Running in Production

We now have a 'real world' application running on our cluster. It is fault tolerant and multi-tiered.

But we are not yet ready for Production.

In this section we will cover

- Readiness and Liveness Probes
- Resource Requests & Limits
- ConfigMap / Secrets

Liveness and Readiness Probes

Kubernetes health checks are divided into liveness and readiness probes.

For running in production, Kubernetes needs a way to ensure pods are actually running and healthy.

Readiness Probe

Readiness probes allow you to specify checks to verify if a Pod is ready for use. There are three methods that can be used to determine readiness. HTTP, Exec or TCPSocket.

```
readinessProbe:
  httpGet:
    path: /readiness
    port: 8080
  initialDelaySeconds: 5  # Delay before probe is called
  timeoutSeconds: 1       # Probe must respond within this timeout
```

Response must be HTTP Status Code between 200 and 399

Liveness Probe

Once the application pod is up we need to confirm that it’s healthy and ready for serving traffic.

Liveness probes are for situations when an app has crashed or isn't responding anymore.

```
livenessProbe:
  httpGet:
    path: /healthcheck
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 10          # Will be called every 10 seconds
  timeoutSeconds: 1          # Probe must respond within 1 second
```

A simple example in Go

```
http.HandleFunc("/healthcheck", func(w http.ResponseWriter, r *http.R
    w.Write([]byte("ok"))
}
http.ListenAndServe(":8080", nil)
```

This needs to be added into the Kubernetes configuration

```
livenessProbe:
  httpGet:
    path: /healthcheck
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

Readiness Probes

Might check other pieces, such as a database connection

```
http.HandleFunc("/readiness", func(w http.ResponseWriter, r *http.Req
  ok := true
  errMsg = ""

  // Check database
  if db != nil {
    _, err := db.Query("SELECT 1;")
  }
  if err != nil {
    ok = false
    errMsg += "Database not ok.\n"
  }

  if ok {
    w.Write([]byte("OK"))
  } else {
```

This can be added to the configuration in the same way

```
readinessProbe:
  httpGet:
    path: /readiness
    port: 8080
  initialDelaySeconds: 20
  timeoutSeconds: 5
```

Advanced Liveness probe

```
livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
          - name: X-Custom-Header
            value: Awesome
      initialDelaySeconds: 15
      timeoutSeconds: 1
```

`httpHeaders` describes a custom header to be used in HTTP probes

Combining readiness and liveness probes help ensure only healthy containers are running within the cluster. With the liveness probe you may also monitor downstream dependencies.

Creating Pods with Liveness and Readiness Probes

Look in "resources" folder for the "Healthy Monolith" Pod configuration.

```
...
      livenessProbe:
        httpGet:
          path: /healthz
          port: 81
          scheme: HTTP
        initialDelaySeconds: 5
        periodSeconds: 15
        timeoutSeconds: 5
      readinessProbe:
        httpGet:
          path: /readiness
          port: 81
          scheme: HTTP
        initialDelaySeconds: 5
        timeoutSeconds: 1
```

Create the healthy-monolith pod using

```
$ kubectl create -f ./resources/healthy-monolith.yaml
```

Thanks to Kelsey Hightower for this application

View Pod details

Pods will not be marked ready until the readiness probe returns an HTTP 200 response. Use the `kubectl describe` to view details for the healthy-monolith Pod.

The healthy-monolith Pod logs each health check. Use the `kubectl logs` command to view them.

Experiment with Readiness Probes

Let's see how Kubernetes handles failed readiness probes.

The monolith container supports the ability to force failures of it's readiness and liveness probes (again thanks to Kelsey)

Use the `kubectl port-forward` command to forward a local port to the health port of the healthy-monolith Pod.

```
$ kubectl port-forward healthy-monolith 10081:81
```

You now have access to the /healthz and /readiness HTTP endpoints

Force the monolith container readiness probe to fail. Use the curl command to toggle the readiness probe status:

```
$ curl http://127.0.0.1:10081/readiness/status
```

Wait about 45 seconds and get the status of the healthy-monolith Pod using the kubectl get pods command:

```
$ kubectl get pods healthy-monolith
```

Use the kubectl describe command to get more details about the failing readiness probe:

```
$ kubectl describe pods healthy-monolith
```

Notice the details about failing probes.

```
Liveness:     http-get http://:81/healthz delay=5s timeout=5s period=
Readiness:    http-get http://:81/readiness delay=5s timeout=1s perio
```

Force the monolith container readiness probe to pass. Use the curl command to toggle the readiness probe status:

```
$ curl http://127.0.0.1:10081/readiness/status
```

If we use the `--watch` (`-w`) flag we can see when the status of the pod changes:

```
$ kubectl get pods healthy-monolith -w
```

Experiment with Liveness Probes

Building on what you learned in the previous tutorial use the kubectl port-forward and curl commands to force the monolith container liveness probe to fail. Observe how Kubernetes responds to failing liveness probes.

```
$ curl http://127.0.0.1:10081/healthz/status
```

Quiz

- What happened when the liveness probe failed?
- What events where created when the liveness probe failed?

Clean Up

```
$ kubectl delete -f ./resources/healthy-monolith.yaml
```

Now it's your turn

Add Probes for our k8s-real-demo service

- Add Readiness probe to the Deployment
  - Hint: endpoints are already implemented in the code `server.go`
- Deploy the updated config
- Add Liveness probe to the Deployment
- Deploy the updated config
- Verify the probes are working

Resource Management

In this section we'll discuss:

- Quality of Service (QoS)
- Resource Quotas

The Resource Model

What exactly is a resource in Kubernetes?

CPU & Memory

- Accounted
- Scheduled
- Isolated

Local Storage (Disk or SSD)

- Accounted (restriction to single partition /)

Nvidia GPU

- Alpha support (1 GPU per-node)

There are two main kinds of resource management for containers/pods.

- Requests
- Limits

Requests

A Request must be fulfilled for a Pod to be scheduled.

Assume I have a pod I want to schedule.

- Request 512Mb RAM & 300m CPU
- Kubernetes looks for nodes with space.
- If a node exists, then the Pod is scheduled.

Requests

- A Pod will not be deployed if there is no suitable node.
- Requested resources can be exceeded (with *potential* consequences).
- Is more of a guideline to resource usage within a Pod.

Limits

A Limit is ignored in the scheduling phase of the Pod lifecycle.

- It is hard, if a Pod tries to exceed it, there are immediate consequences.
- Must be equal to, or greater than the Request (if defined).

Requests and Limits

Repercussions:

- Usage > Request: Pod may 'scavenge' excess resources.
- Usage > Limit: Pod is killed (RAM) or throttled (CPU).

Quality of Service

- In an over-committed system, need to prioritise killing Containers.
- Mark containers as more/less important.
- Kill off less important Containers first.

There are three tiers of QoS, in decreasing order of priority:

- Guaranteed
- Burstable
- Best-Effort

QoS - Guaranteed

- `limits` is non-zero, and set across all containers.
- `requests` are optionally defined and equal `limits`.
- last to be killed off.

```
containers:
    name: foo
        resources:
            requests:
                cpu: 100m
                memory: 100Mi
            limits:
                cpu: 100m
                memory: 100Mi
```

QoS - Burstable

- `requests` is non-zero, and set for at least one container.
- `limits` optionally defined and greater than `requests`.
- if `limits` is undefined, default to capacity of node.

```
containers:
  name: foo
      resources:
          requests:
              cpu: 100m
              memory: 100Mi
```

QoS - Best-Effort

- `requests` and `limits` are not set.
- uses whatever resources it needs/are available.
- first to be killed when resources become scarce.

```
containers:
  name: foo
```

Examples

Each container in a Pod may specify the amount of CPU it requests on a node.

CPU requests are used at schedule time, and represent a minimum amount of CPU that should be reserved for your container to run.

Let's demonstrate this concept using a simple container that will consume as much CPU as possible.

```
$ kubectl run cpustress --image=busybox --requests=cpu=100m \
-- md5sum /dev/urandom
```

This will create a single Pod on your node that requests 1/10 of a CPU, but it has no limit on how much CPU it may actually consume on the node.

To demonstrate this, you can use `kubectl top Pod <PODNAME>` to view the the used CPU shares. (This may take some time before metrics are available)

```
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
cpustress-4101692926-zqw2p    1/1       Running   0          1m
$ kubectl top pod cpustress-4101692926-zqw2p
NAME                          CPU(cores)   MEMORY(bytes)
cpustress-4101692926-zqw2p    924m         0Mi
```

As you can see it uses 924m of a 1vCPU machine.

If you scale your application, we should see that each Pod is given an equal proportion of CPU time.

```
$ kubectl scale deployment cpustress --replicas=9
```

Once all the Pods are running, you will see that each Pod is getting approximately an equal proportion of CPU time.

Clean Up

```
$ kubectl delete deployment cpustress
```

CPU Limit

Setting a limit will allow you to control the maximum amount of CPU that your container can burst to.

```
$ kubectl run cpustress --image=busybox --requests=cpu=100m \
--limits=cpu=200m -- md5sum /dev/urandom
```

You can verify that by using `kubectl top pod`:

```
$ kubectl top pod
NAME                          CPU(cores)   MEMORY(bytes)
cpustress-1437538636-wkzh7    199m         0Mi
```

If you scale your application, we should see that each Pod is consuming a maximum of 200m CPU shares.

```
$ kubectl scale deployment cpustress --replicas=9
```

Once all the Pods are running, you will see that each Pod is getting approximately an equal proportion of CPU time.

```
$ kubectl top pod
NAME                          CPU(cores)   MEMORY(bytes)
cpustress-2801690769-895wj    198m         0Mi
cpustress-2801690769-735dt    198m         0Mi
cpustress-2801690769-gm9cz    199m         0Mi
cpustress-2801690769-ljt1w    199m         0Mi
cpustress-2801690769-wt54n    199m         0Mi
cpustress-2801690769-7c3tc    198m         0Mi
cpustress-2801690769-f2blv    199m         0Mi
cpustress-2801690769-7fm9n    201m         0Mi
cpustress-2801690769-6ssdk    198m         0Mi
```

Memory Requests

By default, a container is able to consume as much memory on the node as possible. It is recommended to specify the amount of memory your container will require to run.

Let's demonstrate this by creating a Pod that runs a single container which requests 100Mi of memory. The container will allocate and write to 200MB of memory every 2 seconds.

```
$ kubectl run memhog --image=derekwaynecarr/memhog \
  --requests=memory=100Mi --command -- /bin/sh \
  -c "while true; do memhog -r100 200m; sleep 1; done"
```

Verify the usage with `kubectl top pod`

```
$ kubectl top pod
NAME                       CPU(cores)   MEMORY(bytes)
memhog-328396322-dh03t     772m         200Mi
```

We request 100Mi, but have burst our memory usage to a greater value. That's called Burstable.

Clean Up

```
$ kubectl delete deploy memhog
```

Memory Limits

If you specify a memory limit, you can constrain the amount of memory your container can use.

For example, let's limit our container to 200Mi of memory, and just consume 100MB.

```
$ kubectl run memhog --image=derekwaynecarr/memhog \
  --limits=memory=200Mi --command -- /bin/sh \
  -c "while true; do memhog -r100 100m; sleep 1; done"
```

```
$ kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
memhog-4201114837-svfjl     632m         100Mi
```

As you can see we are only consuming 100MB on the node.

Let's demonstrate what happens if you exceed your allowed memory usage by creating a replication controller whose Pod will keep being OOM killed because it attempts to allocate 300MB of memory, but is limited to 200Mi.

```
$ kubectl run memhog-oom --image=derekwaynecarr/memhog \
  --limits=memory=200Mi --command -- memhog -r100 300m
```

If we describe the created Pod, you will see that it keeps restarting until it goes into a CrashLoopBackOff.

```
$ kubectl get po
NAME                              READY     STATUS       RESTARTS   AGE
memhog-4201114837-svfjl           1/1       Running      0          11m
memhog-oom-3179143800-gmdbc       0/1       OOMKilled    5          3m
kubectl describe po memhog-oom-3179143800-gmdbc |grep -C 3 "Terminate
      memory:         200Mi
    State:          Waiting
      Reason:         CrashLoopBackOff
    Last State:         Terminated
      Reason:         OOMKilled
      Exit Code:    137
      Started:          Mon, 20 Mar 2017 22:51:57 +0100
```

What if my node runs out of memory?

With Guaranteed resources you are not in major danger of causing an OOM event on your node.

If any individual container consumes more than their specified limit, it will be killed.

With *BestEffort* and *Burstable* resources it is possible that a container will request more memory than is actually available on the node.

If this happens:

- The system will attempt to prioritize the containers that are killed based on their quality of service.
- This is done by using the OOMScoreAdjust feature in the Linux kernel
- Processes with lower values are preserved in favor of processes with higher values.
- The system daemons (kubelet, kube-proxy, docker) all run with low OOMScoreAdjust values.

Containers with *Guaranteed* memory are given a lower value than *Burstable* containers which have a lower value than *BestEffort* containers. As a consequence, containers with *BestEffort* should be killed before the other tier.

Resource Quota

Quotas can be set per-namespace.

- Maximum request and limit across all Pods.
- Applies to each type of resource (CPU, mem).
- User must specify request or limit.
- Maximum number of a particular kind of object. Ensure no user/app/department abuses the cluster.

Applied at admission time.

Pods which explicitly specify resource limits and requests will not pick up the namespace default values.

Apply the LimitRange (`resources/limits.yaml`) to the new namespace

```
$ kubectl create -f resources/limits.yaml -n limit-example-user-<X>
```

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mylimits
spec:
  limits:
  - max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 200m
      memory: 6Mi
    type: Pod
  - default:
      cpu: 300m
      memory: 200Mi
```

Create a deployment in this namespace

```
$ kubectl run nginx --image=nginx --replicas=1 \
  --namespace=limit-example-user-<X>
deployment "nginx" created
```

The default values of the namespace limit will be applied to this Pod (`kubectl describe pod <pod_name> --namespace=<namespace_name>`)

```
$ kubectl get pods --namespace=limit-example-user-<X>
NAME                      READY     STATUS    RESTARTS   AGE
nginx-2371676037-tfncs    1/1       Running   0          4m

$ kubectl describe pod nginx-2371676037-tfncs --namespace=limit-examp
...
Containers:
  nginx:
    Container ID:       docker://dece4453779a2664c045ea7edc21f41382d5
    Image:              nginx
    Image ID:           docker://sha256:e4e6d42c70b3f79c5d57c1705265
    Port:               <none>
    State:              Running
      Started:          Fri, 14 Jul 2017 12:44:24 +0000
    Ready:              True
    Restart Count:      0
```

What if we deploy a Pod which exceeds the limit, using the following yaml:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: invalid-pod
spec:
  containers:
  - name: kubernetes-serve-hostname
    image: gcr.io/google_containers/serve_hostname
    resources:
      limits:
        cpu: "3"
        memory: 100Mi
```

Try to deploy this and see what happens (`./resources/invalid-cpu-pod.yaml`)

Clean Up

```
$ kubectl delete --all pods
$ kubectl delete --all deployments
$ kubectl delete --all po,deploy -n limit-example-user-<X>
```

Now it's your turn

Add Resources to the k8s-real-demo service

- Add a resources section to the Deployment
- Add Limits and Requests for memory and CPU
- Apply the updated config
- Trigger high CPU on our app via the `/mineBitcoin` endpoint and verify the CPU limit is respected

Secure Configuration

ConfigMaps & Secrets

ConfigMaps and Secrets are designed to separate configuration data from your applications. This may include api keys, tokens, passwords, or just configuration parameters.

ConfigMaps

ConfigMaps hold both fine and coarse-grained data. Applications read configuration settings from both environment variables and files containing configuration data.

An example ConfigMap that contains both types of configuration:

```
apiVersion: v1
 kind: ConfigMap
 metadata:
   Name: example-configmap
 data:
   # property-like keys
   game-properties-file-name: game.properties
   ui-properties-file-name: ui.properties
   # file-like keys
   game.properties: |
     enemies=aliens
     lives=3
     enemies.cheat=true
     enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
```

The property-like keys of the ConfigMap are used as environment variables to the single container in the Deployment template, and the file-like keys populate a volume.

Consuming in Environment Variables

A ConfigMap can be used to populate the value of command line arguments. For example, consider the following ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
data:
  special.how: very
  special.type: charm
```

You can consume the keys of this ConfigMap in a pod using configMapKeyRef sections:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
```

When this pod is run, its output will include the following lines:

```
SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm
```

Setting Command-line Arguments

A ConfigMap can also be used to set the value of the command or arguments in a container. This is accomplished using the Kubernetes substitution syntax $(VAR_NAME).

Consuming in Volumes

A ConfigMap can also be consumed in volumes.

The most basic way to consume the ConfigMap is to populate the volume with files where the key is the file name and the content of the file is the value of the key.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
```

When the pod is running, the output will be:

```
very
```

Exercise

- Create a ConfigMap containing a `VERSION` variable
- Mount the ConfigMap in our example app Deployment
- Update the deployment and verify the new VERSION is used.
  - `curl IP:NODE_PORT`

Secrets

Provide a way to handle sensitive information:

- Database passwords
- API Keys
- TLS Certificates

This has become more complicated with microservices.

Ephemeral nature of services/containers means we need automation

Possible Approaches

- Bake into the Docker image
- Use environment variables
- Use a specialist solution
  - Orchestrator built-in solution
  - Dedicated K/V store
- Use volumes

Inside Docker Image

```
FROM mongo:3.0

ENV MONGODB_PASSWORD=DO_NOT_TELL_ANYONE
...
```

This is **not** a good idea.

Environment Variables

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mongo-secret
spec:
  containers:
  - name: mongo
    image: mongo
    ports:
    - containerPort: 27017
    env:
    - name: MONGODB_PASSWORD
      value: "GUEST"
```

PROS

- Popular solution
- Simple / easy to use

CONS

- Secret is still visible
- In config file
- Via kubectl or output of `env`

Hashicorp Vault

Advanced tool for securing, storing and controlling access to secrets.

- Leasing
- Key revocation
- Key rolling
- Auditing
- Unified API
- Encrypted Key Value Store

Kubernetes Secrets

- Can be encrypted since version 1.7
- Stored inside etcd
- Safer and more flexible than directly in image or yaml

Creating a Secret

We want to share the value *some-base64-encoded-payload* under the key *my-super-secret-key* as a Kubernetes Secret for a pod. First you need to base64-encode it like so:

```
echo -n some-base64-encoded-payload | base64
c29tZS1iYXNlNjQtZW5jb2RlZC1wYXlsb2Fk
```

Note the -n parameter with echo; this is necessary to suppress the trailing newline character.

We put the result of the base64 encoding into the secret manifest:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  my-super-secret-key: c29tZS1iYXNlNjQtZW5jb2RlZC1wYXlsb2Fk
```

Currently there is no other type available, also no other "encryption" method despite base64 encoding.

Using the Secret

Create a pod with the secret

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: secret
  name: secret
spec:
  volumes:
    - name: "secret"
      secret:
        secretName: mysecret
  containers:
    - image: nginx
      name: webserver
      volumeMounts:
        - mountPath: "/tmp/mysec"
```

```
kubectl create -f secret.yaml -f pod_secret.yaml
```

Validate the secret was configured correctly

```
kubectl exec -ti secret /bin/bash
cat /tmp/mysec/my-super-secret-key
```

One word of warning here, in case it's not obvious: `secret.yaml` should never ever be committed to a source control system such as Git. If you do that, you're exposing the secret and the whole exercise would have been for nothing.

Working with Secret Volumes

A secret volume is used to pass sensitive information, such as passwords, to pods. You can store secrets in the Kubernetes API and mount them as files for use by pods without coupling to Kubernetes directly.

Secret volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage. Important: You must create a secret in the Kubernetes API before you can use it

We'll serve a webpage via a Volume using secrets. This is definitely the wrong way to do things, but serves as an example of how secrets are dynamically updated.

Creating a Secret Using kubectl

```
kubectl create secret generic index --from-file=configs/secrets/index
```

Validate that it's been created

```
kubectl get secrets

kubectl describe secret index
```

Note that neither get nor describe shows the contents of the file by default

Using secret in a container

```
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: config
      readOnly: true
  volumes:
    - name: config
      secret:
        secretName: index
```

Create a nginx pod

```
kubectl create -f configs/secrets/nginx-controller.yaml
```

Validate that the nginx is working

```
kubectl port-forward <PODNAME> 8080:80 > /dev/null &
```

```
curl localhost:8080
Hello World
```

Update the message

```
echo "Hello again" > configs/secrets/index.html
```

or just with your editor

Update your secret

```
kubectl delete secret index
kubectl create secret generic index --from-file=configs/secrets/index
```

Mounted Secrets are updated automatically but it's using its local ttl-based cache for getting the current value of the secret. The total time is kubelet sync period + ttl of secrets cache in kubelet (~1min). But as we can't do at the moment `kubectl apply --from-file` this isn' working.

Validate that the index.html has updated

```
curl localhost:8080
Hello again
```

What have we Learned?

- How to use and configure Readiness and Liveness Probes
- How to specify Resources
- Setting Limits and Requests and QoS
- To use ConfigMaps and Secrets to share data with our applications

Next up advanced Kubernetes features...