Reckoner Gitops Functionality Design Doc

TABLE OF CONTENTS

Overview

Reckoner historically has functioned as a wrapper around helm to centralize all installed releases in a cluster and the custom values passed to each chart in a convenient "course" file. It is the Infrastructure as Code for our helm installations. Over time it has become clear that there are some issues with the predictability of changes that helm will make. A lot of this is fixed with the implementation of `reckoner diff` but the output there can sometimes be difficult to parse as a human that needs to review these changes.

Enter GitOps. GitOps isn't necessarily a silver bullet, but it does include a lot more safety when it is implemented well. One implementation strategy is to have all Kubernetes resources stored in git in their flat yaml manifest format such that any PR to these resources will make it very clear what will change once they are applied. This is the strategy that we would like to implement, and Reckoner will be the tool to convert helm charts to flat yaml files suitable for this workflow.

Context

Currently we support running `reckoner template` which is a not-so-fancy wrapper around `helm template` . This gets us part of the way there as it will output the generated

yaml manifests, but we want to support specific GitOps tools, like ArgoCD and potentially Flux 2 by also outputting CRD resources that those tools make use of. The other issue with `reckoner template` is that, given a single chart it will output all the resources as one large string of yaml to standard out and not into separate files, ideally one yaml file per resource that will get created. Not only will our solution need to output to files, but the files will need to be in an opinionated directory structure relative to the location of the course file.

Goals

- ArgoCD is the first and foremost implementation we should focus on. FluxCD will not covered by this design doc.
- The output of yaml should have 1 resource per yaml file that have a comment at the top signifying that they were auto-generated by reckoner and should not be edited.
- The first implementation to be completed should be `argocd`. So, when the ArgoCD milestone is complete, a command should look like this: `reckoner template --gitops argocd course.yml` with optional flags for the destination directory. The default destination directory should be `./manifests`.
- We need to have a solution for secret data that gets passed into helm charts that should not be stored in git. We must assume that with whatever implementation we use in the code, that the secrets will be accessible from the cluster where ArgoCD is running. First implementation should be with Hashicorp `vault`.
- Helm hooks should get converted to ArgoCD hooks (in the Argo implementation).

Non-Goals

- We should not attempt to achieve our GitOps model by still deploying helm charts as helm charts. This means there will not be any releases to look at in a cluster when running `helm ls` when using this new model. All relevant helm chart version information will still be visible in the annotations/labels of the resources. It will likely be easiest to view these things in the argocd frontend UI going forward.
- Similar to the above point, we will not support ArgoCD or Flux Helm release types. We will only support the release pattern of flat yaml manifests as it is the easiest way to see all changes in a git PR.
- We should not attempt to generate yaml based on any system other than `helm`. Reckoner is still a helm tool at its heart. That may change in the future, but for this particular project, we should only focus on generating yaml files based on helm charts.
- We will not strip generated resource annotations/labels that are unique to helm, this will allow a user to go back to helm without deleting all the resources first (helm should

adopt the resources with the proper labels/annotations).

Technical Architecture

- The command structure should be: `reckoner template --gitops argocd` for the initial support of ArgoCD.
- The following helm flags should be assumed when doing the `helm template` part of generating the yaml manifests:

Bash

```
--skip-tests
--include-crds
```

- Originally we thought we would also include the flag `--no-hooks` but if the GitOps tool has a feature analogous to helm hooks (ArgoCD has Hooks too), we should generate and convert to the implementation's style.
- Namespace creation: if the GitOps tool supports namespace creation somehow, use that. Otherwise we may need to generate a namespace resource. ArgoCD has a way to create namespaces via its `Application` resource.
- There will need to be a new `gitops` package in the `pkg` directory that starts with a generic `interface` . This interface should have a receiver function that needs to be satisfied for generating the implementation-specific CRDs needed. For example, in the ArgoCD implementation, there will be a `struct` that correlates to the ArgoCD `Application` CRD. That struct should satisfy this interface by implementing the `Generate()` function.

  EXAMPLE:

Go

```
type GitopsCRD interface {
  Generate()    ([]byte, error) // Generate CRD yaml, e.g. ArgoCD Application
  HandleHooks() ([]byte, error) // Generate yaml for the implementations method of handling helm hooks, e.g. ArgoCD Hooks
}
```

- Within the same `gitops` package we will need to define the files output and directory structure to house the output. The base directory can and should be defined by a flag

to the `gitops` command that also has a default to satisfy our internal needs (default: `./manifests`). It can look something like this (there may be more parameters, this is a rough guess):

Go

```
type Output struct {
  Implementation GitopsCRD
  BaseDir        string
}
```

- There should be helper function(s) in the `gitops` package that dangle off the `Output` struct which call the concrete implementation function `Generate()`. Something like:

Go

```
func (o Output) Generate() (error) {
  //Psuedo code
  err := makeTheDirectoryStructure(o.BaseDir)
  if err != nil {
    return nil, err
  }
  renderedCRDBytes, err := o.Implementation.Generate()
  if err != nil {
    return nil, err
  }
  // Psuedo code to generate yaml manifests which
  generateYamlManifests() // This spot will call helm template and maybe do something unique with the templated values if necessary. Also create the specific tool CRD object yaml (e.g. Argo Application resource).
}
```

- Deletions:
  - There should be a function in the `gitops` package that scans for already existing generated files and removes them all when re-generating
    - Add a comment at the top of all generated files (something like `# Generated from Reckoner, do not manually edit`) and the function in the code should use this comment as a way to find the files to remove.

    - We want to allow people to add files to the directory structure we set up without removing them. As long as they don't put the exact same comment at the top of the file, this should work fine.
- On the topic of splitting yaml and commenting at the top: As mentioned in the goals section, we should be splitting up the helm template output such that there is one resource yaml manifest per file. The splitting up into files should be fairly simple to do with a loop over the yaml decode. Here is an example that loops over all yaml manifests in helm template output from pluto. The trickier part will be adding the comment at the top (maybe it won't be but there is not technical suggestion on how to accomplish that currently).

Known Unknowns

- Secrets
    - A lot of helm charts follow the pattern of `existingSecretName` and then you just enter the secret and assume it's in the namespace. We should not care about creating the Secret objects, but allow users to drop flat yaml into the manifests directory structure that is NOT generated by reckoner and have it survive regeneration. This way people can handle secrets the way they want in git.
    - Secrets that are generated by a helm chart need some special work because most of the time these should not be passed into git as-is and there should be a plugin involved. This is related to the below bullet point.
        - Is the answer that we output a warning message if something *looks like* secret data and not solved via argo plugin?
    - If secret data is passed in via values, this should only need to be documented because the value data should be in the pattern of the chosen plugin. For example if we pass secret data into a chart via value and we are using the vault plugin, the value passed in can simply be in the pattern `<path:vault/path#vault_secret_key#optional_secret_version>` . If this is done, there is nothing we need to do in the code.
- Helm hooks when not using ArgoCD
    - ArgoCD has the concept of hooks that we can convert helm hooks to (we think), but that is unknown when we add another implementation like FluxCD
- Reckoner specific metadata on generated yaml manifests
    - We should explore how difficult it would be to add reckoner labels/annotations to the resources that we put in the cluster. These would live alongside the annotations/labels that helm adds.
- Handling charts that use the `Capabilities` functionality of helm templating. We need to figure out how to reliably pass in the API versions that exist in a cluster, or let the user configure that. Related to this Issue.

- Handling list items in a helm chart. While rare, it's possible that a helm chart can contain `kind: List` items. Example from archived prometheus-operator chart.

MVP Milestone

At the end of this we should have a `reckoner template --gitops argocd` command that has the ability to create yaml manifests as well as ArgoCD `Application` yaml suitable for deployment via gitops + ArgoCD.