narrenschiff

Release 3.4.6

The Narrenschiff Authors

Oct 04, 2021

CONTENTS:

CHAPTER

ONE

USER GUIDE

1.1 Introduction

narrenschiff is a configuration management tool for Kubernetes inspired by Ansible. It can be used to easily source control your manifests, deployment workflows, and to encrypt secrets. In addition to encrypting secrets, it can also encrypt whole configuration files. In essence, it is a wrapper around various tools (e.g. helm, and kubectl). All tools are executed locally on the host OS.

1.1.1 Requirements

- Python 3.6 or higher
- kubectl v1.20 or higher
- helm v3.0 or higher
- gcloud 343.0.0 or higher

1.1.2 Installation

You can easily install it with pip:

```
pip install narrenschiff
```

We advise you to install it in virtualenv.

1.1.3 Quickstart

To install Narrenschiff in virtualenv execute:

```
$ mkdir infrastructure && cd infrastructure
$ git init
$ python3 -m venv env  && echo 'env' > .gitignore
$ . env/bin/activate
$ pip install narrenschiff
```

Initialize a course project, and encrypt a treasure:

```
$ narrenschiff dock --autogenerate --location postgres/
$ narrenschiff chest stash --treasure postgresPassword --value "Password123!" --location␣
↪postgres/
```

(continues on next page)

(continued from previous page)

Create a template for Secret Kubernetes resource, using encrypted treasure:

```
$ mkdir postgres/files/
$ cat > postgres/files/secret.yaml << EOF
---
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: postgres
data:
  POSTGRES_PASSWORD: "{{ postgresPassword | b64enc }}"
EOF
```

Create a course:

```
$ cat > postgres/course.yaml << EOF
---
- name: Add secret to default namespace
  kubectl:
  command: apply
  args:
    filename:
      - secret.yaml
  namespace: "default"
EOF
```

Deploy:

```
$ narrenschiff sail --set-course postgres/course.yaml
```

That's it! Secret is now deployed to your cluster. Head over to General Overview to get familiar with Narrenschiff terminology, or to Getting Started to learn how to make your first project.

1.2 General Overview

Basic unit of narrenschiff is a course i.e. file or group of files storing tasks that are to be performed in sequential order. course is YAML file e.g.:

```
- name: Deploy config map
  kubectl:
    command: apply
    args:
      filename: "configmap.yaml"

- name: Apply namespaces and RBAC settings
  kubectl:
    command: apply
    args:
      filename:
```

(continues on next page)

(continued from previous page)

```
        - namespaces.yaml  # filenames are referenced relative to files/ dir
        - rbac.yaml
```

Each YAML file in the project is treated as a template file i.e. each `course` can have template variables. Template language that is powering Narrenschiff is Jinja2.

File paths are referenced relative to the `files/` directory in the *course project* root. `files/` is reserved for Jinja2 templates of Kubernetes manifests.

The basic directory layout should resemble something like this:

```
infrastructure/  <-- root project (all commands are executed from here)
├── .narrenschiff.yaml  <-- root project configuration
└── webapp  <-- course project
    ├── files  <-- Templates for you manifest files
    │   ├── app
    │   │   ├── configmap.yaml
    │   │   └── deployment.yaml
    │   ├── db
    │   │   ├── deployment.yaml
    │   │   └── secret.yaml
    │   ├── namespaces.yaml
    │   └── rbac.yaml
    ├── chest  <-- directory with arbitrary nesting may be used in addition to chest.yaml
    │   ├── database.yaml
    │   └── secrets.yaml
    ├── vars  <-- directory with arbitrary nesting may be used in addition to vars.yaml
    │   ├── common.yaml
    │   ├── domains.yaml
    │   └── apps
    │       └── prod.yml
    ├── overrides  <-- encrypted helm values.yaml overrides i.e. secretmaps
    │   └── values.yaml
    ├── tasks.yaml  <-- course file describing the deployment process
    ├── secretmap.yaml  <-- paths to encrypted files
    ├── chest.yaml  <-- encrypted variables
    └── vars.yaml  <-- cleartext variables
```

`files/` directory is a directory reserved for your Kubernetes manifests. With `narrenschiff` you can use Jinja2 templating language to inject variables into the manifests.

You can use `vars.yaml` and `chest.yaml` to define variables for you project. `vars.yaml` or `vars/` directory contains unencrypted variables. `chest.yaml` or `chest/` directory is a place to stash your *treasures* (i.e. keys, secrets, passwords, etc.). Both `vars/` and `chest/` directories can have arbitrary nesting and files within them can have arbitrary names. However, all variable names contained across these files **must** be unique! All boolean values must be quoted.

You can also encrypt files and commit them to your source code safely. Files are encrypted, and stored at desired location, and relative paths to the files are saved in `secretmap.yaml`. For example, when you deploy a Helm chart, it is often common to override default `values.yaml`. However, this is unencrypted file which can be used to configure secrets. You can stash your `values.yaml` override as a secretmap, and commit it to the source code without any worry of passwords and secrets leaking.

Chest files have flat dictionary structure. No nesting of the keys is allowed:

```
dbPassword: 6Wziywgso3YsosQNfMeufodDZxEaOyujHM+ch9Pxe5u1u2ZO5e7G9bPOhEIVYo8n
hashKey: uSn/rKMdbMArR0SnWcbtP1Z64/Y8LI8LNOZGbVZUmm5ioFLV/NwP6OcyTNGgMSGi
```

The app is deployed as:

```
$ narrenschiff sail --set-course webapp/tasks.yaml
```

After you execute this, the following happens:

1. All variables from `vars` files, `chests`, and `secretmaps` are collected (only those files that are contained within the course project are used - course project is the directory in which the executed `course` is located)

1. Load `vars.yaml`
2. Load all files from the `vars/` directory if it exists
3. Load and decrypt all variables from `chest.yaml`
4. Load all files from the `chest/` directory if it exists
5. Load all variables from `secretmap.yaml`
6. Merge all files

2. Variables are checked for duplicates, if there are any, the ship cannot take this course
3. Course file is supplied with collected variables and executed
4. Tasks are executed in sequential order, each YAML file is supplied with collected variables, and secretmaps are decrypted

You can either use `chest.yaml` or `chest.yml` file per *course project*, but not both. Choose one extension, and stick to it. A *course project* is a directory where course file is located.

Treasure is encrypted using password (`key`) and salt (`spice`). These are stored in simple text files. The root of the project must contain the `.narrenschiff.yaml` configuration file that stores paths to these files. Keep in mind that while `.narrenschiff.yaml` should be source controlled, password and salt file should never be committed to your repo! Here is the example of the configuration file:

```
# .narrenschiff.yaml
key: ~/.infrastructure/password.txt  # path to file containing password for encrypting␣
↪files
spice: ~/.infrastructure/salt.txt  # path to file containing salt (salt should be random␣
↪and long)
```

If you have a fairly complex course, and you want to execute only a specific set of tasks, you can use *beacons*:

```
- name: List all namespaces
  kubectl:
    command: get namespaces

- name: List all pods
  kubectl:
    command: get pods
  beacons:
    - always
    - pods

- name: Check pod resources
```

(continues on next page)

(continued from previous page)

```
  kubectl:
    command: top pods
  beacons:
    - stats
    - pods

- name: Check node resources
  kubectl:
    command: top nodes
  beacons:
    - stats
```

Now you can easily select which collection of tasks you want to execute:

```
narrenschiff sail --set-course stats.yaml --follow-beacons stats,pods
```

Note that `always` is a special keyword for beacons! Tasks marked with `always` are always executed, regardless of the becaons you specified on the command line.

1.2.1 Glossary

course Templated YAML file containing list of tasks to be performed.

course project Directory in which the main course is located. This directory also contains `vars.yaml`, `chest.yaml`, `secretmap.yaml`, and other files needed to run the course.

treasure Sensitive information, keys, secrets, and passwords are stored

chest File or files in which your treasure is stored.

key Master password for encrypting strings

spice Salt used for encrypting strings

secretmap Encrypted file (currently only supported for `helm` module)

1.3 Getting Started

This guide will lead you from set up of a project to deployment of a service. You can follow this guide using a custom namespace in your existing cluster or, better yet, using Minikube. You need to have `kubectl` installed in order for `narrenschiff` to work. **Everything you write using `narrenschiff` will be executed locally** on the host operating system (just as you would execute `kubectl` or `helm` locally).

1.3.1 Before you Start

We advise you to test this tool using Minikube. This section will briefly cover how to setup and use Minikube, and how to switch between contexts (i.e. clusters).

Execute following comands in order to install Minikube:

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-
↪amd64
sudo cp minikube /usr/local/bin/minikube
sudo chmod +x /usr/local/bin/minikube
```

When starting Minikube `kubectl` should automatically change contexts (i.e. the cluster the `kubectl` is associated with). You can check the current context with:

```
kubectl config current-context
```

And you can switch to your context with:

```
kubectl config use-context <CONTEXT_NAME>
```

Start Minikube with:

```
minikube start --driver=docker
```

You can stop and delete cluster with:

```
minikube stop
minikube delete
```

1.3.2 Setting up a Project

There are two ways to start a project. Manually, and using a built in `narrenschiff dock` command. We advise you to set your projects manually. If this is your first time you are using `narrenschiff`, please review the following section, as it describes the fundamental anatomy of a project.

Manually

Start managing your infrastructure by first laying out a basic directory structure:

```
mkdir infrastructure
cd infrastructure
git init
pipenv python --three
pipenv install narrenschiff
mkdir project
touch project/course.yaml
touch project/vars.yaml
touch project/chest.yaml
touch project/secretmap.yaml
touch .narrenschiff.yaml
```

All course projects will be managed in subdirectories of `infrastructure/` (consider this to be your *main project*). In the root of your infrastructure repo you need to place a `.narrenschiff.yaml` configuration file. This configuration file contains password (key) and salt (spice) that are used to encrypt all files and strings accross all of your course projects.

Each course project needs to contain special files that are used to store cleartext and cyphertext variables. These are:

- `vars.yaml` - Used to store variables in cleartext
- `chest.yaml` - Used to store encrypted variables
- `secretmap.yaml` - Used to store paths to encrypted files

All variables contained in these files are injected in your templates when you start deploying with narrenschiff. There is one rule that you need to remember: **no duplicates are allowed**! See Vars Files for detailed explanation.

Last file that needs to be explained is `course.yaml`. The name of this file can be arbitraty, and you can have multiple of these. This is actually the file which contains configuration, deployment, and other instructions. In essence a `course` is the most basic unit of `narrenschiff`. `course` files are YAML files that contain list of tasks to be performed written using a special syntax. Consequently, the project which contains course files, is called a *course project*. In this example a course project is `project/`.

`narrenschiff dock`

You can easily start a project using `narrenschiff dock`. It is advisable to run `narrenschiff` from virtualenv. For this example, we'll use `pipenv` but you can use any other dependency management too:

```
mkdir infrastructure
cd infrastructure
git init
pipenv python --three
pipenv install narrenschiff
pipenv shell
narrenschiff dock --location project --autogenerate
```

This will create a *course project* on path `project/`. `--autogenerate` flag will generate *key* and *spice* for the project (in the home directory of the user), and add them to `.narrenschiff.yaml`.

1.3.3 Configuring a Project

Configuration of a project is fairly simple. You only need to setup `.narrenschiff.yaml` and accompanying files for *key* and *spice*. If you've used `narrenschiff dock` this should already be done for you. However, when you're setting up a main project manually, you'll have to do this step manually too.

Key and *spice* must not be commited into your source code! Store them somewhere else. They are usually stored in the home directory of a user executing `narrenschiff`:

```
mkdir ~/.infrastructure
cd ~/.infrastructure
head -c 30 /dev/urandom | base64 > password.txt
head -c 30 /dev/urandom | base64 > salt.txt
```

Now you can update your configuration file:

```
# Paste this into your .narrenschiff.yaml configuration file
key: ~/.infrastructure/password.txt
spice: ~/.infrastructure/salt.txt
```

1.3.4 Deploying Your First Service

As an example, we will deploy PostgreSQL. Typically, you deploy database as `StatefulSet`, however in this example we will stick to simple `Deployment`, just to make our life easier. Execute this commands from your *main project*:

```
mkdir -p postgres/files
touch postgres/course.yaml
touch postgres/vars.yaml
touch postgres/chest.yaml
touch postgres/secretmap.yaml
touch postgres/files/deployment.yaml
touch postgres/files/secret.yaml
touch postgres/files/configmap.yaml
```

The way you would usually run postgres in a docker is like so:

```
docker run --name postgres \
  -e POSTGRES_USER=user \
  -e POSTGRES_PASSWORD=password \
  -e POSTGRES_DB=db \
  -d postgres:latest
```

When translating this to Kubernetes manifests, we obviously need to split this into several resources: `Deployment` (for the container itself), `ConfigMap` (for database name and user name), and `Secret` (for password). We will place all these manifests in `files/` directory in the *course project*. In `narrenschiff`, `files/` within a *course project* is reserved for Kubernetes manifests. You can write these configuration using Jinja2 templating language, and `narrenschiff` will inject variables from vars files into the manifests.

Let's write Kubernetes manifests. `ConfigMap` is straightforward:

```
# postgres/files/configmap.yaml
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres
  labels:
    app: postgres
data:
  POSTGRES_USER: user
  POSTGRES_DB: db
```

Nothing new here. However, for the secret, we want to utilize encryption. Normally, secrets in Kubernetes are not encrypted in manifests (only base64 encoded). The original reason `narrenschiff` was made is precisely to overcome this problem - so we can encrypt a secret and source control our infrastructure without compromising it. We'll use `narrenschiff chest` to encrypt our password, and store it in the `chest.yaml`.

```
narrenschiff chest stash --location postgres/ --treasure postgresPassword --value␣
↪Password123!
```

If you take a look inside chest.yaml you'll find your secret:

```
cat postgres/chest.yaml
postgresPassword: 3GghhpUTDrGvGroyhO5J/4TLlpSKUX1hBn3FkgLVd/vq0n6dgCD8+nEB08kYdd2G
```

The name of our secret variables is postgresPassword. And we can use it now anywhere in our manifests. But naturally, we'll use it to define a Secret:

```yaml
# postgres/files/secret.yaml
---
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: postgres
  labels:
    app: postgres
data:
  POSTGRES_PASSWORD: "{{ postgresPassword | b64enc }}"
```

You'll notice that instead of usual base64 encoded string we have "{{ postgresPassword | b64enc }}". This is Jinja2 syntax. It says "hey, replace what's between the double curly braces, and then apply the b64enc filter". When you execute deployment with narrenschiff sail, all secrets accross chest files will be collected, decrypted, and passed to Jinja2 templates for rendering. Then, Jinja2 will replace this secret in a template, but not before passing it through b64enc filter (which encodes string with base64). The end product is what you would normally write as a configuration, the only difference being, you can now safely commit it, and track it with source control, without worrying about secrets being leaked. Only people with *key* and *spice* can decrypt a secret.

This is how it will actually look when rendered:

```yaml
---
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: postgres
  labels:
    app: postgres
data:
  POSTGRES_PASSWORD: "UGFzc3dvcmQxMjMhCg=="
```

But you don't really need to be concerned with how it looks when rendered. Finally, we'll define a Deployment:

```yaml
# postgres/files/deployment.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  labels:
    app: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
```

(continues on next page)

(continued from previous page)

```
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:latest
        ports:
        - containerPort: 5432
        envFrom:
          - configMapRef:
              name: postgres
          - secretRef:
              name: postgres
```

For example, if you want to pin down the version of the postgres, and be able to update it easily later, you can replace `image: postgres:latest` with `image: "{{ postgresDockerImage }}"`, and add to your `vars.yaml` specifig version as: `postgresDockerImage: "posgres:12-alpine"`.

Now, all we have to do is to deploy this to our cluster. We would usually do this like so:

```
kubectl appyl -f secret.yaml,configmap.yaml,deployment.yaml --namespace default
```

However, we don't have ordinary Kubernetes manifests anymore. Now we're using templated manifests. Therefore we need to write the equivalent command using Narrenschiff. Open `course.yaml` and write the following:

```
# postgres/course.yaml
- name: Deploy postgres
  kubectl:
    command: apply
    args:
      filename:
        - secret.yaml
        - configmap.yaml
        - deployment.yaml
      namespace: "default"
```

In Narrenschiff, this is called a *taks*, and *course* is a collection of tasks.

Now, all you have to do is to deploy this to our cluster. Or, in other words, set a course and sail this crazy ship:

```
narrenschiff sail --follow-course postgres/course.yaml
```

The output should be similar to this:

```
* [ 2020-07-17 19:33:27.852325 ] * [ Deploy postgres ] ***************

secret/postgres created
configmap/postgres created
deployment.apps/postgres created
```

1.4 More than a satire, a tool!

1.4.1 So, what's your selling point?

Encrypted secrets (strings) and files.

1.4.2 What's up with all the funky terminology?

Why is it that some of the biggest **cloud** tools (Docker, Kubernetes, Helm) out there are named using **ocean** related references? Heck, there's even a cloud provider having ocean in its name.

So I decided to make both a tool, and a satire. Therefore, I needed a name that will be in the spirit of the cloud, i.e. related to seas, and oceans of course. And also a name that would describe the current state of DevOps. Narrenschiff (ship of fools) was the only logical choice.

I also find multitude of terminology *amusing* (to say the least): Ansible calls its YAML files playbooks, and vars files, Kubernetes calls its YAML files manifests, and Helm calls bundled templated YAML files – charts.

And why does everything has to have its own terminology or domain specific language? Didn't we have enough of the tool fatigue? To cite one GitHub user:

> "The concept of tool fatigue is real – the k8s operation space is already crowded as it is, don't make people learn *extra* tools unless there's a really good reason, and this does not seem like one…" jshearer

I fully and unironiracally agree with **jshearer**! Thefore I decided to go even a step further! Not only will I call YAML files by a special name, I will call them by many names: courses, chests, secretmaps, vars. Heck, passwords and salts? Pfff… That's yesterdays story: they are now keys and spices. No more boring old terminology we all got used to! *Stash* your secrets in a chest (don't just say "I'm *encrypting* them"), or *loot* a `--treasure` from the chest (saying "decrypting a secret" is for boomers). Deployment? Nah, board the `narrenschiff`, `--set-course`, and `sail`.

1.4.3 We already have `secretGenerator` option in kustomize. Why should we use Narrenschiff?

Since we all have a tool lock-in with Kubernetes you might as well use all the available options the tool gives you.

That being said, it is really easy to encrypt `values.yaml` if it contains secrets, or to template your Manifests, with Narrenschiff. You can even use `secretGenerator` with Narrenschiff.

Although I consider the encrypted strings and files to be the most powerful features of Narrenschiff, it's also a "procedural/declerative task executor" or whatever you want to call it (it's **YAML instead of** a **bash** script - come to think of it, we should also have an acronym for that - YIOB - it just rolls off the tongue). So this is one more reason to use it. It's a Kubernetes configuration management tool for small business that were insanse enough to start with Kubernetes in the first place.

1.4.4 Ansible has Kubernetes module, and it is a mature tool, why should we choose Narrenschiff over it?

Yes. In general. When I had to make the decision on how to manage the cluster the most important thing for me was to encrypt secrets and push them to the repo. I also wanted to encrypt certificates for Helm Tiller (since Tiller still existed at that point) and source commit them without leaking content. And configuration management (preparation of the cluster, and management later) was a must. Ansible was almost a natural choice, since there wasn't anything similar at the time (at least to my knowledge).

And yes, it has both the Kubernetes and the GCP module. However, when I tried the most basic of basic commands, for listing inventories, it failed. Keep in mind, this was just a first step to do when using the damn plugin. I submitted the bug report and it was actually fixed two days later, however, I didn't want to bother with it anymore, since most basic configuration was failing. And, the thing I found confusing (and not pleasing) was pasting of the whole Kubernetes manifests inside playbooks, or using awkward syntax for template lookup.

Ansible is great tool and battle tested. Narrenschiff at this point is still in beta. The choice is clear... (choose a tool in beta of course! - we always need to be on a bleeding edge and work with more and more tools, and new technologies, r-right?)

And also... I wanted to build my own tool.

1.4.5 But why?

It's just a tool bro.

1.5 Courses

A course in Narrenschiff is a collection of tasks. A single task corresponds to a single command line you would typically execute using e.g. `kubectl` or `helm`.

This is a general anatomy of a task:

```yaml
- name: Deploy nginx container with basic auth  # name of the task (required)
  kubectl:  # module you're using (required)
    command: apply
    args:
      filename:
        - nginx/app/secret.yaml
        - nginx/app/configmap.yaml
        - nginx/app/deployment.yaml
      namespace: default
  beacons:  # tags for alternative execution path (optional)
    - dev  # chose whatever value ("always" is a reserved becaon)
```

`name` of the task describes what task should do. On the other hand, the module you are writing describes what the task is actaully doing. Names are required, so your infrastructure configuration becomes a self-documenting repo. Name is an arbitraty description. Module, however, can only be whatever is implemented in Narrenschiff. At this moment, there are several modules available to use:

- `kubectl`
- `helm`
- `gcloud`
- `kustomization`
- `wait_for_pod`

For more info about modules, visit the Modules section.

Everything nested under module (in this case command, args, filename, namespace) is unique to the module. These options are covered in the module documentation.

beacons are optional, and they provide you a way for alternative execution path of your course. In other words, you can choose which part of the course you want to execute using becaons. Since they are optional, if you don't need them, don't add them. See Beacons for detailed overview.

Tasks in a course are executed sequentially. When you execute a course with narrenschiff sail all variables are gathered, templates rendered, tasks collected in ordered list, and executed one by one.

You can combine multiple courses in a single course using an import feature. For example, you may want to separate your cluster creation from cluster configuration:

```
touch project/course.yaml
touch project/gke.yaml
touch project/init.yaml
```

And in your *course.yaml* you would import other courses as:

```yaml
---
# project/course.yaml
- name: Make cluster with gcloud
  import_course: "gke.yaml"

- name: Configure cluster
  import_course: "init.yaml"
```

All **imports are relative to the** *course project* **directory**. So for example, if you have two courses that share the same set of tasks, you can put them in e.g. includes/ and import them wherever you need:

```
touch postgres/includes/repo.yaml
touch postgres/deployment.yaml
touch postgres/upgrade.yaml
```

And in one of the courses, you can import the task as:

```yaml
---
# postgres/deployment.yaml
- name: Update helm repo
  import_course: "includes/repo.yaml"

# other tasks
[...]
```

1.6 Vars Files

There are three types of variable files in Narrenschiff:

- vars - plain variables i.e. non-encrypted variables
- chest vars - encrypted variables
- secretmap vars - variables that store paths to files encrypted using Narrenschiff

1.6.1 Plain Variables

These are the most basic of all variable types. Plain variables give you ability to easily reuse variables accross your templates. These variables also support arbitraty nesting of variables.

```yaml
# vars.yaml
clusterName: clusty-mc-cluster  # simple "flat" variable
namespaces:  # variable with nesting
  dev: development
  stage: staging
  prod: production
```

You can use these variables anywhere in your course files, or in any template file located in `files/`. For example:

```yaml
# course to create a cluster
---
- name: Cluster create
  gcloud:
    command: "container clusters create {{ clusterName }}"
    args:
      num-nodes: 3
```

In the following example we will use the Jinja2 templating feature to loop through nested variable, and create Kubernetes resources accross multiple namespace in one file (instead of writing multiple files):

```
# files/namespaces.yaml
{% for key, value in namespaces.items() -%}
---
apiVersion: v1
kind: Namespace
metadata:
  name: "{{ value }}"
  labels:
    name: "{{ value }}"
{% endfor %}
```

Plain vars are stored in `vars.yaml` or in `vars/` directory. If you are using `vars/` directory, you can name vars files within it whatever you want, and nest them however you need. The only rule you have to remember is that variable names must be unique accross all vars files (you can't have variable with the same name in chests, and in vars, or secretmaps).

Since vars files (and chests too) are YAML files, all boolean values must be quoted.

1.6.2 Chest

Chest variables behave similar to plain variables. However, there are two important differences:

- You have a command line tool to manage them
- Chest vars don't support arbitrary nesting (they have a flat dictionary structure)

You can also use `chest.yaml` as well as `chest/` directory with custom named files. However, the same rule applies for the chest variable names: they have to be unique accross all vars files.

```
# chest.yaml
# the following two are ok:
dbPassword: enSkvMbU3Sa3YimjyqR3rskZHx3tUYlIpC5U1Xpo3k/qntCCp+HyJfTtjg++tSTF
hashingKey: j9gc0niSm1ADGK/95jVr7ugeUe87wDsCBhUp1zGtw3oJ4nz+h9JJKHfHdmYWFz8b

# nesting of variables is not supported for chest files!
mistery:  # this is NOT OK
  dbPassword: enSkvMbU3Sa3YimjyqR3rskZHx3tUYlIpC5U1Xpo3k/qntCCp+HyJfTtjg++tSTF
  hashingKey: j9gc0niSm1ADGK/95jVr7ugeUe87wDsCBhUp1zGtw3oJ4nz+h9JJKHfHdmYWFz8b
```

When using secrets in your templates, there is no any special step that you have to take in order to decrypt them. Narrenschiff does that for you when it collects all the variables. However, when you're using them with secret files, you have to follow Kubernetes way of writing `Secret` resources, so you'll have to base64 encode them. Narrenschiff offers you a custom Jinja filter (`b64enc`) to easily to that:

```
---
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: postgres
  labels:
    app: postgres
data:
  DB_PASSWORD: "{{ dbPassword | b64enc }}"
  SECRET_KEY: "{{ hashingKey | b64enc }}"
```

`narrenschiff chest` offers you a number of ways to work with secrets. You can either encrypt them on the command line, and paste them into chest files yourself (with `narrenschiff chest lock` and `narrenschiff chest unlock`), or you can dynamicall update `chest.yaml` (with `narrenschiff chest stash` and `narrenschiff chest loot`).

Lock and unlock are useful when you want to try things out. They don't require a location to use (but you need to execute them from the root of the project).

```
$ narrenschiff chest lock --value 'password'
enSkvMbU3Sa3YimjyqR3rskZHx3tUYlIpC5U1Xpo3k/qntCCp+HyJfTtjg++tSTF
$ narrenschiff chest lock --value 'key'
j9gc0niSm1ADGK/95jVr7ugeUe87wDsCBhUp1zGtw3oJ4nz+h9JJKHfHdmYWFz8b
$ narrenschiff chest unlock --value enSkvMbU3Sa3YimjyqR3rskZHx3tUYlIpC5U1Xpo3k/
↪qntCCp+HyJfTtjg++tSTF
password
$ narrenschiff chest unlock --value j9gc0niSm1ADGK/
↪95jVr7ugeUe87wDsCBhUp1zGtw3oJ4nz+h9JJKHfHdmYWFz8b
key
```

However, it's often easier to update `chest.yaml` dynamically, and not worry about whether you copy/pasted whole string from the command line (are you sure you haven't missed that first or last character when selecting?):

```
$ narrenschiff chest stash --treasure dbPassword --value password --location project/
$ narrenschiff chest loot --treasure dbPassword --location project/
```

Also if you want to update `chest.yaml` with treasure that lies on your filesystem, you can test if the encryption works with `lock` and `unlock`:

```
$ narrenschiff chest lock --value "$( cat ~/Downloads/service-account.json )"
```

And you can stash it automatically with:

```
$ narrenschiff chest stash --location project/ --treasure serviceAccount --value "$( cat␣
↪~/Downloads/service-account.json )"
```

If you're in a hurry and you'll need to skim through all variables to find something, you can dump all chest variables to STDOUT with `narrenschiff chest dump`:

```
$ narrenschiff chest dump --location examples/

  dbPassword: password
  hashingKey: key
```

1.6.3 Secretmap

Secretmap variables store paths to encrypted files. Encrypted files don't support Jinja2 templating, and they are only reserved for use with the `helm` module.

These variables are stashed in `secretmap.yaml`, and this file can only be dynamically updated.

The most basic of commands is `narrenschiff secretmap stash`

```
$ narrenschiff secretmap stash --treasure dev --location project/ --source ~/repos/
↪source/dev.yaml --destination overrides/dev.yaml
$ tree project/
project/
├── overrides
│   └── dev.yaml
├── secretmap.yaml
└── course.yaml
```

As you can see, `--destination` is a *path relative to the root of the course project*. Note, it is **not** a path relative to the root project of your infrastructure (where `.narrenschiff.yaml` file is located). The course project is project that contains in its root files such as `secretmap.yaml` and `chest.yaml`. So, in other words, it's a path relative to the `secretmap.yaml`. `--source` on the other hand, can be any path on your filesystem. If you inspect `dev.yaml`, after encryption, in `overrides/`, you'll se that content of the file has been encrypted indeed. And if you inspect `secretmap.yaml` you'll find a relative path to the encrypted file:

```
$ cat project/secretmap.yaml
dev: overrides/dev.yaml
```

How do you reference this in your Narrenschiff configuration? When `narrenschiff sail` gets executed, it needs to decrypt the file before it can be used. We instruct Narrenschiff that following variable is not a simple variable, but a path to a file, with a custom narrenschiff Jinja2 filter:

```
- name: Install Prometheus
  helm:
    command: install
    name: redis
    chart: bitnami/redis
    version: 10.7.11
```

(continues on next page)

(continued from previous page)

```
    opts:
      - atomic
    args:
      namespace: development
      values:
        - "{{ dev | secretmap }}"
```

Therefore, when working with secretmaps, you'll have to pipe your variable to `secretmap` filter in your courses i.e. `{{ dev | secretmap }}`.

The inverse operation of stash is loot. You can decrypt a file and place it somewhere on your filesystem with:

```
$ narrenschiff secretmap loot --treasure dev --location project/ --destination /tmp/dev.yaml
```

However, editing file in such a way is cumbersome. Fortunately, we have `alter` available. It will open a file in your preferred editor (or `vi`):

```
$ narrenschiff secretmap alter --treasure dev --location project/
```

If you want to change default editor, change `EDITOR` environment variable to preferred editor.

Sometimes you just want to preview the file. Narrenschiff got you covered here also. Use `peek` to dump file content to STDOUT:

```
$ narrenschiff secretmap peek --treasure dev --location project/
```

When you have many secretmaps in a course project, it's really hard to peek and manually search through all of them. Narrenschiff gives you ability to grep over those encrypted files with `search`:

```
$ narrenschiff secretmap search --match "ClusterIP" --location project/
```

A really powerful feature of Narrenschiff secretmap search is that match pattern can be a Python regex expression.

If you have two secretmaps within a course project that you want to compare, you can use `diff`:

```
$ narrenschiff secretmap diff --location project/ dev prod
```

And finally, you can delete secretmaps with:

```
$ narrenschiff secretmap destroy --treasure dev --location project/
```

1.6.4 Rules

There is one rule that you need to remember: **no duplicates are allowed**! `narrenschiff` collects all variables in all var files, and if you have duplicate names, the program will exit with an error. And you should also keep in mind this tiny rule:

1. All variables from `vars` files, `chests`, and `secretmap` are collected (only those files that are contained within the *course project* are used)

1. Load `vars.yaml`
2. Load all files from the `vars/` directory if it exists
3. Load and decrypt all variables from `chest.yaml`

4. Load all files from the `chest/` directory if it exists
5. Load all variables from `secretmap.yaml`
6. Merge all files

2. Variables are checked for duplicates, if there are any `narrenschiff sail` will fail

Jokes aside, there is no variable file precedence as in Ansible. All vars files are created equal, and each treasure name within it is unique. If you have duplicates, Narrenschiff will let you know, so you can fix this. Not having to think about vars file precedence streamlines thought process, leaving you more time to think about your infrastructure, rather than the quirks of the tool you're using.

1.7 Beacons

If you only want to execute a part of the course, then you should use beacons. Beacons are essentially tags on you tasks that allow you to create special paths of execution.

Note: We advise you to test the example using Minikube. You can find instructions on how to setup Minikube here and also in the official documentation.

Let's look at simple example:

```
# examples/stats/course.yaml
- name: List all namespaces
  kubectl:
    command: get namespaces
  beacons:
    - always

- name: List all pods in default namespace
  kubectl:
    command: get pods
    args:
      namespace: default
  beacons:
    - default

- name: List all pods in kube-system namespace
  kubectl:
    command: get pods
    args:
      namespace: kube-system
  beacons:
    - kube-system
```

Beacons are passed to `narrenschiff` with `--follow-beacons` flag, like so:

```
$ narrenschiff sail --follow-beacons default --set-course examples/stats/course.yaml

* [ 2020-07-20 10:47:36.901965 ] * [ List all namespaces ] ******************

NAME              STATUS   AGE
```

(continues on next page)

(continued from previous page)

```
default           Active   2d17h
kube-node-lease   Active   2d17h
kube-public       Active   2d17h
kube-system       Active   2d17h


* [ 2020-07-20 10:47:36.959850 ] * [ List all pods in default namespace ] ***

NAME                         READY   STATUS    RESTARTS   AGE
postgres-7bf8d6b875-hp58b    1/1     Running   1          2d15h
```

What happened? `always` is a reserved beacon in Narrenschiff. When you execute a course, if you have any task tagged with `always` it will always execute in addition to the tasks targeted by supplied beacon. In this case, we chose to execute only those tasks marked with `default` i.e. `--follow-beacons default`.

A task can be marked with multiple becaons e.g.

```
beacons:
  - dev
  - stage
  - prod
```

And you can also select multiple becaons from the command line:

```
$ narrenschiff sail --follow-beacons dev,stage --set-course course.yaml
```

Here's a practical example. If you are using Helm to manage you applications, you can pack the upgrade instructions in a single course, but separate environments using becaons.

```
# helm/postgres.yaml
- name: Add bitnami repo to Helm
  helm:
    command: repo add jetstack https://charts.bitnami.com/bitnami
  beacons:
    - always

- name: Update repo
  helm:
    command: repo update
  beacons:
    - always

- name: Upgrade Postgres on development
  helm:
    command: upgrade
    name: postgres
    chart: bitnami/postgresql
    version: 11.8.0
    opts:
      - atomic
      - cleanup-on-fail
      - reuse-values
    args:
```

(continues on next page)

(continued from previous page)

```yaml
      namespace: development
      values:
        - "{{ values | secretmap }}"
  beacons:
    - dev

- name: Upgrade Postgres on staging
  helm:
    command: upgrade
    name: postgres
    chart: bitnami/postgresql
    version: 9.1.1
    opts:
      - atomic
      - cleanup-on-fail
      - reuse-values
    args:
      namespace: staging
      values:
        - "{{ values | secretmap }}"
  beacons:
    - stage

- name: Upgrade Postgres on production
  helm:
    command: upgrade
    name: postgres
    chart: bitnami/postgresql
    version: 9.1.1
    opts:
      - atomic
      - cleanup-on-fail
      - reuse-values
    args:
      namespace: production
      values:
        - "{{ values | secretmap }}"
  beacons:
    - prod
```

Now, if you want to upgrade only your service on the development environment, you can do this without executing other tasks in the course:

```
$ narrenschiff sail --follow-becaons dev --set-course helm/postgres.yaml
```

Beacons can only be used on tasks. They cannot be used on course imports (i.e. `import_course` does not support becaons).

1.8 .narrenschiff.yaml

Every project needs to contain .narrenschiff.yaml configuration file. This file is used to point to the password, and the salt, as well as to define the cluster context. This file should be commited to the source code of your project.

Files containing password, and salt, should never be commited to the source code!

```yaml
# Example of configuration file
key: ./password.txt  # path to file containing password for encrypting files
spice: ./salt.txt  # path to file containing salt (salt should be random and long)

# If you are managing multiple clusters, your project may be tied to one
# particular cluster. Instead of switching context manually with:
#
#   kubectl config use-context CONTEXT_NAME
#
# you can define context in this section of your project configuration file.
# If you are not sure which context you are using, check it with:
#
#   kubectl config get-contexts
#
# or:
#
#   kubectl config current-context
#
# context.name is the name of the context [default: undefined]
# context.use defines whether to use context switching before executing tasks
#   value is (quoted) boolean string [default: "false"]
context:
  name: cloud_provider_project_id_region_zone_project_title
  use: "false"
```

1.9 Modules

1.9.1 kubectl

Description

Execute kubectl commands. Interact with your Kubernetes cluster.

Requirements

- kubectl

Parameters

Parameter	Comment
command	Any kubectl command with nested subcommands
args	flag/value pairs, a flag needs to be listed by its full name (e.g. you can't use `f` for `-f/--filename`, you have to use `filename`)
opts	flags without arguments i.e. switches

Examples

```
# Both get (command) and namespaces (subcommand) are inputs for parameter "command"
# kubectl get namespaces
- name: List all namespaces
  kubectl:
    command: get namespaces

# "args" paramater accepts only flag/value pairs
# kubectl get pods --namespaces default
- name: List all pods in default namespace
  kubectl:
    command: get pods
    args:
      namespace: default

# Use "opts" when you need flags without input values
# kubectl get pods --all-namespaces
- name: List pods in all namespaces
  kubectl:
    command: get pods
    opts:
      - all-namespaces

# If you need to create a namespace you can just put a whole command in "command"␣
↪paramater
# kubectl create namespace cert-manager
- name: Create namespace for cert-manager
  kubectl:
    command: create namespace cert-manager

# Value of a flag depends on underlying input kubectl is expecting
# kubectl apply -f project/files/secretmap.yaml,project/files/configmap.yaml,project/
↪files/deployment.yaml --namespace development
- name: Deploy application
  kubectl:
  command: apply
  args:
    filename:
      - secretmap.yaml  # in Narrenschiff paths are relative to files/ dir in a course␣
↪project
```

(continues on next page)

(continued from previous page)

```
      - configmap.yaml
      - deployment.yaml
    namespace: development

# If you have a single file that you have to pass to -f, you can also use this syntax
# kubectl apply -f project/files/configmap.yaml
- name: Deploy config map
  kubectl:
    command: apply
    args:
      filename: configmap.yaml

# You can also apply configurations from URLs
# kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/release-0.8/
↪deploy/manifests/00-crds.yaml --namespace cert-manager
- name: Install cert-manager
  kubectl:
    command: apply
    args:
      filename:
        - https://raw.githubusercontent.com/jetstack/cert-manager/release-0.8/deploy/
↪manifests/00-crds.yaml
      namespace: cert-manager

# You can mix URLs with file paths
- name: Deploy applications
  kubectl:
  command: apply
  args:
    filename:
      - secretmap.yaml
      - configmap.yaml
      - deployment.yaml
      - https://raw.githubusercontent.com/kubernetes/kubectl/master/testdata/apply/
↪deploy-clientside.yaml
    namespace: development
```

Status

Warning: This module is experimental.

1.9.2 kustomization

Description

A very simple wrapper around `kubectl apply -k`.

Requirements

- kubectl

Parameters

Parameter	Comment
N/A	Module does not support additional parameters

Examples

```
# This module executes "kubectl apply -k <directory>"
- name: Deploy app
  kustomization: app/  # in Narrenschiff paths are relative to files/ dir in a course␣
↪project
```

Status

Warning: This module is experimental.

1.9.3 helm

Description

Execute helm commands. Deploy, upgrade, and delete your charts.

Requirements

- kubectl
- helm

Parameters

Parameter	Comment
command	Any helm command with nested subcommands
args	flag/value pairs, a flag needs to be listed by its full name
opts	flags without arguments i.e. switches
chart	name of the chart
name	name of the chart release

Examples

```yaml
# If you have a simple command with an argument you can write it like this
# helm repo add stable https://kubernetes-charts.storage.googleapis.com/
- name: Add stable repo to Helm
  helm:
    command: repo add stable https://kubernetes-charts.storage.googleapis.com/

# Command (repo) and its subcommand (update) are both placed in the "command"
# parameter of Narrenschiff helm module
# helm repo update
- name: Update repo for stable charts
  helm:
    command: repo update

# use "chart" and "name" to define chart and its release
# helm install ingress-controller stable/nginx-ingress --atomic --namespace default
- name: Install Ingress controller
  helm:
    command: install
    name: ingress-controller  # release of the chart
    chart: stable/nginx-ingress  # chart you're using from the chart repo
    opts:  # all switches are listed in opts as YAML list
      - atomic
    args:  # all flags with inputs go under args
      namespace: default
      version: 1.41.1

# You can use "set" to set values on the command line
- name: Install cert-manager
  helm:
    command: install
    name: cert-manager
    chart: jetstack/cert-manager
    opts:
      - atomic
    args:
      namespace: cert-manager
      version: v0.12.0
      set:  # set is a YAML list of key/value pairs
        - ingressShim.defaultIssuerName=letsencrypt-prod
        - ingressShim.defaultIssuerKind=ClusterIssuer
        - ingressShim.defaultIssuerGroup=cert-manager.io

# In Narrenschiff you can combine "set" with templating if you need to set some secrets
- name: Install PostgreSQL
  helm:
    command: install
    name: postgres
    chart: bitnami/postgresql
    opts:
      - atomic
    args:
```

(continues on next page)

(continued from previous page)

```
      version: 9.1.1
      set:
        - "global.postgresql.postgresqlPassword={{ postgresqlPassword }}"

# Values can be used when you have URLs as your chart values
- name: Upgrade Chart Museum
  helm:
    command: upgrade
    name: museum
    chart: stable/chartmuseum
    opts:
      - atomic
      - cleanup-on-fail
    args:
      version: 2.13.0
      values:
        - https://github.com/helm/charts/blob/master/stable/chartmuseum/values.yaml

# Files in "values" are always passed as secretmaps!
- name: Install Graylog
  helm:
    command: install
    name: "graylog"
    chart: stable/graylog
    opts:
      - atomic
    args:
      namespace: "graylog"
      version: 1.6.9
      values:
        - "{{ ingress | secretmap }}"
        - "{{ service | secretmap }}"
```

Status

> **Warning:** This module is experimental.

1.9.4 gcloud

Description

Execute `gcloud` commands. Manage GKE.

Execution of the module will fail for the commands that require user input. Use `--quiet` instead. See gcloud docs for more info.

Requirements

- `gcloud`

Parameters

Parameter	Comment
command	Any gcloud command with nested subcommands
args	flag/value pairs, a flag needs to be listed by its full name
opts	flags without arguments i.e. switches

Examples

```
# gcloud container clusters create test-cluster --enable-ip-alias --quiet --num-nodes 1 -
↪-machine-type n1-standard-2 --zone europe-west1-a
- name: Create the cluster
  gcloud:
    command: "container clusters create test-cluster"
    args:
      num-nodes: 1
      machine-type: n1-standard-2
      zone: europe-west1-a
    opts:
      - enable-ip-alias  # opts are always listed as a YAML list
      - quiet

# If you do not need any flags or switches you can place everything under the "command"
- name: Get credentials for the cluster
  gcloud:
    command: "container clusters get-credentials test-cluster"

- name: Enable Kubernetes API
  gcloud:
    command: "services enable container.googleapis.com"
```

Status

Warning: This module is experimental.

1.9.5 wait_for_pod

Description

Wait for a pod to be ready.

Requirements

- kubectl

Parameters

Parameter	Comment
namespace	Namespace in which the pod is located
threshold_replicas	How many replicast should be ready in order for task to be considered completed
grep_pod_name	Pod you are waiting

Examples

```yaml
- name: Wait for a pod
  wait_for_pod:
    namespace: cert-manager
    threshold_replicas: 1  # This is 1 in e.g. 1/2
    grep_pod_name: cert-manager-webhook  # not a full name, only a part
```

Status

Warning: This module is experimental.

1.10 Examples

1.10.1 Deploying WordPress and MySQL with Persistent Volumes

This examples is taken from the official Kubernetes documentation which is licensed under CC BY 4.0. The example will be addapted to fit deployment process in Narrenschiff.

Before you Start

We advise you to use Minikube as you follow along this tutorial. Minikube is easy to setup, and manage. See the installation process in the official documentation, or see our getting started tutorial.

You can find the source code for this tutorial in the official Narrenschiff repo under the `examples/` directory. The example in the repo uses `.narrenschiff.yaml` configuration from the repo itself. But, for completeness, we will show you how to properly start a project.

Start the Project

Start by installing Narrenschiff and making a project.

```
$ mkdir wordpress && cd wordpress
$ git init
$ python3 -m venv env  && echo 'env' >> .gitignore
$ . env/bin/activate
$ pip install narrenschiff
```

Now you are ready to make a course project. In the root project, execute following command:

```
$ narrenschiff dock --autogenerate --location wordpress
```

Stash Your Treasure in a Chest

We need a password for MySQL and wel'll start by stashing the password in the chest.

```
$ narrenschiff chest stash --treasure 'mysqlPassword' --value 'Password123!' --location␣
↪wordpress/
```

In the next section we'll add Kubernetes resources to our project.

Add Resource Configuration for MySQL and Wordpress

We'll add MySQL password to the template of the `Secret`. Place the template under `wordpress/files/mysql/secret.yaml`:

```
# wordpress/files/mysql/secret.yaml
---
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: mysql-pass
data:
  MYSQL_ROOT_PASSWORD: "{{ mysqlPassword | b64enc }}"
```

Remember, `files/` is reserved in a course project for templates. All templates are referenced in courses relative to this directory.

Make the following files in `wordpress/files/mysql` (filenames are in the comments):

```
# wordpress/files/mysql/service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  ports:
```

(continues on next page)

(continued from previous page)

```
    - port: 3306
  selector:
    app: wordpress
    tier: mysql
  clusterIP: None

# wordpress/files/mysql/pvc.yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 50Mi

# wordpress/files/mysql/deployment.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        envFrom:
          - secretRef:
              name: mysql-pass
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
```

(continues on next page)

(continued from previous page)

```
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

These are all simple manifests. We didn't have any need to use templating here. But the neat thing about Narrenschiff is that you can add templating to whatever manifest you need whenever you need it. We will now proceed to write manifests for the Wordpress itself:

```
# wordpress/files/wordpress/secret.yaml
---
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: wordpress-env
data:
  WORDPRESS_DB_PASSWORD: "{{ mysqlPassword | b64enc }}"

# wordpress/files/wordpress/service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer

# wordpress/files/wordpress/pvc.yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

(continues on next page)

(continued from previous page)

```yaml
# wordpress/files/wordpress/deployment.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
      - image: wordpress:5.5-apache
        name: wordpress
        env:
        - name: WORDPRESS_DB_HOST
          value: wordpress-mysql
        envFrom:
          - secretRef:
              name: wordpress-env
        ports:
        - containerPort: 80
          name: wordpress
        volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
      volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: wp-pv-claim
```

Deploy

Before deployment, we have to write the course:

```yaml
# wordpress/course.yaml
---
- name: Deploy Mysql
  kubectl:
    command: apply
    args:
```

(continues on next page)

(continued from previous page)

```
      filename:
        - mysql

- name: Deploy Wordpress
  kubectl:
    command: apply
    args:
      filename:
        - wordpress
```

Finally apply your changes to the cluster:

```
$ narrenschiff sail --set-course wordpress/course.yaml
```

Verify

You can verify that wordpress is deployed by accessing it through your web browser to finish the installation:

```
$ minikube service wordpress --url
```

Copy and paste the URL to your web browser, and you can complete the wordpress intallation. Use `minikube stop && minikube delete` to stop and delete the cluster.

1.10.2 Deploying Moodle and MariaDB using Helm

This example will show you how to use secretmaps and Narrenschiff helm module in order to deploy two services to the cluster.

Before you Start

We advise you to use Minikube as you follow along this tutorial. Minikube is easy to setup, and manage. See the installation process in the official documentation, or see our getting started tutorial.

You can find the source code for this tutorial in the official Narrenschiff repo under the `examples/` directory. The example in the repo uses `.narrenschiff.yaml` configuration from the repo itself. But, for completeness, we will show you how to properly start a project.

Start the Project

Start by installing Narrenschiff and making a project.

```
$ mkdir moodle && cd moodle
$ git init
$ python3 -m venv env  && echo 'env' >> .gitignore
$ . env/bin/activate
$ pip install narrenschiff
```

Now you are ready to make a course project. In the root project, execute following command:

```
$ narrenschiff dock --autogenerate --location moodle
```

Stash Your Treasure in a Chest

When deploying application with helm, you can use `--set` to set values on the comand line, or `--values` to read in values from a file. `--set` is useful when you want to edit a single value, but when you need to enter more complex configuration, you would use `--values`.

Because we need to supply database password in two different charts, we will use `--set` to supply it at course runtime. However, all other values specific to the charts themselves can be configured with `--values`.

Therefore we will start with the most basic. Stashing our MariaDB password in the course chest.

```
$ narrenschiff chest stash --location moodle/ --treasure databasePassword --value
↪'Password123!'
```

Prepare Overrides for values.yaml

Narrenschiff's helm module can work with `values.yaml`. But by default, these files need to be encrypted. Values file oftentime contains sensitive information such as passwords, so it's better to have it encrypted. Make the `overrides/` directory in your course file, and make these two files within it:

```
# moodle/overrides/mariadb.yaml
---
replication:
  enabled: false

master:
  resources:
    limits:
      cpu: 500m
      memory: 512Mi
    requests:
      cpu: 50m
      memory: 128Mi

  livenessProbe:
    enabled: false
  readinessProbe:
    enabled: false

  persistence:
    enabled: true
    mountPath: /bitnami/mariadb
    accessModes:
      - ReadWriteOnce
    size: 500Mi

# moodle/overrides/moodle.yaml
---
moodleUsername: admin
moodlePassword: Password123!
```

(continues on next page)

(continued from previous page)

```yaml
moodleEmail: admin@moodle.local

mariadb:
  enabled: false
  secret:
    requirePasswords: false

livenessProbe:
  enabled: false
readinessProbe:
  enabled: false

service:
  type: NodePort
  port: 80
  httpsPort: 443
  nodePorts:
    http: "30080"
    https: "30443"
  externalTrafficPolicy: Cluster

ingress:
  enabled: false
  certManager: false
  hostname: moodle.local

persistence:
  enabled: true
  accessMode: ReadWriteOnce
  size: 500Mi

resources:
  requests:
    memory: 512Mi
    cpu: 500m
```

Encrypt them using:

```
$ narrenschiff secretmap stash --source moodle/overrides/mariadb.yaml --destination overrides/mariadb.yaml --treasure mariadb --location moodle
$ narrenschiff secretmap stash --source moodle/overrides/moodle.yaml --destination overrides/moodle.yaml --treasure moodle --location moodle
```

If you inspect your secretmap file, you will see that it contains paths to the encrypted files:

```
$ cat moodle/secretmap.yaml
mariadb: overrides/mariadb.yaml
moodle: overrides/moodle.yaml
```

Update vars.yaml

It’s a good idea to respect DRY principle in your course files. For this purpose, we will utilize vars.yaml and define common cleartext variables:

```
# moodle/vars.yaml
namespace: default

database:
  user: moodle
  name: moodle
```

Deploy

Before deployment, we have to write the course:

```
# moodle/course.yaml
---
- name: Add bitnami repo
  helm:
    command: repo add bitnami https://charts.bitnami.com/bitnami

- name: Update helm repo
  helm:
    command: repo update

- name: Install MariaDB database
  helm:
    command: upgrade
    name: mariadb
    chart: bitnami/mariadb
    opts:
      - install
      - atomic
      - cleanup-on-fail
    args:
      namespace: "{{ namespace }}"
      version: 7.9.2
      values:
        - "{{ mariadb | secretmap }}"
      set:
        - "db.user={{ database.user }}"
        - "db.password={{ databasePassword }}"
        - "db.name={{ database.name }}"

- name: Install Moodle
  helm:
    command: upgrade
    name: moodle
    chart: bitnami/moodle
    opts:
      - install
```

(continues on next page)

(continued from previous page)

```
      - atomic
      - cleanup-on-fail
    args:
      namespace: "{{ namespace }}"
      version: 8.1.1
      values:
        - "{{ moodle | secretmap }}"
      set:
        - "externalDatabase.user={{ database.user }}"
        - "externalDatabase.password={{ databasePassword }}"
        - "externalDatabase.database={{ database.name }}"
        - "externalDatabase.host=mariadb.{{ namespace }}.svc.cluster.local"
```

Finally apply changes to the cluster:

```
$ narrenschiff sail --set-course moodle/course.yaml
```

Verify

You can verify that Moodle is deployed by accessing it through your web browser:

```
$ minikube service moodle --url
```

Use `minikube stop && minikube delete` to stop and delete the cluster.

CHAPTER

TWO

CONTRIBUTOR GUIDE

2.1 Contributing

2.1.1 Setup development environment

If you decide to contribute to the code base, you'll first have to fork the project on GitHub, and then clone the project in the local environment. You'll need a GitHub account in order to make a fork.

Make a clone of the repository, and install dependencies (just replace <*your_github_username*> with your actual GitHub username). This project is currently using `pipenv` to manage its dependencies. This could be changed in the future.

```
git clone https://github.com/<your_github_username>/narrenschiff.git
pipenv install --dev
```

This project is using `unittest` framework form the Python standard library. After you clone the repo, you can run tests to make sure everything is working:

```
make test
```

If you want to contribute to documentation, you can build it locally:

```
cd docs/
make html
```

Now that you have the development environment all setup, it's best to make a new branch for your feature (or bug fix!), and make changes there:

```
git checkout -b my-new-feature
```

When you finish coding, use `make test` to test the changes. If all is good, push the branch to your fork, and make a pull request. Don't forget to add unit tests for the new feature.

2.1.2 Keeping your fork synced with upstream

If you followed the previously described setup, then you can easily keep your fork up-to-date with the original repository. First, you need to add the upstream repository to your project:

```
git remote add upstream https://github.com/narrenorg/narrenschiff.git
```

Then, you can fetch changes made to the master, and push them to your fork:

```
git fetch upstream
git merge upstream/master
git push origin master
```

2.1.3 Coding Style

This project is mainly styled according to PEP8. Please use `flake8` to check the style of your code before making a pull request. This project comes with *.flake8* configuration file.

2.2 API

2.2.1 Narrenschiff

`narrenschiff.narrenschiff.`**`narrenschiff`**`()`

Base command.

Returns Void

Return type `None`

2.2.2 Task

`exception` `narrenschiff.task.`**`AmbiguousOptions`**

Use when tasks have undefined YAML tags.

`class` `narrenschiff.task.`**`Task`**`(`*task*`)`

Parse the smallest unit of the course.

Each dictionary item from the task list should be wrapped with this class:

```
course = [{'name': 'Kustomize', 'kustomization': 'examples/app/'}]
tasks = [Task(t) for t in course]

task = tasks[0]
task.name  # --> 'Kustomize'
taks.kustomization  # --> instance of Kustomization class
```

`class` `narrenschiff.task.`**`TasksEngine`**`(`*tasks*, *beacons*, *dry_run_enabled*`)`

Run course.

`run``()`

Start executing tasks.

Returns Void

Return type `None`

2.2.3 Templating

class `narrenschiff.templating.`**`ChestVars`**(*template_directory*)

Load chest files.

`load_vars`()

Load variables.

Returns Variables to be used for template rendering

Return type `dict`

class `narrenschiff.templating.`**`PlainVars`**(*template_directory*)

Load cleartext variable files.

class `narrenschiff.templating.`**`SecretmapVars`**(*template_directory*)

Load secretmap files.

class `narrenschiff.templating.`**`Template`**(**args*, ***kwargs*)

Load and manipulate templates and template environment.

`clear_templates`()

Delete all templates from the `/tmp` directory.

Returns Void

Return type `None`

`find_duplicates`(*values*)

Find duplicate keys.

Parameters **`values`** (`list` of `str`) – List of keys from var and chest files.

Returns List of duplicates

Return type `list` of `str`

`load_vars`()

Load variables.

Returns Variables to be used for template rendering

Return type `dict`

This is the order in which files containing variables are loaded:

1. Load `vars.yaml` if it exists
2. Load all files from the `vars/` directory if it exists
3. Load and decrypt all variables from `chest.yaml`
4. Load all files from the `chest/` directory if it exists
5. Load all variables from `secretmap.yaml`
6. Merge all files

Important: Files must not contain duplicate variable names!

`render`(*path*)

Render template on the given path.

Parameters **`path`** (`str`) – Path to the template file

Returns Rendered template

Return type `str`

`render_all_files``()`

Render all templates from the directory.

Returns Void

Return type `None`

Templates are copied to `/tmp` and location of rendered templates is saved in `self.tmp`.

`set_course``(`*path*`)`

Prepare Jinja2 templating environment.

Parameters **`path`** (`str`) – Path of the `tasks.yaml` file

Returns Void

Return type `None`

Directory containing the `course` (e.g. `tasks.yaml`) file will be the directory from where the templates are taken to be rendered.

`exception` `narrenschiff.templating.`**`TemplateException`**

Use for exceptions regarding template manipulation.

`class` `narrenschiff.templating.`**`Vars`**`(`*name*, *template_directory*`)`

Manipulate files containing variables.

`load_vars``()`

Load variables.

Returns Variables to be used for template rendering

Return type `dict`

`exception` `narrenschiff.templating.`**`VarsFileNotFoundError`**

Missing files containing variables.

2.2.4 Chest

`class` `narrenschiff.chest.`**`AES256Cipher`**`(`*keychain*`)`

Encode and/or decode strings.

`decrypt``(`*ciphertext*`)`

Decrypt ciphertext.

Parameters **`ciphertext`** (`str`) – Ciphertext

Returns Plaintext

Return type `str`

`encrypt``(`*plaintext*`)`

Encrypt plaintext.

Parameters **`plaintext`** (`str`) – Plaintext

Returns Ciphertext

Return type `str`

pbkdf2()

Derive a 32 bytes (256 bits) key.

Returns Password hash

Return type `byte string`

`class narrenschiff.chest.`**Chest**(*keychain*, *path*)

Manipulate chest file.

load_chest_file()

Load chest file with encrypted values.

Returns Serialized YAML object

Return type `dict`

show(*variable*)

Show decrypted value of the variable.

Parameters

- **variable** (`str`) – Variable name
- **value** (`str`) – Value of the variable

Returns Decrypted value

Return type `str`

update(*variable*, *value*)

Add or update chest file.

Parameters

- **variable** (`str`) – Variable name to update
- **value** (`str`) – Value of the variable

Returns Void

Return type `None`

2.2.5 Secret Maps

`exception narrenschiff.secretmap.`**CourseLocationError**

Raise exception if course is not found.

`class narrenschiff.secretmap.`**Secretmap**(**args*, ***kwargs*)

Manage secret maps. Secret maps are paths to encrypted files.

clear_all_files()

Delete all decrypted files.

Returns Void

Return type `None`

decrypt(*dest*, *treasure*)

Decrypts file and stores it to given destination.

Parameters

- **dest** (`str`) – Destination filepath of the decrypted file
- **treasure** (`str`) – Name of the variable

Returns Void

Return type `None`

`destroy`(*treasure*)

Delete secretmap file and remove key from the config file.

Parameters **treasure** (`str`) – Name of the secretmap variable

Returns Void

Return type `None`

`diff`(*secretmaps*)

Compare secretmaps line by line.

Parameters **secretmaps** (`tuple`) – Two secretmaps that should be compared

Returns Void

Return type `None`

`edit`(*treasure*)

Edit an encrypted file.

Parameters **treasure** (`str`) – Name of the variable

Returns Void

Return type `None`

`find`(*match*, *treasure*)

Match a pattern in a treasure and print to STDOUT.

Parameters

- **match** (`str`) – Pattern to match
- **treasure** (`str`) – Name of the secretmap variable

Returns Void

Return type `None`

`peek`(*treasure*)

Print encrypted file to STDOUT.

Parameters **treasure** (`str`) – Name of the secretmap variable

Returns Void

Return type `None`

`render_all_files`()

Decrypt and copy all files at the given destination.

`upsert`(*src*, *dest*, *treasure*)

Encrypts file and inserts data to config file.

Parameters

- **src** (`str`) – Source filepath for encryption
- **dest** (`str`) – Destination filepath of the encrypted file
- **treasure** (`str`) – Name of the variable

Returns Void

Return type `None`

2.2.6 Common

`exception narrenschiff.common.AmbiguousConfiguration`

Give warning that something is wrong with configuration.

`class narrenschiff.common.DeleteFile(path)`

Delete file from the file system, ideally in secure manner.

`delete()`

Delete the file.

`class narrenschiff.common.Singleton(name, bases, attrs, **kwargs)`

Define the Singleton.

`narrenschiff.common.flatten(lst)`

Flatten list.

Parameters `lst` (`list`) – A list to be flattened

Returns Flattened list

Return type `list`

`narrenschiff.common.get_chest_file_path(location)`

Check if there are duplicate chest files, and return paths if there are not.

Parameters `location` (`str`) – Relative path to course project directory

Returns Absolute path to chest file

Return type `str`

Raises *`narrenschiff.common.AmbiguousConfiguration`*

2.2.7 Config

`exception narrenschiff.config.ConfigurationException`

Use this when something goes wrong with the configuration.

`class narrenschiff.config.Keychain`

Bundle password and salt.

`class narrenschiff.config.KubectlContext`

Handle context switching.

`switch()`

Switch kubectl context.

`class narrenschiff.config.NarrenschiffConfiguration`

Load configuration file.

This class should never be called directly, nor it should be inherited from. Other classes should use it by composition principle (see implementation of e.g. *`narrenschiff.config.Keychain`*). This should make classes that depend on subset of config easier to test.

2.2.8 Filters

You can extend Jinja with custom filters. The filters should be in the *narrenschiff.filters* module.

`narrenschiff.filters.`**`b64enc`**`(`*value*`)`

Encode a string using base64.

Parameters **`value`** (`str`) – Text to be encoded

Returns Encoded text

Return type `str`

Use this filter for k8s secrets. Values for secrets need to be encoded before the `Secret` resource is deployed to k8s.

`narrenschiff.filters.`**`rtrim`**`(`*value*`)`

Strip trailing whitespace.

Parameters **`value`** (`str`) – Text to be processed

Returns Text without trailing whitespace

Return type `str`

`narrenschiff.filters.`**`secretmap`**`(`*value*`)`

Label path with `{{secretmap}}`.

Parameters **`value`** (`str`) – Path

Returns Labeled path

Return type `str`

2.2.9 Modules

Modules are way of implementing CLI commands e.g. `kubectl`. `narrenschiff` can be extended with new modules. All modules should reside in `narrenschiff.modules` package.

Common

`class` `narrenschiff.modules.common.`**`NarrenschiffModule`**`(`*command*`)`

Abstract class/Interface for the module classes. A module must inherit from this class.

Variables **`DRY_RUN_FLAG`** – `int`

The subprocess module does not have a way of indicating whether a command was run with dry run or not, since that is the responsibility of *narrenschiff.modules.common.NarrenschiffModule.execute()* method. If module or subcommand of module is not supporting dry run, than `DRY_RUN_FLAG` is a reserved return code (rc) that indicates that program should not exit, and output should be printed in special color (blue).

`abstract property cmd`

Get command that module needs to execute later.

Returns Full command with all parameters

Return type `str`

`property dry_run`

Return a dry run flag.

Returns `--dry-run` string

Return type `str`

In general most commands use `--dry-run` so there is no need to override this. However, there are exceptions for some commands where this flag is differently named. This property offers extensibility to the modules that may use different flag.

`abstract` **`dry_run_supported`**(*cmd*)

Check if command supports –dry-run.

Parameters **`cmd`** (`str`) – Command that module should execute

Returns Boolean indicating whether command supports dry run

Return type `bool`

`echo`(*output*, *rc*)

Print output to console, and exit if return code is different from 0.

Parameters

- **`output`** (`str`) – stdout or stderr of a process
- **`rc`** (`int`) – Return code of the process

Returns Void

Return type `None`

`execute`(*dry_run_enabled=False*)

Parse command and its arguments, and execute the module.

Parameters **`dry_run_enabled`** (`bool`) – Boolean indicating whether user turned on dry run for the task

Returns Void

Return type `None`

`subprocess`(*cmd*)

Execute command with shell, and return output and return code.

Parameters **`cmd`** (`str`) – Command to execute

Returns Output and return code

Return type `tuple`

Example:

```
output, rc = self.subprocess('kubectl get pods')
```

`exception` `narrenschiff.modules.common.`**`NarrenschiffModuleException`**

Use when something goes wrong in modules.

Kubectl

class `narrenschiff.modules.kubectl.`**`Kubectl`**(*command*)

`kubectl` module.

property cmd

Get command that module needs to execute later.

Returns Full command with all parameters

Return type `str`

`sanitize_filenames`()

Change relative paths to absolute paths corresponding to rendered files.

Returns Void

Return type `None`

`update_filename_argument`()

Update filename argument as a properly formated string.

Returns Void

Return type `None`

Kustomization

class `narrenschiff.modules.kustomization.`**`Kustomization`**(*command*)

`kustomization` module. Wrapper around `kubectl apply -k dir/`.

property cmd

Get command that module needs to execute later.

Returns Full command with all parameters

Return type `str`

Helm

class `narrenschiff.modules.helm.`**`Helm`**(*command*)

`helm` module.

property cmd

Get command that module needs to execute later.

Returns Full command with all parameters

Return type `str`

`dry_run_supported`(*cmd*)

Check if command supports –dry-run.

Parameters **cmd** (`str`) – Command that module should execute

Returns Boolean indicating whether command supports dry run

Return type `bool`

`parse_secretmaps_args`()

Mutate secretmap arguments. Expand secretmap paths to match files in the `/tmp` directory.

Returns Void

Return type `None`

exception `narrenschiff.modules.helm.`**`HelmException`**

Raise when something is wrong with Helm module.

gcloud

class `narrenschiff.modules.gcloud.`**`Gcloud`**(*command*)

`gcloud` module.

property cmd

Get command that module needs to execute later.

Returns Full command with all parameters

Return type `str`

`dry_run_supported`(*cmd*)

Check if command supports –dry-run.

Parameters **`cmd`** (`str`) – Command that module should execute

Returns Boolean indicating whether command supports dry run

Return type `bool`

Wait for Pod

class `narrenschiff.modules.wait_for_pod.`**`WaitForPod`**(*command*)

Use this module when you need to wait for a pod to become ready.

property cmd

Get command that module needs to execute later.

Returns Full command with all parameters

Return type `str`

`dry_run_supported`(*cmd*)

Check if command supports –dry-run.

Parameters **`cmd`** (`str`) – Command that module should execute

Returns Boolean indicating whether command supports dry run

Return type `bool`

`execute`()

Parse command and its arguments, and execute the module.

Parameters **`dry_run_enabled`** (`bool`) – Boolean indicating whether user turned on dry run for the task

Returns Void

Return type `None`

2.2.10 CLI

Deploy

`narrenschiff.cli.sail.`**`sail`**`(`*course*`)`

Turn tasks into actions.

Parameters **`course`** (`str`) – The path to `course` file. A file containing tasks specified in the YAML format. Tasks are `executable` pieces of configuration. Course is Jinja2 templated file (as are all the kubernetes manifest files)

Returns Void

Return type `None`

Chest

`narrenschiff.cli.chest.`**`chest`**`(`*ctx*`)`

Load keys and spices.

Returns Void

Return type `None`

`narrenschiff.cli.chest.`**`loot`**`(`*keychain*, *treasure*, *value*, *location*`)`

Display value from the chest file.

Parameters

- **`keychain`** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **`treasure`** (`str`) – Name of the variable
- **`location`** (`str`) – Path to the course directory

Returns Void

Return type `None`

`narrenschiff.cli.chest.`**`stash`**`(`*keychain*, *treasure*, *value*, *location*`)`

Dynamically update chest file. Override old value if exists.

Parameters

- **`keychain`** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **`treasure`** (`str`) – Name of the variable
- **`value`** (`str`) – Value of the variable
- **`location`** (`str`) – Path to the course directory

Returns Void

Return type `None`

`narrenschiff.cli.chest.`**`lock`**`(`*keychain*, *value*`)`

Encrypt string and print it to STDOUT.

Parameters

- **`keychain`** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **`value`** (`str`) – Value of the variable

Returns Void

Return type `None`

`narrenschiff.cli.chest.`**`unlock`**`(`*keychain*, *value*`)`

Decrypt string and print it to STDOUT.

Parameters

- **`keychain`** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **`value`** (`str`) – Value of the variable

Returns Void

Return type `None`

Secret Map

`narrenschiff.cli.secretmap.`**`secretmap`**`(`*ctx*`)`

Encrypt, decrypt, and edit files.

Returns Void

Return type `None`

`narrenschiff.cli.secretmap.`**`stash`**`(`*keychain*, *source*, *destination*, *treasure*, *location*`)`

Encrypt and stash file.

Parameters

- **`keychain`** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **`source`** (`str`) – Source filepath for encryption
- **`destination`** (`str`) – Destination filepath of the encrypted file
- **`treasure`** (`str`) – Name of the variable
- **`location`** (`str`) – Location of the secretmap file

Returns Void

Return type `None`

`narrenschiff.cli.secretmap.`**`loot`**`(`*keychain*, *destination*, *treasure*, *location*`)`

Decrypt file from stash.

Parameters

- **`keychain`** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **`destination`** (`str`) – Destination filepath of the decrypted file
- **`treasure`** (`str`) – Name of the variable
- **`location`** (`str`) – Location of the secretmap file

Returns Void

Return type `None`

`narrenschiff.cli.secretmap.`**`peek`**`(`*keychain*, *treasure*, *location*`)`

Print content of the encrypted file to STDOUT.

Parameters

- **`keychain`** (`narrenschiff.chest.Keychain`) – Object containing key and spice

- **treasure** (`str`) – Name of the variable
- **location** (`str`) – Location of the secretmap file

Returns Void

Return type `None`

`narrenschiff.cli.secretmap.`**`alter`**(*keychain*, *treasure*, *location*)

Decrypt and edit the file. After edits, the file is encrypted again.

Parameters

- **keychain** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **treasure** (`str`) – Name of the variable
- **location** (`str`) – Location of the secretmap file

Returns Void

Return type `None`

`narrenschiff.cli.secretmap.`**`destroy`**(*keychain*, *treasure*, *location*)

Delete secretmap file and corresponding key in the secretmap.

Parameters

- **keychain** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **treasure** (`str`) – Name of the variable
- **location** (`str`) – Location of the secretmap file

Returns Void

Return type `None`

You cannot undo this action!

`narrenschiff.cli.secretmap.`**`search`**(*keychain*, *location*, *match*)

Search for a pattern in secretmaps.

Parameters

- **keychain** (`narrenschiff.chest.Keychain`) – Object containing key and spice
- **location** (`str`) – Location of the secretmap file
- **match** (`str`) – Pattern you are looking for

Returns Void

Return type `None`

`narrenschiff.cli.secretmap.`**`diff`**(*keychain*, *location*, *match*)

Compare secretmaps line by line.

It can only compare secretmaps in the same course project.

Parameters

- **location** (`str`) – Location of the secretmap file
- **secretmaps** (`str`) – Two secretmaps that should be compared

Returns Void

Return type `None`

Lint

`narrenschiff.cli.lint.`**`lint`**`(`*ctx*, *location*`)`

Lint project files. Check if they are valid Jinja2 templates.

Parameters **`location`** (`str`) – Path to the directory

Returns Void

Return type `None`

Environment

`narrenschiff.cli.env.`**`env`**`(`*formatted*`)`

Show environment info.

Autocompletion

`narrenschiff.cli.autocomplete.`**`autocomplete`**`()`

Manage autocomplete script for your environment.

Returns Void

Return type `None`

`narrenschiff.cli.autocomplete.`**`add`**`(`*shell*`)`

Add autocomplete script to your environment.

Parameters **`shell`** (`str`) – Type of the shell you are using

Returns Void

Return type `None`

2.2.11 Logging

`class narrenschiff.log.`**`NarrenschiffLogger`**`(`**args*, ***kwargs*`)`

Set narrenschiff log level and print to STDOUT.

`set_verbosity``(`*verbosity*`)`

Set verbosity of the logger.

Parameters **`verbosity`** (`int`) – Verbosity level

Verbosity can be from 1 to 5, corresponding to five log levels.

2.2.12 Autocomplete

`class narrenschiff.autocomplete.`**`ShellAutocomplete`**

Set autocompletion.

`add``()`

Add narrenschiff autocomplete to shell config.

Returns Void

Return type `None`

add_autocompletion(*path*)

Add autocompletion to the config file.

Parameters **path** (str) – Path to configuration file

Returns Void

Return type None

autocompletion_enabled(*config*)

Check if the file contains the autocompletion tag.

Parameters **config** (list of str) – Configuration file

Returns Confirmation whether file contains autocompletion tag

Return type bool

autocompletion_script()

Return autocompletion script.

Returns Autocompletion script

Return type list of str

get_abs_path(**args*)

Join arguments, and return absolute path.

Parameters **args** (list of str) – Paths to be joined

Returns Absolute path

Return type str

If the first argument is tilde (~) then the method will expand user.

get_config_file()

Get type of the config file.

Returns Name of the config file

Return type str

This method returns either .bashrc (which will later be edited for the current user), or activate which symbolises activate script of your virtualenv.

read_file(*path*)

Read configuration file.

Parameters **path** (str) – Path to the file to read

Returns File as a list

Return type list of str

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX