1. Abstract

hej med dig

- The tool can be found here: https://github.com/dag-andersen/kubernetes-reference-highlighter
- and downloaded here: https://github.com/dag-andersen/kubernetes-reference-highlighter/releases
- or downloaded from the VS Code Marketplace.

2. CCS CONCEPTS

- Human-centered computing → Open source software; Software Engineering Tooling → Code Validation

3. Keywords

- Open Source Software, IDE Extension, Code Linting, Static code analysis, Code Validation, Kubernetes

4. Introduction

Copiled language vs. interpreted languages.

Static analysis

Lack of information | external resources
You may not have all resources locally in the same directory.
That mean we need to look outside boarders of the IDE.

This ofc only works if you have read access to those endpoints/resources from you local machine. If you the endpoints is only visible from inside an enclosed envorihemt this will not work.

Magic strings

Terraform does static "compile" time checks, so you will be notified about your broken references beforehand.

Kubernetes does not do that. Kubernetes resources are not meant to be created at the same time or in a given order, so we cant talk about a "compile" time. Often in kubernetes your references will not exist on creation time, but will eventually be created.

Since the your texteditor usually does not verify any of the magic strings, it can quite cumbersome to debug the code. It tend to involve manually reading the magic string over and over until you realize that there is spelling mistake, or each resource exists in two different namespaces. No public and free IDE feature or extension exists that checks this.

Developers often use tools like `Helm` and `Kustoimze` to template their YAML-files. This is done so multiple configuration can inherent/share common code across different configuration. This means that references that is written directly in the files may not exist in any of the files, but may only be generated on `runtime` when one of the templating tools are used. This makes it much harder to give valuable information to the developer because `object` names are dynamically generated. I have not found any extension that tries to tackle this challenge, so currently developer doesn't get any help validating their YAML-files live while coding when using templating tools like Helm and Kustomize.

This paper will try to see if live verification of magic strings in yaml can reduce the time needed to debug a configuration in kubernetes. A tool will be developed and tested on developers.
I will refer to the tool `Kubernetes Reference Highlighter` with KRH
10 developers are given a task to debug a system. half of them will do it with the extension enabled and the others wont. The time it takes for each developer will tracked and evaluated upon.

5. Related tools | Existing tools

In this section i will list related tools that aim to help the developer create correct/valid YAML.

- **JetBrains IntelliJ IDEA Kubernetes Plugin** [link]: The full version IntelliJ that includes support for plugins is a paid product and therefore is not accessible publicly. JetBrains' only takes local files into account when highlighting references. The plugin does not check your current cluster or

build anything with kustomize like KRH. But JetBrains' plugin comes with extra functionality that lets the user jump between files by clicking the highlighted references (something that KRH does not).

- **KubeLinter** [docs] [GitHub]: This is a open source Command Line Tool that validates your kubernetes manifests. KubeLinter is feature-rich tool that informs and warns of all sorts of issue that may be their code. The full list of features can be found [Here]. KubeLinter is ideal to integrate in testing-pipelines where it can function as an initial step for everything YAML before it goes into production. KubeLinter can validate Helm Charts, but can not validate Kustomize templates. The tool only check the local files it is provided, so it does not know what already exists in a cluster.
- **kube-score** [GitHub] [Validation Checks]: kube-score is a tool that performs static code analysis of your Kubernetes object definitions. Even though there is over 30 different checks the value are quite limited. The tool seems to only validate at all the kubernetes object definition individually, and not as a whole (with e.g. reference checking). Furthermore, just like *KubeLinter* it is a Command Line tool and therefore can be integrated as part of a testing-pipeline, and does not provide any live feedback when coding in an IDE.
- **Kubevious** [docs] [GitHub] . This tool comes with 44 built in Kubernetes Resource validations checks. The checks include cross-file checks that e.g validate if a `service` points to two deployments at the same time and that sort of misconfiguration. Kubevious has a powerful toolset just like KubeLinter, but this tools has a UI and needs to be run in a cluster or as a stand alone program.
- **Conftest** [GitHub], **Copper** [GitHub], and **Config-lint** [GitHub] are all similar tools that try to validate Kubernetes Manifests/YAML, but none of them come with built-in checks, so the user have to make their own "rules" that the YAML will be validated against. All these tools are Command Line tools, so they do not provide any live feedback but only validation when command run. These tools accellerate when you want to build you own custom checks (maybe on you own Custom Resource Definition (CRDs)).

6. Prototype | Implementation

6.1. Overview

The tools is build as an Visual Studio Code (VS Code) extension, since it is free and is the most popular IDE/Editor in the industry [source]. The extension is made in typescript, since that is the default for developing VS Code Extensions, since *VS Code* is written in typescript.

The extension highlights the name of an object if it finds a reference in the open file. The highlighting updates in real time while you type, giving you constant feedback on if your references is found or not.

> Fig. 1 shows how the extension would highlight the string `svc` in the VS Code because it found an object with that name file `./service.yml`. On this image the VS Code extension, *Error Lens*, is enabled to better visualize the highlighting.

The extension use Regex and YAML-parsing as the main method for reading the files in the workspace.

A tool like this is close to useless if the developer can not rely on validity of the feedback it gives. This is why this tool aims to only highlight a string if it is sure that the reference exists. It is better to have false negatives than false positives.

This extension make use of `vscode.diagnostics` for highlighting strings. Diagnostics are the squiggly line under code lines, and they severity can either be marked as an `Severity.Error`, `Severity.Warning`, `Severity.Information`, or `Severity.Hint`. This extension use the `Severity.Information` for all references, and it only use `Severity.Error`, when `Kustomize build <path>` fails. Since these Diagnostics are built into the VS Code IDE, then other extensions or tools can also read and act upon the infomration provided by this extension. This makes this extension integrate well with other VS Code features/extensions. In particular i suggest installing *Error Lens* together with this extension to visualize the diagnostics even better for the user.

The extensions functionality can be split up into two components: *Reference Collection* and *Reference Detection*. *Reference Collection* is the step that collects and build a list of all the objects the extension have found. This step is triggered every time you save a YAML-file in the workspace.

Reference Detection is the step that parses the current open file and try to find references that matches objects found in the *Reference Collection*-step. This step is triggered every time a YAML-file is changed in the workspace.

> This paper has not looked into if this tool can be extended to also include Helm Chart reference detection. Further research and development is needed to conclude if that is possible.

6.2.1. Reference Detection

As of Version 0.0.2 the extension highlights references to `Services`, `Deployments`, `Secrets`, and `ConfigMaps`. More kinds can easily be added in the future.

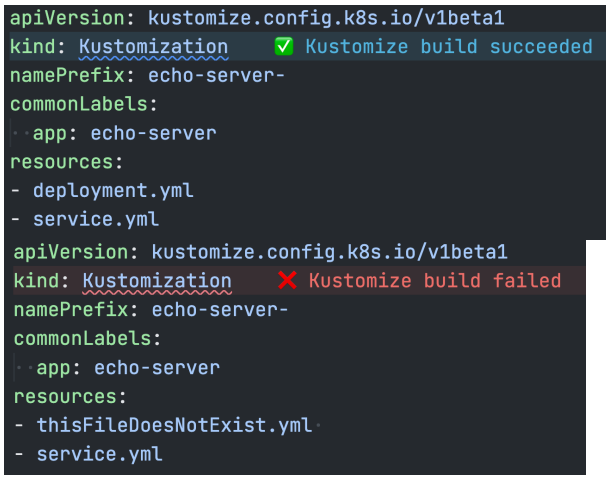
All resources are namespace-sensitive. A reference will not be highlighted if the resource exclusively exists in another namespace and not current open files namespace.

If the string matches with multiple objects the extension will show all the references. e.g the extension will highlight that it found a resource with that name in both the current cluster and in the current editor at the same time.

KubeLinter detects "dangling references", so it inform the user that a references does not exist. This is great, but often times you do not have the whole infrastructure configuration locally, meaning that the dangling references that it detects may not be dangling after all, because the references actually exists in the cluster. This ensures the KubeLinter can report false positives (detecting an issue that is not an issue).

6.2.2. Reference Collection

- **Cluster Scanning**: KRH use the user's current context found in the kube_config to call the Kubernetes Cluster and collects the names of the kubernetes objects that way.
- **Workspace Scanning**: KRH travers all files ending with `.yml`. or `.yaml` in the current open workspace in VS Code. It collects the `kind`, `name`, and `namespace` of all the Kubernetes objects found in the files.
- **Kustomize Scanning**: KRH travers all files named `kustomization.yml`. or `kustomization.yml` in the current open workspace in VS Code, and builds the kustomize files. All the kubernetes objects generated by `kustomize` is collected. In addition to that the extension will highlight if the file builds or not in kustomization file.



Each time you save a YAML-file in VS Code the extension will update its internal list of kubernetes objects.

Each of these scanning techniques can be disabled through the CommandLine or through settings.

7. Evaluation

7.1. Results

7.2. Limitations

8. Conclusion

This paper has

open source
Handles templating
Live feedback
danling references

9. References