GSoC 2019 Kapitan Project Proposal

Task 1: dependency management support

Objective

This feature allows Kapitan to pull from external repositories/sources during compile to store in a particular target directory, as described in the issue[1]. This feature will make Kapitan more versatile in its use and fit for more use cases, increasing the number of users as well as better serving the existing users.

Use

The dependencies and their details should be listed as YAML parameters for each target, in the following format:

```
parameters:
  kapitan:
    dependencies:
      - type: <dependency_type>
        output_path: <relative_path_in_target>
        source: <source_of_dependency>
        … optional, type-specific arguments
```

Supported dependency types

1. git

Use

An example for git-related parameters would look as follows:

```
    - type: git
      output_path: lib
      source: git@github.com:user/repo.git
      subdir: foo
      refs: lib
```

[1] https://github.com/deepmind/kapitan/issues/91

Implementation

As illustrated above, It will optionally support:

- subdirectories in a repository
- Refs, such as branch, tags and commits

We will be using GitPython[2] module for cloning git repositories. The code would look as follows[3]:

```
from git import Repo
git_url = 'https://github.com/deepmind/kapitan.git'
repo_dir = 'path/to/output/dir'
ref = 'master' # can be branch, tag, etc.
Repo.clone_from(git_url, repo_dir, branch=ref)

# to add git clone parameters, we can pass in kwargs as follows
Repo.clone_from(git_url, repo_dir, branch=ref, depth=1)
```

As shown above, to optimize the speed of download, we can use a few options such as **--single-branch** and **--depth**=1[4].

Additionally, we may want to support custom Git executable path, which GitPython already supports, in order to accommodate users with no privilege to install git on their system.

2. HTTP/S

Use

Sample YAML parameters for HTTP[S] would look as follows:

```
    - type: http | https
      output_path: lib
      source: https://raw.githubusercontent.com/...
```

Its sources are:

- Raw files from Github at https://raw.githubusercontent.com
- Helm charts via URL
- Any other files reachable by HTTP[S]

---

[2] https://github.com/gitpython-developers/GitPython
[3] https://github.com/gitpython-developers/GitPython/issues/269
[4] https://stackoverflow.com/questions/1778088/how-do-i-clone-a-single-branch-in-git

Implementation

This is trivial using **requests** module by making an HTTP GET request.

As for Helm charts, the user may specify the URL of the chart, just as they would do for **helm fetch**, as follows:

`<chart_repository_url>/<chart_name>`

For example,

`https://technosophos.github.io/tscharts/alpine-0.1.0.tgz`

All the relevant information on how Helm chart repositories are structured can be found here for reference. On a side note, **helm fetch** command takes in either chart URL or repo/chart as the argument, if the repository mapping is already created locally by **helm repo add**. Therefore, if the user does not know the URL of the repository, they should simply find out by running **helm repo list**. For example,

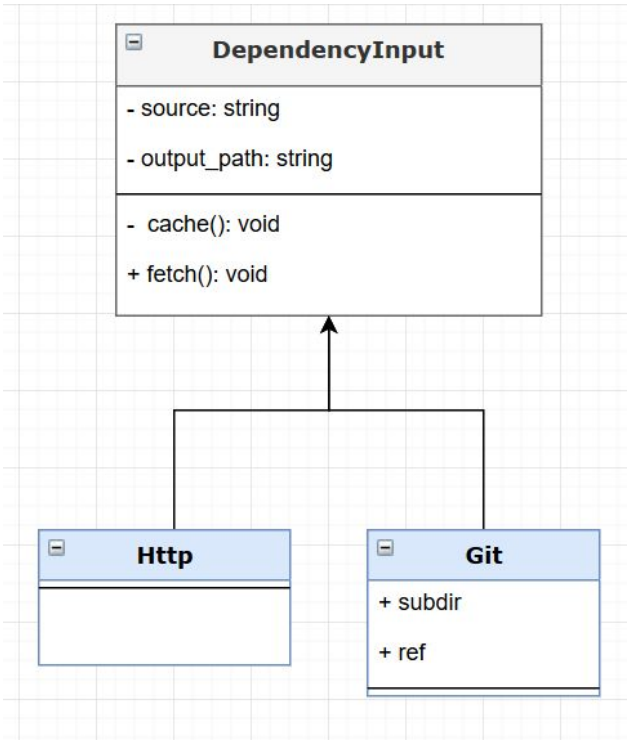
```
$ helm repo add fantastic-charts https://fantastic-charts.storage.googleapis.com
$ helm repo list
fantastic-charts    https://fantastic-charts.storage.googleapis.com
```

Design & overall Implementation

The scripts to support this feature will all be stored under **kapitan/dependencies** subdirectory, which mirrors **kapitan/inputs** subdirectory in structure:

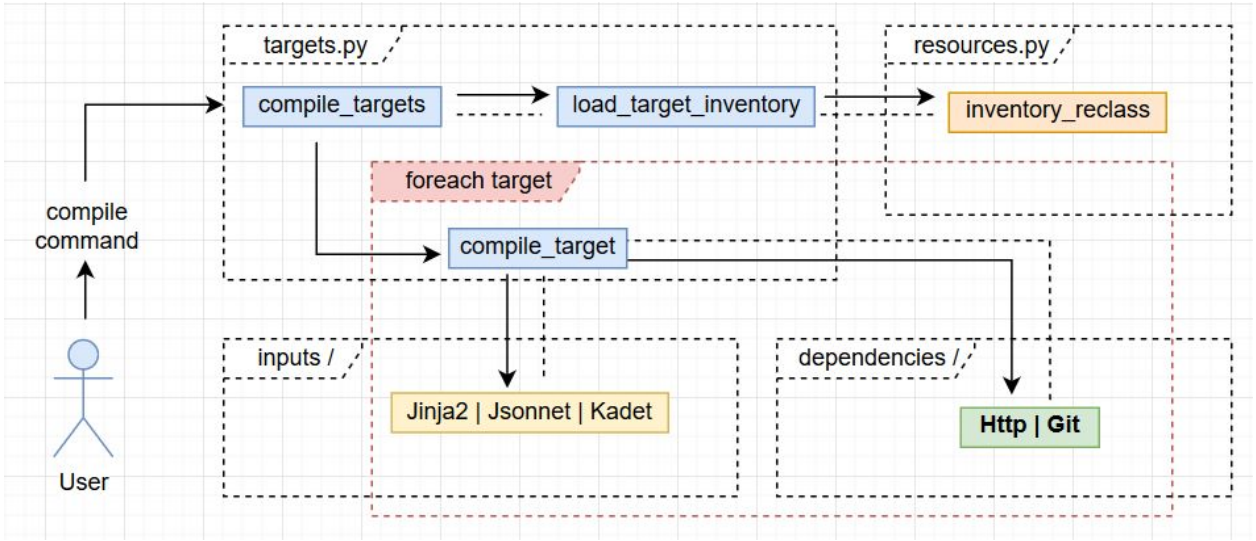
```
kapitan/
   ...
   dependencies/
        __init__.py
       base.py
       http.py
       git.py
```

For all the input types for dependencies, we will make use of inheritance, similar to InputType in **kapitan/inputs,** which looks as follows:



DependencyInput is the superclass of all types of dependency inputs, such as HTTP and Git. This makes it easier to extend the feature by adding more types of dependencies in the future.

The sequence of execution is illustrated below:



Since we are already using multi-threading to compile each target, fetching of the source will be done in parallel (in terms of the target). If a target contains multiple dependencies, we will fetch those dependencies in parallel as well, using multiprocessing module.

Logging and exceptions

It is important to give feedback to the user in case something happened while fetching the source. We should clearly distinguish which error occurred if any, as well as the progress of downloads (at least whether it has started and ended).

If one target fails to compile due to an error in its dependency fetch, the rest of the targets should not proceed to compile either.

Caching

The retrieved files will be cached to **.kapitan** folder in the same way as targets cached by **cached.py.**

On the first compile, dependency files will be cached, after which all the subsequence compile will use the cached files stored locally. We will supply **--fetch** option to **kapitan compile** command for the user to do a fresh fetch of the dependencies.

Testing

Testing will be done using python unittest. For each dependency type:

- Test for proper parsing of the YAML file, whether all the required properties exist
- Test for fetching of files, and whether it is properly stored in output_path
- Test for caching of files
- Test for updating versions in dependencies, and whether new downloads are triggered

and so on. More testing cases are listed in the Timeline section.

Task 2: Templating Helm charts

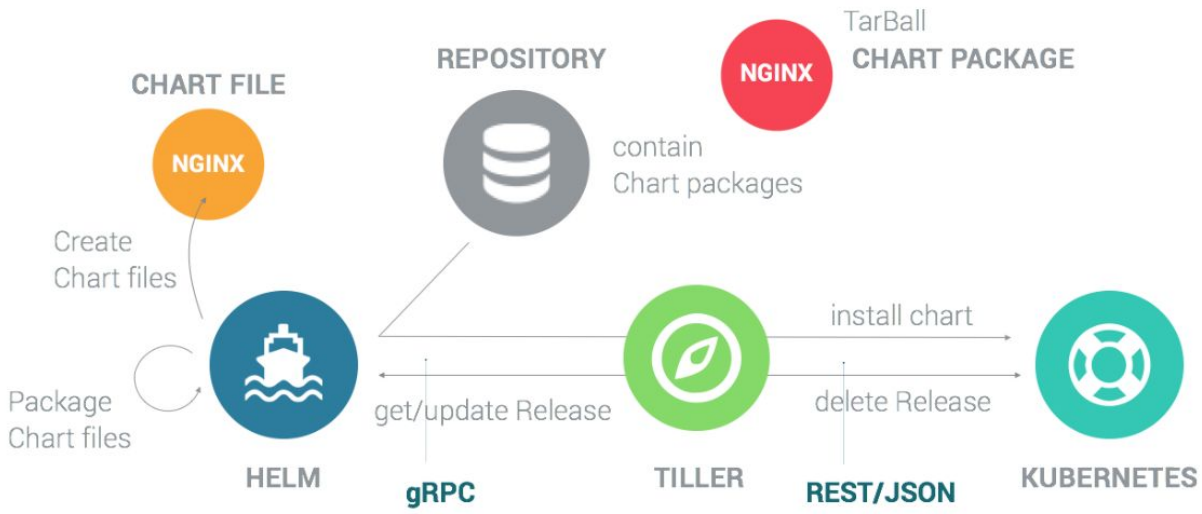
Objective

With HTTP dependency management in place, users will now be able to fetch Helm charts. This task allows users to template those Helm charts via Kapitan.

This will capture the need of the people who are not very satisfied with Helm but cannot easily switch because of the amount of work necessary for migrating to Kapitan templates.

Background on Helm

The overall architecture of Helm looks as follows:



There really are two components to Helm[5], which are:

1. The **Helm client** (CLI)
2. The **Tiller server**, an in-cluster server that does most of the important things like installing chart, tracking releases etc.

Both components are written in Go. While **helm install <chart_name>** requires the Tiller engine to perform the rendering of the templates, Helm also provides **helm template <chart_name>** command that allows users to locally template the charts for manual checking

The limitation of **helm template** is that any values that would normally be looked up or retrieved in-cluster will be faked locally[6]. However, this should not affect the usefulness of this new feature.

Helm charts

Helm charts are files laid out in the following directory structure[7]:

```
mychart/
  Chart.yaml   # info about the chart
  Values.yaml  # default values
  charts/
  templates/   # for templates
  ...
```

[5] https://banzaicloud.com/blog/helm-rest-api/
[6] https://helm.sh/docs/helm/#helm-template
[7] https://helm.sh/docs/chart_template_guide/#charts

The **templates** directory contains the template files, which will be passed to the template engine to render the templates. The output is rendered based on the parameters defined in[8]:

1. values.yaml
2. values.yaml of parent, if the chart is a subchart
3. value file passed to the command using **-f** option
4. values set manually by the command using **--set**

$ **helm template** command

In order to understand how exactly Helm templates charts, we can take a look at the source for **helm template** command[9]. Note that this was previously a third-party plugin, but in newer versions of Helm has been integrated as a built-in command by importing the tool. Therefore, the source is from this repository, and not from that of Helm.

The **run** function below describes how the templating is done, when a chart name is provided as the argument to the command (along with other options).

```
func run(cmd *cobra.Command, args []string) error {
        if len(args) < 1 {
                return errors.New("chart is required")
        }
        c, err := chartutil.Load(args[0])
        if err != nil {
                return err
        }

        vv, err := vals()
        if err != nil {
                return err
        }

        config := &chart.Config{Raw: string(vv), Values: map[string]*chart.Value{}}

        if flagVerbose {
                fmt.Println("---\n# merged values")
```

[8] https://helm.sh/docs/chart_template_guide/#values-files
[9] https://helm.sh/docs/helm/#helm-template

```
                fmt.Println(string(vv))
        }

        options := chartutil.ReleaseOptions{
                Name:      releaseName,
                Time:      timeconv.Now(),
                Namespace: namespace,
                //Revision:  1,
                //IsInstall: true,
        }

        // Set up engine.
        renderer := engine.New()

        vals, err := chartutil.ToRenderValues(c, config, options)
        if err != nil {
                return err
        }

        out, err := renderer.Render(c, vals)
        if err != nil {
                return err
        }
}

func vals() ([]byte, error) {
        base := map[string]interface{}{}

        // User specified a values files via -f/--values
        for _, filePath := range valsFiles {
                currentMap := map[string]interface{}{}
                bytes, err := ioutil.ReadFile(filePath)
                if err != nil {
```

```
                return []byte{}, err
        }

        if err := yaml.Unmarshal(bytes, &currentMap); err != nil {
                return []byte{}, fmt.Errorf("failed to parse %s: %s", filePath, err)
        }
        // Merge with the previous map
        base = mergeValues(base, currentMap)
    }

    // User specified a value via --set
    for _, value := range setVals {
        if err := strvals.ParseInto(value, base); err != nil {
                return []byte{}, fmt.Errorf("failed parsing --set data: %s", err)
        }
    }

    return yaml.Marshal(base)
}
```

There are calls to a few important functions that we should examine closely, namely:

1. chartutil.Load: loads the chart with the given name
2. chartutil.toRenderValues: gets the values to be substituted into the templates according to Helm's priority rules
3. engine.Render: actually renders the template output

If we wish to overwrite some of the values specified in the chart, **helm template** command provides us with **--set** option that takes in a string of key-value mapping of parameters to be overwritten, for example, **--set key=val1,key2=val2**. These values are retrieved in **vals** function above. Therefore, we can use the same mechanism as this from Kapitan to overwrite the values in Helm charts.

According to the Helm documentation[10] and the actual code from the repository, the template engines used in Helm are:

1. Go template engine (text/template)
2. Sprig[11] (a Go library)

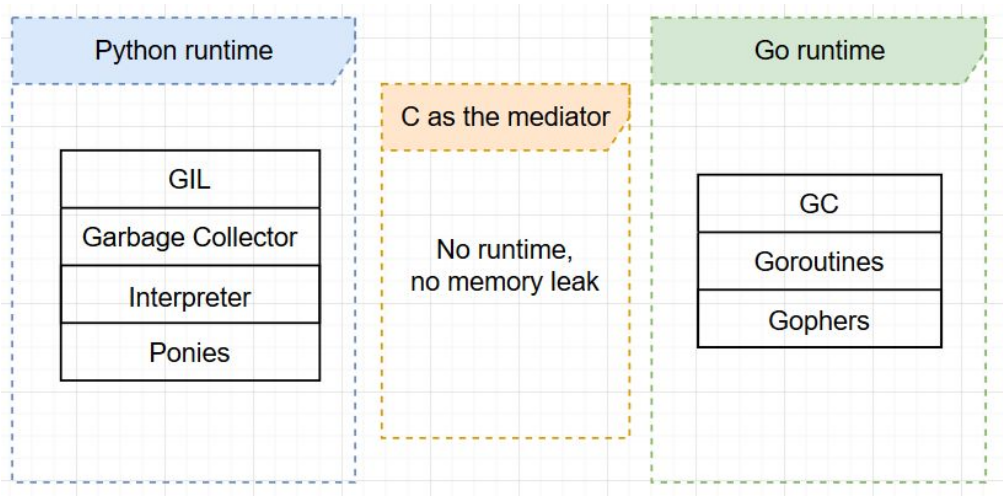
Implementation

We will need to produce python-go binding for Kapitan to template Helm charts. This will be done in two steps:

1. We will write a Go script that exposes an interface to Python, and compile that into .so and .h files.
   This interface MUST BE C-compatible (i.e. inputs and outputs are C-types), because Python talks C, and so does Go, but they can't talk directly with each other.
2. Using **pybindgen**[12], which allows for easy creation of python binding from C/C++ headers, we create Python binding from the header file.

Detour: more on Go-Python binding

If we were to call Go shared library from Python, this would be bridged by C as follows:



Python has its own objects, while Go also has its own as well. We will need to therefore implement code to convert between a Go object and a Python object using C data types.

Another challenge is that we must not share memory between the two runtimes. In order to achieve this, we will need to do things like:

- copy objects/structs in C, and then pass that on between Python and Go runtimes
- Create a simple memory manager in C that [de]references pointers

[10] https://helm.sh/docs/chart_template_guide/#template-functions-and-pipelines
[11] https://github.com/Masterminds/sprig
[12] https://pythonhosted.org/PyBindGen/tutorial.html

As such, if the Go functions to be exposed to Python requires complex inputs and/or outputs, such as a Go interface for callbacks, then it can quickly become complicated to bridge the two using C, such as in this attempt to create a Python library that uses a simple Go web server. Another example is this commit from **go-jsonnet** that creates minimum C-binding.

Fortunately, in our case, we really only need to pass two C strings (assuming we convert from C string to Go string within Go) to the run function, one for the chart name and the other for overwriting values in Helm: and we only need to output error messages, if any. Therefore, the binding should not be complicated. However, if we need to have more control over the process, we can implement some of the functions above in Python such as loading of the chart, and then use the Go shared library that we export to template the chart (slightly more work, but can definitely be done).

More References for building Go-CPython binding:

- https://hackernoon.com/extending-python-3-in-go-78f3a69552ac
- https://blog.filippo.io/building-python-modules-with-go-1-5/#thecompletedemosource
- https://id-rsa.pub/post/go15-calling-go-shared-libs-from-firefox-addon/
- https://github.com/vladimirvivien/go-cshared-examples
- https://medium.com/@liamkelly17/working-with-packed-c-structs-in-cgo-224a0a3b708b
- https://github.com/golang/go/issues/18412
- https://qiita.com/74th/items/0362bea2012ef253c539 [japanese]

Task 3: Creating Kapitan binary

Objective

In addition to the Docker image, having a portable Kapitan binary that “just works” would help increase the use of the tool. We can build a static binary containing all the dependencies we use in Kapitan (Python interpreter itself included as well). The target platforms are Linux x86 & x64.

Choice of tools and implementation

Our requirements for the tool are:

1. Minimal maintenance
2. Speed - the binary is fast to run (possibly achieved by Cythonizing the code first)
3. Space - the binary is not too large
4. Cross-platform: supports Linux x86/x64, and Windows

5. Supports the desired Python versions (>= 3.6)
6. Single-file executable is preferred

What are NOT our requirements:

- Support for native modules (e.g. C libraries)
- Preventing reverse-engineering: it's OSS!

There are a few tools that we can experiment with (non cross-platform solutions such as py2exe and py2app are not considered):

Option 1: **Pyinstaller**[13]

- Supports single-file executables
- > 5100 stars, actively maintained (last commit to master in March 2019)
- Supports Cython

Option 2: **cx_Freeze**[14]

- Does not support single-file executables[15] (but can be overcome with 7zip SFX etc.)
- > 300 stars, actively maintained (last commit to master on September 2018)
- Supports Cython

Option 3: **nuitka**[16]

- More of a compiler than a packager (e.g. it doesn't package Python interpreter, just compiles to C and converts C into executable)
- > 1800 stars, actively maintained (last commit to master in March 2019)

Since the binary distribution is specific to the active operating system and the active version of Python, we need to run Pyinstaller/cx_Freeze on each of the target platforms & versions of Python[17].

To speed up the execution, we should also first Cythonize Python scripts using Cython. This applies to Pyinstaller as well as cx_Freeze, while nuitka is completely different to those two in that it converts scripts into C code and then package it.

---

[13] https://github.com/pyinstaller/pyinstaller
[14] https://github.com/anthony-tuininga/cx_Freeze
[15] https://cx-freeze.readthedocs.io/en/latest/faq.html#single-file-executables
[16] https://github.com/Nuitka/Nuitka
[17] https://pyinstaller.readthedocs.io/en/v3.4/operating-mode.html

Some research into their use has revealed mixed reviews on Pyinstaller, Nuitka and cx_Freeze, including:

- Doesn't work!
- The output is large in file size
- The output runs slow

Therefore, it is wise to keep all options alive and see which one works without error and performs better for kapitan according to our requirements. Our starting choice, however, should be **PyInstaller** since it seems to be well maintained (> 5100 stars, frequent commits & releases) among the three tools, implying good support and decent performance, with support for Cython.

The binary will be platform-specific, so it is wise to containerize the binary production as well as the testing of it. The Docker container images used for binary production are:

1. 32-bit Debian 9
2. 64-bit Debian 9

As for creating binaries for Windows platform, we will use Windows VM for Virtualbox instead of containers, because Windows containers are only available on Windows 10 Pro/Enterprise (or Windows Server 2016) which support Hyper-V. The VM image is found here: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines. The setup of VM will be automated using vagrant. Windows support will be explored should there be time left at the end of the project.

Testing

Whatever the tool we end up using, the python file to build the binary will be included in the repository for testing.

1. Test that desired output is created using **unittest**
2. Run that output locally and test whether it runs successfully (using shell scripting or python subprocess)

Task 4: YAML validation using jsonschema

Objective

As versatile as Kapitan is, its use cases mostly revolve around configuration management for tools such as Kubernetes, Terraform and the like. Therefore, the output of Kapitan compile will

be used against those services, which impose specific requirements in the structure of YAML input. Since we are using jsonnet to convert JSON into YAML, we can make use of json schema to help the user validate the output of compile command. This way, they know the error in the output, if any, before failing to deploy.

Use

User can opt-in for the validation of specific YAML types by specifying the corresponding Kapitan parameter as follows:

```
parameters:
  kapitan:
    validate:
      - output_type: <type of output>
        output_path: <file output path>
        … other type-specific parameters
```

This feature is opt-in, so that users who don't use this feature should be able to continue using Kapitan the same way as before (i.e. backward compatible in terms of user experience).

Supported types

Kubernetes

Overview

Types supported will be based off of the API endpoints for Kubernetes[18]. There is an ongoing work called kubernetes-json-schema, which aims to generate validation schema for different types of Kubernetes configuration, such as Service, Deployment and Pod, for each Kubernetes version.

The way this achieves automated generation of validation schema is by converting the OpenAPI specification of Kubernetes API[19] to json schema, using openapi2jsonschema, which is developed by the same author. OpenAPI/Swagger is a tool that allows APIs to be defined/documented using either JSON or YAML, and since we know Kubernetes depends on this tool, this can be the source of truth for the validation schema.

Use

The following illustrates how the user may opt-in for the validation:

---

[18] https://kubernetes.io/docs/reference/#api-reference

[19] https://github.com/kubernetes/kubernetes/tree/master/api/openapi-spec

```
parameters:
  kapitan:
    validate:
      - output_type: kubernetes.service
        version: 1.6.6
        output_path: <output_path>
```

Without the version specified, it will default to the most recent stable release of kubernetes.

Implementation

We will retrieve raw json files from the above repository for validation, reusing the code for http dependency in Task 1. Assume that the user wants to validate the json output for Kubernetes v1.6.6 deployment configuration. Then, we can retrieve the file from the following URL:

https://raw.githubusercontent.com/garethr/kubernetes-json-schema/master/v1.6.6-standalone/deployment.json

caching

Such files are actually rather large (up to 5MB), and therefore will also be cached locally, so that from second time onwards we can speed up the validation step. It is also possible to ship Kapitan with a few of the most commonly-used kubernetes validation schemas, if desirable.

Terraform

Supporting terraform is somewhat different to supporting Kubernetes as far as I have researched. I was using **terraform validate** command as a starting point, but it seems like the configuration is user-generated and there is no particular standard used for validation. This issue will be explored with any remaining time for the project.

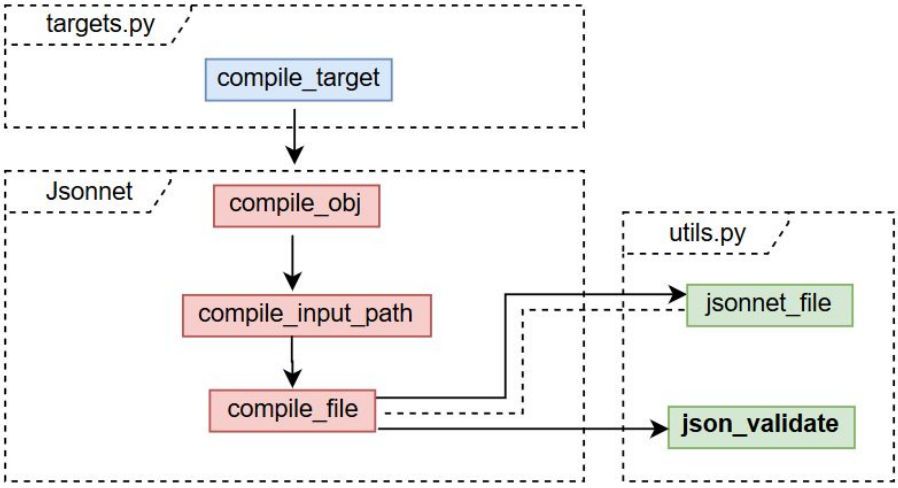
Design & overall Implementation

We will parse the YAML validate parameters as specified above, and validate against the schema before writing jsonnet output to a file. The inputs for the validation step are:

1. **The validator json**:
    - e.g. kubernetes_service.json
2. **The json output rendered by jsonnet**:
    - the JSON output is the return value of **json.loads(json_output)**, which right now exists in **compile_file** method of **inputs/Jsonnet** class.

For each validation type (e.g. Kubernetes, Terraform) the validation json will be retrieved from a different source, as described in the implementation section for each supported type.

Based on this, the additional step (in **bold**) will be integrated with the existing flow as follows:



Logging & Exceptions

There will be another Exception type called SchemaValidationError, that signals the failure of validation to the user. The reason of failure will be logged and propagated to the user.

Testing

Testing will be done using **unittest**. We will need to test:

1. The fetching of validation json can be fetch properly
2. The user parameters related to validation are parsed correctly AND backwards compatible (i.e. users who don't use this feature should be able to continue using kapitan the same way)
3. The caching is done properly, looking at the existence of local files
4. The validation is done correctly

Timeline

During the implementation period, I intend to cover around 30-35 hours per week for the project. From my experience, testing/debugging takes as much time, if not more, than the initial implementation. Therefore, I intend to allocate about 40% of the coding time for them.

Community bonding period: May 7 - May 27

Week 1 (May 7 - 10), Week 2 (May 13- 17):

- Get to know the community/mentors.

- Explore the Kapitan code base back to back, and address minor issues on Github while doing so. Learn the coding standard for Kapitan
- Get to know the various use cases of the tool, and improve the documentation as described in the Github issues with Documentation label to make it even more beginner friendly
- Learn Golang, **cgo,** and Python C APIs for **task 2 - integration with Helm**

Week 3 (May 20 - 24)

- Write more tests for the project in general, to ensure that the addition of new features and/or change in the implementation does not affect the existing features i.e. for regression testing
- Discuss and establish the appropriate YAML parameters to be specified for each of the features with the mentors, including dependency management and json schema validation
- Design the implementation for **task 1 - dependency management**, with all the interfaces clearly defined to ensure smooth integration with the existing code

Coding period: May 28 - August 20, 2019

Week 1 (May 27 - 31)

- design & write the tests for **task 1** fetching of git repositories for the activities below
- Write the code to fetch git repositories
    - YAML parsing
    - Cloning of repository using GitPython (public & private with SSH key)
    - Error handling and logging. Major errors include:
        1. network connection & authentication error
        2. repository not found
        3. subdirectory does not exist
        4. ref does not exist
    - Caching & adding **--fetch** option to **kapitan compile**
- Test the system as a whole with git

Week 2 (June 3 - 7)

- Design & write the tests for fetching of files over HTTP[S] for the activities below
- Write the code to fetch files over HTTP[S]
    - YAML parsing
    - Fetching of files
    - Error handling and logging. Major errors include:
        1. network connection & authentication error
        2. File not found
- Test the system as a whole with git & HTTP[S] dependencies

Week 3 (June 10 - 14), Week 4 (June 17 - 21), Week 5 (June 24 - 28)

- Design & write the tests for task 2 - Helm chart templating
- Implement Python-Go binding
    - Start with no value-overwriting (i.e. render the chart as is)
    - Once the chart can be rendered by calling Go from Kapitan, implement the inventory value overwriting
    - If required, ask questions to Go community via Github issues and mailing list
- Test the system as a whole with Helm templating

Week 6 (July 1 - 5), Week 7 (July 8 - 12)

- Design & write the test for task 4 - json schema validation
- Implement Json schema validation for Kubernetes:
    - YAML parsing
    - Fetching of json schema from online, as described in the task
    - Caching of the files
    - Validation
    - Logging and error messages to the user, if any
- Test the system as a whole with json schema validation

Week 8 (July 15 - 19), Week 9 (July 22 - 26), Week 10 (July 29 - Aug 2)

- Implement kapitan static binary build
    - Setting up the environment (i.e. getting Docker images for Debian 32bit/64bit)
    - Cythonizing the code base
    - Automation of the build process:
        - Write Dockerfile for each image to set up the environment
        - Write the script to build the binary and run it
- Testing:
    - Correctness of the binary: modify the existing tests to test using the binary
    - Speed of the binary
    - Size of the binary

Week 10 (July 29 - Aug 2), Week 11 (Aug 5 - 9)

- System testing as a whole
- Documentation of the work and writing the User Guide & examples for the new features

Week 12 (Aug 12 - 16), Week 13 (Aug 19 - 20)

- Buffer: if any other time available, implement other tasks such as validation of Terraform schemas & help with the implementation of more go-jsonnet C-bindings