

Run Jobs

Run Jobs using parallel processing.

- 1: [Running Automated Tasks with a CronJob](#)
- 2: [Parallel Processing using Expansions](#)
- 3: [Coarse Parallel Processing Using a Work Queue](#)
- 4: [Fine Parallel Processing Using a Work Queue](#)

1 - Running Automated Tasks with a CronJob

You can use a [CronJob](#) to run Jobs on a time-based schedule. These automated jobs run like [Cron](#) tasks on a Linux or UNIX system.

Cron jobs are useful for creating periodic and recurring tasks, like running backups or sending emails. Cron jobs can also schedule individual tasks for a specific time, such as if you want to schedule a job for a low activity period.

Cron jobs have limitations and idiosyncrasies. For example, in certain circumstances, a single cron job can create multiple jobs. Therefore, jobs should be idempotent.


For more limitations, see [CronJobs](#).

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:
 - [Katacoda](#)
 - [Play with Kubernetes](#)

Creating a Cron Job

Cron jobs require a config file. This example cron job config `.spec` file prints the current time and a hello message every minute:

[application/job/cronjob.yaml](#) 

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              imagePullPolicy: IfNotPresent
              command:
                - /bin/sh
```

```
- -c
- date; echo Hello from the Kubernetes cluster
restartPolicy: OnFailure
```

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	*/* * * * *	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	*/* * * * *	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 and pod name are different.

```
# Replace "hello-4111706356" with the job name in your system
pods=$(kubectl get pods --selector=job-name=hello-4111706356 --output=jsonpath={.items[0].metadata.name})
```

Show 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 Cron Job

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](#).

Writing a Cron Job Spec

As with all other Kubernetes configs, a cron job needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see [deploying applications](#), and [using kubectl to manage resources](#) documents.

A cron job config also needs a [.spec section](#).

Note: All modifications to a cron job, especially its `.spec`, are applied only to the following runs.

Schedule

The `.spec.schedule` is a required field of the `.spec`. It takes a [Cron](#) format string, such as `0 * * * *` or `@hourly`, as schedule time of its jobs to be created and executed.

The format also includes extended `vixie cron` step values. As explained in the [FreeBSD manual](#):

Step values can be used in conjunction with ranges. Following a range with `/<number>` specifies skips of the number's value through the range. For example, `0-23/2` can be used in the hours field to specify command execution every other hour (the alternative in the V7 standard is `0,2,4,6,8,10,12,14,16,18,20,22`). Steps are also permitted after an asterisk, so if you want to say "every two hours", just use `*/2`.

Note: A question mark (`?`) in the schedule has the same meaning as an asterisk `*`, that is, it stands for any of available value for a given field.

Job Template

The `.spec.jobTemplate` is the template for the job, and it is required. It has exactly the same schema as a [Job](#), except that it is nested and does not have an `apiVersion` or `kind`. For information about writing a job `.spec`, see [Writing a Job Spec](#).

Starting Deadline

The `.spec.startingDeadlineSeconds` field is optional. It stands for the deadline in seconds for starting the job if it misses its scheduled time for any reason. After the deadline, the cron job does not start the job. Jobs that do not meet their deadline in this way count as failed jobs. If this field is not specified, the jobs have no deadline.

The CronJob controller counts how many missed schedules happen for a cron job. If there are more than 100 missed schedules, the cron job is no longer scheduled. When `.spec.startingDeadlineSeconds` is not set, the CronJob controller counts missed schedules from `status.lastScheduleTime` until now.

For example, one cron job is supposed to run every minute, the `status.lastScheduleTime` of the cronjob is 5:00am, but now it's 7:00am. That means 120 schedules were missed, so the cron job is no longer scheduled.

If the `.spec.startingDeadlineSeconds` field is set (not null), the CronJob controller counts how many missed jobs occurred from the value of `.spec.startingDeadlineSeconds` until now.

For example, if it is set to `200`, it counts how many missed schedules occurred in the last 200 seconds. In that case, if there were more than 100 missed schedules in the last 200 seconds, the cron job is no longer scheduled.

Concurrency Policy

The `.spec.concurrencyPolicy` field is also optional. It specifies how to treat concurrent executions of a job that is created by this cron job. The spec may specify only one of the following concurrency policies:

- `Allow` (default): The cron job allows concurrently running jobs
- `Forbid`: The cron job does not allow concurrent runs; if it is time for a new job run and the previous job run hasn't finished yet, the cron job skips the new job run
- `Replace`: If it is time for a new job run and the previous job run hasn't finished yet, the cron job replaces the currently running job run with a new job run

Note that concurrency policy only applies to the jobs created by the same cron job. If there are multiple cron jobs, their respective jobs are always allowed to run concurrently.

Suspend

The `.spec.suspend` field is also optional. If it is set to `true`, all subsequent executions are suspended. This setting does not apply to already started executions. Defaults to false.

Caution: Executions that are suspended during their scheduled time count as missed jobs. When `.spec.suspend` changes from `true` to `false` on an existing cron job without a [starting deadline](#), the missed jobs are scheduled immediately.

Jobs History Limits

The `.spec.successfulJobsHistoryLimit` and `.spec.failedJobsHistoryLimit` fields are optional. These fields specify how many completed and failed jobs should be kept. By default, they are set to 3 and 1 respectively. Setting a limit to `0` corresponds to keeping none of the corresponding kind of jobs after they finish.

2 - 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. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Katacoda](#)
- [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](#) 

```
apiVersion: batch/v1
kind: Job
metadata:
  name: process-item-$ITEM
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
        - name: c
          image: busybox
          command: ["sh", "-c", "echo Processing item $ITEM && sleep 5"]
      restartPolicy: Never
```

```
# 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-dnfy9	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
          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(Te
```

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.

3 - Coarse Parallel Processing Using a Work Queue

In this example, we 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, we 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 repeats until the end of the queue is reached.

Before you begin

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. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Katacoda](#)
- [Play with Kubernetes](#)

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:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.
```

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

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.
```

```
replicationcontroller "rabbitmq-controller" created
```

We will only use the rabbitmq part from the [celery-rabbitmq example](#).

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:18.04
```

```
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 we can work with message queues.

```
# Install some tools
root@temp-loe07:/# apt-get update
.... [ lots of output ] ....
root@temp-loe07:/# apt-get install -y curl ca-certificates amqp-tools python dnsutils
.... [ lots of output ] ....
```

Later, we will make a docker image that includes these packages.

Next, we will check that we can discover the rabbitmq service:

```
# Note the rabbitmq-service has a DNS name, provided by Kubernetes:

root@temp-loe07:/# 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

# Your address will vary.
```

If Kube-DNS is not setup correctly, the previous step may not work for you. You can also find the service IP in an env var:

```
# env | grep RABBIT | grep HOST
RABBITMQ_SERVICE_SERVICE_HOST=10.0.147.152
# Your address will vary.
```

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

```
# In the next line, rabbitmq-service is the hostname where the rabbitmq-service
# can be reached. 5672 is the standard port for rabbitmq.

root@temp-loe07:/# 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:
# root@temp-loe07:/# BROKER_URL=amqp://guest:guest@$RABBITMQ_SERVICE_SERVICE_HOST:

# Now create a queue:

root@temp-loe07:/# /usr/bin/amqp-declare-queue --url=$BROKER_URL -q foo -d
foo

# Publish one message to it:
```

```
root@temp-loe07:/# /usr/bin/amqp-publish --url=$BROKER_URL -r foo -p -b Hello

# And get it back.

root@temp-loe07:/# /usr/bin/amqp-consume --url=$BROKER_URL -q foo -c 1 cat && echc
Hello
root@temp-loe07:/#
```

In the last command, the `amqp-consume` tool takes 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.

Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our 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

In practice, if there is large data that is needed in a read-only mode by all pods of the Job, you will typically put that in a shared file system like NFS and mount that readonly on all the pods, or the program in the pod will natively read data from a cluster file system like HDFS.

For our example, we 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.

```
/usr/bin/amqp-declare-queue --url=$BROKER_URL -q job1 -d
job1
```


```
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
```

So, we filled the queue with 8 messages.

Create an Image

Now we are ready to create an image that we will run as a job.

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

[application/job/rabbitmq/worker.py](#) 

```
#!/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. If you are working in the source tree, then change directory to `examples/job/work-queue-1`. Otherwise, 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 [Google Container Registry](#), tag your app image with your project ID, and push to GCR. Replace `<project>` with your project ID.

```
docker tag job-wq-1 gcr.io/<project>/job-wq-1
gcloud docker -- push gcr.io/<project>/job-wq-1
```

Defining a Job

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

[application/job/rabbitmq/job.yaml](#) 

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-1
spec:
  completions: 8
  parallelism: 2
  template:
    metadata:
      name: job-wq-1
    spec:
      containers:
      - name: c
        image: gcr.io/<project>/job-wq-1
        env:
        - name: BROKER_URL
          value: amqp://guest:guest@rabbitmq-service:5672
        - name: QUEUE
          value: job1
      restartPolicy: OnFailure
```

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. So we set, `.spec.completions: 8` for the example, since we put 8 items in the queue.

Running the Job

So, now run the Job:

```
kubectl apply -f ./job.yaml
```

Now wait a bit, then 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 2017 16:42:02 +0800
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:          gcr.io/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
  -----
  27s          27s         1       {job }             Normal          SuccessfulCreate
  27s          27s         1       {job }             Normal          SuccessfulCreate
  27s          27s         1       {job }             Normal          SuccessfulCreate
  27s          27s         1       {job }             Normal          SuccessfulCreate
  26s          26s         1       {job }             Normal          SuccessfulCreate
  15s          15s         1       {job }             Normal          SuccessfulCreate
  14s          14s         1       {job }             Normal          SuccessfulCreate
  14s          14s         1       {job }             Normal          SuccessfulCreate
```

All our pods succeeded. Yay.

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.

It 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 [example](#), that executes multiple work items per Pod.

In this example, we use the `amqp-consume` utility to read the message from the queue and run our actual program. This has the advantage that you do not need to modify your program to be aware of the queue. A [different 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.

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.

4 - Fine Parallel Processing Using a Work Queue

In this example, we will run a Kubernetes Job with multiple parallel worker processes in a given 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, we use Redis to store our work items. In the previous example, we used RabbitMQ. In this example, we use Redis and a custom work-queue client library 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. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Katacoda](#)
- [Play with Kubernetes](#)

Be familiar with the basic, non-parallel, use of [Job](#).

Starting Redis

For this example, for simplicity, we 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](#)
- [rediswg.py](#)
- [worker.py](#)

Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our 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 our 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 an Image

Now we are ready to create an image that we will run.

We will 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](#) 

```
#!/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="redis")
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](#), [rediswg.py](#), and [Dockerfile](#) files, then build the image:

```
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](#).

If you are using [Google Container Registry](#), tag your app image with your project ID, and push to GCR. Replace `<project>` with your project ID.

```

docker tag job-wq-2 gcr.io/<project>/job-wq-2
gcloud docker -- push gcr.io/<project>/job-wq-2

```

Defining a Job

Here is the job definition:

[application/job/redis/job.yaml](#) 

```

apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-2
spec:
  parallelism: 2
  template:
    metadata:
      name: job-wq-2
    spec:
      containers:
        - name: c
          image: gcr.io/myproject/job-wq-2
          restartPolicy: OnFailure

```

Be sure to edit the job template 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 the Pods will exit soon. So, we set the completion count of the Job to 1. The job controller will wait for the other pods to complete too.

Running the Job

So, now run the Job:

```
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 2016 17:07:59 -0800
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:           gcr.io/exampleproject/job-wq-2
      Port:
      Environment:    <none>
      Mounts:         <none>
      Volumes:        <none>
Events:
  FirstSeen    LastSeen    Count   From              SubobjectPath    Type    Reason
  -----
  33s          33s         1       {job-controller }              Normal    Succeeded

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 our pods 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>.

