shipper

May 03, 2022

Contents

CHAPTER 1

Documentation overview

- *Introduction*: Brief overview of what Shipper is and why you might be interested
- *Quick start*: 5 minutes to a working Shipper setup
- *User guide*: Using Shipper to deploy your code
- *Administrator guide*: Production installation, monitoring, and cluster fleet management
- *Limitations and known issues*
- *API Reference*: Detailed reference on the Shipper resources

CHAPTER 2

Introduction

2.1 Shipper

Shipper is an extension for Kubernetes to add sophisticated rollout strategies and multi-cluster orchestration.

It lets you use `kubectl` to manipulate objects which represent any kind of rollout strategy, like blue/green or canary. These strategies can deploy to one cluster, or many clusters across the world.

2.1.1 Why does Shipper exist?

Kubernetes is a wonderful platform, but implementing mature rollout strategies on top of it requires subtle multi-step orchestration: *Deployment* objects are a building block, not a solution.

When implemented as a set of scripts in CI/CD systems like Jenkins, GitLab, or Brigade, these strategies can become hard to debug, or leave out important properties like safe rollbacks.

These problems become more severe when the rollout targets multiple Kubernetes clusters in multiple regions: the complex, multi-step orchestration has many opportunities to fail and leave clusters in inconsistent states.

Shipper helps by providing a higher level API for complex rollout strategies to one or many clusters. It simplifies CI/CD pipeline scripts by letting them focus on the parts that matter to that particular application.

2.1.2 What is Shipper from a technical point of view?

Shipper is a collection of *Kubernetes controllers* that work with custom Kubernetes objects to provide a declarative API for advanced rollouts. These controllers continuously monitor the clusters involved, and converge them on the declared state. They act as control loops for the different aspects of a rollout: capacity management, traffic shifting, and Kubernetes object installation.

For example, you might have a Shipper Application like this:

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: Application
metadata:
  name: reviews-api
spec:
  template:
    # helm chart for this application
    chart:
      name: reviews-api
      version: "0.0.1"
      repoUrl: https://charts.example.com
    # how to select clusters to deploy to
    clusterRequirements:
      regions:
      - name: us-east1
    # the rollout strategy
    strategy:
      steps:
      - name: canary
        capacity:
          incumbent: 100
          contender: 10
        traffic:
          incumbent: 9
          contender: 1
      - name: all-in
        capacity:
          incumbent: 0
          contender: 100
        traffic:
          incumbent: 0
          contender: 10
    # the values for the helm chart
    values:
      image:
        repository: image-registry.example.com/reviews-api
        tag: v0.1.0
```

In this example, we're defining an Application named `reviews-api`. It uses a Helm Chart of the same name, and deploys to a cluster in the **us-east1** region. It uses a two step rollout strategy: a basic canary step with a bit of traffic for the new version, then "all-in". It populates the Helm Chart with values specifying the image tag.

In order to make this declared state a reality, Shipper will select a matching cluster, install the Chart objects into that cluster, and with your guidance, progress through the rollout strategy until the new release is fully live.

2.1.3 Multi-cluster, multi-region, multi-cloud

Shipper can deploy your application to multiple clusters in different regions.

It expects a Kubernetes API, so it should work with any compliant Kubernetes implementation like GKE or AKS. If you can use `kubectl` with it, chances are, you can use Shipper with it as well.

2.1.4 Release Management

Shipper doesn't just copy-paste your code onto multiple clusters for you – it allows you to customize the rollout strategy fully. This allows you to craft a rollout strategy with the appropriate speed/risk balance for your particular situation.

After each step of the rollout strategy, Shipper pauses to wait for another update to the *Release* object. This check-pointing approach means that rollouts are fully declarative, scriptable, and resumable. Shipper can keep a rollout on a particular step in the strategy for ten seconds or ten hours. At any point the rollout can be safely aborted, or moved backwards through the strategy to return to an earlier state.

2.1.5 Roll Backs

Since Shipper keeps a record of all your successful releases, it allows you to roll back to an earlier release very easily.

2.1.6 Charts As Input

Shipper installs a complete set of Kubernetes objects for a given application.

It does this by relying on Helm, and using Helm Charts as the unit of configuration deployment. Shipper's Application object provides an interface for specifying values to a Chart just like the `helm` command line tool.

2.2 Getting help

We're happy to take bug reports on the GitHub repo.

For user questions or general discussion you can find us on #shipper on the Kubernetes Slack.

CHAPTER 3

Installing Shipper

3.1 Step 0: procure a cluster

The rest of this document assumes that you have access to a Kubernetes cluster and admin privileges on it. If you don't have this, check out docker desktop, kind, microk8s or minikube. Cloud clusters like GKE are also fine. Shipper requires Kubernetes 1.17 or later, and you'll need to be an admin on the cluster you're working with.[1]

Make sure that `kubectl` works and can connect to your cluster before continuing.

3.1.1 Setting up kind clusters

How to set-up an application kind cluster and a management kind cluster:
We would like to setup two clusters, *mgmt* and *app*.

Lets write a `kind.yaml` manifest to configure our clusters:

```
:caption: kind.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
```

Now we'll use this to create the clusters:

```
$ kind create cluster --name app --config kind.yaml --image kindest/node:v1.15.7
$ kind create cluster --name mgmt --config kind.yaml --image kindest/node:v1.15.7
```

Congratulations, you have created your clusters!

[1] For example, on GKE you need to bind yourself to cluster-admin before `shipperctl` will work.

3.2 Step 1: get shipperctl

`shipperctl` automates setting up clusters for Shipper. Grab the tarball for your operating system, extract it, and stick it in your `PATH` somewhere.

You can find the binaries on the GitHub Releases page for Shipper.

3.3 Step 2: write a cluster manifest

`shipperctl` expects a manifest of clusters to configure. It uses your `~/.kube/config` to translate context names into cluster API server URLs. Find out the name of your context like so:

```
$ kubectl config get-contexts
CURRENT   NAME             CLUSTER          AUTHINFO           NAMESPACE
          kind-app         kind-app         kind-app
*         kind-mgmt        kind-mgmt        kind-mgmt
```

In my setup, the context name of the application cluster is **kind-app**.

This configuration will allow management cluster to communicate with application cluster. The cluster API server URL stored in the kubeconfig is a local address (127.0.0.1), we need an actual ip address for our kind-app cluster. This is how you can get it:

```
$ kind get kubeconfig --name app --internal | grep server
```

Note that **app** is the name we gave to `kind` when creating the application cluster. Copy the URL of the server.

Now let's write a `clusters.yaml` manifest to configure Shipper here:

```
:caption: clusters.yaml

applicationClusters:
- name: kind-app
  region: local
  apiMaster: "SERVER_URL"
```

Paste your server URL as a string.

3.4 Step 3: Setup the Management Cluster

Before you run `shipperctl`, make sure that your `kubectl` context is set to the management cluster:

```
$ kubectl config get-contexts
CURRENT   NAME        CLUSTER                  AUTHINFO         NAMESPACE
          kind-app    kind-app                 kind-app
*         kind-mgmt   kind-mgmt                kind-mgmt
```

First we'll setup all the needed resources in the management cluster:

```
    $ shipperctl clusters setup management -n shipper-system
    Setting up management cluster:
    Registering or updating custom resource definitions... done
Creating a namespace called shipper-system... already exists. Skipping
```

(continues on next page)

(continued from previous page)

```
Creating a namespace called rollout-blocks-global... already exists. Skipping
Creating a service account called shipper-management-cluster... already exists.␣
↪Skipping
Creating a ClusterRole called shipper:management-cluster... already exists. Skipping
Creating a ClusterRoleBinding called shipper:management-cluster... already exists.␣
↪Skipping
Checking if a secret already exists for the validating webhook in the shipper-system␣
↪namespace... yes. Skipping
Creating the ValidatingWebhookConfiguration in shipper-system namespace... done
Creating a Service object for the validating webhook... done
Finished setting up management cluster
```

3.5 Step 4: deploy shipper

Now that we have the namespace, custom resource definitions, role bindings, service accounts, and so on, let's create the Shipper *Deployment*:

```
$ kubectl --context kind-mgmt create -f https://github.com/bookingcom/shipper/
↪releases/latest/download/shipper.deployment.yaml
deployment.apps/shipper created
```

This will create an instance of Shipper in the `shipper-system` namespace.

3.6 Step 5: Join the Application cluster to the Management cluster

Now we'll give `clusters.yaml` to `shipperctl` to configure the cluster for Shipper:

```
$ shipperctl clusters join -f clusters.yaml -n shipper-system
Creating application cluster accounts in cluster kind-app:
Creating a namespace called shipper-system... already exists. Skipping
Creating a service account called shipper-application-cluster... already exists.␣
↪Skipping
Creating a ClusterRoleBinding called shipper:application-cluster... already exists.␣
↪Skipping
Finished creating application cluster accounts in cluster kind-app

Joining management cluster to application cluster kind-app:
Creating or updating the cluster object for cluster kind-app on the management␣
↪cluster... done
Checking whether a secret for the kind-app cluster exists in the shipper-system␣
↪namespace... yes. Skipping
Finished joining management cluster to application cluster kind-app
```

3.7 Step 6: do a rollout!

Now you should have a working Shipper installation. *Let's roll something out!*

3.8 Namespace manager

By design, Shipper does not create namespaces in the application cluster. Shipper requires the existence of a namespace in the application cluster with the same name as the namespace in management cluster where the *Application* objects is installed. In case the namespace does not exist in the application cluster, and this application cluster is selected for a *Release*, Shipper will continue to try and install the charts, and fail. This loop will end only when the namespace is created in the application cluster, or this application cluster is not selected anymore (by deleting the *Release* or *Application* objects).

To help with this, we recommend having some sort of a namespace manager tool. This can be a simple controller that installs a namespace in all the application clusters for each namespace existing in the management cluster, or a more complex tool, depending on your needs.

CHAPTER 4

User guide

4.1 Rolling out with Shipper

Note: This documentation assumes that you have set up Shipper in two clusters. `kind-mgmt` is the name of the context that points to the *management* cluster, and `kind-app` is the name of the context that points to the *application* cluster.

Rollouts with Shipper are all about transitioning from an old *Release*, the **incumbent**, to a new *Release*, the **contender**. If you're rolling out an *Application* for the very first time, then there is no **incumbent**, only a **contender**.

In general Shipper tries to present a familiar interface for people accustomed to *Deployment* objects.

4.1.1 Application object

Here's the Application object we'll use:

```
apiVersion: shipper.booking.com/v1alpha1
kind: Application
metadata:
  name: super-server
spec:
  revisionHistoryLimit: 3
  template:
    chart:
      name: nginx
      repoUrl: https://raw.githubusercontent.com/bookingcom/shipper/master/test/e2e/
↪testdata
      version: 0.0.1
    clusterRequirements:
      regions:
      - name: local
```

(continues on next page)

(continued from previous page)

```
    strategy:
      steps:
      - capacity:
          contender: 1
          incumbent: 100
        name: staging
        traffic:
          contender: 0
          incumbent: 100
      - capacity:
          contender: 100
          incumbent: 0
        name: full on
        traffic:
          contender: 100
          incumbent: 0
    values:
      replicaCount: 3
```

Copy this to a file called `app.yaml` and apply it to your Kubernetes management cluster:

```
$ kubectl --context kind-mgmt apply -f app.yaml
```

This will create an *Application* and *Release* object. Shortly thereafter, you should also see the set of Chart objects: a *Deployment*, a *Service*, and a *Pod*.

4.1.2 Checking progress

There are a few different ways to figure out how your rollout is going.

We can check in on the *Release* to see the progress we're making:

`.status.achievedStep`

This field is the definitive answer for whether Shipper considers a given step in a rollout strategy complete.

```
$ kubectl --context kind-mgmt get rel super-server-83e4eedd-0 -o json | jq .status.
↪achievedStep
null
$ # "null" means Shipper has not written the achievedStep key, because it hasn't␣
↪finished the first step
$ kubectl get rel -o json | jq .items[0].status.achievedStep
{
  "name": "staging",
  "step": 0
}
```

If everything is working, you should see one *Pod* active/ready.

`.status.conditions`

Just like any other object, the `status` field of a *Release* object contains information on anything that is going wrong, and anything that is going right:

This set of conditions shows that the strategy hasn't been executed because Shipper can not contact the *application* cluster called `kind-app`.

`.status.strategy.conditions`

For a more detailed view of what's happening while things are in between states, you can use the Strategy conditions.

```
$ kubectl --context kind-mgmt get rel super-server-83e4eedd-0 -o json | jq .status.
↪strategy.conditions
[
  {
    "lastTransitionTime": "2018-12-09T10:00:55Z",
    "message": "clusters pending capacity adjustments: [microk8s]",
    "reason": "ClustersNotReady",
    "status": "False",
    "type": "ContenderAchievedCapacity"
  },
  {
    "lastTransitionTime": "2018-12-09T10:00:55Z",
    "status": "True",
    "type": "ContenderAchievedInstallation"
  }
]
```

These will tell you which part of the step Shipper is currently working on. In this example, Shipper is waiting for the desired capacity in the microk8s cluster. This means that Pods aren't ready yet.

`.status.strategy.state`

Finally, because the Strategy conditions can be kind of a lot to parse, they are summarized into `estatus.strategy.state`.

```
$ kubectl get rel super-server-83e4eedd-0 -o json | jq .status.strategy.state
{
  "waitingForCapacity": "True",
  "waitingForCommand": "False",
  "waitingForInstallation": "False",
  "waitingForTraffic": "False"
}
```

The *troubleshooting guide* has more information on how to dig deep into what's going on with any given *Release*.

4.1.3 Advancing the rollout

So now that we've checked on our *Release* and seen that Shipper considers step 0 achieved, let's advance the rollout:

```
$ kubectl --context kind-mgmt patch rel super-server-83e4eedd-0 --type=merge -p '{
↪"spec":{"targetStep":1}}'
```

I'm using `patch` here to keep things concise, but any means of modifying objects will work just fine.

Now, if you've got your `kind-app` context set to the same namespace as your **Application** object in the *management* cluster, you should be able to see 2 more pods spin up:

```
$ kubectl --context kind-app get po
NAME                                              READY STATUS  RESTARTS AGE
super-server-83e4eedd-0-nginx-5775885bf6-76l6g    1/1   Running 0        7s
super-server-83e4eedd-0-nginx-5775885bf6-9hdn5    1/1   Running 0        7s
super-server-83e4eedd-0-nginx-5775885bf6-dkqbh    1/1   Running 0        3m55s
```

And confirm that Shipper believes this rollout to be done:

```
$ kubectl --context kind-mgmt get rel -o json | jq .items[0].status.achievedStep
{
  "name": "full on",
  "step": 1
}
```

That's it! Doing another rollout is as simple as editing the *Application* object, just like you would with a *Deployment*. The main principle is patching the *Release* object to move from step to step.

4.2 Troubleshooting Shipper

4.2.1 Prerequisites

To troubleshoot deployments effectively you need to be familiar with core Kubernetes and Shipper concepts (*very briefly* explained below) and be comfortable running *kubectl* commands.

4.2.2 Fundamentals

Shipper objects form a hierarchy:

```
Application
    |
Release
    |
InstallationTarget
CapacityTarget
TrafficTarget
```

You already know Applications and Releases, but there's more. Below Releases you have what we call "target objects". Each represents an important chunk of work we do when rolling out:

Kind	Shortname	Description
InstallationTarget	it	Install charts in *application clusters*
CapacityTarget	ct	Scale deployments up and down to reach desired number of pods
TrafficTarget	tt	Orchestrate traffic by moving pods in and out of the LB

The list is ordered (e.g. we can't manipulate traffic before there are pods).

4.2.3 The universal troubleshooting algorithm

Shipper is a fairly complex system that runs on top of an even more complex one. Things can fail in many different ways. It's not really feasible for us to list all the possible problems and solutions for them. Instead, we'll give you a rough algorithm that should help you deal with commonly encountered problems.

To summarise, the algorithm is roughly:

1. Find what stage you're at by looking at Release conditions and state
2. Inspect the corresponding target object's conditions
3. Act accordingly

In the next sections we'll explain in more detail how to do that.

Finding where you are

Before we attempt to fix anything we need to make sure we know where we are in the rollout process. The starting point is almost always looking at your Release's status:

```
$ kubectl describe rel nginx-vj7sn-7cb440f1-0
...
Status:
  Achieved Step:
    Name:  staging
    Step:  0
  Conditions:
    Last Transition Time:  2018-07-27T07:21:14Z
    Status:                True
    Type:                  Scheduled
  Strategy:
    Conditions:
      Last Transition Time:  2018-07-27T07:23:29Z
      Message:               clusters pending capacity adjustments: [minikube]
      Reason:                ClustersNotReady
      Status:                False
      Type:                  ContenderAchievedCapacity
      Last Transition Time:  2018-07-27T07:23:29Z
      Status:                True
      Type:                  ContenderAchievedInstallation
    State:
      Waiting For Capacity:      True
      Waiting For Command:       False
      Waiting For Installation:  False
      Waiting For Traffic:       False
...
```

We already looked at *status.strategy.state.waitingForCommand* but there are more fields there: one for every type of target objects. If your rollout isn't finished and not waiting for input, these fields tell you which stage you're at.

Field	Meaning
`waitingForInstallation`	Waiting for the chart to be installed in application clusters
`waitingForCapacity`	Waiting for the contender to scale up and/or the incumbent to scale down
`waitingForTraffic`	Waiting for the contender traffic to increase and/or the incumbent to decrease

Release conditions and strategy conditions

Category	Description
Object conditions	Conditions that apply to the object itself. **All** objects have this.
Strategy conditions	Conditions that apply to the strategy of the Release that's being rolled out. Only **Releases** have this.

In the example above, under `.status.strategy` we can find a condition called `ContenderAchievedCapacity`, saying there're still clusters pending capacity adjustments.

Target objects

The next step would be to look at the corresponding target object. Since we're waiting for capacity, we'll be looking at CapacityTarget. The object will have the same name as the release but different kind:

```
$ kubectl describe ct nginx-vj7sn-7cb440f1-0
...
Status:
  Clusters:
    Achieved Percent:    0
    Available Replicas:  0
    Conditions:
      Last Transition Time:  2018-07-27T07:23:29Z
      Status:                True
      Type:                  Operational
      Last Transition Time:  2018-07-27T07:23:29Z
      Message:               there are 1 sad pods
      Reason:                PodsNotReady
      Status:                False
      Type:                  Ready
    Name:                    minikube
    Sad Pods:
      Condition:
        Last Probe Time:        <nil>
        Last Transition Time:  2018-07-27T07:23:14Z
        Status:                True
        Type:                  PodScheduled
      Containers:
        Image:     nginx:boom
        Image ID:
        Last State:
        Name:           nginx
        Ready:          false
        Restart Count:  0
        State:
          Waiting:
            Message:    Back-off pulling image "nginx:boom"
            Reason:     ImagePullBackOff
      Init Containers:  <nil>
      Name:             nginx-vj7sn-7cb440f1-0-nginx-9b5c4d7c9-2gjwl
...
```

Important: For installation the command would be `kubectl describe it <release name>`, for traffic `kubectl describe tt <release name>`.

If we inspect `.status.conditions` of the InstallationTarget we'll notice a condition called `Ready` which has status `False` and reason `PodsNotReady`. Further inspection will reveal that we have a pod called `nginx-vj7sn-7cb440f1-0-nginx-9b5c4d7c9-2gjwl` and that Kubernetes can't pull the Docker image for one if its containers:

```
Message:    Back-off pulling image "nginx:boom"
Reason:     ImagePullBackOff
```

The "boom" Docker tag clearly looks wrong. To fix this you can simply edit the application object and set the correct tag in *.spec.template.values*.

4.2.4 Other sources of useful information

Shipper emits Kubernetes events with useful information. You can look at that, if you prefer:

```
$ kubectl get events
...
1m          1h          238       nginx-vj7sn-7cb440f1-0.154528eb631aac75                  ␣
↪           CapacityTarget                               Normal   ␣
↪CapacityTargetChanged       capacity-controller       Set "default/nginx-vj7sn-
↪7cb440f1-0" status to {[{minikube 0 0 [{nginx-vj7sn-7cb440f1-0-nginx-9b5c4d7c9-
↪2gjwl [{nginx {&ContainerStateWaiting{Reason:ImagePullBackOff,Message:Back-off␣
↪pulling image "nginx:boom",} nil nil} {nil nil nil} false 0 nginx:boom  }] []
↪{PodScheduled True 0001-01-01 00:00:00 +0000 UTC 2018-07-27 09:23:14 +0200 CEST  }}
↪] [{Operational True 2018-07-27 09:23:29 +0200 CEST  } {Ready False 2018-07-27␣
↪09:23:29 +0200 CEST PodsNotReady there are 1 sad pods}]}]}
```

4.2.5 Typical failure scenarios

While we can't list all the possible failures we can list the ones that we think happen more often than others:

Failure	Description
Can't pull Docker image	Strategy condition `ContenderAchievedCapacity` is false, InstallationTarget's `Ready` condition is false and the message is something like "Back-off pulling image "nginx:boom""
Previous release is unhealthy	Release condition `IncumbentAchievedCapacity` is false and the message is something like "incumbent capacity is unhealthy in clusters: [minikube]". In this case, you can try describing the CapacityTarget from the previous release to find out what's wrong. If you're doing a rollout to fix that previous release, though, you can opt for proceeding to the next step in your strategy, as Shipper does not require a step to be completed before moving on to the next.
Can't fetch Helm chart	Release condition `Scheduled` is false and the message is something like "download https://charts.example.com/charts/nginx-0.1.42.tgz: 404"

4.2.6 Make sure you're on the right cluster!

There are cases where the user is checking on the wrong cluster and can't see the pods etc. To make sure you're on the right one:

```
$ kubectl get release
NAME                          CREATED AT
myrelease-cf68dfe8-0          23m

$ kubectl describe release <your app release> | grep release.clusters
Annotations:  shipper.booking.com/release.clusters=kube-us-east-1-a
```

CHAPTER 5

Operations and administration

Shipper is designed to make it easier to manage a fleet of Kubernetes clusters with many teams deploying code to them.

5.1 Cluster architecture

Shipper defines two kinds of Kubernetes clusters, **management** clusters and **application** clusters.

5.1.1 Management clusters

Management clusters are where Shipper itself runs. It has the Shipper *Custom Resource Definitions* installed, and is where application developers interact with the *Application* or *Release* objects. The **management** cluster stores the set of *Cluster* objects and associated *Secrets* that enable Shipper to connect to the **application** clusters.

Typically you have one of these per large deployment, or one with a standby.

5.1.2 Application clusters

Application clusters are where Shipper installs and rolls out user workloads. Shipper does not run any custom software in the **application** clusters: it only needs a service account and associated RBAC configuration.

5.1.3 Patterns

One management, many application

This is the standard arrangement if you have a fleet of Kubernetes clusters that you would like to manage with Shipper. The single management cluster provides application developers with a single place to interface with Shipper's objects and orchestrate their rollouts.

One-and-the-same

It is totally fine if the **management** cluster and the **application** cluster are the same. This is how Shipper is developed, and also how you would use Shipper if you only have a single Kubernetes cluster in your infrastructure. You can think about this configuration as using Shipper to provide a better *Deployment* object, but without any multi-cluster federation.

Multiple management, each with own set of application

While Shipper fully supports namespaces as units of multi-tenancy, it does not yet have any way to limit the set of clusters that an Application can select. So, if your organization has multiple groups of Kubernetes clusters that are consumed by disjoint sets of users, it might make sense to create a **management** cluster for each group of **application** clusters that need strong isolation between each other.

5.2 Using `shipperctl`

The `shipperctl` command is created to make using Shipper easier.

5.2.1 Setting Up Clusters Using `shipperctl clusters` Commands

To set up clusters to work with Shipper, you should create *ClusterRoleBindings*, *ClusterRoles*, *Roles*, *RoleBindings*, *Clusters*, and so forth.

Meet `shipperctl clusters`, which is made to make this easier.

There are two use cases for this set of commands.

First, you can use it to set up a local environment to run Shipper in, or to set up a fleet of clusters for the first time.

Second, you can integrate it into your continuous integration pipeline. Since these commands are idempotent, you can use it to apply the configuration of your clusters.

Note that these commands don't apply a Shipper deployment. You should *deploy Shipper* once you've run these commands.

The commands under `shipperctl clusters` should be run in this order if you're setting up a cluster for a very first time. Once you've followed this procedure, you can use the ones that apply to your situation.

Note that you need to change your context to point to the management cluster before running the following commands.

1. *shipperctl clusters setup management*: creates the *CustomResourceDefinitions*, *ServiceAccount*, *ClusterRoleBinding* and other objects Shipper needs to function correctly.
2. *shipperctl clusters join*: creates the *ServiceAccount* that Shipper is going to use on the **application** cluster, and copies its token back to the **management** cluster. This is so that *Shipper*, which runs on the **management** cluster, can modify Kubernetes objects on the **application** cluster. Once the token is created, this command also creates a *Cluster* object on the *management* cluster, which tells Shipper how to communicate with the **application** cluster.

All of these commands share a certain set of options. However, they each have their own set of options as well.

Below are the options that are shared between all the commands:

--kube-config `<path string>`
: The path to your `kubectl` configuration, where the contexts that `shipperctl` should use reside.

-n, --shipper-system-namespace <string>
The namespace Shipper is running in. This is the namespace where you have a *Deployment* running the Shipper image.

--management-cluster-context <string>
By default, `shipperctl` uses the context that was already set in your `kubeconfig`

(i.e. using `kubectl config use-context`). However, if that's not what you want, you can use this option to tell `shipperctl` to use another context.

`shipperctl clusters setup management`

As mentioned above, this command is used to set up the **management** cluster for use with Shipper.

--management-cluster-service-account <string>
the name of the service account Shipper will use for the management cluster (default "shipper-mgmt-cluster")

-g, --rollout-blocks-global-namespace <string>
the namespace where global RolloutBlocks should be created (default "rollout-blocks-global")

This is the namespace that the users or administrators of the **management** cluster will create a *RolloutBlock* object, so that all Shipper rollouts for *Applications* on that cluster would be disabled.

`shipperctl clusters join`

As mentioned above, this command is used to join the **management** and **application** clusters together using a `clusters.yaml` file. To know more about the format of that file, look at the *Clusters Configuration File Format* section.

--application-cluster-service-account <string>
the name of the service account Shipper will use in the application cluster (default "shipper-app-cluster")

-f, --file <string>
the path to a YAML file containing application cluster configuration (default "clusters.yaml")

Clusters Configuration File Format

The clusters configuration file is a *YAML* file. At the top level, you should specify two keys, `managementClusters` and `applicationClusters`. The clusters you specify under each key are your **management** and **application** clusters, respectively. Check out *Cluster Architecture* to learn more about what this means.

For each item in the list of **management** or **application** clusters, you can specify these fields:

- name (mandatory): This is the name of the cluster. When specified for an **application** cluster,

a *Cluster* object will be created on the **management** cluster, and will point to the **application**. - context (optional, defaults to the value of `name`): this is the name of the *context* from your *kubectl* configuration that points to this cluster. `shipperctl` will use this context to run commands to set up the cluster, and also to populate the URL of the API master. - Fields from the *Cluster* object (optional): you can specify any field from the *Cluster* object, and `shipperctl` will patch the Cluster object for you the next time you run it. The only field that is mandatory is `region`, which you have to specify to create any *Cluster* object.

Examples

Minimal Configuration

Here is a minimal configuration to set up a local *kind* instance, assuming that you have created a cluster called `mgmt` and a cluster called `app`:

```yaml
managementClusters:
- name: kind-mgmt # kind contexts are prefixed with `kind-`
applicationClusters:
- name: kind-app
  region: local
```

Specifying Cluster Fields

Here is something more interesting: having 2 application clusters, and marking one of them as unschedulable:

```yaml
managementCluster:
- name: eu-m
applicationClusters:
- name: eu-1
  region: eu-west
- name: eu-2
  region: eu-west
  scheduler:
    unschedulable: true
```

Using Google Kubernetes Engine (GKE) Context Names

If you're running on GKE, your cluster context names are likely to have underscores in them, like this: `gke_ACCOUNT_ZONE_CLUSTERNAME`. `shipperctl`'s usage of the context name as the name of the Cluster object will break, because Kubernetes objects are not allowed to have underscores in their names. To solve this, specify `context` explicitly in `clusters.yaml`, like so:

```yaml
managementCluster:
- name: eu-m # make sure this is a Kubernetes-friendly name
  context: gke_ACCOUNT_ZONE_CLUSTERNAME_MANAGEMENT # add this
applicationClusters:
- name: eu-1
  region: eu-west
  context: gke_ACCOUNT_ZONE_CLUSTERNAME_APP_1 # same here
- name: eu-2
  region: eu-west
  context: gke_ACCOUNT_ZONE_CLUSTERNAME_APP_2 # and here
  scheduler:
    unschedulable: true
```

5.2.2 Creating backups and restoring Using `shipperctl backup` Commands

`shipperctl backup prepare`

1. The backup must be created by a *shipperctl* command. This guarantees you can restore this backup. Acquire a backup file by running

```
$ kubectl config use-context mgmt-dev-cluster ##be sure to switch to correct context␣
↪of the management cluster before backing up
Switched to context "mgmt-dev-cluster"
$ shipperctl backup prepare -v -f bkup-dev-29-10.yaml
NAMESPACE   RELEASE NAME                 OWNING APPLICATION
default     super-server-dc5bfc5a-0     super-server
default2    super-server2-dc5bfc5a-0    super-server2
default3    super-server3-dc5bfc5a-0    super-server3
Backup objects stored in "bkup-dev-29-10.yaml"
```

The command's default format is yaml. This will create a file named "bkup-dev-29-10.yaml" and store the backup there in a yaml format.

2. Save the backup file in a storage system of your liking (for example, AWS S3)
3. That's it! Repeat steps 1+2 for all management clusters.

shipperctl backup restore

1. Download your latest backup from your selected storing system
2. Make sure that Shipper is down (*spec.replicas: 0*) before applying objects.
3. Use *shipperctl* to restore your backup:

```
$ kubectl config use-context mgmt-dev-cluster ##be sure to switch to correct␣
↪management context before restoring backing up
Switched to context "mgmt-dev-cluster"
$ shipperctl backup restore -v -f bkup-dev-29-10-from-s3.yaml
Would you like to see an overview of your backup? [y/n]: y
NAMESPACE   RELEASE NAME                 OWNING APPLICATION
default     super-server-dc5bfc5a-0     super-server
default2    super-server2-dc5bfc5a-0    super-server2
default3    super-server3-dc5bfc5a-0    super-server3
Would you like to review backup? [y/n]: y
- application:
    apiVersion: shipper.booking.com/v1alpha1
    kind: Application
  ...
  backup_releases:
  - capacity_target:
      apiVersion: shipper.booking.com/v1alpha1
      kind: CapacityTarget
    ...
    installation_target:
      apiVersion: shipper.booking.com/v1alpha1
      kind: InstallationTarget
    ...
    release:
      apiVersion: shipper.booking.com/v1alpha1
      kind: Release
    ...
    traffic_target:
      apiVersion: shipper.booking.com/v1alpha1
      kind: TrafficTarget
    ...
...
Would you like to restore backup? [y/n]: y
```

(continues on next page)

(continued from previous page)

```
application "default/super-server" created
release "default/super-server-dc5bfc5a-0" owner reference updates with uid "a6c587cb-
↪624e-44ec-b267-b48630b0ed1c"
release "default/super-server-dc5bfc5a-0" created
installation target "default/super-server-dc5bfc5a-0" owner reference updates with␣
↪uid "9ccfd876-7f4f-4b1c-9c10-653d295e21d2"
installation target "default/super-server-dc5bfc5a-0" created
traffic target "default/super-server-dc5bfc5a-0" owner reference updates with uid
↪"9ccfd876-7f4f-4b1c-9c10-653d295e21d2"
traffic target "default/super-server-dc5bfc5a-0" created
capacity target "default/super-server-dc5bfc5a-0" owner reference updates with uid
↪"9ccfd876-7f4f-4b1c-9c10-653d295e21d2"
capacity target "default/super-server-dc5bfc5a-0" created
...
```

- The command's default format is yaml. This will apply the backup from file "bkup-dev-29-10-from-s3.yaml" while maintaining owner references between an application and its releases and between release and its target objects.
- The backup file must be created using *shipperctl backup prepare* command.

5.3 Monitoring Shipper

5.4 Cluster fleet management

5.5 Blocking rollouts

You can block rollouts in a specific namespace, or all namespaces (if you have the permissions to do so). To do so, you simply create a *RolloutBlock* object. The *RolloutBlock* object represents a rollout block in a specific namespace. When the object is deleted, the block is lifted.

5.5.1 RolloutBlock object

Here's an example for a RolloutBlock object we'll use:

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: RolloutBlock
metadata:
  name: dns-outage
  namespace: rollout-blocks-global # for global rollout block. for a local one use␣
↪the correct namespace.
spec:
  message: DNS issues, troubleshooting in progress
  author:
    type: user
    name: jdoe # This indicates that a rollout block was put in place by user 'jdoe'
```

Copy this to a file called `globalRolloutBlock.yaml` and apply it to your Kubernetes cluster:

```
$ kubectl apply -f globalRolloutBlock.yaml
```

This will create a *Global RolloutBlock* object. In order to create a namespace rollout block, simply state the relevant namespace in the yaml file. An example for a namespaced RolloutBlock object:

```
apiVersion: shipper.booking.com/v1alpha1
kind: RolloutBlock
metadata:
  name: fairy-investigation
  namespace: fairytale-land
spec:
  message: Investigating current Fairy state
  author:
    type: user
    name: fgodmother
```

While this object is in the system, there can not be any change to the *.Spec* of any object. Shipper will reject the creation of new objects and patching of existing releases.

5.5.2 Overriding a rollout block

Rollout blocks can be overridden with an annotation applied to the *Application* or *Release* object which needs to bypass the block. This annotation will list each RolloutBlock object that it overrides with a fully-qualified name (namespace + name).

For example, mending our Application object to override the global rollout block that we set in place:

```
apiVersion: shipper.booking.com/v1alpha1
kind: Application
metadata:
  name: super-server
  annotations:
    shipper.booking.com/rollout-block.override: rollout-blocks-global/dns-outage
spec:
  revisionHistoryLimit: 3
  template:
    # ... rest of template omitted here
```

The annotation may reference multiple blocks:

```
shipper.booking.com/rollout-block.override: rollout-blocks-global/dns-outage,frontend/
↪demo-to-investors-in-progress
```

The block override annotation format is CSV.

The override annotation **must** reference specific, fully-qualified *RolloutBlock* objects by name. Non-existing blocks enlisted in this annotation are not allowed. If there exists a Release object for a specific application, the release should be the one overriding it.

5.5.3 Application and Release conditions

Application and Release objects will have a *.status.conditions* entry which lists all of the blocks which are currently in effect.

For example:

```yaml
apiVersion: shipper.booking.com/v1
kind: Application
metadata:
  name: ui
  namespace: frontend
spec:
  # ... spec omitted
status:
  conditions:
  - type: Blocked
    status: True
    reason: RolloutsBlocked
    message: rollouts blocked by: rollout-blocks-global/dns-outage
```

This will be accompanied with an event (can be viewed with `kubectl describe application ui -n frontend`). For example:

```
Events:
  Type     Reason              Age                 From                       Message
  ----     ------              ----                ----                       -------
  Warning  RolloutBlock        3s (x3 over 5s)     application-controller    rollout-
↪blocks-global/dns-outage
```

5.5.4 Checking a rollout block status

There are a few simple ways to know which objects are overriding your RolloutBlock object.

`.status.overrides`

This fields will state all living Application and Release objects that override this RolloutBlock object.

```
$ kubectl -n rollout-blocks-global get rb dns-outage -o yaml
```

This might look like this:

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: RolloutBlock
metadata:
  name: dns-outage
  namespace: rollout-blocks-global
# ... spec omitted
status:
  # associated because 'shipper-system/dns-outage' is referenced in override␣
↪annotation
  overrides:
    applications: default/super-server
    release: default/super-server-83e4eedd-0
```

`output wide`

This will show all information about all rollout blocks in the namsespace (default if not specify, *rollout-blocks-global* for all global RolloutBlocks ,‘–all-namespaces‘ for all rollout blocks)

```
$ kubectl -n rollout-blocks-global get rb -o wide
```

This might look like this:

```
NAMESPACE               NAME         MESSAGE                                 AUTHOR␣
↪TYPE    AUTHOR NAME    OVERRIDING APPLICATIONS    OVERRIDING RELEASES
rollout-blocks-global   dns-outage   DNS issues, troubleshooting in progress   user    ␣
↪       jdoe           default/super-server       default/super-server-83e4eedd-0
```

CHAPTER 6

Limitations and known issues

Shipper is just software, and all software has limits. Here are the highlights for Shipper currently. Some of these are not principal problems, just shortcuts that we took while building Shipper.

6.1 Chart restrictions

Shipper expects a few properties to be true about the Chart it is rolling out. We hope to loosen or remove most of these restrictions over time.

6.1.1 Only *Deployments*

The Chart must have exactly one *Deployment* object. The name of the *Deployment* should be templated with `{{.Release.Name}}`. The *Deployment* object should have `apiVersion: apps/v1`.

Shipper cannot yet perform roll outs for *StatefulSets*, *HorizontalPodAutoscalers*, or bare *ReplicaSets*. These objects can be present in the Chart, but Shipper only knows how to manipulate *Deployment* objects to scale capacity over the course of a rollout.

6.1.2 *Services*

The Chart must contain either:

- exactly one *Service*, or
- exactly one *Service* labeled with the label `shipper-lb: production`.

The name of the *Service* should be fixed: either a literal in the Chart template, or a value which does not change from release to release.

The *Service* should have a `selector` which matches the application, not a single release. A *Service* with `release: {{ .Release.Name }}` as part of the *Service* `selector` will cause Shipper to error, as it will not be able to balance traffic between multiple *Releases*.

If you cannot modify the Chart you're rolling out, you can ask Shipper to remove the `release` selector from the *Service* `selector` by adding the `enable-helm-release-workaround: "true"` label to your *Application*. This workaround helps make Charts created with `helm create` work out of the box.

6.2 Load balancing

Shipper uses Kubernetes' built-in mechanism for shifting traffic: labeling *Pods* to add or remove them to a *Service's* `selector`. This means you don't need any special support in your Kubernetes clusters, but it has several drawbacks.

We hope to mitigate these by adding support for service mesh providers as traffic shifting backends.

6.2.1 Pod-based traffic shifting

Traffic shifting happens at the granularity of *Pods*, not requests. While Shipper's interface specifes a traffic weight, small fleets of *Pods* may find that their actual weight differs significantly from the one they requested.

6.2.2 New *Pods* don't get traffic if Shipper is not working

Shipper adds the `shipper-traffic-status: enabled` label to *Pods* after they start. This allows Shipper to correctly manage the number of *Pods* exposed to traffic. However, if a *Pod* is deleted and Shipper is not currently running or cannot contact the cluster, the new *Pod* spawned by the *ReplicaSet* will not get traffic until Shipper is working again.

The primary issue is that we cannot "cork" a successfully completed rollout by adding the traffic label to the *Deployment* or *ReplicaSet* without triggering a native *Deployment*-based rollout. We could solve this by working directly with *ReplicaSets* instead of *Deployments*, but that's probably working against the grain of the ecosystem (most charts contain *Deployments*).

6.3 Lock-step rollouts

Shipper is good at making sure that all clusters involved in a rollout are in the same state. It does this by ensuring that all clusters are in the correct state before marking a rollout step as complete.

However, this means that Shipper cannot perform cluster-by-cluster rollouts, like first `kube-us-east1-a`, then `kube-eu-west2-b`. Our "federation" layer supports this, but we have not yet designed the extension to our strategy language to describe this kind of rollout.

This cluster-by-cluster strategy is important when limiting traffic or capacity exposure to a new change is not enough to mitigate risk: for example, perhaps the new version will change a cluster-local schema once it starts running.

CHAPTER 7

API Reference

7.1 High-level APIs

These objects represent the primary user interface to Shipper. They are the control and reporting layers for any rollout operation.

7.1.1 Application

An *Application* object represents a single application Shipper can manage on a user's behalf. In this case, the term "application" means 'a collection of Kubernetes objects installed by a single Helm chart'.

Application objects are a *user interface*, and are the primary way that application developers trigger new rollouts.

This is accomplished by editing an Application's `.spec.template` field. The *template* field is a mold that Shipper will use to stamp out a new *Release* object on each edit. This model is identical to to Kubernetes *Deployment* objects and their `.spec.template` field, which serves as a mold for *ReplicaSet* objects (and by extension, *Pod* objects).

Application's `.spec.template.chart` contains ambiguity by design: a user is expected to provide either a specific chart version or a *SemVer constraint* defining the range of acceptable chart versions. Shipper will resolve an appropriate available chart version and pin the *Release* on it. Shipper resolves the version in-place: it will substitute the initial constraint with a specific resolved version and preserve the initial constraint in the Application annotation named `shipper.booking.com/app.chart.version.raw`.

The resolved `.spec.template` field will be copied to a new *Release* object under the `.spec.environment` field during deployment.

Example

Listing 1: Application example

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: Application
metadata:
  name: reviews-api
spec:
  revisionHistoryLimit: 1
  template:
    chart:
      name: reviews-api
      version: "~0.1"
      repoUrl: https://charts.example.com
    clusterRequirements:
      capabilities:
      - gpu
      - high-memory-nodes
      regions:
      - name: us-east1
    strategy:
      steps:
      - name: staging
        capacity:
          incumbent: 100
          contender: 1
        traffic:
          incumbent: 100
          contender: 0
      - name: canary
        capacity:
          incumbent: 10
          contender: 90
        traffic:
          incumbent: 10
          contender: 90
      - name: full on
        capacity:
          incumbent: 0
          contender: 100
        traffic:
          incumbent: 0
          contender: 100
    values:
      replicaCount: 2
```

Spec

`.spec.revisionHistoryLimit`

`revisionHistoryLimit` is an optional field that represents the number of associated *Release* objects in `.status.history`.

If you're using Shipper to configure development environments, `revisionHistoryLimit` can be a small value, like `1`. In a production setting it should be set to a larger number, like `10` or `20`. This ensures that you have plenty of rollback targets to choose from if something goes wrong.

.spec.template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a *Release* template. It has the same schema as the *.spec.environment* in a *Release* object.

Application's `.spec.template.chart` can define either a specific chart version, or a SemVer constraint.

Please refer to *Semantic Version Ranges* section for more details on supported constraints.

Status

.status.history

`history` is the sequence of *Releases* that belong to this *Application*. This list is ordered by generation, old to new: the oldest *Release* is at the start of the list, and the most recent (the **contender**) at the bottom.

.status.conditions

All conditions contain five fields: `lastTransitionTime`, `status`, `type`, `reason`, and `message`. Typically `reason` and `message` are omitted in the expected case, and populated in the error or unexpected case.

type: Aborting

This condition indicates whether an abort is currently in progress. An abort is when the latest *Release* (the **contender**) is deleted, triggering an automatic rollback to the **incumbent**.

Type	Status	Reason	Description
Aborting	True	N/A	The **contender** was deleted, triggering an abort. The *Application* `.spec.template` will be overwritten with the *Release* `.spec.environment` of the **incumbent**.
Aborting	False	N/A	No abort is occurring.

type: ReleaseSynced

This condition indicates whether the **contender** *Release* reflects the current state of the *Application* `.spec.template`.

Type	Status	Reason	Description
ReleaseSynced	True	N/A	Everything is OK: **Release** `.spec.environment` and **Application** `.spec.template` are in sync.
ReleaseSynced	False	CreateReleaseFailed	The API call to Kubernetes to create the Release object failed. Check `message` for the specific error.

type: RollingOut

This condition indicates whether a rollout is currently in progress. A rollout is in progress if the **contender** *Release* object has not yet achieved the final step in the rollout strategy.

Type	Status	Reason	Description
RollingOut	False	N/A	No rollout is in progress.
RollingOut	True	N/A	A rollout is in progress. Check `message` for more details.

type: ValidHistory

This condition indicates whether the *Releases* listed in `.status.history` form a valid sequence.

Type	Status	Reason	Description
ValidHistory	True	N/A	Everything is OK. All *Releases* have a valid generation annotation.
ValidHistory	False	BrokenReleaseGeneration	One of the *Releases* does not have a valid generation annotation. Check `message` for more details.
ValidHistory	False	BrokenApplicationObservedGeneration	The *Application* has an invalid `highestObservedGeneration` annotation. check `message` for more details.

Semantic Version Ranges

Shipper supports an extended range of semantic version constraints in Application's `.spec.template.chart.version`.

This section highlights the major features of supported SemVer constraints. For a full reference please see the underlying library spec.

Composition

SemVer specifications are composable: there are 2 composition operators defined: - `,`: stands for AND - `||`: stands for OR

In the example `>=1.2.3, <3.4.5 || 6.7.8` the constraint defines a range where any version between 1.2.3 inclusive *and* 3.4.5 non-inclusive, *or* a specific version 6.7.8 would satisfy it.

Trivial Comparisons

Trivial comparison constraints belong to a category of equality check relationships.

The range of comparison checks is defined as: - `=`: strictly equal to - `!=`: not equal to - `>`: greater than (non-inclusive) - `<`: less than (non-inclusive) - `>=`: greater than or equal to (inclusive) - `<=`: less than or equal to (inclusive)

The rest of the constraints is mainly a semantical syntax sugar and is fully based on this category therefore the forecoming constraints are explained using these operators.

Hyphens

A hyphen-separated range is an equivalent to defining a lower and an upper bound for a range of acceptable versions.

- `1.2.3-4.5.6` is equivalent to `>=1.2.3, <=4.5.6`
- `1.2-4.5` is equivalent to `>=1.2, <=4.5`

Wildcards

There are 3 wildcard characters: `x`, `X` and `*`. They are absolutely equivalent to each other: `1.2.*` is the same as `1.2.X`.

- `1.2.x` is equivalent to `>=1.2.0, <1.3.0` (note the non-inclusive range)
- `>=1.2.*` is equivalent to `>=1.2.0` (the wildcard is optional here)

- `*` is equivalent to `>=0.0.0` (one can use `x` and `X` as well)

Tildes

A tilde is a context-dependant operator: it changes the range based on the least significant version component provided.

- `~1.2.3` is equivalent to `>=1.2.3, <1.3.0`
- `~1.2` is equivalent to `>=1.2, <1.3`
- `~1` is equivalent to `>=1, <2`

Carets

Carets pin the major version to a specific branch.

- `^1.2.3` is equivalent to `>=1.2.3, <2.0.0`
- `^1.2` is equivalent to `>=1.2, <2.0`

A caret-defined constraint is a handy way to say: give me the latest non-breaking version.

7.1.2 Release

A *Release* contains all the information required for Shipper to run a particular version of an application.

To aid both the human and other users in finding resources related to a particular *Release* object, the following labels are expected to be present in a newly created *Release* and propagated to all of its related objects (both in the **management** and **application** clusters):

shipper-app The name of the *Application* object owning the *Release*.

shipper-release The name of the *Release* object.

Example

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: Release
metadata:
  name: reviews-api-deadbeef-1
spec:
  targetStep: 2
  environment:
    chart:
      name: reviews-api
      version: 0.0.1
      repoUrl: https://charts.example.com
    clusterRequirements:
      capabilities:
      - gpu
      - high-memory-nodes
      regions:
      - name: us-east1
    strategy:
      steps:
```

(continues on next page)

(continued from previous page)

```
      - name: staging
        capacity:
          incumbent: 100
          contender: 1
        traffic:
          incumbent: 100
          contender: 0
      - name: canary
        capacity:
          incumbent: 10
          contender: 90
        traffic:
          incumbent: 10
          contender: 90
      - name: full on
        capacity:
          incumbent: 0
          contender: 100
        traffic:
          incumbent: 0
          contender: 100
    values:
      replicaCount: 2
status:
  achievedStep:
    name: full on
    step: 2
  conditions:
  - lastTransitionTime: 2018-12-06T13:43:15Z
    status: "True"
    type: Complete
  - lastTransitionTime: 2018-12-06T12:43:09Z
    status: "True"
    type: Scheduled
  strategy:
    conditions:
    - lastTransitionTime: 2018-12-06T17:48:41Z
      status: "True"
      step: 2
      type: ContenderAchievedCapacity
    - lastTransitionTime: 2018-12-06T12:43:46Z
      status: "True"
      step: 2
      type: ContenderAchievedInstallation
    - lastTransitionTime: 2018-12-06T13:42:15Z
      status: "True"
      step: 2
      type: ContenderAchievedTraffic
    - lastTransitionTime: 2018-12-06T13:43:15Z
      status: "True"
      step: 2
      type: IncumbentAchievedCapacity
    - lastTransitionTime: 2018-12-06T13:42:45Z
      status: "True"
      step: 2
      type: IncumbentAchievedTraffic
    state:
```

(continues on next page)

(continued from previous page)

```
      waitingForCapacity: "False"
      waitingForCommand: "False"
      waitingForInstallation: "False"
      waitingForTraffic: "False"
```

Spec

`.spec.targetStep`

targetStep defines which strategy step this *Release* should be trying to complete. It is the primary interface for users to advance or retreat a given rollout.

`.spec.environment`

The **environment** contains all the information required for an application to be deployed with Shipper.

Important: *Roll-forwards* and *roll-backs* have no difference from Shipper's perspective, so a roll-back can be performed simply by replacing an Application's `.spec.template` field with the `.spec.environment` field of the Release you want to roll-back to.

`.spec.environment.chart`

```
    chart:
      name: reviews-api
      version: 0.0.1
      repoUrl: https://charts.example.com
```

The environment **chart** key defines the Helm Chart that contains the Kubernetes object templates for this *Release*. `name`, `version`, and `repoUrl` are all required. `repoUrl` is the Helm Chart repository that Shipper should download the chart from.

Note: Shipper will cache this chart version internally after fetching it, just like `pullPolicy: IfNotPresent` for Docker images in Kubernetes. This protects against chart repository outages. However, it means that if you need to change your chart, you need to tag it with a different version.

`.spec.environment.clusterRequirements`

```
    clusterRequirements:
      capabilities:
      - gpu
      - high-memory-nodes
      regions:
      - name: us-east1
```

The environment **clusterRequirements** key specifies what kinds of clusters this *Release* can be scheduled to. It is required.

`clusterRequirements.capabilities` is a list of capability names this *Release* requires. They should match capabilities specified in *Cluster* objects exactly. This may be left empty if the *Release* has no required capabilities.

`clusterRequirements.regions` is a list of regions this *Release* must run in. It is required.

`.spec.environment.strategy`

```yaml
  strategy:
    steps:
    - name: staging
      capacity:
        incumbent: 100
        contender: 1
      traffic:
        incumbent: 100
        contender: 0
    - name: canary
      capacity:
        incumbent: 10
        contender: 90
      traffic:
        incumbent: 10
        contender: 90
    - name: full on
      capacity:
        incumbent: 0
        contender: 100
      traffic:
        incumbent: 0
        contender: 100
```

The environment **strategy** is a required field that specifies the rollout strategy to be used when deploying the *Release*.

`.spec.environment.strategy.steps` contains a list of steps that must be executed in order to complete a release. A step should have the follwing keys:

Key	Description
`.name`	The step name, meant for human users. For example, `staging`, `canary` or `full on`.
`.capacity.incumbent`	The percentage of replicas, from the total number of required replicas the **incumbent Release** (previous release) should have at this step.
`.capacity.contender`	The percentage of replicas, from the total number of required replicas the **contender Release** (latest release) should have at this step.
`.traffic.incumbent`	The weight the **incumbent Release** has when load balancing traffic through all Release objects of the given Application.
`.traffic.contender`	The weight the **contender Release** has when load balancing traffic through all Release objects of the given Application.

`.spec.environment.values`

The environment **values** key provides parameters for the Helm Chart templates. It is exactly equivalent to a `values.yaml` file provided to the `helm install -f values.yaml` invocation. Like `values.yaml` it is technically optional, but almost all rollouts are likely to include some dynamic values for the chart, like the image tag.

Almost all Charts will expect some **values** like `replicaCount`, `image.repository`, and `image.tag`.

Status

`.status.achievedStep`

achievedStep indicates which strategy step was most recently completed.

`.status.conditions`

All conditions contain five fields: `lastTransitionTime`, `status`, `type`, `reason`, and `message`. Typically `reason` and `message` are omitted in the expected case, and populated in the error or unexpected case.

`type: Blocked`

This condition indicates whether a *Release* is blocked by a *rollout block* or not.

`type: Complete`

This condition indicates whether a *Release* has finished its strategy, and should be considered complete.

`type: Scheduled`

This condition indicates whether the `clusterRequirements` were satisfied and a concrete set of clusters selected for this *Release*.

`type: StrategyExecuted`

This condition indicates whether a *Release* has achieved a strategy step. This means the installation, capacity and traffic specified in the *.spec.environment.strategy* step were achieved.

`.status.strategy`

This section contains information on the progression of the strategy.

`.status.strategy.conditions`

These conditions represent the precise state of the strategy: for each of the **incumbent** and **contender**, whether they have converged on the state defined by the given strategy step.

.status.strategy.state

The **state** keys are intended to make it easier to interpret the strategy conditions by summarizing into a high level conclusion: what is Shipper waiting for right now? If it is `waitingForCommand: "True"` then the rollout is awaiting a change to `.spec.targetStep` to proceed. If any other key is `True`, then Shipper is still working to achieve the desired state.

7.2 Low-level APIs

These objects represent low-level commands defining the state of specific clusters, as well as the current status of those commands. Together they provide 'just enough federation' to implement Shipper's rollout strategies.

They depend on an associated *Release* object to work correctly: they cannot be created in isolation.

7.2.1 Installation Target

An *InstallationTarget* describes the concrete set of clusters where the release should be installed. It is created by the Release Controller's Scheduler after the concrete clusters are picked using `clusterRequirements`.

The Installation Controller acts on InstallationTarget objects by getting the chart, values, and sidecars from the associated Release object, rendering the chart per-cluster, and inserting those objects into each target cluster. Where applicable, these objects are always created with 0 replicas.

It updates the `status` resource to indicate progress for each target cluster.

Example

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: InstallationTarget
metadata:
  name: api-3f498d25-0
  namespace: service-directory
spec:
  clusters:
  - kube-us-east1-a
  - kube-eu-west2-b
status:
  clusters:
  - conditions:
    - lastTransitionTime: 2018-12-06T16:53:24Z
      status: "True"
      type: Operational
    - lastTransitionTime: 2018-12-06T16:53:24Z
      status: "True"
      type: Ready
    name: kube-us-east1-a
    status: Installed
  - conditions:
    - lastTransitionTime: 2018-12-06T16:53:24Z
      status: "True"
      type: Operational
    - lastTransitionTime: 2018-12-06T16:53:24Z
      status: "True"
```

(continues on next page)

(continued from previous page)

```
      type: Ready
    name: kube-eu-west2-b
    status: Installed
```

Spec

.spec.clusters

The `clusters` field is a list of cluster names *known to Shipper* where the associated *Release* should be installed. Installation means rendering all the objects in the Chart and inserting them into the cluster.

```
spec:
  clusters:
  - kube-us-east1-a
  - kube-eu-west2-b
```

Status

.status.clusters

`.status.clusters` is a list of objects representing the installation status of all clusters where the associated Release objects must be installed.

```
status:
  clusters:
  - conditions:
    - lastTransitionTime: 2018-12-06T16:53:24Z
      status: "True"
      type: Operational
    - lastTransitionTime: 2018-12-06T16:53:24Z
      status: "True"
      type: Ready
    name: kube-us-east1-a
    status: Installed
  - conditions:
    - lastTransitionTime: 2018-12-06T16:53:24Z
      status: "True"
      type: Operational
    - lastTransitionTime: 2018-12-06T16:53:24Z
      status: "True"
      type: Ready
    name: kube-eu-west2-b
    status: Installed
```

The following table displays the keys a cluster status entry should have:

Key	Description
name	The Application Cluster name. For example, **kube-us-east1-a**.
status	**Failed** in case of failure, or **Installed** in case of success.
message	A message describing the reason Shipper decided that it has failed.
conditions	A list of all conditions observed for this particular Application Cluster.

.status.clusters.conditions

The following table displays the different conditions statuses and reasons reported in the *InstallationTarget* object for the **Operational** condition type:

Type	Status	Reason	Description
Operational	True	N/A	Cluster is reachable, and seems to be operational.
Operational	False	TargetClusterClientError	There is a problem contacting the Application Cluster; Shipper either doesn't know about this Application Cluster, or there is another issue when accessing the Application Cluster. Details can be found in the `.message` field.
Operational	False	ServerError	Some error has happened Shipper couldn't classify. Details can be found in the `.message` field.

The following table displays the different conditions statuses and reasons reported in the *InstallationTarget* object for the **Ready** condition type:

Type	Status	Reason	Description
Ready	True	N/A	Indicates that Kubernetes has achieved the desired state related to the *InstallationTarget* object.
Ready	False	ServerError	Shipper could not either create an object in the Application Cluster, or an error occurred when trying to fetch an object from the Application Cluster. Details can be found in the `.message` field.
Ready	False	ChartError	There was an issue while processing a Helm Chart, such as invalid templates being used as input, or rendered templates that do not match any known Kubernetes object. Details can be found in the `.message` field.
Ready	False	ClientError	Shipper couldn't create a resource client to process a particular rendered object. Details can be found in the `.message` field.
Ready	False	UnknownError	Some error Shipper couldn't classify has happened. Details can be found in the `.message` field.

7.2.2 Capacity Target

A *CapacityTarget* is the interface used by the Release Controller to change the target number of replicas for an application in a set of clusters. It is acted upon by the Capacity Controller.

The `status` resource includes status per-cluster so that the Release Controller can determine when the Capacity Controller is complete and it can move to the traffic step.

Example

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: CapacityTarget
metadata:
  name: reviewsapi-deadbeef-0
  namespace: reviewsapi
  annotations:
    "shipper.booking.com/v1/finalReplicaCount": 10
  labels:
    release: reviewsapi-4
spec:
  clusters:
  - name: kube-us-east1-a
    percent: 10
  - name: kube-eu-west2-b
    percent: 10
status:
  clusters:
```

(continues on next page)

(continued from previous page)

```yaml
  - name: kube-us-east1-a
    availableReplicas: 1
    achievedPercent: 10
  - name: kube-eu-west2-b
    availableReplicas: 1
    achievedPercent: 10
    sadPods:
    - name: reviewsapi-deadbeef-0-cafebabe
      phase: Terminated
      containers:
      - name: app
        status: CrashLoopBackOff
      condition:
        type: Ready
        status: False
        reason: ContainersNotReady
        message: "unready containers [app]"
```

Spec

.spec.clusters

`clusters` is a list of clusters the associated *Release* object is present in. Each item in the list has a `name`, which should map to a *Cluster* object, and a `percent`. `percent` declares how much capacity the *Release* should have in this cluster relative to the final replica count. For example, if the final replica count is 10 and the `percent` is 50, the Deployment object for this *Release* will be patched to have 5 pods.

```yaml
    release: reviewsapi-4
spec:
  clusters:
  - name: kube-us-east1-a
    percent: 10
  - name: kube-eu-west2-b
```

Status

.status.clusters

`.status.clusters` is a list of objects representing the capacity status of all clusters where the associated Release objects must be installed.

```yaml
    percent: 10
status:
  clusters:
  - name: kube-us-east1-a
    availableReplicas: 1
    achievedPercent: 10
  - name: kube-eu-west2-b
    availableReplicas: 1
    achievedPercent: 10
    sadPods:
    - name: reviewsapi-deadbeef-0-cafebabe
```

(continues on next page)

(continued from previous page)

```yaml
    phase: Terminated
    containers:
    - name: app
      status: CrashLoopBackOff
    condition:
      type: Ready
      status: False
      reason: ContainersNotReady
      message: "unready containers [app]"
```

The following table displays the keys a cluster status entry should have:

Key	Description
name	The Application Cluster name. For example, **kube-us-east1-a**.
availableReplicas	The number of pods that have successfully started up
achievedPercent	What percentage of the final replica count does **availableReplicas** represent.
sadPods	Pod Statuses for up to 5 Pods which are not yet Ready.
conditions	A list of all conditions observed for this particular Application Cluster.

`.status.clusters.conditions`

The following table displays the different conditions statuses and reasons reported in the *CapacityTarget* object for the **Operational** condition type:

Type	Status	Reason	Description
Operational	True	N/A	Cluster is reachable, and seems to be operational.
Operational	False	ServerError	Some error has happened Shipper couldn't classify. Details can be found in the `.message` field.

The following table displays the different conditions statuses and reasons reported in the *CapacityTarget* object for the **Ready** condition type:

Type	Status	Reason	Description
Ready	True	N/A	The correct number of pods are running and all of them are Ready.
Ready	False	WrongPodCount	This cluster has not yet achieved the desired number of pods.
Ready	False	PodsNotReady	The cluster has the desired number of pods, but not all of them are Ready.
Ready	False	MissingDeployment	Shipper could not find the Deployment object that it expects to be able to adjust capacity on. See `message` for more details.

7.2.3 Traffic Target

A *TrafficTarget* is an interface to a method of shifting traffic between different *Releases* based on weight. This may be implemented in a number of ways: pod labels and Service objects, service mesh manipulation, or something else. For the moment only vanilla Kubernetes traffic shifting is supported: pod labels and Service objects.

It is manipulated by the Release Controller as part of executing a release strategy.

Example

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: TrafficTarget
metadata:
  name: reviewsapi-deadbeaf-0
  namespace: reviewsapi
spec:
  clusters:
  - name: kube-us-east1-a
    weight: 30
  - name: kube-eu-west2-b
    weight: 30
status:
  clusters:
  - achievedTraffic: 100
    conditions:
    - lastTransitionTime: 2018-12-06T12:43:09Z
      status: "True"
      type: Operational
    - lastTransitionTime: 2018-12-06T12:43:09Z
      status: "True"
      type: Ready
    name: kube-us-east1-a
    status: Synced
  - achievedTraffic: 100
    conditions:
    - lastTransitionTime: 2018-12-06T12:43:09Z
      status: "True"
      type: Operational
    - lastTransitionTime: 2018-12-06T12:43:09Z
      status: "True"
      type: Ready
    name: kube-eu-west2-b
    status: Synced
```

Spec

`.spec.clusters`

```yaml
spec:
  clusters:
  - name: kube-us-east1-a
    weight: 30
  - name: kube-eu-west2-b
    weight: 30
```

`clusters` is a list of cluster entries and the desired traffic weight for this *Release* in that cluster. The Traffic controller calculates the correct traffic ratio for this *Release* by summing weights from all *TrafficTarget* objects available.

Status

.status.clusters

`.status.clusters` is a list of objects representing the traffic status of all clusters where the associated Release objects must be installed.

```yaml
status:
  clusters:
  - achievedTraffic: 100
    conditions:
    - lastTransitionTime: 2018-12-06T12:43:09Z
      status: "True"
      type: Operational
    - lastTransitionTime: 2018-12-06T12:43:09Z
      status: "True"
      type: Ready
    name: kube-us-east1-a
    status: Synced
  - achievedTraffic: 100
    conditions:
    - lastTransitionTime: 2018-12-06T12:43:09Z
      status: "True"
      type: Operational
    - lastTransitionTime: 2018-12-06T12:43:09Z
      status: "True"
      type: Ready
    name: kube-eu-west2-b
    status: Synced
```

The following table displays the keys a cluster status entry should have:

Key	Description
name	The Application Cluster name. For example, **kube-us-east1-a**.
status	**Failed** in case of failure, or **Synced** in case of success.
achievedTraffic	The traffic weight achieved by Shipper for this cluster.
conditions	A list of all conditions observed for this particular Application Cluster.

.status.clusters.conditions

The following table displays the different conditions statuses and reasons reported in the *TrafficTarget* object for the **Operational** condition type:

Type	Status	Reason	Description
Operational	True	*N/A*	Cluster is reachable, and seems to be operational.
Operational	False	ServerError	There is a problem contacting the Application Cluster; Shipper either doesn't know about this Application Cluster, or there is another issue when accessing the Application Cluster. Details can be found in the `.message` field.

The following table displays the different conditions statuses and reasons reported in the *TrafficTarget* object for the **Ready** condition type:

Type	Status	Reason	Description
Ready	True	N/A	The desired traffic weight has been successfully achieved.
Ready	False	MissingService	Shipper could not find a Service object to use for traffic shifting. Check `message` for more details.
Ready	False	ServerError	Shipper got an error status code while calling the Kubernetes API of the Application Cluster. Details in the `.message` field.
Ready	False	ClientError	Shipper couldn't create a resource client to process a particular rendered object. Details can be found in the `.message` field.
Ready	False	InternalError	Something went wrong with the math that Shipper does to calculate the desired number of pods. See the `.message` field for the exact error.
Ready	False	UnknownError	Something Shipper couldn't classify has happened. Details can be found in the `.message` field.

7.3 Administrator APIs

These objects represent internal details of a Shipper installation. They expose tools for administrators to configure Shipper or change how Shipper works for application developers.

7.3.1 Cluster

A *Cluster* object represents a Kubernetes cluster that Shipper can deploy to. It is an **administrative** interface.

They serve two purposes:

- Enable Shipper to connect to the cluster to manage it
- Enable administrators to influence how *Releases* are scheduled to this cluster.

The second point allows administrators to perform tasks like load balancing workloads between clusters, shift workloads from one cluster to another, or drain clusters for risky maintenance. For examples of these tasks, see the *administrator's guide*.

Example

```yaml
apiVersion: shipper.booking.com/v1alpha1
kind: Cluster
metadata:
  name: kube-us-east1-a
spec:
  apiMaster: https://10.0.0.1
  capabilities:
  - gpu
  - ssd
  - high-memory-nodes
  region: us-east1
  scheduler:
    unschedulable: false
    weight: 100
```

Spec

`.spec.apiMaster`

`apiMaster` is the URL of the Kubernetes cluster API server. Shipper uses this to connect to the cluster to manage it. This is the same URL as in a `~/.kube/config` for enabling `kubectl` commands.

`.spec.capabilities`

`capabilities[]` is a required field that lists the capabilities the cluster has. Capabilities are arbitrary tags that can be used by Application objects to select clusters while rolling out. For example, one Kubernetes cluster might have nodes provisioned with GPUs for video encoding. Adding 'gpu' as a Cluster capability will allow application developers to specify 'gpu' in their set of Application `clusterRequirements` if their application needs access to that feature.

`.spec.region`

`region` is a required field that specifies the region the cluster belongs to.

`.spec.scheduler`

`scheduler.unschedulable` is an optional field that causes clusters to be ignored during rollout cluster selection. This allows operators to mark clusters to be drained. Default: `false`.

`scheduler.weight` is an optional field that assigns a weight to the cluster. The weight influences the priority of the cluster during rollout cluster selection. Default: `100`.

`scheduler.identity` is an optional field that assigns an identity to the cluster different than its `.metadata.name` value. This allows operators to make one cluster 'impersonate' another in order to transfer all of the Applications on one cluster to another specific cluster. Default: `.metadata.name`.

More information on how to use these fields to manage a fleet of clusters can be found in the *Administrator's guide*.

Status

Cluster objects do not currently have a meaningful `.status` field.

Index

Symbols

C