task example also assumes that you have Docker installed locally. You use Docker to build container images.

Be familiar with the basic, non-parallel, use of Job.

Starting Redis

For this example, for simplicity, you will start a single instance of Redis. See the Redis Example for an example of deploying Redis scalably and redundantly.

You could also download the following files directly:

- redis-pod.yaml
- redis-service.yaml
- Dockerfile
- job.yaml
- rediswq.py
- worker.py

To start a single instance of Redis, you need to create the redis pod and redis service:

```
kubectl apply -f https://k8s.io/examples/application/job/redis/redis-pod.yaml
kubectl apply -f https://k8s.io/examples/application/job/redis/redis-service.yaml
```

Filling the queue with tasks

Now let's fill the queue with some "tasks". In this example, the tasks are strings to be printed.

Start a temporary interactive pod for running the Redis CLI.

```
kubectl run -i --tty temp --image redis --command "/bin/sh"
```

```
Waiting for pod default/redis2-c7h78 to be running, status is Pending, pod ready: false
Hit enter for command prompt
```

Now hit enter, start the Redis CLI, and create a list with some work items in it.

```
redis-cli -h redis
```

```
redis:6379> rpush job2 "apple"
(integer) 1
redis:6379> rpush job2 "banana"
```

```
(integer) 2
redis:6379> rpush job2 "cherry"
(integer) 3
redis:6379> rpush job2 "date"
(integer) 4
redis:6379> rpush job2 "fig"
(integer) 5
redis:6379> rpush job2 "grape"
(integer) 6
redis:6379> rpush job2 "lemon"
(integer) 7
redis:6379> rpush job2 "melon"
(integer) 8
redis:6379> rpush job2 "orange"
(integer) 9
redis:6379> lrange job2 0 -1
1) "apple"
2) "banana"
3) "cherry"
4) "date"
5) "fig"
6) "grape"
7) "lemon"
8) "melon"
9) "orange"
```

So, the list with key `job2` will be the work queue.

Note: if you do not have Kube DNS setup correctly, you may need to change the first step of the above block to `redis-cli -h $REDIS_SERVICE_HOST`.

Create a container image

Now you are ready to create an image that will process the work in that queue.

You're going to use a Python worker program with a Redis client to read the messages from the message queue.

A simple Redis work queue client library is provided, called `rediswq.py` (Download).

The "worker" program in each Pod of the Job uses the work queue client library to get work. Here it is:

application/job/redis/worker.py Copy application/job/redis/worker.py to clipboard

```
#!/usr/bin/env python

import time
import rediswq

host="redis"
# Uncomment next two lines if you do not have Kube-DNS working.
# import os
# host = os.getenv("REDIS_SERVICE_HOST")

q = rediswq.RedisWQ(name="job2", host=host)
print("Worker with sessionID: " +  q.sessionID())
print("Initial queue state: empty=" + str(q.empty()))
while not q.empty():
  item = q.lease(lease_secs=10, block=True, timeout=2)
  if item is not None:
    itemstr = item.decode("utf-8")
    print("Working on " + itemstr)
    time.sleep(10) # Put your actual work here instead of sleep.
    q.complete(item)
  else:
    print("Waiting for work")
print("Queue empty, exiting")
```

You could also download worker.py, rediswq.py, and Dockerfile files, then build the container image. Here's an example using Docker to do the image build:

```
docker build -t job-wq-2 .
```

Push the image

For the Docker Hub, tag your app image with your username and push to the Hub with the below commands. Replace `<username>` with your Hub username.

```
docker tag job-wq-2 <username>/job-wq-2
docker push <username>/job-wq-2
```

You need to push to a public repository or configure your cluster to be able to access your private repository.

Defining a Job

Here is a manifest for the Job you will create:

application/job/redis/job.yaml Copy application/job/redis/job.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: job-wq-2spec:  parallelism: 2  template:    metadata:      name: job-wq-2    spec:      containers:
```

Note:

Be sure to edit the manifest to change `gcr.io/myproject` to your own path.

In this example, each pod works on several items from the queue and then exits when there are no more items. Since the workers themselves detect when the workqueue is empty, and the Job controller does not know about the workqueue, it relies on the workers to signal when they are done working. The workers signal that the queue is empty by exiting with success. So, as soon as **any** worker exits with success, the controller knows the work is done, and that the Pods will exit soon. So, you need to leave the completion count of the Job unset. The job controller will wait for the other pods to complete too.

Running the Job

So, now run the Job:

```
# this assumes you downloaded and then edited the manifest already
kubectl apply -f ./job.yaml
```

Now wait a bit, then check on the Job:

```
kubectl describe jobs/job-wq-2
```

```
Name:             job-wq-2
Namespace:        default
Selector:         controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
Labels:           controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
                  job-name=job-wq-2
Annotations:      <none>
Parallelism:      2
Completions:      <unset>
Start Time:       Mon, 11 Jan 2022 17:07:59 +0000
Pods Statuses:    1 Running / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
                job-name=job-wq-2
  Containers:
   c:
    Image:              container-registry.example/exampleproject/job-wq-2
    Port:
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen    LastSeen    Count    From            SubobjectPath    Type        Reason            Message
  ---------    --------    -----    ----            -------------    --------    ------            -------
  33s          33s         1        {job-controller }                Normal      SuccessfulCreate  Created pod: job-wq-2-lglf8
```

You can wait for the Job to succeed, with a timeout:

```
# The check for condition name is case insensitive
kubectl wait --for=condition=complete --timeout=300s job/job-wq-2
```

```
kubectl logs pods/job-wq-2-7r7b2
```

```
Worker with sessionID: bbd72d0a-9e5c-4dd6-abf6-416cc267991f
Initial queue state: empty=False
Working on banana
Working on date
Working on lemon
```

As you can see, one of the pods for this Job worked on several work units.

Alternatives

If running a queue service or modifying your containers to use a work queue is inconvenient, you may want to consider one of the other job patterns.

If you have a continuous stream of background processing work to run, then consider running your background workers with a ReplicaSet instead, and consider running a background processing library such as https://github.com/resque/resque.

Handling retriable and non-retriable pod failures with Pod failure policy

FEATURE STATE: `Kubernetes v1.31 [stable]` (enabled by default: true)

This document shows you how to use the Pod failure policy, in combination with the default Pod backoff failure policy, to improve the control over the handling of container- or Pod-level failure within a Job.

The definition of Pod failure policy may help you to:

- better utilize the computational resources by avoiding unnecessary Pod retries.
- avoid Job failures due to Pod disruptions (such preemption, API-initiated eviction or taint-based eviction).

Before you begin

You should already be familiar with the basic use of Job.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.25.

To check the version, enter `kubectl version`.

Usage scenarios

Consider the following usage scenarios for Jobs that define a Pod failure policy :

- Avoiding unnecessary Pod retries
- Ignoring Pod disruptions
- Avoiding unnecessary Pod retries based on custom Pod Conditions
- Avoiding unnecessary Pod retries per index

Using Pod failure policy to avoid unnecessary Pod retries

With the following example, you can learn how to use Pod failure policy to avoid unnecessary Pod restarts when a Pod failure indicates a non-retriable software bug.

1. Examine the following manifest:

   /controllers/job-pod-failure-policy-failjob.yaml Copy /controllers/job-pod-failure-policy-failjob.yaml to clipboard

   ```
   apiVersion: batch/v1
   kind: Jobmetadata:  name: job-pod-failure-policy-failjobspec:  completions: 8  parallelism: 2  template:    spec:      restar
   ```

2. Apply the manifest:

   ```
   kubectl create -f https://k8s.io/examples/controllers/job-pod-failure-policy-failjob.yaml
   ```

3. After around 30 seconds the entire Job should be terminated. Inspect the status of the Job by running:

   ```
   kubectl get jobs -l job-name=job-pod-failure-policy-failjob -o yaml
   ```

   In the Job status, the following conditions display:

   - `FailureTarget` condition: has a `reason` field set to `PodFailurePolicy` and a `message` field with more information about the termination, like `Container main for pod default/job-pod-failure-policy-failjob-8ckj8 failed with exit code 42 matching FailJob rule at index 0`. The Job controller adds this condition as soon as the Job is considered a failure. For details, see Termination of Job Pods.
   - `Failed` condition: same `reason` and `message` as the `FailureTarget` condition. The Job controller adds this condition after all of the Job's Pods are terminated.

   For comparison, if the Pod failure policy was disabled it would take 6 retries of the Pod, taking at least 2 minutes.

Clean up

Delete the Job you created:

```
kubectl delete jobs/job-pod-failure-policy-failjob
```

The cluster automatically cleans up the Pods.

Using Pod failure policy to ignore Pod disruptions

With the following example, you can learn how to use Pod failure policy to ignore Pod disruptions from incrementing the Pod retry counter towards the `.spec.backoffLimit` limit.

Caution:

Timing is important for this example, so you may want to read the steps before execution. In order to trigger a Pod disruption it is important to drain the node while the Pod is running on it (within 90s since the Pod is scheduled).

1. Examine the following manifest:

   /controllers/job-pod-failure-policy-ignore.yaml Copy /controllers/job-pod-failure-policy-ignore.yaml to clipboard

   ```
   apiVersion: batch/v1
   kind: Jobmetadata:  name: job-pod-failure-policy-ignorespec:  completions: 4  parallelism: 2  template:    spec:      restart
   ```

2. Apply the manifest:

   ```
   kubectl create -f https://k8s.io/examples/controllers/job-pod-failure-policy-ignore.yaml
   ```

3. Run this command to check the `nodeName` the Pod is scheduled to:

```
nodeName=$(kubectl get pods -l job-name=job-pod-failure-policy-ignore -o jsonpath='{.items[0].spec.nodeName}')
```

4. Drain the node to evict the Pod before it completes (within 90s):

   ```
   kubectl drain nodes/$nodeName --ignore-daemonsets --grace-period=0
   ```

5. Inspect the `.status.failed` to check the counter for the Job is not incremented:

   ```
   kubectl get jobs -l job-name=job-pod-failure-policy-ignore -o yaml
   ```

6. Uncordon the node:

   ```
   kubectl uncordon nodes/$nodeName
   ```

The Job resumes and succeeds.

For comparison, if the Pod failure policy was disabled the Pod disruption would result in terminating the entire Job (as the `.spec.backoffLimit` is set to 0).

Cleaning up

Delete the Job you created:

```
kubectl delete jobs/job-pod-failure-policy-ignore
```

The cluster automatically cleans up the Pods.

Using Pod failure policy to avoid unnecessary Pod retries based on custom Pod Conditions

With the following example, you can learn how to use Pod failure policy to avoid unnecessary Pod restarts based on custom Pod Conditions.

Note:

The example below works since version 1.27 as it relies on transitioning of deleted pods, in the `Pending` phase, to a terminal phase (see: Pod Phase).

1. Examine the following manifest:

   /controllers/job-pod-failure-policy-config-issue.yaml Copy /controllers/job-pod-failure-policy-config-issue.yaml to clipboard

   ```
   apiVersion: batch/v1
   kind: Jobmetadata:  name: job-pod-failure-policy-config-issuespec:  completions: 8  parallelism: 2  template:    spec:      r
   ```

2. Apply the manifest:

   ```
   kubectl create -f https://k8s.io/examples/controllers/job-pod-failure-policy-config-issue.yaml
   ```

   Note that, the image is misconfigured, as it does not exist.

3. Inspect the status of the job's Pods by running:

   ```
   kubectl get pods -l job-name=job-pod-failure-policy-config-issue -o yaml
   ```

   You will see output similar to this:

   ```
   containerStatuses:
   - image: non-existing-repo/non-existing-image:example   ...   state:   waiting:      message: Back-off pulling image "non-exi
   ```

   Note that the pod remains in the `Pending` phase as it fails to pull the misconfigured image. This, in principle, could be a transient issue and the image could get pulled. However, in this case, the image does not exist so we indicate this fact by a custom condition.

4. Add the custom condition. First prepare the patch by running:

   ```
   cat <<EOF > patch.yaml
   status:
     conditions:
     - type: ConfigIssue
       status: "True"
       reason: "NonExistingImage"
       lastTransitionTime: "$(date -u +"%Y-%m-%dT%H:%M:%SZ")"
   EOF
   ```

   Second, select one of the pods created by the job by running:

   ```
   podName=$(kubectl get pods -l job-name=job-pod-failure-policy-config-issue -o jsonpath='{.items[0].metadata.name}')
   ```

   Then, apply the patch on one of the pods by running the following command:

   ```
   kubectl patch pod $podName --subresource=status --patch-file=patch.yaml
   ```

   If applied successfully, you will get a notification like this:

   ```
   pod/job-pod-failure-policy-config-issue-k6pvp patched
   ```

5. Delete the pod to transition it to `Failed` phase, by running the command:

   ```
   kubectl delete pods/$podName
   ```

6. Inspect the status of the Job by running:

```
kubectl get jobs -l job-name=job-pod-failure-policy-config-issue -o yaml
```

In the Job status, see a job `Failed` condition with the field `reason` equal `PodFailurePolicy`. Additionally, the `message` field contains a more detailed information about the Job termination, such as: `Pod default/job-pod-failure-policy-config-issue-k6pvp has condition ConfigIssue matching FailJob rule at index 0.`

Note:

In a production environment, the steps 3 and 4 should be automated by a user-provided controller.

Cleaning up

Delete the Job you created:

```
kubectl delete jobs/job-pod-failure-policy-config-issue
```

The cluster automatically cleans up the Pods.

Using Pod Failure Policy to avoid unnecessary Pod retries per index

To avoid unnecessary Pod restarts per index, you can use the *Pod failure policy* and *backoff limit per index* features. This section of the page shows how to use these features together.

1. Examine the following manifest:

   /controllers/job-backoff-limit-per-index-failindex.yaml Copy /controllers/job-backoff-limit-per-index-failindex.yaml to clipboard

   ```
   apiVersion: batch/v1
   kind: Jobmetadata:  name: job-backoff-limit-per-index-failindexspec:  completions: 4  parallelism: 2  completionMode: Indexed
   ```

2. Apply the manifest:

   ```
   kubectl create -f https://k8s.io/examples/controllers/job-backoff-limit-per-index-failindex.yaml
   ```

3. After around 15 seconds, inspect the status of the Pods for the Job. You can do that by running:

   ```
   kubectl get pods -l job-name=job-backoff-limit-per-index-failindex -o yaml
   ```

   You will see output similar to this:

   ```
   NAME                                          READY   STATUS      RESTARTS   AGE
   job-backoff-limit-per-index-failindex-0-4g4cm   0/1     Error       0          4s
   job-backoff-limit-per-index-failindex-0-fkdzq   0/1     Error       0          15s
   job-backoff-limit-per-index-failindex-1-2bgdj   0/1     Error       0          15s
   job-backoff-limit-per-index-failindex-2-vs6lt   0/1     Completed   0          11s
   job-backoff-limit-per-index-failindex-3-s7s47   0/1     Completed   0          6s
   ```

   Note that the output shows the following:

   - Two Pods have index 0, because of the backoff limit allowed for one retry of the index.
   - Only one Pod has index 1, because the exit code of the failed Pod matched the Pod failure policy with the `FailIndex` action.

4. Inspect the status of the Job by running:

   ```
   kubectl get jobs -l job-name=job-backoff-limit-per-index-failindex -o yaml
   ```

   In the Job status, see that the `failedIndexes` field shows "0,1", because both indexes failed. Because the index 1 was not retried the number of failed Pods, indicated by the status field "failed" equals 3.

Cleaning up

Delete the Job you created:

```
kubectl delete jobs/job-backoff-limit-per-index-failindex
```

The cluster automatically cleans up the Pods.

Alternatives

You could rely solely on the Pod backoff failure policy, by specifying the Job's `.spec.backoffLimit` field. However, in many situations it is problematic to find a balance between setting a low value for `.spec.backoffLimit` to avoid unnecessary Pod retries, yet high enough to make sure the Job would not be terminated by Pod disruptions.

Job with Pod-to-Pod Communication

In this example, you will run a Job in Indexed completion mode configured such that the pods created by the Job can communicate with each other using pod hostnames rather than pod IP addresses.

Pods within a Job might need to communicate among themselves. The user workload running in each pod could query the Kubernetes API server to learn the IPs of the other Pods, but it's much simpler to rely on Kubernetes' built-in DNS resolution.

Jobs in Indexed completion mode automatically set the pods' hostname to be in the format of `${jobName}-${completionIndex}`. You can use this format to deterministically build pod hostnames and enable pod communication *without* needing to create a client connection to the Kubernetes control plane to obtain pod hostnames/IPs via API requests.

This configuration is useful for use cases where pod networking is required but you don't want to depend on a network connection with the Kubernetes API server.

Before you begin

You should already be familiar with the basic use of Job.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.21.

To check the version, enter `kubectl version`.

Note:

If you are using minikube or a similar tool, you may need to take extra steps to ensure you have DNS.

Starting a Job with pod-to-pod communication

To enable pod-to-pod communication using pod hostnames in a Job, you must do the following:

1. Set up a headless Service with a valid label selector for the pods created by your Job. The headless service must be in the same namespace as the Job. One easy way to do this is to use the `job-name: <your-job-name>` selector, since the `job-name` label will be automatically added by Kubernetes. This configuration will trigger the DNS system to create records of the hostnames of the pods running your Job.

2. Configure the headless service as subdomain service for the Job pods by including the following value in your Job template spec:

   ```
   subdomain: <headless-svc-name>
   ```

Example

Below is a working example of a Job with pod-to-pod communication via pod hostnames enabled. The Job is completed only after all pods successfully ping each other using hostnames.

Note:

In the Bash script executed on each pod in the example below, the pod hostnames can be prefixed by the namespace as well if the pod needs to be reached from outside the namespace.

```
apiVersion: v1
kind: Servicemetadata:  name: headless-svcspec:  clusterIP: None # clusterIP must be None to create a headless service  selector:
```

After applying the example above, reach each other over the network using: `<pod-hostname>.<headless-service-name>`. You should see output similar to the following:

```
kubectl logs example-job-0-qws42
```

```
Failed to ping pod example-job-0.headless-svc, retrying in 1 second...
Successfully pinged pod: example-job-0.headless-svc
Successfully pinged pod: example-job-1.headless-svc
Successfully pinged pod: example-job-2.headless-svc
```

Note:

Keep in mind that the `<pod-hostname>.<headless-service-name>` name format used in this example would not work with DNS policy set to `None` or `Default`. Refer to Pod's DNS Policy.